Alma Mater Studiorum · Università di Bologna

Scuola di scienze
Corso di Laurea Magistrale in Informatica

Dynamic Human Robot Interaction Framework Using Deep Learning and Robot Operating System (ROS): a practical approach

Supervisor
Prof. Davide Rossi
Co-supervisor
Prof. Nadia S. Noori
Co-supervisor
Prof. Tim A. Majchrzak

Candidate
Marco Ferrati

sessione I
Anno Accademico 2021-2022

To my family and friends, who always supported me.

Abstract

Trying to explain to a robot what to do is a difficult undertaking, and only specific types of people have been able to do so far, such as programmers or operators who have learned how to use controllers to communicate with a robot. My internship's goal was to create and develop a framework that would make that easier. The system uses deep learning techniques to recognize a set of hand gestures, both static and dynamic. Then, based on the gesture, it sends a command to a robot. To be as generic as feasible, the communication is implemented using Robot Operating System (ROS). Furthermore, users can add new recognizable gestures and link them to new robot actions; a finite state automaton enforces the users' input verification and correct action sequence. Finally, the users can create and utilize a macro to describe a sequence of actions performable by a robot.

Acknowledgment

First of all, I'd like to thanks Prof. Rossi, Prof. Noori and Prof. Majchrzak for the help they gave to me while working on this internship and writing this thesis.

I'd like to thanks my parents for everything they give to me in these years.

I'd like to thanks my friends for the moments we pass all together.

Bologna, Luglio 2021 Marco Ferrati

Sommario

Questa tesi descrive il lavoro svolto durante il mio tirocinio presso l'Università di Agder, in Norvegia all'interno del programma Erasmus+ per Tirocini. Il lavoro è stato svolto sotto la supervisione della Prof.ssa Nadia S. Noori e del Prof. Tim A. Majchrzak.

Quanto fatto è un lavoro di ricerca nell'ambito dell'Human-Robot-Interaction.

L'idea nasce dalla necessità di trovare un modo per semplificare l'interazione tra un umano e un robot in diversi scenari utilizzando tecniche di computer vision e machine learning.

Esistono diverse soluzioni che implementano reti neurali per riconoscere le intenzioni di un utente, quello che differisce nel mio caso è l'utilizzo di ROS come sistema di comunicazione tra le componenti in modo da rendere la soluzione proposta il meno legata possibile all'hardware con la quale interagisce.

Durante il periodo di tirocinio, ho studiato la letterature riguardante la Human-Robot-Interaction e del riconoscimento dei gesti delle mani utilizzando tecniche di computer vision e machine learning. Ho inoltre studiato come funziona ROS in modo da poterlo integrare nella soluzione poi proposta.

Ho quindi realizzato un proof-of-concept di quello che è possibile fare per integrare componenti di intelligenza artificiale all'interno di applicazioni robotiche e pronto per poi essere esteso in sviluppi futuri.

Il prodotto realizzato è composto da due componenti principali la prima è un classificatore di gesti per le mani che opera in real-time utilizzando una webcam come fonte video. Per fare ciò ho utilizzato alcune delle librerie più famose nell'ambito del deep learning e della computer vision. Per quanto riguarda la seconda componente, questa si occupa di interagire con ROS permettendo di legare sequenze di gesti ad azioni compiute da un robot. In particolare l'integrazione raggiunta permette di inviare messaggi ad un altro nodo della rete e impostare una posizione da raggiungere.

La sequenza di gesti viene specificata dall'utente attraverso la descrizione di un automa a stati finiti (DFA) che si assicura di non poter accettare input non attesi.

Durante il lavoro sono emerse alcune caratteristiche interessanti della soluzione che si stava realizzando come la possibilità di insegnare nuove *gesture* con relativa semplicità da parte di un utente e la possibilità di salvare sequenze di *gesture* per poterle eseguire in un secondo momento più rapidamente e con maggior certezza.

Inoltre, per poter provare quanto realizzato il simulatore Gazebo è stato utilizzato per simulare diversi ambienti; in particolare l'ambiente nel quale mi sono maggiormente concentrato è quello di un magazzino. Ho quindi insegnato alla rete neurale a riconoscere l'alfabeto dei segni e altre *gesture* per poter eseguire alcuni compiti, per

esempio: raggiungi un punto della mappa, raccogli un pacco e posalo in un'altra posizione.
La bontà della soluzione proposta è sostenuta da una serie di test che mi hanno permesso di raccogliere varie metriche. I dati raccolti sono stati poi analizzati e confrontati con quelli presenti nella letteratura per problemi simili.
Infine, ho fatto un analisi critica del lavoro svolto individuando quali sono le sue limitazione e le sfide da superare per poterlo migliorare.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background and motivation

Improving the way operators can interact with a robot is a hard challenge in the field of Human-Robot Interaction (HRI). Nowadays, robots are everywhere, and the control of their movements and actions is usually done through a joystick or a dashboard in the case of very complicated tasks.
The idea of using the operators' body to interact with a robot is not a new one, but it brings several challenges when trying to implement it. A possible solution would be to use computer vision and Machine Learning (ML) techniques. In recent years, the computational power of processors and the ability to deploy ML models on cheaper and smaller devices have paved the way to interact with computers and robots in a way previously unfeasible.

1.2 Problem statement

The primary goal of this thesis is to design, implement, and test a framework for human-machine interaction capable of integrating machine vision and artificial intelligence capabilities into robotics operations. In addition, it has to be as dynamic as possible, in the sense that it must be able to adapt to several scenarios, facilitating the learning of new capabilities into robotic systems or autonomous worker units.

The proposed solution exploits computer vision and deep learning techniques to detect a set of hand gestures in real time and convey to a robot the action to take in response to the gesture. In this way, operators can use the expressive power of their hands and gestures' immediacy to communicate with the robot. Furthermore, by implementing this concept as an interface using the Robot Operating System (ROS) framework, we will no longer have to worry about which robot we want to manage in the future; all that is required is that it receives specified messages in order to function. An example of the idea we want to implement is represented in figure 1.1. Moreover, making the system as modular as possible would allow other people to use it as a proper framework for other projects that want to integrate machine vision and/or artificial intelligence capabilities into robotic operations. Finally, great relevance must have how to facilitate the learning capability of the

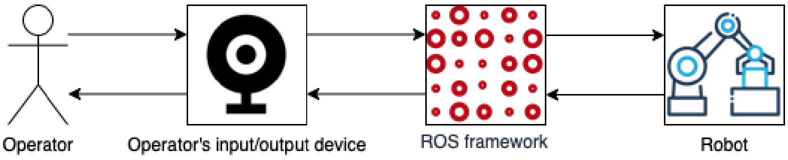


Figure 1.1: Example of system architecture.

system to make it dynamic and able to adapt to various scenarios.
The technology was chosen to be tested in a warehouse scenario because, since 2011, Amazon has been deploying robots within its warehouses, and the number of warehouses using robots to move goods is rapidly increasing [1]. In any case, a modular system would make it simple to adjust the robot's activities in response to a gesture or the motions the system recognizes.

As functional requirements, the system should:

- be able to recognize a set of hand gestures;
- use the ROS framework to communicate with a robot;
- pick up on new gestures from the users;
- create some macros, save them, and run them;
- enables the users to alter the robot's behavior in response to a gesture.

Meanwhile, during the implementation of the solution proposed, it is important to keep in mind that the framework should be usable by people with very low programming skills, so, as a non-functional requirement, all the possible configurations should be easy to read and write.

To fulfill all of these requirements, I combed the literature for information on the state of the art of the HRI, its evaluation methods, and the best way to perform a hand gesture recognition task. In addition, I studied ROS to integrate it in the proposed solution.

To summarize, this thesis is going to answer to the following questions:

- how can the Human-Robot Interaction be improved, especially taking into consideration the use of the human body to explain to a robot what to do?
- what is the state of the art regarding the recognition of hand gestures? Do they work for both static and dynamic? Can they recognize gestures in real-time?
- is it possible to integrate ROS with a hand gesture recognizer? Can the solution proposed be expanded in the future to work with new technologies?
- can users easily teach new gestures to the recognizer and integrate them into new scenarios?
- can the solution be capable of accepting other types of input, such as, for example, textual descriptions of command sequences?

1.3 Related works

In the field of Human-Machine-Interface, there are many attempts to simplify the communication between a human and a machine. Many of them use a third-party device (e.g. a keyboard, a mouse, a controller or a touch screen) to communicate with the machine. This involves a non-immediate understanding of how controllers works, and then operators must spend some time learning how to use them.

A more intuitive way to communicate with a robot is to use some kind of body gesture, mimicking the action that the robot must perform. There are two methods to recognize the gesture and many projects have been built on them [16]:

- **vision-based** methods: use cameras to capture the reality. The images are analyzed using computer vision and deep learning techniques;
- **sensor-based** methods: use a third party device like a Wiimote [6] or a wearable, for example a glove in case of the PowerGlove [8].

There are several projects trying to integrate gesture recognition with the control of robots. A particularly interesting one is the one proposed by Chen et al. in the 2019. They developed a system composed of three components: an online personal feature pre-training system; a gesture recognition system; and a task re-planning system for robot control [3]. Also, searching on GitHub with the keywords "*robot control gesture*" returns hundreds of projects, especially related to controlling robotic arms. More generally, the task of gesture recognition is a well studied task.

It is worth noting that almost all projects are designed to work with a specific robot. ROS is not usually involved, leading to a more difficult integration with future hardware. Moreover, the dynamism of the solution is also not usually taken into account, resulting in a solution that can only be used in that context. In particular, learning a new gesture is usually a difficult task to accomplish.

1.4 Organization

This document is organized as follow:

Chapter two describes the background I obtained studying the literature about HRI, hand gestures recognition and, ROS.

Chapter three describes the technologies and tools used to implement the solution proposed and why they have been chosen to fulfill the requirements. Moreover, the design process that led to the solution proposed is described. In particular, it is focused on the implementation of the system composed by the hand gesture recognizer and a robot employed in a warehouse (i.e. a storage and retrieval robot).

Chapter four presents the results obtained performing several tests with the system developed.

Chapter five presents a discussion on the results presented in chapter four, explaining them and confronting with other solutions.

Chapter six presents my conclusions on the work done for this internship.

Chapter 2

State of the Art

2.1 Human-Robot Interaction

At present, there are several ways to interact with a robot. The best way to do it, generally, depends on the type of robot and the task it has to perform. It is possible to compare two ways for a human to interact with a robot:

- controlling the robot with a third-party device that acts as a controller, such as a joystick or a computer with a user interface;
- using operators' body, specifically hand gestures, voice, or body position.

2.1.1 Using a third party device

The earliest ways of interacting with a robot involve the use of a controller that operators must use to tell the robot what actions to perform. In this case, the controller could be a joystick or a computer program. In the last case, the study of Human-Computer Interaction (HCI) must be considered. Using a third-party device to control a robot is not the easiest way, especially when the tasks to be completed are intricate and the robot's movements are as complicated as the tasks.

2.1.2 Using the human body

The other way to interact with a robot involves the use of the human body. The robot can get input signals through microphones and cameras. Operators can give input through their voice. In this case, Natural Language Processing (NLP) is involved. Operators can also give input with hand gestures, body position, or even facial expressions. For example, in the 2018, Kahuttanaseth et al. achieved an accuracy between the 70% and 80% in converting raw input text into robot commands through NLP techniques [7]. Better results are achieved when the full body position is exploited. Lee, already in the 2006, used body position to recognize gestures such as walking, running, bending, jumping, lying down on the floor, waving a hand, sitting on the floor, raising the right hand, getting down on the floor, and touching a knee and wrist with 95% accuracy [9]. Fujii et al. also reached similar results in 2014 recognizing four gestures [5]. Regarding hand gestures, the literature has some interesting experiments that exploit them to control a computer or a robot. Shanthakumar et al., in the 2020, designed and evaluated six hand gestures to interact with a computer. The results are outstanding with an accuracy of 97% [16]. Finally, in the 2022, the work made by Canuto et al. defined a way to identify which gestures can be used to interact with a robot considering the "Intuitiveness Level" [2]. The proposed methodology is composed of four steps:

1. choice of tasks: select a set of tasks that the robot will perform.
2. capture of gestural data: a user-based approach is suggested. An intuitive gesture comes from the subconscious mind. You can ask some volunteers to perform some gestures related to the task until they are out of ideas. This approach is called, by the author, the "Frustration Based Approach".
3. analysis of captured gestures: each gesture is analyzed to find the most common and most compatible with the task.
4. choice of vocabulary (intuitiveness table): the set of gestures is chosen. The decision is made by looking at the Intuitiveness Level (IL).

2.2 Hand gesture recognition

Hand gesture recognition task is a well-studied task. In the literature, it is possible to find the idea of using hand gestures as a way to interact with a machine since 1987 [21]. At that time, the idea was to use a glove to recognize the position and orientation of users' hand. They were thought to be used for different tasks, like gesture recognition, an interface for a visual programming language, virtual object manipulation, and many others. Nowadays, even if a wearable device to recognize what the hands are doing is highly accurate and precise, for some tasks it is possible to reach a good level of accuracy with a more widely accessible webcam. In particular, thanks to the increase in computing power in small devices and the improved quality of video acquisition devices, the study of computer vision and machine learning techniques to recognize hand gestures is becoming very interesting.

2.2.1 Machine learning

Recognizing a hand gesture given an image or a frame of a video is not something easily algorithmizable. For this, the idea of using a neural network to fulfill the task is a good one.

Neural network

A neural network is a collection of connected neurons. In biology, a neuron is an entity that takes several inputs, sums them together, and, if a threshold is surpassed, emits an output. An artificial neuron is similar. Figure 2.1 represents one of them. It takes N inputs, sums them together with a bias, and passes the result to an activation function. If the result is higher than a threshold, the neuron returns an output.

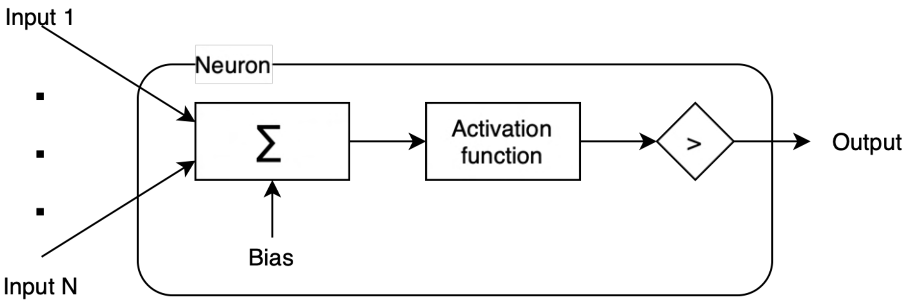


Figure 2.1: Representation of a neuron in a neural network.

An artificial neural network is usually organized into layers, as opposed to biological ones, which are much more complex. Figure 2.2 represents a deep neural network, deep because there are one or more hidden layers. Different kinds of layers exist. The one shown in figure 2.2, other than input and output, is dense layer. In this kind of layer, every output of the previous layer is received as input by each neuron of the layer considered for the reasoning. Regarding the output layer, each neuron returns a probability that the input belongs to a class. When there is only one output neuron the probability is p to belong to the class and $1-p$ to not belong to the class. When there is more than one neuron, each of them returns the probability of belonging to a class, and the highest one is taken as the prediction. A prediction, to be considered correct, must overcome a threshold value. In general, more are the hidden layer, more the network will be accurate but the learning process will take more time.

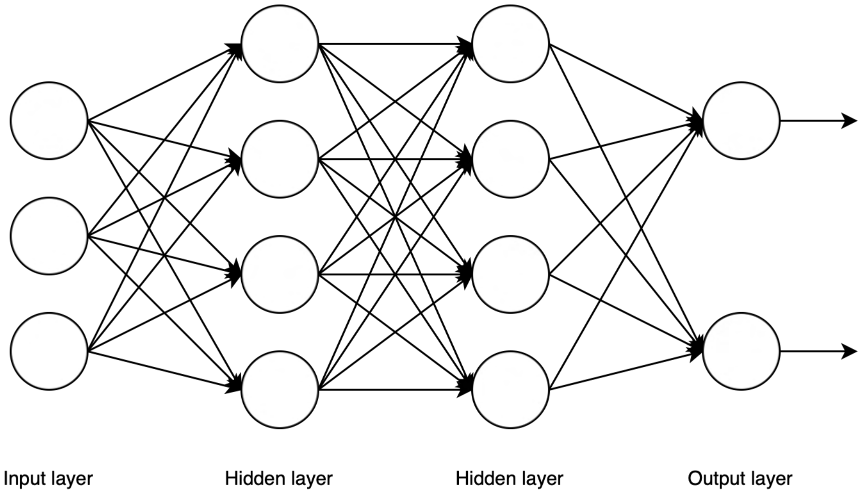


Figure 2.2: An example of a deep neural network

Dataset

When ML is involved, one of the challenges is the need for large datasets on which the network can train. When image recognition is the task to be fulfilled, the datasets are composed of a lot of images, and each one must be labeled to know what it represents and where, inside the image, the position of the object to be detected is. This kind of training is known as supervised learning, which is different from unsupervised learning, in which the dataset has no labels and usually the task is to categorize the elements into macro-categories. Regarding the dataset dimension, it is worth stressing that the bigger the dataset, the slower the training task. It is important to find the right dimension that allows you to have an acceptable accuracy level.

Training

The goal of training a neural network is to find the best weights for each neuron's inputs. The technique for a multi-layer neural network is stochastic gradient descent with back propagation, and data is essential to training a neural network. Three datasets are required to correctly train and evaluate a neural network. Usually, one dataset is randomly divided into three non-overlapping partitions. The first and biggest part of the dataset is used to train the network. In more detail, it is used to adjust the weights of inputs. Then, another part of the dataset is picked as validation during the training process, and the last part is taken to evaluate the trained model at the end.

Evaluation

To evaluate an ML model, it is necessary to collect some data during the training process. The metrics to keep track of are:

- **loss function**: the network's goal is to minimize the loss function. Generally, it represents the prediction error with respect to the ground truth;
- **accuracy** is defined as the ratio of correct predictions to total predictions made.

$$Accuracy = \frac{Number\, of\, right\, predictions}{Number\, of\, predictions} \tag{2.1}$$

- **time**: the amount of time spent training the network. It depends on the dataset size and the complexity of the network. Specifically, a bigger dataset will require more time but will give better results, as well as a more complex network.

The best neural network is the one that guarantees the best trade-off between these metrics.

State of the art

With the help of neural networks, there are several ways to achieve good results in the hand gesture recognition task. The results presented in Wang et al. [18] shows that with a Convolutional Neural Network (CNN), it is possible to achieve an accuracy higher than 90%. CNNs are classifier-based systems that can process input images as structured arrays of data and identify patterns between them. To date, there are two main types of object detection algorithms in the field of deep learning:

- **classification-based** algorithms: firstly, they select a group of Region Of Interests (ROIs) in the images where the chances that an object is present are high; secondly, they apply CNN techniques to these selected regions to detect the presence of an object. A problem associated with these types of algorithms is that they need to execute a detector in each ROI, and this makes the process of object detection very slow and highly expensive in terms of computation.

- **regression-based** algorithms: these types of algorithms are faster than the above algorithms, in that there is no selection of the ROI so that the bounding boxes and the labels are predicted for the whole image at once; they can identify and classify objects within the image at once. Beyond the higher speed, a key point is that the predictions are informed by the global context in the image, thus they generally lead to higher accuracies.

One of the most famous regression-based algorithms is You Only Look Once (YOLO), but it is not the only possible solution to perform this kind of task. Another technique that gives promising results is the combination of the MediaPipe hand tracker, which section 3.2.3 describes, and a feed-forward neural network to recognize the gesture.

All these kinds of solutions suit well in the case of static hand gestures. As Takahashi shows in his repository[17], working with a history of landmarks is possible. For this kind of task, a Long Short-Term Memory (LSTM) neural network is a good starting point. This kind of network tries to add the knowledge of past events to the computation. To do so, there are loops inside them that allow information to persist [12].

2.3 Robot Operating System (ROS)

ROS was born as a framework to design robot software. Its initial presentation was in 2009 by Quigley et al. [13]. In particular it tries to solve the problem of communication but, it has also many other characteristics that make it a good framework for robot development.

2.3.1 Communication

A system based on ROS has a *peer-to-peer* topology. Each node of the network can communicate with any other node. The first version of ROS implemented a custom communication layer but, from the second version onwards, it implements the Data Distribution Service (DDS) protocol.

Data Distribution Service (DDS)

ROS uses DDS as an end-to-end middleware to exchange messages: The adoption of the DDS is one of the main differences between ROS version 1 and ROS version 2. The DDS was chosen by the ROS maintainer for its reliability and flexibility in mission-critical systems, such as:

- battleships;
- extensive utility installations;
- monetary systems;
- spacecraft;

- flight control systems;
- train switchboard systems.

The implementation of DDS is hidden from the users, who uses the methods described in section 2.3.6 with the ROS API to exchange messages. A complete explanation for this change has been given by the developers and can be found on the article written by Woodall [19].

2.3.2 Multi-lingual

ROS supports many programming languages. In particular C++ and Python are the most documented. Each supported language has its own implementation of the communication layer. In this way the developer can follow the best practice for each one but, it is possible to wrap the already written library into another language. "The end result is a language-neutral message processing scheme where different languages can be mixed and matched as desired." [13]

2.3.3 Tools-based

The developers of ROS have decided to adopt a *microkernel* design. In this way they produced different tools and, each of them perform a task:

- navigate the source code tree;
- get and set configuration parameters;
- visualize the network topology;
- measure the band-with utilization;
- graphically plot message data;
- generate documentation;
- launch sequence of tasks

and many others.

2.3.4 Thin

The developers of ROS encourage the creation of small executables by leaving the complexity in the dependencies. This promotes writing code reusable also in other projects.

2.3.5 Free and Open-Source

ROS is free and Open-Source, this has created a strong community around the framework. The community helps finding bugs and developing new algorithms accessible to everyone.

2.3.6 Nomenclature

The key points to understand when using ROS are related to the "ROS 2 Graph", and they are:

- **node**: a system component in charge of a specific task. It is an executable; for example a Python executable or also a C++ executable, and it can communicate with other nodes, exchanging messages. A robotic system is composed of multiple nodes;

- **message**: the method by which nodes exchange data. Each message has its own "type", and this brings advantages in building an interchangeable system because it is possible to change the components of the system with others that can understand the same messages. Messages, in this case, can be seen as interfaces for programming languages. There are three ways for nodes to exchange messages:
  - **topic**;
  - **service**;
  - **action**.

Exchange messages

Topic It is an implementation of the publisher/subscriber pattern. A node can publish messages with a topic, and every other node that is listening to that topic will read that message. This is an asynchronous way of exchanging messages because the sender will not know if the message has been read. For example, this is the best way to broadcast a message without saturating the network.

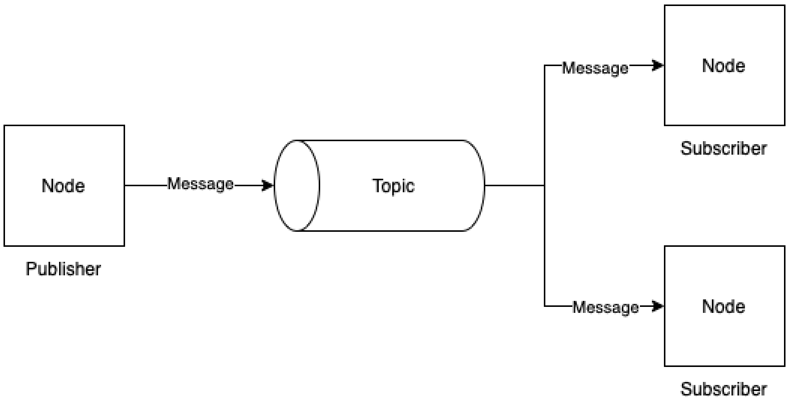


Figure 2.3: Example of message exchange between three nodes using a topic.

Service It is based on a call-response model. In this case, a node requests some data from another node through a request message. The latter replies with a response message. This is a synchronous way of exchanging messages. The node that needs the data waits for the response. There can be many nodes that use the same service to request some data from another node. This is a bit like what happens in a client-server architecture, but it should not be used for long-running processes.

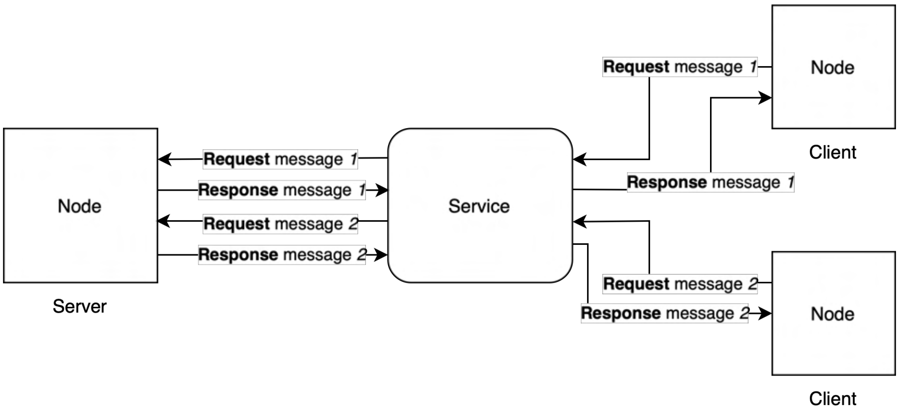


Figure 2.4: Example of messages exchange between three nodes using a service.

Action It uses both topics and services. The functionality is similar to service with the addition of a constant stream of updates from the "server" through a topic to which the "client" subscribes. The sequence of actions is the following:

1. a node (i.e. the client) sends a message (the request for a goal) through a service to another node (i.e. the server). The latter replies with one message through the service. For example, it can reply with an acknowledgment or a message saying it has started working on the task;
2. the server keeps the client updated with the progress of the task through a topic;
3. the client sends a request through another service to the server. When the task is finished, the server will reply to the client.

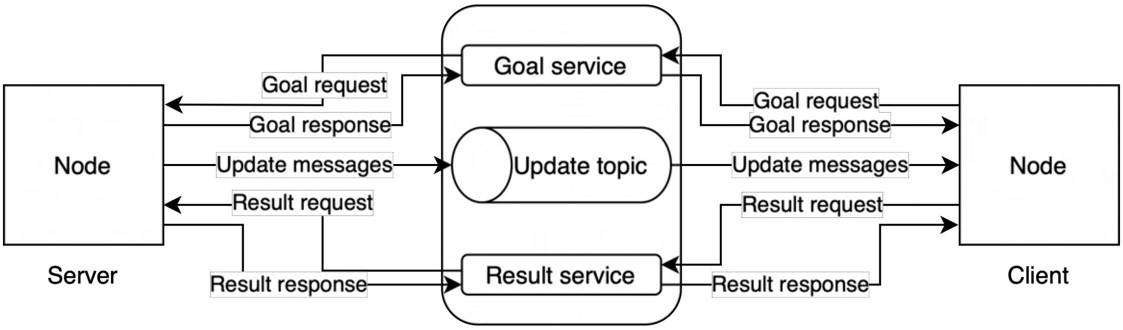


Figure 2.5: Example of messages exchange between two nodes using an action.

Chapter 3

Method

To fulfill the requirements of the project, I adopted an experimental approach. First and foremost, I analyzed the problem in order to understand which kind of sub-problems I was supposed to solve. From that, I was able to determine which tools would have been the most appropriate for doing the computer vision and machine learning tasks and the integration with ROS. Then, to experiment with the solution proposed, I designed and developed a proof of concept capable of recognizing and learning hand gestures made by users and communicating to a robot which action to perform based on the gesture recognized.

3.1 Preliminary study

In analyzing the problem, the following sub-problems emerged to be solved:

- a way to capture webcam's frames;
- a way to recognize hand gestures, both static and dynamic from the frames received from a webcam;
- a way to teach to the system new gestures and connect them to new actions;
- a way to convert the recognized gestures into commands for a robot using ROS as message exchange system. In particular a way to exchange general purpose messages and set navigation goals;
- a way to simulate the proposed solution in different environments.

3.2 Choice of technologies and tools

To achieve goals described above, I decided to use specific technologies and tools available for everyone. The taken decisions have been made based on my knowledge of the technologies and on the necessity to use some specific tools to fulfill the requirements.

3.2.1 Python

Python is a high-level programming language. It is a well-known and highly supported programming language for machine learning tasks. Moreover, it is also supported by ROS whose documentation has sections written for it. The version used for this project is 3.8 because it is the version distributed within the latest release of Ubuntu operating system. However, being able to use the latest version would allow the exploitation of new statements, for example the pattern matching offered through the `match ... case` statement, that would make the implementation easier.

3.2.2 Tensorflow

Tensorflow is an open-source Python library to build ML models. Google began developing it in 2015, and it is now one of the most widely-used Python libraries for performing ML tasks. It offers a huge number of layers, activation functions, and tools to build simple and complex neural network architectures. The best-known counterpart is PyTorch. During my studies, I have had to use both, and I think Tensorflow is more suitable to develop neural networks oriented toward an application. Instead, PyTorch is more appropriate for the development of new and complex neural networks. Moreover, Tensorflow is better integrated with data collection tools like Tensorboard.

Tensorflow lite

Tensorflow Lite is a component of Tensorflow that allows you to convert a Tensorflow model into a compressed flat buffer and then deploy it onto any device (e.g. mobile devices or embedded devices). With the aim to deploy the hand gesture recognizer to an embedded device like the Nvidia Jetson or distribute it to other people, the use of this tool is natural.

Keras

Keras takes advantage of Tensorflow to give the users a powerful API to design and develop deep neural networks. With a few lines of code, it is possible to implement complex neural networks that exploit the latest research in the field of ML.

3.2.3 MediaPipe

MediaPipe is an open-source, real-time, and on-device tool that can track multiple parts of the body. In particular, I am interested in hand tracking. MediaPipe suits very well for this purpose because it offers a pre-trained ML model to recognize and track twenty-one landmarks on each hand (figure 3.1). In particular, it uses a pipeline composed of two ML models:

1. a palm detector that works on a full image locates the palm and identifies the bounding box around it;
2. a hand landmark model that works on the cropped image of the palm and returns the hand landmarks considering the depth also. That means it can track a landmark behind another one.

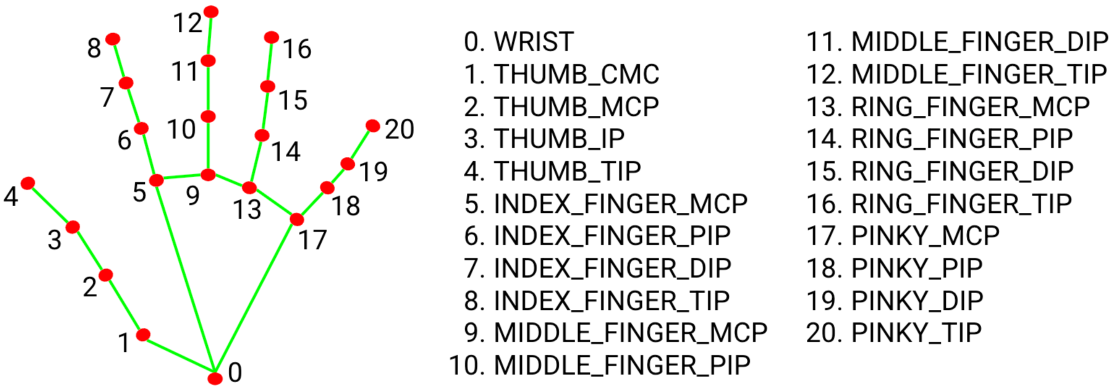


Figure 3.1: Landmarks on a hand recognized by MediaPipe [11].

The precision of this tool is about 96% [20], so it is a good starting point for the hand gesture recognition task. It is possible to get the position of the landmarks and give them input through a deep neural network trained on the gestures of our interest.

The list of coordinates relatives at position of the hand's landmarks is returned by MediaPipe. These coordinates can be saved and used to train a deep neural network instead of images. This is interesting because:

- it reduces the size of the dataset;
- it eliminates environmental factors such as background, lighting, and skin color.

It is interesting to point out that MediaPipe is capable of tracking, in real-time, different parts of the human body, for example, the face and the whole body [11].

3.2.4 OpenCV

OpenCV is an open-source library that fully meets the requirements regarding computer vision and works well with MediaPipe. It is also distributed as a Python package to integrate into users' applications.

3.2.5 Robot Operating System

The use of ROS is a requirement for the project. There are several releases of it. *ROS Galactic Geochelone* is the chosen one because it was the latest at the moment the development started. It has been released in January 2022.

Navigation

"Nav2" is one navigation system compatible ROS. A developer can choose to use their navigation system, but the one developed by Macenski et al. is widely tested and supported [10]. It uses a set of actions whose behaviour is described in section 2.3.6 and provides a complete set of API to control the robot. Specifically, those used are:

- `setInitialPose`: to set the initial position of the robot;
- `goToPose`: to tell to the robot to reach a position;

- `isTaskComplete`: to know when the task is finished;
- `getFeedback`: to receive a feedback (i.e. the current position) from the robot.

A complete list of what this package is capable can be found on the Nav2 documentation.[1]

3.2.6 Gazebo simulator

Gazebo is an open-source simulator for simulating environments involving robots. Gazebo offers the ability to accurately and efficiently simulate populations of robots in complex indoor and outdoor environments. Moreover, there is the possibility to use different robot models using Simulation Description Format (SDF) files and import Collada files into the simulated world. Gazebo is also expandable with plugins. One of those lets you use ROS to communicate with the robots inside the simulation. I used the Gazebo simulator to perform the communication tests between the hand gesture recognizer and the robot.

3.2.7 Git

Git is a distributed version control system. It has been extensively used to store and share the source code and documentation for this project. In particular, GitHub has been used, creating several repositories.

3.3 System design and implementation

The hand gesture recognizer was based on Takahashi's [17] project, to which several improvements (i.e., code refactoring to improve readability, code reuse, and reduction of opportunistic copy and paste) and new functionalities were added.

3.3.1 System capabilities and data flows

Before running the hand gesture detector, users can choose which mode to run the program:

- **operational**;
- **learning**;
- **macro**.

Operational mode

Within this mode, operators can do a sequence of hand gestures that will be translated into commands for the robot and will be sent to it through ROS' communication system.

[1] https://github.com/ros-planning/navigation2/tree/main/nav2_simple_commander

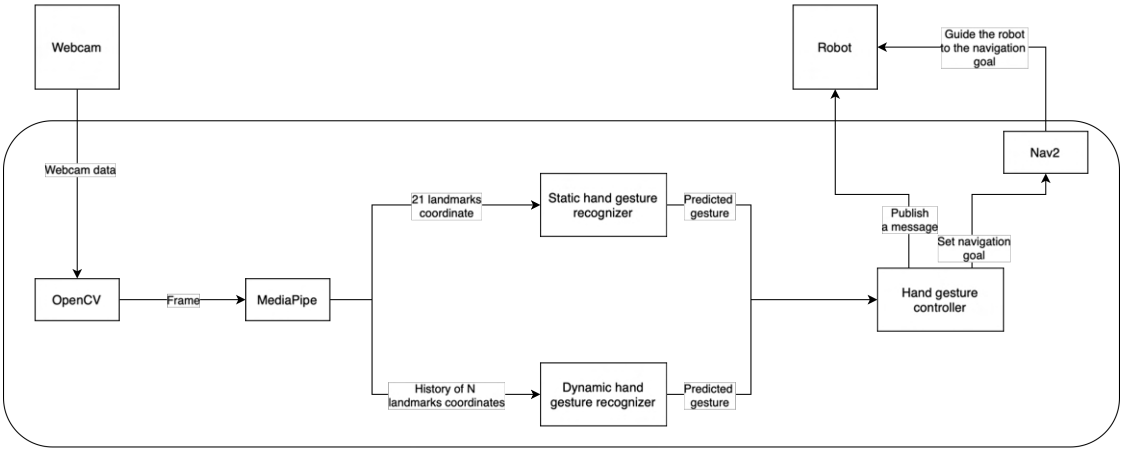


Figure 3.2: Data flow in operational mode.

Figure 3.2 shows the data flow in the operational mode:

1. OpenCV receives the data from the webcam and converts it into a frame;
2. the frame is given as input to MediaPipe, which handles the hand tracking task and returns the list of landmarks if a hand is detected in the frame;
3. the landmarks in the frame are given as input to the "static hand gesture recognizer" model. Meanwhile, the landmarks in the frames along with those from the previous N frames are given as input to the "dynamic hand gesture recognizer";
4. both the recognizers return the predicted gesture;
5. both the predicted gestures are taken into input by the "hand gesture controller" that decides which action to execute. Section 3.3.3 describes in more detail how it works. Generally, it can publish a message on a ROS' topic or set a navigation goal through the "Nav2" package.

Learning mode

In this mode, users can choose to add a new gesture or enhance an already existing one by adding more data to the dataset. In both cases, a command-line interface is used to interact with the user and guide it through the process.

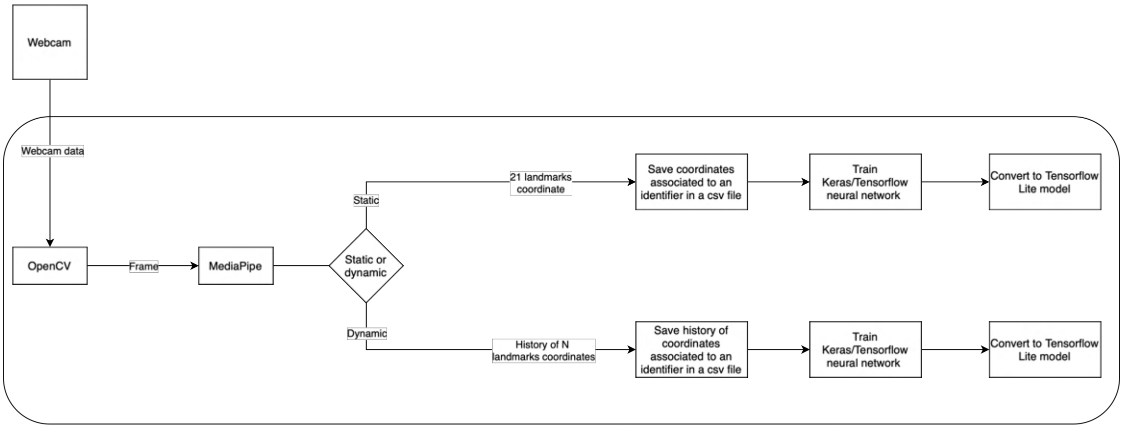

Webcam
Webcam data
Static
21 landmarks coordinate
Save coordinates associated to an identifier in a csv file
Train Keras/Tensorflow neural network
Convert to Tensorflow Lite model
OpenCV
Frame
MediaPipe
Static or dynamic
Dynamic
History of N landmarks coordinates
Save history of coordinates associated to an identifier in a csv file
Train Keras/Tensorflow neural network
Convert to Tensorflow Lite model

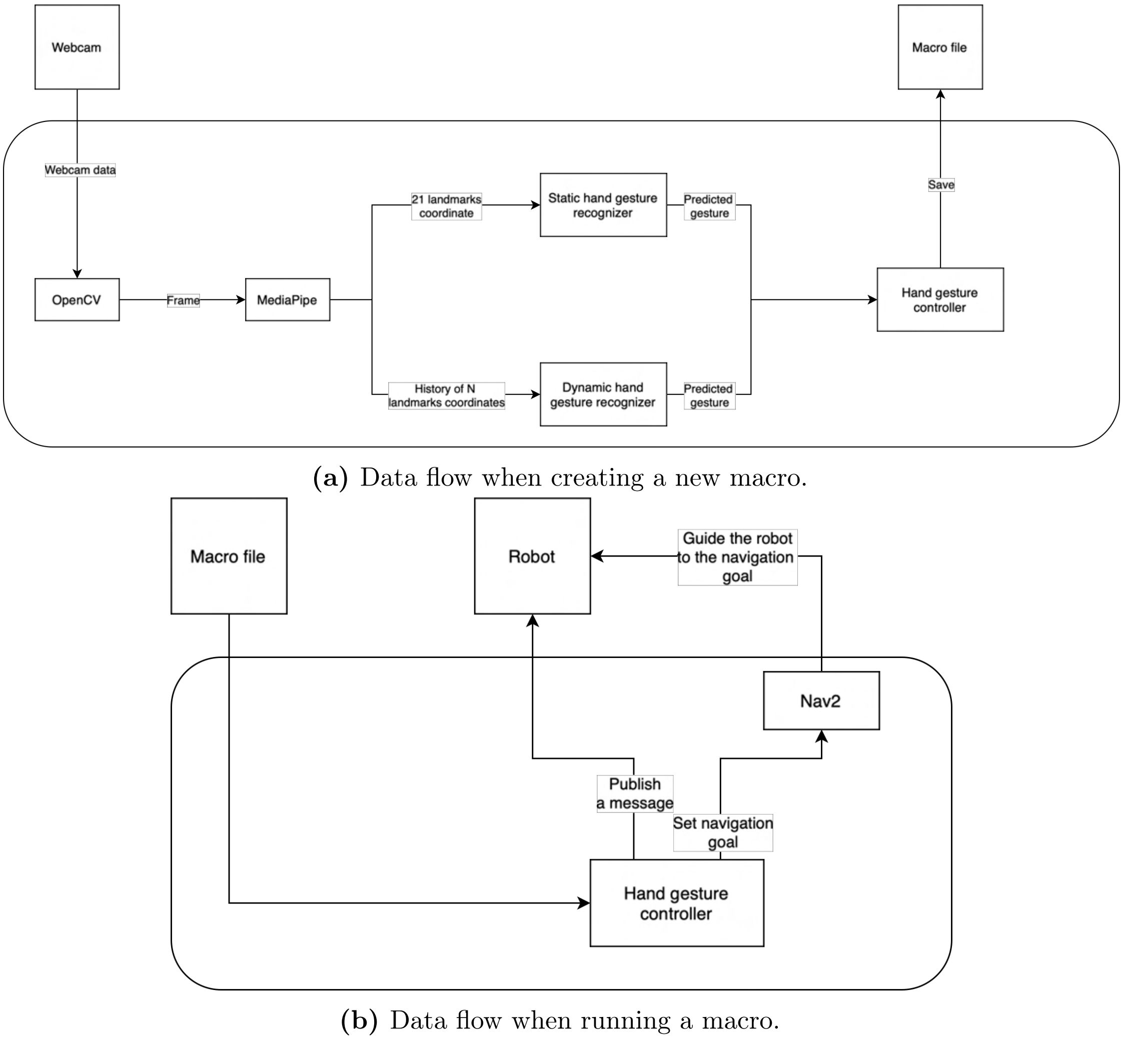
Figure 3.3: Data flow in learning mode.

Figure 3.3 shows the data flow in the learning mode:

1. OpenCV receives the data from the webcam and converts it into a frame;
2. the frame is given as input to MediaPipe, which handles the hand tracking task and returns the list of landmarks if a hand is detected in the frame;
3. if users choose to add a new static gesture or enhance an existing one, the landmarks in the frame are saved in the CSV file with the identifier number that refers to the label of the gesture. Otherwise, if users choose to add a new dynamic gesture or enhance an existing one, the landmarks in the frames along with those from the previous N frames are saved in the CSV file with the identifier number that refers to the label of the gesture.
4. the associated neural network has been trained;
5. the model is converted into a TFLite model, ready to be used by the recognizer.

Macro mode

This mode is similar to the operational but, instead of sending the command in real-time to a robot, users can choose to save a sequence of gestures in a text file or "execute" a previously saved macro file, sending its content to a robot.



(a) Data flow when creating a new macro.

(b) Data flow when running a macro.

Figure 3.4: Data flows in macro mode.

Figure 3.4a shows the data flow when users creates a new macro. The first part is the same as the operational mode explained in section 3.3.1 but, instead of sending the command to the robot, it is saved in a text file. Figure 3.4b shows when a macro is run. The sequence of commands is read from the file created as described above. Then, the commands are given as input to the "hand gesture controller" which communicates with the robot as explained in section 3.3.1.

3.3.2 Hand gesture recognizer

The first challenge to solve was the design and implementation of the hand gesture recognizer. In particular, two types of hand gestures were considered:

- **static hand gestures**: in which the hand does not move and only a snapshot of the finger position is needed to recognize the gesture;
- **dynamic hand gesture**: in which the hand moves. In this case, a sequence of data is necessary to recognize the gesture.

This diversity leads to two different neural networks to classify the hand gestures users are making.

Static hand gestures

As reference, the set of static hand gestures chosen is the American Sign Language (ASL) (in figure 3.5). J and Z are excluded because they involve a movement. To discriminate between dynamic and static gestures, I decided to prioritize the dynamic ones because users have to perform an action to activate them. Instead, with the static gestures, the hand does not move, and the recognizer will always return a possible prediction. Also, the numbers have been excluded.

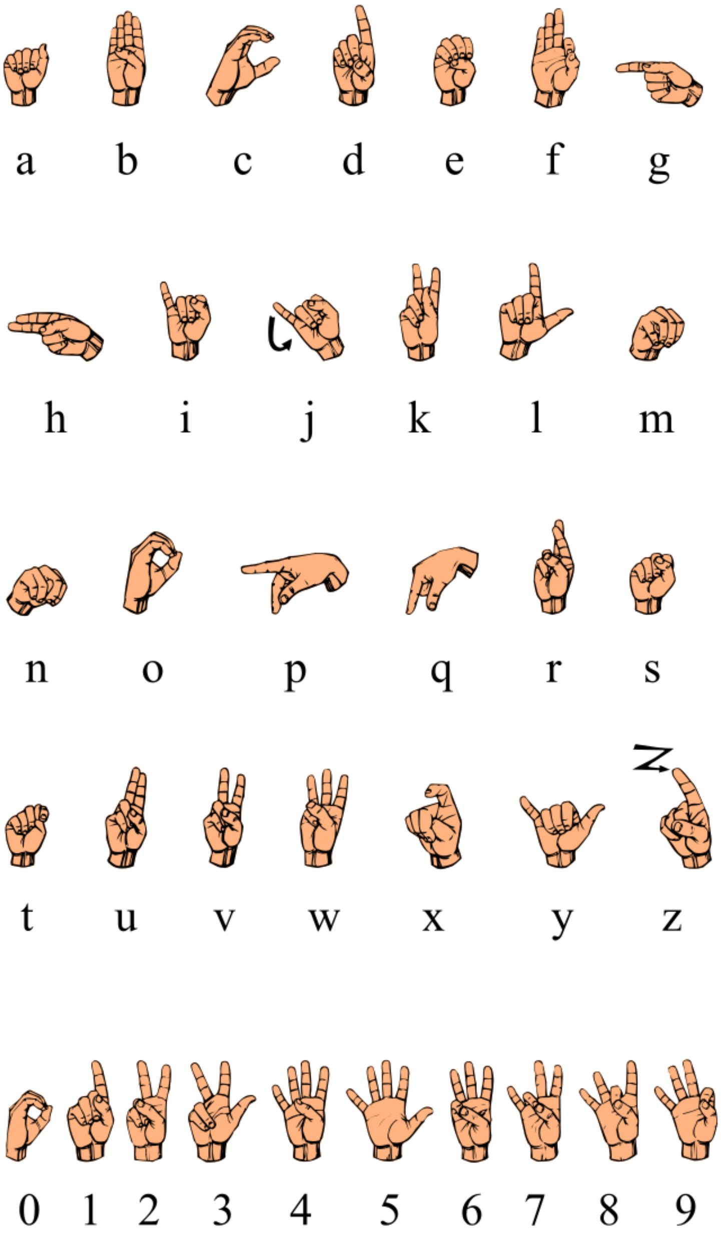


Figure 3.5: American Sign Language [4].

MediaPipe data From MediaPipe, the static hand gesture recognizer gets twenty-one landmarks per frame. These landmarks are shown in figure 3.1. The dataset is a CSV file where each line contains:

- the identifier of the static hand gesture. It refers to the line in the label's file;
- 42 coordinates (i.e. X and Y) of the 21 landmarks, they are relative to the zeroth landmark (i.e. the wrist) and normalized.

Deep neural network The neural network to recognize the hand gesture starting from MediaPipe's landmarks is a feed-forward network.

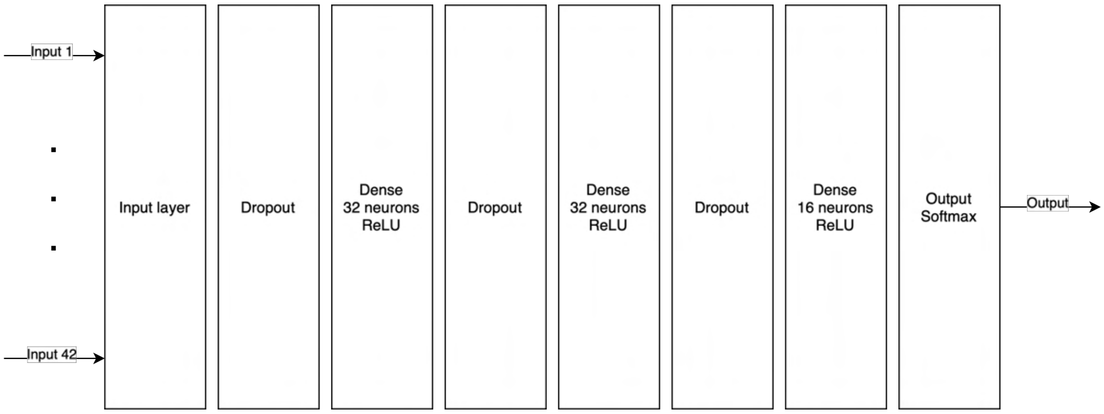


Figure 3.6: Deep neural network for static hand gestures.

Figure 3.6 presents the layers in the network:

- **input layer**: it takes in input the data from the dataset. Exactly 42 float numbers, representing the relative and already normalized coordinates of the landmarks;
- **dropout layer**: it randomly drops the input received, which helps prevent overfitting. If the input is kept, it is passed to the next layer unchanged;
- **dense layer**: it is a layer composed of N neurons. Every one of them takes every input received from the previous layer and uses the ReLU activation function to return the output;
- **output layer**: it is a dense layer with the softmax as activation function.

Training To train the network the script:

1. reads the dataset file;
2. if users request it, a random under-sampling is applied;
3. divides the dataset into three non-overlapping subsets (train set, validation set, and evaluation set);
4. builds and compiles the network with callbacks to save the model during training, to early stop the training if the loss function does not improve anymore, and log analytics for Tensorboard;
5. trains the network on the train set and validates it with the validation set, gathering accuracy, loss function value, and time spent;
6. evaluates the network on the evaluation set;
7. converts the model to a Tensorflow Lite model.

Dynamic hand gestures

Six dynamic hand gestures have been chosen to test the system:

- **Z**: taken from the ASL;
- **J**: taken from the ASL;
- **go to**: to communicate to the robot to move to a location;
- **pick up**: to communicate to the robot to pick up a parcel;
- **drop down**: to communicate with the robot in order for the parcel to be dropped down;
- **static**: for when the hand is not moving.

The recognition of the dynamic hand gestures, except for *static*, is prioritized over static ones.

MediaPipe data From MediaPipe, the dynamic hand gesture recognizer gets $N \times history\_length$ landmarks coordinates per each frame. Where N is the number of landmarks, shown in figure 3.1, saved, and $history\_length$ is the previous frames from which to take the landmarks. Indeed, users can choose how many landmarks to save and how many previous frames to consider. This data is saved in a CSV file where each line contains:

- the identifier of the dynamic hand gesture. It refers to the line in the label's file;
- $N \times history\_length$ coordinates relatives to the zeroth landmark (i.e. the wrist) and normalized.

Deep neural network I tried two neural networks to recognize the dynamic hand gestures starting from MediaPipe's landmarks history. The first one is a feed-forward network.

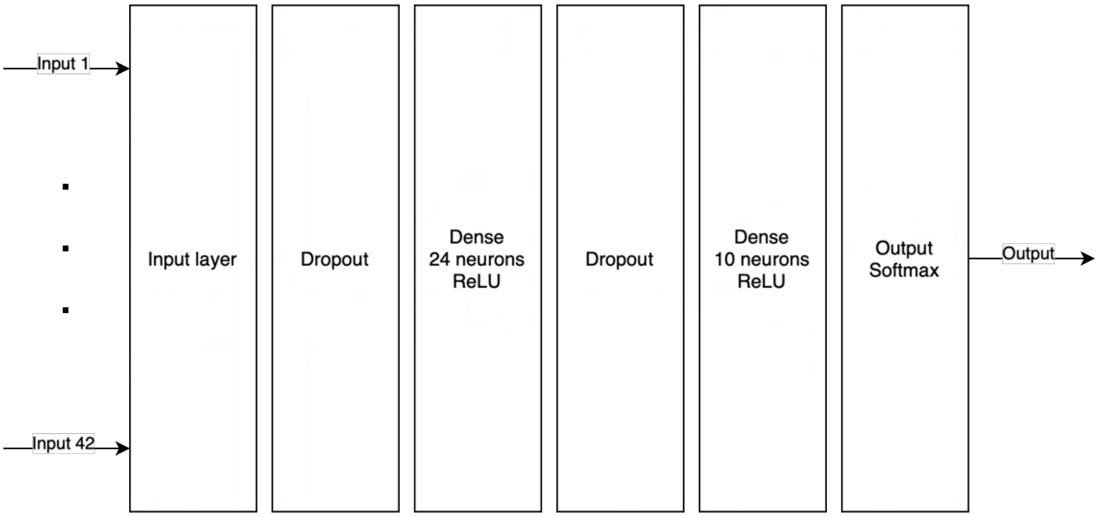


Figure 3.7: Deep feed-forward neural network for dynamic hand gestures.

Figure 3.7 presents the layers in the network. They are the same as those present in paragraph 3.3.2. The second one is an LSTM network.

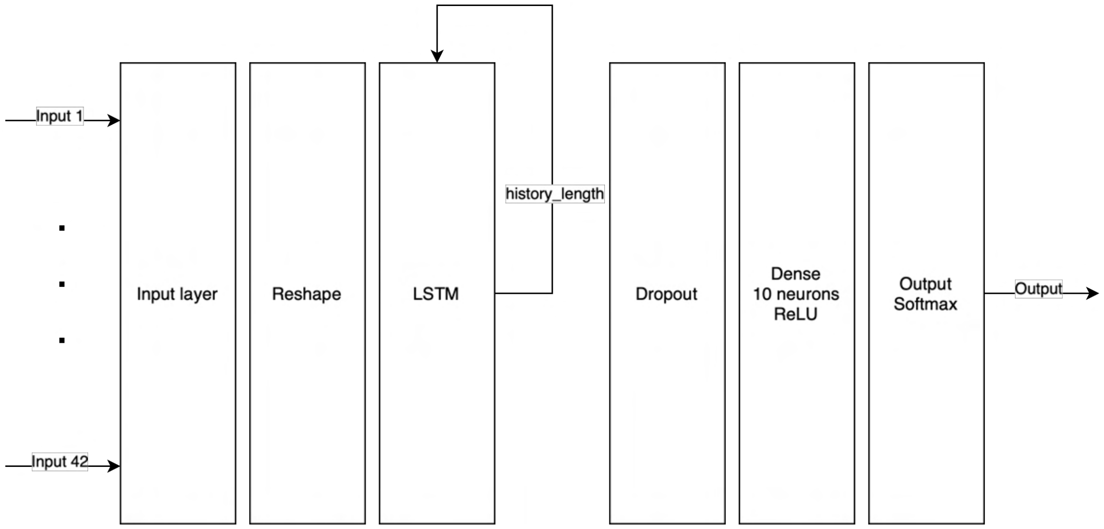


Figure 3.8: LSTM neural network for dynamic hand gestures.

In the model presented in figure 3.8, the input is first reshaped to group together the landmark coordinates from the same frame. Then an LSTM layer is used, cycling through the *history_length*. This model should perform better because it exploits the potential of a recurrent neural network to learn from a time series.

Training The training is similar to the one presented in paragraph 3.3.2, with the addition of the possibility for the users to choose which model to use.

Early stopping

Early stopping is the method adopted to specify an arbitrarily high number of epochs. With this technique, the end of the training is triggered by the recorded variation of a specified metric. In other words, when a metric does not change (or changes just a little) for a specified number of epochs, then the training can be considered ended.

This technique has been used in both static hand gesture training and dynamic hand gesture training. The metric monitored was the loss on the validation dataset, and the number of epochs to control was fifty.

3.3.3 Hand gesture controller

The hand gesture controller is the component that takes as input the recognized hand gestures and converts them into the correct command for the robot. To better have a modular system the hierarchy in figure 3.9 has been implemented.

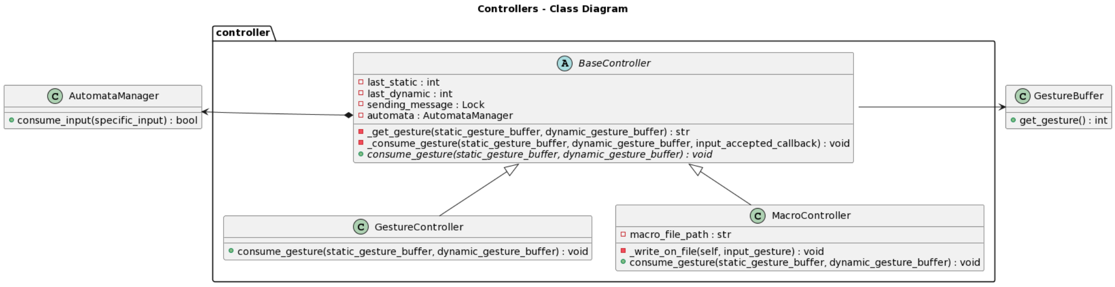

Controllers - Class Diagram
controller
A BaseController
last_static : int
last_dynamic : int
sending_message : Lock
automata : AutomataManager
_get_gesture(static_gesture_buffer, dynamic_gesture_buffer) : str
_consume_gesture(static_gesture_buffer, dynamic_gesture_buffer, input_accepted_callback) : void
consume_gesture(static_gesture_buffer, dynamic_gesture_buffer) : void
C AutomataManager
consume_input(specific_input) : bool
C GestureBuffer
get_gesture() : int
C GestureController
consume_gesture(static_gesture_buffer, dynamic_gesture_buffer) : void
C MacroController
macro_file_path : str
_write_on_file(self, input_gesture) : void
consume_gesture(static_gesture_buffer, dynamic_gesture_buffer) : void

Figure 3.9: UML class diagram for the hand gesture controller.

The class *BaseController* handles the communication between the hand gesture recognizer and the automaton that will communicate with the robot. The *HandGestureController* implements the method *consumer_ input* telling to the automaton to send the messages to the robot in contrast to the *MacroController* that does not. Instead, the *MacroController* saves on a text file the sequence of gestures made by the operators with the purpose to use them in another moment.

Automaton

To ensure the correctness of the sequence of gestures, users have to describe it as a finite-state automaton. The configuration file is a JSON file that requires the users to define:

- the initial state;
- a list of transitions. Each transition is defined by:
  - from which state it comes;
  - to which state it is directed;
  - with which element of the alphabet (Σ) it is triggered;
  - the action to perform when the transition is triggered. It can be:
    - **null** when no action is performed;
    - **set a navigation goal** specifying the coordinates;
    - **send a message** specifying on which topic publish the message, its type, and the raw data to send.

To better interact with the robot, the coordinates and the raw data for the messages can be interpolated with the element of the alphabet that trigger the transition. An example of configuration file can be found in B.1.

Four seconds elapse between the input of two commands, if the first one does not involve any action. Otherwise, before to accept another input the controller waits for the action to end. This is made to allow the users to change the gesture they are doing. The amount of time is configurable but four seconds have been shown to be sufficient.

There is not a common way to evaluate how much complex and understandable a JSON file is but, since it is a file that has to be written and read by an user, it is important to give an evaluation of it. The method adopted to calculate the complexity considers the depth and the type of fields per object:

- strings, numbers, and null values have a complexity of 1;

- arrays have a complexity equal to $1+(average\, complexity\, of\, nested\, elements)$;
- objects have complexity equal to the complexity of their elements;
- every nested elements have a weight d that multiplies the complexity score.

The algorithm to compute the complexity has been implemented in JavaScript and it is showed in A.1.

3.3.4 Integration with ROS

The integration with ROS is managed through two Python packages:

- **rclpy**: it comes with the installation of ROS as described in the documentation and it handles the raw communication between the hand gesture controller and the robot;
- **nav2**: package which implement the Nav2 navigation algorithm. It has been installed with pip after cloning the GitHub repository[2] with the following command:
  `pip3 install navigation2/nav2_simple_commander/`

First and foremost the *rclpy* is initialized with the statement `rclpy.init()`. Then, the hand gesture controller can either publish a message to a topic or set a goal for navigation based on the triggered automaton transition.

- **send a message**: the topic, the message type, and the data for the message are taken from the transition details described in the automaton configuration file. Then, the correct publisher is retrieved and used to publish the message. This kind of action exploits the message exchange system of topics explained in section 2.3.6;
- **set a navigation goal**: the coordinates are taken from the transition details described in the automaton configuration file. Then, the position is given as input to the navigator which returns a feedback until the task is completed. When the task is completed, it return the status of the operation:
  - **succeeded**: if the robot reach the goal correctly;
  - **cancelled**: if the operators cancelled the task;
  - **failed**: if the robot can't reach the set goal for any reason.

3.3.5 User interface

A command line interface is the first interface with which users interact. It asks a few questions about what users wants to do. They can answer writing on the terminal. It is designed to guide users in the execution of the possible tasks:

- launch the operation mode;
- launch the learning mode asking which dataset is to be edited and other gesture's data;
- launch the macro mode asking if users want to execute an existing macro or to create a new one.

Figure 3.10 show the complete activity diagram with all the possibilities for the users.

[2] https://github.com/ros-planning/navigation2.git

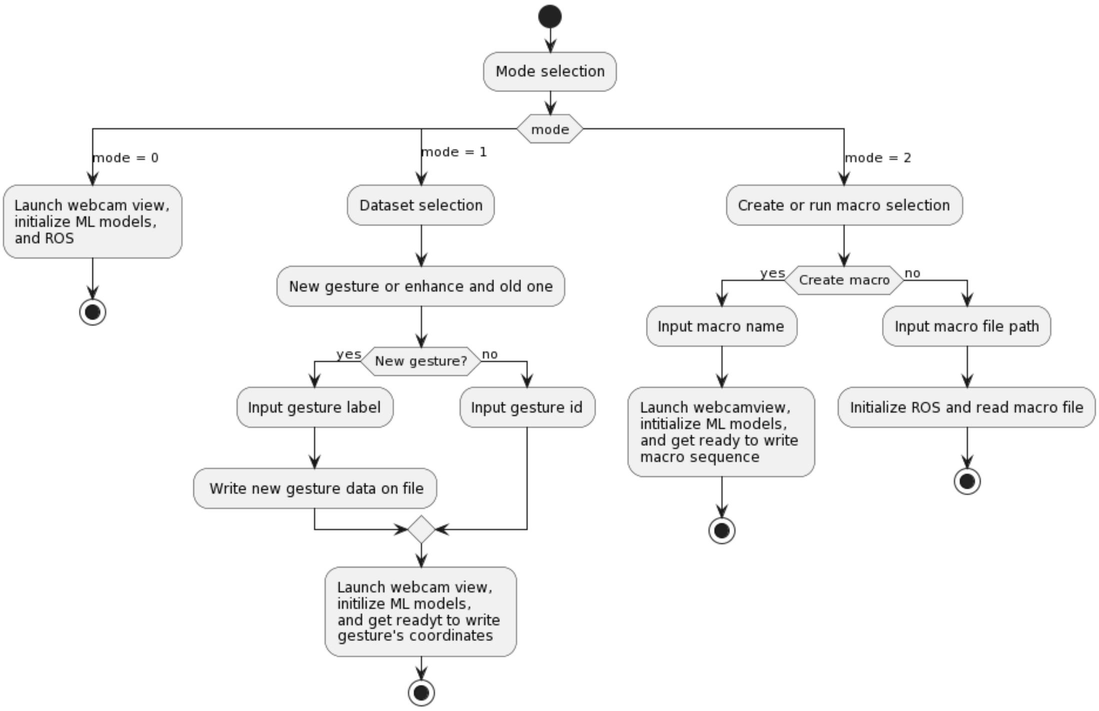
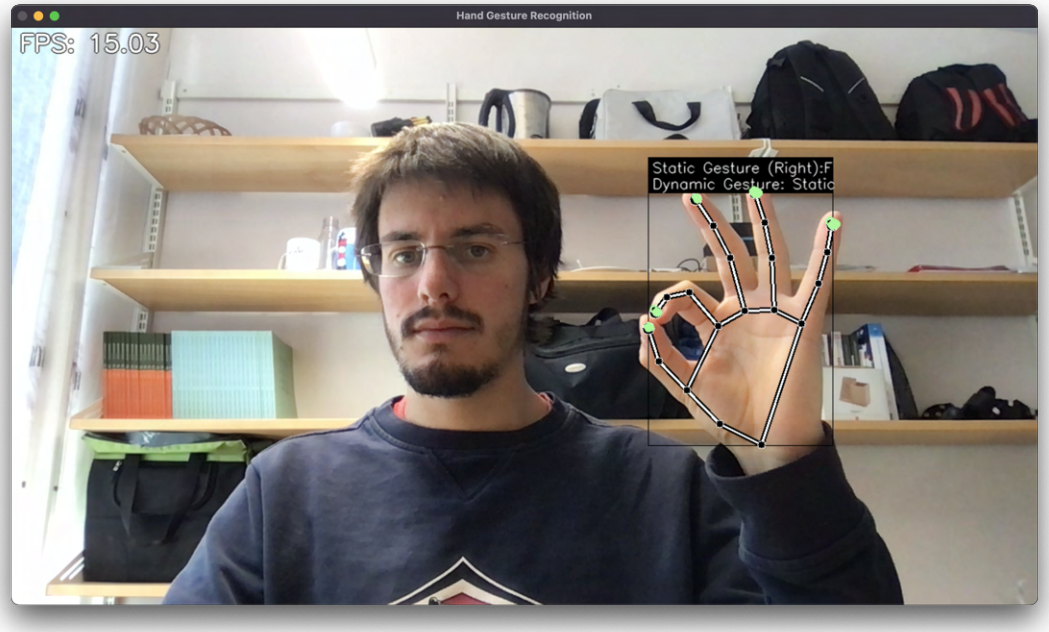

Mode selection
mode
mode = 0
mode = 1
mode = 2
Launch webcam view, initialize ML models, and ROS
Dataset selection
New gesture or enhance and old one
yes
New gesture?
no
Input gesture label
Input gesture id
Write new gesture data on file
Launch webcam view, initilize ML models, and get readyt to write gesture's coordinates
Create or run macro selection
yes
Create macro
no
Input macro name
Input macro file path
Launch webcamview, intitialize ML models, and get ready to write macro sequence
Initialize ROS and read macro file

Figure 3.10: Activity diagram of the command line interface.

When users select a mode that involves the use of the camera a window appears showing to them what the camera is watching, the number of FPS, the recognized gestures, and the selected mode. Moreover, it shows also the hand's landmarks (i.e. the black points), the joints between them, and the landmarks used for the history of key-points in green with a trail when the hand moves. Figure 3.11, for example, shows the recognition of the gesture *F* as static gesture and *Static* as dynamic.



FPS: 15.03

Static Gesture (Right):F
Dynamic Gesture: Static

Figure 3.11: User interface of the program when the webcam is used.

3.4 Documentation

To make the system as understandable as possible a proper documentation has been written. To ensure its readiness *MkDocs* has been used with the theme *MkDocs Material* and *Ligthgallery* to manage the images. Instead, to ensure its availability *GitHub Pages* and *GitHub Actions* have been used. In this way it is available for everyone.[3] The documentation explains how to:

- setup the environment;
- run the simulation;
- interact with the program;
- understand how the program works. In particular it explains:
  - the `main` function;
  - the `GestureDetector` class;
  - the `AutomataManager` class;
  - the `GestureController` class;
  - the `MacroController` class;
  - the `MacroRunner` class.
- train the neural networks;

[3] https://jjocram.github.io/hand-gesture/

3.5 Data collection

3.5.1 Hand gesture recognizer

To collect data from the hand gesture recognizer Tensorboard and Tensorflow metrics have been used. Tensorboard has been connected to the train scripts as the documentation explains. Moreover, the evaluation dataset, obtained from both the static hand gestures dataset and the dynamic hand gestures dataset has been exploited to evaluate the trained network on data it has never seen. The data gathered in this way are:

- accuracy;
- latest loss function value;
- time.

The time is taken exploiting Python's decorators. Before calling the `fit` method the time is taken and when the training is finished the time is taken again. The delta between the two times is the time spent for training.

3.5.2 System resource utilization

To collect data while the program is running I wrote a Python script, showed in A.2, that launches a process (e.g. the recognizer), collects CPU and RAM usage over time and plots them.

Chapter 4

Results

4.1 Configuration

To evaluate the system, first of all an automaton to describe the accepted users input has been designed. Then, several hyperparameters have been tried to train the neural networks that run the hand gesture recognizer.

4.1.1 Environment description

To train the networks a MacBook Pro 2019 with MacOS Montrey has been used. Instead, to test the integration with ROS the same machine has been used but with Ubuntu on a virtual machine with four physical core and $8GB$ of RAM. The virtual machine has been used because Ubuntu is better integrated with ROS and Gazebo. Moreover, a seed has been set to Tensorflow and Pandas (used for the split of the dataset) to make the results reproducible.

4.1.2 Automaton

The system is composed of two components: a node that deals with hand gesture recognition and a node representing the robot to be controlled. The tasks that the robot must perform are:

- pick up a parcel;
- drop down a parcel;
- go to a predetermined position.

A letter identified positions and parcels. In this way, the ASL is exploitable in order to simulate the identification of positions and parcels.

Figure 4.1 shows the transition diagram for the automaton designed to check the users input for these tasks. The alphabet is $\Sigma = \{[A{-}Z], go\_to, pick\_up, drop\_down\}$ and the states are:

- q_0: robot without parcel;
- q_1: robot waiting for parcel id;
- q_2: robot waiting for a “direction” without a parcel;
- q_3: robot with a parcel;
- q_4: robot waiting for a “direction” with a parcel.

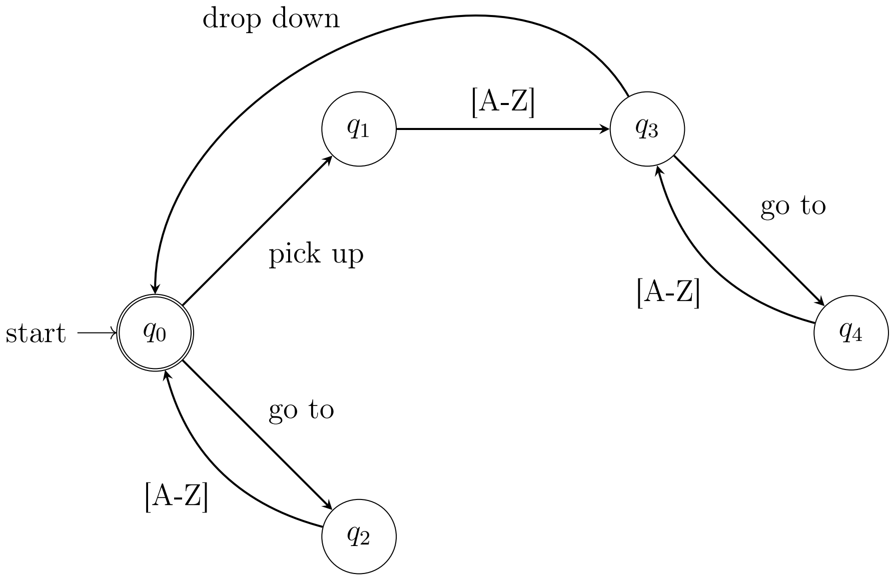


Figure 4.1: Automaton diagram for commands.

To implement the automaton in figure 4.1 the configuration file in B.1 has been used, and its complexity is 3.72. The complexity has been computed with the method described in 3.3.3

4.2 Hand gesture recognizer

4.2.1 Static hand gestures

To train the static hand gesture recognizer the script described in paragraph 3.3.2 has been used. In particular six tests have been performed in order to find the best configuration.

Dataset

The dataset is composed of 3707 elements divided in 24 labels. The labels are the letters of the alphabet, excluding J and Z because they involve a movement. Their distribution is shown in figure 4.2.

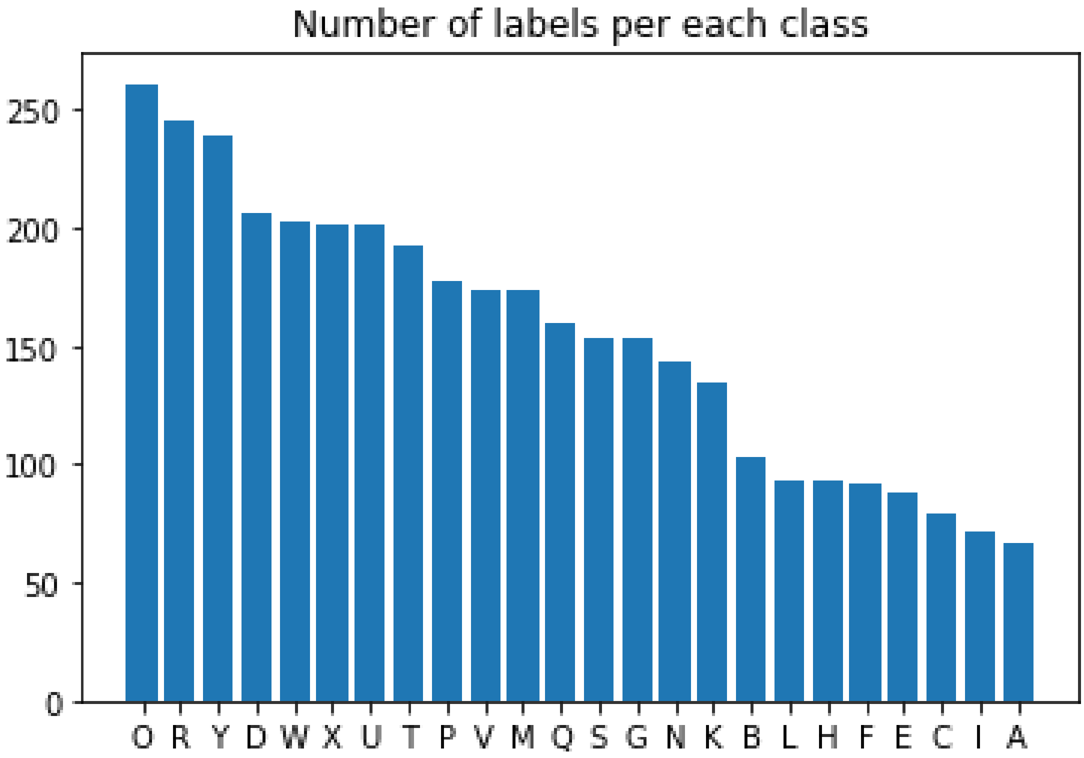


Figure 4.2: Class distribution in the static hand gestures dataset.

Tests

In table 4.1 the tests performed to find the best configuration to recognize static hand gestures are listed. “Not defined” in the column of “Number of elements per class” means that the whole dataset, described in 4.2.1, has been used to train the network.

Test name	Number of elements per class	Dropout rate
full_dataset	Not defined	0.0
full_dataset_dropout	Not defined	0.2
10_samples	10	0.0
7_samples	7	0.0
6_samples	6	0.0
5_samples	5	0.0

Table 4.1: Training configurations for the static hand gesture classifier.

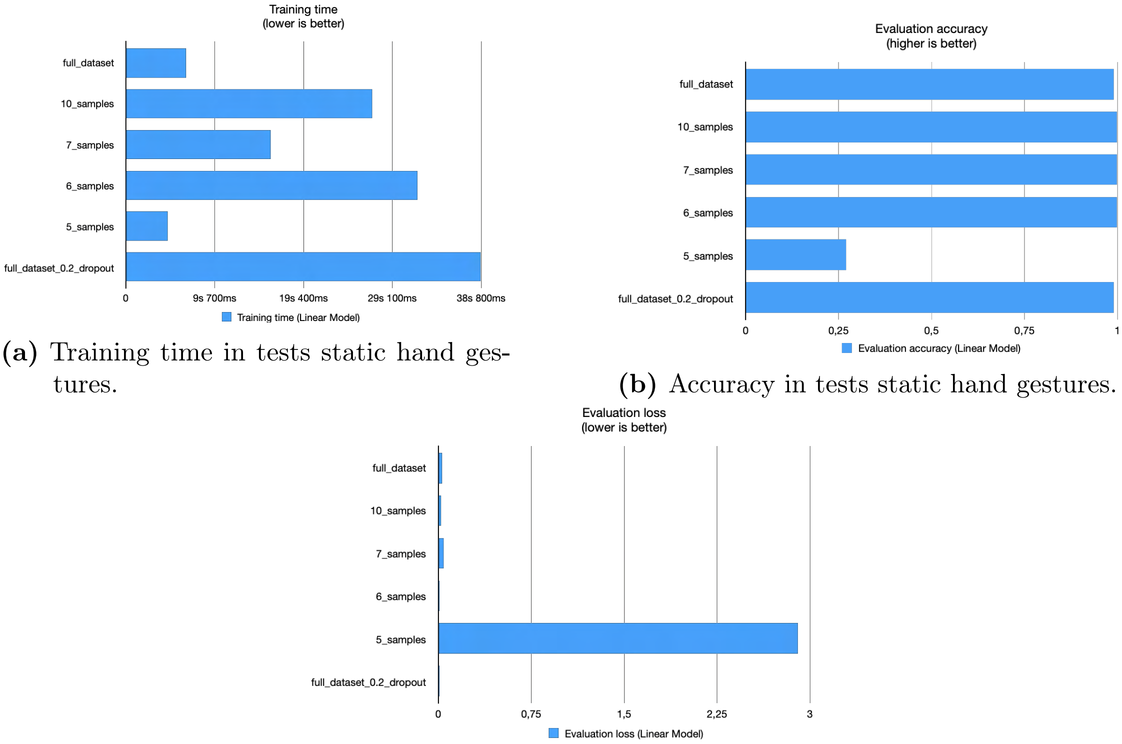
I mainly focused on finding the minimum amount of samples in order to obtain a good accuracy and a short training time.

Results

The data gathered during the training are shown in table 4.2 and has been plotted. The graphs are shown in figure 4.3 and 4.7.

Test name	Evaluation accuracy	Evaluation loss	Training time
full_dataset	0.99	0.03	6s 587ms
full_dataset_dropout	0.99	0.01	38s 769ms
10_samples	1	0.02	26s 883ms
7_samples	1	0.04	15s 804ms
6_samples	1	0.009	31s 839ms
5_samples	0.27	2.90	4s 551ms

Table 4.2: Results for training the static hand gesture recognizer.

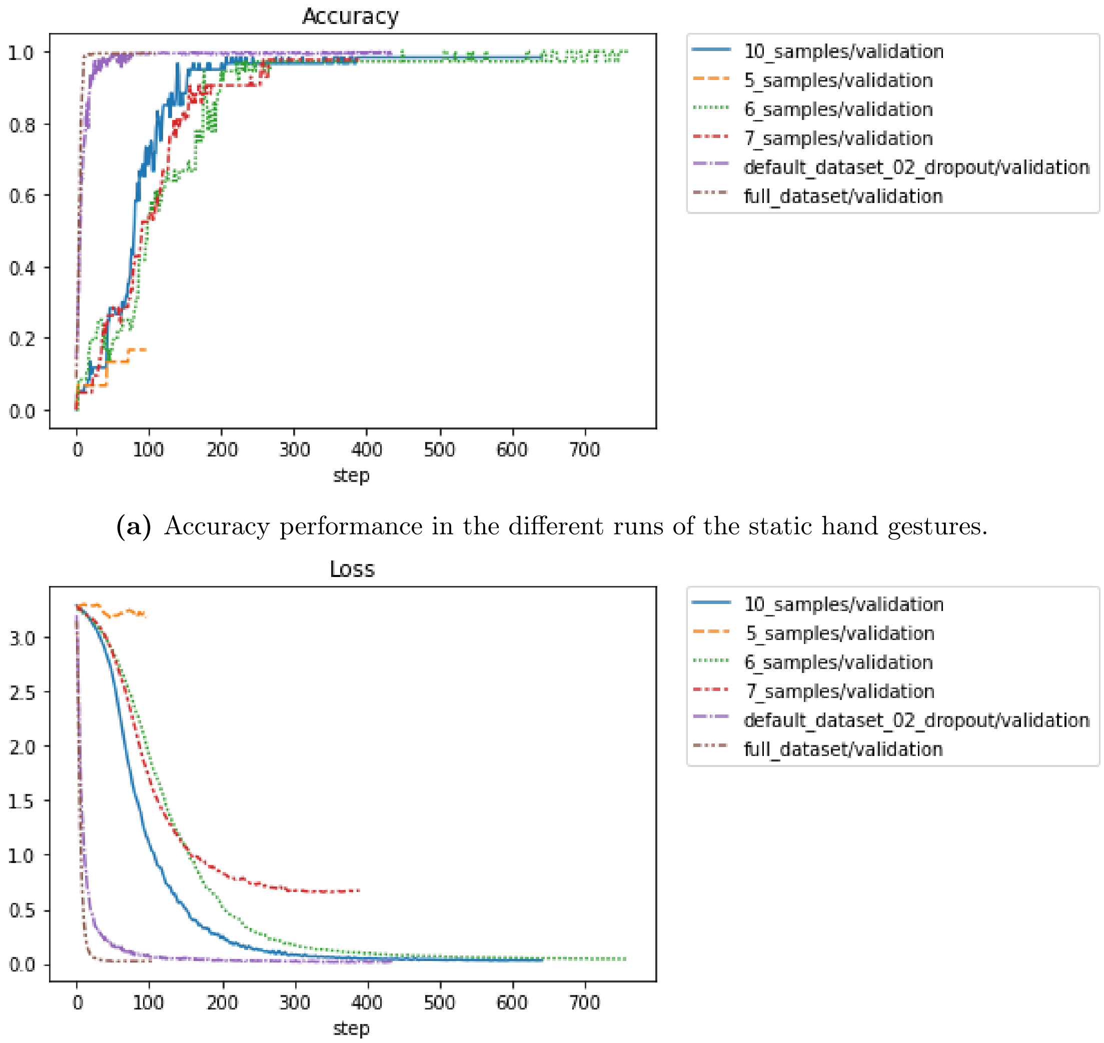


(a) Training time in tests static hand gestures.

(b) Accuracy in tests static hand gestures.

(c) Loss in tests static hand gestures.

Figure 4.3: Results graphs for the training of the static hand gesture recognizer.



(a) Accuracy performance in the different runs of the static hand gestures.

(b) Loss performance in the different runs of the static hand gestures.

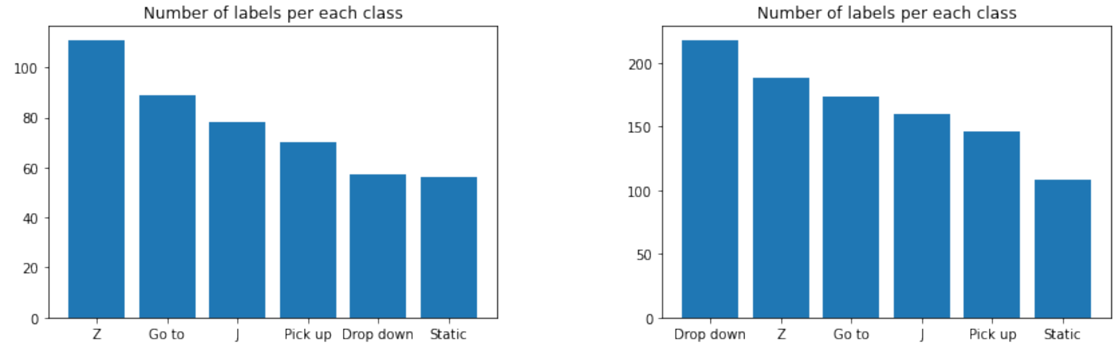
Figure 4.4: Performances in the different runs of the static hand gestures.

4.2.2 Dynamic hand gesture

Dataset

Two datasets have been used:

- **five_fingers**: in this dataset the history of the tip of every finger has been saved. There are 461 elements with a total size of $1.1MB$. Its class distribution is shown in figure 4.5a;
- **one_finger**: in this dataset only the history of the tip of the index finger has been saved. There are 994 elements with a total size of $423KB$. Its class distribution is shown in figure 4.5b.



(a) Class distribution in the dynamic hand gestures dataset with five fingers.

(b) Class distribution in the dynamic hand gestures dataset with one finger.

Figure 4.5: Class distribution in the dynamic hand gestures datasets.

Tests

In table 4.3 the tests performed to find the best configuration to recognize dynamic hand gestures are listed. "Not defined" in the column "Number of elements per class" means that the whole dataset (specified in column "Dataset") has been used.

Test name	Dataset	Network type	Number of elements per class
full_dataset	five_fingers	feed_forward	Not defined
50_samples	five_fingers	feed_forward	50
10_samples	five_fingers	feed_forward	10
full_dataset	five_fingers	lstm	Not defined
50_samples	five_fingers	lstm	50
10_samples	five_fingers	lstm	10
full_dataset	one_finger	feed_forward	Not defined
50_samples	one_finger	feed_forward	50
10_samples	one_finger	feed_forward	10

Table 4.3: Training configurations for the dynamic hand gesture classifier.

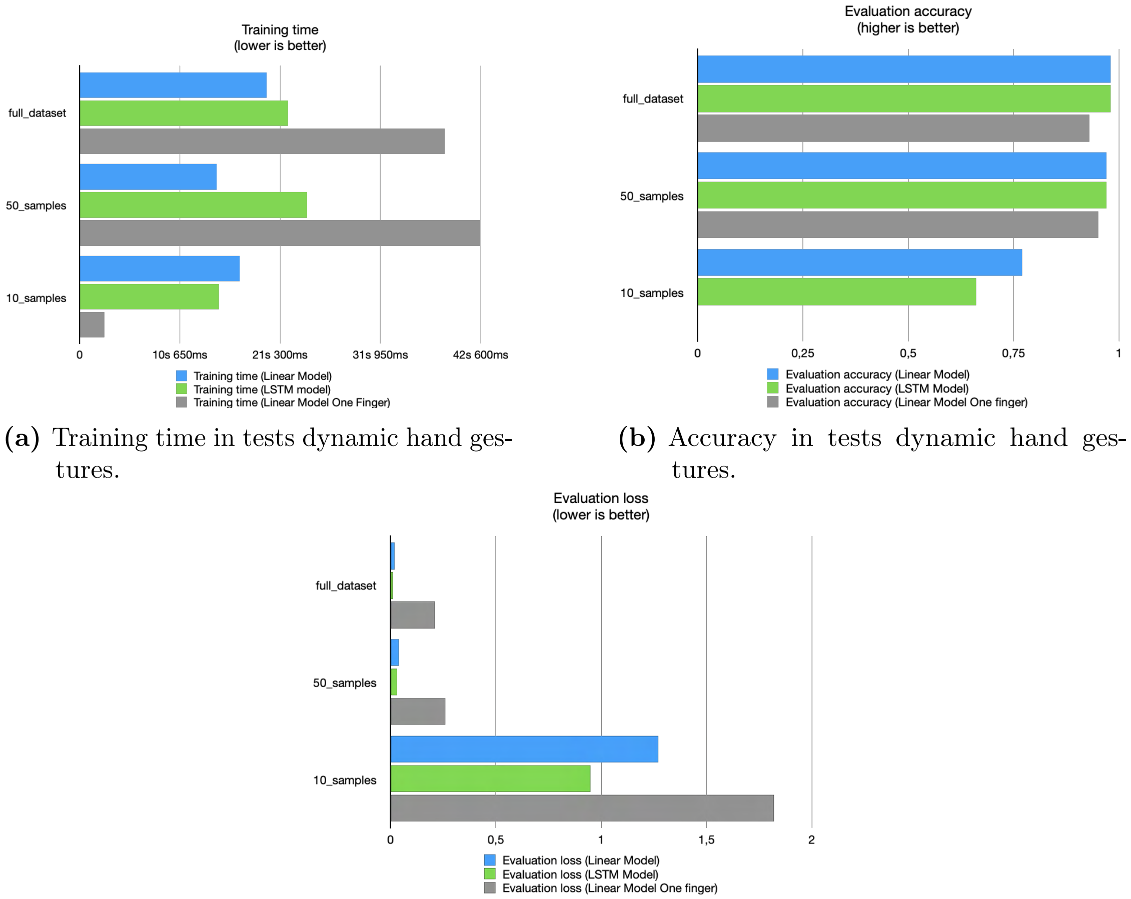
As for the static hand gesture recognizer, I focused on finding the minimum size of the dataset in order to obtain a good accuracy and a short training time.

Results

The data gathered during the training are shown in table 4.4 and has been plotted. The graphs are shown in figure 4.6 and 4.7.

Test name	Dataset	Network type	Evaluation accuracy	Evaluation loss	Training time
full_dataset	five_fingers	feed_forward	0.98	0.02	19s 878ms
50_samples	five_fingers	feed_forward	0.97	0.04	14s 558ms
10_samples	five_fingers	feed_forward	0.77	1,27	16s 970ms
full_dataset	five_fingers	lstm	0.98	0.01	22s 110ms
50_samples	five_fingers	lstm	0.97	0.03	24s 161ms
10_samples	five_fingers	lstm	0.66	0.95	14s 798ms
full_dataset	one_finger	feed_forward	0.93	0.21	38s 749ms
50_samples	one_finger	feed_forward	0.95	0.26	42s 535ms
10_samples	one_finger	feed_forward	0	1.82	2s 622ms

Table 4.4: Results for training the dynamic hand gesture recognizer.

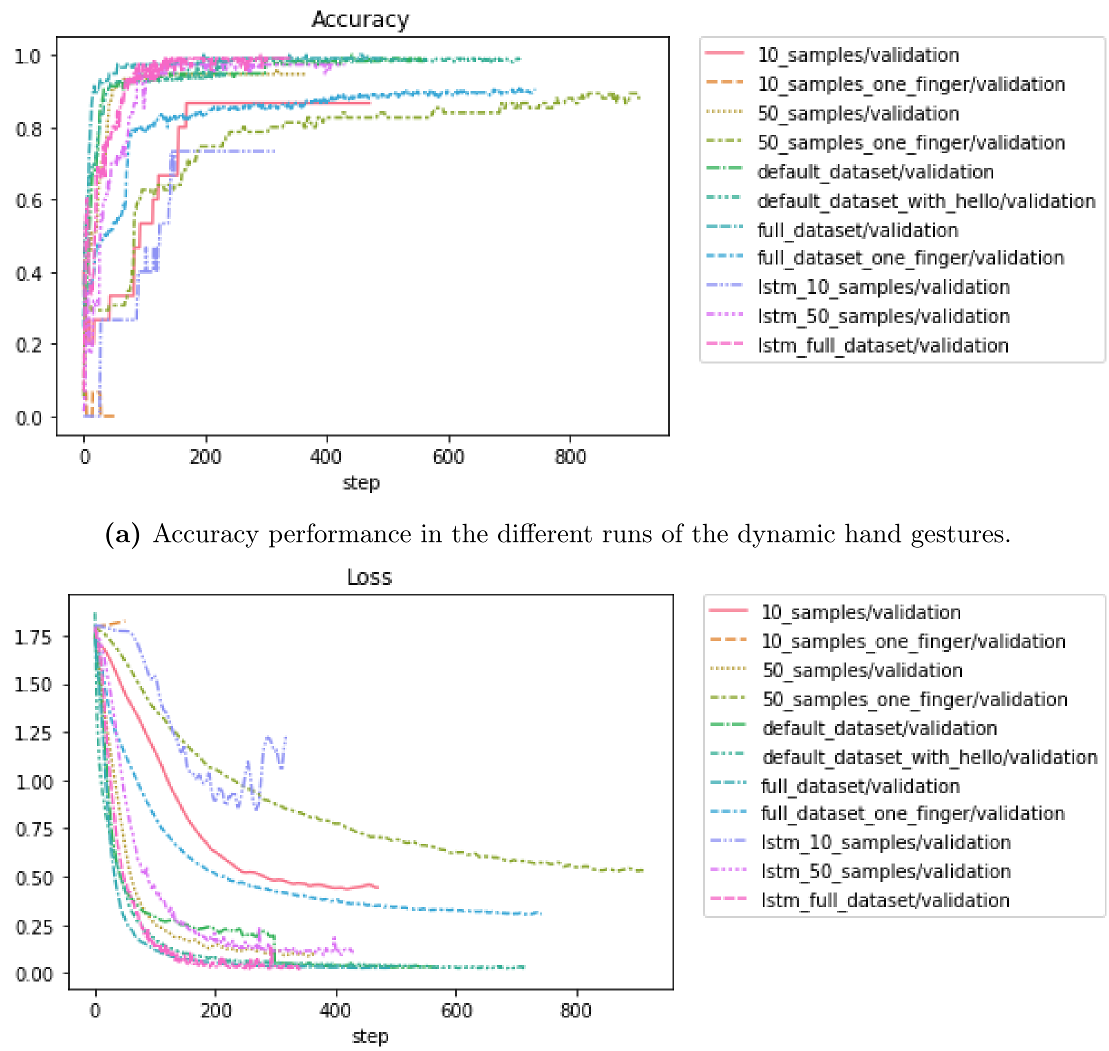


(a) Training time in tests dynamic hand gestures.

(b) Accuracy in tests dynamic hand gestures.

(c) Loss in tests dynamic hand gestures.

Figure 4.6: Results graphs for the training of the dynamic hand gesture recognizer



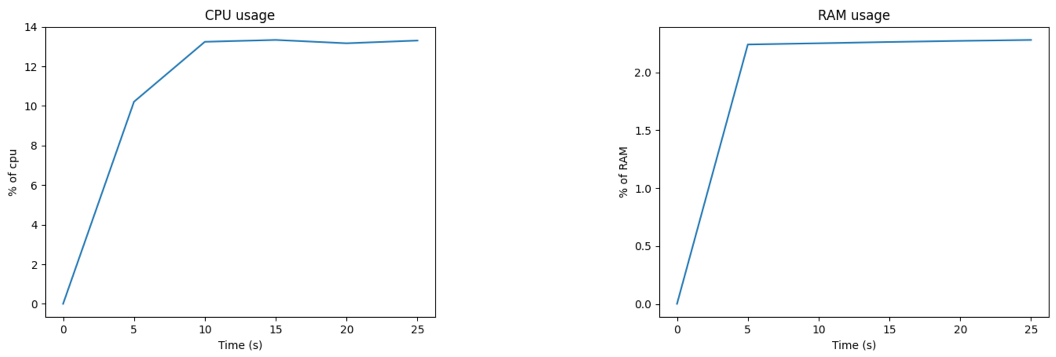
(a) Accuracy performance in the different runs of the dynamic hand gestures.

(b) Loss performance in the different runs of the dynamic hand gestures.

Figure 4.7: Performances in the different runs of the dynamic hand gestures.

4.2.3 System resource usage during training

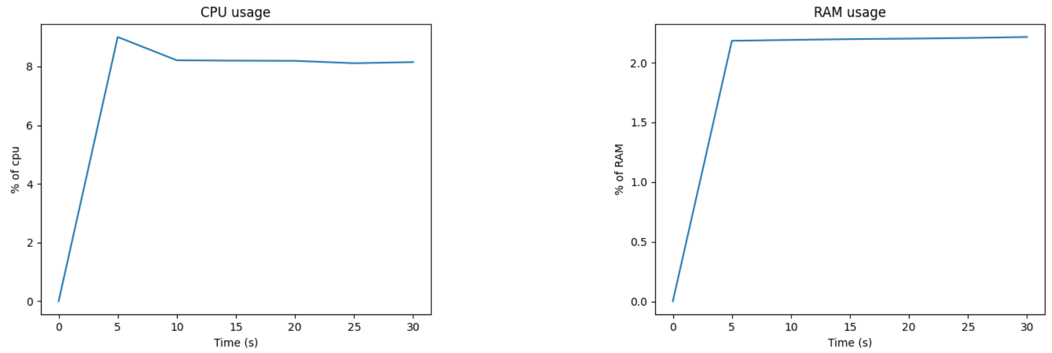
A test has been performed to collect data on the use of system resources, in particular CPU and RAM usage. I added one new static and one new dynamic gesture and then I used the method described in section 3.5.2 to collect the usage of CPU and RAM of the program during the training process. The training process has been executed on the host machine and not in the virtual machine. The figures 4.8 and 4.9 show the results obtained.



(a) CPU usage.

(b) RAM usage.

Figure 4.8: System resource utilization during the static gesture recognizer training.



(a) CPU usage.

(b) RAM usage.

Figure 4.9: System resource utilization during the dynamic gesture recognizer training.

4.3 Integration with ROS

Three incremental tests have been performed to test the correct integration with ROS:

1. **message exchange**: the first test was about the exchange of different messages when different gestures were recognized;
2. **navigation system**: the next step was the correct integration with the navigation system “Nav2” in a simple environment;
3. **complex scenario**: the last test carried on was on a more real scenario, a robot in a warehouse doing different tasks.

In these three tests the network topology was similar:

- the framework creates a node to communicate with the rest of the network;
- the simulated robot could be considerated like another node;
- when the navigation system was integrated, it could be considered like another node.

The framework’s node communicates with the other two nodes and, obviously, the navigation system’s node communicates with the robot to tell it where it has to go. Figure 4.10 shows the network topology implemented to carry on the tests.

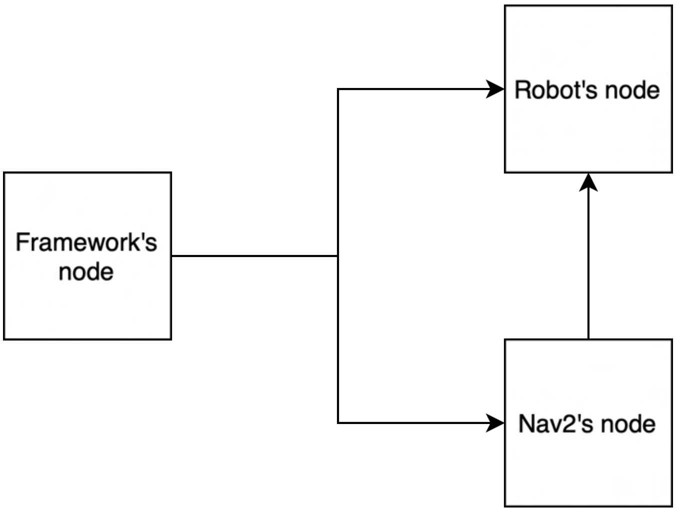


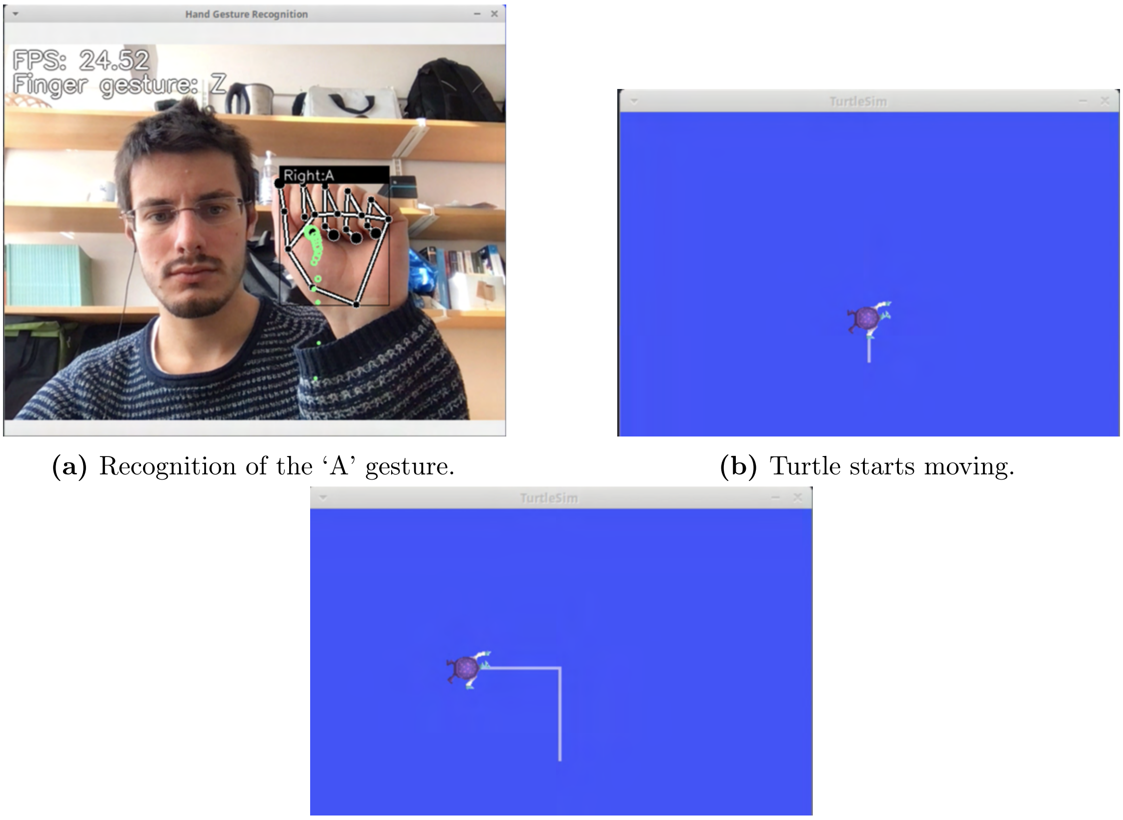
Figure 4.10: ROS' network topology

4.3.1 Integration with ROS' topic

The first results achieved by integrating the framework with ROS have been about the communication between two nodes: the one representing the hand gestures recognizer and the *TurtleSim*. *TurtleSim* is a "simulator" provided by ROS which takes inspiration from the turtle programming languages, where you control a turtle by telling it to go forward, turn left, and turn right. The turtle has a pen attached to it that draws the path on the screen. Figure 4.11 shows what it looks like at the startup.

Figure 4.11: TurtleSim.

At this point in the implementation, the hand gesture recognizer creates a node and communicates with the *TurtleSim* exchanging messages on two topics. The messages sent represented the direction and the velocity of the turtle. Four positions were defined, each of them linked to a static hand gesture (A, B, C, and D). Based on the gesture recognized, a sequence of messages were picked and sent to the turtle. These messages told the turtle how to reach the target position from its current position.



(a) Recognition of the ‘A’ gesture.

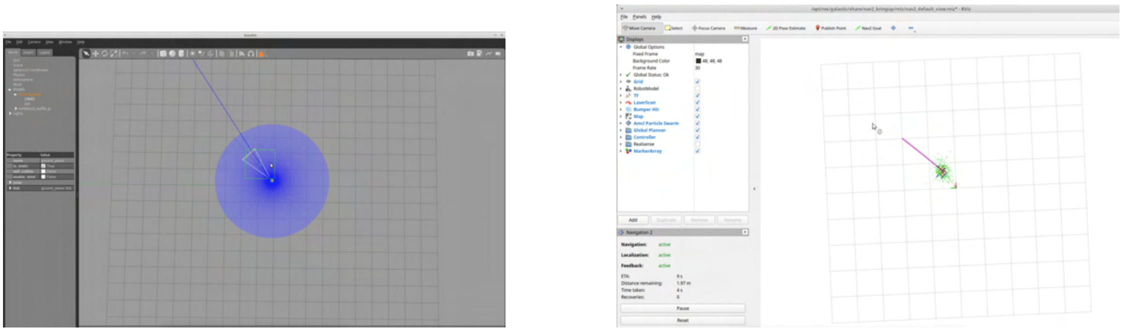
(b) Turtle starts moving.

(c) Turtle reaches position ‘A’.

Figure 4.12: Recognition of the ‘A’ gesture and turtle movement toward position ‘A’.

4.3.2 Integration with the navigation system

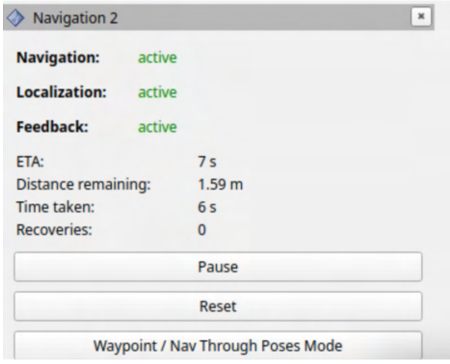
The next step has been testing the integration with the navigation system “Nav2”. I translated the test made with the *TurtleSim* in a Gazebo World. The map used was an empty map (figure 4.13a) to ensure that communication between the hand gesture recognizer and the navigation system is working properly. The test had been successful and the simulated robot moved into the correct position. Figure 4.13b shows the RViz tool, which is integrated with the navigation system and shows the path the robot is following. RViz also gives some feedback and actions regarding the navigation status, as figure 4.14 shows.



(a) Gazebo Empty World.

(b) RViz showing the path to position ‘A’.

Figure 4.13: Test navigation system in a Gazebo Empty World.



Navigation 2

Navigation: active

Localization: active

Feedback: active

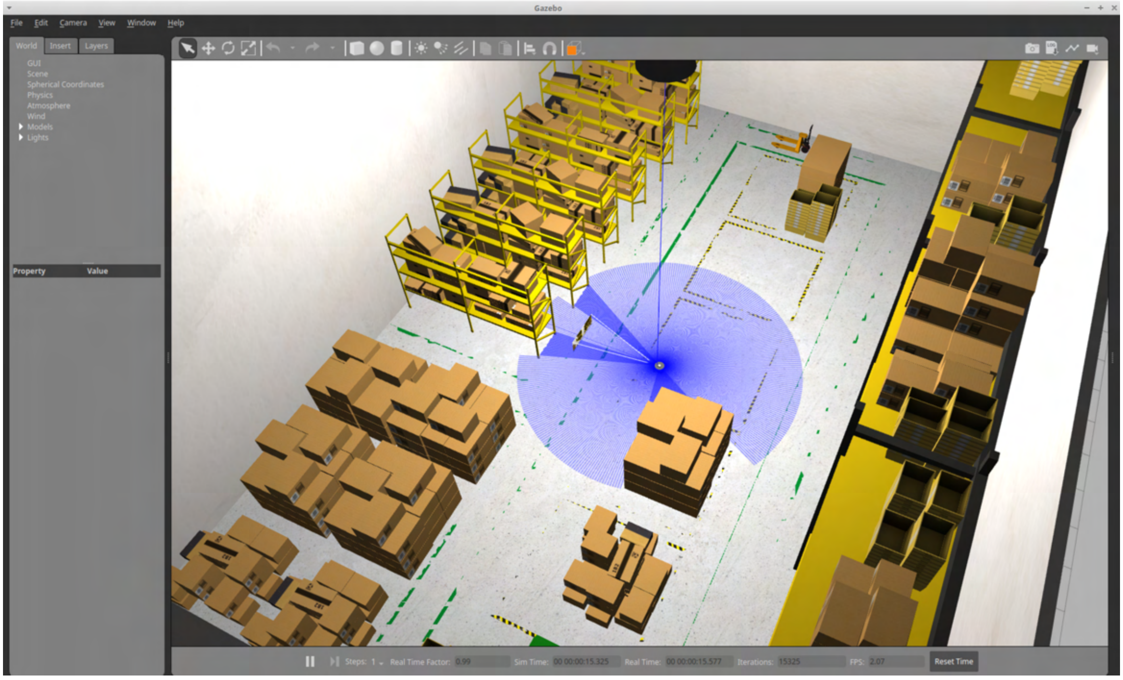
```
ETA:                  7 s
Distance remaining:   1.59 m
Time taken:           6 s
Recoveries:           0
```

Figure 4.14: RViz detail - feedback and actions.

4.3.3 Test in a warehouse environment

The last test to carry on regarding the integration with ROS was in a more real scenario. The small warehouse environment provided by amazon[1], shown in figure 4.15, has been chosen to test the framework’s final implementation with the automaton described in 4.1.2. In the world, the *TurtleBot v3* has been inserted, which executes commands given via hand gestures recognized by the framework.

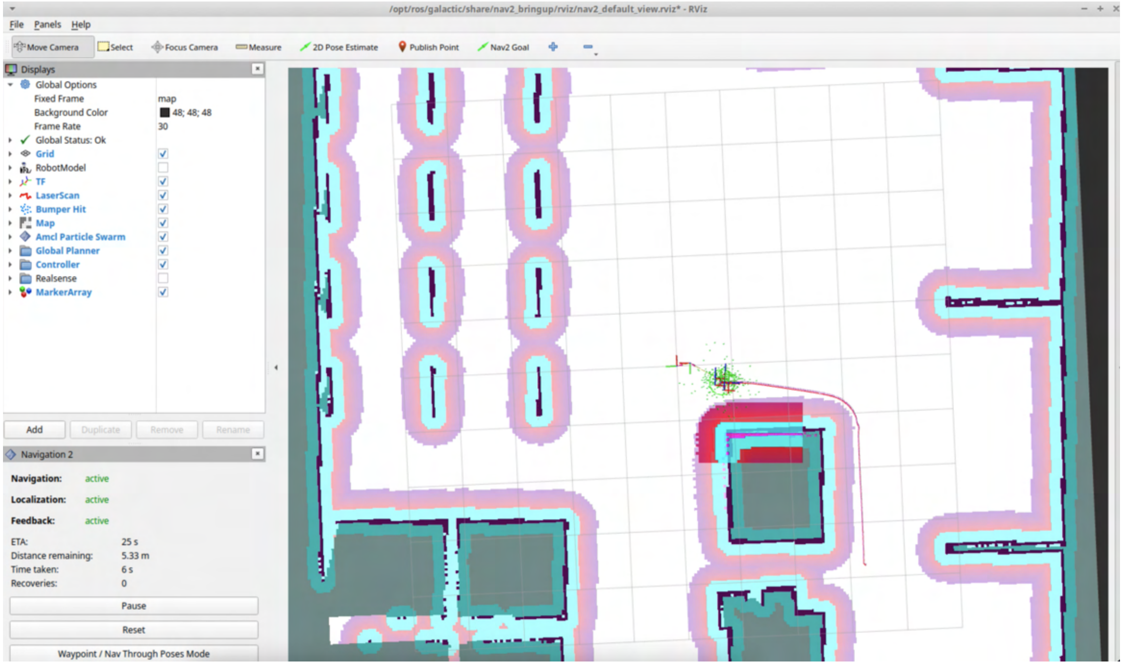
[1] https://github.com/aws-robotics/aws-robomaker-small-warehouse-world



Steps: 1 Real Time Factor: 0.99 Sim Time: 00 00:00:15.325 Real Time: 00 00:00:15.577 Iterations: 15325 FPS: 2.07

Figure 4.15: AWS Robomaker's small warehouse Gazebo world.

When a complex world is used, RViz provides more interesting data. It displays the obstacles seen by the robot as well as the path computed by the "Nav2" algorithm to avoid them, for example. Figure 4.16 shows the RViz's interface when the small warehouse world is loaded.



Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
Grid	☑
RobotModel	☐
TF	☑
LaserScan	☑
Bumper Hit	☑
Map	☑
Amcl Particle Swarm	☑
Global Planner	☑
Controller	☑
Realsense	☐
MarkerArray	☑

Navigation: active
Localization: active
Feedback: active

ETA: 25 s
Distance remaining: 5.33 m
Time taken: 6 s
Recoveries: 0

Figure 4.16: RViz's interface during simulation in the small warehouse world.

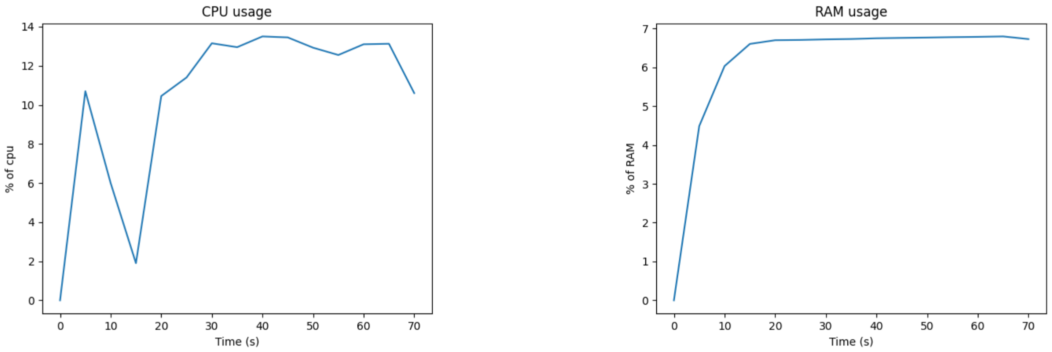
The test carried on regarding the recognition of several sequences of gestures involving both dynamic and static hand gestures, the set of navigation goals, and the proper reception of some messages exchanged on a ROS' topic. Moreover, with this whole environment set up, some data has been gathered during the simulation to prove the capabilities and lightness of the framework.

4.3.4 System resource usage

As in 4.2.3, a tests has been performed to gather some data about the system resource usage during the execution of the program in `operational mode`. This test has been executed in the virtual machine. I ran the simulator and the navigation tool. Then, I used the method described in section 3.5.2 to collect the usage of CPU and RAM of the program. I performed the following sequence of gesture:

1. Go to;
2. B;
3. Pick up;
4. D;
5. Go to;
6. C;
7. Drop down.

The results are shown in figure 4.17



(a) CPU usage. **(b)** RAM usage.

Figure 4.17: System resource utilization during the "operational mode"

4.3.5 Execution time

As explained in 3.3.3 the execution time between two actions is four seconds, at least. The execution time of the action performed by the robot is not related to the framework. Meanwhile, I performed some tests to find which is the minimal amount of time to wait before receiving the next command and four seconds resulted to be the best choice. But, this amount of time is subjective so, it can be changed by the configuration.

Chapter 5

Discussion

5.1 Hand gestures

To choose the dynamic hand gestures to use I took inspiration from the literature. I tried to invent my own gestures but, without the possibility to perform a preliminary study involving people from the context of use, it would have resulted in unrealistic and unusable gestures in the real world.

Regarding the static hand gestures, the choice to use the ASL seemed an obvious choice to me because it is the common way to express letters using hands in societies where the Latin alphabet is used.

5.2 Deep learning models

The results achieved with the deep learning models to recognize hand gestures are astonishing. The choice to use MediaPipe has proven to be a winning one. While I was designing the ML part of the project, I took into consideration the idea of using a YOLO network to recognize the hand gestures because it is known as one of the best networks to work on images and videos. But, the downside of this choice would have been the necessity of a large dataset to train the network. On the Internet, there are a lot of projects that use YOLO to recognize hand gesture.[1] However, none of them are easily adaptable to different scenarios, or, to put it another way, it is extremely difficult to add a new gesture. Moreover, YOLO utilizes images or sequences of them in the case of video, and this is another drawback because when training on images, the network has to manage the environment (i.e., where the image is taken), the lightning, and also the skin color.

The method utilized (i.e. MediaPipe's landmarks coordinates plus a simple feed-forward neural network) leaves all the above-explained drawbacks to MediaPipe engineers because it uses only the landmarks' coordinates (i.e. numbers) to classify the different hand gestures. Obviously, MediaPipe could be substituted with another ML model capable of performing hand tracking and returning the coordinates of

[1] https://www.kaggle.com/search?q=hand+gesture

the conjunctions of the hand, but, looking at the results obtained, it seems to be a very good solution.

5.2.1 Static hand gestures recognizer

The feed-forward deep neural network used to classify the static hand gestures turned out to be a very good solution with an accuracy greater than the 99% and a time of only 6 seconds to train it. Moreover, the results show that it works even with a very small dataset. Furthermore, the time to add a new gesture and train the network is very low.

5.2.2 Dynamic hand gestures recognizer

The two kinds of networks tested, the feed forward network and the LSTM network, achieved similar results. I expected that the latter would perform better than the former because the LSTM network should achieve better results in the context where past data matters. A difference could not be noticed because of the short sequence of landmarks that have been used to train the networks.

Moreover, the results show that the more landmarks are taken into consideration, the more accurate the prediction is. When the tips of all fingers are used, the network achieves an accuracy of 98%. Instead, with just the tip of the index finger, the accuracy is 93%, and the difference increases if the dataset size is reduced. Furthermore, the time spent on training the network is doubled when the *one_finger* dataset is used.

5.3 Integration with ROS

The requirement to work with ROS has been fulfilled. The implementation allows other users to expand the capabilities of the framework. Up to now, it can work with Nav2 to guide the robot to a position and send any type of message to the robot utilizing the ROS topic. In the future, the framework could be expanded, making it compatible with other packages and more ROS' built-ins.

5.4 Resource utilization

The data in 4.2.3 shows that the training process is fast and not resource-demanding. During the training, RAM consumption is only 2% of 16GB. Meanwhile, the CPU usage is around 14% in the case of the static hand gesture recognizer and 8% in the case of the dynamic hand gesture recognizer.

The data gathered during the simulations about the system resource utilization shows that the framework is not resource demanding. All simulations were run in a virtual machine equipped with a 4 CPU core and 8GB of RAM. Regarding the CPU, the results show that after the initialization (the first spike in figure 4.17a) the utilization is between $12\% - 14\%$. While, as far as RAM is concerned, its occupancy stabilizes below 7% and thus for a total of $\sim 560MB$.

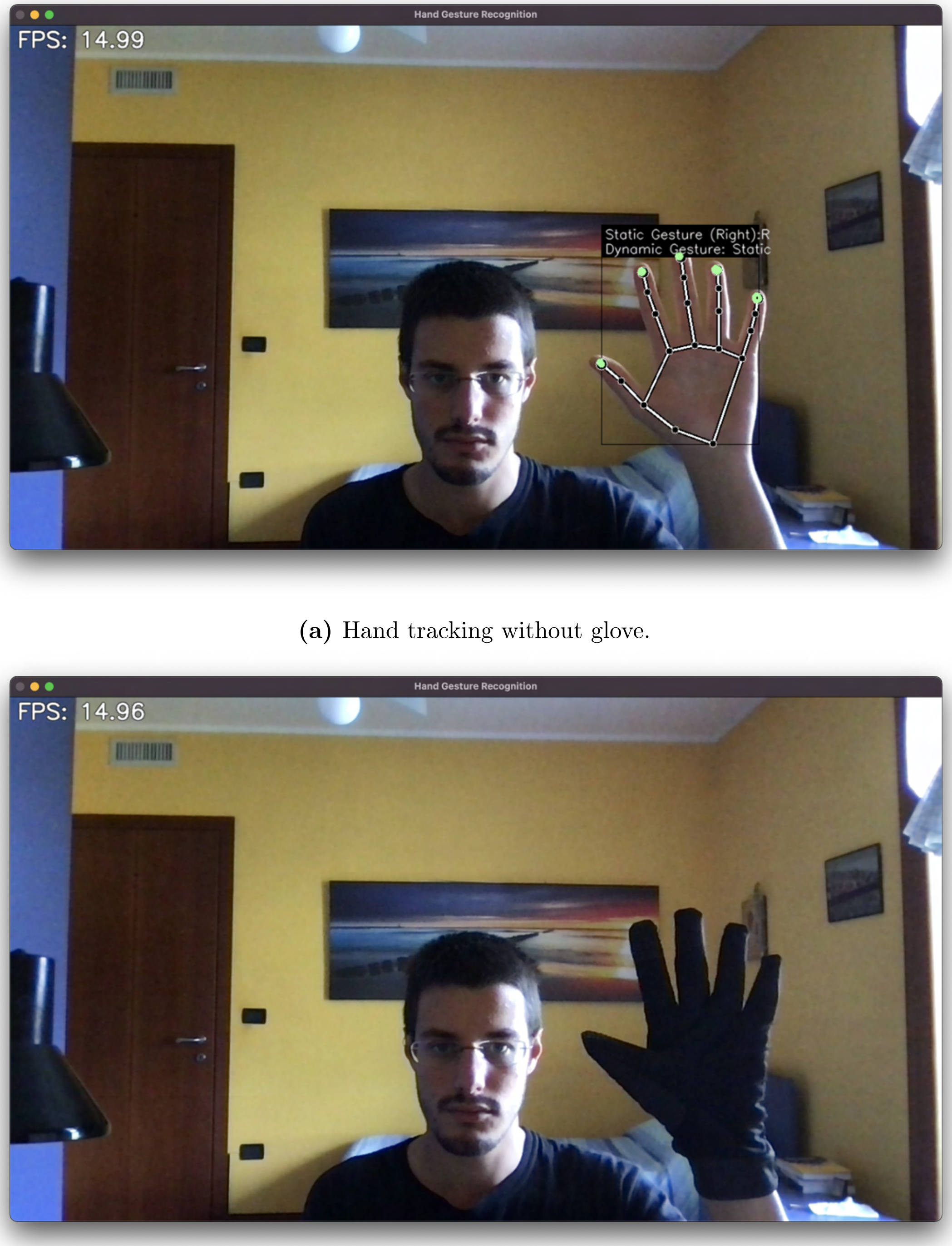
Looking at the results, NVIDIA's Jetson Nano, with its $4GB$ of RAM, a four core CPU, and GPU, could be taken into consideration as a possible hardware to test the framework in a real environment. Maybe one of the latest Raspberry-Pi could handle it as well. This is an important fact because those two micro-computers are among the most widely used for the development of robotic applications.

5.5 Complexity score

There is no common way to evaluate the complexity of a JSON file, as explained in 3.3.3. The method adopted has been chosen to give an objective evaluation of the difficulty of writing and reading a JSON file to meet the non-functional requirement of being easily configurable.
Therefore, it is important to have a low complexity score obtained. To achieve this goal, it is important to wisely design the automaton that describes the accepted input. For example, one can verify that the represented automaton is in its minimized version. The automaton obtained from the configuration can be proven to be in its minimized version.

5.6 Problems

The solution proposed is not without issues. An easy to fix one is the lack of functionalities regarding the integration with ROS. Up to now, the integration is just at the beginning. The framework can only work with Nav2 as the navigation system and can only send messages through topics. Thanks to how it has been implemented, the framework's developers can integrate new functionalities and other packages very easily. It is sufficient to write the wrapper and update the accepted actions in the `AutomataManager` class.
Another problem regards MediaPipe. It is a black box from the point of view of the implementation because the framework gives in input a frame and receives in output a list of landmarks. As explained in 3.2.3, MediaPipe is a very good tool with a high level of accuracy, but it is not without problems. For example, in some scenarios it can not recognize the hand, and without that functionality, the whole framework falls apart. Figure 5.1 shows the difference when the users wear a glove and when they do not. In figure 5.1a the hand tracking works perfectly as the presence of landmarks is highlighted by the black dots on the hand. Meanwhile, in figure 5.1b, the glove prevents MediaPipe from working correctly and no landmark is recognized. This is an important threat to the framework's functionality as it could be used in some environments where the users wear clothes or there are other things that block the key component of the framework from doing its job. Most importantly, there is nothing that can be done to resolve this issue other than to wait for MediaPipe's engineer to find a solution or change tool.



(a) Hand tracking without glove.

(b) Hand tracking with glove.

Figure 5.1: Difference between wearing a glove and not for MediaPipe.

Chapter 6

Conclusion

Through the work done in this thesis, a novel framework to simplify human robot interaction has been proposed, implemented with Python, tested in a Gazebo simulation, and documented to enable other people to use and improve it. The idea behind the framework is to use the hand to control a robot. This was successfully achieved by exploiting MediaPipe to track hands in real-time and two deep neural networks to classify which hand gesture the users are doing by exploiting the coordinates of the hand's conjunctions obtained from MediaPipe. Moreover, the framework handles the communication with the robot through the ROS framework. In this way, the developed framework can communicate with any kind of robot, as long as it is compatible with ROS.

Thanks to the combination of MediaPipe and light deep neural networks (less than ten layers) a set of dynamic and static hand gestures can be recognized in real-time and the training process is really fast. Moreover, the addition of a new gesture is really easy and can be done in a couple of minutes. This is achievable because to recognize hand gestures, the framework uses only the coordinates of twenty-one landmarks representing the junctions of the hand and a sequence of them in the case of dynamic hand gestures.

The possibility to add new gestures "to the need" opens the framework to several scenarios. To handle all the possible input sequences and related actions for the robot to perform, the framework asks the users to declare a finite state automaton through a configuration file whose structure was made as simple as possible. In this way, anyone can relate a sequence of gestures to actions performed by a robot.

The integration with ROS is just at the beginning. At the moment, users can only set a navigation goal and publish a message on a topic. Nevertheless, as a proof-of-concept of what is possible to achieve with this framework, it has been tested in a warehouse environment. The ASL has been used as static hand gestures to recognize and six gestures taken from the literature have been used as dynamic hand gestures to recognize. Thanks to the framework's integration with Nav2 as the navigation system and ROS' topics, a finite state automaton has been described to achieve several tasks, for example, going to a position, picking up a parcel, and dropping off the parcel. A *Turtlebot v3* has been simulated in the Gazebo simulator, and the

AWS small warehouse was the environment where the simulation has been executed.

In addition, a way to describe macros has been implemented. In this way, users can pre-record a sequence of gestures and run it at another moment or edit it with any text editor. The correctness of the sequence is enforced by the same automaton that enforces the correctness of the users input in the real-time scenario.

Furthermore, several tests have been performed to evaluate the quality of the implementation. Those concerning system resource utilization, in particular, are encouraging in terms of a possible deployment on real hardware such as an NVIDIA Jetson Nano or a Raspberry-Pi, two boards widely used in the robot and automation ecosystem.

The solution proposed is not without limitations and challenges to overcome. In particular, the integration with ROS is just at the beginning. For example, the framework can work only with Nav2 as the navigation system and can only send messages through topics. In future development, the framework should be able to use different navigation systems and exploit all the capabilities provided by ROS to exchange messages. Moreover, the ROS community is active and there are a lot of developers who are creating new ROS packages; the framework should be able to integrate with them as it does with Nav2. This would make it really easy for the community around the development of robotic applications to exploit the capability of the framework when the simplification of the HRI is taken into consideration.

Appendix A

Data gathering

A.1 JSON file complexity

Listing A.1: JavaScript script to compute the complexity of a JSON file

```
const {isNull} = require('util');

fs = require('fs')

let isType = (val, Cls) => val != null && val.constructor ===
    Cls;
let getComplexity = (json, d=1.05) => {

  // Here `d` is our "depth factor"

  return d * (() => {

    // String
    if (isType(json, String)) return 1;

    // Number
    if (isType(json, Number)) return 1;

    // Null values
    if (isNull(json)) return 1;

    // Arrays are 1 + (average complexity of nested elements)
    if (isType(json, Array)) {
      let avg = json.reduce((o, v) => o + getComplexity(v, d)
        , 0) / (json.length || 1);
      return avg + 1;
    }

    // Objects are 1 + (average complexity of their keys) + (
      average complexity of their values)
    if (isType(json, Object)) {
      // `getComplexity` for Arrays will add 1 twice, so
         subtract 1 to compensate
      //return getComplexity(Object.keys(json), d) +
         getComplexity(Object.values(json), d) - 1;
      return getComplexity(Object.values(json));
    }

    throw new Error(`Couldn't get complexity of ${json}`);

  })();

};

fs.readFile(process.argv[2], 'utf8', function(err, data) {
  if (err) {
      return console.log(err);
  }

  jsonFile = JSON.parse(data);
  console.log(getComplexity(jsonFile));
});
```

A.2 System resource

Listing A.2: Python script to collect system resource usage by a process

```
import psutil
from subprocess import DEVNULL, STDOUT
from sys import argv
from time import sleep

import seaborn as sns
from matplotlib import pyplot as plt

def plot_and_save(x, y, path, title, ylabel):
    graph = sns.lineplot(x=x, y=y)

    graph.set(title=title)
    graph.set(xlabel="Time (s)")
    graph.set(ylabel=ylabel)
    graph.get_figure().savefig(path)
    plt.close(graph.get_figure())

cmd = [command for command in argv[1:]]
sleep_time = 5.0
graph_path = "graphs"

times = []
```

```
t = 0.0
cpu_percents = []
ram_percents = []
number_of_cpu = psutil.cpu_count()

with psutil.Popen(cmd) as p:
    print("Starting logging...")

    while p.status() == psutil.STATUS_RUNNING or p.status()
       == psutil.STATUS_SLEEPING:
        times.append(t)
        t += sleep_time

        cpu_percents.append(p.cpu_percent())
        ram_percents.append(p.memory_percent())
        #print(f"CPU: {p.cpu_percent()}%")
        #print(f"RAM: {p.memory_percent()}%")
        sleep(sleep_time)

print("...finished")

if len(cpu_percents) > 0 and len(ram_percents) > 0:
    print(f"Data gathered: {len(times)}")
    plot_and_save(times,
                  [cpu_perc/number_of_cpu for cpu_perc in
                     cpu_percents],
                  f"{graph_path}/cpu.png",
                  "CPU usage",
                  "% of CPU")
    plot_and_save(times,
                  ram_percents,
                  f"{graph_path}/ram.png",
                  "RAM usage",
                  "% of RAM")
```

Appendix B

Configuration file

B.1 Automaton configuration

Listing B.1: JSON configuration file for the automaton

```
{
  "initial_state": "q0",
  "transitions": [
    {
      "from": "q0",
      "to": "q1",
      "with": "pick_up",
      "action": null
    },
    {
      "from": "q0",
      "to": "q2",
      "with": "go_to",
      "action": null
    },
    {
      "from": "q2",
      "to": "q0",
      "with": "A–Z",
      "action": {
        "type": "set_navigation_goal",
        "coordinate": "$with"
      }
    },
    {
      "from": "q1",
      "to": "q3",
      "with": "A–Z",
      "action": {
        "type": "send_message",
        "message": {
          "type": "/control_arm",
          "data": "pick_up $with"
        }
      }
    },
    {
      "from": "q3",
      "to": "q0",
      "with": "drop_down",
      "action": {
        "type": "send_message",
        "message": {
          "type": "/control_arm",
          "data": "drop_down"
        }
      }
    },
    {
      "from": "q3",
      "to": "q4",
      "with": "go_to",
      "action": null
    },
    {
      "from": "q4",
      "to": "q3",
      "with": "A–Z",
      "action": {
        "type": "set_navigation_goal",
        "coordinate": "$with"
      }
    }
  ]
}
```

Glossary

ASL Way to represent each letter of the latin alphabet using only one hand. xiii, 20, 57

DDS Standard by the Object Management Group that aims to enable dependable, high-performance, interoperable, real-time, scalable data exchanges using a publish–subscribe pattern. 57

HCI Studies about the usability of computational devices.. 55, 57

HRI The set of interactions that a human and a robot can have. It starts from Human-Computer Interaction and adds the robot component to the equation. 57

IL The Intuitiveness Level of each gesture is defined as the arithmetic mean of the normalized occurrence rates. Namely, the general occurrence rate, the volunteer occurrence rate, and the occurrence rate by time. 57

LSTM Recurrent neural network within you try to make the network remember what happened in the past. 57

Macro A saved sequence of actions that can be executed in a very simple way. 2, 55

ROS The Robot Operating System (ROS) is a set of software libraries and tools that help you build robot applications [15]. 57

SDF SDFormat (Simulation Description Format), sometimes abbreviated as SDF, is an XML format that describes objects and environments for robot simulators, visualization, and control. Originally developed as part of the Gazebo robot simulator, SDFormat was designed with scientific robot applications in mind. 57

YOLO An object detection algorithm of the type regression-based. Firstly introduced by Redmon et al.[14] in 2015 and then deeply studied for its characteristics. 57

Acronyms

ASL American Sign Language. xiii, 19, 20, 22, 29, 43, 47, 55

CNN Convolutional Neural Network. 8

DDS Data Distribution Service. 9, 10, 55

HCI Human-Computer Interaction. 5, 55

HRI Human-Robot Interaction. 1, 2, 3, 48, 55

IL Intuitiveness Level. 6, 55

LSTM Long Short-Term Memory. xiii, 9, 22, 23, 44, 55

ML Machine Learning. 1, 7, 8, 14, 43

NLP Natural Language Processing. 5

ROI Region Of Interest. 8, 9

ROS Robot Operating System. 1, 2, 3, 9, 10, 13, 15, 16, 17, 25, 29, 37, 38, 40, 42, 44, 45, 47, 48, 55

SDF Simulation Description Format. 16, 55

YOLO You Only Look Once. 9, 55

Bibliography

[1] Robert Bogue. “Growth in e-commerce boosts innovation in the warehouse robot market”. In: *Industrial Robot: An International Journal* (2016) (cit. on p. 2).

[2] Clebeson Canuto, Eduardo O Freire, Lucas Molina, Elyson AN Carvalho, and Sidney N Givigi. “Intuitiveness Level: Frustration-Based Methodology for Human-Robot Interaction Gesture Elicitation”. In: *IEEE Access* (2022) (cit. on p. 6).

[3] Bo Chen, Chunsheng Hua, Bo Dai, Yuqing He, and Jianda Han. “Online control programming algorithm for human–robot interaction system with a novel real-time human gesture recognition method”. In: *International Journal of Advanced Robotic Systems* 16.4 (2019), p. 1729881419861764 (cit. on p. 3).

[4] Ds13. *ASL alphabet*. URL: https://commons.wikimedia.org/w/index.php?curid=79729279 (cit. on p. 20).

[5] Tatsuya Fujii, Jae Hoon Lee, and Shingo Okamoto. “Gesture recognition system for human-robot interaction and its application to robotic service task”. In: 1 (2014) (cit. on p. 5).

[6] Cheng Guo and Ehud Sharlin. “Exploring the use of tangible user interfaces for human-robot interaction: a comparative study”. In: (2008), pp. 121–130 (cit. on p. 3).

[7] Wittawin Kahuttanaseth, Alexander Dressler, and Chayakorn Netramai. “Commanding mobile robot movement based on natural language processing with RNN encoderdecoder”. In: (2018), pp. 161–166. DOI: 10.1109/ICBIR.2018.8391185 (cit. on p. 5).

[8] G Drew Kessler, Larry F Hodges, and Neff Walker. “Evaluation of the CyberGlove as a whole-hand input device”. In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 2.4 (1995), pp. 263–283 (cit. on p. 3).

[9] Seong-Whan Lee. “Automatic gesture recognition for intelligent human-robot interaction”. In: (2006), pp. 645–650. DOI: 10.1109/FGR.2006.25 (cit. on p. 5).

[10] Steve Macenski, Francisco Martín, Ruffin White, and Jonatan Ginés Clavero. “The Marathon 2: A Navigation System”. In: (2020). URL: https://github.com/ros-planning/navigation2 (cit. on p. 15).

[11] *MediaPipe website*. URL: https://mediapipe.dev/ (cit. on p. 15).

[12] Christopher Olah. *Understanding LSTM Networks*. URL: http://colah.github.io/posts/2015-08-Understanding-LSTMs/ (cit. on p. 9).

[13] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. "ROS: an open-source Robot Operating System". In: 3.3.2 (2009), p. 5 (cit. on pp. 9, 10).

[14] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. "You Only Look Once: Unified, Real-Time Object Detection". In: *CoRR* abs/1506.02640 (2015). arXiv: `1506.02640`. URL: `http://arxiv.org/abs/1506.02640` (cit. on p. 55).

[15] *ROS website*. URL: `https://www.ros.org` (cit. on p. 55).

[16] Vaidyanath Areyur Shanthakumar, Chao Peng, Jeffrey Hansberger, Lizhou Cao, Sarah Meacham, and Victoria Blakely. "Design and evaluation of a hand gesture recognition approach for real-time interactions". In: *Multimedia Tools and Applications* 79.25 (2020), pp. 17707–17730 (cit. on pp. 3, 5, 6).

[17] Kazuhito Takahashi. *Hand gesture recognition using mediapipe*. URL: `https://github.com/Kazuhito00/hand-gesture-recognition-using-mediapipe` (cit. on pp. 9, 16).

[18] Taiqian Wang, Yande Li, Junfeng Hu, Aamir Khan, Li Liu, Caihong Li, Ammarah Hashmi, and Mengyuan Ran. "A Survey on Vision-Based Hand Gesture Recognition". In: (2018). Ed. by Anup Basu and Stefano Berretti, pp. 219–231 (cit. on p. 8).

[19] William Woodall. *ROS on DDS*. URL: `https://design.ros2.org/articles/ros_on_dds.html` (cit. on p. 10).

[20] Fan Zhang, Valentin Bazarevsky, Andrey Vakunov, Andrei Tkachenka, George Sung, Chuo-Ling Chang, and Matthias Grundmann. "MediaPipe Hands: On-device Real-time Hand Tracking". In: *CoRR* abs/2006.10214 (2020). arXiv: `2006.10214`. URL: `https://arxiv.org/abs/2006.10214` (cit. on p. 15).

[21] Thomas G. Zimmerman, Jaron Lanier, Chuck Blanchard, Steve Bryson, and Young Harvill. "A Hand Gesture Interface Device". In: *SIGCHI Bull.* 18.4 (May 1986), pp. 189–192. ISSN: 0736-6906. DOI: `10.1145/1165387.275628`. URL: `https://doi.org/10.1145/1165387.275628` (cit. on p. 6).