Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

Ole Martin Brokstad

The Cyborg v4.0 - Computer Vision Module

Towards a Socially Intelligent Robot

Master's thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
June 2020

NTNU
Kunnskap for en bedre verden

NTNU – Trondheim
Norwegian University of
Science and Technology

The Cyborg v4.0 - Computer Vision Module

Towards a Socially Intelligent Robot

Ole Martin Brokstad

Master's Thesis in Cybernetics and Robotics
Supervisor: Sverre Hendseth
Co-supervisor: Martinius Knudsen

NTNU
Dept. of Engineering Cybernetics
June 2020

Task Description

The goal of the master project is to implement a computer vision module on the Cyborg robot using the ZED stereoscopic camera and the Jetson TX1 developer kit. The computer vision module should output relevant data about the surroundings, based on the ZED camera recording in real-time. The module should be integrated with the rest of the Cyborg system. The computer vision module should be designed considering the output should contribute to the functionality of the other modules on the Cyborg, such as the navigation and the behavioral module.

The project involves the following tasks:

1. Literature review of relevant work, especially previous work with the Cyborg robot.
2. Get familiarized with Linux and ROS through tutorials.
3. Hardware and software setup of the ZED stereoscopic camera and the Jetson TX1 developer kit.
4. Reimplementation of zedyolo, the computer vision system done previously within the Cyborg project.
5. Discuss and determine requirements for the final delivered system.
6. Design and implement prototypes of the module.
7. Integrate the module with the ROS network on the Cyborg.
8. Test the computer vision module and create tables and visualizations of the results.
9. Discuss the results.
10. Conclude and suggest further work.
11. Gather all written and visualized results from the sub-tasks and write the final report.

Abstract

This thesis presents the computer vision (CV) module designed for the Cyborg robot. Motivated by the advantages of improving interactions between people and robots, this thesis aims to implement a system capable of detecting natural human behavior, allowing the Cyborg to become a socially intelligent robot. The CV module is implemented on the Jetson TX1 Developer board, retrieving images from the first generation ZED Stereoscopic camera.
The thesis presents a discussion on how to create a social robot, featuring elements from psychology. The discussion presented, suggests observing individual human behaviour and facial expressions serve as a foundation for making the Cyborg a socially intelligent robot.
The CV module tracks individual people using YOLO object detection, in combination with SORT multiple object tracking. The module further estimates the tracked people's horizontal relative coordinates. The CV module detects human facial expressions using OpenCV Haar Cascade face and smile classifiers.
In addition, the module counts the number of people located in the surroundings.
The CV module manages sufficiently to detect the mentioned information, with a range of 2 meters to the person.
The CV module is integrated as a package in ROS, and a procedure for connecting the ROS network on the Jetson TX1 board to a ROS Master on an external machine is presented. This allows the CV module on the Jetson TX1 board to be integrated with the rest of the Cyborg ROS system, located on the Cyborg base computer.
The CV module is tested to make the detected information available for a subscribing ROS Node on an external machine within 0.5 seconds, allowing the Cyborg to react in real-time. The total CV module speed is tested to manage an output frequency of about 3Hz, depending on the captured environment in the images.

Sammendrag

Denne avhandlingen presenterer Datasynmodulen, utformet for Cyborg roboten. Motivert av fordelen av å forbedre interaksjoner mellom mennesker og roboter, sikter dette prosjektet på å implementere et system som klarer å detektere vanlig menneskelig adferd, som kan bane vei for at Cyborgen kan bli en sosialt intelligent robot. Datasynmodulen er implementert på et Jetson TX1 utvikler brett, som henter bilder fra et første generasjons ZED stereoskopisk kamera.
Avhandlingen presenterer en diskusjon som omhandler hvordan å lage en sosial robot, med elementer fra psykologifaget. Diskusjonen foreslår at å observere individuell menneskelig adferd og ansiktsuttrykk kan danne et grunnlag for at Cyborgen kan bli en sosialt intelligent robot.
Datasynmodulen sporer individuelle mennesker ved hjelp av YOLO *object detection*, kombinert med SORT *multiple object tracking*. Videre estimerer modulen de relative horisontale koordinatene til de sporede menneskene. Datasynmodulen detekterer menneskets ansiktsuttrykk ved hjelp av OpenCV *Haar Cascade* smil og ansikt klassifikatorer.
I tillegg teller modulen antall mennesker som befinner seg i området.
Datasynmodulen klarer å detektere den nevnte informasjonen tilstrekkelig, med 2 meters rekkevidde til mennesket.
Datasynmodulen er integrert som en pakke i ROS, og det presenteres en prosedyre for å koble ROS nettverket på Jetson TX1 brettet til en ROS Master på en ekstern maskin. Dette muliggjør at Datasynmodulen på Jetson TX1 brettet kan bli integrert med resten av Cyborg ROS systemet, som befinner seg på Cyborg datamaskinen.
Det er testet at Datasynmodulens deteksjoner gjøres tilgjengelig for en abonnerende ROS Node på en ekstern maskin innen 0.5 sekunder, som tillater at Cyborgen kan reagere i sanntid.
Hastigheten til det totale systemet er testet til å klare en utgangfrekvens på rundt 3Hz, avhengig av innholdet i bildene.

Preface

This Master's thesis has been conducted at the Department of Engineering Cybernetics at the Norwegian University of Science and Technology. The thesis concludes the requirements for the Master of Science degree.

I would like to thank my supervisor Sverre Hendseth for guiding me throughout the report writing and the project process, and for emphasizing the importance of having a clear vision of the final goal. I would like to thank my co-supervisor Martinius Knudsen for arranging the team meetings, as the coordinator of the Cyborg project, and for giving me the freedom of being creative with the problem approach. I would like to thank the rest of the Cyborg team members, as of spring 2020; Lasse Göncz, Johanne Kalland and Casper Nilsen, for welcoming me, as I joined the project later in January.

Finally, I would like to thank my family and friends for the support throughout the year.

Table of Contents

Abbreviations

BB	=	Bounding Box
BBs	=	Bounding Boxes
CUDA	=	Compute Unified Device Architecture
CV	=	Computer Vision
EiT	=	Experts in Team
FPS	=	Frames Per Second
GPU	=	Graphics Processing Unit
GUI	=	Graphical User Interface
ID	=	IDentification
IOU	=	Intersection-Over-Union
L4T	=	Linux 4 Tegra
ROS	=	Robot Operating System
R-CNN	=	Region Convolutional Neural Network
SDK	=	Software Development Kit
YOLO	=	You Only Look Once

Chapter 1

Introduction

The presented work in this Master's Thesis is carried out as a part of the ongoing Cyborg project at NTNU. The Cyborg project has the goal of creating a robot that autonomously navigates the NTNU campus while interacting with its surroundings.
The Cyborg robot has been in development since 2015 with several EiT groups and Master's students working on the project. This year, a group of four students write their Master's Thesis with the Cyborg project. Each one is responsible for different parts of the robot. The author's objective is to implement a computer vision module (CV module) integrated with the Cyborg robot system, using a ZED stereoscopic camera and the Jetson TX1 Development board. The author further specifies the objective; to design and implement a CV module detecting human behaviour, enabling the Cyborg to become a socially intelligent robot.

1.1 Motivation

As the proportion of the world's older population is drastically increasing[1], the demand for workers in the health sector is expected to increase[2], driving up the demand for labour in the industry in general. A natural solution to the high demand is to replace some of the workers by using automated systems and robots. The problem is that many of the tasks are too complex for a robot to complete sufficiently. A solution to this is to make a person do the complex tasks, while the robot assists with the simpler time-consuming tasks[3]. In other words; make robots collaborate with humans. Considering this, the author expects higher demand for systems allowing human interaction with robots and other automated systems, in the future.
A known problem within human-robot interaction research is the problem of *intent recognition*[4]. We humans can normally effortlessly understand other people's intentions by observing facial expressions, body language, and other signals, by instinct and through years of social experience. However, designing and programming a robot to detect the complex human behaviour sufficiently to recognize intent, is a difficult problem. If one could create a robot with a good understanding of human behaviour, this would allow more

effective interactions between robots and humans. A way of recognizing human intent for a robot is to use CV technology to detect the visual behaviour of a person. This can involve detecting the person's movement, facial expression and body language, based on recorded images.
An important goal of the Cyborg project is to make the robot capable of social interactions with the surrounding people at the NTNU campus. To realize this, the robot needs to detect the intent of the surrounding people. Maybe the person is trying to communicate with the robot, or the person is not interested at all. A natural way of detecting intent is to detect the person's behaviour using CV as discussed previously in this section.
Many existing robots, designed to interact with humans, require people to learn how to communicate with the robot. This could prevent random people, like for example students at the NTNU campus, from communicating with the robot. This is why the system implemented in this project is designed to only detect natural human behaviour, requiring no prior knowledge by the surrounding people.

1.2 Problem Overview

The objective is to implement new and existing CV technologies in a module on the Cyborg robot, contributing with information about the surroundings. The CV module should be integrated with the Cyborg ROS Network, and be implemented on the Jetson TX1 developer board, using the first generation ZED camera.
Further than this, the author has been given the freedom to decide the objective of the project and the requirements of the system. The motivation leads to how the objective of the Master's project is specified. With reference to section 1.1, many projects, including the Cyborg project, aims to create a social robot, where recognizing human intent is a common challenge. Motivated by this, the author further specifies the goal; to design and implement a CV module detecting natural human behaviour, enabling the Cyborg to become a socially intelligent robot. A part of the problem is therefore to research, discuss and determine which information to predict from the camera images, which is relevant for recognizing human intent.

1.3 Report Structure

First, the background material, including related work, introduction of software and hardware and theory in general, is presented in chapter 2. Included in the background material is a presentation of the reimplementation of zedyolo, which is the previously implemented CV system within the Cyborg project. This is included since it is used as a starting point for the development of the final system presented in this report. The lessons learned from the reimplementation of zedyolo are referenced in some of the design choices made, presented in chapter 4.
Further, the chapters in the report will follow the regular structure: Requirements, design, implementation and testing.
Finally, the testing and the general project results are discussed, and concluded.
The final system presented is referred to as the ”CV module/system” and ”behaviourdetec-

tion system" throughout the report. "behaviourdetection" is the name of the final system, designated by the author.

Chapter 2

Background

This chapter will introduce the reader to the related work, general theory, software and hardware which the project is built upon. The aim is to give the reader an understanding of the context of the project.

2.1 Related Work

This section presents the previous and ongoing work within the Cyborg project, in addition to other work relevant to this Master's Thesis.

2.1.1 The NTNU Cyborg Project Spring 2020

A goal of the NTNU Cyborg project, not yet mentioned, is to create a robot that is integrated with biological neural tissue. This inspires the name "Cyborg" as this is used for describing a robot that is part human and part machine.
The other goal, which is more directly connected to this thesis, is to create a robot which can freely and autonomously wander the NTNU campus, while interacting with the surroundings.
The coordinator of this project is PhD student Martinius Knudsen. The team working on the Cyborg for their Master's Thesis the spring of 2020 consists of 4 students:

- Lasse Göncz is responsible for implementing the navigation module on the Cyborg. This involves reimplementing the navigation system and optimizing the localization performance.
- Johanne Kalland is responsible for the behavioral module on the Cyborg, which involves implementing new features using behaviour trees.
- Casper Nilsen is responsible for creating the GUI module for remote control and monitoring of the Cyborg. A part of his goal is to allow the robot to be maneuvered remotely and in real-time with a click-to-send interactive map.

- Ole Martin Brokstad, the author of this report, is responsible for implementing a computer vision module on the Cyborg.

The future vision for the Cyborg is to become a sort of mascot for NTNU. This involves the Cyborg getting attention from the community. As a result, the team focuses on creating interesting, funny, smart, and creative features for the robot. Also, since several new students will continue working with the project in the future, the created modules should work independently, and be sufficiently documented.

2.1.2 The NTNU Cyborg v2.0: The Presentable Cyborg

The Master's Thesis, written by Jørgen Waløen in 2017[5], aims to make the Cyborg robot ready for presentation. Throughout the report several diagrams are presented, giving a good overview of the planned and existing hardware components architecture, and software modules communication.
The second part of the thesis is a set of guides, attached in the appendix. This includes, guides on how to set up the software for the Jetson TX1 and the ZED camera. However, some of the material is outdated, which should be expected, since the report is 3 years old.

2.1.3 The Cyborg v3.0: Foundation for an NTNU Mascot

The Master's Thesis, written by Areg Babayan[6], presents the work carried out in the spring of 2019, which is the latest Master's Thesis within the Cyborg project. His goal was to further work for an autonomous Cyborg, which involved preparing the robot for a demonstration. The report focuses on gathering the previous work into an overall description of the system, which makes the report a good general introduction for new students.

2.1.4 EiT - Robotvision: zedyolo

In the spring of 2018 an EiT group did a project with the Cyborg[7]. Their goal was to develop a system with the ability to perform object detection and localization on a Jetson TX2 card and a ZED stereoscopic camera. The system was mostly developed using Python-based on frameworks and package solutions such as ROS Lunar, YOLOv2, pyyolo, ZED-ROS-wrapper and ZED SDK. The resulting system managed to detect objects, calculate distance, and publish this on the ROS network.
Since this is an EiT report and not a Master's Thesis, the description of the system is more practical, short, and straight to the point. For detailed implementation the report mostly referrers to existing tutorials for guiding. Since the report is short and straight to the point, it is a good source for getting introduced to the subject.

2.1.5 Relevance to this project

Aregs report, described in section 2.1.3, is used for getting familiar with the Cyborg project, the vocabulary, and the Cyborg ROS system in general.
Waløens report, described in subsection 2.1.2, is used as a source for learning about the history of the Cyborg project, the reasoning behind existing solutions. In addition, it is

used for inspiration when discussing the vision for the Cyborg.
The EiT group's work, described in section 2.1.4, is very relevant, as their goal and system specifications are similar to what's described in the given task, and by the author of this Master's Thesis. The report is used as a starting point for implementing and testing new solutions for the CV module on the Cyborg.

2.2 Software and Hardware Introduction

In this section, the hardware equipment, the software packages and algorithms used throughout the project, are introduced. This includes the theory behind some of the software packages and the CV algorithms developed by the author and other referenced researchers.

2.2.1 Jetson TX1 Development Kit

The Jetson TX1 Development Kit[8] is a development platform for visual computing, which comes pre-flashed with JetPack[9], including the Linux desktop environment "Linux for Tegra", or L4T in short. The Linux environment is derived from Ubuntu 18.04[10].
The development board includes ports, which in combination with a USB-hub, enable the connection of an external monitor, a keyboard and a mouse. This makes it simple and effective to develop and test code just like on a regular computer.

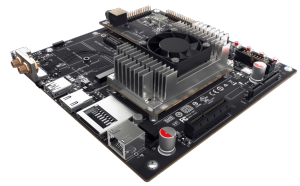


Figure 2.1: Jetson TX1 Development Kit.

The key component on the Jetson kit is the NVIDIA Maxwell GPU. This GPU, among some of the other Nvidia GPUs, is compatible with the parallel computing platform CUDA[11]. The CUDA platform enables accelerated computing using the GPU, and is a requirement for installing the driver software for the ZED Stereo Camera.
As a result, The Jetson development board is widely used for visual computing applications, which requires low power consumption, and small size.
The Jetson TX1 board has some limitations experienced by the author. The board disk space is only 16GB, however, this can be extended by inserting an SD card.

2.2.2 JetPack

JetPack[9] is a software developer kit designed for the Nvidia Jetson developer boards, which installs a Ubuntu derived OS called "L4T", and several useful developer tools, needed to jump-start a development environment. Two relevant developer tools included in JetPack is CUDA and OpenCV.

2.2.3 ZED Stereo Camera

The first generation ZED Stereo Camera[12] is a high-quality 3D sensing camera created by Stereolabs. It can operate in a low-light challenging environment, keeping high frame-rate and crisp images.
The camera can be connected via the integrated USB 3.0 cable for powering and data transmission.
Together with the driver software, ZED SDK, the camera can deliver a depth map, point cloud and video stream in real-time, just to mention a few of the features. The camera is well supported with several possible third-party integrations, maintained by either Stereolabs, or the user-base. Among the integrated third-party software is ROS, Python and OpenCV.

Figure 2.2: ZED Stereo Camera.

2.2.4 Ubuntu

Ubuntu is an open-source Linux operating system[13]. It is widely used for developing applications since it is open-source and highly customizable. Some of the applications in this project require Ubuntu. The Jetson TX1 should run the L4T Ubuntu software. If this is not already installed on the Jetson board, it needs to be flashed with the JetPack[9], which includes the L4T Ubuntu OS, using a host computer running Ubuntu[14].

2.2.5 ROS - The Robot Operating System

ROS is a flexible framework for writing robotics software[15]. The framework simplifies a robust interface between different robotic platforms.

Figure 2.3: ROS Logo.

ROS is used in the Cyborg project to enable communication between the modules. Each Cyborg robotic application is created as a Node in a ROS Package, which gives the modules access to the ROS framework tools.
The communication is made simple using the tool ”ROS Topics”. This tool enables the modules to publish data as ROS Messages to the Topics. ROS Message is a ROS data type that can either be created or imported from the ROS standard Messages. The published Messages are available on Topics for every module running connected to the same ROS Master. The ROS Master is a name service, which helps connected Nodes find each other and the published Topics.

The ROS framework provides client libraries which allow Nodes written in different languages to communicate. For example, ”rospy” is a ROS client library, which when imported into a Python script, can provide the functions for publishing Messages to ROS Topics.
In addition to the ROS tools for communication, the framework provides several commands which can be used in the Terminal window for running nodes and for debugging. Commands like ”rostopic” and ”rosnode” can be used in the Terminal window while the

Nodes are running for monitoring and manually controlling the ROS system.

The wide range of openly available libraries and tools, in combination with sufficient documentation, makes ROS a great framework for collaboration on a robotic development project.

2.2.6 CUDA

CUDA, which stands for Compute Unified Device Architecture, is a platform enabling GPU-accelerated computation, developed by Nvidia[16]. The CUDA technology employs the GPU, which can effectively manipulate large blocks of data, such as images. As a result, the CUDA platform is widely used in computer vision related applications. The platform is only compatible with some Nvidia GPUs, including the GPU on the Jetson TX1[11].

2.2.7 ZED SDK

ZED SDK, short for ZED Software Development Kit, is the architecture around the Camera class, which is used for interaction with the ZED camera. This involves configuration and grabbing output data from the camera. The ZED camera configuring includes setting the resolution, the frame rate, the brightness, etc. The camera provides output data like image stream, depth map and point cloud, which are the most relevant for this project[17]. Less relevant outputs are position tracking and object detection. The ZED SDK object detection module is only compatible with the ZED 2 camera, and position tracking is already taken care of by the navigation module on the Cyborg.

The ZED SDK requires the computer to have at least 4GB of RAM and to run a Nvidia GPU with a computing capability of more than 3[12]. The Jetson TX1 has 4GB of RAM and a Nvidia GPU with a computing capability of 5.3, which should be sufficient. The Nvidia GPU requirement is due to the CUDA dependency. If CUDA is not installed, the camera can still be used for retrieving images. However, more advanced output data, like depth map and point cloud, is not available.

2.2.8 Python

Python is a high-level, object-oriented programming language, with a large userbase. It enables importing and implementing external package- and project-functions, which contributes to efficient development of new software applications. The large userbase produces a wide selection of Python libraries and interfaces which are openly available and free of use.

2.2.9 ZED Python API

The ZED Python API is a package letting you use the ZED Stereo Camera with Python [18]. After installing, the package ”pyzed.sl” is available for import in a Python script.

This package includes all the functions in the ZED SDK for configuration of, opening and retrieving output data from the ZED camera.

2.2.10 YOLO

YOLO is a state-of-the-art, real-time object detection model[19], implemented in the Darknet framework[20]. Darknet is an open-source neural network framework written in C and CUDA, which supports GPU computation.

Many other object detection systems, like R-CNN, apply a classification model on each image at multiple places and scales. This is slow since this sometimes requires thousands of model evaluations for a single image. The YOLO model approach is to feed the whole image through a single convolutional network once, hence the name You Only Look Once. The original YOLO network design consists of 24 convolutional layers followed by 2 fully connected layers. The network divides the image into a 7x7 grid and predicts bounding boxes and class probabilities for each grid cell, simultaneously, as shown in Figure 2.4. As a result, YOLO can only detect a maximum of 49 objects in one image. Since an object is often located in more than one grid cell, several predicted BBs may overlap. The best predicted BB is kept by using non-maximal suppression[21].

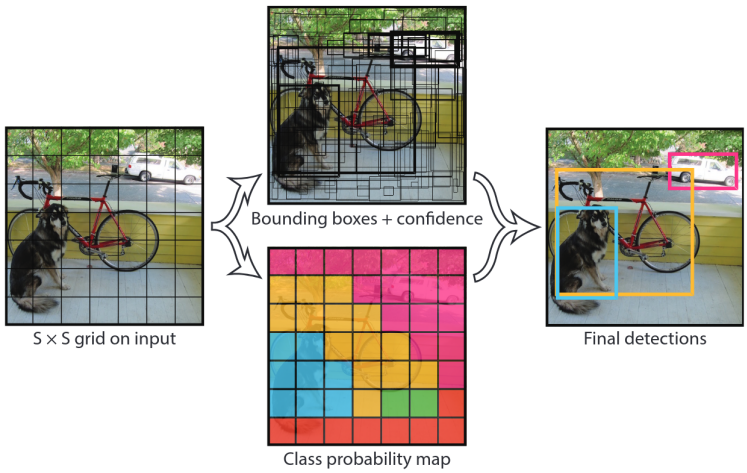


Figure 2.4: Visualization of the YOLO working concept[22].

This single image feed-through method proves to be a lot faster, and has no problem detecting in real-time.

Among the most commonly used object detectors, YOLO repeatedly receive the fastest FPS performance on data-sets like the COCO-dataset[23].
The YOLO framework supports several different model configurations like for example the ”tiny-yolo” versions, which applies a smaller network, with less accuracy, however,

making it a lot faster. The performance of the different configurations is affected by the known concept within object detection; the trade-off between speed and accuracy. The best configuration depends on the system's speed, accuracy, and hardware requirements.

2.2.11 pyyolo

pyyolo is a simple Python wrapper for YOLO[24]. Installing this package enables the use of the YOLO object detection model in a Python script. The package supports installation which exploits the GPU for computational power.

2.2.12 SORT

SORT is a simple, online and real-time tracker[25]. "Online" means the algorithms use only current and past detections. "Real-time" means the object identification is fast enough to run on a real-time detection system.
SORT uses the coordinates of detected objects bounding boxes (BBs) as input, and outputs an identification (ID) number corresponding to each BB. As a result, the SORT algorithm is easy to implement with any object detection model outputting BB coordinates, such as YOLO.
The prediction of the tracking ID is based on data association and state estimation techniques. More specifically SORT models each target (center of BB) with the following state vector:

$$\mathbf{x} = [u, v, s, r, \dot{u}, \dot{v}, \dot{s}] \tag{2.1}$$

where u and v represent the horizontal and vertical pixel coordinates of the target, while s is the scale and r is the aspect ratio of the target BB. The target state is matched with new detection BBs and updated with the optimal velocity component using a Kalman filter framework[26].
Further, data association is used to assign new detections to existing targets. The target BB coordinates are predicted in the current frame, based on the states. A reassignment cost matrix is then calculated as the intersection over union (IOU) distance between the detected BBs and the predicted BBs. The cost matrix is solved optimally using the Hungarian algorithm[27].

A limitation of SORT is that it does not output the matching predicted class with the ID and BB coordinates. This is a result of the algorithm being developed assuming all input BBs are people.

2.2.13 OpenCV - Haar Cascade

OpenCV is an open-source library of computer vision functions aimed at real-time operations [28]. Installing the package "cv2" enables the library to be imported into a Python project.
The library provides great tools for data preparation and for recording and displaying images. It also provides simple classifiers for detecting features in images. This includes Haar Cascade object detectors which can detect face and smile in images[29]. The Haar

Cascade networks are trained on detecting specific Haar features. A Haar feature can be horizontal or vertical lines and edges in the image. If the network is trained on detecting faces, it slides a filter over the image, while calculating the specific the Haar feature response, corresponding to a face.

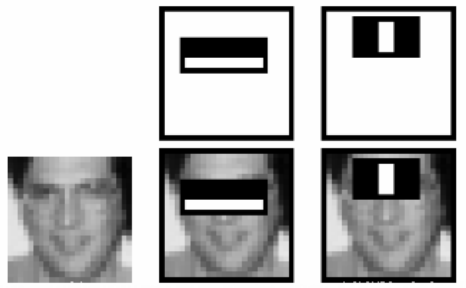


Figure 2.5: Typical Haar features on a face[29].

As shown in Figure 2.5, when detecting faces, it typically looks for a vertical lighter line in the middle of the sliding window, corresponding to the nose.

2.3 General Theory

This section presents the rest of the theory material used throughout the project.

2.3.1 Euclidean distance

Given a point $p = (x, y, z)$ in a 3D space, the straight line distance can be calculated with the formula for Euclidean distance[30]:

$$\mathrm{d} = \sqrt{x^2 + y^2 + z^2} \tag{2.2}$$

If the point is represented in a frame fixed to a camera, the result from Equation 2.2 is the distance from the camera to the point.

2.3.2 Relative Coordinates of Detected Object

Assuming we know the distance to the detected object, the only thing remaining is to estimate the angle to the detected object, before calculating the relative coordinates. If the distance, d, and the angle, θ_0, to the detected object are known, its relative coordinates can be calculated using the trigonometrical formulas for a right triangle:

$$x = \sin\theta_0 \cdot d \tag{2.3a}$$

$$y = \cos\theta_0 \cdot d \tag{2.3b}$$

The angle θ_0 to the detected object can be estimated by assuming the image represents the arch of a circle, as visualized in Figure 2.6.

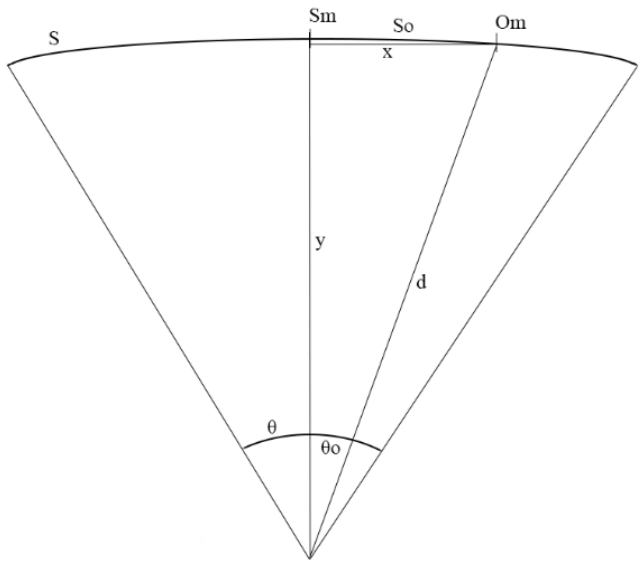


Figure 2.6: Visualization of the calculation of the detected object angle relative to the camera. Image is captured from the zedyolo report[7].

As described in the zedyolo report[7], the total length of the arch assumed to represent the image, S, can be calculated using the formula of arch length:

$$S = \theta \cdot d \tag{2.4}$$

Where θ is the field of view of the camera. Further, the portion of the total arch length corresponding to the angle between the center object and center of camera is found:

$$S_0 = \frac{P_0}{P} \cdot S \tag{2.5}$$

Where the portion value is the number of horizontal pixels in the image, from the center of the image to the center of the object, P_0, divided by the total number of horizontal pixels in the image, P.
Finally, the angle is calculated, using the restructured formula of arch length:

$$\theta_0 = \frac{S_0}{d} \tag{2.6}$$

2.3.3 Social Intelligence

Social intelligence can be defined as ”the ability to understand and manage people”, as suggested by psychologist Edward Thorndike [31]. Similarly, T. Hunt described social intelligence as ”the ability to get along with others” [31]. Since people are different, there is no strict definition of what perfect social intelligence is. However, there exist tests with the goal of measuring a person’s level of social intelligence, used by psychologists. A

famous one is ”The George Washington Social Intelligence Test”, created by the psychologist Dr.Thelma Hunt in 1928, at the University of Washington. The test is measuring the following social abilities[31]:

- Judgment in Social Situations
- Memory for Names and Faces
- Observation of Human Behavior
- Recognition of the Mental States Behind Words
- Recognition of Mental States from Facial Expression
- Sense of Humor

This test could also apply when measuring a robot's level of social intelligence. Some of the bullet points representing the test, depend especially on visually assessing another person: Memory of faces, observation of human behaviour and recognition of the mental states from facial expression. As a result, detecting facial features and human behaviour are relevant to consider when designing a CV module.
The test suggests that a robot capable of reacting appropriately based on a person's behaviour and facial expression, could be experienced as a socially intelligent robot.

2.3.4 Natural Human Behaviour

”Natural human behaviour” is in this report defined as the behaviour of the average person trying to interact with a robot, appearing with social characteristics. To try to describe this behaviour, the author studied several videos of people interacting with social robots, such as ”pepper” and ”NAO”, developed by SoftBank Robotics[32][33][34][35]. SoftBank Robotics specializes in making interactive and friendly robots[36]. A selection of the videos studied is shown in the collage in Figure 2.7.

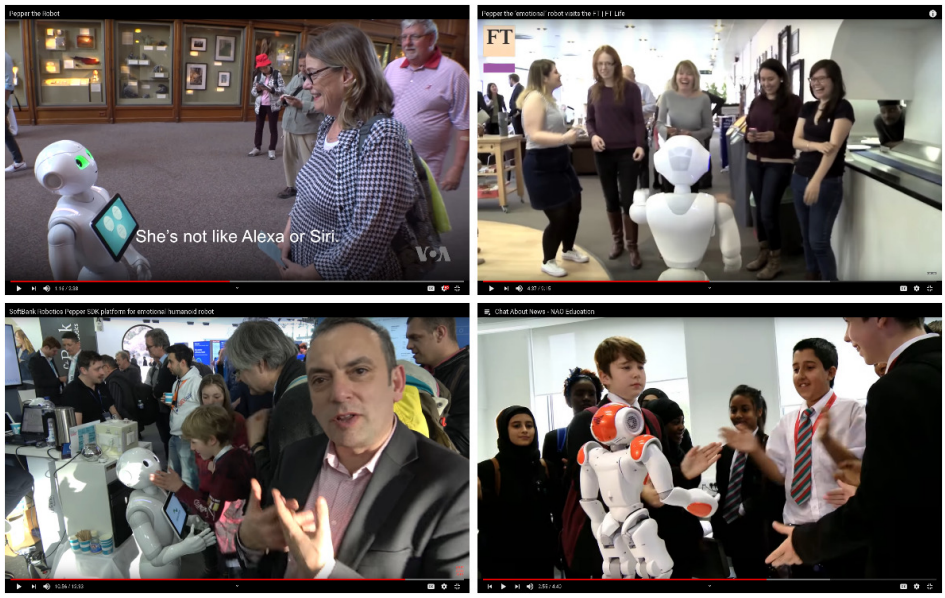


Figure 2.7: Video collage of random people interacting with social robots [32][33][34][35].

The videos studied show that most people, with no prior knowledge about the robots, are very unsure of how to interact. As a result, a presenter often guides the people on how to talk to, touch, or even dance with the robot. However, as seen in the videos, most people interested, stops in front of, and faces the robot, and sometimes even smile and laughs, regardless of any prior knowledge about the robot. Such behaviour could, as a result, be called natural human behaviour, when meeting a social robot.

2.4 The Starting Point: zedyolo

This section presents the reimplementation of zedyolo[7], which is the previously implemented CV system within the Cyborg project. This system is used as a starting point for further development in this project. The results of the reimplementation is the most significant section of this chapter. This is because the good solutions are adopted into the design of the CV module delivered in this project. Moreover, the solutions not fulfilling the system requirements in Table 3.1, are redesigned as presented in chapter 4. It is important to emphasize the limitations of zedyolo, discussed in this section, are not necessarily errors with the original zedyolo system. This is because the reimplemented system is not identical to the original system presented in the zedyolo report[7].

2.4.1 Reimplementation

The zedyolo project is cloned from the ”thentnucyborg” GitHub and set up using the installation guide in the project report[7].

zedyolo depends on the YOLO Python wrapper, pyyolo, and the ROS package, ZED-ROS-Wrapper. The newest ZED-ROS-Wrapper version, at the time of implementation, is cloned and installed from the Stereolabs GitHub. pyyolo is installed using the source files included in the zedyolo project. This means the reimplementation system is slightly different than the original implementation. The main differences between the original zedyolo and the reimplemented system is shown in Table 2.1 below.

	Original zedyolo	**Reimplemented zedyolo**
Developer board	Jetson TX2	Jetson TX1
Operating System	Ubuntu 16.04	Ubuntu 18.04
ROS Distribution	Lunar	Melodic
ZED-ROS-Wrapper commit	bb13787	bdc2fe1
ZED SDK Version	2.3	3.0.2

Table 2.1: Original vs reimplemented zedyolo system.

The zedyolo system is set up by first running ZED-ROS-Wrapper with the command:

```
roslaunch zed_wrapper zed.launch
```

Then by running the object detection ROS node:

```
rosrun object_detection zedyolo.py
```

The zedyolo system retrieves images from the ZED camera via a published Topic by the ZED-ROS-Wrapper. zedyolo only supports VGA image resolution, and as a result, this is configured in the file ”common.yaml”, which is used when the ZED-ROS-Wrapper is launched.

Also, due to the newer version of the ZED-ROS-Wrapper, some of the file-structures and default camera configurations are changed. This involves the name convention of the published camera images, and the format of the retrieved ZED images. The retrieved images are on the RGBA format with 4 channels instead of the 3 channeled RGB format which the original zedyolo implementation expected. The first three channels are the red, R, the green, G, and the blue, B, channels. The last channel, A, stands for alpha and are values between 0 and 1 which represent the transparency of the RGB channels[37].

2.4.2 Results

The system is set up with the ”yolov2” configuration and pre-trained weight files. The following subsections present the main results which do not meet the system requirements for this project.

Speed

The total cycling time of the system is 380ms, which is somewhat slow compared to the original zedyolo implementation, which reported a cycling time of just under 300ms. However, the recorded cycle is on the reimplementation including the visualization of the detection in real-time, which slows the system. Besides, the reimplementation is on a TX1, vs a TX2 in the original implementation. Some of the specs on the Jetson TX2 are upgraded, like a more powerful GPU, which could have affected the performance of the system as

well.
The cycle time of 380ms corresponds to the program only managing 2.63 FPS. This may be too slow when considering that the goal of the project is to add new CV features contributing to the Cyborg, which will reduce the frame rate even more. Of the total cycle time of 380ms, about 270ms is due to pyyolo detecting objects in the image. Consequently, reducing the object detection time should be prioritized.

Detected Objects Relative Position

One of the outputs of the zedyolo system is the relative position of the detected objects. This feature is not calculated correctly in the reimplemented system. Also, the calculated distance is not correct as it increases when moving objects closer to the camera, and decreased when moving them further away. Since the relative position calculation is based on the calculated distance, it suggests the position error source lies in the distance calculation.
The distance is calculated using the depth map produced by the ZED camera which is retrieved via published data from the ZED-ROS-Wrapper. A theory is that the format of the depth map, retrieved from the updated ZED-ROS-Wrapper in the reimplementation, is changed, causing the distance calculation to fail. Nevertheless, the distance calculations should be fixed when moving forward.

Location Dependence

The system is dependent on the location of the program launch. The system can only be launched from the source folder of the ROS package. A convenient feature available, when implementing the program as a Node in the ROS network, is the possibility of running the program from any location in the terminal, only knowing the package and program name. However, this feature requires none of the functions in the program to be dependent on the location of launch, which is the case of the zedyolo reimplementation.

Resolution Bound

A limitation noticed in the zedyolo reimplementation, which also is mentioned in the zedyolo report[7], is that the system is bound to the ZED camera ”VGA” resolution. If configuring a ZED camera resolution of HD720, which is the next step after VGA on the ZED camera, zedyolo fails. The ZED VGA resolution implies the images are captured with a dimension of 672x376. This is sufficient when detecting close objects, however, when a person moves further than 3 meters away from the camera, YOLO has trouble detecting correctly. The bound on the resolution could also be a limitation when implementing new CV features, requiring more detailed images. Examples of such CV features could be facial expression detection and hand gesture recognition.

Delay Time

The most striking potential for improvement noticed in the reimplementation is the delay time of the system. The delay time of the system is about 5 seconds. In other words, if

an object appears in front of the camera, it would take the zedyolo system 5 seconds to detect the object. This delay should be reduced, to not limit the performance of the other modules dependent on the CV module.

Integration with the Cyborg

The zedyolo system is integrated with a ROS network, however, it is not integrated with the Cyborg, and no solution is described for achieving this. A solution for achieving this should be explored moving forward in the project.

2.4.3 Conclusion

To summarize, the main zedyolo results which should be fixed in order to be adopted into the behaviourdetection system, are presented in the bullet points below:

- Speed
- Detected objects relative position
- Location of execution dependence
- Resolution bound
- Delay time
- Not integrated with the Cyborg

The design to fix these limitations is proposed in chapter 4, among the other design solutions fulfilling all the system requirements.
On the other hand, some of the zedyolo solutions satisfy the system requirements. These solutions are adopted into the final system, which is specified throughout the design chapter chapter 4.

Chapter 3

System Requirements

This chapter presents the CV module design and functionality requirements, that is used for development and in testing for quality assurance. These requirements are defined by the author. Before defining the requirements for the final system, it is appropriate to discuss the relevance of different CV information for the Cyborg robot.

3.1 Discussion of CV applications on the Cyborg

The motivation for the work done in this master project is that the resulting CV module and report can be used and build upon by future and ongoing Cyborg projects. As a result, the information published by the CV module should be relevant, precise, and fast enough for other modules to use as sensor data for their functionality.
The main Cyborg modules, besides the CV module, are the Navigation, Behaviour, and GUI module. Each of these modules has their own interest in information from a CV module. Relevant information provided by a CV module is discussed and presented for each module in the bullet points below:

- **GUI module**: The GUI module is implemented as a website, which can be used for monitoring and control of the robot. The CV module output could be especially useful for monitoring of the scene in real-time. A user of the Cyborg GUI could be interested in viewing the situation of the environment in which the robot is maneuvering. Is the environment crowded? What object is suddenly blocking the passage through the corridor? A user of the Cyborg GUI could use information about this when deciding how to control the robot. This would require the CV module to publish video stream from the camera on the Cyborg ROS network, in addition to counting of detected people.

- **Behavioural module**: The behavioral module is responsible for, among other things, the interaction with the surrounding objects and people. One of the goals of the project is to create a robot behaviour that is perceived as socially intelligent, engaging, and likable. These characteristics are difficult to perfectly achieve since people

respond differently to interactions. However, the author will further interpret engaging and likable characteristics based on assumptions about the general student. The interpretation of social intelligence is based on a test for measuring human social intelligence used by psychologists, as presented in subsection 2.3.3 in the Background chapter. The test suggests an important measure of social intelligence is the ability to observe human behaviour and facial expression.
Understanding a human's behavior is a very complex task, affected by several factors studied in psychology. These are factors such as culture, emotion, and personality, just to mention a few[38]. How is it possible to program a robot to understand human behavior, when sometimes even humans can't understand each other's behavior?
A start is to use CV to observe individual people and how they move. Naturally, busy students walking fast across the campus, late for class, will be less likely to appreciate interaction with the Cyborg robot. On the contrary, a person slowly walking and stopping in front of the Cyborg robot is more likely to be open for interaction. Noticing this behavior will require the CV module to detect people and track them individually in the environment.
The Cyborg could also detect more obvious signals indicating interest. Behaviour like waving or eye contact are stronger signals indicating a wish for interaction. This would require the CV module to detect hand gestures or eye movement. A more thorough discussion of how to detect interest, involving both human and technical aspects, is presented below in section 3.2.

- **Navigation module**: The navigation module is responsible for mapping, localization, obstacle avoidance, and path planning. Most of these features are working sufficiently using information from the already integrated sensors, like the laser scanner and sonars. These sensors give information about the location of obstacles, but not what the obstacles are. A CV module could contribute information about how crowded the environment is. The navigation module could use this information to decrease the speed to reduce the risk of accidents.
  Another possible application could involve the navigation, behavior, and CV module. The CV module tracks individual people, their location, and detects signs for interests for interaction. The behavior module evaluates the CV information, and as an example, tells the Cyborg to talk to, face, and follow the interested person. Next, the navigation module receives the command to face and follow the person with a certain ID. Then, the navigation module can subscribe to the published data from the CV module, and use the relative coordinates to the person with this ID as a target position.

Remember, the examples of Cyborg features discussed above, are not necessarily implemented by all the modules throughout the spring semester of 2020. These are suggested examples of how the Cyborg could work. The purpose is to have a clear vision of how the CV module output could be used, and develop the module keeping this in mind.

3.2 How To Detect Human Interest Using CV

The core of this master's thesis is to create a CV system that is able to detect natural human behavior. Especially, detecting human interest which can be used by the Cyborg as signals for a wish of interaction. Since this is not obvious from a CV systems point of view, several alternatives for detecting human interest are considered. The main methods are presented and discussed in the bullet points below:

- **Hand Gesture Recognition**: This will give the system a clear signal which people can use for communicating interest. However, this will require surrounding people to know which hand gesture to use, and what they mean to the Cyborg. As learned from studying human-robot interactions, presented in subsection 2.3.4, most people have no idea what to do unless a presenter suggests how to interact with the social robot. The goal is for the Cyborg to manage interactions completely by itself. Also, after briefly researching other systems using hand gesture recognition software, it is apparent the performance depends strongly on a controlled environment. Often the captured scene is close up of the hand, with a well lit up, uniform background. A CV system integrated with the Cyborg, maneuvering a dynamic scene like the NTNU campus, would have difficulties achieving such predictable scene characteristics. As a result, hand gesture recognition is rejected for implementation on the final system.
- **Object Tracking**: This will give the system information about the individual movement of surrounding people. This information will enable the Cyborg to distinguish between people, where they are located, and for how long. Humans standing close for a longer period, can be interpreted as a subtle indication of interest. Also, after brief research, this feature is found to be working in similar environments as the Cyborg will experience. As a result, this feature is selected to be included in the final system requirements.
- **Face & Smile Detection**: This will give the system an even stronger indication of interest, if combined with object tracking. The face detection will give information which can be used to distinguish between interested people facing the Cyborg, and people just standing close while not noticing the robot. The smile detection will give the Cyborg information about the emotional state of the person. This is information that definitely could improve the Cyborgs social intelligence. Another huge advantage of detecting these signals, is that they do not require the surrounding people to learn how the Cyborg detects interest like hand gesture would. Most people subconsciously smile and face the robot if they are interested, as learned from the "natural human behaviour" study, presented in subsection 2.3.4 in the Background chapter. Also, after a brief research, implementation of face and smile detection on the Cyborg is believed to be achievable. As a result, these features are selected to be included in the final system requirements.

3.3 Vision for a Cyborg Interaction

In this section the vision for how the Cyborg can interact with surrounding people is presented. The purpose of presenting this vision is to compliment the system requirements,

when making design choices and when evaluating the results. A system satisfying the defined requirements should enable the vision described in this section.
The author's vision for how an interaction with a random person could go: A student on his way from a lecture to a lunch break, stops in front of the Cyborg, curious about what is going on with this robot. The Cyborg stops. The Cyborg notices the same person is standing close and still in front of it for about 2 seconds. For the Cyborg, this indicates the person is potentially interested in an interaction. The Cyborg faces the person and detects the person is facing the Cyborg while also smiling. For the Cyborg, this is an even stronger indication the person is interested in an interaction, and also it knows the person is probably in a good mood. The Cyborg then reacts with something fitting for the detected situation, by for example saying hello, or telling a joke. The person thinks the Cyborg is a socially intelligent robot because it does not behave like this with every surrounding person, it seems to react based on the person's behaviour.

3.4 Final System Requirements

In this section the requirements for the final CV module are defined in more detail. The requirements are motivated by the applications discussed in the previous sections.

Requirement	Description
Output	The module should output the following information, with good accuracy: • Detected objects class • Detected objects relative position • Detected objects tracking ID • Detected people face indication • Detected people smile indication • Detected people counting • Camera recording stream
Detection Range	The module should manage to detect a person's behavior who is standing 2 meters away from the camera, or better.
System Integration	The module should be integrated as a package in ROS, and publish the output information on topics on the Cyborg ROS master, located on the Cyborg base computer, where the information is available for subscription by other modules. The the published data should be on a format which is manageable for other modules.
Real-time output	The output should be available for the other modules with maximum 0.5 seconds of delay.
Output frequency	The frequency of the data should be as high as possible, and at least 3Hz
Hardware	The system has to run on a Jetson TX1 Development kit, and use the first generation ZED Stereoscopic camera.

Table 3.1: Final system requirements.

Concerning the requirements ”Real-time output” and ”Output frequency” in Table 3.1, the most important thing is that these factors does not limit the modules subscribing to the CV data. For instance, the navigation module needs position data with almost no delay, to use for obstacle avoidance.

Chapter 4

Design

This chapter will describe the higher-level design which is used the final system. This involves the reasoning behind selected and rejected solutions, as well as, chosen methods, concepts, and program structures.

4.1 Location Independence of Launch

The location of the launch dependency problem is caused by pyyolo searching for the initialization files using a path defined relative to the current location, which is the location in the terminal where the program is executed. To fix this, all the search path definitions in the program is defined with the full path-name instead, which is independent of the launch directory.

4.2 Elimination of Delay

The source of the 5 seconds delay in the reimplemented zedyolo, is found to be the retrieving of images via the ZED-ROS-Wrapper. A visualization of how the reimplemented zedyolo retrieves images from the ZED camera is shown below in Figure 4.1:

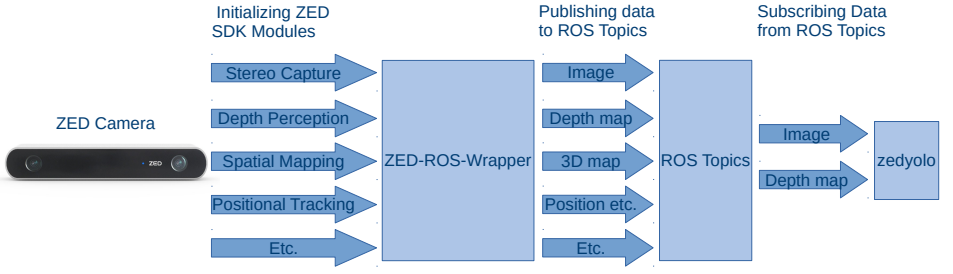


Figure 4.1: zedyolo retrieving images.

As shown in Figure 4.1, the ZED-ROS-Wrapper launches several ZED modules and publishes the data to ROS Topics. However, not all the modules provide data which is needed. This results in unnecessary large consumption of the Jetson board's resources. In addition, the data retrieved takes a detour through the ROS Topics, which also could be the reason for the delay.
As a result, a solution for retrieving data directly from the ZED camera is implemented. In fact, this solution does not require the ZED-ROS-Wrapper to run at all. The new design of retrieving data from the ZED camera is shown below in Figure 4.2:

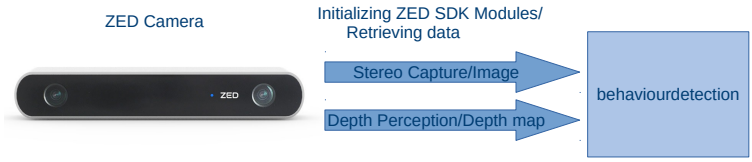


Figure 4.2: New design for retrieving images.

Retrieving data as shown in Figure 4.2, is made possible using functions included in the python package "pyzed.sl", accessible after installing the ZED Python API. This package enables the user to access the ZED SDK Camera class directly using Python, for interaction with the camera.
The design for retrieving data directly from ZED SDK is inspired by the methods used in the ZED tutorials for Python development[39].
The design shown in Figure 4.2, proves to completely remove the 5 seconds delay, experienced in the zedyolo reimplementation. This result is backed up by the testing presented in chapter 6.

4.3 Integration of Module as a ROS Node

To allow the behaviourdetection system to be integrated with the Cyborg it has to be integrated as a ROS Node. The structure of the Node is inspired by the example of a publishing Node, "talker.py", from the ROS tutorials[40]. This structure is chosen since it is a standard way of creating a ROS publisher, which should be easier understood by the author and future students working with ROS within the Cyborg project.
The behaviourdetection system is interfaced with ROS by initializing the system as a ROS Node and by publishing the output on ROS Topic. Initializing the program as a ROS Node establishes the communication with the ROS Master, which enables communication between all the Nodes known by the Master. Since only one ROS Master can run within the same ROS environment, every initialized ROS Node is automatically connected to the same Master.
All the work to be done for each image, including detecting objects, calculating coordinates and publishing to ROS Topic, etc., is placed inside a while-loop which checks if the Node should be running. This is achieved by using the flag "rospy.is_shutdown()" as

the condition in the while loop. For example, this will force the program to exit if using ”Ctrl-C” in the terminal window.

4.4 Integration of Module with the Cyborg ROS Network

To fully integrate the behaviourdetection system with the Cyborg robot the ROS network on the Jetson board is configured to connect with the ROS Master on the Cyborg base computer. The method for integrating ROS networks on different machines is inspired by an example of how to set up rvis over multiple computers[41].
When running ”roscore”, ROS sets up the master at the location defined in the ROS environment variable ”ROS_MASTER_URI”. This is where the ROS Nodes will look for the ROS Master. If the ”ROS_MASTER_URI” environment variable on the Jetson board is defined with the IP address of the Cyborg, the nodes on the Jetson board will register to the ROS Master on the Cyborg.
The IP address of the Cyborg base computer is defined as the Master location for the ROS network on the Jetson board. Specifically how this it set up is described in subsection 5.2.5.
The two ROS networks communicate over WiFi. A high-level visualization of the integration with the Cyborg is shown below in Figure 4.3:

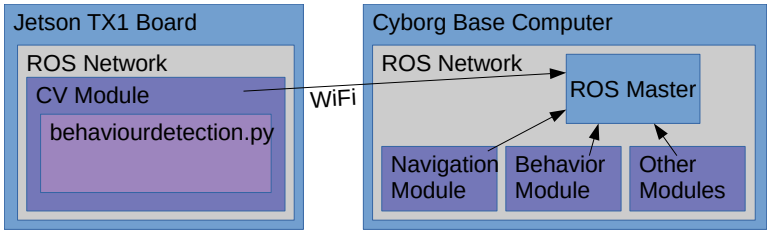


Figure 4.3: High level visualization of the CV module integration with the Cyborg ROS Network.

The simple setup shown in Figure 4.3 will enable all the Nodes on the Cyborg to subscribe to the Topics published by the Nodes on the Jetson board, and vice versa.

4.5 Detected Objects Relative Coordinates

The coordinate calculation design consists of two parts, and is partially an adopted solution from the zedyolo system. First, calculating the straight line distance from the camera to the detected object. Then, calculating the coordinates, based on the distance and the center of the detected objects Bounding Box(BB).
As described in the results of the zedyolo reimplementation in section 2.4.2, the error source is found to be the calculation of the straight line distance from the camera to the detected object. As a result, a new method for calculating this distance is designed.
The new method for calculating distance is inspired by the example code ”depth_sensing.py”, from the ZED tutorials[42]. Instead of using the depth map, this method calculated the

straight line distance using the point cloud map retrieved from the ZED camera. The point cloud returns the image-pixel-colour-values and their corresponding xyz-coordinates, relative to the camera, in millimeters. These values for a specific pixel can be accessed with "point_cloud.get_value(x, y)", where x and y are the concerning pixel image coordinates. Similar to the zedyolo system, this point is chosen to be the center of the detected BB.
Further, the distance is calculated using the euclidean distance[30], just like in the ZED tutorials:

$$\text{distance} = \sqrt{(X[\text{mm}])^2 + (Y[\text{mm}])^2 + (Z[\text{mm}])^2} = \sqrt{X^2 + Y^2 + Z^2}[\text{mm}] \quad (4.1)$$

The resulting distance is given in millimeters since the xyz-coordinates retrieved from the ZED point cloud are in millimeters.
Further, the relative coordinates are calculated based on the distance in Equation 4.1. For this part, the function "calculate_coordinates" in zedyolo.py is reimplemented[43]. This function uses the distance to the detected object, the center of its BB, together with the camera intrinsic parameters to calculate the angle and coordinates relative to the camera. The theory behind this calculation is presented in subsection 2.3.2, in the chapter Background.

4.6 ZED Camera Configuration

Especially two camera configurations are found to affect the system performance. These camera settings are presented in the bullet points below:

- **Exposure**: Throughout the implementation and testing of the object detection algorithms, the author experienced that unstable detection primarily is caused by objects becoming unclear and blurry when moving. To reduce the motion blur in the images, the ZED camera exposure and capture frame rate are configured, which directly and indirectly adjust the shutter time, respectively. Decreasing the shutter time, results in sharper images, however, it reduced the brightness. First, the ZED camera frame rate is fixed to 15 FPS, which is sufficient since the program's total cycle time is assumed to never exceed a speed corresponding to this frame rate. Further, setting the exposure to 30% of the frame rate results in a good trade-off between reduced motion blur and brightness. Keep in mind, the best configuration depends on the light intensity of the scene which the camera is capturing. For example, if applying the CV system in "Glassgården" at NTNU, which is well lit up by the daylight, the exposure could probably be reduced even more.

- **Resolution**: Throughout the project, different image capture resolutions are experimented with. The next possible step up in resolution from VGA on the ZED camera is HD720, which captures images with dimension 1280x720. The next step up in ZED resolution after this is HD1080, which is evaluated to be an unnecessarily high resolution for this project. As a result, the VGA and HD720 are the only relevant

resolutions which are evaluated.
It is found that the object detection speed and accuracy is somewhat independent of the image capture resolutions. This is assumed to be due to YOLO at default downsizing the images to at least 608x608 on the input of the network, depending on the initialization. However, the other parts of the system, like the visualization and the image manipulation, slows the system significantly, when increasing the image resolution. After all, sufficient detection performance of smaller features like face and smile, over 2 meters from the camera, is found to require a higher resolution of HD720. As a result, to fulfill the detection range requirement of 2 meters, defined in Table 3.1, the HD720 resolution is chosen as the best configuration.

4.7 Object Detection

The structure for detecting objects is inspired by the program ”example.py” provided on the pyyolo GitHub repository[24]. The structure involves normalizing and transposing the images, before sending them through the detection network. The detection network is initialized with different configurations and the corresponding pre-trained weights, which are published by the creators of YOLO[19]. Most of the YOLO configurations are trained on the COCO dataset[44], which means it can detect 80 different objects. The pyyolo detection function returns the detected objects BB coordinates, class, and probability. To easily evaluate the detection performance, each BB is drawn on the images, which is visualized in real-time.
The methods for object detection explored in this project can be separated into two: The pyyolo 2018 version compatible with up to YOLOv2, and the pyyolo 2020 version compatible with up to YOLOv3. Both of these methods' performances are tested and evaluated in the result chapter 6.

4.7.1 The 2018 pyyolo Version

This pyyolo version is installed and built using the old source files included in the zedyolo repository cloned from ”thentnucyborg” GitHub. This corresponded to a pyyolo version from early 2018. The 2018 pyyolo design supports the first and second generation of YOLO configurations, but not the third; YOLOv3. Also, it does not support higher ZED resolutions than VGA.

4.7.2 The 2020 pyyolo Version

At the time of the project development presented in this report, the spring of 2020, a newer version of pyyolo is available. The new version of pyyolo is, among other updates, compatible with the YOLOv3 configuration. Considering the limitations experienced with the old pyyolo, the newest version of pyyolo is built and installed using updated source files cloned from digitalbrain79's GitHub[24].
The 2020 pyyolo version enables configurations up to the newest YOLOv3 versions, and it does not bound the image resolution.

4.8 Multiple Object Tracking

Implementing object tracking will give the Cyborg robot information about individual objects movement over time, by giving each object a tracking ID. To achieve this, two methods for tracking objects are considered; Basic SORT and SORT[25].

4.8.1 Basic SORT

The first implemented tracking algorithm is developed by the author and is named ”Basic SORT” in this report. The code can be found under the appendix section A.3.
”Basic SORT” is a simple object tracking algorithm, which assigns a tracking ID to each detected object. The concept of the algorithm is simple; if the detected objects bounding box (BB) coordinates in the current frame are close to the corresponding coordinates of the BB detected the previous frame, the two BBs probably indicate the same object. The concept of measuring BB displacement used in Basic SORT is visualized in Figure 4.4 below:

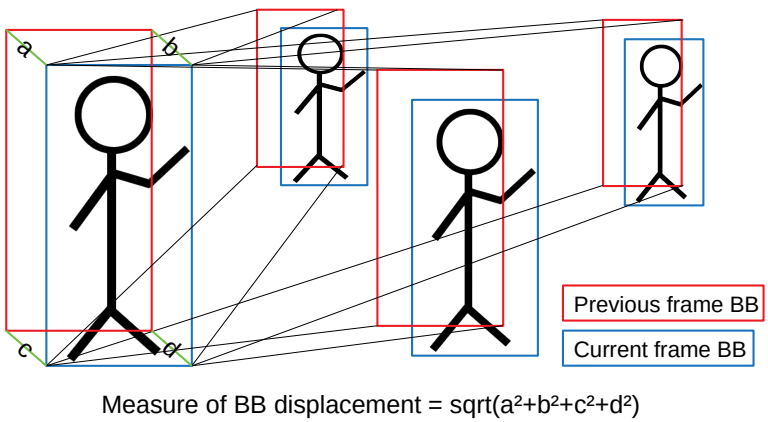


Figure 4.4: Visualization of the measure of BB displacement.

As shown in Figure 4.4, the measure of how close the BBs between the frames are is indicated by calculating the euclidean distance between the BBs coordinates. If the distance is below a certain threshold, the BB in the current frame is reassigned to the same ID as the BB in the previous frame. This threshold is tuned to a sufficient value through testing. Since sometimes several BB can be closer than the threshold, the algorithm chooses the closest BB which ID is reassigned. The method for reassigning an ID is shown in the pseudo-code in algorithm 1.

if $\sqrt{a^2 + b^2 + c^2 + d^2}$ <*threshold* **then**
  if $\sqrt{a^2 + b^2 + c^2 + d^2}$ <*all other BB pairs* **then**
    reassign ID
  end
end

Algorithm 1: The condition for reassigning ID in Basic SORT.

If no match is found in the previous frame, it checks the frame before that. If still no match is found, the object is given a new ID.

During testing it was clear the BB distance conditions for reidentifying objects far away, needed to be stricter than for close objects. As a result, a modification of the Basic SORT algorithm is designed and tested with a dynamic instead of a static threshold condition, when reassigning the same ID. The dynamic condition is defined as (threshold) multiplied by the (area of BB), as shown in algorithm 2.

if $\sqrt{a^2 + b^2 + c^2 + d^2}$ <*threshold·BBarea* **then**
...
end

Algorithm 2: Updated dynamic condition for reassigning ID in Basic SORT.

Since the area of the far away BBs are smaller, the condition is harder to fulfill.

4.8.2 SORT

In the search for even better solutions, SORT[45] is further tested, since it is ranked the best open-source multiple object tracker on the MOT benchmark[46]. SORT is more a more advanced multiple object tracker, compared to Basic SORT, since it in addition to distance, evaluates the velocity of the BB, using Kalman state estimation.
The SORT input is the detected BB coordinates plus the probability, and the output is the detected BB coordinates plus the assigned ID. It does not keep track of the class of the object detected. The algorithm assumes every detected object is a person. As a result, when implementing SORT with behaviordetection, the system is modified to only track detected people. The method for integrating SORT into the behaviourdetection system is inspired by the script ”sort.py” included in the SORT GitHub repository[45].

The object tracking algorithm Deep SORT, was also considered[47]. This algorithm is an extension of SORT, including a deep appearance descriptor of the tracked objects. The descriptor helps to reassign the correct ID if the tracking of an object is lost, then reappears. Still, the SORT algorithm is evaluated as good enough to fulfill the system requirements. Also, the author suspects Deep SORT will slow the system even more than SORT. As a result, Deep SORT is rejected for implementation in the final behaviourdetection system.

4.9 Face and Smile Detection

The face and smile detection are designed using the Haar Cascade Classifiers provided by the OpenCV Python library ”cv2”[28]. This library provides cascade filters pre-trained for detecting face and smile, among other human features.
The algorithm design is inspired by an article written by Stephan Filonov[48]. The design uses a common concept used in object detection: ”From big to small”. In other words, an effective way of detecting small objects is to first detect a larger object containing the smaller object. This directly translates to how the smile is detected. The sequence of the face and smile detection algorithm is presented in Figure 4.5 and the corresponding the numbered list below:

Figure 4.5: Visualization of the face and smile detection procedure.

1. The full image is fed through the YOLO object detection network.

2. The upper half body image inside the detected person’s BB is cut out from the full image.

3. The upper half body image is then fed through the OpenCV face detection network.

4. The face image inside the detected face BB is cut out.

5. The face image is then fed through the OpenCV smile detection network.

Similarly, this could be extended to include the detection of the pupil inside the eye inside the face. The performance of the detection strongly depends on the resolution of the image and the lighting of the face. The resolution of HD720 is definitely preferred over VGA.
The face detection algorithm design exploits the geometry of the human body. Since the face is always positioned on top of the body, only the upper body is evaluated when detecting faces. Similarly, since the smile is always positioned on the lower part of the face, smiles detected in the upper half face are ignored. These assumptions are valid unless the person is upside down in the image, which rarely occurs.

4.10 Integration of CV Module with the Cyborg Modules

To further fulfill the requirement of integration with the Cyborg, a program called ”subscribertest.py” is created. The purpose of the program is to show how the published data from the behaviourdetection system can be retrieved by, and integrated with, another module. To fully demonstrate how the CV module on the Jetson board can be integrated with the Cyborg, a test system is designed as shown in Figure 4.6 below:

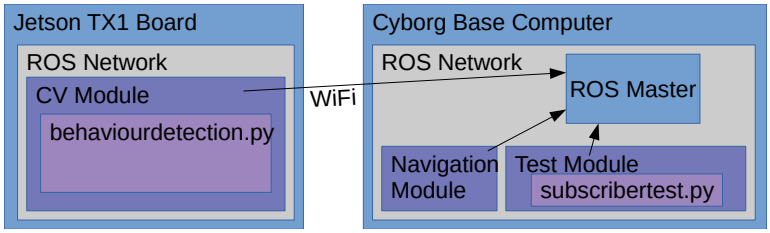


Figure 4.6: Test system which integrates the CV Module with a Cyborg Test Module.

In theory, future students working with the cyborg project, can set up the test system described in Figure 4.6, with the implementation in subsection 5.2.5, and replace ”subscribertest.py” with their own ROS Node. In addition, the method for handling the published data from the CV module, shown in ”subscribertest.py”, can be adopted into the new ROS Node. More specifically, the program shows how to extract info about a person’s ID and if this person in smiling. In addition, the program prints the video stream published by the CV module. Both of the described applications of the CV data are considered possible integrations with existing Cyborg modules, as described in section 3.1.

The method for testing, described in Figure 4.6, requires the total CV system to be set up, including the Jetson board, all the software programs, and package dependencies. As a result, to allow other students within the Cyborg project, a simple way of testing the integration of the CV module, a rosbag file is created. The rosbag file named ”testbagfile.bag” located in the ”thentnucyborg/ComputerVision” GitHub repository, contains a sample of recorded data published from the behaviourdetection system. By running ”rosbag play testbagfile.bag” in the ROS workspace, the sample of data is published on the ROS Topics, just as if the actual CV module was running.

4.11 behaviourdetection Program Flow

In this section the structure of the behaviourdetection program is presented in pseudo-code, as shown below in algorithm 3. This will give a general understanding of the flow of the system.

```
Import all program dependencies
Initialize publishing ROS Node
Initialize YOLO object detector
Initialize OpenCV face and smile detectors
Configure ZED camera
Open ZED camera
while program should be running do
    if ZED camera data is available then
        Retrieve ZED image and point cloud
        Format data
        Detect objects in image using YOLO
        Keep only detected people
        Update tracking ID using SORT
        for tracked objects do
            Calculate straight line distance
            Calculate relative coordinates
            Detect face and smile using OpenCV
            Format data to be published on ROS Topics and Visualization
        end
        Publish data on ROS Topic
        Visualize results
    end
end
```

Algorithm 3: Structure of the behaviourdetection system.

Chapter 5

Implementation

This chapter will describe how the final CV system is implemented. This involves guides on how to set up hardware and software, in addition to, how the system is coded. For some of the software tools, the author refers to external tutorials and installation guides, since these provide sufficient information. If the implementation is not straight forward, it is explained in more detail throughout this chapter.

The first part of the implementation is to set up the software and hardware. This includes the physical components, the operating system, and the system software dependencies. To enable future students to reimplement the exact same system, the version and git commit for each system component is presented, as shown in Table 5.1. The second part of the implementation is the coding of behaviourdetection.py and the dependent CV functions, which are described in section 5.7.

System Component	Description
ZED Stereo Camera	Generation: 1. Firmware: 1523.
Jetson TX1 Developer Kit	Flashed with JetPack 4.3 or 4.2
OpenCV	Version: 3.3.1
CUDA	Version: 10.0.326
L4T	Version: 32.2.3
Ubuntu	Version: 18.04
ZED SDK	Version: 3.0.2
Python	Version: 2.7.17
ROS	Distribution: Melodic
ZED Python API	Commit: 8e77500
pyyolo	Commit: 3d1969c
Darknet	Commit: f6d8617
SORT	Commit: 54e63a7

Table 5.1: System Description.

5.1 Hardware Setup

The Jetson TX1 developer board should be flashed with JetPack, which installs several developer tools and an L4T desktop environment. Among the developer tools already installed with JetPack relevant to the CV system are CUDA and OpenCV. The L4T operating system on the Jetson board has both Python 2 and 3 pre-installed. Throughout this project the Python 2.7 version is mainly used, since ROS 1 works best with Python 2, and all of the current Cyborg modules are using the ROS 1 framework. ROS2, on the other hand, was built with Python 3 in mind[49].
If the Jetson TX1 developer board is not correctly flashed with JetPack, re-flashing can be done by connecting the board to a host computer running Ubuntu via the micro-b USB port, then flash using the Nvidia SDK Manager and following their guide[50].

During development the Jetson board is connected to an external monitor, a keyboard, a mouse, and the ZED camera. Since the board includes only one USB port, a USB hub is connected to allow the setup. To extend the disk space on the board an 8GB SD card is inserted. The board is connected to the internet via Ethernet cable to a router. In addition, the supplied antennas are mounted on the board to allow internet connection via WiFi.
At this point, the Jetson board can be powered on, and the system should be ready for further implementation.

5.2 ROS setup

Before diving into the ROS setup, for future students, the author strongly recommends getting familiarized with the basic Linux command-line tools. The installation and general interaction with ROS happen thorough commands in the terminal window. This also concerns most of the installation and interaction with the other programs used in this project. A recommended tutorial for getting introduced to the basic Linux command line tools is published by the University of Surrey[51].

5.2.1 Installing and Configuring ROS Environment

ROS Melodic is installed using the official Melodic installation guide[52]. A catkin workspace named ”catkin_ws” is created, by following the ROS tutorial ”Installing and Configuring Your ROS Environment”[53]. ROS Melodic is installed instead of Kinetic which the Cyborg base uses, since Kinetic does not support Ubuntu 18.04. However, both distributions are ROS, as opposed to ROS2, so it is possible to integrate for communication between modules on the Cyborg and the Jetson board[54].

5.2.2 Creating and Building ROS Package

A ROS catkin package is created named ”jetsontx1_cvmodule”, following the ”Creating a ROS Package” ROS Tutorial[53]. This package is what is referred to as ”CV-Module” throughout this report. The command for creating a catkin package automatically sets

up the structure of a basic ROS package, including the files ”CMakeLists.txt” and ”package.xml”. These files are modified to define dependencies and include custom messages, which is described in subsection 5.2.4, before building the package.
The package is build using the command ”`catkin_make`” in the catkin work-space folder ”catkin_ws” terminal window.

5.2.3 Creating Publishing and Subscribing Nodes

After creating the ROS package ”jetsontx1_cvmodule”, it is in addition to the files ”CMakeLists.txt” and ”package.xml”, automatically created a folder named ”src”. This folder is where the ROS Nodes should be placed, and where the behaviourdetection.py program is created. The behaviourdetection.py system is created inspired by the example of a publishing ROS Node ”talker.py”, in the ROS tutorial ”Writing a Simple Publisher and Subscriber”[53]. The exact coding of the publishing ROS Node behaviourdetection.py is presented in section 5.7.
Similarly, the program for testing system integration ”subscribertest.py”, is created inspired by the example of a subscribing ROS Node ”listener.py”, in the ROS tutorial ”Writing a Simple Publisher and Subscriber”[53], which code is presented in section 5.8.
To make the Nodes executable the commands `chmod +x behaviourdetection.py` and `chmod +x subscribertest.py` are used in the src folder terminal. This allows the Nodes to be launched from anywhere in the terminal window using:

```
rosrun jetsontx1_cvmodule behaviourdetection.py
rosrun jetsontx1_cvmodule subscribertest.py
```

Before launching, all the messages used for publishing and subscribing by the Nodes, should be created and built, which is described in the next subsection.

5.2.4 Creating ROS msg

For a ROS Node to be able to subscribe and publish to a Topic, the message type of the data needs to be imported into the script. These message types can be imported from packages providing common message types or custom made message types.
For the publication and subscription of the ”videostream” Topic, the message type ”Image” is imported from the ROS package ”sensor_msgs”[55], which provides several other message definitions for common sensor types.
However, for the other Topics; ”predictions” and ”peoplecount”, the message type is customized for this project, by following the ROS tutorial ”Creating a ROS msg and srv”[53]. First, a folder named ”msg” is created inside the package folder ”jetsontx1_cvmodule”, since this is where ROS will look for message types to build. Inside the msg folder, the files ”Prediction.msg” and ”Predictions.msg” are created for the Topic ”predicitons”. The two files are copied from the zedyolo GitHub repository[43], and edited to include data on detected face and smile and tracking ID, as shown in Listing 5.1 and Listing 5.2 below.

```
1 Prediction[] predictions
```

Listing 5.1: Predictions.msg

```
1 string[] classes
```

```
float64[] probabilities
int64 xmin
int64 ymin
int64 xmax
int64 ymax
int64 id
string face
string smile
float64 distance
float64 angle
float64 xcoord
float64 ycoord
```

Listing 5.2: Prediction.msg

As seen in Listing 5.1 Predictions.msg is defined as an array of the message type Prediction.msg, shown in Listing 5.2. This allows the behaviourdetection system to publish only one message including predictions about all the detected objects for each frame. As opposed to, as an example publishing 10 messages to the Topic ”predicitons” for each frame, if 10 objects are detected.
Further, the file ”Peoplecount.msg” is created in the msg folder, for the Topic ”peoplecount”:

```
int64 tot_detected_people
```

Listing 5.3: Peoplecount.msg

The behaviourdetection system will use the message type Peoplecount.msg, as seen in Listing 5.3, to publish the number of detected people to the Topic ”peoplecount”. As seen in Listing 5.3, the ROS message type files should be defined on the form; ”field type” ”name”.

Now that the required message types are created, the msg files can be turned into source code for Python, which will enable the message types to be imported into a Python script. To achieve this, the ”CMakeLists.txt” file in the ”jetsontx1_cvmodule” ROS package folder, is edited to include the messages; Prediction.msg, Predictions.msg and Peoplecount.msg. Also, the ”package.xml” file is edited to include ”message_generation” and ”message_runtime”. For a better description of how to edit these files, the author refers to the ROS tutorial ”Creating a ROS msg and srv”[53].
Further, the Python source messages files can be built by building the ROS packages: use the command ”`catkin_make`” in the catkin work-space folder ”catkin_ws” terminal window.

5.2.5 Connecting to remote ROS Master

The setup described in this subsection allows the CV-module on the Jetson TX1 board to be integrated with the Cyborg robot, via WiFi, as described in section 4.4. The method is inspired by an example of how to set up rvis over multiple computers[41]. Specifically, the setup described allows ROS Nodes located on different machines to publish and subscribe to each other's Topics. Due to limited access to the Cyborg, the spring of 2020, the actual Cyborg base computer is not integrated. Instead, the CV-module on the Jetson board is

integrated with the ROS Melodic network on an Acer laptop, running Ubuntu 18.04. Also, the setup procedure is tested with a Dell PC running ROS Kinetic and Ubuntu 16.04. However, since the Cyborg base computer is running an equivalent system, the setup should be almost identical. Following, the procedure for connecting the ROS environment on the Jetson TX1 to the ROS Master on the Acer PC is presented:
In Acer PC terminal:

1. Check Acer PC IP address with command: `ip address`
   Result: `123.456.789.36`
2. Define location of Acer ROS master to be Acer PCs IP address with command: `export ROS_MASTER_URI=http://123.456.789.36:11311`
3. `export ROS_IP=123.456.789.36`
4. Start ROS Master with command: `roscore`

In Jetson TX1 board terminal:

1. Check Jetson TX1 board IP address with command: `ip address`
   Result: `123.456.789.25`
2. Define location of Jetson board ROS master to be Acer PCs IP address with command: `export ROS_MASTER_URI=http://123.456.789.36:11311`
3. `export ROS_IP=123.456.789.25`

The ROS network on the Jetson board will connect a launched Node to the remote master on the Acer PC, if the Node is launched in the **same terminal window** as the ”export ROS_MASTER_URI” command.

Note 1: The ROS environmental variables need to be set up in every new command window you open. To avoid this, add the export commands to the .bashrc file. The commands defined in the .bashrc file will be executed for every new terminal window opened.

To test if the ROS networks are integrated, the total integration test system can be launched:
In Jetson TX1 board terminal:

1. Start behaviourdetection system with command:
   `rosrun jetsontx1_cvmodule behaviourdetection.py`

In Acer PC terminal:

1. Start subscribertest system with command:
   `rosrun acer_testmodule subscribertest.py`

If the test program subscribertest.py prints visualises the video-stream as expected, the ROS networks are successfully integrated.

Note 2: The total integration test requires subscribertest.py to be set up like a ROS Node, in a ROS package named ”acer_testmodule”, build inside a catkin ROS work-space, on the Acer PC. In addition, the ROS msg files; Prediction.msg and Predictions.msg,

imported by the subscribertest.py Node, needs to be built on the Acer PC. This can be achieved by following the same ROS setup described in the above subsections in section 5.2.

To test if the integration was successful, the published output from the behaviourdetection can also be printed manually from the ROS network on the Acer PC:
In Acer PC terminal:

1. Check the published prediction using the command:`rostopic echo /predictions`

2. Or check the ROS communication visually with rqt_graph launched with the command: `rosrun rqt_graph rqt_graph`

If the topic ”predictions” prints out detection information as expected, the ROS networks are successfully integrated.
The author suggests by replacing the Acer PC with the Cyborg base computer, the same procedure described throughout this subsection, can be used to integrate the CV-module on the Jetson board with the Cyborg. More specifically, using a terminal window on the Cyborg, and defining `ROS_MASTER_URI` with the Cyborg base computer IP address.
Also, the same method should be possible using an Ethernet connection instead of WiFi, if this is preferred.

5.2.6 Recording and Playing Published Data

ROS supports recording and playing back published messages to ROS Topics by using the `rosbag` command. First, launch behaviourdetection.py and use the following command to record the published Topics:

```
rosbag record -a
```

Then, type Ctrl-C to stop recording. Run the module subscribing to the published data from behaviourdetection.py. For instance, launch subscribertest.py.
Finally, play back the recorded data:

```
rosbag play testbagfile.bag
```

This will publish the recorded messages to ROS Topics. subscribertest.py should now visualize the sample of the published topic ”videostream”. Also print in the terminal information about detected people ID and smile/face indication.

5.3 ZED SDK setup

Since CUDA is automatically installed with JetPack, the system is ready for the installation of ZED SDK. The newest version, at this time of the project, ZED SDK 3.0.2 for JetPack 4.3 is installed. This version of ZED SDK is installed since it supports the software this Jetson board is running, including the CUDA and L4T versions, as described in Table 5.1.
First, the ZED camera is connected to the Jetson board via the integrated USB 3.0 cable. Then, the procedure of installing ZED SDK is completed by following the guide published

by Stereolabs[56].
After completing the installation, to check if the set up was successful, the ZED SDK tool ZED Explorer is launched by changing directory to the location of the ZED SDK tools:

```
cd /usr/local/zed/tools/
```

Then running ZED Explorer:

```
./ZED_Explorer
```

If an update of the ZED firmware is available this will automatically be notified when launching ZED Explorer. After updating the ZED firmware, the ZED Explorer should display the captured image from the ZED camera. Similarly to the ZED Explorer, other provided tools like ZED Depth Viewer and ZED Diagnostic can be launched. ZED Diagnostic will run tests on the system to check if the system is correctly set up. For example, if the systems GPU is not compatible, the ZED Diagnostic will indicate this.

5.4 ZED Python API setup

With ZED SDK installed, the ZED Python API is installed to enable the use of the ZED camera in Python. ZED Python API is dependent on the python packages Numpy and Cython, which is installed using pip:

```
pip install cython numpy
```

ZED Python API GitHub repository is then cloned, built, and installed, using the provided README instructions[18]. The package is built and installed for Python 2.7, as opposed to Python 3, since the rest of the project is going to use Python 2.7.
After successfully installing ZED Python API, the package pyzed can be imported into a Python script like following:

```
import pyzed.sl as sl
```

This enables access to all the functions for interacting with the ZED camera.

5.5 pyyolo setup

Installing pyyolo enables the YOLO object detection network to be used in Python.

5.5.1 Build and Install

The pyyolo source files are cloned recursively into the src folder of the ROS package folder ”jetsontx1_cvmodule”:

```
git clone --recursive https://github.com/digitalbrain79/pyyolo.git
```

The pyyolo repository has to be cloned recursively to include the Darknet source files, which is an independent GitHub repository. pyyolo could also be cloned into any other location in the file system, as long as it is included in the Linux search path, i.e. the Linux environmental variable ”PATH”[57].
Before building and installing, it is important to check if the Makefile is configured correctly to function with the system. The Makefile in the pyyolo folder is edited with following:

1. Set variables GPU=1 and CUDNN=1, to use GPU computation.
2. Comment out all the ARCH definitions. The ARCH definition will build the system for a specific GPU architecture. If this is not correct the system will fail. By commenting the ARCH definition out in the pyyolo Makefile, the system will instead use the ARCH definitions in the Darknet Makefile when building, which proves to be sufficient.

Further, pyyolo is built and installed using the following commands as instructed in the README[24]:

1. `make`
2. `rm -rf build/` (If rebuilding and a build folder already exist)
3. `python setup_gpu.py build`
4. `sudo python setup_gpu.py install`

5.5.2 Configure

The different YOLO configuration files are included in the ”cfg” folder inside the darknet folder. The corresponding pre-trained weight files are downloaded. The pre-trained YOLOv3-tiny weights are downloaded by running the following command in the pyyolo folder terminal window:

```
wget https://pjreddie.com/media/files/yolov3-tiny.weights
```

The network is configured as YOLOv3-tiny by setting the following variables in the behaviourdetection.py script:

```
cfgfile = 'cfg/yolov3-tiny.cfg'
weightfile = '../yolov3-tiny.weights'
```

This is described in more detail in the behaviourdetection.py code implementation, in section 5.7.
To configure pyyolo as for example YOLOv2, just substitute ”yolov3-tiny” with ”yolov2”, and repeat the procedure.

5.6 SORT setup

The SORT source files are cloned from the GitHub repository[45] into the src folder of the ROS package ”jetsontx1_cvmodule” with following command:

```
git clone https://github.com/abewley/sort.git
```

The main file ”sort.py” is then copied into the src folder of the ROS package ”jetsontx1_cvmodule”, which then enables SORT to be imported into the behaviourdetection.py Python script with following line of code:

```
from sort import *
```

Further, the dependencies of the SORT system is installed using following commands:

1. `pip install filterpy`

2. `pip install numba`

3. `pip install scikit-image`

If the scikit-image installation fails, just comment out following line of code in the sort.py script:

```
from skimage import io
```

This is possible since the package is only used to show images when running sort.py itself, not when importing the sorting algorithm, which is done in this project.

5.7 Coding behaviourdetection.py

In this section the key lines of the behaviourdetection.py code are presented. The complete script is found in the appendix section A.1. The code is presented in sections divided into lines executed once, in ”Initialization”, lines executed repeatedly, in ”Main Loop”, and at last sections presenting the additional function blocks.

5.7.1 Initialization

It is important to start the script with the following ”shebang”:

```
#!/usr/bin/env python
```

This tells the system to interpret the program as Python code when launching with the command: `rosrun`.
Further, the script is initialized to publish to the ROS Topics predicitons, peoplecount and videostream, like following:

```
pub = rospy.Publisher('predictions', Predictions, queue_size=10)
```

Initialize the script as a ROS Node to register to the ROS Master:

```
rospy.init_node('detector', anonymous=True)
```

Define the full path to the Darknet files to allow the program to be independent of launch location:

```
darknet_path = '/home/ubuntu/catkin_ws/src/jetsontx1_cvmodule/src/pyyolo
  /darknet' # Only './darknet' is dependent on location of rosrun
  command
```

Initialize pyyolo with the configuration files defined with the location path:

```
pyyolo.init(darknet_path, datacfg, cfgfile, weightfile)
```

Initialize the OpenCV face and smile detection networks, like following:

```
smile_cascade = cv2.CascadeClassifier('/usr/share/OpenCV/haarcascades/
  haarcascade_smile.xml')
```

Further, the ZED camera is configured and opened, as shown in lines 47-56 in behaviour-detection.py in section A.1.

5.7.2 Main Loop

The repeated work done in the script is placed inside a while loop:

```
while not rospy.is_shutdown():
```

Check if a new image is available from the ZED camera:

```
    if zed.grab(runtime_parameters) == sl.ERROR_CODE.SUCCESS:
```

Retrieve the image:

```
        zed.retrieve_image(image, sl.VIEW.LEFT)
```

Further, in lines 68-76 shown in section A.1, the point cloud is retrieved, and the data is prepared to feed through the detection network.
Detect objects in the image using pyyolo:

```
        outputs = pyyolo.detect(width, height, 4, Data, 0.5, 0.8)
```

Extract the detected people from the pyyolo detections:

```
        for output in outputs:
          if output['class'] == 'person':#track only people
            count = count+1# Count detected people
            dets = np.append(dets, [[output['left'], output['top'], output['
    right'], output['bottom'], output['prob']]], axis=0)
```

Update the tracking ID of the detected people using SORT:

```
        trackers = mot_tracker.update(dets)
```

The detected people BB and ID info is now stored in the ”trackers” variable. Further, the prediction about each detected person is achieved by obtaining the individual detection info in a for-loop:

```
        for d in trackers:
```

A dictionary is created with the detected person info to order the data:

```
          detectinfo = {'left': d[0], 'top': d[1], 'right': d[2], 'bottom':
    d[3], 'class': 'person', 'ID': int(d[4])}
```

Further in lines 97-100 in the script shown in section A.1, the detected person straight line distance, relative coordinates and face/smile indication is predicted, by sending the ”detectinfo” variable through the corresponding function blocks, presented in sections 5.7.3, 5.7.4 and 5.7.6.
The custom ROS message type ”Prediction.msg” is created as an object called ”pred”:

```
          pred = Prediction()
```

The predicted info about the detected person is then assigned to the message objects attributes:

```
          pred.xmin                = detectinfo['left']
```

The prediction message about one person is then added to an array which includes the info about all detected people in one image:

```
          preds.predictions.append(pred)
```

After the for loop is done evaluating all the detected people in one image, the array of predicted info about all the detected people is published to ROS Topics:

```
      pub.publish(preds)
```

5.7.3 Straight line distance to Object - euclidean_distance

Lines 156-160 in the script shown in section A.1 calculates the center of the detected objects BB, which coordinates is saved in variables (x, y). The XYZ-coordinates, corresponding to the center of the objects BB, is then attained from the point cloud:

```
  err, point_cloud_value = point_cloud.get_value(x, y)
```

The euclidean distance to the object is then calculated using these XYZ-coordinates:

```
  distance = math.sqrt(point_cloud_value[0] * point_cloud_value[0] +
    point_cloud_value[1] * point_cloud_value[1] + point_cloud_value[2] *
    point_cloud_value[2])
```

5.7.4 Relative Coordinate Calculation - relative_coordinates

Since this function is all straight forward math, the author refers to lines 155-165 in the behaviourdetection.py script in section A.1 for implementation instructions.

5.7.5 Multiple Object Tracking - Basic SORT

First, check if there exist any tracked objects in the 2 previous frames, if not give the detected person a new ID:

```
  if prev_outputs and prev_prev_outputs == []:
    output['ID'] = randID
    randID += 1
```

Else, compare the detected person's BB with all the BBs from the last frame in a for loop:

```
    for prev_output in prev_outputs:
```

Calculate the distance vector components between the corresponding BB coordinates, like following:

```
      a = output['right']-prev_output['right']
```

Calculate the Euclidean distance of the resulting distance measure vector:

```
      dist = math.sqrt(a*a + b*b + c*c + d*d)
```

Check if the distance measure is below the threshold condition and if it is the smallest of all the BB pairs, and assign the same ID if so:

```
      if dist < threshold*area:
        if dist < minval:#chooses the closest BB, not the last BB under
    the threshold
          output['ID'] = prev_output['ID']
          minval = dist
```

If no ID is found to satisfy the above conditions, the same evaluation is repeated on the tracked people from the 2nd previous frame. If still no match is found, the detected person is given a new ID.

5.7.6 Face and Smile Detection - facesmile_detect

First the upper half of the detected person BB is cut out from the full image:

```
gray_body = gray_picture[detectinfo['top']:detectinfo['bottom']-int((
  detectinfo['bottom']-detectinfo['top'])/2.0), detectinfo['left']:
  detectinfo['right']] # cut the top half gray body frame out
```

Detect faces in the upper half body BB:

```
faces = face_cascade.detectMultiScale(gray_body, 2, 5) #1.3, 5)
```

Obtain the coordinates of the detected faces BBs:

```
for (x,y,w,h) in faces:
```

Cut out the face BB from the upper body image:

```
  gray_face = gray_body[y:y+h, x:x+w]
```

Detect smiles in the cut-out face image:

```
  smiles = smile_cascade.detectMultiScale(gray_face)
```

Obtain the coordinates of the detected smiles BB:

```
  for (sx,sy,sw,sh) in smiles:
```

Then, check if the detected smile is in the lower half of the face image. If so, a smile is predicted:

```
    if sy < h/2:
      pass
    else:
      detectinfo['smile'] = 'yes'
```

5.8 Coding subscribertest.py

The script is initialized as a ROS Node:

```
rospy.init_node('videosubscriber', anonymous=True)
```

The program is then initialized to subscribe to the Topics ”videostream” and ”predictions”, like following:

```
rospy.Subscriber('/videostream', Image, callback, queue_size=10)
```

Instead of a while-loop, the following line is added to prevent the program from exiting:

```
rospy.spin()
```

The subscribed data is passed into two callback functions, which extract and formats the data before printing or visualizing. Following callback function shows how to extract the image from the Topic ”videostream”, then visualizing it:

```
def callback(data):
  image = np.fromstring(data.data, np.uint8)
  image = image.reshape((720, 1280, 4))
  print image.shape
  cv2.imshow("stream", image)
  cv2.waitKey(35)
```

Following callback function shows how to extract the ID and corresponding smile indication from the Topic ”predictions”, then printing the info in the terminal:

```
def callback1(data):
  for person in data.predictions:
    print('Person with ID: %d, has %s smile, and is located at (x=%f,y=%f)
     in mm relative to me.' %(person.id, person.smile, person.xcoord,
    person.ycoord))
```

This is valuable information needed for integrating the behaviourdetection.py output with other Cyborg modules.

Chapter 6

Results

In this chapter the test results of the implemented system are presented. Each of the tests performed in this chapter has the purpose of presenting the performance of the final system with respect to the defined system requirements in Table 3.1. In the object detection result section 6.2 and object tracking result section 6.3, more than one implementation is tested, from which the best solution is discussed and selected for the final system.

6.1 Relative Coordinates Test

In this section, the detected objects relative coordinate calculation is tested and a screenshot of the result is shown below in Figure 6.1. The result in Figure 6.1 is captured from the system during development, and therefore, it does represent the exact final system. However, the system presented uses the exact same algorithm for calculating relative coordinates, as the final system.

As seen in Figure 6.1, the distance in mm is written over the detected people in the visualization. This distance can be used to find the corresponding prediction information printed in the terminal window above the visualization in Figure 6.1. When just evaluating the printed distances, angles and coordinates, and comparing them to the image, the calculations show promising results. The person to the left in the image has a predicted negative angle, while the person to the right has a positive angle. The person to the left has a predicted distance of 1961mm, while the person to the right has a distance of 894mm. All the predictions seem to be correct when compared with the image.
To further validate the relative coordinate calculation, the predictions in Figure 6.1 are plotted in a coordinate system, as shown in Figure 6.2.

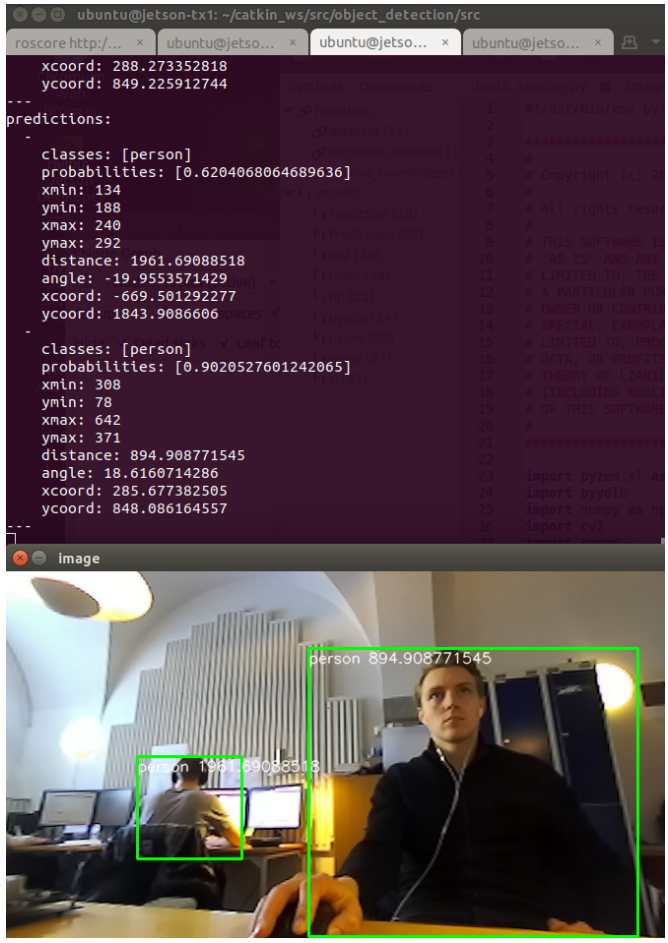


Figure 6.1: System published predictions, and their corresponding visual BB on image.

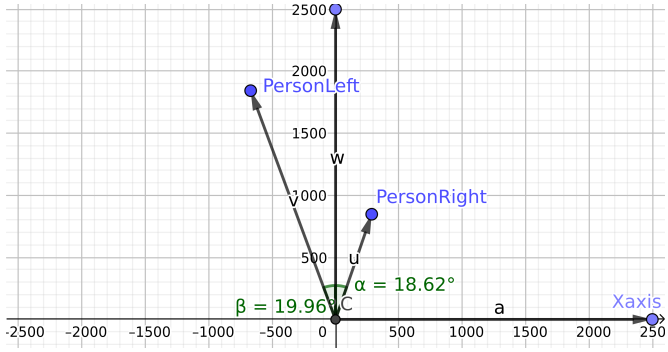


Figure 6.2: The predictions in Figure 6.1 plotted in a 2D coordinate system.

Plotting the calculated coordinates, as shown in Figure 6.2, gives a stronger indication that the predictions are correct when compared to the image in Figure 6.1. If one imagines the camera is positioned at the origin of the 2D coordinate system, pointed towards the y-axis, the resulting points in the plot appears to correspond to the two people visible in the image in Figure 6.1. Moreover, the plot shows how the camera body frame x- and y-axis are defined, which is necessary information when using the CV module output in the other Cyborg modules, such as the Navigation module.

6.2 Object Detection Performance Tests

The object detection performance is tested on the accuracy and speed, for the different YOLO configurations. Two separate tests are performed; the first test, using a pyyolo version from 2018, and the second test, using a pyyolo version from 2020.
The test strategy is to initialize YOLO with different configuration files, adjust network input size, and use the corresponding pre-trained weights while evaluating the trade-off between speed and accuracy of the detection.
The speed is timed exactly using the difference between the recorded time before and after the detection call for each frame, while the accuracy is visually evaluated by the author moving in front of the camera. As a result, no exact numerical value is recorded for evaluating the accuracy of the detection. Instead, the accuracy is evaluated by how often the program fails to detect an object appearing in the image(false negative), and how often it detects objects not appearing in the image(false positive). ”Very robust” means no false negatives and no false positives.

6.2.1 Test 1 - 2018 Version pyyolo

In addition to the VGA resolution bound, the 2018 pyyolo version is not compatible with the third generation of YOLO models ”YOLOv3”. Also, some of the configurations like the tiny-yolo model, using the standard 416x416 resolution, shows no detection results. After some testing, with the ZED resolution set to VGA, two configurations presents promising results, as shown in Table 6.1 below:

YOLO Configuration File	Network Input Size	Detection Time	Accuracy
yolov2.cfg	288x288	190ms	Robust
tiny-yolo.cgf	288x288	100ms	Not robust

Table 6.1: The 2018 pyyolo version performance with respect to configuration.

As shown in Table 6.1, using tiny-yolo configured with a network input size of 288x288, manages the fastest detection time. However, when objects move, the detection becomes unstable. It sometimes fails to detect even with still objects. This configuration is, as a result, rejected for the final system.
The other configuration, shown in Table 6.1, yolov2 configured with an input size of

288x288, detects 90ms faster than the original 416x416 configuration, while still maintaining robust detections.

6.2.2 Test 2 - 2020 Version pyyolo

The 2020 pyyolo version is compatible with all the available YOLO configurations. It also allows HD720 resolution images as input. However, since all the tested configurations resize the images to approximately the VGA resolution (672x376), the ZED resolution is still set to VGA for this test. The newest configurations, YOLOv3, are tested and compared to the older, YOLO and YOLOv2. The most interesting results are presented in Table 6.2 below:

YOLO Configuration File	Network Input Size	Detection Time	Accuracy
yolov3.cfg	416x416	410ms	Very robust
yolov3.cfg	320x320	280ms	Very robust
yolov3-tiny.cfg	416x416	70ms	Robust
yolov3-tiny.cfg	288x288	50ms	Robust
yolov2.cfg	416x416	200ms	Robust
yolov2.cfg	288x288	155ms	Robust
yolov2-tiny.cfg	416x416	67ms	Robust but some false positives
yolov2-tiny.cfg	288x288	45ms	Robust but some false positives

Table 6.2: The 2020 pyyolo version performance with respect to configuration.

Several interesting results can be read from Table 6.2. The first thing noticed are the YOLOv2 configurations all result in at least 45ms faster detection time, compared to the 2018 pyyolo version, shown in Table 6.1. A theory is that either, the 2020 pyyolo includes a newer and more effective Darknet version, or the 2020 pyyolo version is built configured with the proper GPU architecture, while the 2018 pyyolo is not. Building with the correct GPU architecture is described in subsection 5.5.1 in chapter 5.
The second thing noticed in Table 6.2, is the slow detection time of the ”yolov3.cfg” configurations. These configurations are very accurate, however, way to slow for the requirements for this project. As a result, the standard ”yolov3.cfg” configurations are rejected as an option for the final system.
The third thing noticed in Table 6.2, is that the configuration resulting in the fastest detection time is actually the second generation ”yolov2-tiny.cfg”. The drawback is that the accuracy could be better. The system manages to detect the appearing objects robustly, however, it also detected some objects not appearing in the images.
Considering all the tested configurations, shown in Table 6.2, ”yolov3-tiny.cfg” is evaluated to be the best option for the final system. It resulted in the fastest detection time while still maintaining accurate detection. The 416x416 configuration is chosen over 288x288, because it manages almost the same speed, but is tested to have a slightly better accuracy[58].

6.3 Multiple Object Tracking Performance Tests

The performance of the implemented multi object tracking algorithms, are tested using the data-set ”ETH-Bahnhof” from the MOT(Multiple Object Tracking) Challenge, instead of the images captured from the ZED camera. This data-set is used since it is made for testing object tracking algorithms. Besides, the low placement of the camera capturing a crowded sidewalk is very similar to what the Cyborg will experience when maneuvering through the NTNU campus.

6.3.1 Test 1 - Basic SORT with Static Threshold

First test is using the implementation of Basic SORT with a static threshold, as described in algorithm 1 in subsection 4.8.1. The threshold is tuned to a sufficient value, ”threshold = 80”. The result of testing the Basic SORT algorithm on the ETH-Banhof data-set is shown in Figure 6.3 below:

Figure 6.3: Basic SORT algorithm tracking performance on the ETH-Bahnhof data-set.

For clarity, both the tracking ID number and a corresponding coloured BB is drawn on the visualizations for each detected object. As shown in Figure 6.3, the Basic SORT algorithm manages to track the people close to the camera with an individual ID. The problem occurs when objects appear far away, with smaller BBs, while also being close to each other, as seen in the last image in Figure 6.3. Since the calculated distance between most of the far away, and close together BBs, fulfills the threshold condition, they receive the same ID.

6.3.2 Test 2 - Basic SORT with Dynamic Threshold

The second test is using the implementation of Basic SORT with a dynamic threshold, as described in algorithm 2 in subsection 4.8.1. The threshold is tuned again, since the condition is changed, and a sufficient value is found to be ”threshold = 0.010”. The same test is performed with the updated condition, and the result is shown in Figure 6.4 below:

Figure 6.4: Basic SORT algorithm tracking performance on the ETH-Bahnhof data-set with dynamic threshold.

As shown in the last image in Figure 6.4, the Basic SORT algorithm now manages to assign an individual ID correctly to almost every detected person, independent of how large the BB is.
Yet, the Basic SORT algorithm has some weaknesses. Like most of the object tracking algorithms, it is very dependent on the performance of the object detection model. If the system fails to detect an object for two or more frames, the Basic SORT algorithm loses track of the object. In addition, the algorithm will still sometimes reassign the same ID to different objects if they appear close enough to each other.

6.3.3 Test 3 - SORT

The third test is on the system using an implementation of SORT[45]. The SORT algorithm is tested on the same data-set as Basic SORT, for an easy comparison of performance. The result is shown in Figure 6.5 below:

Figure 6.5: SORT algorithm tracking performance on the ETH-Bahnhof data-set.

As shown in Figure 6.5, SORT never reassigns the same ID to different objects, no matter how close the BBs are. This is an improvement from Basic SORT.
On the other hand, the SORT algorithm has some weaknesses, not noticeable in Figure 6.5. If the system loses detection of a person because of other objects blocking the view, SORT will assign a new ID. Also, like all other tracking algorithms, the SORT performance is dependent on the performance of the object detection model. For example, if YOLO fails to detect the objects correctly, it does not matter how good the tracking algorithm is.
The implementation of SORT is about 20ms slower than Basic SORT. However, since the performance is better, SORT is the selected solution for object tracking in the final system.

6.4 Face and Smile Detection Tests

The performance of the implemented face and smile detection is tested on the accuracy and detection range requirements, defined in Table 3.1. The accuracy is measured by visually evaluating the detected smile BB while a person is moving and standing still. The detection range is measured by recording the maximum distance to the person, while still maintaining good accuracy.
The two measures of performance are highly affected by the resolution of the images retrieved from the ZED camera. As a result, two tests are performed; one with VGA, and one with HD720.

6.4.1 Test 1 - VGA resolution

The result of the face and smile detection test, with ZED camera capturing VGA images, is presented in Figure 6.6 below:

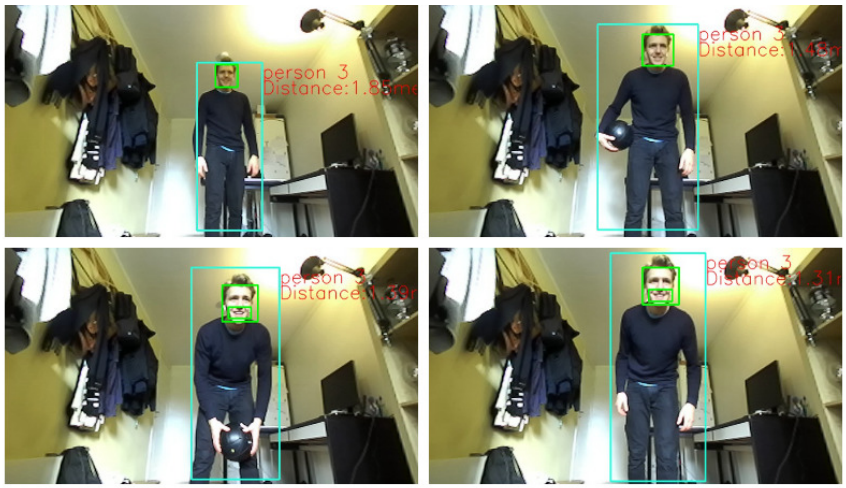


Figure 6.6: Captured result from face and smile detection test, with ZED camera capturing VGA images.

As shown in Figure 6.6, the system configured with VGA, does not manage to detect the smile of a person from a distance more than about 1.4 meters. At a distance of more than 1.5 meters, the system has a hard time even detecting faces. At distances less than 1.4 meters to the person, face and smile detection are both robust, as shown in Figure 6.6. This does not meet the detection range requirement of 2 meters.
On the other hand, the lighting condition during the test is not optimal. As shown in Figure 6.6, the captured scene is affected by varying light intensity, due to the sunlight from the window. This results in some of the areas being overexposed to light, preventing clear details of the captured human features.

6.4.2 Test 2 - HD720 resolution

The result of the face and smile detection test, with ZED camera capturing HD720 images, is presented in Figure 6.7 below:

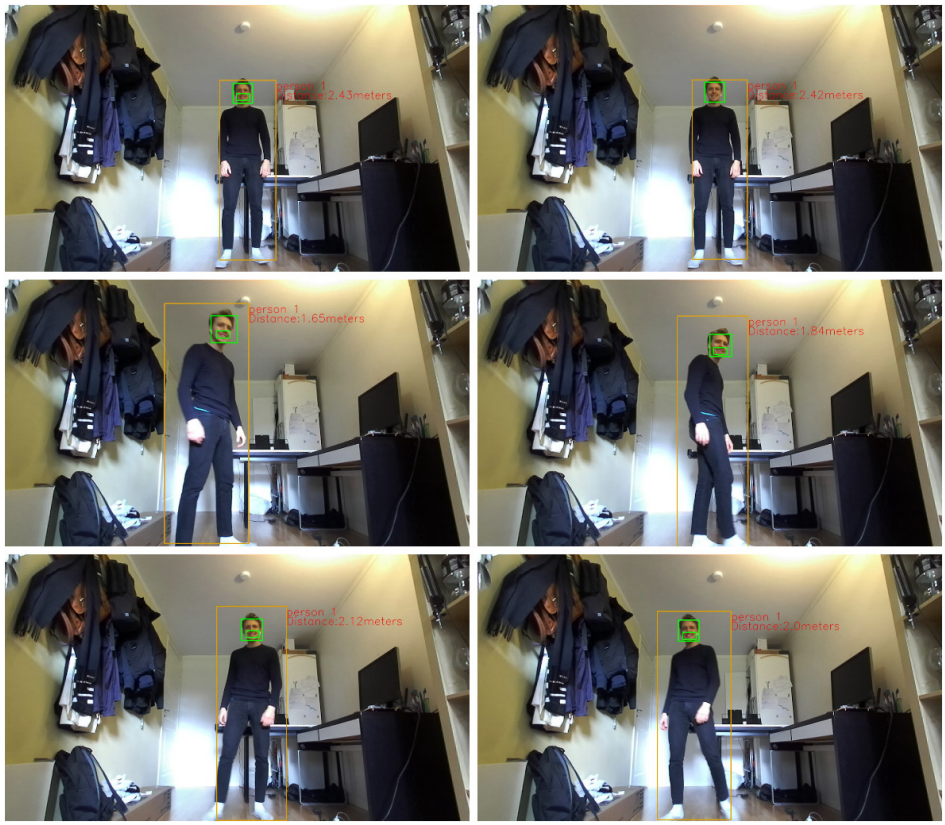


Figure 6.7: Captured result from face and smile detection test, with ZED camera capturing HD720 images.

As shown in Figure 6.7, the system configured with HD720, manages to detect the

smile of a person from a distance up to about 2.4 meters. However, at 2.4 meters the smile detection is not very accurate. At a distance of around 2 meters, the person can move around, while still maintaining good accuracy of smile detection. This satisfies the detection range requirement of 2 meters.

Figure 6.8: Captured result from face and smile detection test, with ZED camera capturing HD720 images.

As shown in Figure 6.8, the face detection was tested to be accurate at the distance of 3 meters, but at this distance no smile is detected. Longer distances than 3 meters were not tested.
Also, experienced when testing is that the person's face needs to be facing the camera with an angle smaller than about 45 degrees, for accurate detection. This is expected since the OpenCV face classifier is not trained to detect the side of a head.

6.5 System Integration Test

The system integration test is set up with the design described in section 4.10, and implemented as described in subsection 5.2.5. The same setup is tested both on a Dell PC running Ubuntu 16.04 and ROS Kinetic, and an Acer PC running Ubuntu 18.04 and ROS Melodic. The test with the Acer PC is presented in this report. The ROS network on the Jetson board is set up to connect to the ROS Master on the Acer. The behaviourdetection.py program is launched from the Jetson board. The subscribertest.py program is launched from the Acer. A picture of the two machines integrated is presented in Figure 6.9.

As seen in Figure 6.9, the subscribertest.py program on the Acer manages to retrieve the published images from the behaviourdetection.py program on the Jetson board. The data published on the "videostream" Topic retrieved via WiFi is delayed with about 4 seconds on the Acer, due to the large HD720 sized images. With the resolution set to VGA, the delay is reduced to about 2 seconds. However, the prediction info published on the Topic "predictions", is delayed with only about 0.5 seconds.

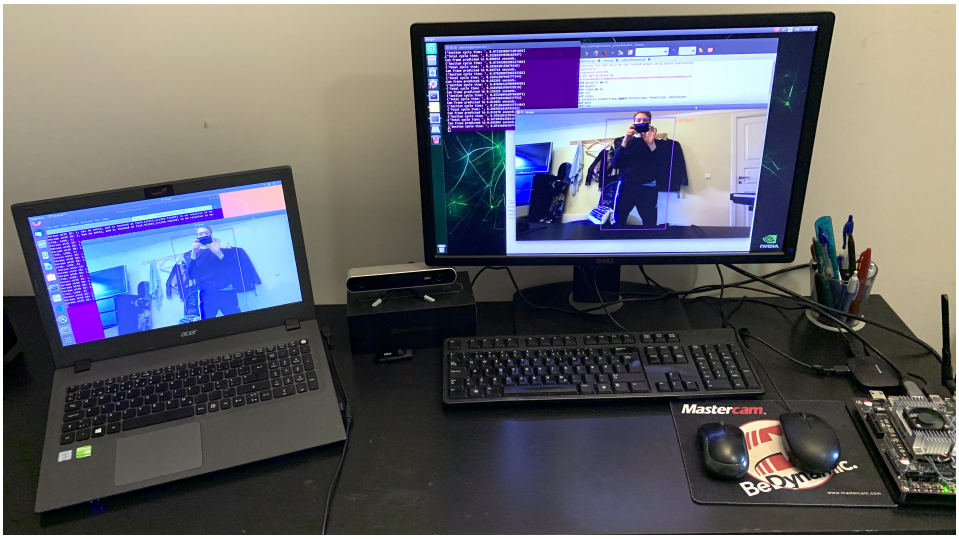


Figure 6.9: System integration test showing two separate machines connected to the same ROS Master.

6.6 Total System Speed Tests

The total system speed test is performed by recording the behaviourdetection.py cycling time, at four different controlled scenarios. Due to the design of the program, the cycle time is affected by the number of detected people, faces and smiles, in each frame. As a result, the cycle time is recorded when the system detects the following four scenarios, shown in Figure 6.10:

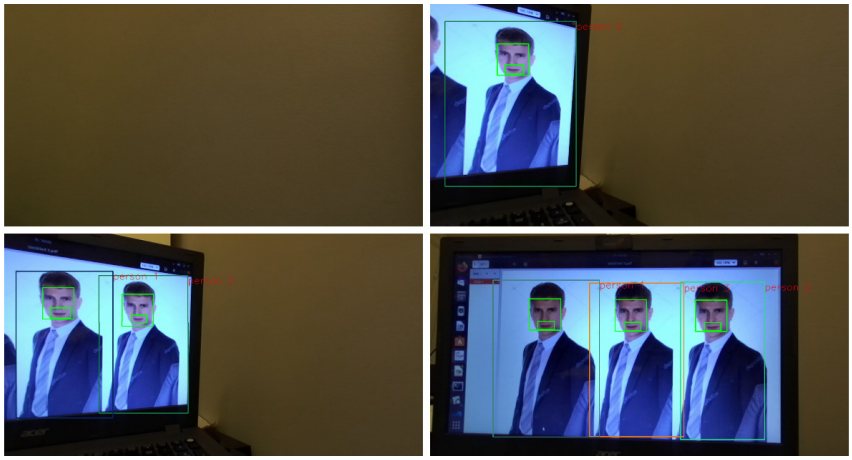


Figure 6.10: The four scenarios where the total cycle time is recorded.

In addition to the number of detections, another factor affecting the total cycling time is image resolution. The face and smile detection test, presented in section 6.4, clearly shows the higher resolution, results in better detection range and accuracy. However, the higher resolution slows the system's total speed down. The question is: What is the best trade-off between total system speed and detection accuracy, affected by the image resolution? To help answer this question, the total system speed test is performed on both the system capturing VGA images and the system capturing HD720 images. The best trade-off is evaluated with respect to the system requirements defined in Table 3.1.

6.6.1 Test 1 - VGA resolution

The first test is performed on the system capturing VGA images. The ZED camera is placed to steadily detect the different numbers of people, faces, and smiles, as shown in Figure 6.10. The system speed is also affected by whether a visualization is printed at run-time. As a result the total system speed is recorded with and without visualization. Without visualization, the steady detection of the different scenarios is found by printing the "predictions" Topic in the terminal.
The total system speed test, capturing VGA images, without visualization, is presented in Table 6.3 below:

Number of People/ Faces/Smiles	Total Cycle Time	FPS
0	100ms	10
1	210ms	4.76
2	225ms	4.44
3	240ms	4.17

Table 6.3: Total system speed test result, capturing VGA images, without visualization.

The total system speed test, capturing VGA images, with visualization, is presented in Table 6.4 below:

Number of People/ Faces/Smiles	Total Cycle Time	FPS
0	130ms	7.69
1	240ms	4.17
2	255ms	3.92
3	270ms	3.70

Table 6.4: Total system speed test result, capturing VGA images, with visualization.

When comparing the cycling times in Table 6.3 and Table 6.4, it is apparent the cycle time is reduced by 30ms when the system is not visualizing the images. The visualization is only necessary when testing and evaluating the detection results. Therefore, the speed without visualization is the result which is evaluated with respect to the system requirements.

As seen in Table 6.3, all the scenarios meets the requirement of 3 FPS, as defined in Table 3.1. Between the scenarios of 0 and 1 detected person, smile and face, the cycle time jumps up a solid 110ms. For every new person, face and smile detected after this, the cycle time increases about 15ms.

6.6.2 Test 2 - HD720 resolution

The same scenarios, shown in Figure 6.10, are tested on the system, capturing HD720 images, without the visualization. The result is presented in Table 6.5 below:

Number of People/ Faces/Smiles	Total Cycle Time	FPS
0	200ms	5
1	280ms	3.57
2	300ms	3.33
3	320ms	3.13

Table 6.5: Total system speed test result, capturing HD720 images, without visualization.

The total system speed test, capturing HD720 images, with visualization, is presented in Table 6.6 below:

Number of People/ Faces/Smiles	Total Cycle Time	FPS
0	230ms	4.37
1	310ms	3.23
2	330ms	3.03
3	350ms	2.86

Table 6.6: Total system speed test result, capturing HD720 images, with visualization.

Similarly to the VGA test, when comparing the results in Table 6.5 and Table 6.6, it is apparent the cycle time is reduced by about 30ms when the system is not visualizing the images.
The HD720 speed results show a similar pattern to the VGA results, regarding the number of detected people, faces, and smiles. From 0 to 1 detected person, smile, and face, the detection time makes a larger jump of 80ms. With each added detected person, face, and smile after 1, the cycle time increases with about 20ms. If this pattern continues, the system without visualization will not meet the speed requirements of 3 FPS, if 4 or more people, faces and smiles are detected in the same frame.

Chapter 7

Discussion

In this chapter the result of the final implemented system is discussed. This includes evaluating the validness of the test results, and whether the system meets the requirements defined in Table 3.1.

7.1 Relative Coordinates

The relative coordinate output test indicates the estimated objects' relative positions are correct. To get an even more reliable result, the detected objects' actual coordinates could be measured with measuring tape and then compared to the calculated coordinates. Instead, the actual coordinates were measured by eye. This suggests the coordinate calculation is tested to be approximately correct, but the exact error margin of the coordinate estimation is not found. Due to the design of the algorithm, the position estimated is the point on the surface of the object corresponding to the center of the detected BB. Depending on the size of the detected object, its actual horizontal relative coordinates are slightly further away than what is estimated.

7.2 Object Detection

The object detection performance is tested to sufficiently produce the output data as described in the requirements. The accuracy performance is described vaguely with phrases such as ”robust” and ”very robust”, which is not a specific measure. This is not conventional, however, assumed to be sufficient when considering the purpose of the test. The goal of the testing presented is not to find the exact YOLO performance measures since this is already well tested [19]. Instead, the goal is to test to find the best YOLO configuration fit for the systems hardware requirements; the ZED camera and the Jetson TX1 board. The best YOLO configuration found for this system is the yolov3-tiny with an input size of 416x416. The input size of 416x416 is preferred over the 288x288 input size because the smaller input size resulted in some false detections. The false positive detections could

possibly be removed by increasing the confidence threshold. Throughout this test the confidence threshold was kept at 0.5. For further testing, it could be an idea to also test with different confidence thresholds.

7.3 Multiple Object Tracking

Since the tracking algorithm performances are dependent only on the correct BBs, and not directly the images retrieved from the ZED camera, the performance is tested on an external dataset. The dataset "ETH-Bahnhof" presents environmental characteristics that are assumed to be similar to the application of the Cyborg. The motivation for the implementation of object tracking is to enable the Cyborg to distinguish between and identify individual people, which can allow more advanced interactions. Judging by the testing, the implementation of SORT tracks people sufficiently enough, to meet the requirements. The implementation of Basic SORT tracked sufficiently the people close to the camera, but failed when distinguishing between the people far away. It can be argued that tracking only close people is acceptable since these are the individuals who are relevant for interactions. Either way, since the SORT algorithm is almost as fast as Basic SORT and also better, SORT is chosen for the final system.

7.4 Face and Smile Detection

The face and smile detection performance is the function most affected by the resolution of the images. A higher resolution gives a better detection range, due to the smaller facial features becoming unclear when further away from the camera. However, a higher resolution also slows the system's speed. As a result, the two relevant ZED resolutions; VGA and HD720, are tested, to find the best fit. From evaluating the first test, it becomes clear that for smile detection to work at an acceptable range, the resolution should be higher than VGA. When VGA is configured the person has to come closer than about 1.3m from the camera, for accurate detection. Closer than 1.3m is not an expected distance from where people will try to interact with the Cyborg.
From evaluating the second test, it becomes clear that the HD720 resolution is definitely preferred over VGA due to the better smile detection range of about 2m. The 2m range is a more expected range from where people will try to interact with the Cyborg. Comparing the distance to the person in Figure 6.6 and Figure 6.7, the one with HD720 resolution shows more natural and expected human behaviour. This is important since the goal for the CV module is to detect natural human behaviour, and not for the person to adapt his behaviour to meet the limitations of the Cyborg. As a result, the HD720 resolution is chosen for the final system, as long as the system is kept within the speed requirements. The test result does not necessarily restrict itself to smile detection performance vs image resolution. The test result could probably also translate to any other detection test of smaller features, such as hand gesture, eye movement and facial emotions in general.
The design of the smile detection algorithm could probably be used for detecting other smaller features. The design prevents the detection network from looking for a smile in the entire image. Instead, it looks where a smile is expected to be; inside a face BB, which

is found inside a body BB. This saves the system a lot of time when looking for a specific feature. Let us use hand gesture detection as an example. A person's hands have a physical limitation on where it is expected to be located since it is attached to the body. A possibility is to restrict the hand gesture detection to a BB which corresponds to the expected reach of the person's arms, estimated from the body BB.

7.5 System Integration

The system integration test is one of the most important tests, concerning the requirements. For the other modules to react based on the detected human behaviour, the published info needs to be available on the Cyborg ROS Topics with minimal delay. The test results prove with high reliability the ROS Networks between two machines are successfully integrated. The system integration test also suggests how to manage the published data by a subscribing Node. The Topic "prediction", containing the detected behaviour info, is delayed by less than 0.5 seconds when retrieved by the subscribing Node. This meets the real-time output requirements. The small delay enables for example the behavioural module on the Cyborg to react to things happening in real-time. The "videostream" Topic, however, is delayed with about 4 seconds. This means that if the GUI module wants to publish the recorded images on the GUI website, the stream will be at least 4 seconds delayed. On the other hand, the delay recorded is strongly affected by the speed of the WiFi connection used. The speed of the WiFi available on the NTNU campus is probably way faster than what is used during the system integration test presented in this report.
Whether the test result is valid for the integration with the actual Cyborg, should be discussed. The test presents the integration of the Jetson TX1 board with an Acer PC running Ubuntu 18.04 and ROS Melodic, and not the actual Cyborg computer. The same method for integration is also tested on a Dell PC running Ubuntu 16.04 and ROS Kinetic, which is not presented in this report. The Cyborg computer is running the same software; Ubuntu 16.04 and ROS Kinetic. As a result, the author suggests the integration test result is valid for integration with the actual Cyborg computer, as well.

7.6 Total System Speed

The total system speed is affected by the resolution of the retrieved images. The smile and face detection test indicated the higher HD720 resolution is required for an acceptable detection range. The main purpose of the speed test is, as a result, to determine if the higher resolution results in an acceptable speed. Both the speed of the VGA and the HD720 configuration is tested. Two main results are drawn from the tests. First result; the system configured with VGA manages to detect up to 6 people, faces, and smiles while meeting the speed requirement. Second result; the system configured with HD720 manages to detect up to 3 people, faces, and smiles while meeting the speed requirements. Since detecting only up to 3 people, faces, and smiles are tested, these measures are estimated based on the pattern of the speed compared to the number of detections. Even when considering the consequence of slower speed caused by the higher resolution, the HD720 resolution is chosen for the final system, since it is essential for the performance of the

face and smile detection.
The implemented system configured with HD720 is expected to quickly exceed the speed requirements, when maneuvering the NTNU campus, assuming the Cyborg will detect more than 3 people. A solution could be to only look for face and smile in detected people BBs which are close to the Cyborg. The Cyborg is not expected to interact with far away people regardless of whether they smile or face the Cyborg. For example, the behaviour-detection system can be modified to only detect the face and smile of the 3 closest people. Since the speed of YOLO is not affected by the number of objects detected in an image, ignoring the face and smile of far away people could keep the speed within the requirements.
Considering the speed test results, and the discussed solution of limiting the face and smile detection the closest 3 people, the final system is evaluated to meet the speed requirements of 3 FPS.
Future students, who may want to implement additional CV functions in the behaviour-detection system, should consider upgrading the hardware in order to still meet the speed requirements. Upgrading from the Jetson TX1 board to the TX2, would probably make the system faster, because of the more powerful GPU and CPU. Before making this decision, it is worth discussing whether the speed requirement of 3 FPS is correct.

7.7 Discussion of Social Intelligence

This thesis focuses on detecting human behaviour to allow the Cyborg to become a socially intelligent robot. Since none of the other Cyborg modules integrated the CV module output, throughout the project duration, the report does not present a test measuring if the Cyborg is actually perceived as a socially intelligent robot. As a result, the CV module is not concluded to successfully make the Cyborg a socially intelligent robot. However, the CV module is tested to detect some human behaviour and emotion, which is an essential part of being perceived as socially intelligent, as described in the Background subsection 2.3.3. As a result, the author suggests it is reasonable to conclude the presented CV module successfully serves as a foundation for completely reaching this goal.

7.8 Discussion of Further Work

It is valuable to discuss how future students could build on the work presented in this thesis. The implemented system is tested to meet the requirements, which the author argues should allow the Cyborg to become a socially intelligent robot. Remaining for future students is to design the behavioural module to react based on the detected information, to actually make the Cyborg act like a socially intelligent robot. The author suggests two different approaches to react based on the detected information:

- The first suggestion is already described in the system requirements chapter 3. This involves specifically programming the Cyborg to react in a certain way when detecting a certain behaviour. An example could be to initiate interaction with a person if the person is standing closer than 2 meters for over 2 seconds while facing the robot. Further, the robot could tell a random joke if the person is smiling.

- The second suggestion is not yet mentioned in the report. The suggestion is to implement a framework that learns good social behaviour from the feedback of detected human behaviour and emotion. For example, the Cyborg could have a set of different jokes, verbal responses, and light patterns to choose from, and learn by trying, what reaction maximizes smiling, laughter, or maybe interest. This sounds very similar to what reinforcement learning algorithms do. In reinforcement learning an agent (the Cyborg) have a set of actions (reactions) to choose from, and the algorithm learns by exploring, to maximize the defined reward (smiles detected) in the environment (real-time images from the Campus) [59]. Most reinforcement learning algorithms are trained in simulators since this is much faster. However, real-time learning should also be possible, it would just learn a lot slower. Reinforcement learning is just a suggestion. Maybe other machine learning algorithms are more effective. Either way, the author believes that robots learning social behaviour from human reactions is worth exploring.

Learning complex social behaviour may require complex detection of facial expressions. The presented CV module only detects the smile on a person. A smile does not always indicate a person is happy. It could also be a nervous smile. With more information about the facial expression, the Cyborg can predict with more confidence the person's actual mood. Motivated by the possibilities for robots learning good social behaviour based on human reactions, a suggestion is to further develop the CV module to detect additional complex facial expressions and behaviour.

Chapter 8

Conclusion

The initial requirements given in the task were to implement a CV system using the Jetson TX1 board and the first generation ZED Stereoscopic camera, which is integrated with the Cyborg robot using ROS. The author further specified the goal to make the system detect human behaviour, which will allow the Cyborg to become a socially intelligent robot. This involves detecting natural human emotions, movement, and intent, which can be used for interaction with surrounding people, where ”natural” is a keyword. A core goal of the project is to detect natural human behaviour, which does not require the person to learn how to interact with the Cyborg. The author further specified the requirements in Table 3.1, in order to meet these goals.

The presented final system test results are throughout the discussion chapter found to satisfy most of the requirements presented in Table 3.1. The CV-module predicts the correct output with a detection range of at least 2m. The system is integrated with ROS and connected to a ROS Master on an external machine, allowing the module to be integrated with the Cyborg. The detection output is retrieved by a subscribing module on an external machine within 0.5 seconds, allowing other Cyborg modules to react, based on the detection output, in real-time. The video-stream, however, does not meet the requirement, with a 4-second delay. As long as the video-stream is not used in any real-time required application, this should be fine. The output frequency, corresponding to the system speed in FPS, meets the requirement of 3Hz, given that 3 or fewer people, faces, and smiles are detected. A modification making the system only detect the face and smile of the 3 closest people is suggested in section 7.6, which in theory should cause the system to always meet the speed requirements. Lastly, the system is implemented on the Jetson TX1 developer board connected with the first generation ZED Stereoscopic camera, which satisfies the hardware requirements.

The system is concluded to satisfy all the essential requirements described in Table 3.1. As a result, the delivered CV-module is further concluded to successfully detect natural human behaviour, laying the foundation for the Cyborg to become a socially intelligent robot.

Bibliography

[1] WHO, "Ageing and health," 2018. [Online]. Available: https://www.who.int/news-room/fact-sheets/detail/ageing-and-health

[2] H. S. Borji, "4 global economic issues of an aging population," 2016. [Online]. Available: https://www.investopedia.com/articles/investing/011216/4-global-economic-issues-aging-population.asp

[3] A. Bauer, "Human-robot collaboration: 3 case studies," 2020. [Online]. Available: https://www.wevolver.com/article/humanrobot.collaboration.3.case.studies

[4] R. Kelley, A. Tavakkoli, C. King, M. Nicolescu, and M. Nicolescu, "Understanding activities and intentions for human-robot interaction," 2010. [Online]. Available: https://www.intechopen.com/books/human-robot-interaction/understanding-activities-and-intentions-for-human-robot-interaction

[5] J. Waløen, "The cyborg v3.0: Finalizing the foundation for an ntnu mascot," 2019.

[6] A. Babayan, "The cyborg v3.0 - finalizing the foundation for an ntnu mascot," 2019.

[7] T. Opheim, A. Moltunmyr, E. Henriksen, and F. Vatsendvik, "EiT - Robotsyn," 2018.

[8] Nvidia, "Unleash your potential with the jetson tx1 developer kit," accessed: 2020-02-21. [Online]. Available: https://developer.nvidia.com/embedded/jetson-tx1-developer-kit

[9] ——, "Jetpack," accessed: 2020-02-21. [Online]. Available: https://developer.nvidia.com/embedded/jetpack

[10] ——, "L4t," accessed: 2020-02-21. [Online]. Available: https://developer.nvidia.com/embedded/linux-tegra

[11] ——, "Cuda gpus," accessed: 2020-02-21. [Online]. Available: https://developer.nvidia.com/cuda-gpus

[12] Stereolabs, "The camera that senses space and motion," accessed: 2020-02-21. [Online]. Available: https://www.stereolabs.com/zed/

[13] Ubuntu, “Ubuntu,” accessed: 2020-02-21. [Online]. Available: https://ubuntu.com/

[14] Nvidia, “L4t system requirements,” accessed: 2020-02-21. [Online]. Available: https://developer.nvidia.com/embedded/linux-tegra

[15] ROS, “About ros,” accessed: 2020-02-21. [Online]. Available: https://www.ros.org/about-ros/

[16] Nvidia, “Cuda toolkit - develop, optimize and deploy gpu-accelerated apps,” accessed: 2020-02-21. [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[17] Stereolabs, “Sdk introduction,” accessed: 2020-02-21. [Online]. Available: https://www.stereolabs.com/docs/api_2.X/index.html

[18] ——, “Stereolabs zed - python api,” accessed: 2020-02-21. [Online]. Available: https://github.com/stereolabs/zed-python-api

[19] J. Redmon and A. Farhadi, “Yolov3: An incremental improvement,” *arXiv*, 2018.

[20] J. Redmon, “Darknet: Open source neural networks in c,” http://pjreddie.com/darknet/, 2013–2016.

[21] A. Kamal, “Yolo, yolov2 and yolov3: All you want to know,” 2019. [Online]. Available: https://medium.com/@amrokamal_47691/yolo-yolov2-and-yolov3-all-you-want-to-know-7e3e92dc4899

[22] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, “You only look once:unified, real-time object detection,” 2016. [Online]. Available: https://pjreddie.com/media/files/papers/yolo_1.pdf

[23] S.-H. Tsang, “Review: Yolov3 — you only look once (object detection),” accessed: 2020-02-21. [Online]. Available: https://towardsdatascience.com/review-yolov3-you-only-look-once-object-detection-eab75d7a1ba6

[24] digitalbrain79, “pyyolo,” 2018. [Online]. Available: https://github.com/digitalbrain79/pyyolo

[25] A. Bewley, Z. Ge, L. Ott, F. Ramos, and B. Upcroft, “Simple online and realtime tracking,” in *2016 IEEE International Conference on Image Processing (ICIP)*, 2016, pp. 3464–3468.

[26] R. Kalman, “New approach to linear filtering andprediction problems,” 1960. [Online]. Available: http://www.unitedthc.com/DSP/Kalman1960.pdf

[27] K. Moore, N. Landman, and J. Khim, “Hungarian maximum matching algorithm,” 2020. [Online]. Available: https://brilliant.org/wiki/hungarian-matching/

[28] OpenCV, “Opencv about,” 2020, accessed: 2020-03-5. [Online]. Available: https://opencv.org/about/

[29] W. Berger, “Deep learning haar cascade explained,” 2018. [Online]. Available: http://www.willberger.org/cascade-haar-explained/

[30] NIST, “Euclidean distance,” accessed: 2020-02-21. [Online]. Available: https://www.itl.nist.gov/div898/software/dataplot/refman2/auxillar/eucldist.htm

[31] J. Kihlstrom and N. Cantor, “Social intelligence,” 2020. [Online]. Available: https://www.ocf.berkeley.edu/~jfkihlstrom/social_intelligence.htm

[32] VOANews, “Pepper the robot,” 2018. [Online]. Available: https://www.youtube.com/watch?v=AG_xxVyMI9I

[33] FinancialTimes, “Pepper the ‘emotional’ robot visits the ft — ft life,” 2016. [Online]. Available: https://www.youtube.com/watch?v=i8bk39a9xM0

[34] Charbax, “Softbank robotics pepper sdk platform for emotional humanoid robot,” 2017. [Online]. Available: https://www.youtube.com/watch?v=BA-fbmA1Cco

[35] SoftBankRoboticsEurope, “Chat about news - nao education,” 2013. [Online]. Available: https://www.youtube.com/watch?v=NURFYWlyC24

[36] SoftBankRobotics, “About softbank robotics,” 2020. [Online]. Available: https://www.softbankrobotics.com/emea/en/company

[37] Wikipedia, “Rgba color model,” 2020, accessed: 2020-02-21. [Online]. Available: https://en.wikipedia.org/wiki/RGBA_color_model

[38] ——, “Human behavior,” accessed: 2020-02-21. [Online]. Available: https://en.wikipedia.org/wiki/Human_behavior

[39] Stereolabs, “Tutorials,” accessed: 2020-02-21. [Online]. Available: https://www.stereolabs.com/docs/tutorials/

[40] OpenRobotics, “Writing a simple publisher and subscriber (python),” accessed: 2020-02-21. [Online]. Available: http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber(python)

[41] C. L. Teo, “Readme on getting rvis to work over multiple computers.” [Online]. Available: http://users.umiacs.umd.edu/~cteo/umd-erratic-ros-data/README-rvis-remote

[42] Stereolabs, “Tutorial 3: Depth sensing with the zed,” accessed: 2020-02-21. [Online]. Available: https://github.com/stereolabs/zed-examples/tree/master/tutorials/tutorial3-depthsensing/python/

[43] F. Vatsendvik, “thentnucyborg/zedyolo: Used for object (person) detection.” accessed: 2020-02-21. [Online]. Available: https://github.com/thentnucyborg/zedyolo

[44] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick, and P. Dollár, “Microsoft coco: Common objects in context,” 2015. [Online]. Available: https://arxiv.org/pdf/1405.0312.pdf

[45] A. Bewley, “Sort,” https://github.com/abewley/sort, 2018.

[46] MOTchallenge, "2d mot 2015 results," 2015. [Online]. Available: https://motchallenge.net/results/2D_MOT_2015/

[47] N. Wojke and A. Bewley, "Deep sort," 2018. [Online]. Available: https://github.com/nwojke/deep_sort

[48] S. Filonov, "Tracking your eyes with python," 2019. [Online]. Available: https://medium.com/@stepanfilonov/tracking-your-eyes-with-python-3952e66194a6

[49] O. Ben-Bassat, "How to setup ros with python 3," 2020, accessed: 2020-03-5. [Online]. Available: https://medium.com/@beta_b0t/how-to-setup-ros-with-python-3-44a69ca36674

[50] NVIDIA, "Install jetson software with sdk manager," 2020. [Online]. Available: https://docs.nvidia.com/sdk-manager/install-with-sdkm-jetson/index.html

[51] M. Stonebank, "Unix tutorial for beginners," 2001. [Online]. Available: http://www.ee.surrey.ac.uk/Teaching/Unix/

[52] OpenRobotics, "Ubuntu install of ros melodic," 2020. [Online]. Available: http://wiki.ros.org/melodic/Installation/Ubuntu

[53] ——, "Ros tutorials," 2020. [Online]. Available: http://wiki.ros.org/ROS/Tutorials

[54] V. Mazzari, "Ros vs ros2," 2019, accessed: 2020-03-5. [Online]. Available: https://www.generationrobots.com/blog/en/ros-vs-ros2/

[55] T. Foote, "sensor_msgs." [Online]. Available: http://wiki.ros.org/sensor_msgs

[56] Stereolabs, "How to install zed sdk on nvidia jetson," 2020. [Online]. Available: https://www.stereolabs.com/docs/installation/jetson/

[57] TheLinuxInformationProject, "Path definition," 2007. [Online]. Available: http://www.linfo.org/path_env_var.html

[58] J. Jung, "Demo nr4: Yolov3," 2020. [Online]. Available: https://github.com/jkjung-avt/tensorrt_demos

[59] S. Perera, "An introduction to reinforcement learning," 2019. [Online]. Available: https://towardsdatascience.com/an-introduction-to-reinforcement-learning-1e7825c60bbe

Appendix A

Python Code

This section contains the Python scripts created in the project.

A.1 behaviourdetection.py

```
#!/usr/bin/env python
################################################################################
# Cyborg 2020, CV module.
# Tracking detected people using Sort and Yolo.
# Detecting face and smile on the tracked people using OpenCV haarcascade
# classifiers.
# Integrated with ROS.
# By Ole Martin Brokstad.
################################################################################
import pyzed.sl as sl
import pyyolo
import numpy as np
import cv2
import rospy
from jetsontx1_cvmodule.msg import Predictions, Prediction, Peoplecount
import math
import time
from sort import *
from sensor_msgs.msg import Image
from cv_bridge import CvBridge, CvBridgeError

def detector():
  # Initialize publisher ROS node
  pub = rospy.Publisher('predictions', Predictions, queue_size=10)
  pub1 = rospy.Publisher('peoplecount', Peoplecount, queue_size=10)
  pub2 = rospy.Publisher('videostream', Image, queue_size=1)
  rospy.init_node('detector', anonymous=True)
  # Ceate sort object
  mot_tracker = Sort()
  # Define paths for yolo files
```

```
31  darknet_path = '/home/ubuntu/catkin_ws/src/jetsontx1_cvmodule/src/pyyolo
      /darknet' # Only './darknet' is dependent on location of rosrun
      command
32  datacfg = 'cfg/coco.data'
33  cfgfile =  'cfg/yolov3-tiny.cfg'
34  weightfile = '../yolov3-tiny.weights' #'/media/ubuntu/SDcard/yoloWeights
      /yolov2-tiny.weights' this also works but it loads way slower
35  filename = darknet_path + '/data/person.jpg'
36  # Image resolution parameters
37  (width, height) = (1280, 720) # Use (672,376) for VGA and (1280,720) for
       HD720 resolution
38  # Initialize visualization
39  fourcc = cv2.VideoWriter_fourcc(*'MJPG')
40  video = cv2.VideoWriter('predictionstest.avi', fourcc, 10, (width,height
      ))
41  # Initialize pyyolo
42  pyyolo.init(darknet_path, datacfg, cfgfile, weightfile)
43  # Initialize face detector
44  face_cascade = cv2.CascadeClassifier('/usr/share/OpenCV/haarcascades/
      haarcascade_frontalface_default.xml')
45  smile_cascade = cv2.CascadeClassifier('/usr/share/OpenCV/haarcascades/
      haarcascade_smile.xml')
46  # Create a Camera object
47  zed = sl.Camera()
48  # Create a InitParameters object and set configuration parameters
49  init_params = sl.InitParameters()
50  init_params.camera_resolution = sl.RESOLUTION.HD720  # Use HD1080, HD720
       or VGA video mode
51  init_params.camera_fps = 15  # Set fps at 30
52  # Open the camera
53  err = zed.open(init_params)
54  if err != sl.ERROR_CODE.SUCCESS:
55    exit(1)
56  zed.set_camera_settings(sl.VIDEO_SETTINGS.EXPOSURE, 50)
57  image = sl.Mat()
58  point_cloud = sl.Mat()
59  colours = 255*np.random.rand(32,3) # For drawing different colours on BB
60
61  runtime_parameters = sl.RuntimeParameters()
62  while not rospy.is_shutdown():
63    start = time.time()
64        # Grab an image, a RuntimeParameters object must be given to grab
      ()
65    if zed.grab(runtime_parameters) == sl.ERROR_CODE.SUCCESS:
66      # A new image is available if grab() returns SUCCESS
67      zed.retrieve_image(image, sl.VIEW.LEFT)
68      data = image.get_data()
69      gray_picture = cv2.cvtColor(data, cv2.COLOR_BGR2GRAY)# Make picture
    gray for face/smile detection
70      # Retrieve colored point cloud. Point cloud is aligned on the left
    image.
71      zed.retrieve_measure(point_cloud, sl.MEASURE.XYZRGBA)
72      Data = data.transpose(2,0,1)
73      start5 = time.time()
74      Data = Data.ravel()/255.0
75      end5 = time.time()
76      Data = np.ascontiguousarray(Data, dtype=np.float32)
```

```
    start1 = time.time()
    outputs = pyyolo.detect(width, height, 4, Data, 0.5, 0.8)
    end1 = time.time()
    print("Section cycle time: ", end1 - start1)
    dets = np.empty((0,5), int)
    count = 0
    for output in outputs:
      if output['class'] == 'person':#track only people
        count = count+1# Count detected people
        dets = np.append(dets, [[output['left'], output['top'], output['
right'], output['bottom'], output['prob']]], axis=0)
    people = Peoplecount()
    people.tot_detected_people  = count
    pub1.publish(people)
    preds = Predictions()
    trackers = mot_tracker.update(dets)
    for d in trackers:
      d = d.astype(np.int32)
      # Create a dictionary to store info for publishing
      detectinfo = {'left': d[0], 'top': d[1], 'right': d[2], 'bottom':
d[3], 'class': 'person', 'ID': int(d[4])}

      euclidean_distance(detectinfo, point_cloud)
      relative_coordinates(detectinfo, width)
      start3 = time.time()
      facesmile_detect(detectinfo, gray_picture, data, face_cascade,
smile_cascade)
      end3 = time.time()
      #print("facesmile detect cycle time: ", end3 - start3)
      #print detectinfo['smile']

      pred = Prediction()
      #pred.probabilities.append()
      pred.classes.append(detectinfo['class'])
      pred.xmin               = detectinfo['left']
      pred.ymin               = detectinfo['top']
      pred.xmax               = detectinfo['right']
      pred.ymax               = detectinfo['bottom']
      pred.id          = detectinfo['ID']
      pred.face          = detectinfo['face']
      pred.smile         = detectinfo['smile']
      pred.distance            = detectinfo['distance']
      pred.angle         = detectinfo['angle']
      pred.xcoord        = detectinfo['x']
      pred.ycoord        = detectinfo['y']
      preds.predictions.append(pred)

      label = detectinfo['class'] + " " + str(detectinfo['ID'])

      cv2.rectangle(data, (d[0],d[1]), (d[2],d[3]), (colours[d[4]%32,0],
colours[d[4]%32,1],colours[d[4]%32,2]), 2)
      font = cv2.FONT_HERSHEY_SIMPLEX
      cv2.putText(data, label, (d[2],d[1]+25), font, 1,(0,0,255),1,cv2.
LINE_AA)
    pub.publish(preds)
    msg_frame = CvBridge().cv2_to_imgmsg(data, "8UC4")#BGRA8
    pub2.publish(msg_frame)
```

```
      #video.write(data[:,:,:3])#because data.shape is (376,672,4) and it only supports 3 channels.
      cv2.imshow("image", data)
      cv2.waitKey(35)
    end = time.time()
    print("Total cycle time: ", end - start)
  # Close the camera
  zed.close()


def facesmile_detect(detectinfo, gray_picture, data, face_cascade, smile_cascade):
  detectinfo['face'] = 'no'
  detectinfo['smile'] = 'no'
  gray_body = gray_picture[detectinfo['top']:detectinfo['bottom']-int((detectinfo['bottom']-detectinfo['top'])/2.0), detectinfo['left']:detectinfo['right']] # cut the top half gray body frame out
  faces = face_cascade.detectMultiScale(gray_body, 2, 5) #1.3, 5)
  for (x,y,w,h) in faces:
    detectinfo['face'] = 'yes'
    cv2.rectangle(data, (detectinfo['left']+x,detectinfo['top']+y), (detectinfo['left']+x+w,detectinfo['top']+y+h), (0,255,0), 2)
    gray_face = gray_body[y:y+h, x:x+w]
    smiles = smile_cascade.detectMultiScale(gray_face)
    for (sx,sy,sw,sh) in smiles:
      if sy < h/2:
        pass
      else:
        detectinfo['smile'] = 'yes'
        cv2.rectangle(data, (detectinfo['left']+x+sx,detectinfo['top']+(y+sy)), (detectinfo['left']+x+sx+sw,detectinfo['top']+y+sy+sh), (0,255,0), 2)

def euclidean_distance(detectinfo, point_cloud):
  l = detectinfo['left']
  r = detectinfo['right']
  t = detectinfo['top']
  b = detectinfo['bottom']
  (x, y) = (r-(r-l)//2, b-(b-t)//2)
  detectinfo['center'] = (x, y)

  err, point_cloud_value = point_cloud.get_value(x, y)
  distance = math.sqrt(point_cloud_value[0] * point_cloud_value[0] + point_cloud_value[1] * point_cloud_value[1] + point_cloud_value[2] * point_cloud_value[2])
  detectinfo['distance'] = distance

def relative_coordinates(detectinfo, width):
  c = detectinfo['center']
  d = detectinfo['distance']
  St = d*np.pi/2  # Total lenght of arc at distance from camera
  S = (float(c[0]-width/2)/width)*St  # Length of arc between center of picture and center of object
  angle = S/d  # Angle in radians towards object, relative to center of image, calculated from the piece of the arc and it's distance
  x = np.sin(angle)*d  # x coordinates (left/right) in meters, relative to center of camera
```

```
  y = np.cos(angle)*d  # y coordinates (forwards/backwards) in meters,
    relative to center of camera
  detectinfo["angle"] = (angle/(2*np.pi))*360
  detectinfo["x"] = x
  detectinfo["y"] = y


if __name__ == "__main__":
  try:
    detector()
  except rospy.ROSInterruptException:
    pass
```

A.2 subscribertest.py

```
#!/usr/bin/env python
#############################################################################
# Cyborg 2020, CV-test module.
# Subscribes to the Topics published by behaviourdetection.py.
# Shows how to extract and use the published data.
# By Ole Martin Brokstad.
#############################################################################
import rospy
from sensor_msgs.msg import Image
import numpy as np
import cv2
from jetsontx1_cvmodule.msg import Predictions, Prediction

def callback(data):
  image = np.fromstring(data.data, np.uint8)
  image = image.reshape((720, 1280, 4))
  print image.shape
  cv2.imshow("stream", image)
  cv2.waitKey(35)

def callback1(data):
  for person in data.predictions:
    print('Person with ID: %d, has %s smile, and is located at (x=%f,y=%f)
     in mm relative to me.' %(person.id, person.smile, person.xcoord,
    person.ycoord))

def listener():
  rospy.init_node('videosubscriber', anonymous=True)
  rospy.Subscriber('/videostream', Image, callback, queue_size=10)
  rospy.Subscriber('/predictions', Predictions, callback1, queue_size=10)

  rospy.spin()

if __name__ == '__main__':
  listener()
```

A.3 basic_sort.py

```
#############################################################################
# Cyborg 2020, basic_sort.
```

```
# Tracking detected people by evaluating the relative distance between
# the BBs
# Could be integrated with behaviourdetection.py.
# for this to work with behaviourdetection.py you need to add following
# lines in the initialization:
#   prev_outputs = []
#   prev_prev_outputs = []
#   threshold = 0.010
#   global randID
#   randID = 1
# and following lines after the detections for one frame:
#   prev_prev_outputs = prev_outputs
#   prev_outputs = outputs #save previous bounding boxes
# By Ole Martin Brokstad.
##############################################################################
import math
def basic_sort(prev_outputs,prev_prev_outputs,output,threshold,randID):
  minval = 1000000
  area = (output['right']-output['left'])*(output['bottom']-output['top'])
  print area*0.025
  if prev_outputs and prev_prev_outputs == []:
    output['ID'] = randID
    randID += 1
  else:
    for prev_output in prev_outputs:
      a = output['right']-prev_output['right']
      b = output['left']-prev_output['left']
      c = output['top']-prev_output['top']
      d = output['bottom']-prev_output['bottom']
      dist = math.sqrt(a*a + b*b + c*c + d*d)
      if dist < threshold*area:
        if dist < minval:#chooses the closest BB, not the last BB under
    the threshold
          output['ID'] = prev_output['ID']
          minval = dist
    if 'ID' not in output:
      for prev_prev_output in prev_prev_outputs:
        a = output['right']-prev_prev_output['right']
        b = output['left']-prev_prev_output['left']
        c = output['top']-prev_prev_output['top']
        d = output['bottom']-prev_prev_output['bottom']
        dist = math.sqrt(a*a + b*b + c*c + d*d)
        if dist < threshold*area:
          if dist < minval:#chooses the closest BB, not the last BB under
    the threshold
            output['ID'] = prev_prev_output['ID']
            minval = dist
    if 'ID' not in output:
      output['ID'] = randID
      randID += 1
```

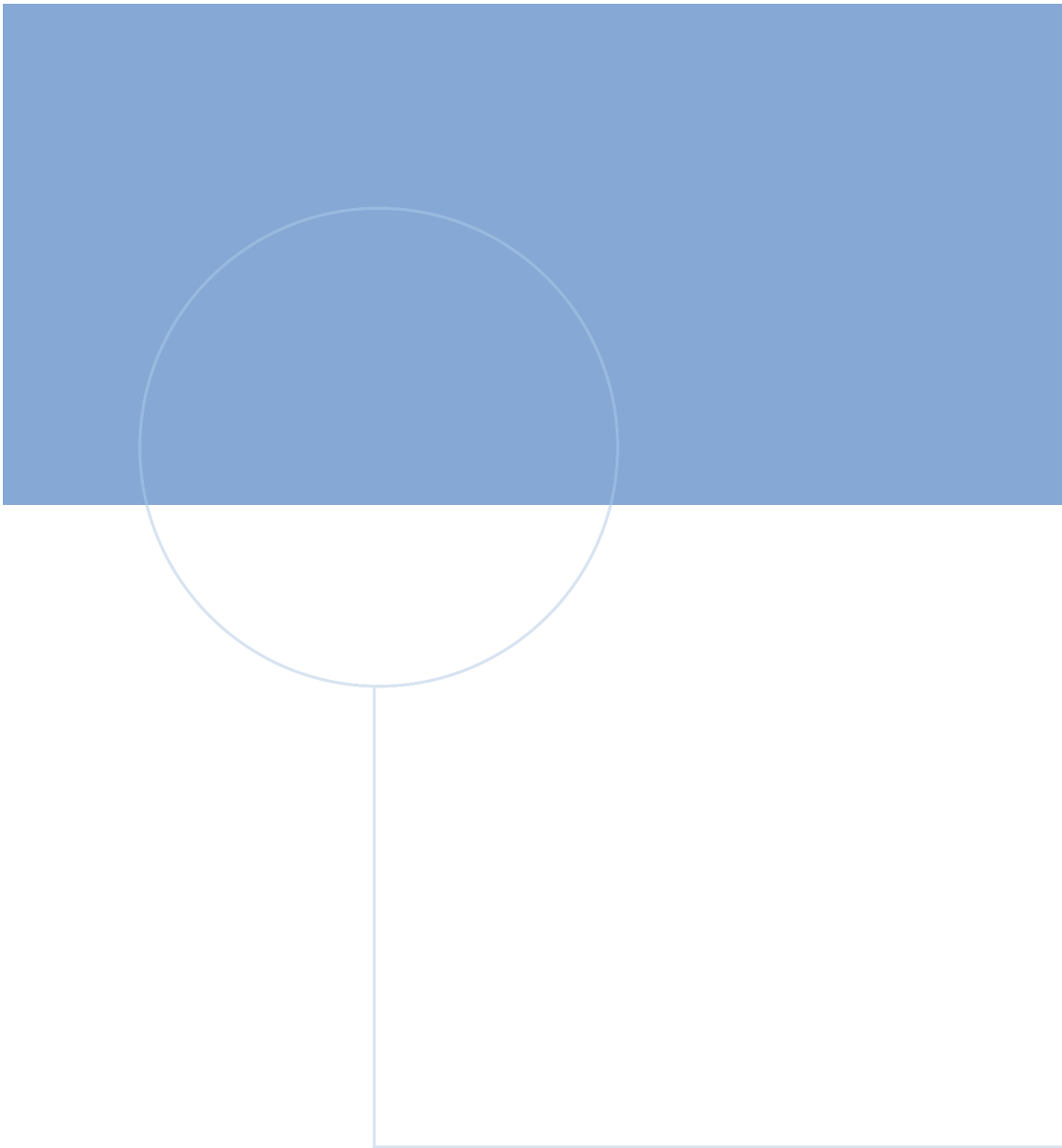
Appendix B

Video Attachments

This appendix chapter contains a description of the attached videos.

B.1 examplevid.avi

Included in the delivery is a video called ”**examplevid.avi**” recorded from a smile detection test. The purpose of including the video is to present the robustness of the behaviour-detection system, which is easier to notice in a video.



NTNU

Kunnskap for en bedre verden