University of Applied Sciences and Arts
Northwestern Switzerland

Machine Learning Based Autonomous Robot with NVIDIA Jetson Nano

A Thesis Submitted in Partial Fulfilment of the Requirements for the Degree of

Bachelor of Science in Computer Science

Submitted by

Muralt, Dimitri

7th semester, iCompetence

Schlossstrasse 143, 3008 Bern

dimitri.muralt@students.fhnw.ch

Wenk, Christoph

7th semester, iCompetence

Lerchenweg 1, 8600 Dübendorf

christoph.wenk@students.fhnw.ch

Presented to

University of Applied Sciences and Arts Northwestern Switzerland

School of Engineering

Supervised by

Graber, Michael, Department of Data Science

Co-supervised by

Märki, Fabian, Department of Data Science

Windisch, 17. March 2020

Summary

In the thesis at hand a small autonomous robot called JetBot is being built and programmed. It is based on the Jetson Nano Developer Kit by NVIDIA. The goal of this project is to evaluate the capabilities of the JetBot by assembling the hardware and implementing an example for the programmatic control. The JetBot should be able to solve a simple task by leveraging its machine learning abilities including object recognition.

In this thesis a representative task is being selected that showcases the capabilities of the JetBot. Gesture control has been selected as a suitable task to solve. It describes the selection, implementation and evaluation of a gesture recognition pipeline. Gesture control has current relevance for mobile and embedded devices since it is being used in a wide range of applications such as smart TVs and virtual reality.

The hardware for the JetBot has been selected, ordered and assembled including 3D printing and soldering the electrical parts. The JetBot is able to stream video with a mounted camera and is able to move around with its motors while being powered by a single battery.

From the machine learning perspective, the JetBot is capable to detect multiple hands and classify those into gestures. The JetBot executes an action depending on the classified gesture like following a hand. For this purpose, three neural network models run on the JetBot. One model supports the robot with collision avoidance. It helps to detect potentially dangerous situations like an edge of a table. A second model detects hands and a third model classifies the detected hands into known gestures. For this thesis how-to guides have been written that document all the relevant steps including the inference, the data collection and training of the selected models. This should allow an easy retraining and adjustment of the proposed system.

Table of contents

1 Introduction

Mobile, autonomous robots enable several interesting use cases and are under heavy development. Examples include robots developed by Swiss Post, that are delivering parcels to our doorstep, and vacuum cleaner or lawn mower robots, that already have found their way into our everyday lives. NVIDIA has recently published instructions how to build a small, autonomous vehicle leveraging their Jetson Nano Developer Kit (Welsh & Yato, 2019a) called JetBot (Figure 1). The Jetson Nano platform and JetBot seem to be an adequate way to explore and develop mobile, machine learning based applications in educational environments.

The goal of this project is to evaluate the platform to showcase the capabilities of the JetBot. For this purpose, the JetBot has to be built first according to the documentation that is provided by NVIDIA on GitHub and programmed afterwards. The JetBot should be able to solve a task, which contains an object-recognition-step by leveraging its machine learning capabilities.

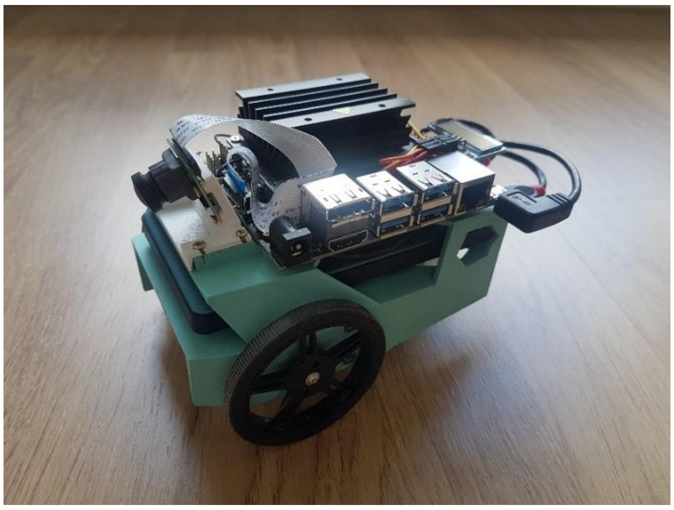


Figure 1: One of the two JetBots built for this thesis

Further definition of the task to solve for the JetBot has been declared as part of the challenge of this project. A representative task must be defined for the robot taking into account its limited hardware capabilities. The task should also serve as a demonstration case which can be used to showcase embedded machine learning to visitors of the University of Applied Sciences and Arts Northwestern Switzerland FHNW.

There are several difficulties that need to be tackled. The explorative nature of this project offers plenty of freedom to select an interesting problem to solve. However, it is also a challenge to pick a task that will meet the hardware and software requirements of the JetBot. It should also be a problem that finds the right balance of complexity, that is, challenging but also manageable, so it may be solved within the time constraints of this thesis. Fundamental knowledge of machine learning is needed to be able to choose a suitable task. As there is little previous machine learning knowledge available within the project team this knowledge needs to be first built up. Then the current state of research needs to be explored. With this knowledge it will then be possible to define a task that fulfils the requirements. Additionally, as the JetBot platform is very new only few projects from other developers exist. It is to be expected that technical problems will occur for which yet no solutions exist. These problems will need to be filtered, that is, determine which problems may be solved and which fall out of the scope of this project due to complexity as advanced knowledge in machine learning and embedded systems may be needed. This means the project will be challenging but is also a good opportunity to learn as much as possible in the area of machine learning.

The JetBot needs to fulfill a number of requirements as defined in the project agreement. First, the parts for the JetBot need to be ordered. Afterwards the robot needs to be built. Then the JetBot needs to be programmed by leveraging its hardware. The robot should be able to move around autonomously and have a self-driving strategy that is suitable for the task at hand. The user should see a live video stream from the onboard camera on his screen. The machine learning aspect of this task should be to include an object detection task that allows the robot to detect multiple objects at once. Furthermore, the JetBot system should possess the ability to use an object classification model that allows it to recognize objects. These two skills should then be combined into a system to solve a simple real-world task. Additionally, how-to guides or explanations should be included that allow readers to reproduce this project and that further explain potential pitfalls of the official guides. The system should be programmed in Python as this offers high compatibility to other machine learning frameworks.

This new solution does not have to be based on completely new model architectures and could also include already existing models. Existing models can be used as a baseline and can be retrained to recognize new objects. The hardware does not have to be modified and can be used as is. However, the hardware pieces need to be selected, ordered and assembled.

After analyzing various other projects for the Jetson Nano and the JetBot the use case could be narrowed down to hand gesture recognition. This is an interesting use case as it allows users to control the JetBot with gestures. Compared to other objects that could be used for interaction, like a cup, hands and therefore gestures are always available independently from time and location. However, the researched sources indicate that hand detection is a difficult topic because of the many variations of a hand in terms of view and form (Mohammed, et al., 2019, p. 1). Nevertheless, gesture recognition was selected as demonstration case as it promised to be an interesting and useful way to interact with the JetBot.

The demonstration case implemented in this thesis runs three neural networks on the JetBot. One model supports the robot with collision avoidance. It helps to detect potentially dangerous situations like an edge of a table. A second model detects hands and a third model classifies gestures from the detected hands. The goal is to read the current camera frame, detect any hands if available, classify any visible hands as known gestures and then let the robot react depending on whichever gesture it has detected. A simplified version of this pipeline can be found in Figure 2. More details are explained in chapter 3.

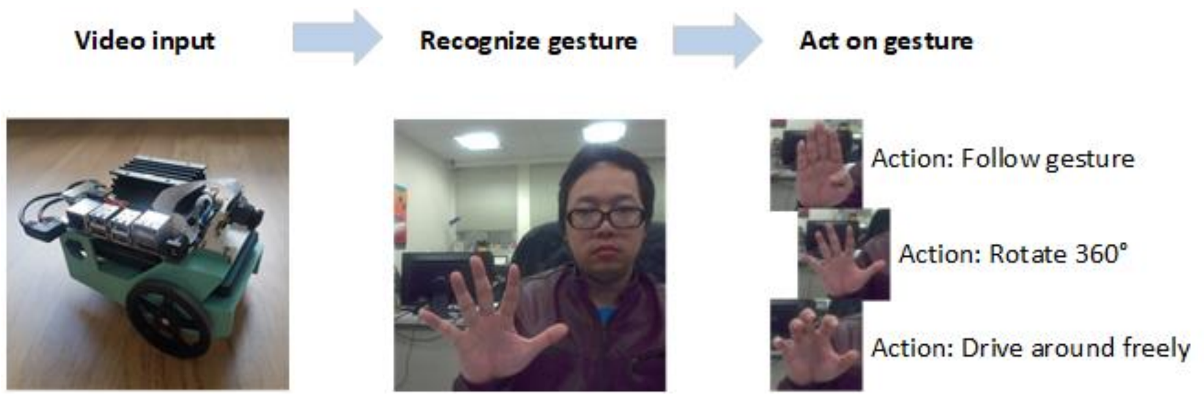


Figure 2: Gesture recognition demonstration case (Sanchez-Riera, et al., 2015)

A number of tasks were completed in order to create the proposed system. First, the JetBot parts were selected and ordered. Then, the parts were assembled. While waiting for the parts to arrive from international shipping the machine learning knowledge had to be built up and the current state of research was investigated. Once the JetBot was assembled, familiarity with its capabilities and features needed to be built up by following the example guides. To achieve the goal of controlling the JetBot with gestures several system architectures were evaluated. A hand detection model and hand classification model were trained. Three input datasets for these models were prepared and partially manually annotated.

The following chapters are structured as follows. Chapter 2 introduces the Jetson Nano Developer Kit and its main features covering the assembly and setup of a JetBot from hardware to software level. Chapter 3 explains how the demonstration case was selected and what its general idea is. Chapter 4 covers the machine learning part of this thesis. In it the selection of the system architecture is explained along with its implementation and evaluation. Chapter 5 summarizes and reflects the achieved results.

2 The Jetson Nano Developer Kit

This chapter introduces the Jetson Nano Developer Kit and its main features. The chapter covers the assembly and setup of a JetBot from hardware to software level as well as the issues that surfaced during the process. It also explains how the parts for the robot have been acquired.

2.1 Technical specifications

This section provides an overview of the technical specifications and features of the Jetson Nano Developer Kit.

NVIDIA promotes the Jetson Nano as an easy to learn entry-level platform for developers interested in machine learning on their website.

> “NVIDIA® Jetson Nano™ Developer Kit is a small, powerful computer that lets you run multiple neural networks in parallel for applications like image classification, object detection, segmentation, and speech processing. All in an easy-to-use platform that runs in as little as 5 watts.” (NVIDIA, 2020e)

It is part of the NVIDIA’s Jetson product family for embedded artificial intelligence.

The Jetson Nano Developer Kit is based on a modified Tegra X1 system on a chip (SoC) developed by NVIDIA (Tegra X1, 2020). The X1 integrates an ARM-architecture based central processing unit (CPU), a graphical processing unit (GPU) as well as other chips and controllers. Combined with an operating system, this provides a platform that can be used as a mini computer.

The Nano features a quad-core ARM Cortex-A57 64-bit CPU that runs at 1.43 GHz. The X1 SoC specifies a NVIDIA Maxwell-architecture-based GPU with 256 cores. With 128 cores the Nano offers half of that. The Jetson Nano comes with 4 Gigabytes of memory that is shared between the GPU and the CPU (Tegra X1, 2020).

Comparable devices with the X1 chipset are the NVIDIA Shield or the Nintendo Switch. However, these are based on another model of the Tegra X1 that offers more GPU and CPU cores as well as higher frequencies and more memory (Tegra X1, 2020).

The Jetson Nano Developer Kit offers plenty of interfaces, which are explained in detail in the user guide (NVIDIA, 2019j, pp. 5-7). This paragraph gives a short overview of the ones used for the JetBot. The video output can be delivered over DisplayPort or a HDMI outlet. Network access is possible with either a wireless adapter, that can be inserted into the M.2 slot or over a Gigabit Ethernet port. External devices like a keyboard or a mouse can be connected over one of the four USB 3.0 ports. Additional controllers and devices can be connected to the 40-pin expansion header. This interface has been used for connecting the OLED-Display as well as the motor controller. The camera can be connected to a specialized camera connector on the board.

The Jetson Nano has two power modes at 10W and 5W (NVIDIA, 2019j, pp. 7-8). The 10W mode utilizes all four cores and runs the CPU and GPU on a higher frequency while the 5W mode disables two of its cores and lowers the clock rate significantly as seen in Table 1.

Power Mode	**10W**	**5W**
Mode ID	0	1
Online CPU cores	4	2

CPU Max Frequency (MHz)	1428	918
GPU Max Frequency (MHz)	921	640
Memory Max Frequency (MHz)	1600	1600

Table 1: CPU and GPU usage at different power modes (Franklin, 2019)

These modes can also be toggled as described in section 2.4 and subsection 2.5.3.

2.2 Procurement of parts

This section describes how the various parts for the JetBot have been collected and the challenges that had to be faced. It begins with the order process and proceeds then with the printing of the 3D parts. At last, possible alternatives will be discussed.

2.2.1 Order process

NVIDIA provides a bill of materials that are needed to build a JetBot on GitHub (Welsh & Yato, 2019c). This list is mainly targeted at people living in the United States and therefore needs to be adapted for the Swiss market.

This creates several issues that need to be tackled. First and foremost, some vendors mentioned in the original bill do not deliver to Switzerland or do this with considerably higher shipping cost and additional customs charges due to international shipping. The solution to this is to select Swiss vendors and order the parts from the domestic market if possible. The idea is to keep the number of vendors small in order to save shipping costs and have a better overview of the orders.

Another advantage of ordering from local vendors is the shorter shipping time. However, some parts are not available from local vendors and need to be purchased from foreign businesses or switched for similar parts. A part where this particularly becomes a problem is the suggested power bank, which was neither available in Switzerland nor on Amazon. Other developers have faced this issue as well and needed to replace it with other ones that have similar specifications (JetBot Issue 16, 2019). This is problematic as switching a part might lead to incompatibilities and requires a bit of luck to find a piece that works well together with the other components. The first power bank ordered for this project had to be returned as well as it was not able to output the required amount of power on both outlets.

As many of the required parts are produced by the same international vendor it was decided to follow a hybrid approach where whenever possible the parts would be ordered from this producer and the rest would be ordered from domestic vendors as can be seen in the customized bill of materials in appendix 0. This allowed to reduce the overall number of vendors as there was no single vendor in Switzerland that had all of this producer's products.

2.2.2 3D printing

The JetBot Chassis where the Jetson Nano will be mounted on needs to be 3D printed (Welsh & Yato, 2019b). For his NVIDIA provides 3D models as STL files and a 3D printing guide on the JetBot Wiki (Welsh & Yato, 2019c). NVIDIA recommends either the Monoprice Voxel or the Ultimaker 2+ (Welsh & Yato, 2019b). For this project the Ultimaker 2 and 3 have been chosen. The 3D laboratory of the University of Applied Sciences and Arts Northwestern Switzerland provides free access to those printers provided that you have attended a workshop on how to use them.

NVIDIA provides general setup instructions for the software Cura, which is used to configure the 3D printer (Ultimaker B.V., 2019). This proved to be helpful as a general orientation as neither member of the project team had any previous experience with 3D printing. However, as each printer behaves a little bit differently the experience and help of the laboratory instructors proved to be helpful to find the right settings.

The JetBot uses four 3D printed parts. A chassis, where all the other parts are being mounted on, a camera mount, as well as a caster base and a caster shroud, which are needed to hold a caster ball that acts as a third wheel (Welsh & Yato, 2019c).

2.2.2.1 Prototype 1

The goal of the first prototype was to test if the 3D models provided by Welsh and Yato can be printed. Then in a second iteration the printout should be improved based on the findings of the first prototype.

Only the chassis should be printed for this first prototype as it is the part with the highest volume and the most complex structure. It is to assume that if this part can be printed the other parts should not cause any issues as well.

It took approximately 14 hours to print the first prototype of the chassis. After the removal of the support structures the prototype proved to be very sturdy in general.

However, the whole front part had not been printed at all as depicted in Figure 3. According to the laboratory instructor this happened probably because of a malfunction of the printer itself and not because of wrong printing instructions. For the second iteration another printer should be tried.

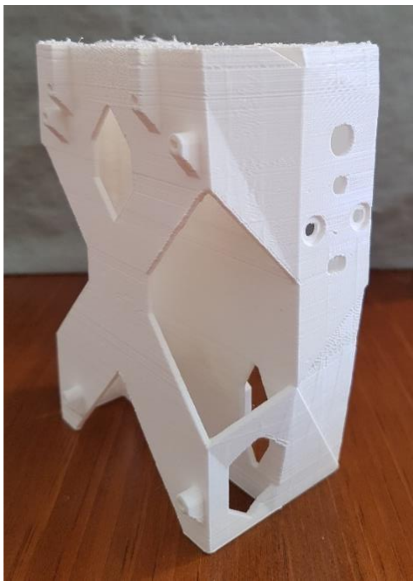


Figure 3: Prototype 1 with missing front

2.2.2.2 Prototype 2

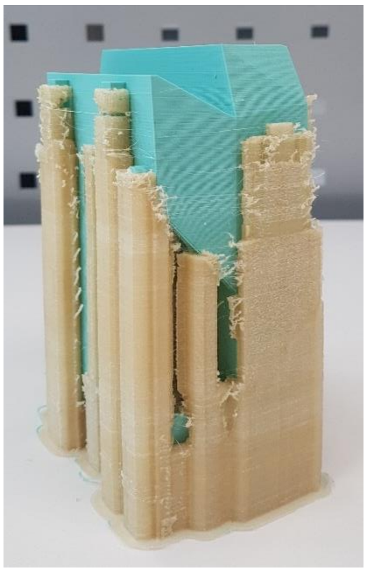


Figure 4: Prototype 2 with beige colored support structure

An Ultimaker 3 was used to print the second prototype. This printer allows the usage of a second water-soluble filament. This second filament can be used as a carrier structure as seen in Figure 4.

The models had to be printed in two batches as the printout takes a considerable amount of time and the laboratory was booked for other workshops. Thus, the chassis and the smaller parts have a different color.

It was easier to remove the support structure than it was with the first prototype as it could be simply dipped into water to dissolve the filament.

The chassis of the second prototype proved to be stable and was printed completely and correctly this time. The smaller models had been printed to an acceptable degree as well.

The parts proved to be sturdy enough to be used for the first JetBot.

2.2.2.3 Prototype 3

After the successful printing of the second prototype an Ultimaker 3 has been used for the third prototype as well. As with the second prototype the Ultimaker 3 completed the print without any apparent issues.

The parts were printed in a quality suitable to be used for the second JetBot.

2.2.3 Alternative procurement method

Planning and making everything yourself like performed and described in this thesis is a very rewarding way to assemble a JetBot. It allows one to gain insights how to search for fitting parts and what needs to be considered in regards to compatibility and feasibility. It also offers a chance to gather first experience with 3D printing.

However, it is also time consuming and one needs to be careful that the choice of parts fits well together. Ordering from different shops needs some planning ahead and attention needs to be paid in order to keep an overview of the different order states. One missing part could delay the project for weeks or even months. If on a tight schedule, this might jeopardize the whole project.

To solve this issue NVIDIA promotes JetBot kits that allow the assembly of a JetBot by oneself while also offering the advantage of a complete kit that is being sold by a single vendor (NVIDIA, 2019c). Some of these kits also remove the necessity of soldering or exclude the 3D printed parts if one is interested in doing that manually. This method can also decrease the overall costs as only a single shipping fee needs to be payed and kits can be sold in bulk which potentially lowers the cost compared to each part being sold separately.

2.3 Hardware setup

The JetBot Wiki provides a step-by-step description on how to assemble the JetBot out of the previously mentioned bill of materials (Welsh & Yato, 2019d). This proved to be a reliable source as the instructions are clear and for each step there are multiple photographs that illustrate the construction process.

The construction process consists of 15 distinguishable steps as seen in Table 2. This work does not cover all of them as they are already explained in detail on the JetBot Wiki page (Welsh & Yato, 2019d).

Step #	Description
1	Clean 3D printed parts
2	Mount motors
3	Solder motor driver
4	Strip motor driver power cable
5	Mount motor driver
6	Mount WiFi antennas
7	Remove Jetson Nano module from developer kit
8	Attach WiFi module to developer kit
9	Mount ball caster
10	Solder header onto PiOLED display

11	Wire motor driver to PiOLED display
12	Mount camera
13	Attach wide angle sensor
14	Mount battery
15	Organize wires

Table 2: Hardware assembly steps (Welsh & Yato, 2019d)

A few steps need soldering. The guide does not provide any instructions on how this needs to be done and what should be considered. It is therefore recommended to have previous experience with soldering to achieve a satisfying result. A soldering iron for electronics with a slender soldering tip should be available as some soldering joints are hard to get to. This also proved to be a problem particularly during the construction of the second JetBot. The Raspberry Pi OLED (PiOLED) display had unclean soldering joints (step 10) which caused the display to malfunction. These had to be removed and soldered again. The same issue caused a malfunction of the motor controller that is being prepared in step 3. The 2-pin and 3-pin male headers as well as the 2-pin and 4-pin screw terminals must be soldered clean and carefully for the controller to work properly.

An issue that later arose during the walkthrough of the examples was that the left and right motors were spinning backwards. As the software remained unchanged it became clear that the source of the issue lies in the hardware itself. The JetBot hardware setup guide step 5 (Welsh & Yato, 2019d) describes the order of the cables from the motors to the motor controller as red1, black1, red2, black2. This had to be rewired to black1, red1, black2, red2 in order for the bot to take directions correctly. In the meantime, the Wiki page has been edited accordingly.

The 160° camera objective not only covers a wider horizontal but also a wider vertical field of view compared to the default camera. After replacing the standard camera with the new objective portions of the battery pack were visible in the video output. This could be resolved by relocating the power bank a few millimeters back. Thinner adhesive pads for the camera mount helped to lift the camera angle a few degrees upwards.

2.4 Software setup

In similar fashion to their hardware guide NVIDIA provides a software setup guide (Welsh & Yato, 2019e) with step-by-step descriptions for setting up a JetBot. It includes the installation of the operating system, its configuration as well as the setup of the required software to control the JetBot. This work does not explain all the steps as seen in Table 3 in detail as this information is already covered by the official Wiki (Welsh & Yato, 2019e).

Step #	Description
1	Flash JetBot image onto SD card
2	Boot Jetson Nano
3	Connect JetBot to WiFi
4	Connect to JetBot from web browser
5	Install latest software (optional)
6	Configure power mode

Table 3: Software setup steps (Welsh & Yato, 2019e)

The user has to flash the image onto a MicroSD card, which then can be inserted into the Nano. After this step the JetBot can be powered on and booted.

The guide assumes that the user is familiar with the Ubuntu operating system. This proved to be very accessible due to many similarities with Windows and MacOS. The Ubuntu graphical user interface (GUI) is only needed for connecting the Nano to a wireless access point. Once a connection has been established the Nano can also be accessed by opening a secure shell (SSH) connection. The Ubuntu GUI is not needed anymore after this step.

The example guides from NVIDIA focus on working with Jupyter Notebooks. The Jupyter server running on the JetBot can be accessed from a web browser of another computer within the same network. For this the developer has to enter the JetBots IP address and the port the Jupyter server is running on. The IP address can be read from the PiOLED display on the rear of the JetBot. The default port is 8888 (Welsh & Yato, 2019e). E.g.

```
http://<jetbot_ip_address>:8888
```

The password is `jetbot` by default.

The Jupyter server will boot up automatically when the JetBot is powered on.

Optionally the developer can update the provided examples with code directly from the JetBot GitHub repository (Welsh & Yato, 2019e). This might be useful as the code in the repository is likely more recent than the example code provided with the JetBot image.

At last, the guide proposes to limit the power consumption to 5W (Welsh & Yato, 2019e). This should mitigate the JetBot from drawing more power than the power bank can supply.

2.5 Software package

This section focuses on the software that comes preinstalled on the JetBot image as well as its setup. It gives an overview of the main software packages used for machine learning and their purpose.

2.5.1 Operating system and utilities

The JetBot-image provided by NVIDIA uses Ubuntu version 18.04 LTS as an operating system. Ubuntu comes preinstalled with a graphical user interface based on the GNOME desktop environment. This allows the user to interact with the system additionally with a mouse rather than just with a keyboard.

The Nano can also be used as an alternative to a laptop or a desktop computer. It offers Libre Office for document processing as well as Chromium for browsing the internet (Ubuntu, 2019). With Mozilla Thunderbird there is an e-mail client included as well.

2.5.2 NVIDIA JetPack SDK

JetPack is a software development kit (SDK) that is being provided by NVIDIA. According to the product website (NVIDIA, 2019a) "NVIDIA JetPack SDK is the most comprehensive solution for building AI applications". It serves as an all in one solution and removes the need to install the products individually. This also adds the benefit of matching versions which otherwise could be a source of errors. At the time of writing, the version provided by the JetBot image was 4.3.

Jetpack includes NVIDIA L4T (Linux for Tegra). L4T provides the bootloader, Linux kernel, firmware and NVIDIA drivers for the Jetson product line (NVIDIA, 2019h). NVIDIA also provides a custom implementation of OpenCV that is optimized for the Jetson platform (NVIDIA, 2019i). The OpenCV version included in JetPack 4.3 is 4.1.1 (NVIDIA, 2019b). JetPack includes the NVIDIA Nsight toolchain. These tools help developers to analyze, debug, optimize and cross-compile their applications (NVIDIA, 2019a).

Other notable NVIDIA products with their respective versions included in the development kit are listed below.

- TensorRT 6.0.1
- cuDNN 7.6.3
- CUDA 10.0.326

The following subsections explain these three software packages briefly. The focus is to give an overview rather than an in-depth look as the details can be looked up in the software documentation of the respective products.

2.5.2.1 TensorRT

Among other software the SDK includes TensorRT. NVIDIA describes it as "[…] an SDK for high-performance deep learning inference. It includes a deep learning inference optimizer and runtime that delivers low latency and high-throughput for deep learning inference applications." (NVIDIA, 2019d). The purpose of TensorRT is to increase the performance of neural network models: "After applying optimizations, TensorRT selects platform specific kernels to maximize performance on Tesla GPUs (Graphics Processing Units) in the data center, Jetson embedded platforms, and NVIDIA DRIVE autonomous driving platforms". The model is being optimized in terms of latency, throughput, power efficiency, and memory consumption. It can further improve the performance by running the model in lower precision (NVIDIA, 2020b). TensorFlow has TensorRT already integrated and can use it to run optimized models directly (NVIDIA, 2020a).

2.5.2.2 cuDNN

"NVIDIA cuDNN is a GPU-accelerated library of primitives for deep neural networks. It provides highly tuned implementations of routines arising frequently in DNN applications" (NVIDIA, 2019e). Machine learning frameworks like TensorFlow need the cuDNN libraries to be installed on the system in order to run training and inference on the GPU (TensorFlow, 2020).

2.5.2.3 CUDA

The CUDA (Compute Unified Device Architecture) toolkit aims to provide a development environment for creating GPU-accelerated applications (NVIDIA, 2019f). It can be used to distribute computations across single or multi-GPU configurations.

It allows moving computations to the GPU instead of the CPU. This enables access to the highly parallel computation capabilities of the GPU (NVIDIA, 2019g), which is useful to process large data sets simultaneously like it is needed for machine learning. It results in a much higher throughput than what would be possible with a CPU (NVIDIA, 2019g). NVIDIA explains this with the higher number of transistors that are used for data processing: "The reason behind the discrepancy in floating-point capability between the CPU and the GPU is that the GPU is specialized for highly parallel computation […] and therefore designed such that more transistors are devoted to data processing rather than data caching and flow control" (NVIDIA, 2019g). CPUs need cache and flow control in order to minimize the latency from accessing the memory. GPUs can leverage their parallel computation capabilities to compensate for the memory access latency: "This conceptually works for highly parallel computations because the GPU can hide memory access latencies with computation instead of avoiding memory access latencies through large data caches and flow control." (NVIDIA, 2019g). CUDA aims to provide the tools and an interface to implement GPU-based computations easier.

Machine Learning Frameworks like TensorFlow need CUDA to gain access to the parallel processing power of the GPU (TensorFlow, 2020). This allows the developers to access the GPU with a high-level API so that they can concentrate on the task rather than the implementation (NVIDIA, 2019f).

2.5.3 jtop

JetPack 4.3 includes the jtop utility 1.7.8, which had to be installed manually with JetPack 4.2, developed by Raffaello Bonghi. Bonghi describes the tool as "[…] system monitoring utility that runs on the terminal and see and control real time the status of your NVIDIA Jetson. CPU, RAM, GPU status and frequency and other... [sic]." (Bonghi, 2020). Furthermore, jtop allows you to monitor temperatures and power usage. It can also be used to call the `jetson_clocks` script that activates all four cores of the Jetson Nano as only two are enabled by default. Additionally, Bonghi included a setting that allows to start `jetson_clocks` automatically on system boot. It can also be used to switch from 5W power mode to 10W mode. However, NVIDIA does not recommend this for stability reasons (Welsh & Yato, 2019e). If a fan has been installed the fan speed can be controlled from jtop as well. jtop can be run with the `jtop` command from the terminal. Bonghi suggests to also start it with `sudo` (Bonghi, 2020).

2.5.4 JupyterLab

JupyterLab, which is needed to run Jupyter Notebooks, is available as well and will be started on boot up. The NVIDIA JetBot team exclusively uses Jupyter Notebooks for its examples. As such it is the main development environment a developer needs in order to get to know how the JetBot works (Welsh & Yato, 2019f).

It allows to execute Python scripts step by step. These steps are partially repeatable which enables a developer to make corrections to the code without re-executing time consuming scripts like the import of a model. Documentation and commentaries can be entered directly between the code cells and can be formatted with Markdown (Project Jupyter, 2018).

System terminals can be opened directly from JupyterLab and allow the interaction with the host system.

Each Jupyter Notebook has its own Python kernel. This is needed to have a clean separation between the different execution environments. It also allows to close the notebook and open it later to check on the status of a long running action (Project Jupyter, 2018). However, it also adds a resource overhead which can be difficult to deal with considering the JetBot has only a limited amount of RAM. For this project only one notebook was open at the time to ensure all system resources are available to that runtime.

2.5.5 Other software

The JetBot image comes with preinstalled Python 3 as well as Python 2 as well as pip and pip3. A Conda environment for virtual Python environments is not available and needs to be installed manually if needed.

A NVIDIA customized TensorFlow-GPU 2 for their Jetson platform is preinstalled. NVIDIA also provides Jetson-optimized TensorFlow versions and installation instructions for TensorFlow 1.15 (NVIDIA, 2020c).

In order to run the JetBot examples provided by NVIDIA, PyTorch 1.3.0 must be used.

2.6 Encountered issues

This section lists the software and hardware issues that were encountered during this work and how they can be resolved. It focuses on issues that stand in relation to the development environment rather than issues related to machine learning.

2.6.1 Widgets not being displayed correctly

John Welsh (JetBot Issue 151, 2019) mentions that there might be problems with `ipywidgets` using certain browsers. He advises to use Google Chrome to ensure the best compatibility. In the beginning of this work Firefox was used to connect to the Jupyter server running on the JetBot. Sometimes this caused problems displaying the camera image and the JetBot controls. Switching to Chrome proved to be more reliable and stable.

2.6.2 Sideways drifting

Both JetBots built for this project tend to drift slightly to the right out of the box. Welsh explains that this is unavoidable due to the open-loop control scheme the JetBot uses (JetBot Issue 68, 2019). The JetBot has no sensors besides its camera. This would be required to feed the input of these sensors into the control loop in order to correct a deviation (DiStefano, et al., 1990, p. 3).

Welsh also mentions that the motors can be calibrated with software if needed (JetBot Issue 68, 2019). The motor implementation provides an `alpha` and a `beta` value, which can be used to calculate the turn speed `y` based on an input `x` for each motor.

```
y = alpha * x + beta
```

The alpha value can be directly manipulated for both motors.

```
robot.left_motor.alpha = 0.3

robot.right_motor.alpha = 0.7
```

For example, one of the JetBots could be calibrated by setting the alpha value of the right motor slightly above the alpha value of the left motor:

```
robot.right_motor.alpha = robot.left_motor.alpha + 0.02
```

2.6.3 Red tint on camera feed

The first tests with the camera showed a red tint on the camera feed as can be seen in Figure 5. This existed with both the standard Raspberry Pi Camera V2 as well as the 160° camera objective.

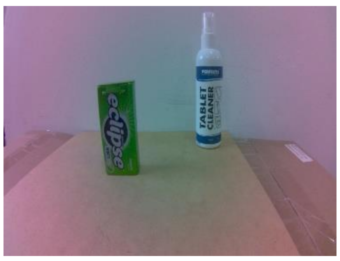


Figure 5: Red tint on camera feed (Tse, 2019)

The fact that this issue existed with both cameras excluded a hardware issue as the root cause. Research showed that the camera needs a new tuning profile, which was not mentioned in the setup guide. Jonathan Tse wrote a tuning article on how to fix this issue with a new camera profile (Tse, 2019).

For this the new profile needs to be downloaded and unpacked first.

```
wget https://www.waveshare.com/w/upload/e/eb/Camera_overrides.tar.gz
tar zxvf Camera_overrides.tar.gz
```

After this, the profile must be copied to the target directory with the correct owner specifications.

```
sudo cp camera_overrides.isp /var/nvidia/nvcam/settings/
sudo chmod 664 /var/nvidia/nvcam/settings/camera_overrides.isp
sudo chown root:root /var/nvidia/nvcam/settings/camera_overrides.isp
```

Afterwards the Nano needs to be rebooted in order to apply the new settings.

2.6.4 Frequent camera crashes

At the beginning of this work the JetBot showed very unstable behavior when executing the Jupyter Notebooks a second time. Often, the camera could not be instantiated when a notebook had been executed before. This problem even persisted after restarting the Python kernel. Only a complete shutdown and reboot of the system helped to be able to run another notebook.

While doing research about the problem it became evident that the issue is caused by the camera daemon itself and not the Python API (JetBot Issue 35, 2019). John Welsh proposes to restart the camera daemon before calling another script.

```
sudo systemctl restart nvargus-daemon
```

This helped indeed to solve the issue. To prevent further problems the daemon can be automatically restarted by executing the restart command at the beginning of all the Jupyter Notebooks that make use of the camera.

```
!echo jetbot | sudo -S systemctl restart nvargus-daemon
```

2.6.5 Not enough RAM

During the testing of the Jupyter Notebook examples it became clear that one of the impediments of the JetBot is its limitation of 4 Gigabytes of RAM. Often the RAM limit had been reached quickly which resulted in the JetBot behaving slow and uncontrollable.

To mitigate this the graphical user interface can be disabled as described in an issue on *Ask Ubuntu* (How to disable GUI on boot in 18.04 (Bionic Beaver)?, 2018).

The GUI can be disabled by running the following command on the command line.

```
sudo systemctl set-default multi-user.target
```

After issuing the command the JetBot must be rebooted for the changes to take effect. The JetBot will now boot in command line mode. This also disables autologin, which means a display and keyboard need to be connected to be able to enter login name and password. The default username and password are “jetbot”.

The GUI can also be enabled again.

```
sudo systemctl set-default graphical.target
```

After restarting again, the graphical user interface will be back on screen.

During testing the disabled GUI freed up approximately 600 Megabytes of RAM.

2.6.6 Connect to wireless access point from a terminal

In case that the GUI has been disabled on the JetBot to free up RAM, connecting to a new wireless access point is not as intuitive anymore. However, a connection can be established from the terminal with a few command lines. A keyboard and a display must be connected to the JetBot for this to work. John Welsh and Chitoku Yato of NVIDIA provide a guide for this matter on their JetCard repository (Welsh & Yato, 2019). They break it down to three command lines: Scan the available wireless connections, list them and connect to a specific one.

Welsh and Yato use the NetworkManager command line tool nmcli (The GNOME Project, 2014) in order to setup wireless connections. First a `wifi rescan` should be performed to refresh the networks that are known to NetworkManager (Welsh & Yato, 2019) .

```
nmcli device wifi rescan
```

This command will not show the available access points. To show them `wifi list` can be used, which will display a list of SSIDs.

```
nmcli device wifi list
```

Once identified an access point can be connected to with the `wifi connect` command. This assumes the password for that specific SSID is known.

```
nmcli device wifi connect <ssid_name> password <password>
```

The JetBot should then be connected to the network.

2.7 Example projects

This section will focus on existing projects for the Jetson Nano. This includes examples provided by NVIDIA as well as examples from other developers and makers. This should help to get an overview of what is possible with the platform. The examples demonstrate various use cases and provide code samples for interfaces. These examples have been relevant for selecting and building further use cases for the JetBot as described in chapter 3.

2.7.1 NVIDIA example projects

In this section the example projects provided by NVIDIA shall be described. Welsh and Yato of NVIDIA provide the code for a total of five example projects. Four of these have been documented on the official JetBot Wiki (Welsh & Yato, 2019f). These are also the ones that have been tested and executed as part of this project and which will be explained further in the following subsections.

NVIDIA provides the example projects in the form of Jupyter Notebooks. For each example Welsh and Yato provide a quick overview on the “Examples” Wiki page (Welsh & Yato, 2019f). They describe how to connect to the robot and where to find the notebooks for each example. For some projects they also include pre-trained models. During the testing of the examples the provided models proved to be very accurate and allow users to gain an overview of the JetBots capabilities quickly.

The code for the notebooks is located in the `notebooks/` directory within the JetBot Repository. Every example has its own subdirectory. In the directory `jetbot/` Python utility classes needed by the Jupyter Notebooks can be found.

Every example can be accessed from the JupyterLab Web-GUI like it is described in section 2.4.

2.7.1.1 Basic Motion

The first example project aims to show the movement capabilities of the JetBot as well as the general structure of the Jupyter Notebooks (Welsh & Yato, 2019a). It introduces the way of working with notebook cells and explains some of the Python principles such as its package structure. The code can be found in the `basic_motion/` directory in the `basic_motion.ipynb` notebook.

The notebook explains how the `Robot` class of the `jetbot/` package needs to be used. The class acts as an interface to the Adafruit motor controller. It provides functions to steer the robot into a certain direction or stop it completely. The speed is set as a percentage of the maximum speed capability of the motors. As an example, the code `robot.left(speed=0.3)` lets the robot rotate to the left with 30% of the maximum speed (Welsh & Yato, 2019a).

Welsh and Yato also give a quick overview on how to link the motor functions to graphical widgets. This allows users to steer the JetBot from a GUI instead of Python code level.

2.7.1.2 Teleoperation

The second example project focuses on operating the JetBot remotely. It explains the usage of the camera and how to access the video feed. The usage of a game controller to steer the JetBot is also covered. The code for this project is in the `teleoperation/` directory in the `teleoperation.ipynb` notebook.

Welsh and Yato make use of the `Controller` widget that is already included as part of the Jupyter Widgets (Project Jupyter, 2017). This makes it easy to connect a gamepad and remote control the JetBot like an RC car.

Later in the notebook they also introduce the `Camera` class of the `jetbot/` directory. The camera produces only values in BGR8 (blue, green, red, 8bit) format (Welsh & Yato, 2019a). Thus, the image needs to be converted in order to display it correctly on an image widget. For this the NVIDIA engineers provide a converter function `bgr8_to_jpeg` that wraps the image conversion code inside `image.py` in the `jetbot/` directory into a better understandable function (Welsh & Yato, 2019a) Once the camera is setup the robot can be controlled across rooms by navigating through the video feed.

2.7.1.3 Collision avoidance

The third example project uses an image classification model to prevent the JetBot from driving into dangerous situations. The JetBot will move around by itself and decide based on the camera output if the way is `free` or `blocked` (Welsh & Yato, 2019f). Welsh and Yato provide two approaches to test this example. Developers can either collect all data and train the model by themselves or use a pretrained collision avoidance model provided by NVIDIA (Welsh & Yato, 2019f). The goal is to create a model that prevents the robot from falling off a ledge and from hitting other objects in its path (Welsh & Yato, 2019a). The code can be found in the `collision_avoidance/` directory.

The example consists of three Jupyter Notebooks. The first one `data_collection.ipynb` provides the code to collect and label images which then will be used later for the training. The data is collected directly with the JetBot camera. (Welsh & Yato, 2019a).

Once the data has been collected the training can be started with the notebook `train_model.ipynb`. It is intended to run this notebook on a more powerful machine than the Jetson Nano to increase the training speed. NVIDIA uses the PyTorch framework for this task (Welsh & Yato, 2019a). NVIDIA selected the `alexnet` model from `torchvision` for transfer learning. Welsh and Yato also explain how to repurpose the model for the current use case as well as how to execute the training on the GPU with CUDA. Finally, the model can be trained. By default, it will be saved as `best_model.pth`.

In the third notebook `live_demo.ipynb` the previously created model or the model provided by NVIDIA can be used to teach the robot about potentially dangerous situations. Welsh and Yato use PyTorch for the application of the model (inference) as well. The notebook describes how to load the model and how to move it to the GPU for execution. As the camera format does not exactly match the format of the training data, each camera frame needs to be reformatted with a preprocessing function before it can be loaded into the model. For example, the color format needs to be converted from BGR (blue, green, red) to RGB (red, green, blue) (Welsh & Yato, 2019a). They then create a change listener that attaches the inference function to arrival of a new camera frame.

The trained model is able to perform a binary classification by predicting whether the current frame belongs to the class “blocked” or “free”. No further machine learning techniques are applied such as predicting the location of obstacles. When the collision avoidance outputs the status “free” the JetBot will move forward. When the collision avoidance outputs the status “blocked” the JetBot rotates left. Figure 6 visualizes the concept of this loop.

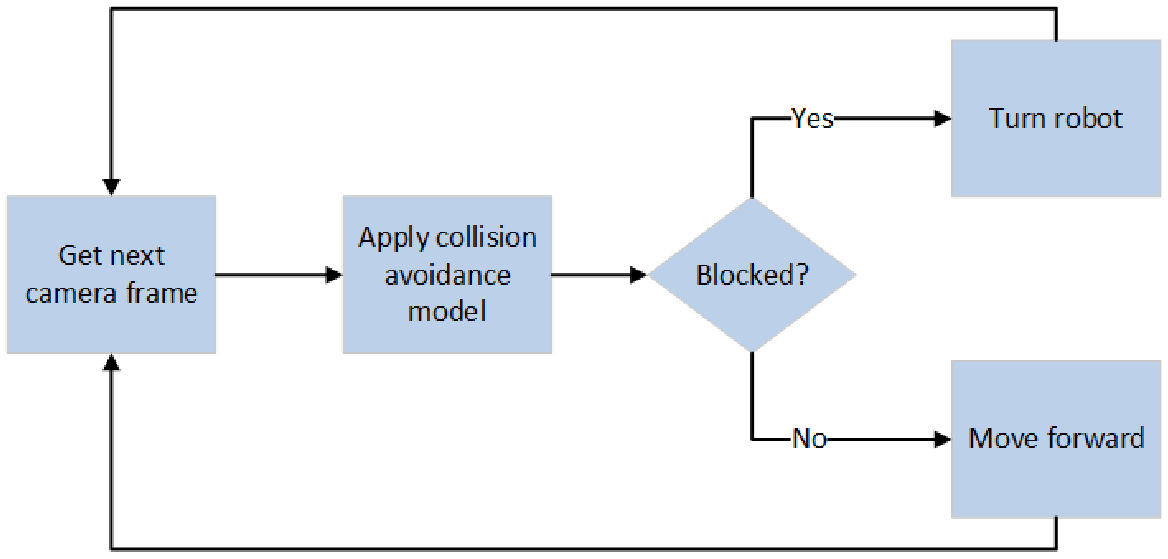


Figure 6: Collision avoidance loop

2.7.1.4 Object following

This example aims to teach the JetBot how to follow an object out of the COCO dataset (Welsh & Yato, 2019f). According to Lin et al. COCO “The Microsoft Common Objects in Context” provides 91 different object categories trained on 328’000 images (Lin, et al., 2015). Among other objects, it can detect people, cups or dogs. For this example, Welsh and Yato provide a pretrained model based on SSD MobileNet V2 that needs to be uploaded to the JetBot.

The sample project consists of only one notebook `live_demo.ipynb`. It can be found in the `object_following/` directory. The provided model is optimized with TensorRT and is available as a TensoRT engine. NVIDIA provides the `ObjectDetector` class, that is capable of importing TensorRT models. NVIDIA explains that “It also takes care of preprocessing the input to the neural network, as well as parsing the detected objects” (Welsh & Yato, 2019a). However, this only works for engines that have been created by using the `jetbot.ssd_tensorrt` package (Welsh & Yato, 2019a). Welsh admits that their engine creation code is a bit “hacky” and subject to fail if the implementation of certain external dependencies changes.

The example allows the JetBot to detect multiple objects and output the label, confidence and the bounding box of each object onto the video feed. The developer can select an object class, which the robot would then follow. Additionally, the collision detection model from the previous examples will be loaded to avoid hitting obstacles. The robot displays detected objects in blue while the object that it is currently following is drawn in green (Welsh & Yato, 2019a).

2.7.2 Other example projects

Some inspiration for the demonstration case of this project also came from projects featured on the NVIDIA Jetson Community Projects website. NVIDIA encourages developers to share their projects there by awarding the best repositories and their developers and handing out prizes (NVIDIA, 2020d).

One of these repositories is “ShAIdes” by Nick Bild (NVIDIA, 2020d). He uses image classification to control his home devices with gestures. Bild describes his project on GitHub as follows:

> “A small CSI camera is attached to the frames of a pair of glasses, capturing what the wearer is seeing. The camera feeds real-time images to an NVIDIA Jetson Nano. The Jetson runs two separate image classification Convolutional Neural Network (CNN) models on each image, one to detect [classify] objects, and another to detect [classify] gestures made by the wearer.
>
> When the combination of a known object and hand is detected [classified], an action will be fired that manipulates the wearer's environment. For example, the wearer may look at a lamp and motion for it to turn on. The lamp turns on.” (Bild, 2019)

The idea to use gestures to control the devices in your home seems simple and obvious. It is intuitive and effective. It allows users to control the Jetson Nano without the need of additional tools that may or may not be at hand at the time of execution. Different kinds of gestures can be mapped to different behaviors. This allows for a wide range of control mechanisms. Bild builds a rather simple gesture control pipeline. Two classification models are used for inference. One classifies whether an arm is visible on the image or background. The second model classifies whether a TV or a lamp or a background is visible on the image. When a combination of an arm respectively a TV is classified an action like switching on the lamp is executed. PyTorch is used for training and inference in this example.

Another project featured on the Jetson Community Projects website is “Rock-Paper-Scissors” by Alan Mok. Similar to “ShAIdes” it allows users to control the Jetson Nano with gestures. Mok restricted himself to three gestures: “Rock”, “Paper”, “Scissors”, that are needed to play the child’s game with the same name (Mok, 2019). The Nano will recognize your gesture and will answer randomly with one of its own. The results are then presented in the console. Mok trained his model with only 50 images per class (gesture). This project uses PyTorch as machine learning Framework.

The “Finding-path-in-maze-of-traffic-cones” project by Dimitri Villevald is one of the projects featured on the Community Projects website that uses the JetBot repository as a base (Villevald, 2019). Villevald extended the collision avoidance example by NVIDIA to navigate through a traffic cone maze and find its way out. He trained the JetBot to not only recognize if it is blocked or not but also to decide if it should turn left or right. His idea remained close to the original example and used the `alexnet` model for classification.

3 Demonstration case “Gesture Control”

One goal of this project is to build an example case for the programmatic control of the JetBot. The JetBot should be able to solve a simple task by leveraging its machine learning capabilities including object recognition. This example shall serve as a demonstration of the capabilities of the JetBot. Along with already available examples described in the previous chapter one additional demonstration case has been developed and described in this thesis. This chapter describes the requirements and selection of this demonstration case.

From the perspective of used hardware, the demonstration case shall include motion control and utilize the available video stream and motors. No further sensors shall be used. The machine learning part shall detect multiple objects within an image and classify them. The demonstration case shall be something similar to the already available example described in subsection 2.7.1.4. This example enables the JetBot to detect multiple objects from the COCO dataset (Welsh & Yato, 2019f) and steer towards one of these objects.

After assessing several suitable demonstration cases like detecting Lego bricks, faces or handwritten digits as well as analyzing publicly available datasets and the general feasibility of the demonstration case, gesture control has been selected as a preferred example.

An advantage of the JetBot compared to other AI solutions is that the JetBot is able to move around. Therefore, one of the primary objectives must be to find a solution to control its motion. There are options like steering it via buttons within a Jupyter Notebook or a gaming controller. The downside of these options is the need of additional hardware, which is not always available. Additionally, those methods do not use any machine learning, which is the strength of the JetBot. Hence, it makes sense to develop a system to control the JetBot with no additional hardware utilizing its machine learning capabilities. Many of the existing examples like described in the subsection 2.7.2 include gestures but none of them include a detection step, which enables localization of a gesture and detection of multiple gestures simultaneously.

The demonstration case “gesture control” shall work as follows: The JetBot detects hands via the video feed if there are any and recognizes any known gestures. When a person shows a predefined gesture the JetBot executes an action accordingly, such as driving towards this gesture. Furthermore, a certain level of robustness in this developed system must be achieved. It would be irritating to control the JetBot via gestures if there were many detection or classification errors and the JetBot would drive into an unwanted direction. Also, very important is the security while driving. For instance, it would be fatal if the JetBot would drive over the edge of a table or knock over a cup. Therefore, the gesture control shall include an active collision avoidance system simultaneous to the gesture detection.

The final gesture control demonstration case works as shown in Figure 7. The JetBot utilizes three different models. One detects hands, a second one classifies the detected hands into known gestures and a third one monitors if a collision is ahead. The detections from the classification and collision avoidance model are used to execute a certain action. Those actions can be freely defined and attached to any of the detected gestures or depending on whether the road is free or blocked. The collision avoidance model has been taken from NVIDIA. It is same that is being used in the collision avoidance example described in subsection 2.7.1.3.

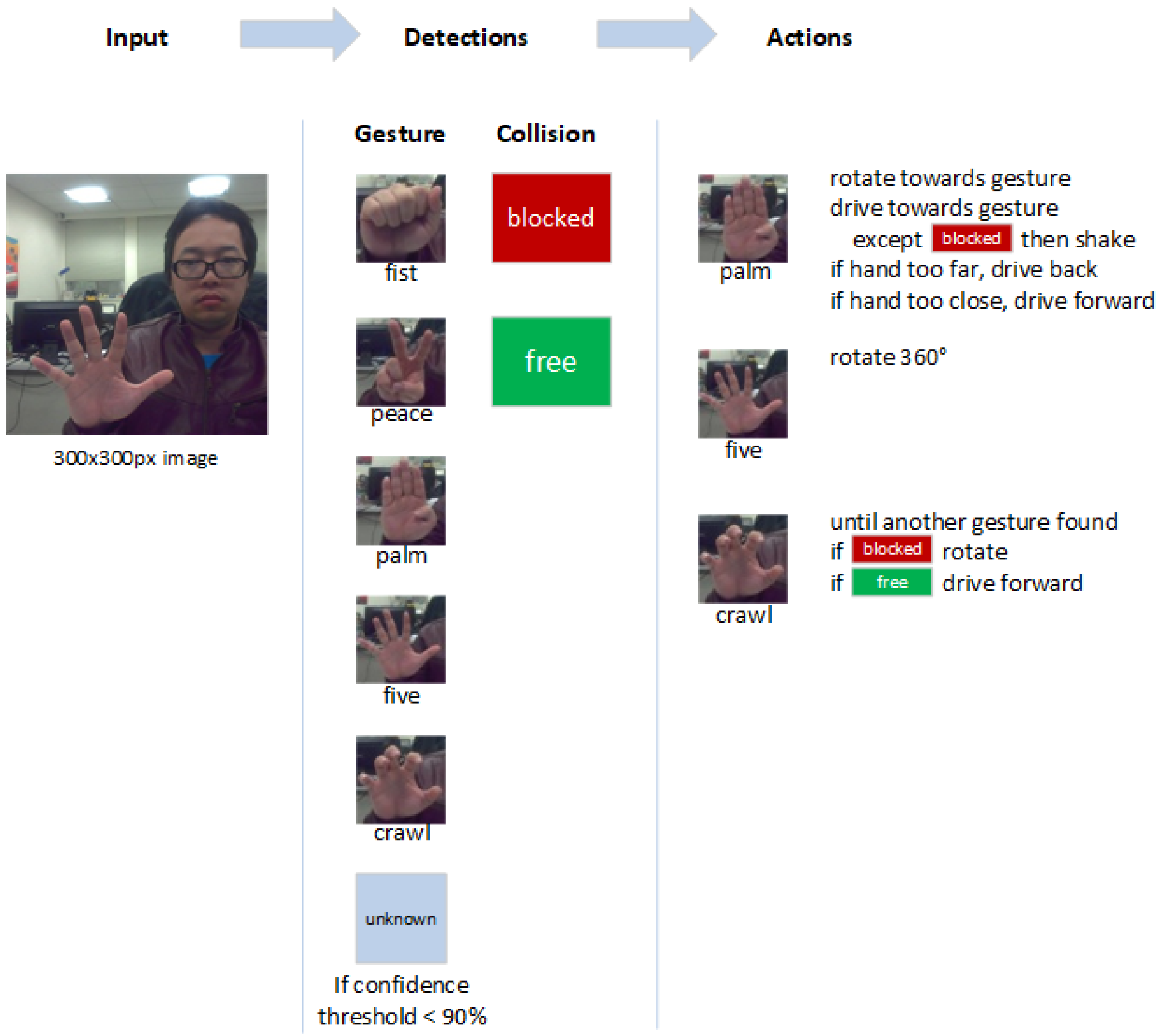


Figure 7: Overview of the gesture control pipeline with LaRED example (Sanchez-Riera, et al., 2015)

An input image can contain more than one hand. All the hands are being detected and also classified into gestures. To keep the demonstration case simple, only one gesture per image is used to execute an action. When multiple gestures are detected the gesture occupying the biggest area is considered the main gesture and is used to determine the action to fire. A gesture is considered found when the classification confidence score is above 90%. Otherwise the gesture is discarded. Three actions have been defined for the demonstration case. If the JetBot detects the gesture "crawl" it starts to move around freely and rotates if a collision is detected. If "five" is detected the JetBot rotates 360° and if "palm" is detected the JetBot follows this gesture by driving towards it when the gesture is far away or driving back if the gesture is too close. Should the JetBot be forced to drive into a collision it ignores the gestures and shakes shortly to communicate that a collision is ahead. Those actions can be easily redefined, attached to other gestures or even mapped to multiple gestures.

The following chapter describes the development of the hand detection and gesture classification module for the demonstration case including the current state of the art methods, the design of the architecture and the implementation details.

4 Hand detection and gesture classification

This chapter covers the design and implementation of the demonstration case gesture control. It describes some aspects of the current research in the area of gesture detection and the selected detection architecture. Furthermore, the most relevant implementation details are elaborated and finally the system performance is evaluated. This chapter is mainly based on a paper called “A Deep Learning-Based End-to-End Composite System for Hand Detection and Gesture Recognition“ by Mohammed et al. (Mohammed, et al., 2019) and “Impact of machine learning techniques on hand gesture recognition” by Idoko et al. (Idoko, et al., 2019). Both have been published in the fall of 2019, which coincides with the start of this thesis project. Those papers tackle the task of developing a robust gesture detection system in general considering state of the art methods. These research results will be applied to the limited capabilities of Jetson Nano.

Besides using gestures to control the motion of the JetBot, gesture recognition is being used in a wide range of applications such as controlling computer software or smart devices like TVs, sign language communication or virtual reality environments. This underlines the relevance of the selected demonstration case gesture control. However, according to Mohammed et al. gesture recognition is a very demanding task, because of the many variations of a hand in terms of view and form. The paper goes even further and states that a “Robust hand gesture detection and recognition in cluttered environments […] presents a challenging problem that has not yet been solved in computer vision and machine learning.” (Mohammed, et al., 2019, p. 1) It brings with it, that there is not much publicly available training data and to our knowledge there are no already developed models ready for usage capable of detecting and classifying multiple gestures. On the other hand, both Mohammed (Mohammed, et al., 2019) and Idoko (Idoko, et al., 2019) are proposing a system which seems to be robust enough and with some adjustments also fast enough to be suitable for the JetBot. More details about the selected architecture are described in the following sections.

4.1 Related works

This section is based on the explanation of (Mohammed, et al., 2019, pp. 3-5) and explains the proposed approaches so far to tackle hand detection and gesture classification. Those approaches can be grouped into four categories.

1. Skin based detection: Skin detection algorithms rely on skin color segmentation trying to separate hands from the background using color hue, saturation and value. This method can be combined with other methods such as CNN achieving better results. In summary skin-based segmentation is not robust enough in practice as it is suffering from constraints such as skin tone variation, poor illumination or background clutter.
2. Depth-based detection. This method requires a specialized depth camera which is one of its biggest downsides. Studies show good results in well prepared environments.
3. Hand-crafted features: Hand crafted features show great success in face detection. A few approaches have been introduced to transfer these methods to hand detection but they still face issues due to the large variation of hands appearances. Combined methods with skin-based approaches show good results but suffer from slow performance.
4. CNN based approach: Convolutional Neural Networks show great results in image classification and subsequently numerous methods have been proposed for object detection using CNNs. Methods using the Faster R-CNN framework show good accuracy for challenging datasets but they suffer from high computational cost.

4.2 Gesture recognition pipeline

To robustly detect and classify multiple gestures on an image the pipeline contains two modules. It consists of a Single Shot Multibox Detection (SSD) for hand detection and a lightweight Convolution Neural Network (CNN) for gesture classification. More details on the selection of model architectures can be found in section 4.2. The same approach has been chosen by Idoko et al. and it showed good results in terms of accuracy (Idoko, et al., 2019, p. 49). Mohammed et al. use a custom build RetinaNet detection architecture and MobileNet as a lightweight CNN for gesture classification (Mohammed, et al., 2019, pp. 5-9) The SSD can detect multiple hands on an image. Each detected hand is processed as follows: The detection model outputs the bounding boxes for each hand. This means detection boxes can have different ratios. To unify the input for the classification model a minimum sized square around the hand is cropped and resized to 64 x 64 pixel (px). The 64 x 64 px cropped hand image is then passed to the lightweight CNN for classification. The CNN yields a gesture name if a known gesture is being recognized and surpasses a defined confidence threshold. The recognized gesture or gestures and their position within the image are then used to control the JetBot. The described pipeline is visualized in Figure 8.

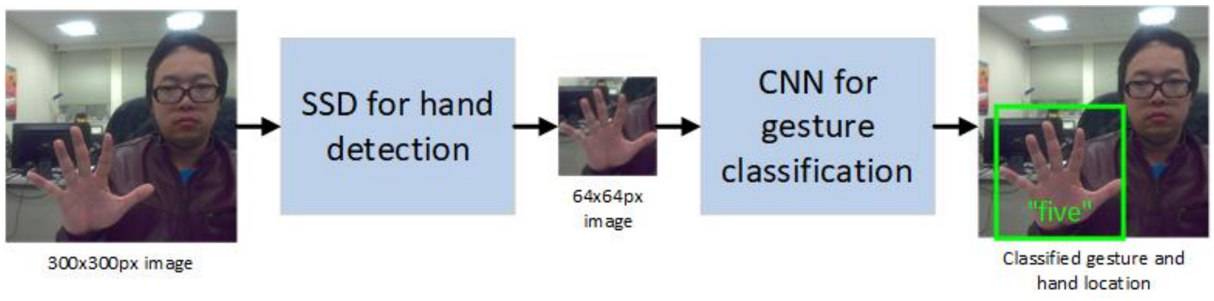

SSD for hand detection
64x64px image
CNN for gesture classification
"five"
300x300px image
Classified gesture and hand location

Figure 8: Gesture detection pipeline with LaRED example (Sanchez-Riera, et al., 2015)

To avoid random actions by the JetBot due to inaccurately recognized gestures both steps, the detection and the classification, use a confidence threshold before yielding a result. As a first step, a detected hand is only passed on for classification if the module is certain enough that the detected object is really a hand. It also means that if an image does not contain any hands, no gesture is being classified and no new action is being fired to control the JetBot. Secondly, only when the classification module is certain enough to have classified a known gesture it is passed on to execute a defined action. Hence, if an input image contains multiple hands, but none of them show one of the known gestures, no new action is being fired to control the JetBot. The aforementioned confidence thresholds have been empirically defined while working with the JetBot. The hand detection confidence threshold has been set to 80% and the gesture classification threshold has been set to 90%.

4.2.1 Hand Detection

The detection networks can be divided into two categories: One-stage and two-stage object detectors.

Two-stage detectors generate regions of interest in the first stage and the second stage subsequently classifies the proposals into foreground and background classes. Those two-stage detectors yield high accuracy but show slow inference speed. (Mohammed, et al., 2019, p. 5).

One-stage detectors like SSD and YOLO have been developed for fast inference but show lower accuracy. Those detectors skip the region proposal step for the detection of the foreground candidates (Mohammed, et al., 2019, p. 6). Since gesture control of the JetBot requires processing the images via the live video feed at a fast speed, an “inference speed first” approach and with this a one stage detector has been chosen. This is needed to be able to react to classified gestures fast enough. Idoko et al. also have chosen an SSD to detect hands (Idoko, et al., 2019, p. 43). They used the original SSD architecture invented and described by Liu et al.

in 2016. Testing with the Pascal VOC dataset and using the SSD300 architecture, which takes images of size 300 x 300 px, the authors achieved a mean average precision (mAP) of 74.3% with 46 frames per second (FPS) on a NVIDIA Titan X GPU. This outperforms Faster R-CNN (7 FPS and mAP 73.2%) and YOLO with a VGG16 backbone (21 FPS and mAP 66.4%) in both accuracy and speed (Liu, et al., 2016, p. 15).

Mohammed et al. (Mohammed, et al., 2019, pp. 5-8) propose a modified architecture to improve the accuracy of the one-stage detectors. Their implementation is not available as open source. Additionally, it would go beyond the scope of this thesis to implement such an architecture.

To further select the best architecture for detection improvements of the SSD architecture in the last years have been researched. Huang et al. perform an extensive comparison of the detection architectures Faster R-CNN, R-FCN and SSD using different backbone architectures. Their experiments show that replacing the VGG backbone of the SSD with a MobileNet backbone slightly reduces the average precision but clearly increases the inference speed. SSD with a MobileNet backbone uses almost three times less GPU time than SSD with VGG backbone. As a tradeoff the mean average precision on the COCO dataset drops from a little less than 22% to 20% (Huang, et al., 2017, pp. 7,11). The experiments from the original MobileNet paper show similar results. The mean average precision drops from 21.1% to 19.3% but reducing the computational complexity by a factor of 29 (Howard, et al., 2017, p. 7).

Sandler et al. introduced a new mobile architecture called MobileNetV2 which outperforms MobileNetV1 in both inference speed and accuracy. Testing on the ImageNet MobileNetV2 it achieved an accuracy of 72% compared to 70.6% for MobileNetV1 and an inference speed increase from 113 ms to 75 ms on a CPU (Sandler, et al., 2019, p. 7). Also, according to Huang et al. an SSD architecture using MobileNetV2 achieves a mean average precision of 22% with 31 frames per second compared to a mean average precision of 21% at 30 frames per second when using MobileNetV1 as backbone (Huang, et al., 2019). Those tests have been performed on the COCO dataset.

After the above described research, an SSD architecture with MobileNetV2 as backbone architecture has been selected for the hand detection module.

4.2.2 Gesture classification

When a hand is detected it is cropped, resized to a size of 64 x 64 pixel and passed on to the classification model to recognize any known gestures. Both, Mohammed et al. (Mohammed, et al., 2019, p. 1) and Idoko et al. (Idoko, et al., 2019, p. 44) use a lightweight CNN for classification. We have selected the same CNN architecture as Mohammed et al., namely MobileNet. It offers a good tradeoff between speed and accuracy. (Mohammed, et al., 2019, p. 9). “MobileNets are based on a streamlined architecture that uses depthwise separable convolutions to build light weight deep neural networks.” (Howard, et al., 2017, p. 1).

Bianco et al. have conducted a comparison of different state of the art Deep Neural Networks and summarized the results in their paper. MobileNet shows a good accuracy similar to VGG16 but uses a much smaller model and a fraction of computational cost (Bianco, et al., 2018, p. 273). Additionally, MobiletNet is part of already available models in Keras (Chollet, 2015) which therefore has been used to develop the classification model.

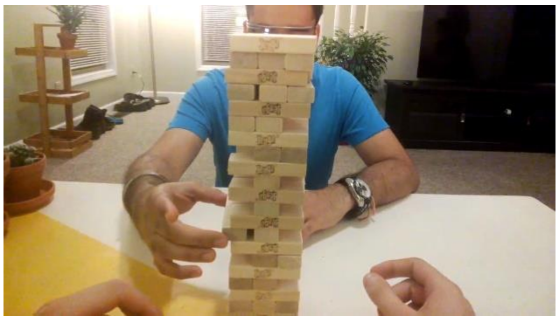
4.3 Datasets

Two kinds of datasets are required to train the described gesture control pipeline. To train the hand detection model images of hands containing information about their location are necessary. For the training of the gesture classifier images of gestures with their labels are required.

4.3.1 Hand detection datasets

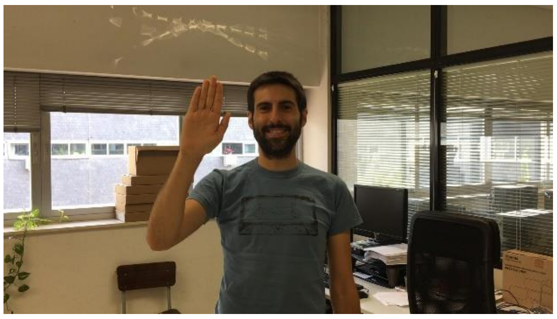
Hands can have many shapes depending on the perspective and gesture performed and therefore a wide-ranging dataset is indispensable. Mohammed et al. have created their own dataset for hand detection collecting and manually annotating a total of 24'535 images with over 41'000 hand instances (Mohammed, et al., 2019, pp. 2,10). Their dataset has not been publicly shared. The limited resources of this thesis do not allow such an extensive data collection. However, due to the importance of a wide-ranging dataset a considerable amount of time has been invested into searching, partially manually annotating and preparing different datasets. This resulted in a total of 8'325 images and 13'200 hand instances whereof 2'773 hand instances have been manually annotated. The detection dataset consists of three publicly available datasets:

1. EgoHands (Bambach, et al., 2015): This dataset contains a total of 4'800 images and over 15,053 ground truth annotations and is available at http://vision.soic.indiana.edu/projects/egohands/. The images have been taken from 48 Google Glass videos in which two people are performing different activities like playing chess in different environments. For the purpose of this thesis a subset of 3169 images and 7838 hand instances have been used. The other instances have been filtered out since many hands were only partially visible like only one finger which is not what the model in this work should learn since a gesture can only be recognized when its fully visible. The images have been cropped to a square form for training purposes so that no image squeezing is necessary. Therefore, some of the pictures were discarded where the hands occupied more than a square area on the image. Subsequent, the dataset has been divided into two subsets, 80% were used for training and the remaining 20% for validation.

   
   Figure 9: Example image EgoHands dataset (Bambach, et al., 2015)

   Relevant to mention is that, experiments were performed where the detection model has been trained only on this dataset. Manual tests with a video stream using a web camera have shown that hands were detected well when they remained on a table and the fingers were in a resting position. However, when showing gestures like a palm or a fist the hands were not recognized as such. This makes sense, since the dataset does not contain any instances of gestures. Consequently, more datasets are necessary to be able to control the JetBot with gestures.

2. TinyHands (Bao, et al., 2017): Since the dataset has no name in the original paper we will call it TinyHands as proposed by Mohammed et al. (Mohammed, et al., 2019, p. 10). The dataset itself is available at https://sites.google.com/view/handgesturedb/home. It contains images of 7 hand gestures collected from 40 people. The images have been taken in different locations with two setups. One setup is more rudimentary, where people are sitting in front of a gray wall so that the upper part of the body with the head is visible. The second setup is more complex where people are standing in front of a cluttered background. The hand occupies about 10% of the pixels in the image. Every gesture for each person is composed of about 420 to 1'800 pictures. In totality, there are about 260'000 images. This dataset is very promising for training the hand detection model since the subjects are showing gestures in front of cluttered environments. Detecting gestures in such cluttered environments is crucial for a reliable

   
   Figure 10: Example image TinyHands dataset (Bao, et al., 2017)

gesture control of the JetBot (Mohammed, et al., 2019, p. 4). Unfortunately, this dataset does not include any information about the location of the hands. Therefore, a subset of the images has been manually annotated with bounding boxes. This subset contains 2'771 images. This seems like very few compared to the 260'000 total images. On the one hand, annotating images manually is resource and time intensive. On the other hand, one instance of a hand composed of about 2'000 images was recorded with a video and therefore most images are almost identical. For each gesture instance 10 out of the 2000 available images were taken and annotated with bounding boxes, which leads with 7 gestures and 40 subjects to the total of about 2800. An open source tool called labelImg has been used for annotating (Lin, 2015).

3. LaRED (Sanchez-Riera, et al., 2015): To further improve the detection of hand gestures the LaRED dataset has been partially added to the combined detection dataset. Relevant to mention here is that this dataset has also been used to train the classification model. The LaRED dataset contains 243'000 color images with 27 gestures in 3 different rotations leading to a total of 81 classes. The pictures have been taken from a total of 10 subjects recording 300 images per gesture. According to Mohammed et al. it is the largest open source hand gestures dataset that is currently available (Mohammed, et al., 2019, p. 10). In addition to the RGB image a depth file and a mask file generated from the depth file is provided. We have used the mask file to extract bounding boxes to train the detection model. The downside of using the mask generated form the depth file is the insufficient accuracy of the mask file. Often a small part of the forearm is considered as hand which is inconsistent with the two datasets described above. Advantageous is the versatile background noise and the challenging illumination of the images. Also, some of the hands take a rather large part of the whole picture in contrast to the two other datasets. The addition of this dataset should enable the JetBot to recognize gestures in challenging light conditions. To match the size of the TinyHands dataset a subset of 2430 images out of the 243'000 available has been selected. Only 5 of the available 81 gestures have been used. These are the same gestures as were later used to train the classification model.

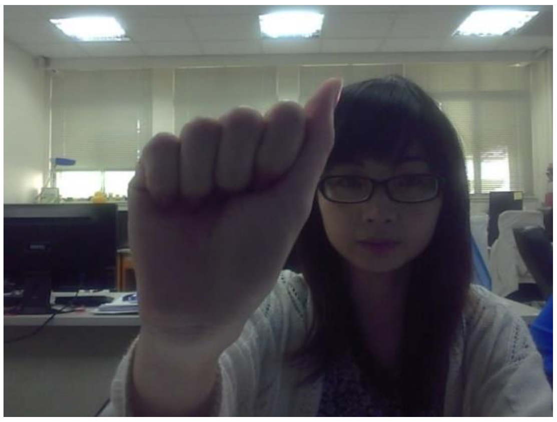


Figure 11: Example image LaRED dataset (Sanchez-Riera, et al., 2015)

The three datasets have been merged containing a total of 8352 hand images and 13'020 hand instances annotated with bounding boxes. 80% of the images have been used for training and 20% for validation. The final model was trained with an image size of 300 x 300 px which allows the JetBot to process the images with an acceptable speed. The source code of all the data preparation and transformation steps is documented and well prepared for being reused and adjusted and can be found in the thesis repository in the `howto/` directory.

4.3.2 Gesture classification datasets

To train the classification model images of gestures labeled with the gesture name respectively gesture id are required. In the final gesture control pipeline, a hand is detected, cropped and passed on for classification. This means that the hand will occupy almost the whole image. Therefore, for training the classification dataset the gestures must be cropped already. Additionally, the images must be taken in front of considerable background noise, since to reliably control the JetBot it should be able to classify gestures in as many scenarios as possible.

Examples include poor illumination or a partially visible human face. The LaRED (Sanchez-Riera, et al., 2015) dataset fulfills these requirements well and has been chosen to train the classification model. The same dataset has been partially used to train the detection model and is described in the subsection 4.3.1. The dataset consists of 240'000 images taken by 10 subjects performing a total of 81 gestures in 300 images per gesture. For the purposes of this thesis, 5 gestures were chosen to control the JetBot which resulted in a total of 15'000 images. Besides challenging background noise and illumination the dataset offers another advantage with the available masks. Those make it possible to crop the gesture for training which corresponds to the format in the gesture control pipeline. For training purposes, a square form around the gesture is cropped with the help of the mask and subsequently resized to a uniform size of 64 x 64 px. This size has been chosen since the gesture occupies only a small part of the picture and makes a good tradeoff between speed and accuracy.

4.4 Implementation details

This section presents the methods and tools how the classification and the detection model have been trained. It also explains the environment and its setup.

There were two different environments on two different hosts in use for training the models. This had the advantage of being able to train the classification and the detection model simultaneously. The classification model has been trained on a notebook with Windows 10 and a NVIDIA GTX 1050 Ti GPU with 4 Gigabytes of Video-RAM (VRAM). The detection model has been trained on a desktop computer with an Ubuntu 18.04 LTS installation and a NVIDIA GTX 970 GPU with 4 Gigabytes of VRAM. Both systems had CUDA 10.2 and cuDNN 7.6.5 installed. It took approximately 27 hours to train the detection model with this setup. The classification model can be trained in roundabout 80 minutes. With k-fold validation this spiked to ca. 15 hours.

The training of the classification model used TensorFlow 2.0 as machine learning framework while the detection model has been trained with TensorFlow 1.14 due to compatibility reasons.

4.4.1 Implementation details hand detection

This subsection provides a deepened description of the used frameworks, tools and configuration for training the hand detection model. In subsection 4.3.1 the used datasets have been described and in subsection 4.2.1 the selection of the model architecture is elaborated, namely an SSD architecture with MobileNetV2 as a backbone.

To implement the hand detection model the TensorFlow Object Detection API has been used (Huang, et al., 2017). The source code is available at https://github.com/tensorflow/models/tree/master/research/object_detection.

TensorFlow Object Detection API is an open source framework built on top of TensorFlow. Its aim is to make it easy to construct and train object detection models. The repository is well maintained with new releases coming out every few months containing tools to train with state-of-the-art detection architectures. The downside of the TensorFlow Object Detection API is that it does not come as part of TensorFlow itself and is meant for research purposes as it has experimental features that are not well documented and coordinated with the TensorFlow framework. As an example, the training of detection models could only be performed with TensorFlow version 1 and the inference documentation is written for TensorFlow version 2.

To train a detection model a pipeline must be configured that defines all the relevant training parameters such as the training data and duration of the training (Huang, et al., 2018).

Subsequently, the selected training configuration is discussed.

The detection dataset defined in subsection 4.3.1 has been used for training. TensorFlow Object Detection API requires a specific data format called TFRecord (Huang, et al., 2019a).

The configuration of the training pipeline happens in one single configuration file. Sample configurations for various model architectures are available on GitHub at https://github.com/tensorflow/models/tree/master/research/object_detection/samples/configs

For this thesis the SSD - MobileNetV2 configuration has been chosen and adjusted for the hand detection dataset. The whole configuration file along with all necessary artefacts to execute a training can be found in the thesis repository in the `howto/` directory.

The input size of the images is 300 x 300 px. Should an image of arbitrary size be passed to the final model it resizes it automatically to 300 x 300 px. The maximum of detected boxes can be limited by a minimum confidence threshold, a minimum intersection union score or a maximum number of boxes. In our experiments the number of detected boxes has been limited to 4. Returning hand instances from the model with an unrestricted confidence threshold allows experiments by setting the threshold later in the pipeline in the source code. Following these experiments, the confidence threshold has been set to 0.8. With a confidence threshold of 0.8 the JetBot detects hands also from a reasonable distance while rarely mistaking hand-similar objects like ears or noses for hands. As a future improvement this threshold could be added to the model configuration allowing further filtering on GPU, since currently the filtering of the four fixed returned hand instances occurs on CPU.

The batch size has been decreased to 12 for memory reasons during training. The already preconfigured RPSProp optimizer has been used with an initial learning rate of 0.004. Some experiments were conducted changing the optimizer to Adam and using other learning rates but none of the experiments showed better results as the already preset parameters. Unfortunately, the TensorFlow Object Detection API does not offer the option to decay the learning rate on plateau. Therefore, a fixed learning rate decay by a factor of 0.1 has been defined after a fixed amount of training steps, namely 100’000. TensorFlow Object Detection API does not use the metric epoch like Keras but only the training steps, where a training step is the process of one training batch.

The model has been initialized with pretrained weights from the coco dataset. In TensorFlow Object Detection API this happens by loading a so-called checkpoint which can be downloaded directly from the repository (Huang, et al., 2019).

To reduce overfitting several data augmentation techniques offered by the TensorFlow Object Detection API have been applied. Those augmentation techniques have to take into account, that along with the image also the bounding boxes have to be altered accordingly. The following augmentation techniques have been applied: Random rotation by 90-degree, random addition of black patches, random contrast adjustments and a technique called random SSD crop. The random SSD crop randomly crops the images by never cropping the bounding boxes of available hand instances. Unfortunately, there are no augmentation options to randomly rotate by a few degrees or to randomly distort the images.

All the possible data augmentation techniques offered by the TensorFlow Object Detection API can be found at https://github.com/tensorflow/models/blob/master/research/object_detection/protos/preprocessor.proto.

More details about the evaluation metrics can be found in section 4.5.

4.4.2 Implementation details gesture classification

The classification model has been developed with Keras, a deep learning Python library (Chollet, 2015) using the TensorFlow backend (Abadi, et al., 2015). The training has been conducted on five gestures from the LaRED dataset using 81% of the data for training and 19% for testing. The selected gestures, namely "crawl", "fist", "five", "palm" and "peace" can be seen in Figure 12.

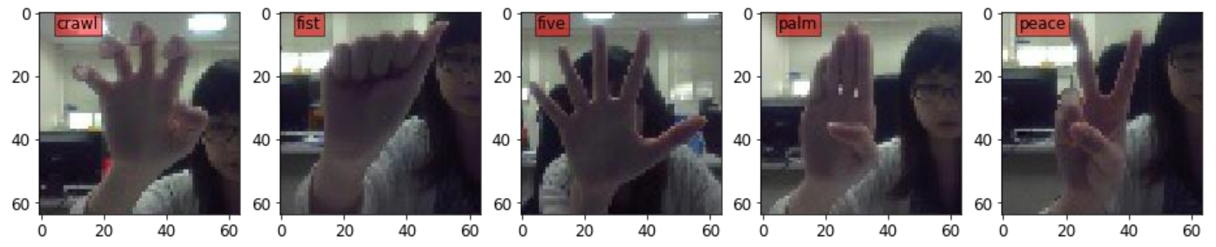


Figure 12: 64x64 pixel images of the classification model gestures

Half of the test set consists of images taken from one dedicated subject not used in the training set and another half of randomly chosen images taken from the remaining 9 subjects. To tune the hyperparameters a technique called k-fold validation has been applied. Since there is a total of 2430 training images per gesture and those can be additionally rather similar to each other, taking a random portion of images for validation may result in high variance depending on the validation split resulting in an inaccurate validation score. K-fold cross-validation diminishes this effect by splitting the training data into k partitions then training on k-1 partitions and validating on the remaining one. Usually k is 4 or 5 (Chollet, 2018, pp. 87-90). In this case the training set was split into 9 partitions, since there are 9 subjects in the training set. This allows to train on images from 8 subjects and validate on the remaining one, who was not used during training. After training on all 9 partitions the average validation score is calculated and used for a final training run on all training data using the best hyperparameters.

As mentioned in subsection 4.2.2 a lightweight CNN architecture called MobileNet has been used (Howard, et al., 2017). MobileNet is implemented as part of the Keras library (Chollet, 2015). The model takes an image of shape 64 x 64 x 3 where 3 means the number of color channels. Additionally, Keras offers the possibility to initialize the model with pretrained COCO weights. Unfortunately, those are available for an input size of 128 x 128 px or bigger only. The whole model architecture has been used including the fully-connected layers with a softmax classifier which yields probability scores for the five available gestures. Same as Mohammed et al. the Adam optimizer with an initial learning rate of 0.001 and a minibatch size of 32 has been used to train the model (Mohammed, et al., 2019, p. 12). Since the problem type is a multiclass single label classification, categorical cross entropy has been used as loss function (Chollet, 2018, p. 114).

Chollet proposes four main ways to prevent overfitting: More training data, reduce the capacity of the network, weight regularization and dropout (Chollet, 2018, p. 110).

More training data has been artificially introduced by using data augmentation. Here different techniques have been applied to enable the model to correctly recognize gestures under challenging conditions. The image is being randomly flipped horizontally and rotated up to 20 degrees to recognize slightly rotated hands and also left hands, since the dataset contains only right-hand gestures. In addition, the images are being slightly and randomly shifted in width and zoomed in or out for the case where the detection model does not align the bounding box perfectly. Lastly, the brightness of the input images is being randomly manipulated to allow to recognize gestures under challenging conditions.

To reduce the capacity of the network MobileNet introduces a parameter called "width multiplier" or alpha in Keras. The width multiplier thins out net network layers uniformly at each layer. Setting the width multiplier to 0.5 reduces the number of parameters from 4.2 million to 1.3 million and reduces the computational complexity by a factor 4. Howard et al. performed tests which show that as a tradeoff the accuracy on the ImageNet dataset has been reduced from 70.6% to 63.7 % (Howard, et al., 2017, pp. 4,5). In the case of the gesture classifier used for this thesis reducing the network width helped to reduce overfitting.

As the third and last measure against overfitting a dropout of 0.5 has been applied. Adding weight regularization as proposed by Chollet (Chollet, 2018, p. 110) is not supported for the MobileNet model out of the box in Keras and has been not prioritized.

During the k-fold training a technique called "reduce learning rate on plateau" has been applied. The learning rate has been reduced if the validation loss did not improve for 33 consecutive epochs.

4.5 Evaluation

The following subsections aim to describe the evaluation methods and results of the gesture recognition pipeline. Subsections 4.5.1 and 4.5.2 describe the evaluation of the detection respectively the classification model separately while subsection 4.5.3 is analyzing the gesture recognition pipeline as a whole.

4.5.1 Evaluation detection model

There are 1671 images in the detection test set. These images contain 2604 hand instances. The test set represents 20% of the total dataset and has been assembled by taking every fifth image out of the whole data set.

The model has been evaluated using the COCO detection metrics (Lin, et al., 2015a). TensorFlow Object Detection API offers the evaluation during training out of the box when an evaluation set is provided. The metrics can also be monitored during training with the help of TensorBoard (Abadi, et al., 2015).

We report a mean average precision (mAP) and average recall (AR). AP has been calculated with an intersection over union (IOU) of 0.5. Intersection of union means the overall percentage of the ground truth box and the predicted box. An IOU of 0.5 means an overlap of 50%. mAP is a popular metric to determine the performance of a detection model. For example, all pretrained TensorFlow detection API models have been evaluated with this metric (Huang, et al., 2019).

We have obtained an mAP of 97.5% and an AR of 79.3%. We have received this metrics on our custom detection dataset consisting of the three merged datasets TinyHands, LaRED and EgoHands as described in subsection 4.3.1. Therefore, these metrics cannot be directly compared with the results of Mohammed et al. The introduction to chapter 4 already mentions that there are only a few models available that are trained to detect hands. For example, the model zoo of the TensorFlow Object Detection API does not contain a single model that covers the topic of detecting hands (Huang, et al., 2019). However, an mAP 97.5% is a very good value that means that almost all hand predictions are correct. An AR of 79.3% is lower but is still a good value and means that most hand instances have been detected. For the use case of controlling the JetBot with gestures it means that the detected hands are almost always correct. This allows for a reliable gesture control. The fact that 79.3% of all hand instances are being detected means that 20% of the time hands are not recognized despite expectations. In that case the JetBot will not perform any action. Regardless, reliable gesture control is assured since only a few hands (2.5%) are wrongly detected as false positives.

To better be able to compare the results of our detection model an additional evaluation has been conducted evaluating only on the EgoHands dataset. Mohammed et al. have also tested their model on this dataset (Mohammed, et al., 2019, p. 13).

	AP (%)	**AR (%)**
Mohammed et al.	93.1	94.4
Ours	94.1	68.6

Table 4: Comparison of mAP and AR of Mohammed et al. and our model

Unfortunately, these numbers cannot be directly compared as well. There are some differences in the test setup. Mohammed et al. do not mention the input image resolution, but we assume that the highest possible resolution has been used in their tests. Our images are reduced to an input size of 300 x 300. Additionally, as described in subsection 4.3.1 the EgoHands dataset has been preprocessed by removing hand instances where the hand is visible only partially, e.g. only one finger.

Our mAP is very similar to the one of Mohammed et al. which assumes that both models can detect hands accurately. Our AR value is clearly below the AR of Mohammed et al. We assume the reason is that we used a smaller image size which makes it harder to detect small hands.

4.5.2 Evaluation classification model

The test set consists of a total of 2850 images. This corresponds to 19% of the total amount of images. 1500 of the test images have been taken from a dedicated subject that has not been included in the training set. This allows to test the model on a set of images of a person that it has not seen before. The remaining 1350 test images have been taken randomly from the remaining 9 subjects to add some more diverse images to the test set.

To assess the performance of the model a set of evaluation metrics provided by the Scikit-learn Python library has been used (Pedregosa, et al., 2011). The results for our model can be found in Table 5.

Accuracy	**Precision**	**Recall**	**F1-Score**
99.82	99.83	99.82	99.82

Table 5: Classification model metrics

There are no other model architectures that are trained on the LaRED dataset using the same subset of gestures. Therefore, a direct comparison cannot be performed. However, the results can be compared with Mohammed et al. taking the differences into account. The MobileNet classification model of Mohammed et al., trained on all 81 gestures of the LaRED dataset, achieved an accuracy of 97.25% (Mohammed, et al., 2019, p. 16). Training the MobileNet classification model on the simple images of the TinyHands dataset achieved an accuracy of 99.00% (Mohammed, et al., 2019, p. 15). The simple TinyHands dataset consists of images of 7 gestures that have been taken in front of simple backgrounds. Taking into account that our model has to classify only 5 gestures the accuracy of 99.82% is similar to Mohammed et al. (Mohammed, et al., 2019, pp. 15-16). The confusion matrix seen in Table 6 shows the model performance on the test set. The errors occurred when confusing the gestures "five" and "crawl". 5 "five" gestures have been mistakenly classified as "crawl".

		Predicted labels				
Actual		**crawl**	**fist**	**five**	**palm**	**peace**
	crawl	570	0	0	0	0
	fist	0	570	0	0	0

	five	5	0	565	0	0
	palm	0	0	0	570	0
	peace	0	0	0	0	570

Table 6: Classification model confusion matrix of the classification model

This can be explained as the two gestures "crawl" and "five" look very similar as seen in Figure 13. Additionally, by looking at the misclassified pictures shown in this figure one can see, that the "five" gestures have been slightly wrongly cropped during the image preprocessing and therefore the fingertips are not fully visible.

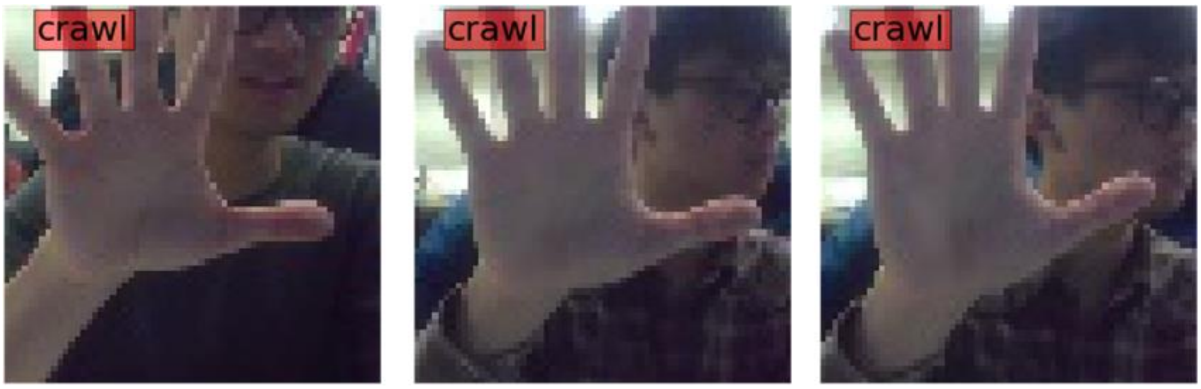


Figure 13: Misclassified gestures

In summary the classification results using MobileNet when training on the LaRED dataset are very good even if compared to the already good results by Mohammed et al. Related to the gesture control of the JetBot those results must be also relativized. The gesture classification has been trained and tested on only one dataset. It is a very good dataset containing much background noise and many images. However, classifying gestures in highly cluttered environments, changing illumination and possibly blurred images due to motion like in the case of the JetBot remains a challenging task. Random tests when controlling the JetBot with gestures using the complete gesture control pipeline have shown that the gesture classification works correctly in almost all cases. However, now and then a gesture is misclassified.

4.5.3 Evaluation end-to-end system

After testing the hand detection model and the classification model separately this section aims to evaluate the performance of both modules in conjunction. The test setup has been set up exactly like the inference setup when controlling the JetBot. An input image of size 300 x 300 px is passed first to the detection model. The detection model detects hands in the image if there are any present. Then the cutout is being resized to 64 x 64 px and passed on to the classification model. These hands are then classified as a gesture.

The selection of an appropriate test set poses a rather difficult challenge. For being able to compare the predicted and actual labels annotated images of a size of at least 300 x 300 px with gestures known to the classification model are needed, namely "crawl", "fist", "five", "palm" and "peace". The only dataset containing such information is the LaRED dataset itself.

To test the whole pipeline the same test set has been used as for testing the classification model. The only difference is, that not the already manually cropped and resized images have been taken but the uncropped images of size 300 x 300 px. Since both, the detection and classification model, have been trained on the LaRED dataset, it is important to mention that both models have never seen the test images during training.

One of the biggest downsides of this test set is that the hands occupy a rather large area in the image and therefore are easy to detect. So, this test has little significance for the detection model. But since the problem, that the pipeline is trying to solve, is very task-specific this dataset is still the best option to evaluate the pipeline.

The test set consists of a total of 2850 images. The same detection confidence threshold of 0.8 as used during the JetBot gesture control inference has been set. A very positive test result is that all but one hand instances have been detected correctly. It means that in each image exactly one hand has been detected. In one image no hands have been detected. This image is shown in Figure 14. If the detection confidence threshold is being set to or below 0.3, more than one hand instance is mistakenly being detected. With a threshold between 0.3 and 0.7 all hands are detected correctly. Setting the threshold between 0.7 and 0.94 the hand in Figure 14 is not detected. Setting the threshold above 0.94 multiple hands are not being detected. This means when looking at the present test set a threshold between 0.3 and 0.7 would be perfect. However, the empirically set threshold of 0.8 has been kept, based on experiments with the JetBot pipeline. When setting the threshold to 0.7 and running the pipeline on the JetBot objects like an ear or arm are mistakenly detected as hand from time to time.

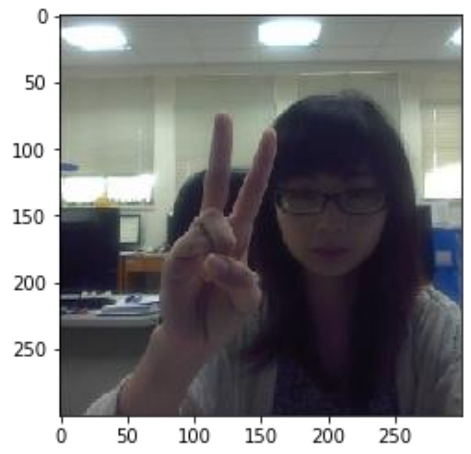


Figure 14: Undetected hand instance „peace"

The whole pipeline has been tested a similar way as the classification model. 2849 hands detected in 2850 images are passed to the classification model and the predictions are compared to the actual labels outputting a confusion matrix. The one undetected hand is marked as wrongly classified. As seen in Table 7 the same amount of classification errors, namely 5 in total, occurred when classifying detected hands passed by the detection model as classifying hands that were manually cropped with the help of mask files. The errors occurred when misclassifying the gesture "five" for "crawl" and "peace".

		Predicted labels					
		hand not detected	**crawl**	**fist**	**five**	**palm**	**peace**
Actual labels	**-**	0	0	0	0	0	0
	crawl	0	570	0	0	0	0
	fist	0	0	570	0	0	0
	five	0	3	0	565	0	2
	palm	0	0	0	0	570	0
	peace	1	0	0	0	0	569

Table 7: Confusion matrix of the end to end pipeline

By taking a closer look at the misclassified images, one notices that two errors do not occur anymore compared to the confusion matrix of the testing of the classification model on the manually cropped hands. This is because the detection model cropped the hands more precise than the manually written scripts with mask files. Still, three times a gesture "five" is misclassified as "crawl". To better understand the root of the error three images are shown side by side. The first image has been cropped with the detection model and classified, respectively misclassified with the classification model in the same manner as it happens in the gesture control pipeline. The second is shown for comparison reasons and been cropped manually with the mask file. The third image is shown for comparison reasons too and represents the original image before any cropping.

Looking at the first misclassified gesture “crawl” as depicted in Figure 15 it can be determined that the hand already occupies the whole image and therefore nothing can be cropped. Additionally, the gesture “five” is not completely visible on the original image and consequently to the small image for classification, confusing

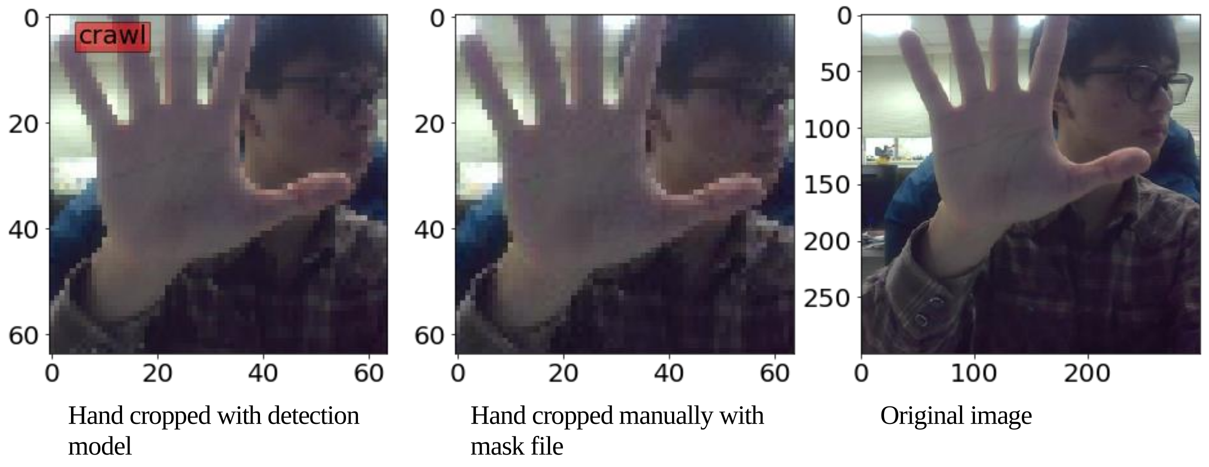

crawl
Hand cropped with detection model
Hand cropped manually with mask file
Original image

Figure 15: Gesture "five" misclassified as "crawl"

the classification model and leading to a misclassification. These results can also be interpreted in the context of the whole gesture control pipeline. It may happen that the JetBot detects a hand which is not completely visible on the input image but still resembles a known gesture leading to a misclassification.

In the other case, shown in Figure 16, the gesture “five” has been misclassified as “peace”. Interestingly, this image has been misclassified when using the manually cropped hands visible in the middle. Both hands are cropped almost identical. There is only a slight difference in the horizontal axis. In this case it is hard to tell the reason of the misclassification.

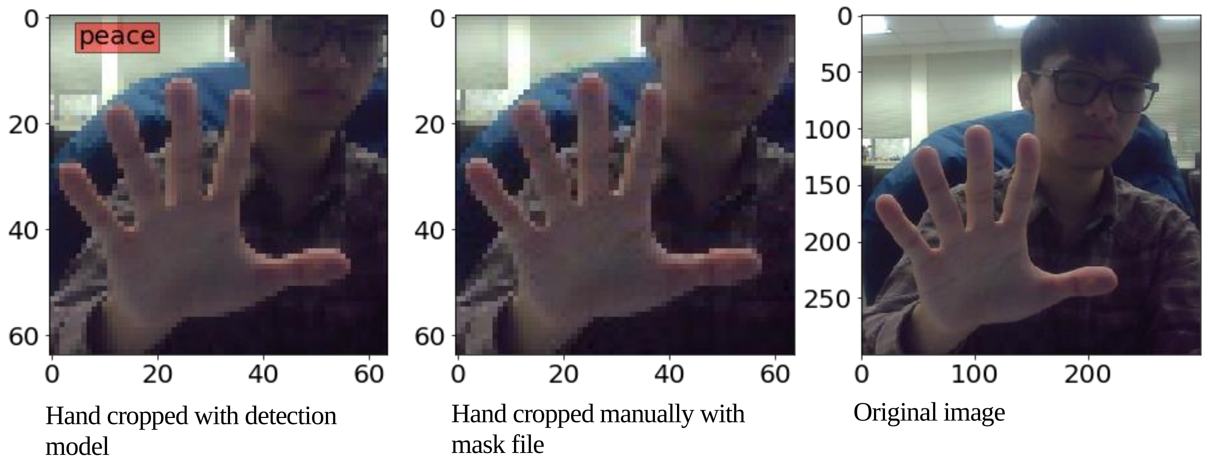

peace
Hand cropped with detection model
Hand cropped manually with mask file
Original image

Figure 16: Gesture "five" misclassified as "peace"

The five above described misclassifications plus the one additional error due to the undetected hand leads to an accuracy of 99.79 % for the whole detection pipeline tested on the LaRED dataset.

To be better be able to determine the performance of the gesture detection pipeline further tests on other datasets would have to be conducted. However, this requires the classification model to be trained on other data to be able to classify other gestures. Due to time constraints such experiments have not been conducted.

In summary the conjunct test of the hand detection and gesture classification model has shown very good results. Testing with the LaRED dataset the detection model has made only a single mistake and the classification accuracy has remained the same as when testing with the manually cropped hands.

5 Conclusion

In this thesis, a JetBot has been built from ground up and a demo case has been implemented. A representative task for the JetBot to solve by using its machine learning capabilities has been selected, namely gesture control. The selected demonstration case proved to be a good choice. It offers a good balance between being too difficult to implement while remaining challenging. It is a representative task that has current relevance for mobile and embedded devices since it is being used in a wide range of applications such as smart TVs and virtual reality.

Concerning the requirements defined in the project agreement we can state that all of them have been implemented. The hardware has been assembled, the JetBot is able to stream videos and has a strategy for moving around. The machine learning tasks are also covered so that the JetBot is capable of detecting multiple hands and classifying those into gestures in order to solve a real-world task. All relevant steps to assemble the JetBot, setup of the inference and even retrain and adjust the models are documented in the how-to guides available in the thesis repository in the `howto/` directory.

To fulfill the mentioned requirements many tasks have been completed when building the hardware and software of the JetBot. Surprisingly, there have been little problems with the hardware specifically when building the JetBot. Overall, working with the display, the motors and the camera was easy even though there have been some soldering issues and problems with the camera profile.

One difficulty had already appeared before beginning with the hardware steps, namely the definition of the task to be solved for the JetBot. This definition has been a requirement of the project. This led to some uncertainties at the beginning of the thesis. For example, it was not clear which parts of the machine learning theory had to be looked into and applied to the project. There are many great projects like “Deploying Deep Learning” by Dustin Franklin (Franklin, 2020) that could be leveraged to implement a task to solve for the JetBot but many of them require different theoretical background including knowledge of the frameworks to use or the level of the machine learning theory required. This led to several detours during the project which could have been avoided if the task had been defined earlier.

One of the hardest parts of the project has been the implementation of the detection model. The training of detection models is not supported by the Keras library out of the box and PyTorch also only offers the models Faster R-CNN and Mask R-CNN by default (Paszke, et al., 2019). Lesser known libraries had to be used which led to several issues due to incomplete or outdated documentation and several incompatibility issues. All these experiments and issues consumed a considerable amount of time that is not evident in this thesis.

Even though the project fulfills the requirements there are some aspects that could be improved. The current models can be optimized with TensoRT for even faster inference. Some attempts to optimize the models have been conducted but were discarded due to time constraints. The pipeline could be enhanced to allow the recognition of more gestures which would then allow the execution of more actions. The current actions are few and are implemented rather simply so that they are suitable to fulfill the requirements of this thesis. They are implemented in a way so they could be enhanced without too much further effort. The pipeline architecture is easily extensible since only the classification model has to be retrained to allow more gesture actions. Lastly, the evaluation of the detection pipeline is rather superficial and needs a deeper analysis. Further tests on other datasets must be conducted to be able to compare the performance to other models tested under specific conditions.

Although we faced many challenges these were very informative and rewarding. The selected and implemented demonstration case is successfully showcasing the capabilities of the JetBot. Overall, we are very pleased with the final product and the implementation of the demonstration case.

6 Bibliography

Abadi, M. et al., 2015. *TensorFlow: Large-scale machine learning on heterogeneous systems.* [Online]
Available at: https://www.tensorflow.org/
[Accessed 05 03 2020].

Bambach, S., Lee, S., Crandall, D. & Yu, C., 2015. Lending A Hand: Detecting Hands and Recognizing Activities in Complex Egocentric Interactions. *The IEEE International Conference on Computer Vision (ICCV),* 12, pp. 1949-1957.

Bao, P., Maqueda, A. I., del-Blanco , C. R. & García, N., 2017. Tiny hand gesture recognition without localization via a deep convolutional network. *IEEE Transactions on Consumer Electronics,* Volume 63, pp. 251-257.

Bianco, S., Cadene, R., Celona, L. & Napoletano, P., 2018. Benchmark Analysis of Representative Deep Neural Network Architectures. *IEEE Access,* pp. 270-277.

Bild, N. A., 2019. *GitHub, ShAIdes.* [Online]
Available at: https://github.com/nickbild/shaides/
[Accessed 29 02 2020].

Bonghi, R., 2020. *GitHub, Jetson stats.* [Online]
Available at: https://github.com/rbonghi/jetson_stats
[Accessed 11 02 2020].

Chollet, F., 2015. *Keras.* [Online]
Available at: https://keras.io
[Accessed 03 03 2020].

Chollet, F., 2018. *Deep Learning with Python.* Shelter Island: Manning Publications Co.

DiStefano, J. J., Stubberud, A. R. & Williams, I. J., 1990. *Feedback and control systems.* 2. ed. United States of America: McGraw-Hill.

Franklin, D., 2019. *Hello AI World - Meet Jetson Nano.* [Online]
Available at: https://info.nvidia.com/rs/156-OFN-742/images/Jetson_Nano_Webinar.pdf
[Accessed 03 03 2020].

Franklin, D., 2020. *GitHub, Jetson Inference.* [Online]
Available at: https://github.com/dusty-nv/jetson-inference
[Accessed 14 03 2020].

How to disable GUI on boot in 18.04 (Bionic Beaver)?, 2018. *Ask Ubuntu.* [Online]
Available at: https://askubuntu.com/questions/1056363/how-to-disable-gui-on-boot-in-18-04-bionic-beaver
[Accessed 11 02 2020].

Howard, A. G. et al., 2017. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications,* s.l.: arXiv, arXiv:1704.04861.

Huang, J. et al., 2017. *Speed/accuracy trade-offs for modern convolutional object detectors,* s.l.: arXiv, arXiv:1611.10012.

Huang, J. et al., 2018. *GitHub, TensorFlow Models, Configuring the Object Detection Training Pipeline.* [Online]
Available at:
https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/configuring_jobs.md
[Accessed 10 03 2020].

Huang, J. et al., 2019a. *GitHub, TensorFlow Models, Preparing Inputs.* [Online]
Available at:
https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/using_your_own_dataset

.md
[Accessed 10 03 2020].

Huang, J. et al., 2019. *GitHub, Tensorflow detection model zoo.* [Online]
Available at:
https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md
[Accessed 08 03 2020].

Idoko, J. B., Rahib, A. & Murat, A., 2019. Impact of Machine Learning Techniques on Hand Gesture Recognition. *Journal of Intelligent & Fuzzy Systems,* 9 9, Issue 37, pp. 41 - 52.

JetBot Issue 151, 2019. *GitHub, JetBot, Issue 151.* [Online]
Available at: https://github.com/NVIDIA-AI-IOT/jetbot/issues/151
[Accessed 05 01 2020].

JetBot Issue 16, 2019. *GitHub, JetBot, Issue 16.* [Online]
Available at: https://github.com/NVIDIA-AI-IOT/jetbot/issues/16
[Accessed 30 12 2019].

JetBot Issue 35, 2019. *GitHub, JetBot, Issue 35.* [Online]
Available at: https://github.com/NVIDIA-AI-IOT/jetbot/issues/35
[Accessed 11 02 2020].

JetBot Issue 68, 2019. *GitHub, JetBot, Issue 68.* [Online]
Available at: https://github.com/NVIDIA-AI-IOT/jetbot/issues/68
[Accessed 05 01 2020].

Lin, T.-Y.et al., 2015a. *COCO Detection Evaluation Metrics.* [Online]
Available at: http://cocodataset.org/#detection-eval
[Accessed 08 03 2020].

Lin, T.-Y.et al., 2015. *Microsoft COCO: Common Objects in Context,* s.l.: arXiv, arXiv:1405.0312.

Lin, T. T., 2015. *GitHub, LabelImg.* [Online]
Available at: https://github.com/tzutalin/labelImg
[Accessed 10 03 2020].

Liu, W. et al., 2016. SSD: Single Shot MultiBox Detector. *Lecture Notes in Computer Science,* p. 21–37.

Mohammed, A. A. Q., Lv, J. & Islam, S., 2019. A Deep Learning-Based End-to-End Composite System for Hand Detection and Gesture Recognition. *Sensors,* 30 11, Volume 19(23), pp. 1-23.

Mok, A., 2019. *GitHub, Rock-Paper-Scissors with Jetson Nano.* [Online]
Available at: https://github.com/mokpi/Rock-Paper-Scissors-with-Jetson-Nano
[Accessed 29 02 2020].

Muralt, D. & Wenk, C., 2019. *FHNW GitLab, 2019HS_JetBot.* [Online]
Available at: https://gitlab.fhnw.ch/ml/student-projects/2019hs_jetbot/
[Accessed 30 12 2019].

NVIDIA, 2019a. *JetPack.* [Online]
Available at: https://developer.nvidia.com/embedded/jetpack
[Accessed 10 02 2020].

NVIDIA, 2019b. *NVIDIA Jetpack 4.3 Release Notes.* [Online]
Available at: https://docs.nvidia.com/jetson/jetpack/release-notes/
[Accessed 10 02 2020].

NVIDIA, 2019c. *Get Your AI in Gear with a JetBot AI Robot Kit.* [Online]
Available at: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetbot-ai-robot-kit/
[Accessed 31 12 2019].

NVIDIA, 2019d. *NVIDIA TensorRT.* [Online]
Available at: https://developer.nvidia.com/tensorrt
[Accessed 10 02 2020].

NVIDIA, 2019e. *cuDNN Developer Guide.* [Online]
Available at: https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html
[Accessed 10 02 2020].

NVIDIA, 2019f. *CUDA Toolkit.* [Online]
Available at: https://developer.nvidia.com/cuda-toolkit
[Accessed 10 02 2020].

NVIDIA, 2019g. *CUDA C++ Programming Guide.* [Online]
Available at: https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html
[Accessed 10 02 2020].

NVIDIA, 2019h. *NVIDIA Jetson Linux Driver Package Developer Guide.* [Online]
Available at: https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/index.html
[Accessed 11 02 2020].

NVIDIA, 2019i. *NVIDIA Jetson Linux Driver Package Developer Guide.* [Online]
Available at: https://docs.nvidia.com/jetson/archives/l4t-archived/l4t-3231/index.html#page/Tegra%20Linux%20Driver%20Package%20Development%20Guide/overview.html
[Accessed 11 02 2020].

NVIDIA, 2019j. *Jetson Nano Developer Kit User Guide.* [Online]
Available at: https://developer.nvidia.com/embedded/dlc/jetson-nano-dev-kit-user-guide
[Accessed 03 03 2020].

NVIDIA, 2020a. *Accelerating Inference In TF-TRT User Guide.* [Online]
Available at: https://docs.nvidia.com/deeplearning/frameworks/tf-trt-user-guide/index.html
[Accessed 11 02 2020].

NVIDIA, 2020b. *TensorRT Developer Guide.* [Online]
Available at: https://docs.nvidia.com/deeplearning/sdk/tensorrt-developer-guide/index.html
[Accessed 10 02 2020].

NVIDIA, 2020c. *Installing TensorFlow For Jetson Platform.* [Online]
Available at: https://docs.nvidia.com/deeplearning/frameworks/install-tf-jetson-platform/index.html
[Accessed 11 02 2020].

NVIDIA, 2020d. *Jetson Community Projects.* [Online]
Available at: https://developer.nvidia.com/embedded/community/jetson-projects
[Accessed 29 02 2020].

NVIDIA, 2020e. *Jetson Nano Developer Kit.* [Online]
Available at: https://developer.nvidia.com/embedded/jetson-nano-developer-kit
[Accessed 03 03 2020].

Paszke, A. et al., 2019. *PyTorch: An Imperative Style, High-Performance Deep Learning Library.* [Online]
Available at: https://pytorch.org/docs/stable/torchvision/models.html#object-detection-instance-segmentation-and-person-keypoint-detection
[Accessed 14 03 2020].

Pedregosa, F. et al., 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research,* 12 October, pp. 2825-2830.

Project Jupyter, 2017. *Jupyter Widgets.* [Online]
Available at: https://ipywidgets.readthedocs.io/en/latest/examples/Widget%20List.html
[Accessed 25 02 2020].

Project Jupyter, 2018. *JupyterLab Overview.* [Online]
Available at: https://jupyterlab.readthedocs.io/en/stable/getting_started/overview.html
[Accessed 12 02 2020].

Sanchez-Riera, J. et al., 2015. *Large RGB-D Extensible Hand Gesture Dataset.* [Online]
Available at: http://mclab.citi.sinica.edu.tw/dataset/lared/lared.html
[Accessed 04 03 2020].

Sandler, M. et al., 2019. *MobileNetV2: Inverted Residuals and Linear Bottlenecks,* s.l.: arXiv, arXiv:1801.04381.

Tegra X1, 2020. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Tegra#Tegra_X1
[Accessed 03 03 2020].

TensorFlow, 2020. *GPU support.* [Online]
Available at: https://www.tensorflow.org/install/gpu
[Accessed 06 02 2020].

The GNOME Project, 2014. *GNOME Developer, nmcli.* [Online]
Available at: https://developer.gnome.org/NetworkManager/stable/nmcli.html
[Accessed 18 02 2020].

Tse, J., 2019. *Fix pink tint on Jetson Nano wide angle camera.* [Online]
Available at: https://medium.com/@jonathantse/fix-pink-tint-on-jetson-nano-wide-angle-camera-a8ce5fbd797f
[Accessed 11 02 2020].

Ubuntu, 2019. *Bionic Beaver Release Notes.* [Online]
Available at: https://wiki.ubuntu.com/BionicBeaver/ReleaseNotes
[Accessed 10 02 2020].

Ultimaker B.V., 2019. *Ultimaker Cura.* [Online]
Available at: https://ultimaker.com/software/ultimaker-cura
[Accessed 30 12 2019].

Villevald, D., 2019. *GitHub, Finding-path-in-maze-of-traffic-cones.* [Online]
Available at: https://github.com/dvillevald/Finding-path-in-maze-of-traffic-cones
[Accessed 03 02 2020].

Welsh, J., 2019, 50:25-51:20. *AI for Makers - Learn with JetBot.* [Online]
Available at: https://www.youtube.com/watch?v=zOCSRzDUI-Y
[Accessed 03 01 2020].

Welsh, J. & Yato, C., 2019a. *GitHub, JetBot.* [Online]
Available at: https://github.com/NVIDIA-AI-IOT/jetbot
[Accessed 29 12 2019].

Welsh, J. & Yato, C., 2019b. *GitHub, JetBot, 3D-printing.* [Online]
Available at: https://github.com/NVIDIA-AI-IOT/jetbot/wiki/3D-printing
[Accessed 30 12 2019].

Welsh, J. & Yato, C., 2019c. *GitHub, JetBot, Bill of Materials.* [Online]
Available at: https://github.com/NVIDIA-AI-IOT/jetbot/wiki/Bill-of-Materials
[Accessed 30 12 2019].

Welsh, J. & Yato, C., 2019d. *GitHub, JetBot, Hardware Setup.* [Online]
Available at: https://github.com/NVIDIA-AI-IOT/jetbot/wiki/Hardware-Setup
[Accessed 31 12 2019].

Welsh, J. & Yato, C., 2019e. *GitHub, JetBot, Software Setup.* [Online]
Available at: https://github.com/NVIDIA-AI-IOT/jetbot/wiki/Software-Setup
[Accessed 03 01 2020].

Welsh, J. & Yato, C., 2019f. *GitHub, JetBot, Examples.* [Online]
Available at: https://github.com/NVIDIA-AI-IOT/jetbot/wiki/Examples
[Accessed 12 02 2020].

Welsh, J. & Yato, C., 2019. *GitHub, JetCard.* [Online]
Available at: https://github.com/NVIDIA-AI-IOT/jetcard
[Accessed 18 02 2020].

7 List of tables and figures

7.1 List of tables

7.2 List of figures

A. Appendix

A1. Declaration of authorship

We, Dimitri Muralt and Christoph Wenk, hereby confirm that we authored this thesis without the help of a third party and only by using the aforementioned sources.

Last Name, First Name	Place, Date	Signature
Muralt, Dimitri	Bern, 16.03.2020	[illegible]
Wenk, Christoph	Dübendorf, 17.03.2020	[illegible]

A2. Bill of materials

This is a custom bill of materials derived from the original on GitHub (Welsh & Yato, 2019c). The list has been edited to use Swiss vendors whenever possible.

The quantity adheres to the number of pieces that are needed to build the JetBot and not how many need to be ordered. As an example, four M3 screws are needed for the bot which are sold in packages of 100 pieces.

Be aware that depending on where you order from customs duties might be added on top of your order.

Please see the following pages for details.

Vendor parts

Table 1. Common parts

Part	Quantity	Cost CHF	Vendor CH	Cost USD	Vendor Int	Notes
Jetson Nano	1	113.20	Distrelec	99.00	Arrow	Alternative CH vendor: Maker-Shop
Micro SD card	1	26.90	Digitec	11.99	Amazon	64GB
Power supply	1	25.00	Digitec	14.95	Adafruit	Micro USB, 5V, 2.5A; NVIDIA Power Supply Considerations
Motor	2	6.90	Play-zone	2.95	Adafruit	"TT" form factor
Motor driver	1	18.50	Distrelec	19.95	Adafruit	
Caster ball	1	~~4.50~~	~~ZigoBot~~	6.30	BC Precision	1-inch diameter; Caster Ball Supplier in Europe
Battery	1	39.20	Digitec	14.88	Aliexpress	2x 5V/3A output, 10,000mAh; PowerBank Size
USB cable pack	1-2	2 x 9.00	Digitec	6.99	Amazon	Type A to Micro, right angle
PiOLED display	1	19.90	Play-zone	14.95	Adafruit	
PiOLED header	1	1.50	Play-zone	5.95	Adafruit	2x(3+) right angle male

Table 2. Camera

Part	Quantity	Cost CHF	Vendor CH	Cost USD	Vendor Int	Notes
Camera	1	33.00	Digitec	23.90	Adafruit	Raspberry Pi Camera V2
Camera lens attachment	1	15.90	Pi Shop	12.45	AliExpress	160-degree FoV, 8 MP

Table 3. Wifi

Part	Quantity	Cost CHF	Vendor CH	Cost USD	Vendor Int	Notes
WiFi card	1	30.10	Digitec	24.89	Amazon	M2, Intel Wireless-AC 8265
WiFi antenna	2	10.30	Digitec	2.49	Arrow	U.FL connectors

Table 4. Wheels						
Part	**Quantity**	**Cost CHF**	**Vendor CH**	**Cost USD**	**Vendor Int**	**Notes**
Wheel	2	3.40	Play-zone	2.50	Adafruit	60mm diameter

Table 5. Assembly Hardware				
Part	**Quantity**	**Cost CHF**	**Vendor CHF**	**Notes**
Adhesive pads	2	5.20	Bauhaus	
M2 screw	20	5.35	Bauhaus	8mm long, self tapping
M3 screw	4	4.65	Bauhaus	25mm long
M3 nut	4	1.75	Bauhaus	
Jumper wires	4	11.30.-	Digitec	Female-female, ~20cm
Soldering Iron	1	30.20.-	Digitec	
Solder	1	17.20-	Digitec	

Shipping costs

Table 6. Shipping costs for each vendor		
Vendor	**Cost**	**Shipping time**
Amazon	USD 65.95	13-26 days with Amazon Standard Shipping
Adafruit	USD 36.42	2-4 days with DHL Express Worldwide delivery.
AliExpress	USD 2.17	15-22 days with AliExpress Standard Shipping
Arrow	USD 50.99	2-4 days with DHL Express Worldwide delivery.
BC Precision	USD 23.50	8-14 days with US Postal Services.

Play-Zone	CHF 9.00	B-Post
Distrelec	CHF 8.00	B-Post
Zigobot	CHF 9.00	B-Post
Pi-Shop	CHF 6.90	B-Post

Total costs

Table 7. Order variants and total costs

Vendor variant	**Total product costs (excluding assembly hardware)**	**Total shipping costs**	**Total overall costs**	**Description**
All Swiss vendors	CHF 378.1	CHF 32.90	CHF 411.00	Swiss vendors selected if possible. Not all parts same as in example project which might cause problems. Easiest to order and cheapest. Fastest shipping. Small ecological footprint.
Example project vendors	USD 278.62	USD 153.36	USD 431.98	Same vendors selected as in the original project if possible. Least problems to be expected. Order very difficult and expensive. Longest shipping. Big ecological footprint.
Adafruit preferred	CHF 352.00	CHF 92.82	CHF 444.82	Swiss vendors except for products made by Adafruit and BC Precision. Few problems to be expected. Order manageable but around 33.- more expensive than cheapest variant. Ecological footprint justifiable.

3D printed parts

Table 8. 3D printed parts

Part	Quantity	Cost	Vendor	Notes
Chassis	1	n/a	FHNW Maker Studio	STL file for chassis
Camera Mount	1	n/a	FHNW Maker Studio	STL file for Camera mount
Caster base	1	n/a	FHNW Maker Studio	STL file for 60mm wheel
Caster shroud	1	n/a	FHNW Maker Studio	STL file for 60mm wheel

Additional information

The Raspberry Pi camera has been picked because the hardware guide uses this as well. The assumption is that this will lead to fewer problems when installing the part.

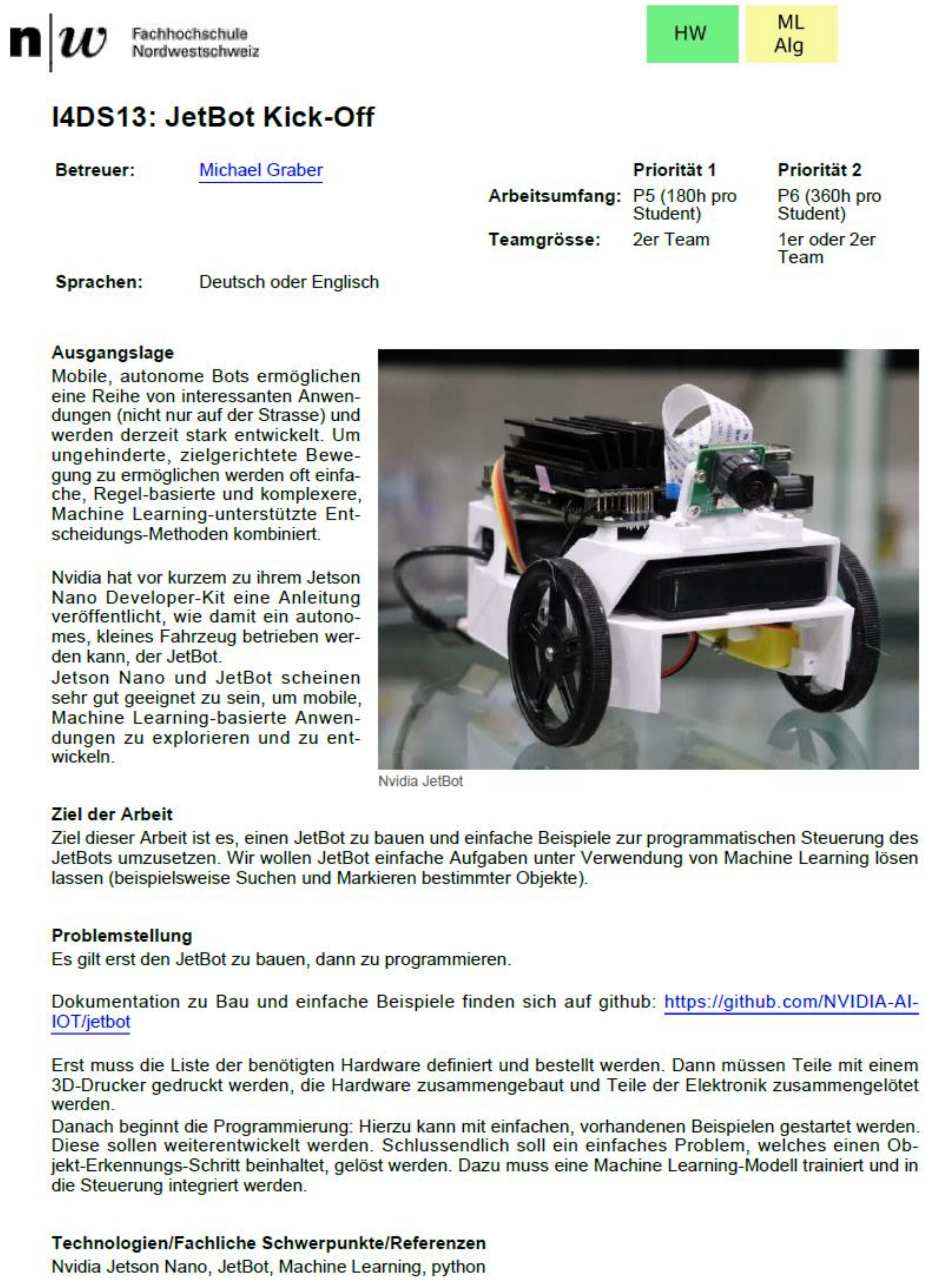
The M2 Wifi module has been picked because the hardware guide uses this as well. The assumption is that this will lead to fewer problems when installing the part.

The 60mm wheels have been picked because JetBot was originally designed to use these. They are also better available from Swiss shops. However, it could be that the 65mm wheels perform a little bit better on uneven ground and obstacles. In case of any issues 65mm wheels can be ordered from Adafruit.

Zigobot is the only Swiss vendor with a fitting caster ball. However, it cannot be recommended as the usability of the website is considerably below standards and the checkout process of the website is dubious.

Originally a soldering iron sold by Bauhaus had been picked. This proved to be very unhandy and broke after a short while. There is another cheap model included in the bill of materials. However, a higher end soldering station is highly recommended for the best results.

A3. Problem Statement



Fachhochschule
Nordwestschweiz

HW	ML Alg

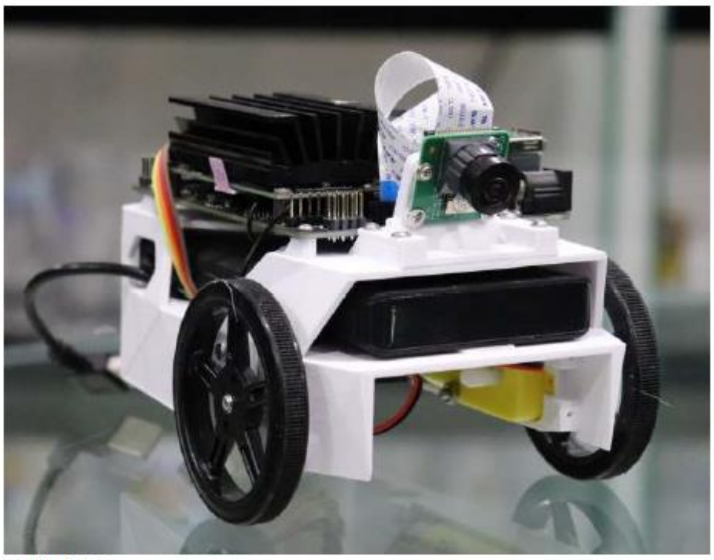
I4DS13: JetBot Kick-Off

			Priorität 1	Priorität 2
Betreuer:	Michael Graber			
		Arbeitsumfang:	P5 (180h pro Student)	P6 (360h pro Student)
		Teamgrösse:	2er Team	1er oder 2er Team
Sprachen:	Deutsch oder Englisch			

Ausgangslage

Mobile, autonome Bots ermöglichen eine Reihe von interessanten Anwendungen (nicht nur auf der Strasse) und werden derzeit stark entwickelt. Um ungehinderte, zielgerichtete Bewegung zu ermöglichen werden oft einfache, Regel-basierte und komplexere, Machine Learning-unterstützte Entscheidungs-Methoden kombiniert.

Nvidia hat vor kurzem zu ihrem Jetson Nano Developer-Kit eine Anleitung veröffentlicht, wie damit ein autonomes, kleines Fahrzeug betrieben werden kann, der JetBot.
Jetson Nano und JetBot scheinen sehr gut geeignet zu sein, um mobile, Machine Learning-basierte Anwendungen zu explorieren und zu entwickeln.


Nvidia JetBot

Ziel der Arbeit

Ziel dieser Arbeit ist es, einen JetBot zu bauen und einfache Beispiele zur programmatischen Steuerung des JetBots umzusetzen. Wir wollen JetBot einfache Aufgaben unter Verwendung von Machine Learning lösen lassen (beispielsweise Suchen und Markieren bestimmter Objekte).

Problemstellung

Es gilt erst den JetBot zu bauen, dann zu programmieren.

Dokumentation zu Bau und einfache Beispiele finden sich auf github: https://github.com/NVIDIA-AI-IOT/jetbot

Erst muss die Liste der benötigten Hardware definiert und bestellt werden. Dann müssen Teile mit einem 3D-Drucker gedruckt werden, die Hardware zusammengebaut und Teile der Elektronik zusammengelötet werden.
Danach beginnt die Programmierung: Hierzu kann mit einfachen, vorhandenen Beispielen gestartet werden. Diese sollen weiterentwickelt werden. Schlussendlich soll ein einfaches Problem, welches einen Objekt-Erkennungs-Schritt beinhaltet, gelöst werden. Dazu muss eine Machine Learning-Modell trainiert und in die Steuerung integriert werden.

Technologien/Fachliche Schwerpunkte/Referenzen

Nvidia Jetson Nano, JetBot, Machine Learning, python

A4. Project agreement

Authors: Dimitri Muralt, Christoph Wenk
Principal and supervisor: Michael Graber
Co-supervisor: Fabian Märki

Introduction

This document acts as agreement between the principal Michael Graber, who also takes the responsibility as advisor, the Co-supervisor Fabian Märki and the project team, consisting of Dimitri Muralt and Christoph Wenk. The document contains contact data as well as general information about the project schedule and scope. It also determines what has to be completed by the end of the project.

This agreement is based on the project description *I4DS13_Project_description_Jetbot_Nano.pdf* as well as the meetings on 16. September 2019 and 7. October 2019.

Base line

Mobile, autonomous bots enable a number of interesting use cases and are under heavy development. Often, simple rule-based and more complex machine learning supported decision-methods get combined to enable unobstructed, target-oriented movements.

Nvidia has recently published instructions how to build a small, autonomous vehicle leveraging their Jetson Nano developer kit.

Jetson Nano and JetBot seem to be a very adequate way to explore and develop mobile, machine learning based applications.

Goals

The goal of this project is to build a JetBot and implement simple examples for the programmatic control of the JetBot. The JetBot should be able to solve simple tasks by leveraging its machine learning capabilities including object recognition.

Problem statement

Since the instructions how to build the JetBot and use the Jetson Nano developer kit have been published just recently it isn’t clear how well suitable the platform is for usage in education and what the capabilities of the JetBot are.

With this project the platform shall be evaluated showcasing the capabilities of the JetBot. For this purpose the JetBot has to be built first according to Nvidia’s documentation on GitHub and afterwards programmed. A task, which contains an object-recognition-step using machine learning, shall be solved. As an example the JetBot could detect Lego bricks or hand-written numbers in a room and execute an appropriate action like move them to a certain location or update an inventory.

Deliverables

This chapter describes the artifacts to be created and delivered during the project.

Table 1. Artifacts

Artifact	Content (not necessarily chapters)
Project agreement	• Base line • Goals • Problem statement • Functional- / Non-functional requirements • Product application • Project planning • Approval & Permission
Thesis	• Key aspects • Approach • Analysis • Architecture • Technologies & Software • Conclusion
How-to guides	• Step-by-step descriptions • Material list
Hardware	• Assembled functional JetBot
Software	• Produced code for JetBot

Requirements

Functional requirements

Table 2. Functional requirements

#	Title	Description	Priority
F-1	Motion	JetBot is assembled and capable to move on even ground	high
F-2	Video Stream	JetBot streams video data live with built-in camera.	high
F-3	Motion control	JetBot has a strategy to move around on the ground suitable to the problem to be solved. This includes self-driving strategies and possibly manual control via keyboard.	high
F-4	Basic classification	JetBot is capable to solve a classification problem with the help of a trained machine learning model e.g. classify hand written digits.	high
F-5	Single object classification	JetBot is capable to recognize an object (e.g. lego brick) and classify it (e.g. size) from the video feed with the help of a trained machine learning model. It's enough to recognize only one object at a time and no image segmentation is needed.	high
F-6	Multiple object detection	JetBot is capable to detect one or multiple objects on an image with the help of image segmentation (e.g. three lego bricks) and classify	medium

		it (e.g. size) from the video feed with the help of a trained machine learning model.	
F-7	Real world task	JetBot solves a simple real word task with the help of a trained machine learning model including an object recognition step.	medium
F-8	How to guide hardware	A how to guide is available with which the JetBot can be assembled completely.	high
F-9	How to guide software	A how to guide is available with which central JetBot functionalities can be recreated.	high

Non-functional requirements

Table 3. Non-functional requirements

#	Description
NF-1	The documentation shall be written in English
NF-2	The JetBot has to be assembled by the project team. Including the 3D printing.
NF-3	Python based machine learning libraries shall be used for the project.
NF-4	Nvidia Jetson Nano shall be used as underlying platform.

Product application

Range of applications

The product shall act as a prototype to show what is possible with the Jetson Nano platform. It will display the capabilities of the JetBot with a simple machine learning example task.

The finished JetBot can help lecturers to decide if the platform is suitable for their classes.

Target audience

The target audience are lecturers, particularly at FHNW.

Project planning

The project planning consists of milestones and issues on GitLab. The milestones contain delivery dates and are linked to issues. An agile approach is preferred and project planning is being kept to a minimum.

A rough project plan is also delivered in this agreement to give an overview of the project milestones.

The project lasts 24 working weeks and includes a total working time of 360 hours per team member. Each team member plans to work 15 hours per week in average which leads to the total of 360 hours (24x15).

Table 4. Project overview

Week #, Date Tuesday	Milestone
38, 17.09.2019	
39, 24.09.2019	
40, 01.10.2019	
41, 08.10.2019	
42, 15.10.2019	Parts ordered

43, 22.10.2019	
44, 29.10.2019	Parts delivered
45, 05.11.2019	Project agreement finalized
46, 12.11.2019	**Milestone 1: Hardware** JetBot assembled
Project week	
48, 26.11.2019	F-1 Motion F-2 Video Stream
49, 03.12.2019	
50, 10.12.2019	
51, 16.12.2019	**Milestone 2: First Steps** JetBot can solve first simple problems with object recognition. Already available examples can be used and altered if needed for this milestone.
christmas	
christmas	
2, 07.01.2020	
3, 14.01.2020	F-4 Basic classification
4, 21.01.2020	
5, 28.01.2020	**Milestone 3: Single object classification** F-5 Single object classification
6, 04.02.2019	
7, 11.02.2020	
8, 18.02.2020	
9, 25.02.2020	
10, 03.03.2020	**Milestone 4: Multiple object classification** F-6 Multiple object classification
11, 10.03.2020	
12, 17.03.2020	**Milestone 5: Project finalized** How to Guides finalized Thesis finalized

Meetings

A weekly slot for face-to-face meetings is being reserved on Monday from 14.00 - 15.00. If there is no need for a meeting, it can be cancelled by both sides with a short notification in advance. Additional questions can always be asked on GitLab by creating an issue and the @User function. Meetings can also be held on Skype if needed.

Approval & Permission

This project agreement sets out the objectives and conditions for the IP-6 project *I4DS13: Jetbot Kick-Off*. This document serves as a contractual basis between the aforementioned parties.

This agreement will be accepted by all parties with the merge request to the master branch of the GitLab repository https://gitlab.fhnw.ch/ml/student-projects/2019hs_jetbot with all parties involved approving the merge.

Changes to this document are possible if they are accepted by all parties. Those changes have to be approved again via a merge request with all parties as reviewer on GitLab.

A5. Source Code

The source code for this thesis can be found on the FHNW GitLab instance:

https://gitlab.fhnw.ch/ml/student-projects/2019hs_jetbot