BO20-G38

MANEUVER AUTONOMOUS VEHICLES WITH ARM GESTURES

RALF S. SJØVOLD LEISTAD
SANDER LADE HELLESØ
SVERRE STAFSENGEN BROEN
WILLIAM SVEA-LOCHERT

BACHELOR THESIS 2020

ØSTFOLD UNIVERSITY COLLEGE
DEPARTMENT OF INFORMATION TECHNOLOGY

HØGSKOLEN I ØSTFOLD

Avdeling for Informasjonsteknologi
Remmen
1757 Halden
Telefon: 69 21 50 00
URL: www.hiof.no

BACHELOROPPGAVE

Prosjektkategori: Utvikling	X	Fritt tilgjengelig
Omfgang i studiepoeng: 20	(30/12 2029)	Fritt tilgjengelig etter
Fagområde: Avdeling for Informasjonsteknologi	(X)	Tilgjengelig etter avtale med oppdragsgiver

Tittel: Maneuver autonomous vehicles with arm gestures	Dato: May 28, 2020
Forfatterere: Ralf Sebastian Sjøvold Leistad, Sander Lade Hellesø, Sverre Stafsengen Broen, William Svea-Lochert	Veileder: Roland Olsson
Avdeling / Program: Avdeling for Informasjonsteknologi - Bachelorstudium i informatikk - design og utvikling av IT-systemer	Gruppenummer: BO20-G38
Oppdragsgiver: Forsvarets Forskningsinstitutt	Kontaktperson hos oppdragsgiver: Idar Dyrdal

Ekstrakt:
Autonomous vehicles need to be reliable if they are going to be used in a safe and efficient manner. Forsvarets forskningsinstitutt (FFI) is currently exploring the feasibility of developing a system that allows autonomous vehicles to be controlled with arm gestures. This could have a great impact on everyday life within the military as it would save time and resources. This thesis addresses topics and technologies used to lay the foundation, for which a fully-fledged system can be based on.
Developing this system required a significant amount of research on machine learning, robotics, and related frameworks and technologies. The proof of concept developed for this project serves as a prototype, and should preferably utilize the camera and robotics technology already found in a TurtleBot3 Waffle Pi. The TurtleBot was provided by FFI and mimics a tracked vehicle.
To better understand how such a system could be developed, an examination of several similar solutions were conducted. After evaluating the research, the process of manufacturing data started, while simultaneously developing a machine learning algorithm. In collaboration with FFI, a decision was made concerning the requirements for what gestures should be used. A conclusion was drawn to produce data using data augmentation, as no open-source datasets were available for the given requirements.
Image processing is a central part of this thesis. The group discovered promising solutions to create and label a substantial amount of data within a short time period. Transmitting images between two devices was inevitable, thus, finding a solution to do this efficiently was essential.
There were no instructions on which machine learning algorithm to use. The decision was based on research and experimentation with different neural network architectures, before concluding the development as a working convolutional neural network (CNN) was in place. The classification algorithm reached 99,9% accuracy during testing, which proved that our CNN model could confidently recognize and classify arm gestures. The model was integrated with the TurtleBot, that we had to develop custom functionality for. This integrated system culminates in the development of our prototype.
The research done throughout the project period, along with the prototype, confirms that a system for maneuvering autonomous vehicles with arm gestures is a realistic development opportunity. This thesis can be used as a point of reference for developing a fully operational and reliable system.

3 emneord:

Autonomous vehicles
Image processing
Machine learning

Abstract

Autonomous vehicles need to be reliable if they are going to be used in a safe and efficient manner. Forsvarets forskningsinstitutt (FFI) is currently exploring the feasibility of developing a system that allows autonomous vehicles to be controlled with arm gestures. This could have a great impact on everyday life within the military as it would save time and resources. This thesis addresses topics and technologies used to lay the foundation, for which a fully-fledged system can be based on.

Developing this system required a significant amount of research on machine learning, robotics, and related frameworks and technologies. The proof of concept developed for this project serves as a prototype, and should preferably utilize the camera and robotics technology already found in a TurtleBot3 Waffle Pi. The TurtleBot was provided by FFI and it mimics a tracked vehicle.

To better understand how such a system could be developed, an examination of several similar solutions were conducted. After evaluating the research, the process of manufacturing data started, while simultaneously developing a machine learning algorithm. In collaboration with FFI, a decision was made concerning the requirements for what gestures should be used. A conclusion was drawn to produce data using data augmentation, as no open-source datasets were available for the given requirements.

Image processing is a central part of this thesis. The group discovered promising solutions to create and label a substantial amount of data within a short time period. Transmitting images between two devices was inevitable, thus, finding a solution to do this efficiently was essential.

There were no instructions on which machine learning algorithm to use. The decision was based on research and experimentation with different neural network architectures, before concluding the development as a working convolutional neural network (CNN) was in place. The classification algorithm reached 99,9% accuracy during testing, which proved that our CNN model could confidently recognize and classify arm gestures. The model was integrated with the TurtleBot, that we had to develop custom functionality for. This integrated system culminates in the development of our prototype.

The research done throughout the project period, along with the prototype, confirms that a system for maneuvering autonomous vehicles with arm gestures is a realistic development opportunity. This thesis can be used as a point of reference for developing a fully operational and reliable system.

Acknowledgements

We would like to thank Roland Olsson, our thesis advisor for his technical expertise and guidance. We would also like to thank Idar Dyrdal and Kim Mathiassen, our contacts and advisors at FFI for their invaluable assistance during this period and allowing us to work on such an interesting project. Finally, we want to express our gratitude to Allan Lochert for his continuous guidance throughout this project, and helping us get this assignment. Additionally, we would like to thank Fredrik Lauritzen and Simon Myhre for helping us gather data by volunteering as subjects in our video capturing process.

Contents

List of Figures

List of Tables

Code list

Chapter 1

Introduction

In a rapidly modernizing society where the importance of technology and resource management is increasing there is a need for smarter systems. Utilizing automation to reduce manpower and increase efficiency is something several organizations and corporations could benefit from[1]. The boundaries for what is possible to achieve with automation is constantly growing. Taking advantage of the numerous benefits of autonomous systems is important, not only for civilian corporations but also for government agencies such as the military. Forsvarets forskningsinstitutt (FFI) has already developed systems for autonomous vehicles that could potentially benefit the military as well as society[2].

However, this project is primarily aimed at the military and how utilizing machine learning can increase the efficiency within a team of soldiers. FFI wants to explore the possibility of relieving a soldier of their duty of using a remote control to maneuver a vehicle. Achieving this would allow the soldier to be more involved with the task at hand, rather than being occupied with the vehicle. FFI is looking for a proof of concept to show that such a system is a viable development opportunity. The research and results formed during the development of a proof of concept can be used as guidance for how an operational system can be developed.

FFI wants the group to create a system acting as a proof of concept, that will allow for maneuvering autonomous vehicles with arm gestures. To do this there has to be done evaluations and analysis on different machine learning techniques, frameworks, data gathering techniques, and implementation of a prototype.

[1] https://en.wikipedia.org/wiki/Automation#Advantages_and_disadvantages
[2] https://www.ffi.no/publikasjoner/arkiv/den-autonome-framtid

Project Group

The group is made up of four 3rd year computer science students, studying at Østfold University College (HIOF), Halden. The group members started working together when they attended California State University Monterey Bay (CSUMB) while doing a student exchange program. After the stay in California, the group continued to collaborate.

Ralf Sebastian Sjøvold Leistad leistad.ralf@gmail.com
Ralf Leistad is 24 years old and currently living at Remmen, Halden. He moved to Halden from Malvik, Trøndelag. Previously he has experience as an infantry soldier in the Norwegian Armed Forces, and from sales and customer support. He has gained a lot of technological and programming knowledge through his studies at HIOF, where he now works as a teaching assistant (TA). His second year at HIOF he spent as an exchange student at CSUMB where he actively participated in the programming team.

Sander Lade Hellesø sanderlade@hotmail.com
Sander Lade Hellesø is 24 years old and comes from Ålesund. Currently living on the campus at HIOF, department Halden. He attended CSUMB as an exchange student for the second year of his degree. During his stay in California, he got introduced to competitive programming and participated in the programming team, while also helping a professor teach students about open source programming. Sander is also working part-time as a Software Developer for Sportradar in Oslo, where he will continue to work full-time after the degree.

Sverre Stafsengen Broen sverre.sb@gmail.com
Sverre Stafsengen Broen is 25 years old and comes from Fredrikstad. He is currently living on campus at HIOF, department Halden. Previously he has taken preparatory courses in math, physics, and chemistry at HIOF, department Fredrikstad. During his time at HIOF, he also attended CSUMB as an exchange student. Sverre was introduced to competitive programming through CSUMB's programming team. He is currently working as a TA in Algorithms and Data structures, previously he was TA in Operating Systems. His previous work experience is working in a convenience store, call center, and psychiatric housing.

William Svea-Lochert dmuwilliamsl@gmail.com
William Svea-Lochert is 22 years old and comes from Drøbak. He now lives at Remmen in Halden at the student apartments. Previously he has completed a one-year study at Nord Universitet in Bodø studying math, chemistry, biology and, logistics. He has also completed 2 years with media courses and one year of additional courses at high school. While studying at HIOF he has also attended CSUMB as an exchange student. William has previous experience working in bars and as a construction worker.

Employer

FFI was established in 1946 when there was a common consensus that technology played a big and important role in the outcome of the 2nd world war. The Norwegian Armed Forces had to be rebuilt with modernized weaponry[3]. The new institute was given three main goals, contributing to modernize; the Norwegian Armed Forces, the industry, and the scientific fields in Norway. Three technological areas were chosen as the most central- electronics, rocket technology, and atomic energy.

FFI is the official research institution for the Norwegian Defence Department. The institution has high-tech expertise, along with military and political insight. They work closely with the Norwegian Armed Forces, the Norwegian Defence Department, in addition to other political sectors, national and international research institutions, and several other countries through partnerships.

Some of the work they do is aiding the Norwegian Armed Forces with problems ranging from operational capability to long-term planning and supplying the Norwegian Defence Department with the appropriate knowledge to face current and future challenges.

Idar Dyrdal

Idar graduated from the University of Oslo in 1978 with physics as his main topic. After finishing military service, in 1980 he was employed by FFI and later took a one-year scholarship in the USA.

Over the years he has mainly been working on image analysis, signal processing, and pattern recognition (traditional machine learning). Idar has also worked with deep learning for different defense-related uses. He has also worked a lot with signal processing with machine learning for acoustic sensor systems.

In later years Idar has been working with machine vision for steering autonomous vehicles. In conjunction with working at FFI, he has a part-time job at the University of Oslo - Institute for technology systems, where he holds lectures and supervises bachelor and master students.

Kim Mathiassen

Kim Mathiassen received a Master's degree in engineering cybernetics from the Norwegian University of Science and Technology in 2010 and a Ph.D. in robotics from the University of Oslo in 2017. His Ph.D. thesis was on a semi-autonomous system for diagnostics and treatment using medical ultrasound. Before his studies, he attended the Norwegian Army Officer Candidate School for one year and served one year as a sergeant in the Norwegian Army Signals Battalion.

In 2015 he started working at FFI with Unmanned Ground Vehicles (UGV) and autonomous systems. In addition to his research position, he is an associate professor at the University of Oslo teaching advanced robotics. His current research interests span from low-level robot control to high-level planning and reasoning.

[3] https://www.ffi.no/om-ffi

1.1 Assignment

FFI is currently looking into the possibility to use machine learning, image processing, and object recognition to maneuver vehicles. Their long-term goal is to have autonomous vehicles for transporting cargo, when necessary, this should be possible to override to enable maneuvering the vehicle by using arm gestures.

For this project, a prototype shall be developed by the group, the main part is to develop software that can handle video streams and respond with the appropriate action. What programming languages, frameworks, and libraries to use are not defined. The group will conduct research to find appropriate tools for developing the software. What is most important is good support and documentation for machine learning and processing video streams in the technology the group decides to use. The prototype will be tested on a TurtleBot3 Waffle Pi (TurtleBot). The TurtleBot will record movement, make requests to the software using image frames from a video stream, and the software will respond with an action that the robot will execute.

1.2 Why, What and How: Purpose, Deliveries and Method

1.2.1 Purpose

The purpose of this project is to establish a proof of concept. As mentioned in the assignment, FFI is looking into the possibility of maneuvering vehicles with arm gestures. To achieve this we are developing a prototype using a TurtleBot and building a neural network corresponding with the defined minimal viable product. The thesis will contain reasoning for frameworks and technologies chosen, this will be done by researching and testing the different technologies.

Main goal Establish a proof of concept by laying the groundwork towards a possible solution for the problem described.

- **Intermediate objective 1** Research whether there are existing solutions for similar gesture recognition.
- **Intermediate objective 2** Develop a functioning and accurate model.
- **Intermediate objective 3** Develop functionality for the TurtleBot.
- **Intermediate objective 4** Connect the neural network and the TurtleBot.

1.2.2 Method

Bachelor thesis

We will approach this report by introducing the problem presented by FFI, which will be the basis for which sources, articles, and technologies we decide to examine. We will continue to analyze and summarize some of our findings to build a better understanding of how we can develop a prototype/concept proof. Then give a walk-through of our approach and methodology for solving the problem at hand. Finally, we will form a conclusion based on our progress and knowledge regarding the system.

Prototype

The entire process of developing a prototype for this project is going to start by reading research papers, articles, and documentation based on problems similar to ours.

Then we will start manufacturing datasets that will later be used to train and test the neural network model. During the process of creating datasets, we will start developing the actual model. We aim to start this early to account for any problems along the way.

However, since there are four people working on this project we are able to work on multiple parts simultaneously. We aim to start setting up and developing custom functionality for the TurtleBot while also working on the model. Once both parts are done we are going to integrate the model and the TurtleBot.

Throughout this process, each part of the prototype will be continuously tested. This includes testing each individual component of the prototype and the entire prototype as a whole.

1.3 Report Structure

Chapter 2 will present the different technologies and concepts related to the assignment, covering both the theory and its usage.

Chapter 3 begins with a more detailed description of the project, then proceeds to examine related technologies. The chapter will also cover related works and give a brief summary of how similar problems have been solved previously. Chapter 3 ends with relevant use cases for a system as described in the thesis.

Chapter 4 covers the planning phase of the project. It starts with the initial plans for the project for the data and test -sets, the machine learning model, development of the TurtleBot, and the integration of the model with the TurtleBot. The chapter concludes with how we picked the right frameworks to use, as well as problems and solutions faced along the way.

Chapter 5 covers how the datasets, testsets, and models were made, in addition to the development of the TurtleBot and integrating the model with the TurtleBot. The chapter will also cover a comparison of how all the different machine learning models performed in training, validation, testing.

Chapter 6 covers the testing and evaluation of the results from chapter 5. It will cover the initial tests and their results, then proceed to cover reliability testing of the system. The developed system will be evaluated, and suggestions for potential improvements for a future system will be discussed.

Chapter 7 contains a discussion of the implementations and further evaluate the suggested improvements from Chapter 6.

Chapter 8 concludes the project, the group will come with a conclusion of the project.

Chapter 2

Background

2.1 Machine Learning

Machine learning (ML) is a field of study that uses algorithms and statistical models, allowing computer systems to perform certain tasks based on patterns and intuition rather than explicit instructions[1]. A machine learning algorithm builds a model by iterating through training data. A model can be described as a mathematical representation of a real-world process. This mathematical representation is fed data and trained to recognize certain patterns. When properly trained, a model can be used to evaluate data it has never seen before[1]. There are different types of models that have been researched and used in machine learning systems[2]. Which type of model you choose depends entirely on the problem you aim to solve.

Additionally, there are several different ways of training a model, which approach chosen depends on the input/output data and the specific task/problem that the resulting model should solve[3]. Some of these methods will be briefly introduced in the following Subsections 2.1.1, 2.1.2 and 2.1.3.

2.1.1 Supervised learning

When using a supervised learning algorithm to train a machine learning model one would use training data that contain both the inputs and the desired outputs[9]. Since the desired output is known to the model it can correct itself and how it makes decisions.

2.1.2 Unsupervised learning

Unsupervised learning algorithms use training data that only contain uncategorized/unlabeled input. Such an algorithm requires the model to learn from commonalities rather than direct feedback[4].

[1] https://docs.microsoft.com/en-us/windows/ai/windows-ml/what-is-a-machine-learning-model

[2] https://en.wikipedia.org/wiki/Machine_learning#Models

[3] https://en.wikipedia.org/wiki/Machine_learning

[4] https://en.wikipedia.org/wiki/Machine_learning#Unsupervised_learning

2.1.3 Feature learning

Feature learning often uses pre-processing of the data to discover better representations of the given input[10]. This allows the algorithm to extract features from the input while also trying to preserve the information relevant for the given task. Such an algorithm can be both supervised and unsupervised.

2.2 Neural networks

Neural networks (NN), or more specifically artificial neural networks (ANN) are inspired by the biological neural networks found in animals brains[2]. A neural network consists of artificial neurons, a neuron can process a signal from another neuron before signaling other neurons connected to it. The signal between connected neurons (edge) is a real number, however, the output for each neuron is some sum of the inputs calculated by a chosen non–linear function, also called an activation function[5]. Typically, in a neural network, neurons are aggregated into different layers. Signals travel from the network's input layer (first layer) through its hidden layers before getting to the output layer (last layer). Traversing the hidden layers can be done multiple times depending on the design of the network.

Connections between neurons in the different layers can be structured differently depending on the architecture of the network. Every neuron can be connected to every neuron in the next layer (fully connected) Figure 2.1, or a group of neurons can be connected to a single neuron in the next layer (pooling, this is described in more detail in Section 3.2.2).

Neurons and edges usually have an associated weight[6]. These weights will be adjusted as the network is learning, this is done to increase the accuracy of the result. This can be achieved by taking measures to reduce the error rate and get it as close to 0 as possible.

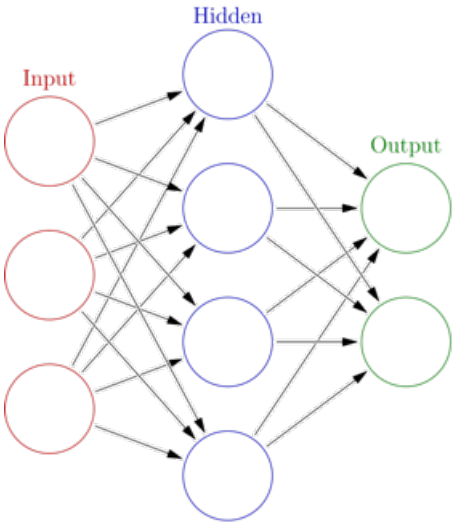


Figure 2.1: A simple illustration of a fully connected neural network.
Obtained from https://en.wikipedia.org/wiki/Artificial_neural_network

[5] https://en.wikipedia.org/wiki/Artificial_neural_network
[6] https://en.wikipedia.org/wiki/Artificial_neural_network

2.2.1 Overfitting

Normally a machine learning model or algorithm is trained on some data where the output is known. When a model is trained the goal is for it to perform well when predicting validation data, this is data that the model has never seen before in training.

Overfitting occurs when a machine learning model or algorithm begins to memorize the training data, rather than learning from it and generalizing the data [7].

2.2.2 Underfitting

Underfitting in machine learning is something that happens when a machine learning algorithm can't sufficiently record the fundamental features in the training data. Underfitting happens when the model or algorithm does not fit the data sufficiently[8].

2.3 Frameworks

This section will give a brief overview of some available frameworks that we as a group deem viable to utilize in this project. In Chapter 3.3 we will discuss the chosen frameworks for this project.

2.3.1 PyTorch

The open-source machine learning library PyTorch is based on the ML library Torch[9]. PyTorch is used for applications like computer vision and natural language processing. The library is mainly developed by Facebook's AI Research lab. PyTorch is free, open-source software released under the Modified BSD license. PyTorch's Python interface is considered to be more refined than the C++ interface, thus making Python's interface the primary focus for improving the library.

2.3.2 TensorFlow

TensorFlow is a free, open-source software library. The library has multiple use-cases like dataflow, differentiable programming, symbolic math, and machine learning[10]. TensorFlow was developed by the Google Brain team for internal use at Google. On November 9th 2015 TensorFlow was publicly released under Apache License 2.0.

TensorFlow is Google Brain's 2nd generation system. TensorFlow can run on multiple CPUs and GPUs. The library is available for 64-bit Linux, macOS, Windows, and mobile platforms such as Android and iOS.

[7] https://en.wikipedia.org/wiki/Overfitting
[8] https://en.wikipedia.org/wiki/Overfitting#Underfitting
[9] https://en.wikipedia.org/wiki/PyTorch
[10] https://en.wikipedia.org/wiki/TensorFlow

2.3.3 PlaidML

PlaidML is an advanced and portable, open-source tensor compiler[11]. It was created to enable machine learning on platforms such as laptops, embedded devices, and devices installed with poorly supported hardware. PlaidML works underneath other more common machine learning frameworks like Keras, ONNX, and nGraph. The tensor compiler enables developers to use a wider range of hardware than frameworks like TensorFlow (2.3.2). When PlaidML is used as a component under Keras (2.3.4) it can help accelerate training workloads. It works especially well on GPUs, and there is no requirement of using CUDA supported hardware.

August 2018 Intel acquired Vertex.ai, the original creators of PlaidML. Vertex.ai was a startup that wanted to make deep learning accessible on every platform. PlaidML was released as free software under Apache License 2.0 with a goal to improve compatibility with nGraph, TensorFlow, and other frameworks. The software is available on 64-bit Linux, macOS, and Windows[12].

2.3.4 Keras

Keras is an open-source neural network library written in Python. It was designed with the developer in focus for enabling fast experimentation and to be able to go from an idea to a result as fast as possible.

With Keras one can develop machine learning models with a variety of different deep learning backends, this means that we are able to train the model using one backend and load the model with a different backend. Some of the frameworks Keras is able to run on top of is TensorFlow, Theano, and PlaidML[13].

2.3.5 OpenCV

OpenCV (Open source computer vision) is a library that contains functions for computer vision. The library was originally developed by Intel and was later supported by Willow Garage and Itseez.

The library is cross-platform, free to use, and currently under the open-source BSD license. Robot Operating System (ROS) uses OpenCV as its primary vision package[14].

2.3.6 matplotlib

Matplotlib is a library made for plotting with Python and NumPy[15]. Matplotlib was released in 2003 and was originally written by John D. Hunter. The library is released under a BSD-style license.

2.3.7 NumPy

NumPy is an open-source, numerical mathematics extension for Python[16]. The library supports high-level mathematical functions for multi-dimensional arrays (ndarray), matrices, and large collections.

[11] https://plaidml.github.io/plaidml/
[12] https://en.wikipedia.org/wiki/PlaidML
[13] https://keras.io/why-use-keras/
[14] https://en.wikipedia.org/wiki/OpenCV
[15] https://en.wikipedia.org/wiki/Matplotlib/
[16] https://en.wikipedia.org/wiki/NumPy

2.4 TurtleBot3 Waffle Pi

TurtleBot is a robotic vehicle that was released in November 2010. The research lab that created this product was Willow Garage, with lead developers Melonee Wise and Tully Footee[17]. Willow Garage offers hardware and open source software for personal robotics applications. Their vision is to enable robotics applications for personal usage in everyday life, thus creating the Robot Operating System (ROS)[18].

This section will contain information about the TurtleBot's physical components in addition to the operating systems. The section provides an overview of the specifications for the individual components, as well as giving an insight into how these features are valuable in robotics applications. See Figure 2.2 for an illustration of the TurtleBot3 Waffle Pi.

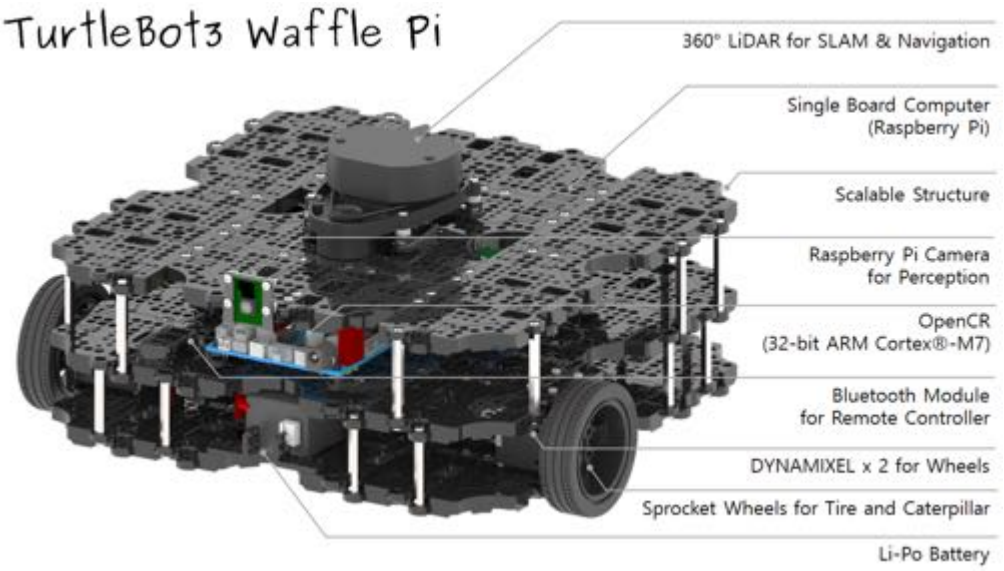


Figure 2.2: Illustration of the components in the TurtleBot3 Waffle Pi.
Obtained from http://emanual.robotis.com/docs/en/platform/turtlebot3/specifications

2.4.1 Raspberry Pi 3 Model B+

A Raspberry Pi handles the interface for the TurtleBot. Raspberry Pi is a single-board computer that runs on the Rasbian operating system. The computer was founded by the Raspberry Pi Foundation[19]. They are a charity that builds affordable computers to make the power of computing accessible to more people. Their first computer was released in 2012. Since then four Raspberry Pi versions have been released with a combined model count of 13[20]. Raspberry Pi 3 Model B+ is the 10th model and was released in 2018, making it relatively new and updated with useful specifications for robotics applications.

[17] https://www.turtlebot.com/about/
[18] http://www.willowgarage.com/pages/about-us
[19] https://www.raspberrypi.org/about/
[20] https://en.wikipedia.org/wiki/Raspberry_Pi

This model includes a 64-bit 1.4GHz CPU[21], this is enough processing power for some automation in the developed prototype. It has a 1GB LPDDR2 SDRAM. The wireless LAN allows for both communication through frequency 2.4GHz and 5GHz, it has a LAN port as well as Ethernet through USB 2.0. A CSI (Camera Serial Interface) camera port is used for connecting a Raspberry Pi camera module to the Raspberry Pi, which enables the TurtleBot to capture images and record videos. It also has 4 USB-ports that is used to connect to the OpenCR1.0 component[22], and a 360° LiDAR for SLAM and navigation. Peripherals such as keyboard and mouse can be connected to navigating the interface and developing functionalities for the prototype. It also has an HDMI-port that is used for displaying the interface on the Raspberry Pi.

2.4.2 OpenCR1.0

The OpenCR1.0 is an open-source control module that is developed for ROS embedded systems[23]. The processor is a 32-Bit MCU (Multipoint control unit), it has high performance and good responsiveness, which makes it powerful for automotive applications, image processing, sensor fusion, and motor control[24]. The RS485 communication port is used for connecting to the wheels. The control module is connected to a battery and distributes power to the Raspberry Pi.

2.4.3 Raspberry Pi camera

The Raspberry Pi camera module v2 is a small PCB (Printed circuit board) containing a lens, the PCB is connected to the Raspberry Pi through the CSI using a ribbon cable. The 8-megapixel camera sensor allows for high-definition video recordings up to 1080p with 30fps and 720p with 60fps.

2.4.4 LiDAR sensor

A LiDAR (Light and radar) sensor is used for measuring the distance to objects[25]. It uses a laser to illuminate a target, and the reflection is measured with a sensor. This allows for the mapping of the environment. The sensor on top of the TurtleBot is a 360° 2D laser scanner[26]. It collects data by scanning the surroundings of the TurtleBot, then using the data to create a two-dimensional map of objects within reach of the sensor. The sensor can detect the surroundings between 12 centimeters and 3.5 meters, with a 5% loss in accuracy and a 3.5% loss in precision on longer distances.

2.4.5 Operating systems

The TurtleBot is using two operating systems to operate as a robotic vehicle. Rasbian is the main operating system on the TurtleBot, this is the interface that is used for developing functionality for the robot. ROS is used for controlling the robot operations.

Rasbian is not the only operating system that could run on the Raspberry Pi, but it is the recommended one since it is developed specifically for the single board computer[27]. Rasbian is

[21] https://static.raspberrypi.org/files/product-briefs/Raspberry-Pi-Model-Bplus-Product-Brief.pdf
[22] http://emanual.robotis.com/docs/en/platform/turtlebot3/specifications/
[23] http://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_opencr1_0/
[24] https://developer.arm.com/ip-products/processors/cortex-m/cortex-m7
[25] https://en.wikipedia.org/wiki/Lidar
[26] http://emanual.robotis.com/docs/en/platform/turtlebot3/appendix_lds_01/
[27] https://www.raspberrypi.org/documentation/raspbian/

based on Debian which is an open-source Linux distribution. The operating system was mainly developed by Mike Thompson and Peter Green, they also rely on the input and development from the Raspberry Pi community[28].

ROS is a framework that was built to simplify the development of complex tasks. ROS was built from the ground up using open source. The following section will explore ROS further.

2.5 Robot Operating System

ROS is a framework for writing software for robotics in a flexible manner, mainly created by Open Robotics[29]. It is a collection consisting of libraries and tools (Figure 2.3), that will help to simplify the development of more complex robotics behavior. The reason for the development of ROS was mainly because of the lack of a general-purpose robot software that is easy to use. ROS is relevant for the project since the tracked vehicle FFI is already utilizing is running on ROS.

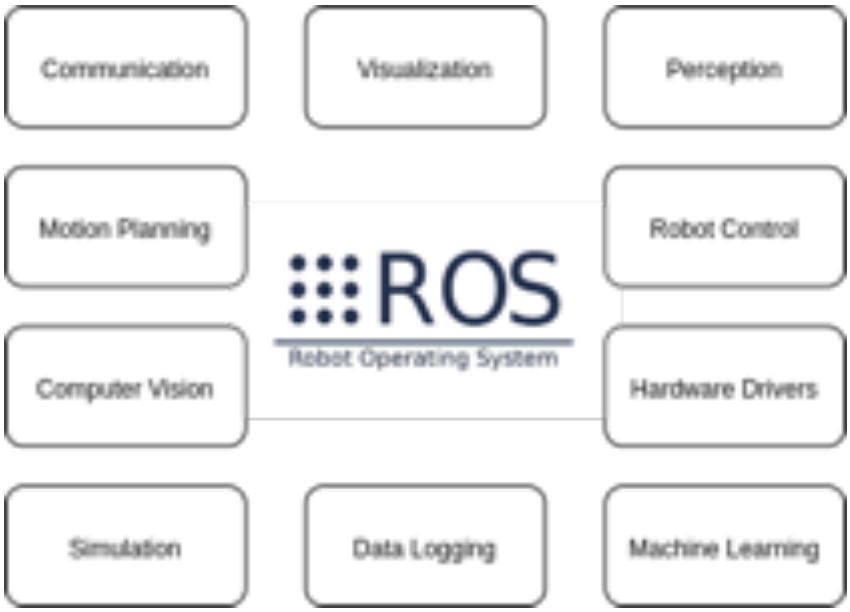


Figure 2.3: Illustration of the features ROS offers.
Obtained from https://maker.pro/ros/tutorial/robot-operating-system-2-ros-2-introduction-and-getting-started

[28] https://www.raspbian.org/RaspbianAbout
[29] https://www.ros.org/about-ros/

2.5.1 ROS as a collaborative tool

ROS was built from scratch to encourage software development for robotics. From the beginning, ROS was developed at multiple institutions and for multiple robots, including many institutions that received PR2 robots from Willow Garage. Although it would have been far simpler for all contributors to place their code on the same servers, over the years, the ”federated” model has emerged as one of the great strengths of the ROS ecosystem.

Any group can start their own ROS code repository on their own servers, and they maintain full ownership and control of it. They don’t need anyone’s permission. If they choose to make their repository publicly available, they can receive the recognition and credit they deserve for their achievements, and benefit from specific technical feedback and improvements like all open-source software projects.

As there are multiple laboratories and experts within the robotics field, working on different technologies and concepts, ROS will allow these different groups to collaborate and build upon each other’s work, The ROS ecosystem now consists of tens of thousands of users worldwide, working in domains ranging from tabletop hobby projects to large industrial automation systems.

Chapter 3

Analysis

This chapter begins with a detailed walk-through of the project described in Section 3.1. Section 3.2 gives an overview of some related technologies to get a basic understanding of how machine learning works, in addition to analyzing core concepts and frameworks that are used in this project. Section 3.3 will review some of the functionalities and tools for the frameworks used in this project. In Section 3.4 we give an overview of ROS. Section 3.5 examines related works. To finish the chapter, Section 3.6 highlights some use cases of a system that recognizes gestures for maneuvering a vehicle.

3.1 Project

This section gives an in-depth description of the project and the prototype that will be developed. Along with a more thorough description of the project, we also address whether or not we are going to take advantage of surveys or focus groups.

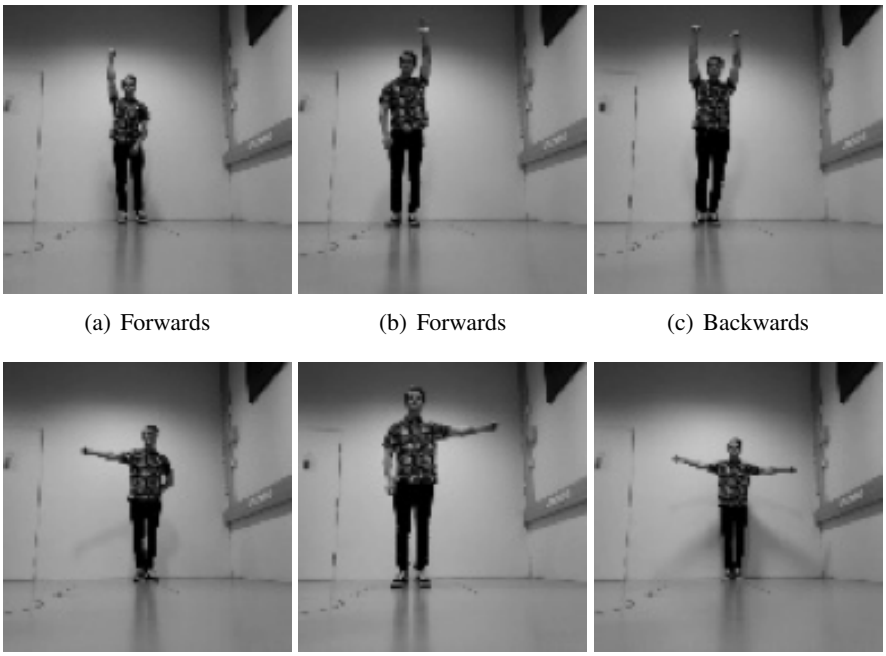
3.1.1 An in-depth description

FFI has been working with autonomous vehicles for some time. In this context, they want a proof of concept regarding their project of letting humans maneuver autonomous vehicles by using arm gestures. To accomplish this FFI would like to utilize machine learning. The proof of concept prototype will potentially serve as a base for further development.

FFI's vision at the moment is to use this technology on a tracked vehicle used for equipment transportation (Figure 3.1). This vehicle is currently being controlled by a soldier using a remote control. By using a remote control this soldier is bound to the task of controlling the vehicle, thus the person can no longer be an active part of the team. This is the problem FFI aims to solve by integrating a system for maneuvering the vehicle autonomously.

Figure 3.1: The tracked vehicle (UGV) FFI wants to use together with software for gesture recognition, image captured by: FFI/Espen Hofoss

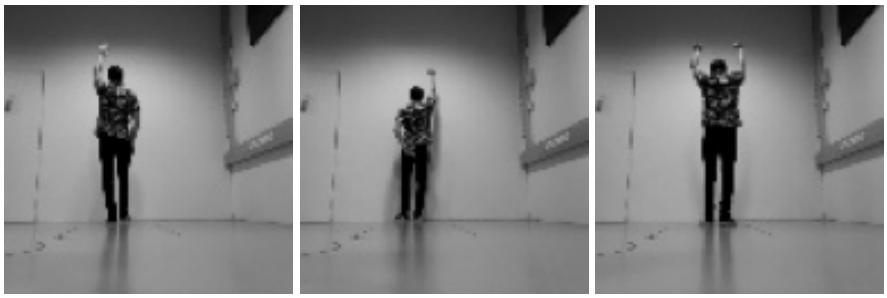
There are five classes or gestures that the group along with FFI have decided to use. The arm gestures are for moving forward, backward, right, left, and stop (Figure 3.2). It was collectively decided that the gesture for moving forward should be possible to signal with either arm, thus raising the right or the left arm should result in the TurtleBot moving forward. The group also came to the conclusion that the machine learning model should be able to recognize the given signal independent of the user's orientation (facing towards vs. away from the camera Figure 3.3).



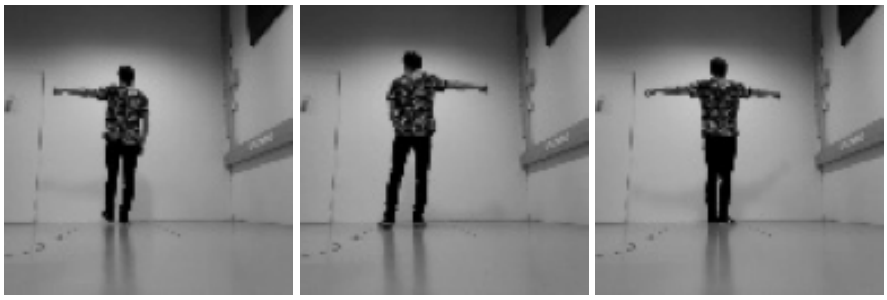
(a) Forwards (b) Forwards (c) Backwards

(d) Left (e) Right (f) Stop

Figure 3.2: Screen-dumps from the data-set after processing the images



(a) Forwards (b) Forwards (c) Backwards



(d) Left (e) Right (f) Stop

Figure 3.3: Screen-dumps from the backward-facing data-set after processing the images

3.1.2 Proof of concept

All use cases of this system aside, this project's purpose is to experiment and build the foundation for further development towards a working solution. Through this project, we aim to develop a proof on concept. We want to show that such a system can be developed and function properly.

Since this project strictly is going to represent a proof of concept, the group has decided there is no need for a focus group or any form of survey. This decision was made because whatever progress we make, FFI is looking for a proof of concept that a system like described can be made, therefore we deemed it more important to provide a functioning prototype. This allows FFI to look at the steps taken to make the prototype and improve our method, to create a better performing system.

3.2 Related Technologies

To successfully complete this project there are a lot of topics we have to research to gain a better understanding of the project and its requirements. Firstly, we will start with an overview of some related technologies.

3.2.1 Fully-connected neural network

A fully-connected neural network is one of the simplest NN architectures[1]. In a neural network like this, every neuron is connected to every neuron in the next layer, as seen in Figure 2.1. This architecture consists of only dense layers, which is explained in Section 3.2.4.

3.2.2 Convolutional neural networks

A convolutional neural network (CNN) is a class of deep neural networks that is most commonly used for analyzing and classifying visual elements such as images or videos. CNN's have the advantage of being able to learn filters/features from the input, rather than a developer having to hand-engineer these filters like in many other image classification algorithms[2].

Like more general neural networks, CNNs consists of an input layer, multiple hidden layers followed by an output layer. The hidden layers are typically a series of convolutional layers, ReLU layers (most commonly used activation function), pooling-, fully connected-, and normalization -layers. Finally, this type of neural network often includes backpropagation to optimize the model by adjusting the weights based on the calculated error[11].

Convolutional layer

Convolutional layers in CNNs are the core building blocks of the neural network. These layers consist of learnable kernels. Each kernel is convolved across the height and width of the input, computing the dot product, which produces an activation map. This results in the kernels being able to detect specific features extracted from the input (Figure 3.4).

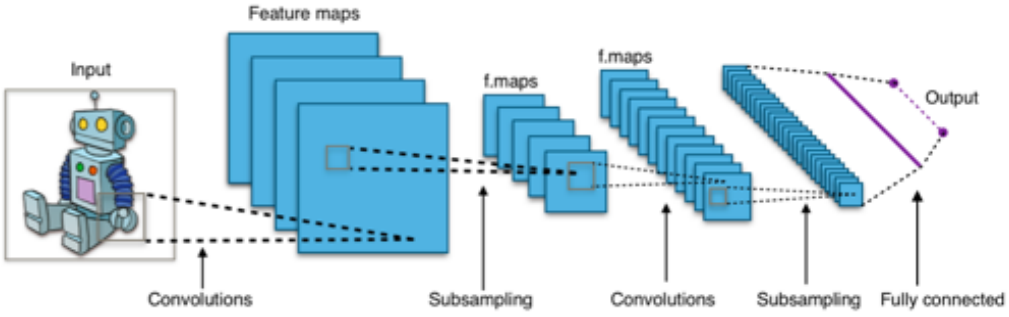


Figure 3.4: Visualization of convolution.
Obtained from
https://en.wikipedia.org/wiki/Convolutional_neural_network

[1] https://towardsdatascience.com/under-the-hood-of-neural-networks-part-1-fully-connected-5223b7f78528
[2] https://en.wikipedia.org/wiki/Convolutional_neural_network

Pooling layer

Processing large images requires more computations, more parameters, and memory. To reduce the required computational power, CNN's introduce pooling. Pooling is a form of non–linear downsampling[3].

Max pooling

There are several different pooling functions that can be implemented, however, max pooling is most commonly used. A max-pooling operation partitions the input into non–overlapping quadrants, and outputs the maximum for each sub-region. Most pooling layers use a 2x2 filter size (Figure 3.5).

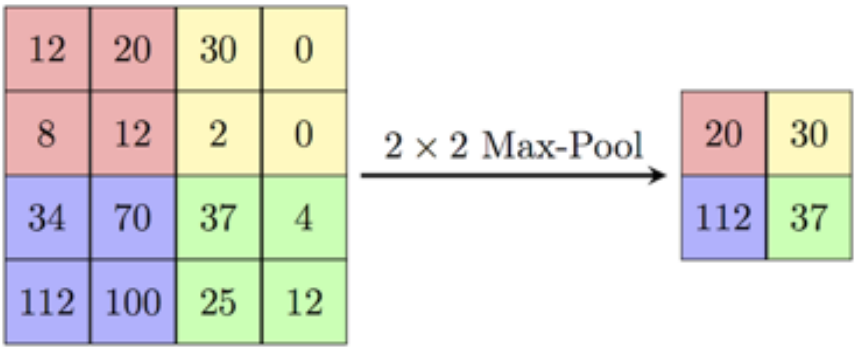


Figure 3.5: Visualization of 2x2 max pooling operation.
Obtained from
https://computersciencewiki.org/index.php/Max-pooling_/_Pooling

Average pooling

Average pooling works much like max pooling. This pooling operation, however, outputs the average of each non-overlapping quadrant, shown in Figure 3.6.

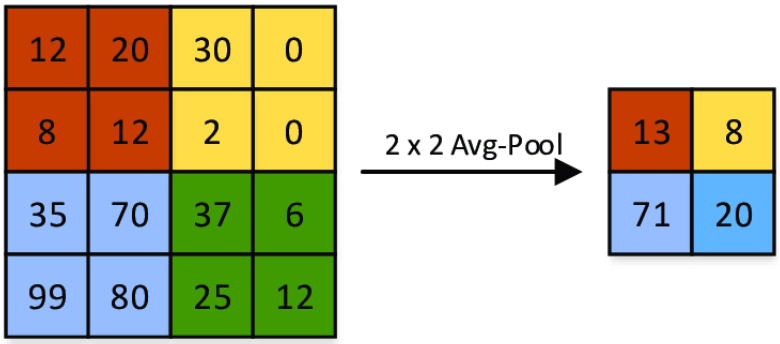


Figure 3.6: Visualization of 2x2 average pooling operation.
Obtained from https://www.researchgate.net/figure/Average-pooling-example_fig21_329885401

[3] https://en.wikipedia.org/wiki/Convolutional_neural_network#Pooling_layer

3.2.3 Activation layer

This section covers the activation layers. Activation layers are used in both fully connected neural networks (Subsection 3.2.1) and convolutional neural networks (Subsection 3.2.2). In a neural network, an activation function defines the output of a given neuron's input[4].

ReLU

There are different activation functions to increase non-linearity. The preferred function is the rectified linear unit (ReLU). ReLU layers increase the nonlinear properties of the decision function by setting negative values in an activation map to zero. Due to its simplicity, it is trivial to implement using a max function.

```
relu(input) = max(0, input)
```

The ReLU activation function is linear for values greater than zero and nonlinear for negative values as they are set to 0[5].

ELU

The exponential linear unit (ELU) is very similar to ReLU, except for negative values[6]. However, ELU smooths out slower than ReLU, as seen in Figure 3.7. ELU introduces a constant, alpha, which should be a positive number. ELU continues to slowly smoothen out until the output equals -alpha.

Softmax

Softmax is a popular activation function to use in the final layer of neural network-based classifiers[7]. The softmax function produces output values in the range 0 - 1, with the sum of these values adding up to 1. These values represent the probability distribution for each class, where the target class should have the highest probability. The probability distribution from Softmax is calculated using Formula 3.1[7].

$$f(x_i) = \frac{exp(x_i)}{\sum_j exp(x_j)} \tag{3.1}$$

Sigmoid

Sigmoid is one of the most common activation functions, often seen in shallow neural networks[12]. The Sigmoid activation function is popular in models that predict the probability of a specific outcome, as the output is in the range 0 - 1.

[4] https://en.wikipedia.org/wiki/Activation_function
[5] https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks/
[6] https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html#elu
[7] https://medium.com/@himanshuxd/activation-functions-sigmoid-relu-leaky-relu-and-softmax-basics-for-neural-networks-and-deep-8d9c70eed91e

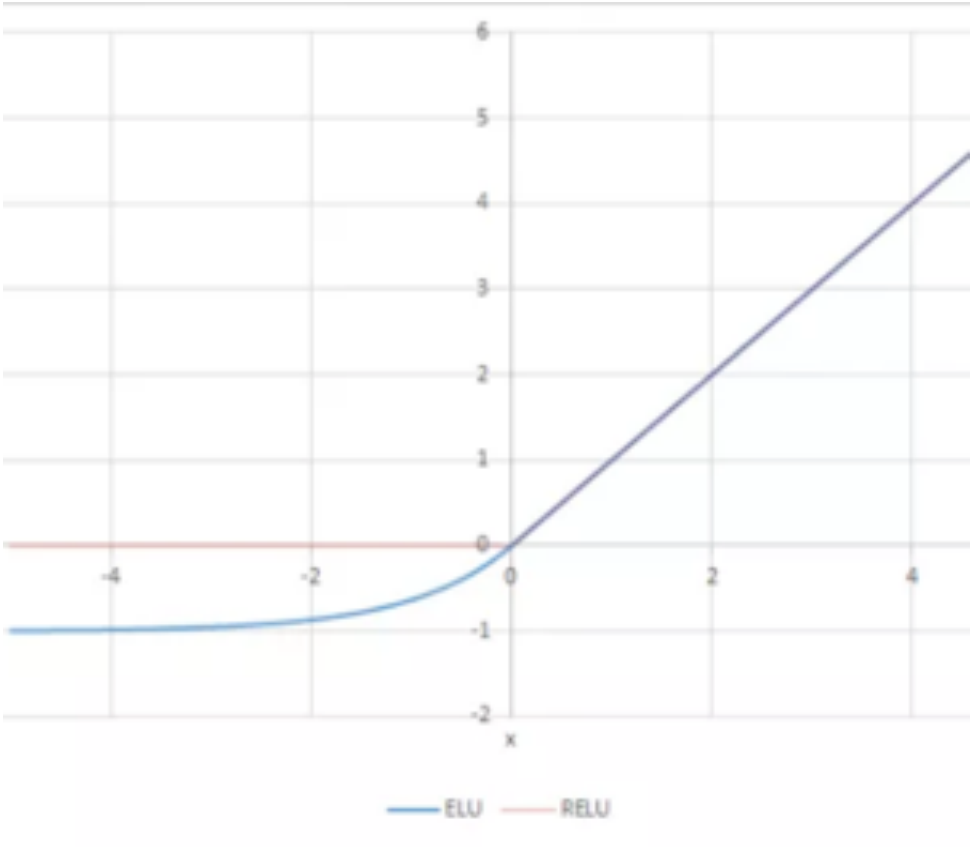


Figure 3.7: Visualization of how both ReLU and ELU smooths out.
Obtained from https://ml-cheatsheet.readthedocs.io/en/latest/activation_functions.html

3.2.4 Dense

A dense layer also referred to as a fully connected layer is used to connect every neuron in a layer to every neuron in the next layer[8].

[8] https://heartbeat.fritz.ai/classification-with-tensorflow-and-dense-neural-networks-8299327a818a

3.2.5 Dropout

Dropout is a method of regularization in deep neural networks where random neurons are dropped[3], as shown in Figure 3.8. This is done during training to prevent overfitting the neural network. The dropped neurons can be in either of the hidden and visible layers.

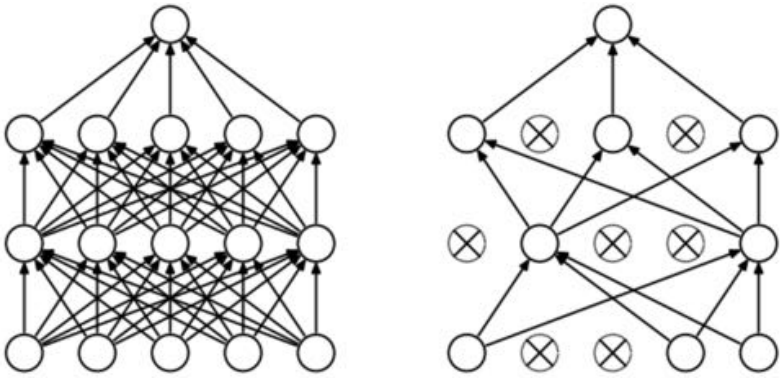


Figure 3.8: Visualization of how dropout works.
Obtained from https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

3.2.6 Flatten

Flattening layers take a pooled feature map as input and transforms it into a single column[9] (Figure 3.9). This single column is then fed to the network for processing.

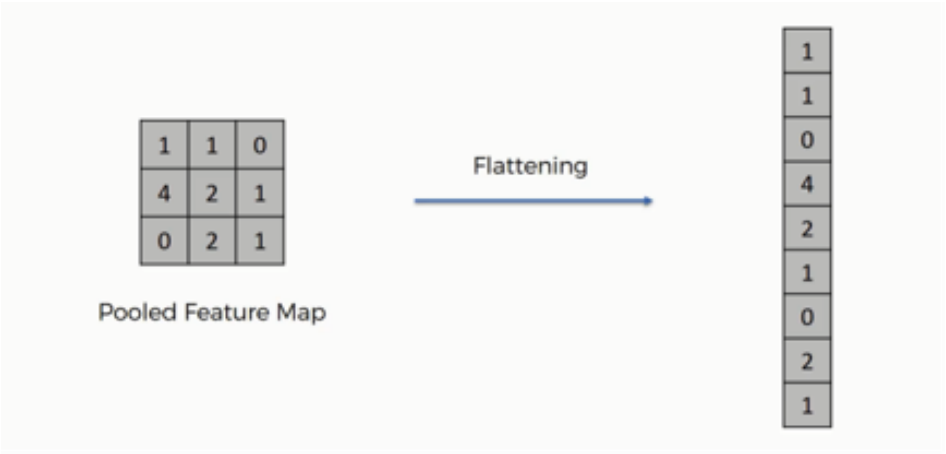


Figure 3.9: Visualization of flattening.
Obtained from https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-3-flattening

[9] https://www.superdatascience.com/blogs/convolutional-neural-networks-cnn-step-3-flattening

3.2.7 Optimizers

Machine learning as many other fields has a relationship with optimization[10]. Problems occurring when it comes to learning is described as minimization of some loss function when training on a set of data. A loss function shows the deviation between the previous and current predictions made by the model when training. The main goal is to generalize the data, and the purpose is to use the optimization algorithm to minimize the loss during training for new data. The general concern is to minimize the loss for data that has not been seen by the model so that it can produce an accurate prediction on new data .

Gradient descent

Gradient descent is an optimization algorithm used in machine learning to minimize some function[11]. This is done by moving down in the steepest descent. The descent is defined as the negative of the gradient. A visual representation of gradient descent can be seen in Figure 3.10.

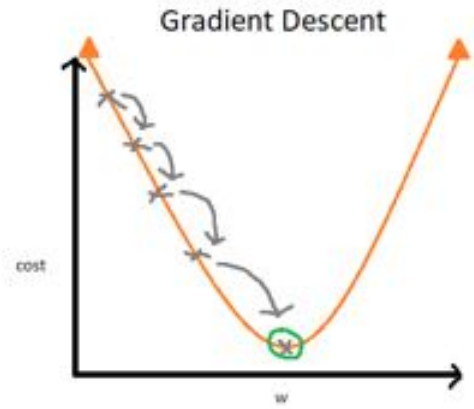


Figure 3.10: Simple visualization of gradient descent

In machine learning, there are parameters like weights. By using gradient descent the parameters in a neural network can be updated and adjusted. By starting at the top of the graph, then proceed to take the first step down in the direction that has been given by the negative gradient. After the first step, recalculate the negative gradient and take the next step in the direction that was given by the calculation. This process continues until it hits the bottom of the graph and it is not possible to continue to go downhill, or until a local minimum has been hit instead of the intended goal of hitting the global minimum.

[10] https://en.wikipedia.org/wiki/Machine_learning#Relation_to_optimization
[11] https://ml-cheatsheet.readthedocs.io/en/latest/gradient_descent.html

Adaptive moment estimation

Adaptive moment estimation or ADAM for short is an updated version of the optimizer RMSProp[12]. ADAM will take the average of both gradients the 2nd time that the gradients are being used. The process is done as follows, first, it finds the exponentially weighted average of past gradients, then it has to compute the exponentially weighted average for the squares of past gradients[13]. As the averages do have a bias towards zero, ADAM counteracts this by applying a bias correction. The parameters are then updated by using the information from the calculated averages.

Stochastic gradient descent

Stochastic gradient descent or SGD for short is a highly used optimization algorithm[4]. The learning rate is one of the most important and critical parameters for the SGD algorithm. When training a model it is important to slow down the learning rate over time. This is done because the gradient estimator of SGD will introduce some noise in the form of a random sampling of training examples.

3.3 Frameworks

This section covers the functionality of the frameworks that have been researched for this project.

3.3.1 Keras

Keras is a high-level API for training and building machine learning models[14]. It is used for rapid prototyping, production, research and has some key advantages. Keras is modular and composable, the models produced with Keras are made by connecting configurable blocks with few restrictions. As Keras is a high-level API, it is simple to use and has a consistent interface for common use cases.

3.3.2 TensorFlow

TensorFlow is a framework with multiple use cases as explained in Section 2.3.2. What makes this framework interesting for this project is all the tools that are available as standard.

Keras with TensorFlow

As standard, TensorFlow includes with Keras. This makes it a more beginner-friendly approach to machine learning for developers[15].

TensorBoard

TensorBoard is a visualization tool for machine learning[16]. TensorBoard is able to track and visualize metrics for loss, accuracy, model graphs, histograms of weights, biases, and much more.

[12] https://en.wikipedia.org/wiki/Stochastic_gradient_descent
[13] https://ml-cheatsheet.readthedocs.io/en/latest/optimizers.html#adam
[14] https://keras.io/why_keras/
[15] https://www.tensorflow.org/guide/keras
[16] https://www.tensorflow.org/tensorboard

GPU accelerated training

A GPU is a hardware component that is designed to perform computations needed for 3D graphics[17]. Normally a GPU is used for video games to compute polygons, to show the player on a screen. For simplicity sake, a GPU can be described as a large array of small processors that perform parallel computations. Even though the processors on a GPU is slow, there is a large quantity of them which allows for efficient parallelism. Since they are specialized in numerical processing, a GPU will perform better when training and validating a machine learning algorithm than a traditional CPU.

Since TensorFlow supports GPU accelerated training of models, the process of training a model will take a shorter amount of time[18]. The disadvantage with TensorFlow is the requirement of having a CUDA NVIDIA GPU[19].

3.3.3 PlaidML

PlaidML is an advanced portable tensor compiler as explained in Section 2.3.3. What makes this framework interesting for this project is support for hardware that is not supported by other frameworks like TensorFlow.

Keras with PlaidML

PlaidML can be used as a component under Keras the same way TensorFlow does[20]. This, in turn, means that with PlaidML a developer has the same user-friendliness as they would using TensorFlow with Keras.

GPU accelerated training

As with TensorFlow, PlaidML is also supporting the option of GPU accelerated training of models[21]. TensorFlow requires a CUDA NVIDIA GPU, while PlaidML supports most hardware.

Visualization tools with PlaidML

PlaidML does not have a built-in tool to visualize data like TensorFlow's TensorBoard (Subsection 3.3.2). However, since Plaid can be used under Keras, a package like matplotlib (Subsection 2.3.6) can be used to graph metrics like loss and accuracy from a training session. Keras' `fit()` function call that is used for training, returns a history object that contains all the recorded training loss and metrics values at successive epochs, in addition to all the validation loss and metric values. Because of history object that is returned, all the values can be plotted with matplotlib[22].

[17] https://databricks.com/tensorflow/using-a-gpu
[18] https://www.tensorflow.org/guide/gpu
[19] https://www.tensorflow.org/install/gpu
[20] https://plaidml.github.io/plaidml/
[21] https://plaidml.github.io/plaidml/
[22] https://keras.io/api/models/model_training_apis/#fit-method

3.3.4 OpenCV

OpenCV is a library for computer vision. The use cases range from machine learning to video and image processing. There are valuable features such as object detection, motion analysis, object tracking, and feature detection[23]. These are all great features that could be used in a gesture recognition system for controlling a vehicle. The built-in feature that will be the most useful for this project is the functionality for image processing. The machine learning model that is used in the prototype will use images for training, validation, and testing. Therefore, there is a need for processing a fully scaled image before it is fed to the model. The functions for writing, resizing and reshaping an image is using Numpy ndarray as input, while the input for reading is an image file and returns a ndarray[24] and the other functions takes the ndarray as input[25].

3.3.5 rospy

The rospy library is a Python interface to ROS, allowing easy access to functionality and communication with the components of the TurtleBot[26]. One of the key features of the rospy library is the publisher and subscriber functions, this allows for communication between scripts and nodes that could be utilized for asynchronous executions[27]. Controlling the movement of the TurtleBot could simply be executed by publishing the topic directly to ROS from a Python script. The manual approach for executing a movement is using the command line and rostopic[28].

```
rostopic pub /cmd_vel geometry_msgs/Twist -- [0, 0, 0] [0, 0, 0]
```

Being able to automate this execution of movement could lead to a responsive prototype that moves smoothly in the directions that are predicted by the ML model. Moving the TurtleBot is possible by inserting values into the linear and angular arrays at the end of a rostopic command. The arrays contain values for x, y, and z that represent coordinates of the TurtleBot's relative position in a given space[29]. The value for x is used for movement on the horizontal axis, which is forward and backward and is used in the linear array. The value for z is passed to angular and is used for movement on the z-axis in a three-dimensional space, this is left and right for the TurtleBot. The value for y represents the vertical axis, making it irrelevant for this project considering the TurtleBot is a ground vehicle.

[23] https://docs.opencv.org/2.4/modules/refman.html
[24] https://docs.opencv.org/2.4/modules/highgui/doc/reading_and_writing_images_and_video.html
[25] https://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html
[26] https://wiki.ros.org/rospy
[27] https://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers
[28] https://wiki.ros.org/Robots/TIAGo/Tutorials/motions/cmd_vel
[29] http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

3.3.6 PiCamera

The PiCamera package is a Python interface for the camera module connected to a Raspberry Pi[30]. The package has built-in functionality that allows for easy control when using the interface. With this package, it is possible to configure a lot of preferences for the captured frames, along with format, resolution, and frame rate. PiCamera captures images, and the user can specify the preferred image format. If no format is specified, the method will attempt to guess the format based on an extension in the file name, if the guess fails, an error will be raised. The format can be any object as long as it supports Python's buffer protocol[31]. This means that an image could be captured as a numpy ndarray containing pixel values, which is a great combination with OpenCV for pixel manipulation. Images can be captured with `sleep()` to capture n amount of images within a desirable time frame, and it can capture frames continuously with a defined frame rate.

3.4 Robot Operating System

ROS allows for complex software design, without knowing how certain hardware components on the robot works. The system provides a way to connect a network of nodes with a central hub[32]. These nodes can be run on multiple devices, and they connect to the main hub in various ways. This section will introduce some of the core functionalities in ROS.

3.4.1 Nodes

ROS is designed to be a loosely coupled system that utilizes nodes. A node is simply a process running on the robot that performs computation and is responsible for one single task. Nodes are combined together into a graph and communicate with each other using messages[33]. A robot control system will usually consist of many nodes. For example, one node controls a laser range-finder, one Node controls the robot's wheel motors, one node performs localization, one node performs path planning, one node provides a graphical view of the system.

3.4.2 Messages

Nodes communicate with each other using messages. Messages are a simplified message description language for describing the data values that ROS nodes publish. This description makes it easy for ROS tools to automatically generate source code for the message type in several target languages. Message descriptions are stored in .msg files in the msg/sub-directory of a ROS package. A message is described as a list of data, the message data consists of two parts: fields and constants[34]. Fields are the data that is sent inside of the message. Constants define useful values that can be used to interpret those fields.

[30] https://picamera.readthedocs.io/en/release-1.13/index.html
[31] https://picamera.readthedocs.io/en/release-1.13/recipes2.html
[32] http://wiki.ros.org/ROS/Introduction
[33] https://roboticsbackend.com/what-is-a-ros-node/
[34] https://www.allaboutcircuits.com/technical-articles/an-introduction-to-robot-operating-system-ros

3.4.3 Topics

Messages are passed between nodes and exchanged via logical channels called topics. Topics are named buses and have publish/subscribe semantics[35]. In general, nodes are not aware of who they are communicating with. Instead, nodes that are interested in data subscribe to the relevant topic. Nodes that generate data publish to the relevant topic. There can be multiple publishers and subscribers to a topic (Figure 3.11).

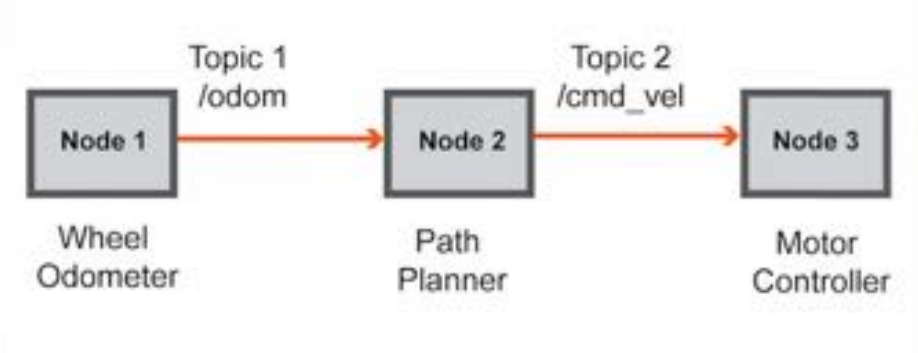


Figure 3.11: Visualization of how nodes communicate using topics.
Obtained from https://www.allaboutcircuits.com/technical-articles/an-introduction-to-robot-operating-system-ros

3.5 Related Works

In this section, we will look at some related works and give a brief overview of how these solutions were able to achieve their goals. Even though these publications necessarily do not use the same technologies as we do, there is a lot to learn from these processes.

3.5.1 Posture recognition powered by Kinect

In the paper *Human posture recognition using human skeleton provided by Kinect*[8], Thi-Lan Le, and colleagues experiment with Kinect in the context of a health monitoring framework. In their experiment, they were able to accomplish high accuracy predictions by using a self-made joint database. They were able to predict the following postures using SVM (Support Vector Machine) and joint angles:

- Standing
- Sitting
- Lying
- Bending

[35] https://www.allaboutcircuits.com/technical-articles/an-introduction-to-robot-operating-system-ros

Figure 3.12: Overview of the joints Kinect tracks.
Obtained from https://possiblywrong.wordpress.com/2012/11/04/kinect-skeleton-tracking-with-visual-python/

Kinect has a feature that tracks a person in the frame by storing properties about the individual's joints (Figure 3.12). One of these properties is orientation, which is given by four values (x, y, z, and w). These four values represent the skeleton's position, depth, and rotation relative to the Kinect[36].

Thi-Lan Le and colleagues were able to identify which joints were significant to differentiate between the different positions. Based on the identified joints they calculated different sets of angles from the skeleton view (Figure 3.13), which could be used to identify the posture.

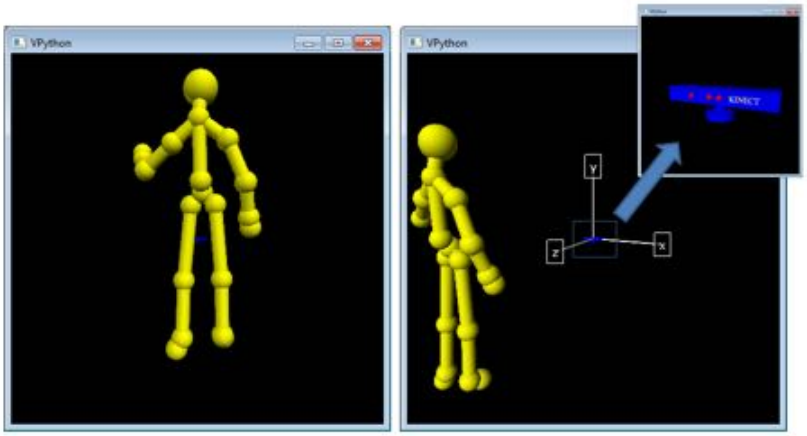


Figure 3.13: Kinect skeleton view.
Obtained from https://possiblywrong.wordpress.com/2012/11/04/kinect-skeleton-tracking-with-visual-python/

[36] https://medium.com/@lisajamhoury/understanding-kinect-v2-joints-and-coordinate-system-4f4b90b9df16

During testing, the authors ran both offline and online evaluations. In their online evaluations, they tested with data of a person not present in the training data. The test-data were captured directly from Kinect to ensure authenticity. However, variations between the subject's position in relation to the Kinect in the test data compared to the training data caused the recognition accuracy to drop. In these cases, joint angles were more relied on.

3.5.2 Posture recognition apparatus and autonomous robot patent by Honda Motor Co Ltd

Honda Motor Co Ltd has in the patent *Posture recognition apparatus and autonomous robot*[6], been able to develop a posture recognition apparatus that is able to recognize instructions signified by postures of persons that are present in images obtained with an image capture device.

By using the images obtained by the capturing device, the posture recognition apparatus is able to extract outlines of a body that might be a person. Once the potential person is outlined, a calculation of the distance to the body is performed.

Based on the outline and the distance to the body represented by the outline, the apparatus is able to search the candidate for a person's hand. Once a person's hand has been located, it is able to determine an instruction corresponding to the relative position of the candidate (Figure 3.14), and output the result.

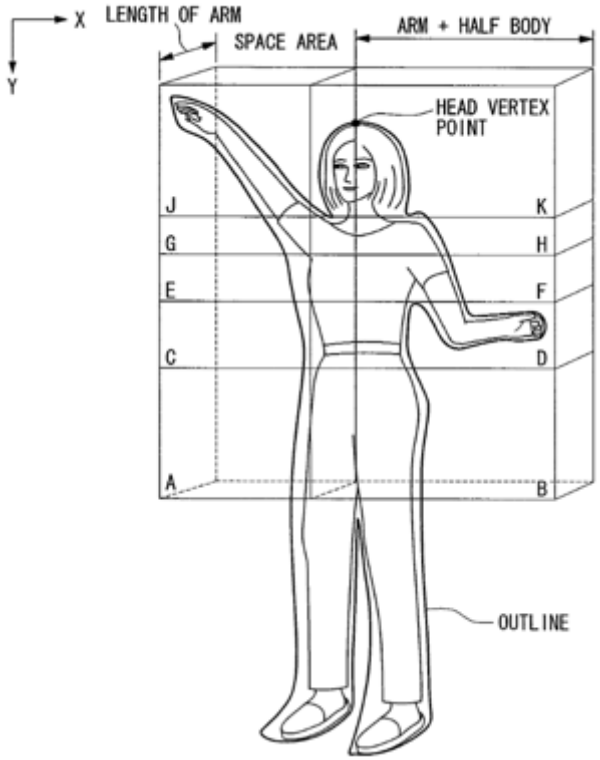


Figure 3.14: Example of person candidate and its relative position.
Obtained from https://patentimages.storage.googleapis.com/US7340100B2/US07340100-20080304-D00009.png

Honda and its inventors used a priority sequence consisting of 10 different postures. All of which were given an id, as well as an area code on the potential person, where the posture was most likely to be present (Figure 3.15). Utilizing a priority sequence, determined by the complexity of the gesture, the apparatus can check for the most difficult postures first, even though the posture might share common properties as a different gesture.

Since the apparatus is able to identify multiple people, and thus multiple gestures at once, it is able to identify common gestures such as ”move left” or ”move right” that only require one hand gesture, but also more complex gestures that require the use of both hands such as ”waving for attention” or ”shaking hands”. This allows the technology to be used in a wider variety of applications, mainly where more complex gesture recognition is required.

POSTURE ID	AREA SECTION	POSTURE	PRIORITY SEQUENCE
1	A	SHAKE HANDS	4
2	B	SHAKE HANDS	4
3	C	ATTENTION	3
4	D	ATTENTION	3
5	E	STOP	1
6	F	STOP	1
7	G	MOVE RIGHT	2
8	H	MOVE LEFT	2
9	J	GOODBYE	5
10	K	GOODBYE	5

Figure 3.15: The priority sequence diagram used for the apparatus.
Obtained from https://patentimages.storage.googleapis.com/US7340100B2/US07340100-20080304-D00008.png

3.5.3 Real time interaction with mobile robots using hand gestures

Using arm gestures for Human-robot interaction is not a new concept, and it has been conducted a lot of research around this topic. K. Konda, A. Königs, D., and H. Schulz has researched if it is possible to develop a model with acceptable accuracy for recognizing simple gestures[5]. This team has developed a model using convolutional neural networks for gesture recognition.

The model developed by this team is detection-based, meaning that their model is not tracking the joints of the person in the frame. This makes for a simpler model that could execute with better resistance for rapid movements. The robot interaction system they developed is relatively old (2012), meaning that the frameworks and architecture are somewhat outdated in terms of new technology that has emerged. More research has been conducted throughout the years, and frameworks for building simple and efficient machine learning models have been improved.

The team that developed this prototype concludes that it is possible to develop a model that allows for interaction between humans and robots outdoors. After testing their model, they managed to get 0.005 classification error on training data and 0.01 on test data. This shows that it is possible to develop a similar model. The model was built by using a dataset of 10000 depth images.

3.6 Use Cases

This project focuses on developing a system that can interpret what gesture a user signals to the robot, this will be done to see if it is viable to use this technology on tracked autonomous vehicles. This isn't the only use case for machine learning and image recognition, there are plenty of opportunities that could be explored.

3.6.1 Heavy machinery

This system could be implemented in heavier machinery, which would be beneficial for the industries that are currently relying on these types of vehicles. Construction, transportation, and agriculture are some of the industries where the system we are proposing could be useful. We will have a look at the benefits and challenges of using this system in this particular industry.

Benefits

Being able to maneuver a vehicle while standing outside of it gives the operator more control, this provides a full view of the vehicle's surroundings. The most general use cases for these vehicles is guiding it onto a trailer and backing up towards a trailer to hitch it to the vehicle.

Backing up towards a trailer is not the most difficult operation an experienced truck or tractor driver face as today's vehicles have mirrors and cameras that will improve the view from inside the vehicle. However, having the opportunity to stand outside of the vehicle and get a full view of the surroundings would be helpful for safety measures. The operator can ensure that there are no obstacles behind the vehicle, avoiding potential damage.

The most useful feature would be to load machines onto trailers. To load a vehicle on top of a trailer needs two people, one driver and one that can guide the driver and overlook the path towards the trailer. By maneuvering the vehicle with arm gestures instead of guiding a driver, one operator can be in control of the entire vehicle, resulting in relieving another from their duty. Having only one operator could reduce the number of failures, as the operation is heavily relying on the accuracy of the model and proper usage.

Challenges

The most challenging part of this system for heavy machinery would be safety. These vehicles are of significant weight and size, meaning that a minor flaw could have severe consequences. There are several safety measures that are necessary to handle this system to be viable in this industry.

An important safety measure would be to implement a LiDAR sensor to map out obstacles and the general surroundings, or using a proximity sensor[37] for alerting the operator when nearby obstacles are too close. Having these sensors as part of the autonomous vehicles will allow the system to stop the vehicle when the operator tries to signal a movement towards an unsafe area. Distance between operator and vehicle needs to be tracked, to be certain that they do not end up to close to each other.

Another challenge would be to implement this system in different vehicles. Manufacturers of vehicles use different software and embedded systems, this means that this system would need to be adaptable to different manufacturers. The features that we would need from the different vehicles are steering and throttle control, a camera module is also necessary for enabling gesture recognition.

3.6.2 Aircraft ground maneuvering

A system like this could be implemented in aircrafts, this could be beneficial for the industry and minimize the risk of human error. Planes are already integrated with advanced computer systems and most passenger and freight aircraft have a camera mounted to the landing gear at the nose of the plane which could be put to use[38].

Benefits

Currently, an operator on the ground is guiding the aircraft into the gates while the pilot or co-pilot is following the specified directions. The basic requirements for implementing a system like this is already in place. However, there are some significant modifications needed to make this a viable solution for this use case.

Challenges

Implementing a system like this into an aircraft would raise a few challenges. Human safety, the margin of error, the pilot's option to override the system if necessary, compatibility with the aircraft's current hardware, ease of use, new gestures.

Using this system in an aircraft would require an immense amount of testing to ensure safety. This is for the safety of the personnel on the ground in addition to the passengers and staff on-board the plane. In case of a failure, the pilot or co-pilot would have to be able to take instant control over the aircraft.

Airports today have standard signals given by the ground personnel. Therefore, the system would have to be specifically trained on these gestures as it is unreasonable to implement a new set of gestures for the ground personnel to use. Training the system with the already existing signals would also enable a more seamless transition into using the system.

There is still the issue of compatible hardware for implementing this system into different aircrafts. For the system we are developing to be useful, the hardware and embedded systems would need support for ROS. Without this support, the system would need to be customized for aircraft's, meaning further development would be necessary to integrate the system.

[37] https://en.wikipedia.org/wiki/Proximity_sensor

[38] https://eu.usatoday.com/story/travel/columnist/cox/2017/07/02/onboard-cameras/442856001/

Chapter 4

Planning

This chapter covers the planning of the project. Section 4.1 covers possible options regarding how to create the datasets for this project. Continuing, Section 4.2 goes over choosing the type of neural network that we could use for the project. Section 4.3 goes over the process of picking the correct framework for the prototype. To finish the chapter, Section 4.4 will contain the plan for developing the TurtleBot Waffle Pi to become a feasible prototype.

4.1 Datasets

This section will cover our plans regarding gathering data. The resulting data will be used for training and testing the machine learning model. We will give a brief summary of all the different methods we considered.

4.1.1 Discussing different solutions

When we started planning what the dataset should look like in terms of size and content, we looked at what was needed from a dataset to get high accuracy and the gestures we wanted to use for maneuvering the TurtleBot. We want to use the camera module on the TurtleBot to create data from the point of view of the actual vehicle we are using for the prototype. After knowing which gestures to use and what device to record with, we started discussing if we should use images or videos in the dataset.

Option 1 - Video

The first option is to record ourselves performing each gesture and use the video-stream as input to the neural network. However, we quickly realized that this option raises some concerns. Firstly, using a video-stream as input requires a more complex neural network than we had anticipated. Secondly, we deemed it unnecessary to record videos of ourselves holding a static gesture.

Option 2 - Images

Our second option is to use a script to take a given amount of pictures in a set time-frame. This option would allow us to use images as input to the neural network, the issue we are facing here is capturing a lot of images within a short time frame. However, looking into this option made it clear that somehow using images would be the better option.

Option 3 - Frames

The last option combines both option 1 and 2. We realized we could record ourselves holding each gesture for two minutes before processing the video file. We could write a processing script which takes a video file as input and saves a frame from that video several times each second.

4.1.2 Creating a dataset

This subsection will give an overview of our thought-process before starting to manufacture datasets and a possible, more efficient solution than our original plan.

Initial plan

We chose to use images for the datasets by processing video files (Option 3 in Subsection 4.1.1). The reason for this decision was how it allows us to produce fairly large datasets quickly.

We figured the best method for creating the datasets was to record ourselves holding each gesture while walking around in the frame. When we are extracting frames from the video we would have a large number of images for each gesture with various positioning in frame.

Initially, we wanted a dataset containing a simple background with few distractions. In addition, our goal is to make datasets where the user in the frame could be either facing the camera or have their back turned towards it. This means that we have to record each gesture twice to get both perspectives.

We were planning on creating several datasets, all with varying complexity and distractions. We realized this had to be done for the deep learning model to be able to work in a real-life scenario.

Revised plan

After creating a few datasets with our original solution we came across a few issues in the process. Due to these issues, the model struggled with classifying gestures in unseen data. We realized we had to take a step back and reevaluate the process for creating data if we were going to be able to train a functioning model.

The first measure to be taken is to only use data where the person in the frame is facing the camera. This will allow us to narrow down the possibilities of where the model might get confused. In addition, this would shorten down the time it takes to create a dataset by 50%.

The second measure to take into consideration is strictly using images instead of extracting frames from a video. Each image would be of a person simply holding a gesture. We plan to do this by capturing images using the TurtleBot. One person will stand in front of the greenscreen at Østfold University College, before applying data augmentation as described in Subsection 4.1.2.

By switching to still images in front of a greenscreen we can mass-produce data with an infinite variety. Also, by switching to the described solution we could save a lot of time considering we no longer had to record videos for each new dataset. However, this method would require more processing time due to data augmentation.

Data augmentation

Increasing the diversity of the data is crucial because of the complexity it adds to the deep learning model. However, gathering an immense amount of data is time-consuming. Luckily, there is a way to easily produce more data from existing data.

Greenscreen

By utilizing a greenscreen we are able to segment out the green background and insert a new background of our choosing. This provides the opportunity to dynamically create different environments for each dataset, giving us more control over the complexity of each dataset. This allows us to easily train the model for a specific environment.

Transformations

Since we already have a greenscreen available, it suggested that we could easily perform basic transformations on the person in the frame to alter their position and orientation. This would allow us to expand each dataset immensely.

When processing the images, before replacing the green background we can use a script to manipulate the image and move the person to other positions within the frame. This procedure is used before replacing the green background to avoid any confusion about whether or not a pixel is a part of the background or the person.

4.2 Model

There were many new and foreign concepts we had to familiarize ourselves with before starting the process of choosing a neural network architecture. Through researching the subject we quickly discovered that convolutional neural networks are the most common when working with image classification. However, there are other architectures we want to consider as well. This section covers our thoughts regarding architecture and how we should proceed to develop an ML model.

Convolutional neural network

As stated in the brief introduction to this section, we decided to mainly focus on developing a convolutional neural network (CNN). Throughout the process of working on Chapters 2 and 3 we got a lot of ideas as to how we could build a CNN. Our first thought is to experiment with a few different CNN architectures to see how each of them perform. Whichever model gives the best results will be our primary choice for this project.

Fully connected

Even though developing a CNN model seemed like the best option for this project, we still wanted to experiment with different types of neural networks. In comparison to a convolutional neural network, a fully connected is much simpler. By using a simpler architecture we do not expect any astonishing results, but it may serve as guidance to how our other attempts perform.

4.3 Picking the Right Frameworks

This section will cover our thought-process in regards to choosing the appropriate frameworks to use. There was a lot of back and forth within the group when making this decision as most frameworks seem reliable and easy to use.

4.3.1 Initial plan

Initially, we had some discussion with people at the IT department at HiOF, who all seemed to recommend PyTorch (Subsection 2.3.1). However, we will also consider TensorFlow (Subsection 2.3.2) with Keras (Subsection 2.3.4) because of built-in tools like TensorBoard[1] for plotting training accuracy and loss, in addition to having GPU accelerated training. The TensorFlow documentation is better than PyTorch's, making it a more sensible choice.

Looking at most introductions to neural networks and image classification, TensorFlow seemed to be the go-to framework to use. In addition, we plan on using Keras on top due to its simplicity and user-friendliness.

Having decided what framework to use for the machine learning portion, we had to pick a framework for image processing. We plan on using OpenCV due to previous experience with this framework (Subsection 2.3.5) and the fact that the TurtleBot uses ROS in collaboration with OpenCV for image handling.

4.3.2 Revised plan

After a period of working with TensorFlow and Keras, we were facing problems with setting it up correctly for training a model on a GPU, even though we had a compatible graphics cards. This meant that the work times to train a model would take to much time. Another problem that surfaced was how hard it was to train our model using our laptops, which did not have supported GPUs with TensorFlow.

To solve these problems, Allan Lochert advised us to look into PlaidML (Subsection 2.3.3) which allows us to train on any GPU. Keras works on top of PlaidML and all models trained with it are able to be used with TensorFlow. By using PlaidML we solve our problems with the TensorFlow setup process and training time. Because of that, we aimed to stay with PlaidML for the rest of the project.

By changing to PlaidML we were not able to continue to use TensorBoard for plotting training accuracy and loss. The easiest fix for this is to use Matplotlib (Subsection 2.3.6) to graph all the training values.

4.4 TurtleBot3 Waffle Pi and ROS

There are several necessary steps required to prepare the TurtleBot for integration with the model. The first step is to record videos using the Raspberry Pi and the camera installed. Through this step, we want to start developing the dataset and get a confirmation that the camera module for the Raspberry Pi works properly. After the first step, we want to connect a remote computer to the TurtleBot so we can SSH into the Raspberry and work from a more reliable computer. The third step is to make the TurtleBot move on command using ROS, this is to have everything ready before starting on the fourth step. The fourth step is to connect the model to the TurtleBot, so we can handle video-streams and pass in frames to the model and use the response from the model to make the TurtleBot move.

[1] https://www.tensorflow.org/tensorboard

4.4.1 Recording videos

Having decided that we were going to select frames from video-streams to create the dataset, we have to start the development process by using the TurtleBot to record videos of the group doing the gestures. Because of this, we decided to set aside the development for moving the TurtleBot until we are done gathering data. We plan on getting the data from the camera on the TurtleBot through Raspberry Pi to get the exact point of view we are going to use when testing the prototype.

4.4.2 Configuring the remote computer

After recording all the video-streams for the dataset we are going to connect a remote computer to the TurtleBot. When we manage to do this, we will map out the development process and plan how to get the prototype to work together with the model. We laid out a four-step plan to develop the prototype, the plan is described in this subsection.

4.4.3 Developing the prototype

Step 1 - Configure movement

Firstly, we were going to develop a program to control the TurtleBot based on simple input. The input will be a sequence of integers based on the type of movement that should be conducted. This is to tweak the amount of torque the wheels should have when it should execute an action based on image sequences later.

Step 2 - Configure model with the TurtleBot setup

After figuring out how to run a python script on the TurtleBot to control it, we are going to start tweaking the singular movements for each gesture. When the movement is finished, we will test it with simulated values that will imitate the responses from the model. During this step, we will decide whether it is more efficient to use commands to control the TurtleBot or use the rospy library for Python.

Step 3 - Connecting the model with movement script

After having the movement for the TurtleBot in place, we can start the process of connecting the model to the movement script. From here we are going to test the model by passing in singular images to the model and use the response to pick the correct function to move the TurtleBot. The main goal for this step is to run the model on the actual TurtleBot so we can proceed to use live video-streams to control it.

Step 4 - Handle video-stream to control the TurtleBot

The last step is to develop a script for recording video-streams and pick out frames that will be passed into the model. This step will contain a lot of tweaking to determine the amount of time it will take to process the video-stream, get a response from the model, and then do the correct maneuver.

Chapter 5

Implementation

This chapter describes how the group found what seems to be the best solution for classifying the gestures by doing experiments with multiple different machine learning model architectures and datasets. It also covers the setup process of our TurtleBot3 Waffle Pi and the integration between our model and the TurtleBot.

The chapter starts with Section 5.1 that describes the processes used to manufacture high-quality datasets for training and testing. Section 5.2 covers the architectures of the different models that were created. Continuing to Section 5.3 that describes how the TurtleBot was set up. The chapter finishes with the integration of the model on the TurtleBot in Section 5.4.

5.1 Dataset

This section covers the methods that were used to process and manufacture the datasets used in the project. There were two main methods used to create the necessary data, one more efficient than the other. The second method was implemented due to unexpected flaws in the first manufacturing process. Both methods and the resulting datasets are described in the following subsections.

5.1.1 Initial datasets

For this project, we decided to create datasets for training and testing ourselves. The group uses a TurtleBot3 Waffle Pi for prototyping. To create the datasets we use two different techniques. The initial datasets were recorded at 15 frames per second with a resolution of 1000x1000 pixels. This was done for each member of the group, along with some volunteers. Everybody involved was recorded using the gesture categories forward, backward, left, right, and stop. All datasets contain images with the people facing the camera and with their backs towards the camera. These datasets where made with the initial plan (Subsection 4.1.2).

While recording each video we had one member overlook the entire process to make sure that there was as little deviation in the gesture quality as possible, while also making sure the person stayed in the frame.

After each recording, the video was been properly named. This included the name of the subject, which gesture, back or front, and the length of the video. With such a naming scheme and additionally grouping every gesture into its own folder, we could simplify the script for extracting frames. This allowed us to automate the process of labeling each image based on the name of the video it was extracted from.

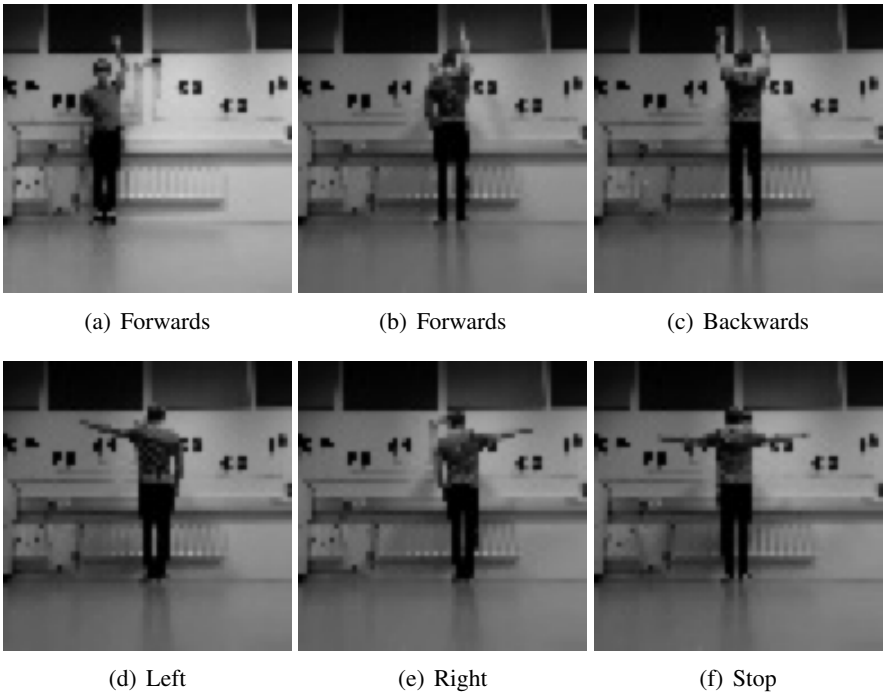
Dataset 1

The first dataset was made to be as simple as possible. This was done to give the neural network a gentle introduction to the gestures when starting training. Dataset 1 contains 107 650 images, with 21 530 images for each category of arm gestures. The dataset is completely balanced for each category, avoiding that the model prefers one gesture over another due to imbalanced training. A small representation of the dataset is shown in Figure 3.2 and 3.3.

Dataset 2

Dataset 2 was made with the intention of making a more complex dataset. This was done to train the model on more advanced backgrounds. In this dataset, we introduced more objects and distractions in the background, such as images, different colored panes, and everyday objects.

Dataset 2 is made up of 89 735 images split over the five-arm gesture categories with 17 947 images per category. A small representation of the dataset is shown in Figure 5.1.



(a) Forwards (b) Forwards (c) Backwards

(d) Left (e) Right (f) Stop

Figure 5.1: Screen-dumps from dataset 2 after processing the videos and converting them to images.

Dataset 3

Dataset 3 was made to further increase the complexity and introduce the model to more realistic data. Since this system is intended to be used outdoors we decided to use a forest as the background. However, since TurtleBot is not very suited for being outdoors we had to place it in an open window. This allowed us to record outside while not risking damaging the robot. The set consists of 70 415 images divided over 5 categories; forward, backward, left, right, and stop.

Testset 1

Testset 1 was made purely for testing, which means that the model has not seen this data in training. It is made with a simple background, this is done so we can test models and see if it predicts correctly. This set is split up in the same five categories as dataset 3 (Subsection 5.1.1).

5.1.2 Greenscreen datasets

After having some problems with training our model, we concluded that our datasets were the problem. Due to an insufficient amount of data with poor quality, the model did not perform well in training and validation.

Trying to fix these problems we revamped the data manufacturing process. We took one image for each of the categories; forward, backward, right, left, and stop. These images were taken in front of a greenscreen. This allowed us to change the background dynamically (Figure 5.2) for each new dataset without being in a physical location. Those images were then manually manipulated to become 29 images for each category. From there we could use data augmentation to alter the images and by that create new images on a much larger scale. By creating a base dataset with a greenscreen we are able to create as many datasets we want (Subsection 4.1.2).

We realized that we would also have to classify gestures that were wrong. This had to be done, otherwise, the model would classify an invalid signal as one of the valid categories, which would make the TurtleBot move when it is not supposed to. To do this we captured 68 different images of incorrect signals, that has also been altered with data augmentation and are included in every dataset made with this technique. Description of how the processing is done can be found in Section 5.1.3.

A decision was also made when it comes to the forward signal. Trying to decrease factors that could confuse the model, the forward signal was changed to just have the right hand pointed towards the sky as seen in Figure 5.2(b). In the previous dataset, the gesture for left and right was pointing the TurtleBot towards a direction. This was changed due to labeling in this dataset. By holding the left arm out, the vehicle will turn left (Figure 5.2(a)).

(a) Left (b) Forwards

Figure 5.2: Screen-dumps from 2 different datasets showing the result after slightly re-positioning the person and changing the background.

Datasets

For this project, eight datasets for training have been made, cumulatively consisting of 3 937 800, 100x100 pixel grayscale images. By using the process described in Subsection 5.1.3, we were able to create countless datasets with varying complexity, for both training and testing. The main limitation to how many datasets we can manufacture is processing power.

Testsets

When making the greenscreen testsets we used the same technique that was done with the greenscreen datasets (Subsection 5.1.3). In total, we made 984 450 images for testing. A screen dump from the greenscreen testset 1 can be seen in Figure 5.3.

Figure 5.3: Screen dump from Greenscreen tetset 1, backward gesture.

5.1.3 Processing greenscreen images

To process the images, we wrote a script to iterate over all the images in the specified folder. For each image, the script makes a copy on which all the processing will be done. The script starts with masking out and moving the person, then it places the new background image. Finally, the image is scaled down from 1000x1000 pixels to 100x100 and converted into grayscale. All the new images from this process are then saved to a new folder for that specific category. Every image is then ready to be labeled. The script can be found in Appendix E data_augmentation.py.

5.1.4 Converting video files to images

As we require a wide variety of datasets to train our models, a script was written to simplify the creation of data from video. The script requires a set of different arguments to function properly. The arguments must be passed to the script in the following order:

- videoSourcePath - Path where the videos to extract images from are located
- directoryName - Defines the directory where extracted images are saved
- imgSize - Specifies the size of each extracted image

The script can then be executed in the following way:

```
python capture.py <datapath> <outputpath> <imgsize>
```

The script is mainly composed of two parts, where the first part is extracting frames from video data and organizing the frames into labeled folders. The second part generates a training dataset using the labeled folders, generated by the scripts first part. For every video found in the passed in datapath argument, the script will iterate over each video and start frame extraction. The frame extraction will for every frame of the video, take a picture and label it with the correct naming format required for the dataset creation. The naming of each frame is as follows:

```
videoFileName + '-frame-' + str(currentFrame) + '-bW.jpg'
```

Where currentFrame refers to the index of the frame used.

Once the image has been labeled it is processed, making it easier for the model to process. This is done by converting the original color of the image to grayscale, as well as setting the image size to be the specified argument passed to the script. After processing the image, its pixel values are added to the main dataset. This process is done for all frames extracted from the videos, which will in the end result in a list containing the pixel values as features, and label for each individual frame extracted.

Once all frames are extracted and the list is created, the second part of the script will begin. This part randomly shuffles the list of images, note that this is to create a randomized ordering, thus the pixel values of each image represented in the list are unchanged. Once done the script generates an X & Y pickle file. The script will then iterate over each feature and label in the list, appending the features to the X pickle and labels to the Y pickle. We then reshape the X pickle, giving it a different shape to simplify the training data, without modifying any of its data and finally saving both pickles to the output path specified as an argument when running the script.

5.1.5 Labeling the datasets

After processing the images for the given dataset, they have to be labeled correctly into their respective categories. To do this we wrote a labeling script which can be seen in Appendix F. The script will iterate over a specified folder and label all the images to their corresponding category. For the script to work it is important that the name of the folder is the same as the category of the images it contains. The script will export two pickle files. One will contain all the labels for the images and the other one is a numpy array containing all the image information. These files can then later be loaded into a project to train or validate the model. This labeling script is specifically used for processing our greenscreen datasets. To see the implementation go to Appendix F, label.py.

5.1.6 Loading the datasets

Before the model can be trained, all the data should be normalized so that the pixel values are between 0 and 1. The data also has to be loaded into the project so that the model can be trained or tested. Therefore, the loading functions used in this project load the datasets from its pickle files and then normalizes the data by dividing all the pixel values by 255. To see how this has been implemented, see Appendix G, dataset.py.

5.2 Model

This section covers the implementation of each model that was created for this project.

5.2.1 Conv-Model 1

Conv-Model 1 consisted of multiple convolutional (Subsection 3.2.2) model architectures that were tested and finally discarded because of its bad results in training validation. All of the architectures that were tested were trained on each of the five-arm gestures with dataset 1, but all produced the same exact result, not being able to get accuracy above 22% while training. When this was discovered the model was scrapped.

5.2.2 McQueen Fully Connected

McQueen Fully Connected is the 2^{nd} model that was created. This came to be because of the trouble we encountered by using a convolution neural network the first time. The new model showed great potential very quickly, with decent results in training and validation (Figure 5.4(a)), which seemed to be a decent amount of loss (Figure 5.4(b)).

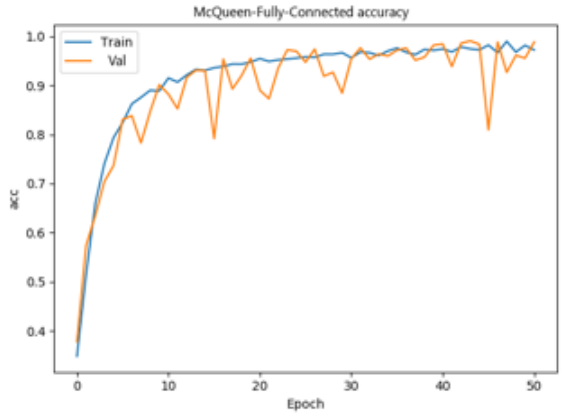
The architecture of the fully connected model is described below:

- Input layer: I_0 with 512 neurons and ReLU as its activation function.
- Dense layer: D_0 and D_1 with a sizes of 1024 and 2048 with ReLU as their activation function.
- Output layer: O_0 with a size of 5 and Softmax as its activation function.

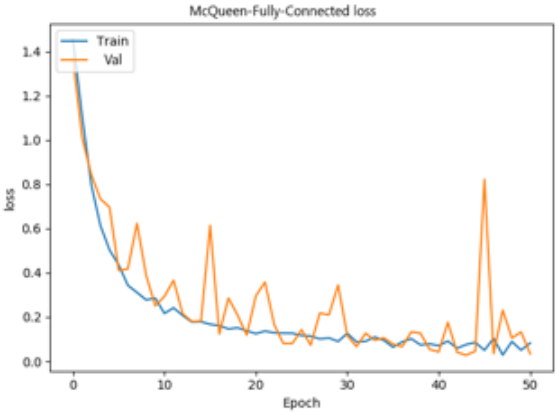
The layers are connected in the order: I_0 - D_0 - D_1 - O_0 where all the layers are fully connected.

The optimizer used in this model was ADAM (Subsection 3.2.7), with the default values set in Keras (Subsection 2.3.4). Callback for early stopping was also applied with patience set to 4 epochs while it was monitoring the validation loss. The code for generating this model can be seen in Appendix H, create_mcqueen_fully_connected.py.

A batch prediction on testset 1 (Subsection 5.1.1) resulted in the accuracy peaking at 50,3%. We collectively decided not to pursue a fully connected model after we got the test prediction results from the model.



(a) McQueen Fully Connected training and validation accuracy.



(b) McQueen Fully Connected training and validation loss.

Figure 5.4: Training and validation graph of McQueen Fully Connected model. This model was initially set for 100 epochs, but the early stop callback kicked in because of no further improvements in validation loss where made in the last 50 epochs.

5.2.3 Lightning McQueen CNN

Lightning McQueen CNN is the final iteration of models we have created, Phase One, Phase Two, and Phase Three. The 3rd phase turned out to be the model that gave the best results in both training and testing.

Phase One

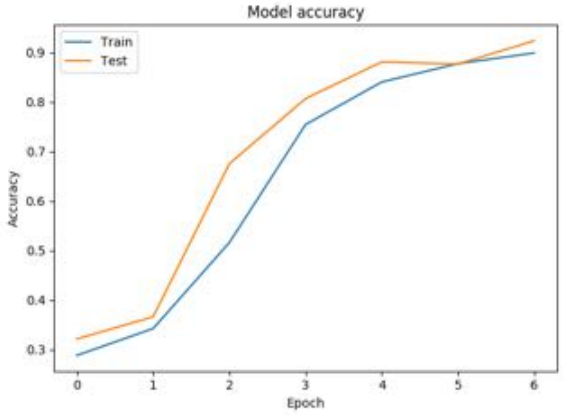
Lightning McQueen Phase One was the model where we had our breakthrough with a CNN model. It showed promising results in the training phase as seen in Figure 5.5.

Lightning McQueen Phase One was trained on our initial datasets (Subsection 5.1.1). However, when retraining, the results were not sufficient. The model performed especially bad when testing with testset1 (Subsection 5.1.1). The structure of the Lightning McQueen Phase One model:

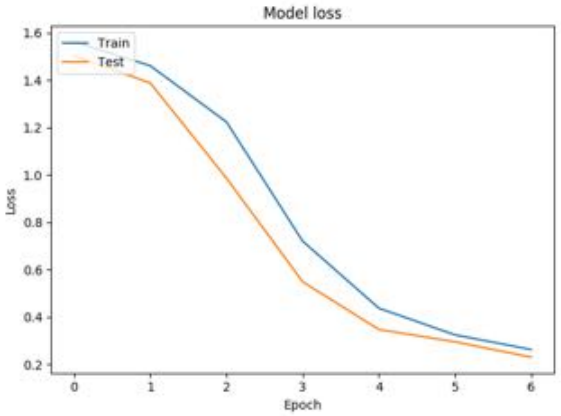
- Input layer I_0 with 32 neurons, kernel size of 5x5 and ReLU as its activation function.
- Two convolutional layers C_0 and C_1 with 64 and 128 neurons, with both having a kernel size of 5x5 with ReLU as their activation function.
- Three max-pooling layers P_0, P_1 and P_2 with a pool size of 2x2.
- One dense D_0 layer with 50 neurons and ReLU as its activation function.
- Output layer O_0 with 5 neurons and Softmax as its activation function.

The layers are connected in the order: I_0 - P_0 - C_0 - P_1 - C_1 - P_2 - D_0 - O_0, where D_0 and O_0 is fully connected.

The optimizer used for this model was ADAM (Subsection 3.2.7), with the default values that are set in Keras (Subsection 2.3.4). The callback for early stopping was also applied with patience set to 2 epochs while it was monitoring the validation loss. The code for generating this model can be seen in Appendix I, create_mcqueen_phase_one.py.



(a) Phase One training and validation accuracy.



(b) Phase One training and validation loss.

Figure 5.5: Accuracy and loss graphs for the 1st training of Lightning McQueen Phase One, showed good potential in training.

Phase Two

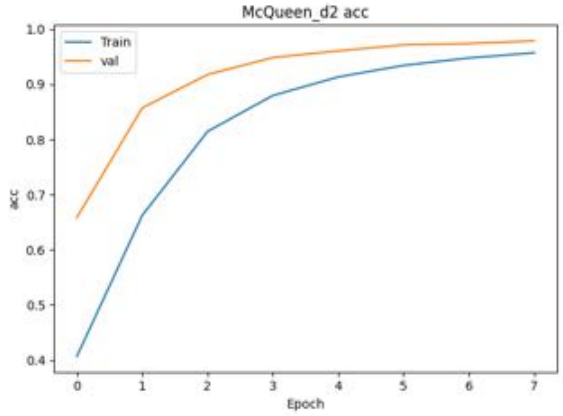
Lightning McQueen Phase Two was our first successful CNN model, both in training and in testing. It was trained and tested on our greenscreen datasets. Figure 5.6 shows the 3rd training of Phase two.

The structure of the Lightning McQueen Phase Two model:

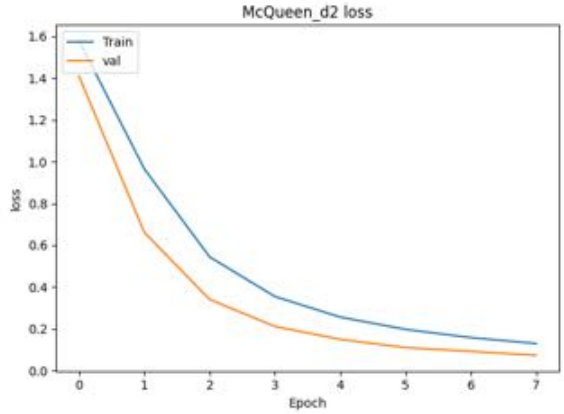
- Input layer I_0 with 20 neurons, kernel size of 5x5 and ReLU as its activation function.
- Two convolutional layers C_0 and C_1 with 32 and 40 neurons, with both having a kernel size of 5x5 with ReLU as their activation function.
- Three max-pooling layers P_0, P_1 and P_2 with a pool size of 2x2.
- Two dense D_0, D_1 layer with 900 and 350 neurons, respectively, with ReLU as its activation function.
- Output layer O_0 with 6 neurons and Softmax as its activation function.
- Four dropout layers Do_0, Do_1, Do_2 and Do_3 with 70, 50, 25 and 50 percent.

The layers are connected in the order: I_0 - P_0 - Do_0- C_0 - P_1- Do_1 - C_1 - P_2 - Do_2 - D_0 - Do_3 - D_0 - O_0, where D_0, D_1 and O_0 is fully connected.

The optimizer that was used for this model was SGD (Subsection 3.2.7), with a learning rate of 0.001. The callback for early stopping was also applied with patience set to 3 epochs while it was monitoring the validation loss. The code for generating this model can be seen in Appendix J, create_mcqueen_phase_two.py.



(a) Phase Two training and validation accuracy.



(b) Phase Two training and validation loss.

Figure 5.6: Accuracy and loss graphs for the 3rd training of Lightning McQueen Phase Two, training is plotted with dropout active.

Phase Three

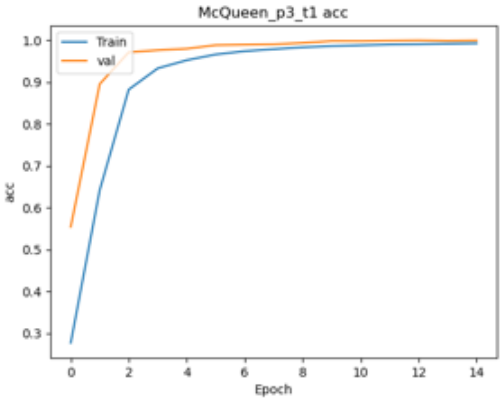
Lightning McQueen Phase Three is the final and most promising model we made. Phase Three is our best performing model in training and in validation as seen in Figure 5.7.

The structure of the Lightning McQueen Phase Three model:

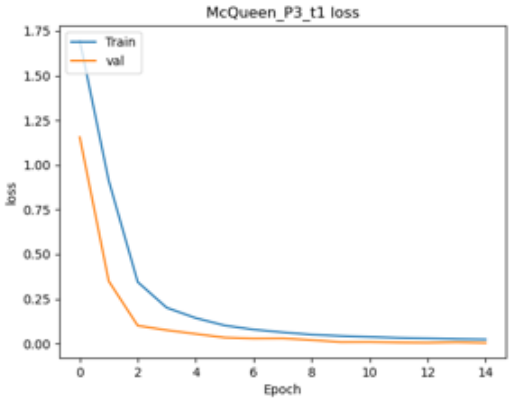
- Input layer I_0 with 20 neurons, kernel size of 5x5 and ReLU as its activation function.
- Two convolutional layers C_0 and C_1 with 32 and 40 neurons respectively, with both having a kernel size of 5x5 with ReLU as their activation function.
- Three max-pooling layers P_0, P_1 and P_2 with a pool size of 2x2.
- Two dense D_0, D_1 layer with a size of 700 and 350 neurons, with ReLU as its activation function.
- Output layer O_0 with 6 neurons and Softmax as its activation function.
- Two dropout layers Do_0, Do_1 with 70 and 30 percent.

The layers are connected in the order: I_0 - P_0 - C_0 - P_1 - C_1 - P_2 - Do_0 - D_0 - Do_1 - D_0 - O_0, where D_0, D_1 and O_0 is fully connected.

The optimizer that was used for this model was SGD (Subsection 3.2.7), with the learning rate initially set to start at 0.001 and decay over set to 1e-6. The callback for early stopping was also applied with patience set to 2 epochs while it was monitoring the validation loss. The code for generating this model can be seen in Appendix K, create_mcqueen_phase_three.py.



(a) Phase Three training and validation accuracy.



(b) Phase Three training and validation loss.

Figure 5.7: Traning and validation accuracy and loss graphs for the 1st training 15 epochs of Lightning McQueen Phase Three, training is plotted with dropout active.

5.2.4 Plotting the training history

To be able to look at how the model performed in training, plotting the training history is a great tool. Keras (Subsection 2.3.4) has this mostly handled for us, as the function call of training a model with Keras returns the training history. Because of this, we are able to use matplotlib to quickly plot all the data points from the training. An example of the results can be seen in Figure 5.7. By looking at the graphs we can get an indicator of how the training went and if the model is over or underfitting. The code implementation for plotting can be seen in Appendix O,

plot_training.py.

5.2.5 Retraining models

To get an accurate model, it has to be trained multiple times on multiple datasets of varying complexity. Therefore, we created a script to load our previously trained model and retrain it to adjust the current model. Early stopping was included when the model was retrained, this stops the training session if there is no progress. The history data points from the training session are returned to later plot a graph. The script can be seen in Appendix L, train_mcqueen.py.

5.2.6 Testing the models

To make sure that a model is performing well, it has to be tested. A good tool to utilize for testing a model is a confusion matrix, which displays the overall performance of a model. An example of a confusion matrix can be seen in Figure 6.1. To create the matrix, the model is tested on a testset and each prediction is checked against the correct value for the image prediction. The implementation of the confusion matrix can be seen in Appendix N, ConfusionMatrix.py, and the implementation for testing the model can be seen in Appendix M, test_mcqueen.py.

5.3 TurtleBot3 Waffle Pi

This section will cover how the TurtleBot was set up to prepare it for integration with the machine learning model.

5.3.1 Configuring the TurtleBot with remote computer

In order to connect the remote computer, also referred to as the ROS master to the TurtleBot, it needs to know the IP-address of the remote computer. In order to accomplish this, we need to add two variables in the .bashrc configuration file on the TurtleBot. We can achieve this by either manually add them to the TurtleBot, or utilize a technology called SSH as seen in Appendix C.3. SSH will allow us to remotely connect to the TurtleBot and control it from any other computer as long as we know the authentication credentials of the TurtleBot.

Once the command is executed, it will prompt us for the username and password for the root user of the TurtleBot. Simply enter these credentials and if correct, we can control the bot remotely. If any problems occur with SSH, it could help to check the IP-address of the TurtleBot manually with a monitor and keyboard to ensure SSH is allowed and the IP is correct. Once connected, the file /.bashrc must be updated with two variables. VRPN must be installed and executed to enable the remote computer and TurtleBot to transfer data between the two units.

5.3.2 Developing movement

There are two options for controlling the TurtleBot using a Python script, either using the shell for passing in ROS commands or by integrating the rospy library and use built-in functions. We chose the former after having tried setting up a package containing python scripts using the rospy library. Setting up a package like this turned out to be a lot of struggle since there was so much that had to be configured for it to work. Based on simplicity in the development of the prototype, we decided to use commands for the shell to move the TurtleBot.

Constructing the functionality for movement started with building a Python script that executes commands based on the arguments sent to the script. This argument holds the value of the predicted gesture from the prediction script. The movement script (Appendix R) handles the argument using a dictionary containing indices from 0-5. These indices hold the functions to move left, right, forward, backward, stop, and still, in that order. The argument will help pick out the correct function. There will be returned two arrays from the chosen function, one for linear and one for angular directions. The arrays contain the values of x, y, and z, therefore the usage of linear arrays should be with x to move forward. While the angular arrays should be used with both x and z to move forward and to the right or left. After finding the values for x and z to move forward, backward, left and right, we started tweaking the amount of movement for each command. The values in the linear and angular arrays are converted to string values and concatenated, before being concatenated with the rostopic command for `geometry_msgs/Twist`. The command is executed using a subprocess that is using the shell. A command that is passed to the shell looks like the example below, this is for moving backward.

```
rostopic pub /cmd_vel geometry_msgs/Twist -- [-1, 0, 0] [0, 0, 0]
```

The movement script (Appendix R) is part of a repository that is developed and executed on the remote computer. The reason for this decision was purely based on performance, which will be discussed in Subsection 6.3.4.

5.4 Integration of Neural Network with TurtleBot

After developing and testing the movement script, we started to integrate the model on the remote computer. To be able to execute the functionality of the prototype properly, a virtual environment had to be configured, as well as developing an API that could handle the communication between the scripts on the TurtleBot and the remote computer.

5.4.1 Virtual environment

A Python application will often use packages that are not a part of the standard Python library. At times an application might need a specific version of a package due to some package versions causing issues and bugs in said application. The way to work around this is the use of a virtual environment. A virtual environment is a self-contained directory tree, a developer can specify the Python version and the necessary packages to be installed[1]. To see how to set up a virtual environment see Appendix A.3.

[1] https://docs.python.org/3/tutorial/venv.html

5.4.2 API

In order to upload images to the remote computer, an API was developed. The API consists of multiple script files, that individually perform an important task. A complete flow showing how the TurtleBot uploads images to the API, and how the API further calls the different scripts can be seen in Figure 5.8.

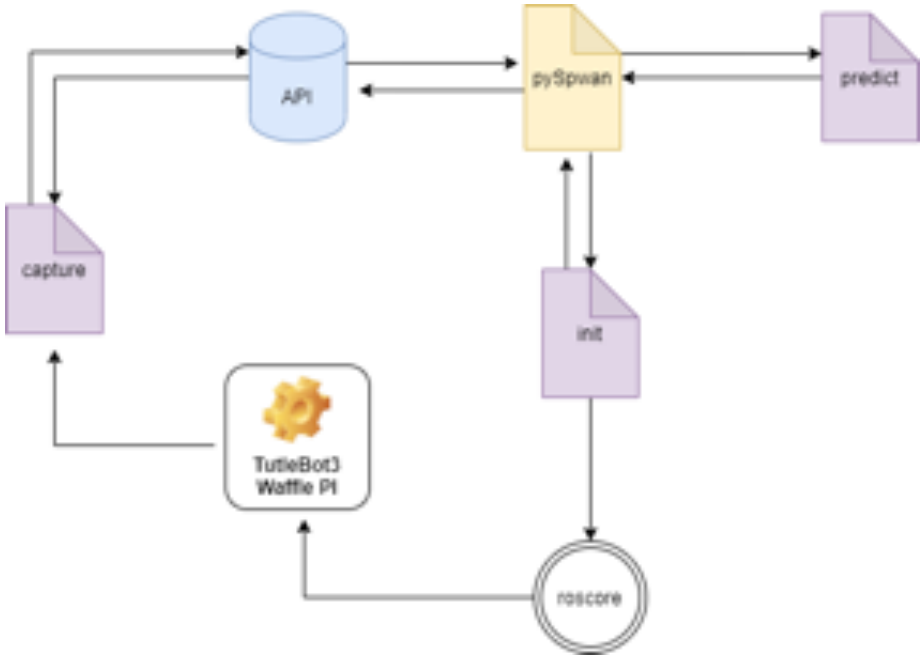


Figure 5.8: Flow diagram representing the current system.

- index - Starts running the API on port 3000
- upload - Configure file upload options
- router - Processes request sent to server
- pySpawn - Spawn child processes to run python scripts

The API runs on the remote computer. By starting up the index file of the API (Appendix T), the server will begin to listen for requests to the route 127.0.0.1:3000/. It is configured to only allow POST requests to be sent to the route, it is also enforced to contain a valid image within the request body. Any request that does not fulfill the rules mentioned above will result in a failed request with a status code of 400 as seen in the router file (Appendix V). Whenever a request is successfully sent to the route, the image contained in the body will be saved in a folder named ”directions” with the file name direction.jpg. Note that there will at any time only be one image in the directions folder due to the fact that the image is overwritten every time a new image is posted. Once the image has been uploaded to the folder, the route will call on the pySpawn script (Appendix S) that creates a new child process to run the prediction script (Appendix Q), the script is responsible for loading the model and predict the gesture from the uploaded image. Once the child process running the prediction script receives a message back with the predicted direction index, pySpawn will terminate the running child process and initialize a new process to run the movement script (Appendix R). This script is responsible for executing the movement on the TurtleBot, and finally ending the POST request with a 200 response code, meaning the action was successfully handled.

5.4.3 Prediction script

The model is used in a script called predict (Appendix Q), this script is opened as a process through the API that was set up to communicate between the TurtleBot and the remote computer. The file is a grayscaled image of size 100X100 pixels, this is captured through the camera_capture script (Appendix P) that will be described in Subsection 5.4.6. After having accessed the image through the API, the script calls on a function to predict the gesture in the image. The predict function takes two parameters, the first parameter takes the path where the model can be accessed. The second parameter is the path to where the image was stored after being handled by the API, it was more efficient to send the string of the image path to the function and fetch it when needed, compared to sending an image through multiple functions. The predict function finds the correct model and predicts the gesture category on a prepared image using `prep_img()`, which takes the image path as an argument. The prepare image function fetches the image in grayscale, reshapes it to a numpy array with normalized pixel values between 0 and 1. Since the model has been trained on normalized pixel values, it can only predict accurately with equally formatted data.

Now that the data is correctly formatted, it's then used by the model to predict what signal is given. The script uses Keras' `predict_classes()` function which returns a real number from 0 and 5, which all correspond to a label that we have trained the model with. For example, the number 3 corresponds to backward, the number 4 corresponds to stop.

5.4.4 Environment

To decide which computer to run the model on, we had to time the execution of the prediction script on both the Raspberry Pi and the remote computer. When executing the prediction script on the TurtleBot we used the TensorFlow framework, this decision was made after the configuration of PlaidML failed. When the neural network tried to make predictions on the Raspberry Pi it depleted the available memory, causing it to work with reduced capacity. Whereas the remote computer kept a stable execution time and made predictions at a far better rate than the Raspberry Pi. The results from this test led us to set up the API on the remote computer.

5.4.5 Controlling TurtleBot with predicted gesture

After setting up the neural network to run properly on the remote computer, the connection to the movement script could be configured. Through this subsection, you will get an insight into the process for executing the movement script with the prediction sent from the prediction script through the API.

Moving the TurtleBot

When the movement script (Appendix R) starts executing, it runs a function that calls on `move()` and sends the predicted value to a function containing a dictionary that selects the movement type based on the string value of the prediction. When the correct movement type is chosen from the list, it executes the command through a subprocess that opens a shell. The process runs for one second before it is terminated, giving it enough time to execute the movement. Using `sleep()` to halt the system while the subprocess is executing might result in an action not being performed.

Described in Subsection 5.3.2, we can see the logic for how the movement script executes the commands. The method for executing could have been improved by using the rospy library. This would allow for calling the movements directly through rospy functions rather than opening a new process for executing the movement. This integration would reduce the response time from calling a movement by calling it directly through cmd_vel using rostopic and geometry Twist messages. The error handling could be improved by logging the time and reason for any occurring errors.

5.4.6 Handling image sequences for gesture recognition

Throughout this subsection, a breakdown of the script that captures images used for gesture recognition will be given. The script for capturing images and transmitting it to the remote computer can be found in Appendix P.

Capturing the image

As stated in Subsection 5.4.5, the process of making a prediction and then execute the movement command was time exhausting. We had to consider the time for the process of executing one movement when developing the capture script. The initial plan was to handle video streams with a goal of predicting multiple frames per second, this setup would have led to a fast responsive control system for the TurtleBot. The PiCamera package for Python was a viable option for the system we created. When using this package together with `sleep()` we can decide the number of frames to capture in a given time period. The `sleep()` function is currently waiting for two seconds in between every iteration, this is enough time to send the image through the API, predict the gesture, and execute the movement script. The PiCamera package includes functions for configuring the resolution of the image, the width and height is set to 1000x1000 pixels. Our neural network is trained with images of size 100x100, we wanted the image to already be a square before processing it to keep all the dimensions in the image in their relative sizes.

Image preparations

After having successfully captured an image, the process of preparing the image starts executing. The prepare function takes the string of the captured image path as a parameter. The pixel values for the image are fetched using the function `cv2.imread()` by sending in the image path of the captured image and adding a flag for grayscale image. The cv2 package handles the images as a matrix containing pixel values, since this is grayscaled it is a 2D-array[2]. Using the matrix, the `cv2.reshape()` function can reshape it to the desired size of 100x100 pixels, before returning the matrix. Since the image has to be sent through an API to make a prediction, the file size had to be as small as possible before making the API call to decrease the time for transmitting the file. To finish the process of resizing the image, the `cv2.imwrite()` function has to be used to store the actual image before using it with the API. Storing the image has to happen successfully to access the functionality of the API.

[2] https://pythonexamples.org/python-opencv-read-image-cv2-imread/

Making API calls

If the resized image is successfully written to the directory, the process of making a post request to the API begins. The built-in `open()` function in Python fetches the image from the directory of resized images. This image will be added as a file object in the POST request body that is being sent to the API. The API returns a response containing a status code before the connection to the API closes. The status code from the response could be used for error handling and logging.

Chapter 6

Testing and Evaluation

This chapter covers the testing and evaluation of the machine learning models that were created for the project, and the reliability in conjunction with the TurtleBot3 Waffle Pi. The chapter starts with Section 6.1 which covers the evaluation of all the models and datasets that were made for the project. Section 6.2 covers the reliability testing of the model and TurtleBot3 in different environments. The last Section 6.3 covers the evaluation of the components that are used to connect the TurtleBot3 and the neural network.

6.1 Model Evaluation

This section is an evaluation of all the models and datasets that were created for this project.

6.1.1 McQueen Fully Connected

McQueen Fully Connected was one of our first attempts at creating a machine learning model for this project. When we started we did not know too much about what we were doing and in hindsight, we could have modified the architecture of the model to obtain better results.

In training we quickly found that there was something noteworthy going on, as seen in all the spikes in the training graphs in Figure 5.4. In testing, this model did not perform well, which is understandable as the dataset that was used did not contain enough data to generalize for new data. As seen in Table 6.2, the accuracy when testing was not very impressive. The result was reasonable and encouraged further development since this model was a part of our initial exploration with machine learning models.

6.1.2 Lightning McQueen CNN

Phase One

Phase one was the first step to a successful CNN model we had, even though the testing result was not what we had anticipated, it showed us that it was possible to build a CNN model for our project. It was trained on our initial datasets (Subsection 5.1.1) and got a fairly good training accuracy. When we started testing the model, we noticed it did not perform as well as we wanted to. As seen in Table 6.2 Phase One got a 50.3% accuracy on the testsets.

Phase Two

Phase Two is the model we thought were going to be used in our prototype, it had great results in training and very promising results in testing compared to the earlier models. The results can be seen in the confusion matrix below (Figure 6.1). A problem that occurred with Phase Two was in testing, as the accuracy did not rise above 98% on any of the testsets.

	forward	right	left	backward	stop	still	accuracy
forward	76470	0	0	2816	0	0	96.4%
right	0	79913	0	0	0	0	100.0%
left	0	0	79219	0	67	0	99.9%
backward	2666	0	0	77285	0	0	96.7%
stop	0	0	0	0	79286	0	100.0%
still	39	696	24	232	2837	75951	95.2%
Precision	96.6%	99.1%	100.0%	96.2%	96.5%	100.0%	98.0%

Figure 6.1: Confusion matrix after the 10th training of Lightning McQueen Phase Two, showing the accuracy and precision of each signal prediction. These predictions were executed on greenscreen testset 1.

As seen in the confusion matrix (Figure 6.1), the accuracy is good, however, the precision was not good enough. The problem with precision between the signs for forward and backward stems from how similar they look. This will be discussed in Subsection 6.1.3. We did try to train Phase Two more afterward, this had no impact on the results shown in Figure 6.1.

Phase Three

Phase Three is the last and most promising iteration of the Lightning McQueen CNN model. This version performs very well both in testing and training. Looking at the confusion matrix in Figure 6.2, we can see that the accuracy and precision is high. However, there are some incorrect predictions made by the model. This is something that can be expected due to the resemblance between the signals for forward and backward. When comparing Phase Two (Figure 6.1) against Phase Three (Figure 6.2) we can see how the latter performs better than its predecessor while being tested on the same amount of data.

	forward	right	left	backward	stop	still	accuracy
forward	79286	0	0	0	0	0	100.0%
right	0	79286	0	0	0	0	100.0%
left	0	0	79286	0	0	0	100.0%
backward	2	0	0	79284	0	0	100.0%
stop	0	0	0	0	79286	0	100.0%
still	0	0	0	400	0	79220	99.5%
Precision	100.0%	100.0%	100.0%	99.5%	100.0%	100.0%	99.9%

Figure 6.2: Confusion matrix after the 11th training of Lightning McQueen Phase Three, showing the accuracy and precision of each signal prediction. These predictions are done on greenscreen testset 2.

Looking at the confusion matrix above (Figure 6.2), you can see that two images signaling forward were predicted as backward. As previously mentioned, this might be caused by how much the gestures resemble each other. Additionally, 400 still gestures were categorized as backward. This might be due to the still images resembling backward. It might also be because the model has not been trained on enough incorrect data, seeing as there are an extreme amount of different ways an incorrect gesture could be presented.

6.1.3 Evaluation of datasets

When developing a machine learning model it needs to have good data to learn from. This means that an evaluation of the datasets is necessary before any conclusions are drawn about a model's performance.

Initial datasets

The initial datasets had a couple of flaws that were discovered when the model did not perform as well as we hoped. These datasets only contained five gestures, one for each action. However, what we had not taken into account was how the system should behave if it sees a gesture that does not correspond to one of the predefined categories.

Additionally, the first batch of datasets did not have the desired quality we aimed for. We realized it would be hard to get an accurate model with these sets of images. Another problem we discovered with the process that manufactured the initial datasets was how we categorized the different images. With this method, both images where the subject is facing the camera and images where the subject's back is turned were all accumulated. These should have been separated and handled as two different gestures.

Greenscreen datasets

The greenscreen datasets are significantly better than the initial datasets, but they also have their flaws. These datasets are made with the same person in every image and the data is very strict in such a way that the correct signal is perfectly held every time.

For the greenscreen dataset, we added the category still. This category was made to handle invalid gestures. The greenscreen data is definitely cleaner and should in theory be better than the initial datasets. In these datasets, we also eliminated the problem of which way the person in the image is facing, as the datasets only contain images of the subject facing the camera.

6.1.4 Overall comparison of all models

When looking at Table 6.2, the goal is to determine which model should be used in the prototype. Quickly it can be seen that McQueen Phase Two and Three are the two most successful models. The models from phase two and three both performed well in testing, while also being trained to take wrong gestures into account, thus, making one of these the most viable option for the prototype.

Phase Two and Three, as previously mentioned was trained on our greenscreen datasets (Subsection 5.1.2), this means that the models have been trained on a wider range of complex data. The model from phase two was trained on five greenscreen datasets, while its predecessor was trained on eight greenscreen datasets.

Model	Datasets	Dataset type	Gestures	Epochs trained	Training accuracy	Testing accuracy
Conv-Model 1	2	Initial datasets	5	66	22%	22%
McQueen-fully connected	2	Initial datasets	5	87	98%	50,3%
McQueen-phase one	3	Initial datasets	5	50	94%	50,4%
McQueen-phase two	5	Greenscreen datasets	6	61	99,4%	98.0%
McQueen-phase three	8	Greenscreen datasets	6	101	99,9%	99.9%

Table 6.2: This table is a comparison of all the models that were made for this project.

6.1.5 Evaluation of results

The models' initial accuracy test was done with images from our greenscreen testset (Subsection 5.1.2). On single image predictions, the results were as expected when looking at Table 6.2. Phase Three managed to outperform Phase Two by a good margin. The testing of Phase Three with the greenscreen datasets showed the same as seen in the confusion matrix (Figure 6.2). Some of the backward signals were classified as still, which means that the TurtleBot3 would not move. The results shown in the confusion matrix looked promising, thus McQueen Phase Three was the model we chose to use in our prototype.

6.2 Reliability Testing

The COVID-19 pandemic made testing more challenging. Due to the pandemic, reliability testing of the model and the TurtleBot3 in different environments has been limited to our own student dorms.

There have been varying results during testing, which could be caused by several factors. The datasets are made with a person of average height and build, which seems to have affected the prediction results. Two of the group members are above average height, so when these members have tested the prototype the results of the model's predictions have not been as accurate as the initial testing would suggest. When tested on a person closer to average height and build, but not the same person as is in the datasets, the accuracy of the predictions improved. The model manages to recognize the signals for forward, right, and backward, in some cases. We have not been able to run live tests with the person that is in our datasets, because of living situations as a result of the pandemic. However, when testing with images sent from the person in the datasets the accuracy did not surpass 20%, while peaking at 31,7% and a loss value of 23,5 when testing with data of another group member. This result shows that the model has not been able to generalize enough from the training data. We were not able to test in the location where the original images of the training data were captured due to the COVID-19 pandemic. Therefore, we are not able to draw a conclusion if the prototype would perform better in that environment.

6.3 TurtleBot3 Waffle Pi Components and Functionality

This section covers the evaluation of the components that are used to connect the TurtleBot3 and the neural network.

6.3.1 Capture script

The development of the capture script (Appendix P) was based around the performance of the API, prediction script, and the process of moving the TurtleBot. Since this process spends about two seconds to execute (Subsection 5.4.6), the capture script does not need to send more than one image every two seconds. Sending more images withing that time period would have been redundant for the system that was developed. Sending more frames could also lead to gestures being ignored because the previous frame is overwritten by the newest captured image when reaching the API. This setup was viable to get a proof of concept even though the responsiveness is slow.

The script can be easily modified to send more frames within a given time period. Configuring the time for `sleep()` can be done based on the performance of the prototype, by changing the argument to a smaller number. The number can be a float, meaning that we can iterate multiple times each second. Instead of using `sleep()` as a condition for waiting, we could utilize the responses from the API. By using the responses from the API we could send a new image straight after getting a confirmation that the previous image has been properly handled.

The `capture()` function in the PiCamera package stores an image file when executed. This has the potential of exceeding the storage limit on the TurtleBot or the Raspberry Pi, if the prototype is tested for a long period of time without errors. This is solvable by using the `capture_continuously()` function in the package[1]. This function iterates through every frame of a video stream and can store the images in an in-memory stream instead of on disk. There is also an option to implement a flush function for removing the images in a specified directory.

There is currently no error handling in the capture script, other than a counter for the amount of failed `cv2.imwrite()` function calls and if the response from the API is anything other than 200. This is not a viable solution for handling errors in a more developed prototype. Both of these could be used separately for handling errors and logged the type of error with suitable data.

6.3.2 API

The API (Appendix T) was developed to handle uploading images to the remote computer (Subsection 5.4.2), captured by the TurtleBot. This had to be done in order to improve the performance of the model. The hardware on the TurtleBot caused the model to spend a long time processing, mainly due to the fact that the model is utilizing the Raspberry Pi's CPU, rather than GPU. Developed as a REST service, we can utilize this technology to easily send data from the TurtleBot over to the remote computer, where the API is running.

[1] https://picamera.readthedocs.io/en/release-1.10/recipes1.html

There are a few potential improvements that could be made to the service. Since the API runs locally on a network where only the TurtleBot and remote computer are connected, the security and error handling is minimal. In theory, anyone connected to this private network could send images to the API and cause the bot to move.

The performance of the API is generally fast. Since both the TurtleBot and the remote computer running the service are connected to the same local network, the requests posted to the API, containing the captured image are resolved within 100ms. Saving the uploaded image is done directly on the computer's file system, storing it locally in the same directory as the API is hosted, resulting in a generally quick save time.

6.3.3 Prediction script

The prediction script (Appendix Q) is executed by the API (Subsection 5.4.3) when it has finished storing the captured image sent from the TurtleBot. This script needs to load all the necessary packages and the model before making a prediction on the image. Loading this slows the process down and makes the prototype less responsive. This script should run continuously and listen to messages from the API. By using the Python package socket[2], we could create a connection between the API and the prediction script. With this , we can communicate to the prediction script that an image is ready to be processed and predicted, and respond with the prediction to the API.

Using sockets could also improve the communication between the prediction script and the movement process. The current script is returning the predicted value to the API, then the API opens the movement script (Appendix R). This process could be improved by continuously running the movement script, then send the predicted value directly to the script by using sockets.

The model was set to run on the remote computer after having tested the model on the TurtleBot. The CPU and RAM on the Raspberry Pi (Subsection 2.4.1) are less powerful than the remote computer, which has up to 3.0 GHz dual-core CPU and 8GB RAM. This is an important factor for the time it takes to process the entirety of the system's functionality. Not being able to predict the images directly on the TurtleBot opens up the issue of transmitting files efficiently. In our prototype, the loss in efficiency when transmitting a file was less than the loss in efficiency when running the model on the Raspberry Pi. This set up made the prototype more responsive, thus making it viable for us to use. In a more developed system, the loss when transmitting files might be greater considering access to high-speed Ethernet might not be optional. There are also security risks involved when predictions are handled on a remote computer instead of predicting directly on the actual robot.

The model for predicting a gesture that is used for the prototype is trained on the augmented dataset (Subsection 5.1.2), issues with this dataset are discussed in Subsection 7.1.3. To give a preview, there is only one captured image for every gesture, this gesture is done with a static posture that is relatively perfect in execution. When testing the model with a live feed of gestures taken with the capture script, the model performs poorly. This leads to a poorly performing prototype, making it really difficult to control the TurtleBot using gestures. From our testing, we have figured that the posture is really important for making correct predictions, and the predictions are more accurate with a person of average height.

The functioning prediction script has not implemented any error handling or logging. To better improve the system it is necessary to handle the errors for every function and communication between the scripts.

[2] https://docs.python.org/3/library/socket.html

6.3.4 Movement

The movement script (Appendix R) is a some what time exhausting process with poor control over the executed commands. This script is viable for the prototype but needs to be further developed for the TurtleBot to be responsive enough to control it with a proper flow. A proper flow would be continuously listening after predictions while running the correct movement until a new prediction is made or an error has occurred. The current implementation waits for a prediction to be sent from the API (Subsection 5.3.2), then it finds the correct action in a Python directory, and executes the movement through a subprocess that publishes a rostopic from the command line. The `sleep()` function gets called each time a subprocess is created, this is to give the process a time frame before it is terminated. This logic is just an assumption of how long a process runs. Communication between scripts with sockets is a necessity to improve the responsiveness.

The rospy package could help improve the responsiveness, this could be done by using the topics in ROS. Topics can be used for passing messages between publisher and subscriber. The movement script could publish a topic for moving the TurtleBot through the integrated functionality in rospy that communicates directly with the TurtleBot. Publishers and subscribers are asynchronous[3], meaning that the movement script can execute an action while listening to new predictions. This set up could improve the responsiveness by always executing movements and predictions without having to wait for processes to finish.

Error handling is minimal in the current solution. This could be improved using the connection and messages sent between publisher and subscriber. Using the exceptions that are raised when the script fails for logging would help to further improve the system by always knowing what happened when the script fails to execute properly.

6.4 Results

Testing each individual component of the prototype has gone well, however, we ran into some trouble when testing the complete prototype. Due to COVID-19 and other miscellaneous issues such as network connectivity, the prototype testing conditions has not been optimal. The tests we ran with the integrated system gave mediocre results. The results from testing the individual components however were better than what we had anticipated. A short summary of the tests will be given in the following Subsections 6.4.1 and 6.4.2.

6.4.1 Overall evaluation of model

We created four different models in the span of this project, one fully connected and three iterations of a CNN model. After training and testing all models the last iteration of our CNN models showed the most promise. Thus, we decided to proceed with Lightning McQueen Phase Three as our working model for the prototype.

Theoretically, the model is performing well with the test data that was created, we have not been able to see the same results in live testing with the prototype. We believe that this is due to the quality and strictness in the training data. This issue could be fixed by investing more time and resources to create better data. The model could also be improved by implementing a more advanced architecture and further tuning the hyperparameters.

[3] https://wiki.ros.org/rospy/Overview/Publishers%20and%20Subscribers

6.4.2 Overall evaluation of the prototype

The solution for the prototype is currently working based on the requirements that were set in Section 1.1. It can handle video streams and respond with appropriate actions based on predictions. The communication between components has been developed to create a working solution. The API used for sending frames from the TurtleBot to the remote computer is optimized based on the hardware we had available during the development process. The loss of time when transmitting a file is less than the loss of time when predicting a gesture on the TurtleBot compared to the remote computer. The process after the API call is made is where the bad responsiveness stems from. The prediction script (Appendix Q) is opened from the API when the frame is received, this means that every package and the model has to be loaded for every new frame. This process is exhaustive and will gradually fill up the physical memory, making the system less responsive. Making the actual prediction with the received frame is relatively fast on the remote computer, making it viable to do the predictions on a remote computer for the prototype if loading the packages and model happens just once. The prediction script returns the predicted value to the API that opens the movement script (Appendix R), this script handles the value of the prediction and opens a subprocess for executing the movement through the command line.

Chapter 7

Discussion

This chapter covers the discussions of the project. Section 7.1 goes over the goals we initially set for the project, the chapter finishes with a look at what can be implemented when looking at further development in Section 7.2.

7.1 Goals

In this section, we will discuss the goals that were set at the beginning of this project, in Subection 1.2.1. We will cover how each goal was reached and highlight specific choices that were made and how certain parts could have been improved.

7.1.1 Establish a proof of concept by laying the groundwork towards a possible solution for the problem described

Having analyzed and researched the different solutions that already exist when it comes to gesture recognition using real-time video-streams, we have come up with a lightweight solution that recognizes gestures using images. Our model is created using an architecture that has been tweaked over time and should be able to get good results in terms of accuracy within the bounds set by the training data. However, there are measures that can be taken to improve the current proof of concept solution.

7.1.2 Research whether there are existing solutions for similar gesture recognition

While researching similar solutions we struggled to find any with the same functionality and technology we were developing. However, that is not to say that there are none. We discovered that there is a lot of different possibilities when working towards posture/gesture recognition.

As seen in Subection 3.5.1, Thi-Lan Le and colleagues used skeleton tracking with the Kinect to achieve their results in posture recognition. Their solution uses fairly simple technology that could have been implemented in this project. However, it would require some changes in the hardware and development of this project.

We were also able to find more complicated solutions for posture recognition, mainly the patent held by Honda Motor Co Ltd which is described in Subsection 3.5.2. Their patent uses a more intricate system where most of the specifics are not discussed. However, their patented solution achieves what we are trying to do with this project.

There were no real challenges or setbacks reaching this goal as it was purely theoretical. Most problems we encountered during this phase of the project were related to inadequate knowledge regarding some topics that were heavily focused on papers.

7.1.3 Develop a functioning and accurate model

For us to be able to create a functioning prototype we had to create a machine learning model that could recognize and classify images, thus determining the TurtleBot's action. There were multiple necessary steps that had to be discussed to reach this goal.

The first step was picking the right frameworks to use (Section 4.3). Next was to create the data for training and testing (Section 5.1) and then proceed to create a model that could be trained, tested, and used in the final prototype (Section 5.2). This section will discuss the challenges that we faced while trying to develop a functioning and accurate model.

Datasets

While manufacturing datasets for the CNN model in phase one we were certain our method was fairly optimized and would produce a large amount of good data. However, we underestimated how important good data is for the model to be able to learn and generalize the data. It became clear to us that our data was not good enough. This realization should have come sooner as it would have given us more time to plan and prepare the new process of creating data using a greenscreen (Subsection 4.1.2). At the point when we realized we needed to change the methodology, there were little resources and we ended up using only one volunteer for the dataset. In each image with the greenscreen as the background, the volunteer held each gesture as close to perfect as possible. This left the system with very little leeway when it is given a gesture that does not conform with the standard in our dataset. This is something that could be fixed if we had more time. We would have to capture more images with the greenscreen in the background while the volunteer had some more variety in the gesture quality, without overlapping any other gestures or simply showing a wrong gesture.

During continuous development and testing of the models, it became more apparent to us the importance of good data for a model to train on to perform optimally. Thus, data gathering became a big part of the project. Since we resorted to taking a small number of pictures in front of a greenscreen instead of extracting thousands of frames from videos we were left with an underwhelming amount of data. However, this gave us the opportunity to utilize data augmentation. Luckily, the greenscreen was easy to mask out before performing the desired transformations to the image. This allowed us to manufacture and greatly expand datasets on demand. Due to the data augmentation transformations, we were also able to adjust the complexity of different batches in a single dataset. This was done by regulating the volunteer's size and position. This process could have been done without using the greenscreen but it would require a lot more work to extract the pixels belonging to the person.

Safety is a huge part of any system that utilizes autonomous vehicles, thus, we aimed to develop a model that was as accurate as possible. In the beginning, we thought the accuracy mainly was influenced by the architecture of the neural network. However, as a reoccurring realization, we understood that the data used for training the model is just as important. When starting to develop the system, our main focus was creating data for every category that would make the TurtleBot perform an action. By doing so we inadvertently excluded all gestures that the system should recognize as wrong, meaning the model should not try to categorize that gesture into one of the categories making the TurtleBot move. Overlooking this would raise some serious safety issues as the vehicle could end up making several unintentional maneuvers. However, this issue was addressed during the development of Lightning McQueen Phase Two when we decided to add a 6^{th} category to our datasets. This category would represent any gesture or signal that could be derived from moving between different gestures. Thus, minimizing the risk of categorizing a gesture that should not be processed as one of the five main categories for movement.

Although we spent a lot of time optimizing the data manufacturing process, there are still some issues left that are reflected in the results after testing the system on a live video-feed. There could be a couple of reasons for this, but after examining the datasets in comparison to the video-feed, we believe it is caused by the datasets only containing the same person showing a perfect version of the gesture. This may confuse the model as it is presented with a person it has never seen before. If the model had been trained with different people as subjects it would be more generalized and would hopefully focus more on the gesture rather than the person. As mentioned earlier, we were unable to gather data of more people due to a lack of resources because of the COVID-19 pandemic.

Choosing a network type

Initially, we wanted to use a CNN model because of the research we had done, along with recommendations from our advisors. As we had little success with our first CNN model, we started contemplating the usage of a fully connected neural network. The results from testing the fully connected NN, showed that the accuracy was still not high enough.

Conv-Model 1

As we were new to the field of machine learning, Conv-model 1 (Subsection 5.2.1) was our first attempt at creating a machine learning model. Conv-model 1 did not perform well at all. At the time we did not understand why, but we were advised to try a fully connected neural network (Subsection 3.2.1) instead. Therefore we scrapped Conv-model 1 and proceeded to McQueen Fully Connected.

McQueen fully connected

McQueen Fully Connected (Subsection 5.2.2) performed a lot better than what Conv-model 1 did, but we did not see a fully connected neural network as the model we wanted to use in our prototype. Therefore, we used McQueen Fully Connected as an exercise to learn more. After initial testing of McQueen Fully Connected, the model was quickly discarded to make way for the McQueen CNN Phase models.

Phase One

While continuously developing and testing we found that we were using the wrong activation function for our output layer. Previously we had used Sigmoid (Subsection 3.2.3) which really stagnated the performance of the model, this was confirmed by a 72% increase in accuracy after changing to Softmax (Subsection 3.2.3). At this stage, we had not yet addressed the issue of how the system should handle a wrong signal. Therefore, if a wrong gesture was given to the model it would try to classify it into one of the existing categories and thus make an unexpected action.

Phase Two

As stated above we had a significant increase in accuracy after changing the activation function, but the model still did not perform well in testing. We discussed the issue with our advisors while doing more research on our own. After evaluating the feedback and our most recent research, we came to the conclusion that adding dropout and changing the optimizer from ADAM to SGD could help solve the issue of poor test results. Changing the optimizer was one of the minor changes we had expected we might resort to, along with tuning other hyper-parameters. However, adding dropout was something we had not yet considered at this stage.

In this Phase There was also added a 6^{th} category to our set of possible predictions. This was done to address the issue when the model is given an incorrect signal. We realized the dangers proposed by not having any form of security mechanism in place to handle incorrect gestures. Along with adding another category we also improved the process of manufacturing datasets by using a greenscreen.

By making these improvements we got Phase Two (Subsection 5.2.3) to outperform Phase One (Subsection 5.2.3) by 47,6% accuracy in testing.

Phase Three

After training the Phase Two model (Subsection 5.2.3) we were unable to achieve any higher than 98% accuracy in testing. Even though the model from Phase Two performed really well we still wanted to improve it a little. This decision was made due to the ramifications of using a less accurate model. When using such a system you want, and expect it to be as accurate as possible to avoid damage or injury to personnel or anything of materialistic value.

While trying to improve the model we decided to tweak its hyper-parameters even more. This included decreasing the number of neurons in the first dense layer and also reducing the number of dropout layers from four to two. The second and last dropout layer were also set to a drop rate of 30% (Subsection 5.2.3). These small tweaks to the model's architecture resulted in our final model with an accuracy of 99,9% in testing (Subsection 6.1).

7.1.4 Develop functionality for the TurtleBot

Through this section, there will be given an insight into the performance of the current working solution for a prototype. There will be suggestions to how a better performing prototype could be developed based on the analysis of different frameworks and functionality, as well as discussing the flaws in the system that should be fixed.

Impression of ROS

We faced a lot of issues throughout the setup and development stage of the TurtleBot. In the setup stage, we got a lot of errors when connecting the TurtleBot to a remote computer. The error messages were difficult to understand and the documentation for ROS did not help much. Many of the issues we faced were common issues that developers had posted in ROS answers along with other forums, but the answers in these threads lacked content. After being stuck on a trivial issue, we approached one of our advisors, Kim. He helped us setting up the TurtleBot so that we could maneuver it with commands through the terminal.

After getting the TurtleBot correctly configured to the remote computer we started the development of a system to maneuver it. Initially, we where given instructions on how to control the TurtleBot with commands. Originally this was our backup solution since we wanted to use the rospy library to develop this functionality. To use rospy, we had to create a ROS package so that we could develop subscribers and publishers. Facing error messages the documentation could not answer became a growing issue. We quickly figured that we would rather create a working solution utilizing the command line for controlling the TurtleBot, rather than spending needless amount of hours trying to find help through the forum. This decision meant that we had to exclude the rospy package, which could have been helpful for the responsiveness in the system.

Examination of the movement logic

After having evaluated and tested the movement logic in Subsection 6.3.4, we have found several weaknesses in our system. Some of these weaknesses are easier to fix than others, the more complex ones will be discussed in further development (Section 7.2).

The first step to improve performance is implementing the process of movement directly into the prediction script (Appendix Q) or improve communication between scripts. The current solution (Subsection 5.3.2) is sending the predicted value from the prediction script to the API, then opening the movement script (Appendix R). There are a couple of issues with this approach. Firstly, going through a separate connector before passing the predicted value to the movement script will cause more delay than passing it directly to the movement functionality. Secondly, having to load every package each time a prediction is made is exhaustive in terms of time. This could be solved by integrating the movement functions directly into the prediction script and run the prediction script continuously. Or by running both the movement and prediction script continuously and use sockets for direct communication between the two scripts.

The process of executing a movement is by opening a subprocess that sends a rostopic through the command line (Subsection 6.3.4). Having this process timed by a static value for seconds is generally bad practice. The wait time is set to one second, and with this comes two scenarios that might occur during execution. One, the process of executing the movement script takes less time to execute than one second, this leads to unnecessary waiting time that slows down the process. Two, the process for the movement script takes longer than one second to execute. In this scenario, the execution of the movement would never happen. The last scenario is similar to timing out, which should raise an error. The subprocess is not able to respond with a confirmation or error message after trying to execute a movement, making it impossible for the system to know if there are issues with movement or not.

The actual execution of movements through the command line using rostopic is flawed. The reason for this is that we are opening a process and assumes the time it takes to execute by using `sleep()` before terminating the process. Since we were not able to configure the rospy package, we lost a lot of functions that would have been helpful when executing the movements using topics. This package would improve the execution of the movement by using built-in functions for sending messages, making the subprocess for executing commands obsolete.

The LiDar sensor could help improve the stopping mechanism for situations where the TurtleBot is to close to obstacles in the direction it is supposed to drive towards, potential development of this feature will be discussed in Section 7.2.

Examination of the capture logic

The capture script (Appendix P) was developed to fit the rest of the system's performance for predicting the gesture and executing a movement. In a real-world system for an application like this, the camera script we have developed would have been too slow and time-consuming. The logic of the capture script will be discussed in this section and potential solutions will be examined.

Considering that the capture script had to take the performance of the other components into account to function properly (Subsection 6.3.1), the capture script could be seen as viable for the current prototype. It does its executions rather efficiently and captures clear images that are occasionally predicted correctly. Preparing the images on the TurtleBot reduces the file size from approximately 0.67 MB to 5 KB. The resized image file would be transmitted efficiently even on low-speed Ethernet. The time it takes for the file to be sent from the TurtleBot to the remote computer is not affecting the responsiveness of the control system, thus, using an API for communication between these two devices is a viable solution for this prototype. There are some parts of the script that is either exhaustive or could have been implemented using different methods. Having to write two images to directories for storing them and later reuse them could have been improved, for saving both storage space and reduces the need for memory.

A relatively easy implementation that could make this script more efficient is to use live videostreams and picking out frames. This implementation could be solved by using the PiCamera function `capture_continuous`. Using this we would change the way we iterate to transmit images. To capture frames we would use a for-loop to iterate through every frame of the videostream. Executing operations when picking a frame would naturally slow down the process for picking the next frame, this is not necessarily bad. Sending too many frames per second could lead to exhausting the other components of the system without implementing proper handling. This functionality gives us access to an actual image that is stored in an in-memory stream when using the stream parameter, thus, reducing the number of images stored in a directory by one. The remaining lines would execute the same way as with the current solution. The resized image needs to be stored in a directory for it to be sent through the API since the since it can not handle pixel values.

7.1.5 Connecting the neural network with the TurtleBot

Connecting the neural network with the TurtleBot is a process that requires multiple programs and units to communicate together. This section will cover how these individual components connect to each other and the functionality it provides.

Model

When connecting the TurtleBot and the Lightning McQueen Phase Three model (Subsection 5.2.3), we faced some problems. In this case, the problem was at the remote machine that we were using for predictions with the model.

When we started developing the scripts for handling the predictions, we had a problem with setting up PlaidML (Subsection 2.3.3). The cause of this problem was the remote machine which was using Python 2.7 when it should have been using a Python3 release. We had to install the right version of Python on our remote machine so that we could set up PlaidML correctly.

API

As mentioned in evaluation (Subsection 6.3.2), there are improvements that should be implemented, mainly regarding security and logging. The API should only allow requests sent from the TurtleBot, this could be implemented such that each request made to the service is checked. In this check, it would look at the IP address and compare it to the IP address of the TurtleBot, if a non-match was found the request should be denied with a response code of 403, forbidden. Validation of the incoming file should be implemented as an extra layer of security. Whenever a file is sent to the service, both size and mime-type of the file must be checked to ensure the file is not too large to delay the processing as well as avoid invalid file types to be uploaded.

Having a system to log predictions and the associated image could be beneficial for further improvement of the model and finding mistakes. By saving the uploaded images, the model could gain more data to train on, as well as having a way to identify where the model is good or bad. Logging the number of requests processed, time, file size, and denied requests are also important to implement for a greater overview of how the entire system works.

Assuming the hardware of the unit running the system is capable of running the model in an effective way, the API's usage would be modified. Instead of being responsible for both image upload and calling the prediction script, it would serve as a logging tool to save the predicted image as mentioned above. This would reduce the complexity and bottlenecks we currently face when running the programs after each request is received. Insteadm, the unit would simply process the prediction and instantly move, then send the captured image and prediction to the API for further logging.

Prediction script

The model is loaded in the prediction script (Appendix Q) where it reads the uploaded image and returns a prediction index representing one of the six possible categories. The script then executes the movement script (Appendix R) together with the category index as an argument.

A big bottleneck in the system that was found once we started testing, is the process of loading the model, which is very memory consuming (Subsection 6.3.3). As the model has to be loaded every time a new request is sent to the API, it will slowly cause the system to slow down and use a longer time to predict the directions. A potential solution to this issue is to simply move the loading of the model out of the script, and run this once in a different program and keep an active connection to the model open. Optimally, the prediction script would run continuously and listen to the API for a signal that a new image is ready to be handled. This solution would lead to the model and packages being loaded just once.

The new architecture of the system would be much more efficient in terms of memory and the actual time it takes to run the entire process of capturing an image, run the prediction and cause the unit to move using the predicted gesture.

7.2 Further Development

This section will contain the work that has to be done to further develop a system for an autonomous vehicle that can be controlled by arm gestures. We will look into the features we initially wanted to implement that we did not have time to do, and features we have discussed as a group that we believe would be useful in a fully-fledged system.

7.2.1 Dataset and recognition

During the development of the model, we encountered some obstacles that led us to make the dataset simpler. Initially, we wanted to create a dataset containing images of the group facing the camera, as well as having our backs to it. In Subsection 5.1.2 we have explained the decision to not include gestures with our back to the camera.

For further development, it should be spent more time on creating higher quality datasets, with more diversity. The datasets that we made for this project were not optimal, and we would highly recommend putting a lot of time into creating good data.

7.2.2 Pose estimation

A tool that could be applied to a system as we have developed is TensorFlow lite model, PoseNet. It detects human figures in an image or video and determines which limbs and facial features can be seen in the frame[1]. The algorithm can estimate where the key body joints are in the image. This could be implemented with a system like ours to try and enhance the performance and accuracy of the system.

Another approach would be to make a system that resembles whats discussed in Subsection 3.5.1.

7.2.3 Increasing reliability

Even though our model performed well in testing there are measures that can be taken to make the system more reliable and safe.

Multiple models

Our prototype is currently only using Lightning McQueen Phase Three (Subsection 5.2.3) for predicting the gestures. When looking at what can be done for further development, we would like to do research into using multiple models to make a more precise prediction. By taking an average prediction from multiple models we can be more confident in the final prediction output.

[1] https://www.tensorflow.org/lite/models/pose_estimation/overview

Prediction averaging

A prediction averaging algorithm is used to avoid flickering predictions. Implementing this would solve any issues of the system jumping rapidly between two or more different categories. Lowering the chances of this happening would greatly improve the overall safety and reliability when using the system. This is done by maintaining a list containing a set amount of the last predictions, averaging each category and selecting the one with the highest probability[2].

7.2.4 Follow me feature

Implementing a feature that enables the vehicle to follow the operator would make the system more user friendly. This feature would let the operator signal a gesture, and the vehicle would follow every step the operator takes. During the initial planning stages of the dataset, the group discussed the possibility of implementing this feature and had a goal to do so if we had time at the end of the project period. This feature was also discussed with FFI and they found the idea interesting. The feature would make the vehicle autonomous when there is no access to GPS. The operator's responsibility would just be to map out the route that the vehicle is able to drive on, instead of having to manually steer the vehicle towards the correct objective with gestures. This feature would make it more efficient to control the vehicle over greater distances.

To implement this feature we would need to further develop the prototype. First, we would need to further develop the dataset with a new unique arm gesture. After gathering the data for the gesture, we would need to train the model with the new data. Lastly, we would need a functionality for the TurtleBot to mimic the exact movement of the operator.

7.2.5 Additional sensors

For a system like this to be reliable, there should be more than just one single camera for it to work. For simple maneuvering like we are doing in this project, one camera works just fine. Ultimately for a more advanced system, there would be radar, LiDAR, ultrasonic sensors, and multiple cameras giving the vehicle more information about its surroundings. These sensors should be considered for something like our systems in further development.

7.2.6 Safety measures

The safety measures the system needs is two-sided. We need to consider the safety for both the operator and the vehicle. Not recognizing a gesture is an output from the model that will be triggered when the operator is too close to the camera, this will stop the vehicle from moving any further. In a more developed system, there is a need for measures that do not rely entirely on not recognizing a gesture. There should be a minimum distance between the vehicle and the operator, there are multiple options to do this. There is already existing technology using radar sensors for recognizing distance to objects. These sensors could be used for extra safety for both vehicle and operator since these sensors would recognize that the operator is too close to the vehicle or objects that may harm the vehicle. There is also the possibility of using a LiDAR sensor to map out the vehicle's surroundings to detect any objects in its path.

[2] https://www.pyimagesearch.com/2019/07/15/video-classification-with-keras-and-deep-learning/

7.2.7 Logging

Logging is critical in order to achieve a good overview of how the system performs as well as to be aware of potential issues that may occur. The system needs to save any uploaded images with the respective predicted category, using storage with multiple directories, representing each of the potential categories the model can predict. Each image logged would be added to the correct directory, representing the predicted gesture with the timestamp of the prediction. By separating the different gestures, further analysis and statistics can be done on the data to improve the model and find potential flaws and errors. All requests made to the API that results in a denied response should be logged along with the timestamp of the request and the reason for the error. The average file size of the uploaded images should be stored and logged to determine if the system is able to process the file sizes in a reasonable time.

7.2.8 rospy

The decision to not use rospy came down to the fact that we had a limited amount of time to develop the required programs. Since we had already investigated and found a way to move the TurtleBot using rostopic directly through the command line, we decided it was better to build programs around this instead of learning a new tool that wraps around the rostopic command. While this is generally a less efficient solution and does not fully utilize the performance and speed rospy could provide, it was necessary to accomplish the development of a working prototype.

Implementing the rospy library in our system could enhance the performance of the prototype. The library is built for having an interface to the integrated features in ROS. To execute a movement using rospy, a topic containing a message to move the TurtleBot could be published. This would replace the current functionality in the movement script, where a command shell is opened as a process and executes the rostopic for publishing a geometry message of type Twist, with both linear and angular arrays containing values for directions. Using `publish()` and messages in the rospy library we can simply publish a message when a prediction is made.

7.2.9 Communication between components

The communication from the API to the movement script in the current solution is not viable in an operational system. After opening the subprocess in the movement script, there is no way of knowing if the movement was executed or not. To provide a better flow when sending commands and files between scripts, we could use sockets for instant messaging when something needs to be executed or when reporting back that something has been executed. Having sockets between the directory containing posted images and the prediction script would allow the prediction script to continuously execute and listen to changes in the image directory and fetch the new images. There would also be socket communication between the prediction script and the movement script, allowing instant messages between the scripts for confirmation that an action is executed. The communication between the capture script and prediction script through an API should not be used in a fully developed vehicle. A vehicle should have enough processing power to execute the predictions. This would create a product that could be executed without having Ethernet access and would run more smoothly because there could be communication between the capture script and prediction script using sockets to send images when a new frame is captured.

Chapter 8

Conclusion

The main purpose of this project was to develop a prototype that can serve as a proof of concept towards further development of a system that allows humans to maneuver autonomous vehicles using arm gestures. During research for similar solutions, we found little to no information on technologies that conforms to the requirements set in this project. Thus, we had to gain a deeper understanding of machine learning, image recognition, and robotics to be able to develop the entire prototype ourselves from the ground up.

High quality and varied data became of great importance during the development of the prototype. Since the gesture-data produced in the project period was lacking in a few aspects, which resulted in our model not performing optimally during live testing.

When developing the CNN model, there were several important discoveries made. By applying dropout and the right optimizer, we were able to improve the performance of the models, which lead to the Lighting McQueen Phase Three model. The most important discovery made was the significance of training data. If the training data is inadequate for real-world scenarios, the accuracy of a given prediction would decrease. Testing the model with data augmented images performs remarkably well with an accuracy of 99.9%, however, making predictions in real-world situations reduces the accuracy drastically.

Developing functionality for the TurtleBot3 Waffle Pi improved our comprehension of what is needed in a fully developed system. Potential improvements have been discussed and would be necessary for this system to be feasible in a realistic scenario, however, in a controlled environment the current solution will suffice. Features that are currently implemented would be obsolete in a better-equipped vehicle, modifying these features would be essential for further development.

To improve the performance of the model, more time must be invested in creating high quality and accurate training data, combined with hyperparameter tuning and a more advanced model architecture search. To enhance the general quality of the system, logging the output from a prediction alongside the captured image is a required functionality. This is a valuable source for gathering real-world data that can be re-categorized by manually scraping it. The re-categorized data can then, in turn, be used for retraining the model to improve the accuracy of the predictions. It is vital to always ensure safety for the operator and the vehicle while using the fully-fledged system. Greater safety can be assured by utilizing the additional sensors on the vehicle to map the surroundings.

Finally, we conclude that our project serves as a good building foundation for developing a fully operational system. Even though the prototype created for this project is suboptimal, it serves as a proof of concept that a system like this can be pursued and implemented.

Bibliography

[1] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer Science+Business Media, 2006.

[2] Yu-Hsiu; Kung Chia-Ching; Chung Ming-Han; Yen I.-Hsuan Chen, Yung-Yao; Lin. *Design and Implementation of Cloud Analytics-Assisted Smart Power Meters Considering Advanced Artificial Intelligence as Edge Analytics in Demand-Side Management for Smart Homes*. Sensors, 2019.

[3] Yarin Gal; Zoubin Ghahramani. A theoretically grounded application of dropout in recurrent neural networks. 2016.

[4] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press.

[5] Hannes Schulz Dirk Schulz Kishore Konda, Achim Königs. Real time interaction with mobile robots using hand gestures. 2012.

[6] Honda Motor Co Ltd. Posture recognition apparatus and autonomous robot, 2008.

[7] Chigozie Nwankpa; Winifred Ijomah; Anthony Gachagan; Stephen Marshall. Activation functions: Comparison of trends in practice and research for deep learning. 2018.

[8] Thi-Lan Le; Minh-Quoc Nguyen; Thi-Thanh-Mai Nguyen. Human posture recognition using human skeleton provided by kinect. 2013.

[9] Stuart J. Russell; Peter Norvig. *Artificial Intelligence: A Modern Approach, Third Edition*. Prentice Hall, 2009.

[10] Y. Bengio; A. Courville; P. Vincent. *Representation Learning: A Review and New Perspectives*. IEEE, 2013.

[11] Li Yin. Explain feedforward and backpropagation. 2018.

[12] Bin Ding; Huimin Qian; Jun Zhou. Activation functions and their characteristics in deep neural networks. 2018.

Glossary

ANN Artificial Neural Network, a computing system based on the neural networks in animal's.

architecture Defines the various layers and structure in a machine learning model.

backpropagation backpropagation, is a algorithm for computing the gradient of a loss function in respect to the weights of the neural network.

CNN Convolutional neural network, a class of deep neural network mostly used for image analysis.

CPU Central Processing Unit, electric circuitry that executes instructions such as basic arithmetic, input/output operations, logic and controlling.

CUDA Compute Unified Device Architecture, is a parallel computing platform and API created by NVIDIA.

dataset A collection of labeled data, in the case of this project a dataset is a collection of labeled images used for training a machine learning model.

downsampling downsampling, a way to randomly subsample negative samples to make the an unbalanced dataset balanced.

FFI Forsvarets forskningsinstitutt, the Norwegian defence sector's own research institution.

GPU Graphical Processing Unit, is a piece of hardware that is designed to do simple numerical computations.

Kinect Motion sensing input devices produced by Microsoft.

LiDAR LiDAR, is a tool to measure distance by using a laser.

mime-type mime-type, is a way of identifying files on the Internet according to their nature and format.

ML Machine learning, focuses on providing computers with data so it can learn by itself.

ndarray N-dimensional array, multidimensional and homogeneous array of fixed sized items.

NN Neural network, algorithms loosely modeled after the human brain.

pickle pickle files, is a module in Python that implements a binary protocols for serializing and de-serializing Python object structures.

pixel values pixel values, in the case of gray scale images a pixel value is a single number representing the brightness of the pixel.

REST REST, is a software architectural style that defines a set of constraints to be used for creating Web services.

ROS Robot operating system, operating system used to control the TurtleBot3 Waffle Pi.

SDRAM Synchronous dynamic random-access memory, enables better performance by finishing operations while at the same time receiving new commands.

SLAM Simultaneous localization and mapping, is a computation for updating or creating a map of an unknown environment while keeping track of the vehicle inside it.

SSH SSH, a secure way to access a computer over an unsecured network.

testset A collection of labeled data, in the case of this project a dataset is a collection of labeled images used for testing a machine learning model.

TurtleBot TurtleBot3 Waffle Pi, throughout this thesis these two names are used interchangeably.

VRPN VRPN, or Virtual-Reality Peripheral Network that allows for controlling of remote units.

Code

Appendix A

First time setup of Lightning McQueen

A.1 Intro

Lighting McQueen is our phases of models created for arm gesture recognition. Bellow you will find the setup process for being able to run the code used for machine learning part of this project.

The Lightning_McQueen project contains:

- data_augmentation.py
- Label.py
- dataset.py
- create_mcqueen_fully_connected.py
- create_mcqueen_phase_one.py
- create_mcqueen_phase_two.py
- create_mcqueen_phase_three.py
- plot_training.py
- model.py
- test_mcqueen.py
- ConfusionMatrix.py

A.2 Requirements

- Python 3.6 or 3.7
- We recommend using PyCharm IDE (https://www.jetbrains.com/pycharm/)

A.3 Configure and activate virtual environment.

A.3.1 PyCharm

If a virtual environment. was not created when setting up a new project in PyCharm follow JetBrains guide on setting up a virtual environment (https://www.jetbrains.com/help/pycharm/creating-virtual-environment.html)

A.3.2 Terminal

Check to see if your install of Python has pip

```
pip -h
```

Install virtualenv package

```
pip install virtualenv
```

Create virtual environment Make sure to be in the project directory.

```
virtualenv venv
```

Activate virtual environment Mac OS and Linux

```
source venv/bin/activate
```

Windows

```
venv\Scripts\activate
```

If you want to deactivate the virtual environment simply run `deactivate` in the terminal window.

A.4 Installation of needed packages

Clone the GitHub repository into your project from: `https://github.com/BO20-G38/Lightning_McQueen`

Use the package manager pip for installing needed packages.

```
pip install plaidml-keras plaidbench
pip install -U matplotlib
pip install opencv-python
pip install tqdm
pip install termcolor
pip install sty
pip install pickle
```

A.5 Setup PlaidML

If you have a dedicated GPU its recommended to select it as your accelerator.

Choose which accelerator you'd like to use (many computers, especially laptops, have multiple) In the terminal of your python project (venv) write:

```
plaidml-setup
```

- Enable experimental mode
- Select your accelerator

Now try benchmarking MobileNet:

```
plaidbench keras mobilenet
```

- You are now good to go!

Appendix B

First time setup of API Move service

B.0.1 Intro

The API service is responsible for handling upload of images used to run predictions on. The setup process, packages and requirements for running the service can be found below.

Project files

- index.js
- upload.js
- router.js
- pySpawn.js
- package.json
- .babelrc

B.0.2 Requirements

The following must be installed on the system in order to run the service. Package manager NPM can optionally be used instead of Yarn.

- Node version 10 or higher
- Yarn

B.0.3 Installation of required packages

Dependencies are listed in package.json, installed using yarn.

```
yarn install
```

B.0.4 Running the service

The service will run on localhost port 3000 and handle POST requests to base route.

```
yarn start
```

Appendix C

Running the prototype

Below is the procedure for running the prototype, separated into individual sections that must be configured.

C.1 Setup TurtleBot3

The following steps must be configured on the TurtleBot to allow communication with the remote computer.

Open `~/.bashrc` in any editor and add two lines to the end `ROS_MASTER_URI` and `ROS_HOSTNAME` then edit so it looks as the following:

- ROS_MASTER_URI = http://LAPTOP_IP:11311
- ROS_HOSTNAME = TURTLEBOT_IP

Substitute *LAPTOP_IP* with the ip address of your laptop and *TURTLEBOT_IP* with the ip address of the TurtleBot. Remember to save and then `source ~/.bashrc`.

C.2 Setup Remote Computer

The following steps must be configured on the remote computer to allow communication with TurtleBot.

EDIT ROS_MASTER and ROS_HOSTNAME

Similar to the turtlebot, `ROS_MASTER_URI` and `ROS_HOSTNAME` need to be defined on the remote computer. Open `~/.bashrc` and add two lines containing the following data:

- ROS_MASTER_URI = http://LAPTOP_IP:11311
- ROS_HOSTNAME = LAPTOP_IP

Substitute *LAPTOP_IP* with the ip address of your laptop. Remember to save and then `source ~/.bashrc` as was done on the turtlebot.

C.2.1 Install TurtleBot3 ros packages

The following TurtleBot3 packages are required to interface the TurtleBot:

sudo apt-get install ros-kinect-joy ros-kinect-teleop-twist-joy ros-kinect-teleop-twist-keyboard ros-kinect-laser-proc ros-kinect-rgbd-launch ros-kinect-depthimage-to-laserscan ros-kinect-rosserial-arduino ros-kinect-rosserial-python ros-kinect-rosserial-server ros-kinect-rosserial-client ros-kinect-rosserial-msgs ros-kinect-amcl ros-kinect-map-server ros-kinect-move-base ros-kinect-urdf ros-kinect-xacro ros-kinect-compressed-image-transport ros-kinect-rqt-image-view ros-kinect-gmapping ros-kinect-navigation ros-kinect-interactive-markers

cd ~/catkin_ws/src git clone https://github.com/ROBOTIS-GIT/turtlebot3_msgs.git git clone https://github.com/ROBOTIS-GIT/turtlebot3.git cd ~/catkin_ws catkin_make

C.2.2 Install vrpn_client_ros

The vprn_client_ros is the ROS node responsible for interfacing the vprn protocol and allow data to be transfered between the turtlebot and remote computer.

sudo apt-get install ros-kinetic-vrpn cd ~/catkin_ws/src

 # The kinetic-devel branch works for both ROS kinetic and melodic
git clone --branch kinetic-devel https://github.com/ros-drivers/vrpn_client_ros.git cd ~/catkin_ws catkin_make

C.3 Connect TurtleBot and remote computer

Below is the required steps needed to allow data transfer between the remote computer and the TurtleBot. All steps assumed the current directory is `catkin_ws` for both remote computer and TurtleBot.

1. roscore on the remote:

   ```
   roscore
   ```

2. Start the vrpn_client_ros, replace `LAPTOP_IP` with the ip of your laptop

   ```
   roslaunch vrpn_client_ros sample.launch server:=LAPTOP_IP
   ```

3. Either SSH into the TurtleBot or manually start up TurtleBot ros node, replace `TURTLEBOT_IP` with ip of the TurtleBot if using SSH.

   ```
   ssh pi@TURTLEBOT_IP
   roslaunch turtlebot3_bringup turtlebot3_robot.launch
   ```

4. Assuming instructions of API setup is followed from Appendix B, Start up the Direction API service to allow captured images to be sent from TurtleBot to remote computer.

```
cd direction_api
yarn start
```

5. Access TurtleBot as done in step 3 and start the capture script.

```
python camera_capture.py
```

Appendix D

video_to_img.py

Code D.1: video_to_img.py, Python script for converting videos to images and labeling it as a dataset.

```
#
#  Author: Sander Helles  and William Svea-Lochert
#  Run script:  python capture.py <datapath> <outputpath> <imgsize>
#
# This script is for creating a dataset. You feed it folder containing
# videos, and it will return as many  * pictures for every sec as the
# FPS of the video file has, for every video found in the passed in folder.
#
#

# Importing all necessary libraries
import cv2
import os
import sys
import numpy as np
import random
import pickle
from datetime import datetime

videosSourcePath = sys.argv[1]
directoryName = sys.argv[2]
imgSize = int(sys.argv[3])

CLASS_CATEGORIES = ["forward", "right", "left", "backward", "stop"]
trainingData = []


# recurse over dirs, and extract frames from every video found
def read_dir_and_start_frame_extraction(path):
    directory = os.fsencode(path)

    print("Extracting frames from files found under: " + path)
    for file in os.listdir(directory):
        filename = os.fsdecode(file)
        fullFilePath = create_path(path, filename)

        if filename.endswith(".h264") or filename.endswith(".mp4"):
            extract_frames_from_video(fullFilePath)
        elif os.path.isdir(fullFilePath):
            read_dir_and_start_frame_extraction(fullFilePath)
```

```
def extract_frames_from_video(videoSource):
    # Read the video from specified source
    cam = cv2.VideoCapture(videoSource)

    category = videoSource.split('-')[1]
    classNum = CLASS_CATEGORIES.index(category)
    imageOutputPath = build_output_image_path(category)
    currentframe = 0

    while True:

        # reading from frame
        ret, frame = cam.read()

        if ret:
            # if video is still left continue creating images
            videoFileName = clean_filename(os.path.basename(videoSource))
            outputImageName = videoFileName + '-frame-' +
                              str(currentframe) + '-bW.jpg'
            outputDestination = create_path(imageOutputPath, outputImageName)

            # writing the extracted images
            cv2.imwrite(outputDestination, frame)

            # manipulate extracted image
            img_ = cv2.imread(outputDestination, cv2.IMREAD_ANYCOLOR)
            gray = cv2.cvtColor(img_, cv2.COLOR_BGR2GRAY)
            img_ = cv2.resize(gray, (imgSize, imgSize))

            # overwrite extracted image with updated properties
            cv2.imwrite(outputDestination, img=img_)

            # create image array from image and add to training data
            add_img_array_to_training_data(outputDestination, classNum)

            # increasing counter so that it will
            # correctly name the file with current frame counter
            currentframe += 1
        else:
            break

    # Release all space and windows once done
    cam.release()
    cv2.destroyAllWindows()


# helper to clean the filename from its extension
def clean_filename(fileName):
    strList = fileName.split('.')
    strList.pop()

    return "".join(strList)


# helper to create joined path with OS specific delimiters
def create_path(root, path):
```

```
    return os.path.join(root, path)


# combine passed in directory name and source category
def build_output_image_path(category):
    imageOutputPath = create_path(directoryName, category)

    try:
        # creating a folder named data
        if not os.path.exists(imageOutputPath):
            os.makedirs(imageOutputPath)

        # if not created, then raise error
    except OSError:
        print('Error: Creating directory: ' + imageOutputPath)

    return imageOutputPath


def add_img_array_to_training_data(imgPath, classNum):
    imgArray = cv2.imread(imgPath, cv2.IMREAD_GRAYSCALE)
    trainingData.append([imgArray, classNum])


def create_training_data():
    random.shuffle(trainingData)

    # features
    X_PICKLE_OUT_PATH = create_path(directoryName, "X.pickle")
    X = []

    # labels
    Y_PICKLE_OUT_PATH = create_path(directoryName, "y.pickle")
    y = []

    for features, label in trainingData:
        X.append(features)
        y.append(label)

    X = np.array(X).reshape(-1, imgSize, imgSize, 1)

    create_pickle(X, X_PICKLE_OUT_PATH)
    create_pickle(y, Y_PICKLE_OUT_PATH)


def create_pickle(pickleData, path):
    pickleOut = open(path, "wb")
    pickle.dump(pickleData, pickleOut)
    pickleOut.close()


def start():
    startTime = datetime.now()

    read_dir_and_start_frame_extraction(videosSourcePath)
    create_training_data()

    timeToRun = datetime.now() - startTime
```

```
    print("Done in " + str(timeToRun.seconds) + " seconds.")


start()
```

Appendix E

data_augmentation.py

Code E.1: data_augmentation.py, Python script for changing background on greenscreen images.

```
# ---------------------------------------------------------- #
# Author William Svea-Lochert
# Date written: 21.03.2020
# Script for changing a greenscreen to a new background
# ---------------------------------------------------------- #

import cv2
import numpy as np
import random
import os
from tqdm import tqdm


CLASS_CATEGORIES = ["forward", "right", "left", "backward", "stop", "still"]
training_data = []
dir_data = 'OutputDir'


def show_image(image_file):
    cv2.imshow('asd', image_file)
    cv2.waitKey(0)
    cv2.destroyAllWindows()


def clean_filename(filename):
    strList = filename.split('\'')
    strList.pop()
    strList.pop(0)
    return "".join(strList)


def create_path(root, path):
    return root + '/' + path


def read_dir(path, img_count, output):
    directory = os.fsencode(path)
    convert = True

    for file in os.listdir(directory):
```

```
        filename = os.fsencode(file)
        full_file_name = create_path(path, clean_filename(str(filename)))
        if convert:
            change_bg(full_file_name, dir_data+'/' +
                      output + '/', clean_filename(str(filename)) +
                      '-', img_count)


def change_bg(input_path, output_path, name, image_amount):
    # Original Image
    image = cv2.imread(input_path)
    positive_roll = 220
    negative_roll = -220

    # Roll the image between -251 and 251
    if output_path == dir_data+'/still' or output_path == dir_data+'/stop':
        positive_roll = 200
        negative_roll = -200

    for i in tqdm(range(image_amount)):
        image_copy = np.copy(image)
        image_copy = cv2.resize(image_copy, (1000, 1000))
        image_copy = np.roll(image_copy, 3 * random.randint(negative_roll,
                                                            positive_roll))

        lower_blue = np.array([0, 50, 0])  # [R value, G value, B value]
        upper_blue = np.array([55, 255, 55])

        mask = cv2.inRange(image_copy, lower_blue, upper_blue)
        masked_image = np.copy(image_copy)
        masked_image[mask != 0] = [0, 0, 0]

        # loading new background image
        background_image = cv2.imread('background.jpg')

        # Depending on the width of the background image change
        # randint parameters
        background_image = np.roll(background_image, 3 *
                                   random.randint(-1800, 1800))

        # If the background image need to be resized uncomment this line.
        # background_image = cv2.resize(background_image, (1344, 1014))

        crop_background = background_image[0:1000, 0:1000]

        crop_background[mask == 0] = [0, 0, 0]

        final_image = crop_background + masked_image
        final_image = cv2.resize(final_image, (100, 100))


        #            FOLDER/        name-number.jpg
        cv2.imwrite(output_path + name + str(i) + '.jpg', final_image)
```

Appendix F

label.py

Code F.1: label.py, Python script for labeling a image dataset.

```
import numpy as np
import os
import cv2
from tqdm import tqdm
import random
import pickle


CATEGORIES = ["forward", "right", "left", "backward", "stop", "still"]

training_data = []
IMG_SIZE = 100


def create_training_data(DATADIR):
    for category in CATEGORIES:

        path = os.path.join(DATADIR, category)
        class_num = CATEGORIES.index(category)

        for img in tqdm(os.listdir(path)):
            try:
                img_array = cv2.imread(os.path.join(path, img))
                training_data.append([img_array, class_num])
            except Exception as e:
                pass

    # Shuffle dataset
    random.shuffle(training_data)

    # Features
    X = []

    # Labels
    y = []

    for features, label in training_data:
        X.append(features)
        y.append(label)
```

```
X = np.array(X).reshape(-1, IMG_SIZE, IMG_SIZE, 3)

pickle_out = open(DATADIR + "/X.pickle", "wb")
pickle.dump(X, pickle_out)
pickle_out.close()

pickle_out = open(DATADIR + "/y.pickle", "wb")
pickle.dump(y, pickle_out)
pickle_out.close()
```

Appendix G

dataset.py

Code G.1: dataset.py, Python script for loading and normalizing image datasets.

```
# ------------------------------------------------------------ #
# Author: William Svea-Lochert
# Date written: 16.02.2020
# loading dataset into project and normalizing the values
# between 0 and 1. All function calls for all datasets
# is not provided in this file.
# ------------------------------------------------------------ #
import pickle
import numpy as np


def load_x_1():
    pickle_in = open("E:/pro_datasets/dataset_1_room/X.pickle", "rb")
    x = pickle.load(pickle_in)
    x = x/255
    return x


def load_y_1():
    pickle_in = open("E:/pro_datasets/dataset_1_room/y.pickle", "rb")
    y = pickle.load(pickle_in)
    y = np.asarray(y)
    return y
```

Appendix H

create_mcqueen_fully_connected.py

Code H.1: create_mcqueen_fully_connected.py, Python script for creating McQueen fully connected model.

```
# ---------------------------------------------------------- #
# Author: William Svea-Lochert
# Date written: 20.01.2020
# Initial creation of the McQueen fully connected model
# for recognising arm gestures.
# ---------------------------------------------------------- #
import os
from cars.load.dataset import load_y_1, load_x_1
from cars.plotting.plot_training import plot_model

os.environ["KERAS_BACKEND"] = "plaidml.keras.backend"

from keras.models import Sequential
from keras.layers import Dense, Flatten
from keras.utils import normalize
from keras.callbacks import EarlyStopping

NAME = 'McQueen_fully_connected'  # Model name

# Loading dataset
X = load_x_1()
y = load_y_1()

# Model definition
model = Sequential()
model.add(Flatten())
model.add(Dense(512, activation='relu', input_shape=X.shape[1:]))
model.add(Dense(1024, activation='relu'))
model.add(Dense(2048, activation='relu'))
model.add(Dense(5, activation='softmax'))

model.compile(optimizer='adam', loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Stops the training early of the val_loss has stopped improving
early_stop = EarlyStopping(monitor="val_loss", min_delta=0, patience=4,
                           verbose=0, mode="auto", baseline=None,
                           restore_best_weights=False)

# Trains the model and saves the history of the training
```

```
history = model.fit(X, y, batch_size=150, epochs=100, validation_split=0.2,
                    callbacks=[early_stop])

model.save(NAME + '.model')

# Plotting the accuracy & loss of the training and validation
plot_model(history, 'acc', 'McQueen_Fully_connected', 'acc.png')
plot_model(history, 'loss', 'McQueen_Fully_connected', 'loss.png')
```

Appendix I

create_mcqueen_phase_one.py

Code I.1: create_mcqueen_phase_one.py, Python script for creating McQueen phase one model.

```
# ------------------------------------------------------------ #
# Author: William Svea-Lochert
# Date written: 16.02.2020
# Initial creation of the McQueen phase one model
# for recognising arm gestures.
# ------------------------------------------------------------ #
import os
from cars.plotting.plot_training import plot_model
from cars.load.dataset import load_x_1, load_y_1


os.environ["KERAS_BACKEND"] = "plaidml.keras.backend"

from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D
from keras.callbacks import EarlyStopping
from keras.optimizers import Adam


NAME = 'McQueen_phase_one'  # Model name

# Loading dataset
X = load_x_1()
y = load_y_1()

# building the mode
model = Sequential()
model.add(Conv2D(32, kernel_size=(5, 5), strides=(1, 1), activation='relu',
                 input_shape=X.shape[1:]))

model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(64, (5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(128, (5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(6, activation='softmax'))

# Stops the training early of the val_loss has stopped improving
```

```
early_stop = EarlyStopping(monitor="val_loss", min_delta=0, patience=3,
                           verbose=0, mode="auto", baseline=None,
                           restore_best_weights=False)

model.compile(loss='sparse_categorical_crossentropy', optimizer='adam',
              metrics=['accuracy'])

history = model.fit(X, y, batch_size=150, epochs=8, verbose=1,
                    validation_split=0.3, callbacks=[early_stop])


# Plot training & validation accuracy & loss values
plot_model(history, 'acc', 'McQueen', NAME + '_acc.png')
plot_model(history, 'loss', 'McQueen', NAME + '_loss.png')

model.save(NAME + '.model')
```

Appendix J

create_mcqueen_phase_two.py

Code J.1: create_mcqueen_phase_two.py, Python script for creating McQueen phase two model.

```
# ------------------------------------------------------------ #
# Author: William Svea-Lochert
# Date written: 27.02.2020
# Initial creation of the McQueen phase two model
# for recognising arm gestures.
# ------------------------------------------------------------ #
import os

os.environ["KERAS_BACKEND"] = "plaidml.keras.backend"

from cars.load.dataset import load_y_1, load_x_1
from cars.plotting.plot_training import plot_model

from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, Dropout
from keras.callbacks import EarlyStopping
from keras.optimizers import SGD


NAME = 'McQueen_phase_two'  # Model name

# Loading dataset
X = load_x_1()
y = load_y_1()

# Model definition
model = Sequential()
model.add(Conv2D(20, kernel_size=(5, 5), strides=(1, 1), activation='relu',
          input_shape=X.shape[1:]))

model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Dropout(0.70))
model.add(Conv2D(32, kernel_size=(5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.50))
model.add(Conv2D(40, kernel_size=(5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.25))
model.add(Flatten())
model.add(Dense(900, activation='relu'))
```

```
model.add(Dropout(0.5))
model.add(Dense(350, activation='relu'))
model.add(Dense(6, activation='softmax'))

model.compile(optimizer=SGD(lr=0.001), loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])

# Stops the training early of the val_loss has stopped improving
early_stop = EarlyStopping(monitor="val_loss", min_delta=0, patience=3,
                           verbose=0, mode="auto", baseline=None,
                           restore_best_weights=False)

# Trains the model and saves the history of the training
history = model.fit(X, y, batch_size=320, epochs=15,
                    validation_split=0.2, callbacks=[early_stop])

model.summary()
# Plotting the acc & loss of the training and validation
plot_model(history, 'acc', 'McQueen_Phase_Two_CNN', 'acc.png')
plot_model(history, 'loss', 'McQueen_Phase_Two_CNN', 'loss.png')

model.save(NAME + '.model')  # Saving the model
```

Appendix K

create_mcqueen_phase_three.py

Code K.1: create_mcqueen_phase_three.py, Python script for creating McQueen phase two model.

```
# ------------------------------------------------------------ #
# Author: William Svea-Lochert
# Date written: 15.05.2020
# Initial creation of the McQueen phase three model
# for recognising arm gestures.
# ------------------------------------------------------------ #
import os

os.environ["KERAS_BACKEND"] = "plaidml.keras.backend"

from cars.load.dataset import load_y_2, load_x_2
from cars.plotting.plot_training import plot_model

from keras.models import Sequential
from keras.layers import Dense, Flatten, Conv2D, MaxPooling2D, Dropout
from keras.optimizers import SGD
from keras.callbacks import ModelCheckpoint, EarlyStopping

FOLDER = "model_dir/"
NAME = FOLDER + 'McQueen_p3'  # Model name

# Loading dataset
X = load_x_2()
y = load_y_2()

# Model definition
model = Sequential()
model.add(Conv2D(20, kernel_size=(5, 5), strides=(1, 1), activation='relu',
          input_shape=X.shape[1:]))

model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(32, kernel_size=(5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(40, kernel_size=(5, 5), activation='relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.7))
model.add(Flatten())
model.add(Dense(700, activation='relu'))
model.add(Dropout(0.3))
model.add(Dense(350, activation='relu'))
```

```
model.add(Dense(6, activation='softmax'))

early_stop = EarlyStopping(monitor="val_loss",
                           min_delta=0,
                           patience=2,
                           verbose=0,
                           mode="auto",
                           baseline=None,
                           restore_best_weights=True)

model.compile(optimizer=SGD(lr=0.001, momentum=0.8, decay=1e-6),
loss='sparse_categorical_crossentropy', metrics=['accuracy'])

# Trains the model and saves the history of the training
history = model.fit(X, y, batch_size=200, epochs=10,
                    validation_split=0.3, callbacks=[early_stop])

model.summary()
# Plotting the acc & loss of the training and validation
plot_model(history, 'acc', 'McQueen_p3_t1', FOLDER+'acc.png')
plot_model(history, 'loss', 'McQueen_P3_t1', FOLDER+'loss.png')

model.save(NAME + '.model')  # Saving the model
```

Appendix L

model.py

Code L.1: model.py, Python Script for training retraining models

```
# ------------------------------------------------------------ #
# Author: William Svea-Lochert
# Date written: 16.02.2020
# Proceeds then to load a given model, specify early stop
# callback for escaping overfittment. Then proceeds to train
# the given model for the specified amount of epochs, give
# a summary of the model then save the model and return the
# history object created by the fit() function call.
# ------------------------------------------------------------ #
import os

os.environ["KERAS_BACKEND"] = "plaidml.keras.backend"

from keras.models import load_model
from keras.callbacks import EarlyStopping


# Train the given model
def train_model(X, y, epochs, batch_size, val_split, patience,
                                        model, save_location):
    # Load the model from the given location
    model = load_model(model)  # Load model

    # Stops the training early of the val_loss has stopped improving
    early_stop = EarlyStopping(monitor="val_loss", min_delta=0,
                               patience=patience, verbose=1,
                               mode="auto", baseline=None,
                               restore_best_weights=True)

    # Train model with early stop callback
    history = model.fit(X, y, epochs=epochs,
                        batch_size=batch_size,
                        validation_split=val_split,
                        callbacks=[early_stop])  # train the model

    # Saves the newly trained model to the specified location.
    model.save(save_location)

    # Return history object
    return history
```

Appendix M

test_mcqueen.py

Code M.1: test_mcQueen.py, Python script for testing a model, and create a confusion matrix.

```
# ------------------------------------------------------------ #
# Author: William Svea-Lochert
# Date written: 01.03.2020
# test with testset and create a confusion matrix
# ------------------------------------------------------------ #
from keras.models import load_model
from cars.load.dataset import load_x_testset_m_2, load_y_testset_m_2
from cars.matrix.ConfusionMatrix import ConfusionMatrix


def test_model(model_name):
    categories = ['forward', 'right', 'left', 'backward', 'stop', 'still']

    x_test = load_x_testset_m_2()  # load dataset
    y_test = load_y_testset_m_2()  # load dataset

    print('Loading model: ' + model_name)
    model = load_model(model_name)  # load the model
    print('Testing model!')
    y_new = model.predict_classes(x_test)  # Prediction

    matrix = ConfusionMatrix(categories)

    for i in range(len(x_test)):  # check the predictions
        matrix.add_pair(predicted_category=categories[y_new[i]],
                        actual_category=categories[y_test[i]])

    print(matrix)

    # Prints loss function first and then acc on the testset
    score = model.evaluate(x_test, y_test, verbose=0)
    print(score)
```

Appendix N

ConfusionMatrix.py

Code N.1: ConfusionMatrix.py, Python script for testing a model, and create a confusion matrix.

```
# Code provided by Allan Lochert

import numpy as np
from sty import fg
from typing import List


class ConfusionMatrix:

    def __init__(self, categories: List = []):
        self.categories = categories
        self.matrix = np.zeros((len(categories), len(categories)))

    def add(self, item):
        self.matrix[self.categories.index(item.category),
                    self.categories.index(item.predicted_category)] += 1

    def add_pair(self, predicted_category, actual_category):
        self.matrix[self.categories.index(actual_category),
                    self.categories.index(predicted_category)] += 1

    def __str__(self):
        result = f'{fg.li_blue}'
        for i in range(0, len(self.categories)):
            result += f'{self.categories[i]:>8}'
        result += f'accuracy{fg.rs}\n'
        all_total = 0
        for i in range(0, len(self.categories)):
            line = f'{fg.li_blue}{self.categories[i]:<10}{fg.rs}'
            total = 0
            for j in range(0, len(self.categories)):
                total += self.matrix[i,j]
                if i == j:
                    clr = fg(255)
                else:
                    clr = fg(248)
                line += f'{clr}{self.matrix[i,j]:8.0f}{fg.rs}'

            line += f'{fg.yellow}{self.matrix[i,i]/total*100:9.1f}%{fg.rs}'
```

```
            result += f'{line}\n'
            all_total += total

        total_correct = 0
        line = f'{fg.li_blue}Precision{fg.yellow}'
        for i in range(0, len(self.categories)):
            total = 0
            for j in range(0, len(self.categories)):
                total += self.matrix[j, i]
            total_correct += self.matrix[i, i]
            line += f'{self.matrix[i,i]/total*100:7.1f}%'
        result += f'{line}'

        result += f'{fg(255)}{total_correct/all_total*100:9.1f}%{fg.rs}'

        return result
```

Appendix O

plot_training.py

Code O.1: plot_training.py, Python script for plotting the training history.

```
# ---------------------------------------------------------- #
# Author: William Svea-Lochert
# Date written: 16.02.2020
# Plot training & validation accuracy & loss values
# ---------------------------------------------------------- #
import matplotlib.pyplot as plt


# Plot the given model via history object from training
def plot_model(history, metric, name, save_location):
    plt.plot(history.history[metric])
    plt.plot(history.history['val_' + metric])
    plt.title(name + ' ' + metric)
    plt.ylabel(metric)
    plt.xlabel('Epoch')
    plt.legend(['Train', 'val'], loc='upper left')
    plt.savefig(save_location)
    plt.show()
```

Appendix P

camera_capture.py

Code P.1: camera_capture.py, script that executes on Raspberry Pi and transmits images to api

```
# ------------------------------------------------------------------------ #
# Author: Sverre Stafsengen Broen
# Run script: python camera_capture.py
#
# This script is for capturing images on the Raspberry Pi
# It captures an image, prepares it by gray scaling and resizing
# Then it sends the image through post request to API
# It catches the response and prints it
# Error count is to stop the script if too many errors has occurred
#
# Removing path before importing cv2 had to be done to bypass import error
# ------------------------------------------------------------------------ #

# Importing necessary packages
from time import sleep
from picamera import PiCamera
import requests
import sys
import cv2

def prep_img(filepath):
    IMG_SIZE = 100  # 50 in txt-based

    # read in the image, convert to grayscale
    img_grayscale = cv2.imread(filepath, cv2.IMREAD_GRAYSCALE)

    # resize image to match model's expected sizing
    img_resized = cv2.resize(img_grayscale, (IMG_SIZE, IMG_SIZE))
    return img_resized


# Preparing paths
IMG_PATH = '/home/pi/catkin_ws/capture_script/img_lib/imgs/'
RESIZE_PATH = '/home/pi/catkin_ws/capture_script/img_lib/imgs_resized/'
URL = 'http://192.168.1.146:3000'

# Initiating variables
error_count = 0
image_count = 0
```

```
# Preparing camera module
camera = PiCamera()
camera.resolution = (1000, 1000)

camera.start_preview()
while error_count < 5:
    sleep(2)

    # Preparing name of captured image
    image_count += 1
    capture_name = 'image' + image_count + '.jpg'

    # Capturing image, prepares it, and stores it in resize path
    camera.capture(IMG_PATH + capture_name)
    image = prep_img(IMG_PATH + capture_name)
    isWritten = cv2.imwrite(RESIZE_PATH + capture_name, image)
    if isWritten:  # If image was written successful to resize path
        # Fetches resized image from path
        resized_image = open(RESIZE_PATH + capture_name, 'rb')

        # Does request, gets status code and closes connection
        response = requests.post(URL, files=dict(file=resized_image))
        status_code = response.status_code
        response.close()

        print(status_code)

        # Increase error if status code isn't 200
        if str(status_code) != str(200):
            error_count += 1
    else:
        error_count += 1


camera.stop_preview()
sys.exit()
```

Appendix Q

predict.py

Code Q.1: predict.py, script for making a prediction on an image sent from the API

```
import os
import time
import sys
import subprocess as sp
import cv2
import warnings
warnings.filterwarnings('ignore', category=FutureWarning)

os.environ["KERAS_BACKEND"] = "plaidml.keras.backend"

from keras.models import load_model

CATEGORIES = ["forward", "right", "left", "backward", "stop", "still"]


def prep_img(filepath):
    IMG_SIZE = 100
    img_array = cv2.imread(filepath, cv2.IMREAD_GRAYSCALE)
    img = img_array.reshape(-1, IMG_SIZE, IMG_SIZE, 1)
    return img / 255


def pred(model_path, img):
    model = load_model(model_path)
    predict = model.predict_classes(prep_img(img))
    return predict[0]


workingDir = os.path.dirname(os.path.realpath(sys.argv[0]))
prediction = pred(workingDir + '/t11/McQueen_p3.model',
                  workingDir + '/direction_api/directions/direction.jpg')

sys.stdout.write(str(prediction))
```

Appendix R

init.py

Code R.1: init.py, script for moving the TurtleBot based on the prediction from predict.py

```
import subprocess
import time
import sys
import os

initial_dir = sys.argv[1]

BASE_CMD = "rostopic pub /cmd_vel geometry_msgs/Twist -- "

# Functions for movement stop, left, right, forward, backward
def stop():
    linear = [0, 0, 0]
    angular = [0, 0, 0]
    return stringify(linear) + " " + stringify(angular)


def left():
    linear = [1, 0, 0]
    angular = [0, 0, -2]
    return stringify(linear) + " " + stringify(angular)


def right():
    linear = [1, 0, 0]
    angular = [0, 0, 2]
    return stringify(linear) + " " + stringify(angular)


def forward():
    linear = [1, 0, 0]
    angular = [0, 0, 0]
    return stringify(linear) + " " + stringify(angular)


def backward():
    linear = [-1, 0, 0]
    angular = [0, 0, 0]
    return stringify(linear) + " " + stringify(angular)
```

```
# Turns arrays in to string type and wraps it in quotations
def stringify(arr):
    return "'" + str(arr) + "'"


# Hub for finding correct movement and execute command
def move(movement_type):
    switch = {
        "0": forward(),
        "1": right(),
        "2": left(),
        "3": backward(),
        "4": stop(),
        "5": stop()
    }

    coordinate = switch.get(movement_type)
    command = BASE_CMD + coordinate
    print(command)
    process = subprocess.Popen(command, shell=True)
    time.sleep(1)
    process.terminate()


# Start process of move
def start():
    move(initial_dir)


start()
sys.stdout.write("1")
```

Appendix S

PySpawn.js

Code S.1: PySpawn.js, Script for executing predict and init.py

```
import path from "path";
import shell from "shelljs";

const scriptDir = path.join(__dirname, "../..");

const options = {
  async: true,
  silent: true
};

const commands = {
  predict: `python ${scriptDir}/predict.py`,
  init: `python ${scriptDir}/init.py`
};

// Recieves predicted direction from predict.py and run movement script
const predictPy = res => {
  const child = exec(commands.predict);
  child.stdout.on("data", direction => movePy(direction, res));
};

// Excecute turtlebot movement then resolves HTTP request
const movePy = (direction, res) => {
  const child = exec(`${commands.init} ${direction}`);
  child.stdout.on("data", _ => res.json({ direction }));
};

const exec = cmd => shell.exec(cmd, options);

export default predictPy;
```

Appendix T

index.js

Code T.1: index.js, Script for starting HTTP server on remote computer

```
import express from "express";
import router from "./app/router";

const app = express();
app.use(router);

app.listen(3000, () => console.log("API running..."));
```

Appendix U

upload.js

Code U.1: upload.js, Script for handling upload of images to remote computer

```
import multer from "multer";
import path from "path";

const destDir = "directions";
const fileName = "direction.jpg";
const dest = path.join(__dirname, `../${destDir}/`);

const storage = multer.diskStorage({
  destination: (req, file, cb) => cb(null, dest),
  filename: (req, file, cb) => cb(null, fileName)
});

const upload = multer({ storage });

export default upload;
```

Appendix V

router.js

Code V.1: router.js, Script for handling incoming HTTP requests to API

```
import express from "express";
import upload from "./upload";
import pySpawn from "./pySpawn";

const router = express.Router();

router.post("/", upload.single("file"), (req, res) => {
  if (!req.file) {
    return res.status(400);
  }

  // run movement processes once image is uploaded
  pySpawn(res);
});

export default router;
```