NTNU

Norwegian University of
Science and Technology

Control System and Object Detection System for the NTNU Cyborg

Morten Berg Mjelva

Master of Science in Cybernetics and Robotics
Submission date: June 2018
Supervisor: Sverre Hendseth, ITK

Norwegian University of Science and Technology
Department of Engineering Cybernetics

Norwegian University of Science and Technology

Masters Thesis

Control System and Object Detection System for the NTNU Cyborg

---

Author:
Morten Mjelva

Supervisor:
Dr. Sverre Hendseth
Assistant supervisor:
Martinius Knudsen

A thesis submitted in fulfillment of the requirements for the degree of Master of Technology

in the

Department of Engineering Cybernetics

NTNU – Trondheim
Norwegian University of
Science and Technology

Abstract

NTNU Cyborg is developing a cybernetic organism, combining neuroscience and robotics, as a research platform in these fields. This thesis adds incremental improvements to the robotics part of the project. Firstly, by creating a new control system using behavior trees. The aim of this system is to be able to represent more life-like behavior than the current finite state machine control system. Secondly, by creating a software application for visualizing the running behavior tree. This application will make the control system easier to debug. Thirdly, by creating an object detection and classification system that fetches video from the Cyborg's stereoscopic camera and analyzes this using a neural network. The author has also assisted Experts in Teamwork (EiT) groups that are involved with the Cyborg project.

To implement the described functionality, the *Robot Operating System (ROS)* is used, as well as other software from the ROS project, in addition to software from MobileRobots. The library *behavior3* is used for implementing behavior trees. The library *OpenCV* and the YOLO detection system is used for object detection and classification.

The final software architecture for each contribution is presented, as well as a discussion on the strengths and weaknesses of the chosen solution compared to relevant alternatives.

Forord

NTNU Cyborg utvikler en kybernetisk organisme, en kombinasjon av nevrovitenskap og robotikk, som en forskningsplattform innen disse feltene. Denne oppgaven bidrar med inkrementelle forbedringer til robotikkdelen av prosjektet. For det første, ved å lage et nytt kontrollsystem basert på behavior trees. Målet med systemet er at det skal være i stand til å representere mer realistisk oppførsel enn dagens system som baserer seg på tilstandsmaskiner. For det andre, ved å lage en programvareapplikasjon for å visualisere behavior treet mens det kjører. Denne applikasjonen vil gjøre kontrollsystemet lettere å feilsøke, noe som vil gjøre det lettere å implementere avansert oppførsel i det nye systemet. For det tredje, ved å lage et objektdetekterings- og klassifiseringssystem som henter video fra robotens stereokamera, og analyserer disse ved hjelp av et nevralt nettverk.

For å implementere denne funksjonaliteten benyttes *Robot Operating System (ROS)*, i tillegg til annen programvare fra ROS-prosjektet i tillegg til programvare fra MobileRobots. Biblioteket *behavior3* benyttes for å implementere behavior trees. Biblioteket *OpenCV* og detekteringssystemet YOLO til objektdetektering og klassifisering.

Den endelige programvarearkitekturen for hvert bidrag blir presentert, sammen med drøfting av fordelene og ulempene med løsningene som ble valgt sammenliknet med relevante alternativ.

Acknowledgements

I would like to thank my co-advisor Martinius Knudsen for his guidance and my partner, Katharina Fechtner, for her support and feedback during the work of this thesis. Finally, I would like to extend my deepest gratitude to my mother, Birgit Mjelva, for all her support throughout my education.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

NTNU Cyborg is a project to develop a cybernetic organism. The main goal of the project is to enable communication between live nerve tissue and a robot. The robot acts as a research platform for the study of neural signals, robotics and, cybernetic machines. It is the stated intention of the project to bring NTNU to the forefront of international research in these research areas.

This thesis contributes to this goal by implementing the following functionalities in the Cyborg:

- A robust and extensible control system, based on behavior trees
- An application for visualizing the control system during execution
- An object detection and classification system using stereoscopic imagery and neural networks

In the following, the two main research areas for which the Cyborg serves as a platform are introduced, and the existing functionalities are summarized.

1.0.1 Neuroscience

The Cyborg has a biological "brain" which consists of a collection of nerve cells grown over a Micro-Electrode Array (MEA). These spontaneously organize into neural networks, and communicate with each other using electronic impulses. The impulses are captured by the MEA, which allows for streaming of nerve activity data to the Cyborg. The long-term intent of the neuroscientific part of the Cyborg project is to analyze the electrical input and stimulate the network, in order to form a closed-loop system which allows the network to be trained. Further information on this research can be found in [1].

1.0.2 Robotics

The Cyborg has an autonomous robotic body, consisting of a commercial research robot as well as additions designed at NTNU [2]. The intent of the robotics part of the Cyborg project is to create a robot able to roam the hallways of the university, and interact with the people it meets. The robot is intended to receive data from the MEA sensors described above, and display a representation of the measured neural activity.

1.1 Previous work on the Cyborg project

Previously, student groups have added a rudimentary body for the Cyborg, upon which the existing LED box is mounted. This acts as a mounting frame for existing equipment, including the power supply, Jetson computer, Raspberry Pi computer and so on. There is a stereo camera mounted on the Cyborg, although it is currently not in use. Work has been done to pass information from the Micro-Electrode Array to the existing LED box, but this is not in use at the outset of this thesis.

1.2 Contributions of the author

This thesis aims to enhance the Cyborg's control system, based on the *Robot Operating System (ROS)*, by implementing an alternative to traditional state machines called a *behavior tree*. This involves integrating a behavior tree implementation, ideally one that already exists and is well tested, with the ROS architecture. Furthermore, it involves developing computer software for visualization of the behavior tree so that the state of the behavior tree can be inspected during Cyborg operation. The mounting frame that is already on the Cyborg is improved by installing industry-standard 10 inch rack-mounting rails to the frame. This allows for secure, modular integration of hardware within the Cyborg body. Finally, imagery from the stereo camera is analyzed to detect objects in the Cyborg's view field, such as humans and other everyday objects. These are classified by a neural network, and the distance to each object is calculated so their location relative to the Cyborg can be determined.

The goal of these tasks is to take the Cyborg a few steps closer to a state where it is able to autonomously roam the hallways of the university and act as a mascot for the university and the Cyborg project.

Several existing software applications are used to achieve these goals, and will be described in more detail in the following chapter. The software used by the author is primarily the Robot Operating System (ROS). The ros software is run on the Cyborg's *Pioneer LX* base, produced by *MobileRobots*. ROS is a middleware system, composed by a number of modules, for the sake of this thesis primarily those delivered by MobileRobots. The ROS modules, or nodes, developed by the Cyborg project and by the author interface with these existing modules. A library called *behavior3*, for implementing behavior trees, has been used to add a control system to our ROS environment. Visualization of our behavior tree is developed as a module, or plugin, which extends the *rqt* software, a graphical interface for ROS. Object detection is implemented using the *OpenCV* library.

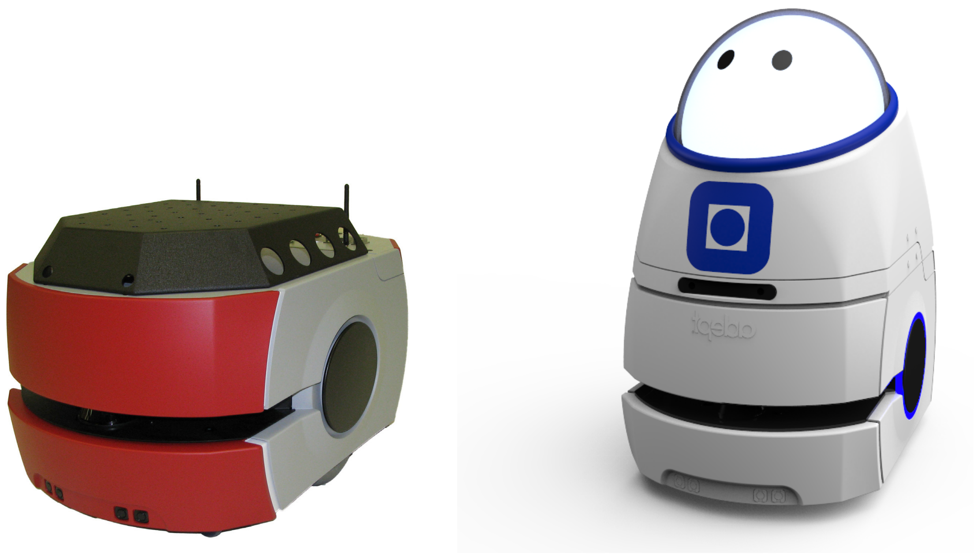
2 Background

The Cyborg is composed of a number of major and minor hardware and software components. In the following sections, these will be described in order to provide the necessary background for the subsequent steps undertaken in this thesis to implement the previously described functionalities.

2.1 Hardware

The Cyborg itself is a MobileRobots Pioneer LX base, with additions made by the Cyborg project. In addition to a custom power supply, the Cyborg has a Stereolabs *ZED* stereo camera, as well as a Nvidia *Jetson TX2* embedded computer for image processing. Here, some detail will be given on each hardware component that is relevant to this thesis.

2.1.1 Pioneer LX



(A) MobileRobots Pioneer LX. (B) NTNU Cyborg.

FIGURE 2.1: Comparison of the original Pioneer LX and the Cyborg.

The NTNU Cyborg uses the Pioneer LX robot, from MobileRobots (fig. 2.1a) [3]. Omron Adept MobileRobots, previously MobileRobots, is a manufacturer of intelligent mobile robots for commercial and industrial use.

The Pioneer LX comes equipped with the required sensors for safe, autonomous operation. The robot has a *SICK S300* 270° laser rangefinder for navigation and object detection, in addition to a *SICK TiM 510* laser for frontal sensing near floor level. Additionally, the robot has rear facing ultrasonic sonar sensors and front bumpers for collision detection. These sensors allow for Simultaneous Mapping and Localization (SLAM), which enables the robot to navigate without pre-mapping the environment. It is designed for continuous operation for 13 hours before recharging, which can be performed autonomously.

2.1.2 Stereolabs ZED camera

FIGURE 2.2: Stereolabs ZED camera.

The Stereolabs ZED (fig. 2.2) is a dual 4 megapixel camera. It provides 1920×1080 video at 30 Frames per Second (FPS) frames per second, or 800×400 at 100 FPS, with a 110° angle of view [4].

The camera is designed to emulate the stereoscopic operation of human eyes. By recording an image from two slightly offset points, depth and motion in space can be inferred by comparing the displacement of pixels in the left and right images. This allows for the camera to provide a depth map of the recorded scene, where pixels are defined by an (x, y, z)-tuple rather than the normal (x, y) coordinate pair. The information is exported as a black and white depth map. By identifying the pixels of an object, the distance to the object can be determined by calculating the mean intensity of these pixels in the depth map [4].

2.1.3 Nvidia Jetson TX2

The Nvidia Jetson TX2, shown in fig. 2.3, is an embedded computer from Nvidia, designed for real-time data processing for *artificial intelligence*. The stated purpose of the device is to move data processing from a central location to the edge, meaning on the robot or drone itself. The TX2 has a small form factor of 17 by 17 centimeters, and a power consumption of 7.5 watts under normal load. Despite the low footprint in size and power consumption, the onboard GPU delivers in excess of 1 TFLOPS (10^{12} Floating Point Operations per Second) of computing power [5].

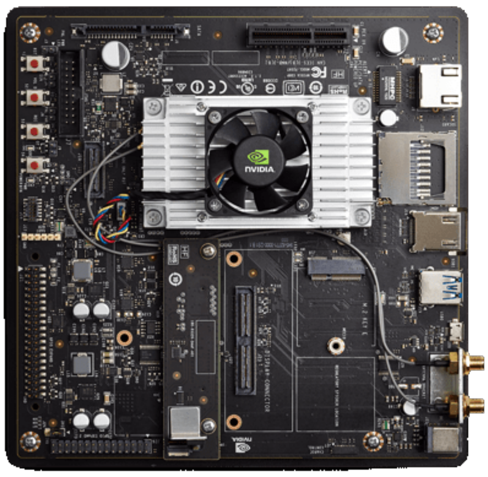


FIGURE 2.3: Nvidia Jetson TX2 development kit, image courtesy of Nvidia.

2.2 Software

The Cyborg's onboard computer comes pre-installed with Ubuntu Linux and ROS. MobileRobots provide a number of software packages for efficient use of the robot. Among these are the Advanced Robot Interface for Applications (ARIA) library, the Advanced Robot Navigation and Localization (ARNL) library and the software applications MobileSim and MobileEyes, which enable implementation of advanced functionality in ROS.

2.2.1 Robot Operating System (ROS)

The Robot Operating System (ROS)[1] is an open-source framework for writing robot control software. It consists of a collection of libraries and tools that simplify the task of implementing complex and robust control software for different robot platforms. It is not an operating system in the traditional sense of managing and scheduling processes but acts as middleware, providing a communications layer for a distributed control system.

[1]https://www.ros.org

The framework provides middleware for individual software modules, known as nodes, so that modules from various sources can interact using a standard interface. These can all be implemented separately and can communicate with each other using the ROS communications layer. This approach simplifies development, by allowing components developed by separate individuals, teams or organizations to communicate using a standardized interface. This promotes collaboration and code reuse, enabling different research teams to focus on a subset of robot control rather than building a complete solution from the ground up.

The following gives a run-down of the core concepts in ROS. The components of ROS are *nodes*, *messages*, *topics* and *services* [6].

Nodes

Nodes are processes that perform computation, and are primarily implemented in Python or C++. Nodes are self-contained processes, and communicate with other nodes by passing messages through the ROS middleware. ROS modules are generally constructed from a number of nodes. This approach follows the Unix philosophy of "do one thing and do it well", which simplifies software development by limiting the scope of each individual node. ROS allows for a control system to be divided into any number of nodes, which may run on the same machine or on different machines [7].

When implementing nodes in C++, it is also possible to implement them as nodelets. A nodelet is implemented similarly to a normal node, but is instantiated by a special node, the *nodelet manager*. This can be done by creating a separate node, which loads all the required nodelets, or it can be done using a launch file. The advantage of running multiple nodelets in one node is that communication is done using shared memory, thereby avoiding the overhead of TCP communication [6].

Messages

The information passed between nodes is encapsulated in messages, which may be thought of as analogous to structs in C. Messages are composed of standard data types such as integers, floats and so on, and arrays of these. A message can also be composed of other messages. Many standard message formats are available in ROS, and it is also possible to define custom message types. Nodes publish messages in one of two ways, either by broadcasting them to a topic or by answering service requests [6].

Topics

Messages can be published to a topic, that can be subscribed to by other nodes. This publish-subscribe model allows for sharing of data in a broadcast manner, and there may be any number of publishers and subscribers for a topic. One example of such a node is the feedback node, from move_base, which publishes the current position of the robot. Any other node that wants to know the current position can subscribe to this, and will receive the new position whenever the robot moves [6].

Services

While the publish/subscribe model works well for many types of information, it is necessarily an asynchronous form of communication. For applications where synchronous communication is required, it is possible to use services. A service is defined by two messages, the input message and the result message. When called, the service will execute and the result is returned to the caller. Services can be used to request information, request the execution of a physical action, or some other task. Listing 2.1 shows one such service, in use by the Cyborg project [6].

Actions

While service calls are useful for remote procedure calls which execute quickly, they are blocking and should be avoided for long-running tasks, or tasks that may have to be preempted [7]. For this purpose, ROS provides actions. Actions are used for procedure calls that cause the robot to perform a long-running task, such as moving to a location or some other real-world action. Actions are able to keep state for the lifetime of a provided goal, and will provide feedback to each client that issues a goal to the server [6].

2.2.2 ARIA

Advanced Robot Interface for Applications (ARIA) is a C++ library for all robots from MobileRobots. The library allows for dynamic control of the robot's velocity, heading, relative heading as well as other parameters. This can be achieved both using a low-level interface and a higher-level Actions infrastructure. ARIA receives sensor data from the robot platform, such as position estimates, sonar data and laser rangefinder data. While written in C++, the ARIA library can be accessed from other languages, amongst others Python [8].

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

from __future__ import print_function

import rospy
from cyborg_nav.srv import DistanceToGoal, DistanceToGoalResponse
from cyborg_types import Path, Pose
from rosarnl.srv import MakePlan

NAME = 'distance_to_goal_server'


class DistanceToGoalHandler():
    def __init__(self):
        rospy.init_node(NAME)

        srv_name = '/cyborg/nav/get_distance_to_goal'
        rospy.Service(srv_name, DistanceToGoal, self.__distance_cb)

        srv_name = '/rosarnl_node/make_plan'
        rospy.wait_for_service(srv_name)
        try:
            self._plan_svc = rospy.ServiceProxy(srv_name, MakePlan)
        except rospy.ServiceException as e:
            rospy.logerr("Service call to %s failed: %s" % (srv_name, e))

        rospy.spin()

    def __distance_cb(self, data):
        # Create a Pose message to validate our input data
        goal = Pose.from_pose(data.goal)

        # Get a path to the given goal
        path = Path.from_posearray(self._plan_svc(goal).path)

        # Calculate distance if a path was found, or return inf
        distance = path.length if path else float('inf')

        return DistanceToGoalResponse(distance=distance)


if __name__ == "__main__":
    DistanceToGoalHandler()
```

Listing 2.1: Example of an ROS service, written by the author.

2.2.3 ARNL

Advanced Robot Navigation and Localization (ARNL) is C++ library from MobileRobots built on top of ARIA which provides intelligent navigation and localization. ARNL provides information about the current position of the robot, and an interface for requesting that the robot move to a given location. The software updates the current position automatically, using data from the robot's sensors and map.

These features are provided in ROS as a node, called *ros-arnl*, which exposes this functionality in the form of topics and services. This node provides a simple interface for higher-level software to monitor and control the position of the robot [9].

2.2.4 MobileEyes

MobileEyes is a graphical interface from MobileRobots for monitoring robot motion and sensor output. The program allows the user to monitor the movement of the robot on the map, and it is possible to send commands to the robot remotely. The software also allows for reconfiguring the robot, and it is possible to send custom commands or create custom overlays that are shown on the map [10].

2.2.5 MobileSim

MobileSim is a simulation software package from MobileRobots, which allows for testing ROS modules in simulation. The software emulates the physical robot so that other parts of the ROS integrate without any necessary changes. This simulation includes data streams from sensors such as the sonar and laser rangefinders, which allows for efficient testing of control software during development [11].

2.2.6 ROS 2

Here, we will give an overview of *ROS 2*[2], and how the project has attempted to remedy some perceived shortcomings in ROS. As outlined in [12], ROS was developed based on the use case of the Willow Garage *PR2* robot. Due to this, development was guided by the characteristics of the PR2 robot, including

- Support for a single robot.
- Significant computational resources available.

[2]http://www.ros2.org/

- No real-time requirements.
- Strong network connectivity available.
- Mainly academic applications.

Since the beginning of ROS in 2007, several of these assumptions have changed, and ROS has been used on a far wider range of robots than for what it was originally designed [12]. Among the new use cases outlined, are

- Control of multiple robots;
  Currently, there is no standard way to control more than a single robot using ROS. ROS has a single-master architecture, and multi-robot support does not elegantly integrate into this design.
- Limited computational resources;
  ROS is not designed to run on micro controllers. Therefore, nodes must interact with these through a device driver. ROS 2 is designed so that these controllers can be implemented as nodes, and thereby participate directly in the control system as first-class citizens.
- Built-in real-time support.
- Non-ideal networks;
  As seen in earlier work on this project by [2], ROS does not degrade gracefully when run on unreliable networks. ROS 2 aims to alleviate this.
- Academic and industrial applications:;
  ROS 2 aims for ROS to remain the platform of choice in academic robotics, while also becoming increasingly relevant in industrial applications.

The central feature of ROS is the publish-subscribe middleware which allows for loose coupling of individual nodes. As ROS was begun in 2007, there was no sufficiently mature off-the-shelf technology that provided this, and this system was built essentially from scratch. The ROS project implemented the necessary framework for node discovery, message definition, serialization and transport. Since 2007, a number of technologies have matured that provide this capability, and it would not have been necessary to build a custom solution today. Several advantages to using one of these off-the-shelf technologies are listed by [12]:

- Less code to be maintained by the project developers
- Third party solutions may offer features outside the scope of what the project could develop themselves

- The project benefits from ongoing improvements to third party solutions
- Third party solutions may be rigorously proven, and thereby improve the perceived reliability of ROS

2.3 Biological neural networks

As a long-term goal, the Cyborg project hopes to use biological neurons in a robot control loop. Here, we will give some background on relevant biological concepts. Further detail on these concepts can be found in [1].

Neurons are electrically excitable cells, which process and transmit information using electrical and chemical signals. The signals travel via synapses, which are specialized connections between neurons. The sum of neurons and their connections are referred to as a *neural network*. They are the core components of the central nervous system, and along with the *ganglia* form the core of the of the peripheral nervous system.

When the *membrane potential* of a neuron is excited past a certain threshold, an *action potential* is triggered. An action potential is a rapid rise and fall in the membrane potential of the neuron, which causes the *axon hillock* to fire an electrical signal down the *axon* of the neuron. The membrane potential of the neuron is excited by the firing of upstream neurons, and other extracellular *potential changes*.

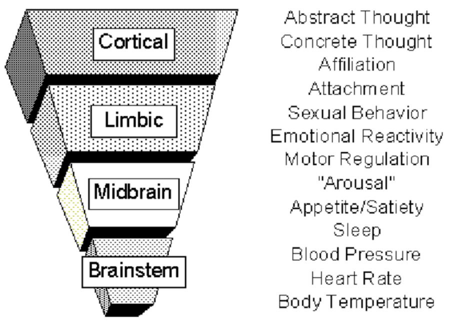


FIGURE 2.5: Model of the hierarchical structure of the brain [13].

A *synapse* is a structure which connects two neurons, and allows for the transmission of an electrical or chemical signal. Synaptic communication generally travels along axons and synaptic *terminals* of the upstream neuron to the *dendrites* of the downstream neuron. Synapses may either transmit electric signals directly, in *electrical synapses*, or by using neurotransmitters, in *chemical synapses*.

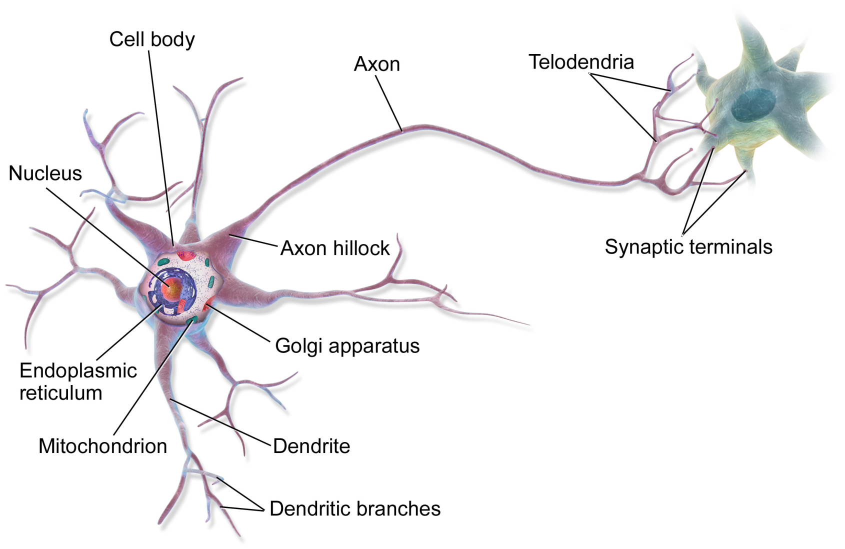


FIGURE 2.4: Model of a biological neuron, courtesy of Blausen Medical.

The neurons in the brain are organized in a hierarchical manner. Signals enter through the *brain stem*, and travel upwards as shown in fig. 2.5. As shown, higher layers of the brain are responsible for increasingly sophisticated levels of thought. This layering is also present in the *neocortex*. Taking vision as an example, lower levels of the neocortex are responsible for simple features such as edges and corners. Low-level patterns are combined at mid-levels into more complex features such as curves and textures. Finally, at higher levels of the neocortex, complex objects such as cars and houses are recognized.

2.4 Artificial neural networks

The Cyborg is intended to move around in its surroundings, and is equipped with a stereo camera in order to orient itself. Here, an overview is given of the techniques used in computer and robot vision, and the research that has driven the explosion of activity within this field in the past few years. This section provides some background on how artificial neural networks relate to robot vision, as well as an overview of how they function in general.

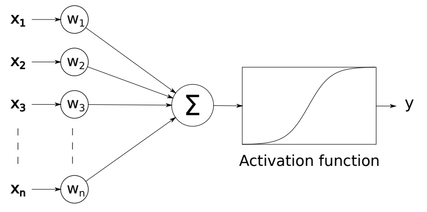


FIGURE 2.6: Model of an artificial neuron.

Artificial neural networks are an approach to perform complex computational tasks as an emergent process of a large number of simple interconnected units. This approach is inspired by the activity of neurons in the brain. The artificial neuron is modelled by an *activation function*, which outputs a value as a function of the sum of its inputs. Historically, this has been one of several possible *sigmoid functions*, for example $g(x) = \frac{1}{1-e^{-x}}$. An illustration of an artificial neuron is shown in fig. 2.6.

Nodes in artificial neural networks are organized into *layers* of units which are connected to the units in the consequent and previous layers. The value of the signal flowing into a node is a function of the value flowing out of the previous layer, and the weights assigned to the particular connections. A neural network with a sufficient number of units and a continuous, bounded and non-constant activation function, is able to approximate any mathematical function [14][15].

A simple example, which approximates the Exclusive Or (XOR) function, is illustrated in fig. 2.7. The weights in this example are determined by construction. In a real scenario, the weights are found by minimizing some *cost function* through the process of *gradient descent*, which is referred to as training the network [16]. Through training, the network would find a different set of weights while still achieving the same output.

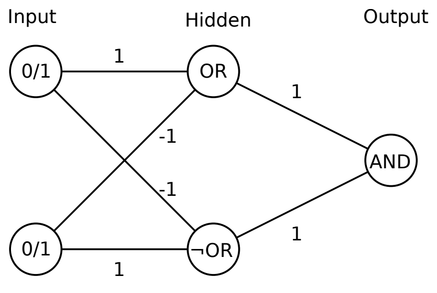


FIGURE 2.7: XOR network, illustrating how neurons can implement basic logic functions.

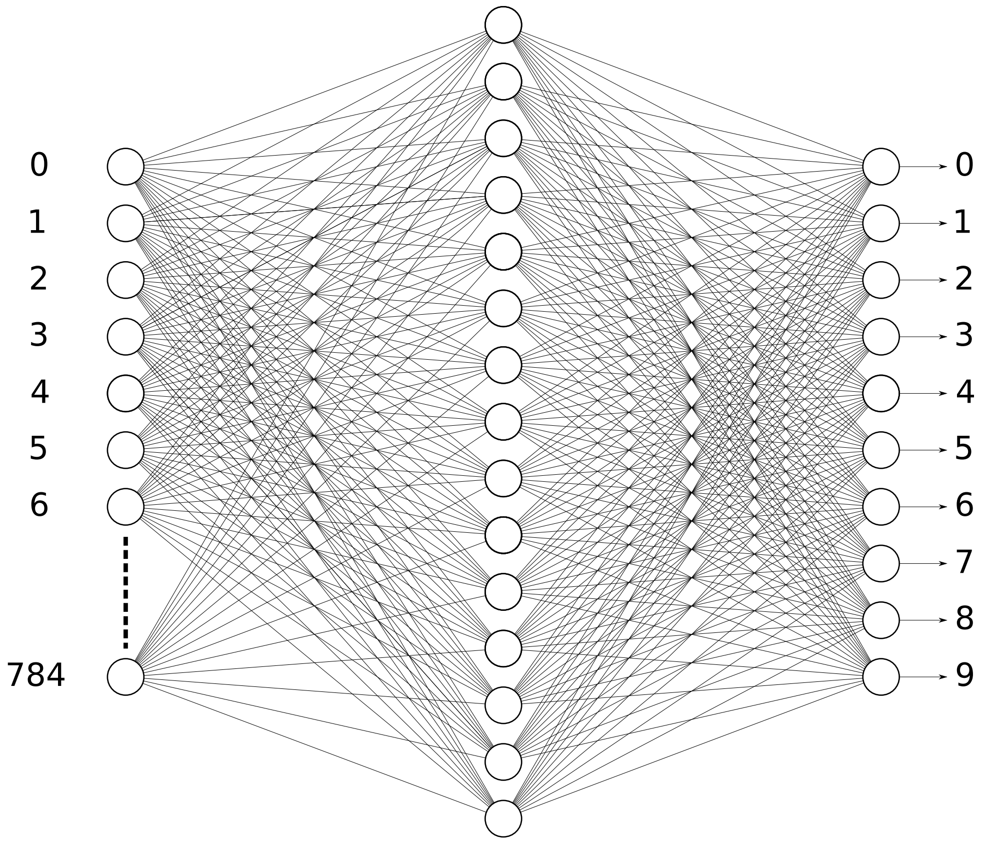


Figure 2.8: An example of a neural network with one fully connected hidden layer. The input to the network is a 28×28 pixel image of a single digit, flattened to a 784 element vector. The output is a confidence score for each of the possible digits. The network was trained by the author, and achieved 92.4% verification accuracy on the MNIST handwritten digit dataset. More complex networks exceed 99% accuracy on this dataset [17].

2.4.1 Convolutional Neural Networks (CNNs)

For a fully connected network as the one shown in fig. 2.8, every neuron in the *hidden layer* is connected to each input. In this case there are 784 connections, one for each pixel in the input image. However, with one input per color channel, per pixel, this number would quickly balloon into the hundreds of thousands for a larger image. Instead of flattening an image to a column vector, as done in the fully connected network shown before, a CNN arranges its neurons in a 3D volume corresponding to the width, height and color channel depth of the input image. Furthermore, each neuron in a hidden layer is connected only to a smaller region of the previous layer, as shown in fig. 2.9. Between *convolutional layers*, *pooling layers* have traditionally been inserted to reduce the spatial size of the network. This decreases the computational complexity of the network, and by lowering the number of parameters also limits *overfitting* [19].

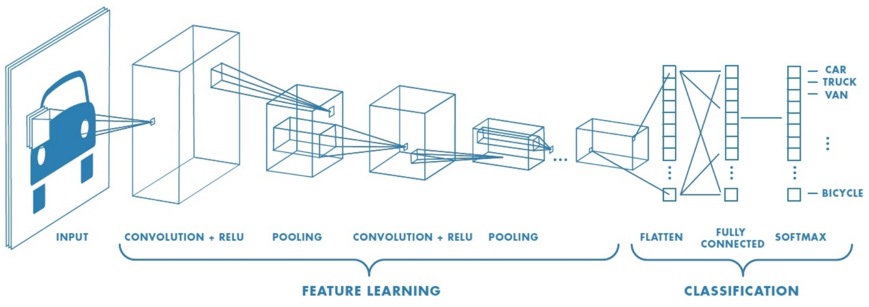


FIGURE 2.9: Structure of a Convolutional Neural Network (CNN) [18].

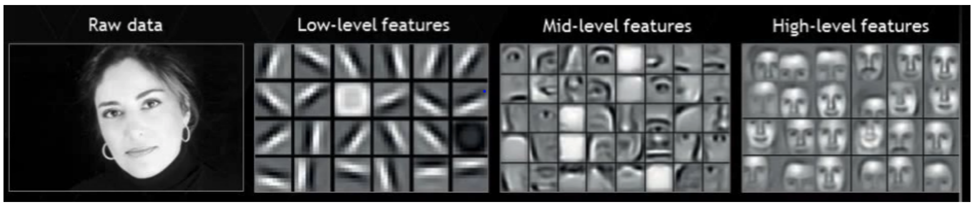


FIGURE 2.10: Neural network learning increasingly abstract features [20].

A CNN consists of convolutional, pooling, and fully connected layers, which perform the following tasks. A convolutional layer consists of a set of learned *filters*. These filters are moved across the input image, and as the network is trained it learns filters that activate when presented with some visual feature. As shown in fig. 2.10, early layers may learn basic features such as horizontal or vertical lines. Later layers learn increasingly high-level features, such as the shape of an eye, the corner of a mouth or some other building block of an image [20].

2.4.2 Image classification

Image classification refers to the task of assigning images to one of a set of possible classes, based on the contents of the image. Several competitions are hosted yearly, where research teams compare progress in detecting a wide variety of objects. While neural networks have been known since they were first described in literature in 1943 [16][21], their utility in complex tasks such as this has only begun to be developed in the past few years. This development has largely been attributed to advances in available computing power, the size of available data sets and algorithmic advances which allow for training larger neural networks.

As explained in [22], the ability to correctly classify complex images into one of several thousand possible categories requires a model with a large learning capacity. However, even with the millions of images in a dataset such as ImageNet it is infeasible to train an fully connected neural network to perform this task. Convolutional Neural Networks enable networks to detect features in images, with a lower computational complexity than standard

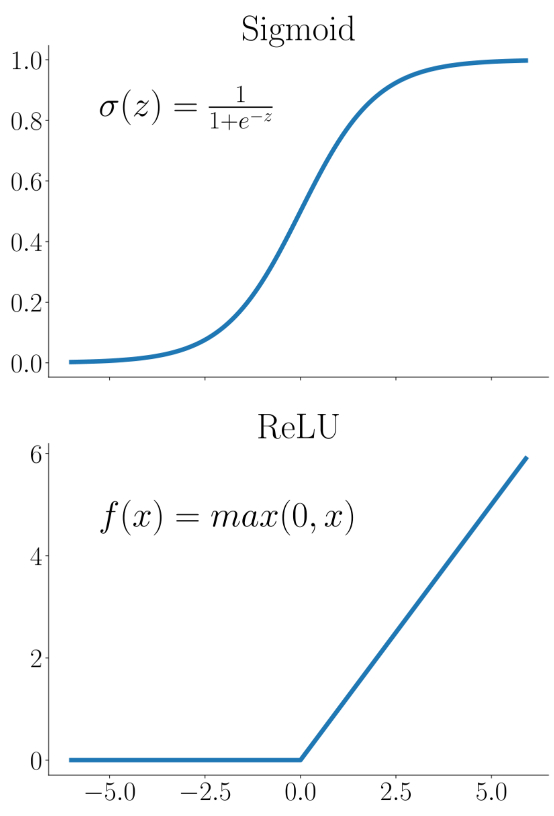


FIGURE 2.11: Illustration of Sigmoid and ReLU activation functions.

nets. At the most basic levels of the network, such features may be simple horizontal and vertical lines while later layers may learn more abstract features. This is illustrated in fig. 2.10 and fig. 2.9. Compared to fully connected networks, such as the one illustrated in fig. 2.8, CNNs have fewer connections and parameters and are therefore easier to train.

Until 2012, the top performing algorithms in the yearly ImageNet Large Scale Visual Recognition Challenge (ILSVRC) was dominated by algorithms requiring a large amount of manual hand coding of features, and that still had an error rate of over 26%. In 2012, researchers from the University of Toronto presented a deep CNN, made possible by using a new activation function termed the *Rectified Linear Unit (ReLU)*, see fig. 2.11. While the Sigmoid function quickly saturates which slows down training in deeper layers, the ReLU function does not. This makes networks using ReLU less susceptible to disappearing gradients, which has allowed for training of a deeper network than previously possible [22]. Their

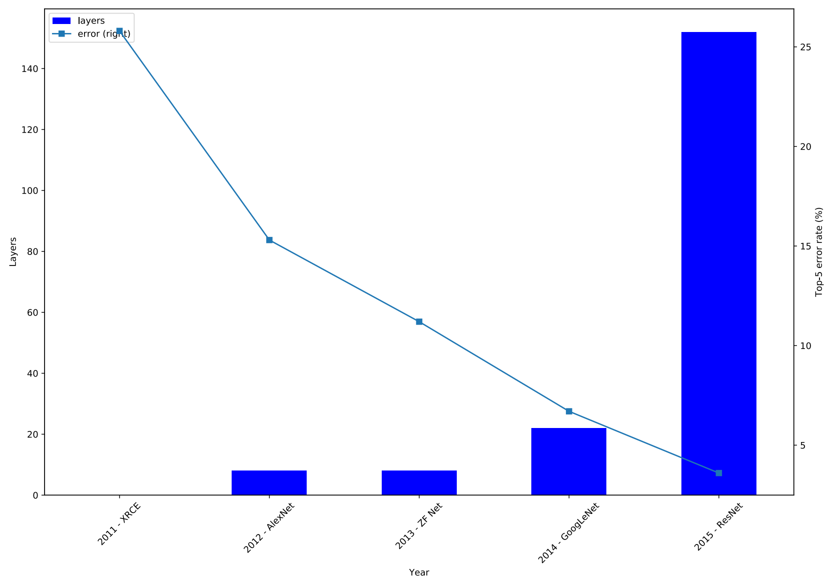


FIGURE 2.12: Progress of image classification, and the growth of deep networks. The graph shows the top-5 classification errors of each year's ILSVRC winner, and the depth of the network used. The 2011 winner did not employ a neural network solution. The data for this figure is from [22], [23], [19] and [24].

network was entered in the 2012 ILSVRC competition, and was able to classify images with an error rate of 15.3%. This outcompeted previous winners by over 10 percentage points, as shown in fig. 2.12. While the concept of CNNs and deep learning had been known for many years, this was the first significant use of such networks in computer vision.

Later advances have improved the applicability of deep neural networks further. In 2015, the authors of [24] proposed a novel technique of feed-forward from the input signal, leading to great improvement of the trainability of extremely deep networks. As explained, a neural network can approximate any mathematical function. The key insight of the authors is that if a network of a given depth can approximate such a function, and a functionally identical deeper network can be created by inserting unity layers, then a deeper network should not yield poorer performance than the original network. However, the result found in practice was that as network depth increases, network performance flattens out and eventually degrades. The authors proposed that this is a problem of training the network, rather than a fundamental problem of extremely deep networks. To overcome this, the authors note that the desired mapping is likely to be closer to its input $\boldsymbol{x}$, than a zero mapping. If the original desired mapping is the function $\mathcal{H}(\boldsymbol{x})$, the network is instead trained to approximate $\mathcal{F}(\boldsymbol{x}) = \mathcal{H}(\boldsymbol{x}) - \boldsymbol{x}$. This greatly improves the trainability of the network and allows for the increase in network depth shown in fig. 2.12. Unlike previous

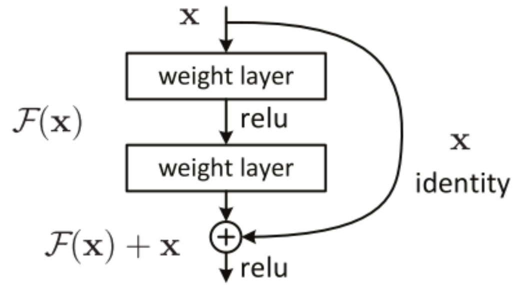


FIGURE 2.13: Illustration of a residual block [24].

approaches, the network achieves high accuracy through a deep but simple and repeating architecture composed of residual blocks as shown in fig. 2.13. This approach gives the network a computational complexity much lower than the number of layers might seem to imply [24].

As deeper networks are able to capture more abstract features in images, machine vision becomes more robust and consequently applicable to a larger range of tasks. The organizers of ILSVRC have announced that the 2018 competition will involve classifying 3D objects as well as 2D images.

2.4.3 Object Detection

In image classification tasks, the algorithm is trained to classify an image of a single object, or alternatively the principal object in the image for images with more than one object. In *object detection* tasks, the goal is to locate and classify all objects in the image, including identifying the boundary between objects and how they relate to one another [25]. Just as deep learning has revolutionized object classification, great strides have been made in object detection in recent years. As these developments are highly relevant to the field of robotics and the Cyborg project, this section gives an overview of these developments.

The same networks that are used for classifying images with a single object can also be applied to classify individual objects within a more complex image. The challenge in object detection is to identify these individual objects, so that they can be classified in the manner described in the previous section [25].

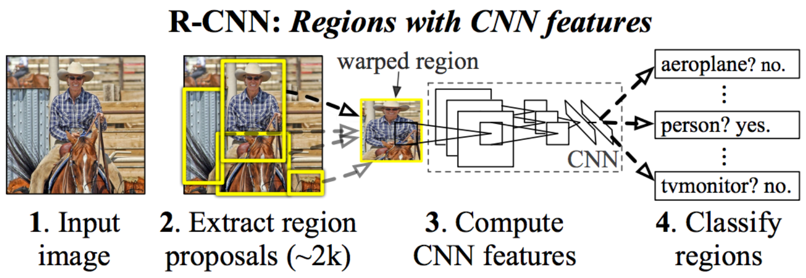


FIGURE 2.14: Object detection and classification using R-CNN [25].

Similar to the progress seen in object classification, previous to the deep learning revolution the best-effort approaches used other techniques than neural networks [25]. The very earliest attempts at object detection using CNNs employ a naive technique of moving a *sliding window* across the image. Using windows of various sizes and classifying these sub-images one by one, it is possible to detect objects within an image composed of multiple objects. However, this is a brute force approach and for fine-grained detection requires classifying thousands of windows per image.

Regions with CNN (R-CNN)

The approach taken by [25], called *Regions with CNN (R-CNN)* extracts region proposals from the image by a process called *selective search*. The selective search algorithm looks at the image through windows of different sizes, and identifies relevant regions by grouping adjacent pixels by color, texture or intensity. The resulting groups are reshaped into bounding boxes, and the contents of each bounding box are fed to an image classifier. If the contents of a box are successfully classified, the algorithm attempts to tighten the bounding box using a linear regression model. The process is shown in fig. 2.14. The approach has low error rate, but the connection of three different models leads to high complexity which makes the system difficult to train.

Fast R-CNN

While the approach of generating region proposals is significantly less computationally expensive than using sliding windows, it still requires that around 2000 region proposals are classified. In a follow-up paper, Gerschick proposes an approach that is improved in two significant ways called *Fast rcnn*. Firstly, rather than running individual regions of the image through the feature classifier one by one, features are computed on the entire image in a single pass to create a feature map, in a process called Region of Interest Pooling.

Subsequently, the features for each region can be obtained by selecting the appropriate area from the pre-computed feature map. Secondly, the three models used previously (region proposer, image classifier and bounding box regression model), are combined into a single model. This allows for end-to-end training, which greatly improves trainability [26].

Faster R-CNN

While the advances outlined above greatly improve on the efficiency of object detection algorithms, they expose the region proposer as a significant bottleneck [27]. Selective search is used to generate region proposals, while a CNN is used to extract features, classify the image and compute a bounding box. The work done by [27] uses the features computed by the CNN discussed previously, and combines this with a separate CNN, called the *Region Proposal Network*. They name this approach *Faster rcnn*. By making use of the same feature map that is used to classify images, the authors enable essentially cost-free region proposals. The network passes a sliding window over the feature map, and computes region proposals along with an objectness-score. The objectness-score measures the probability that the region contains an object, and allows for selecting only the regions that meet some minimum threshold.

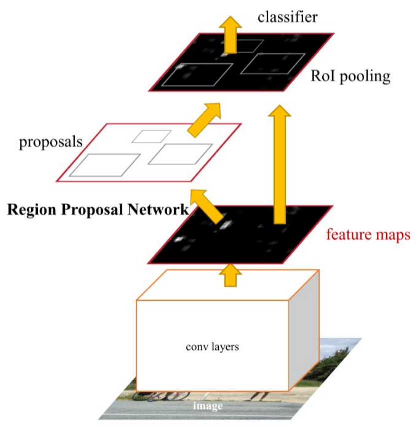


FIGURE 2.15: Object detection using R-CNN [27].

Mask R-CNN

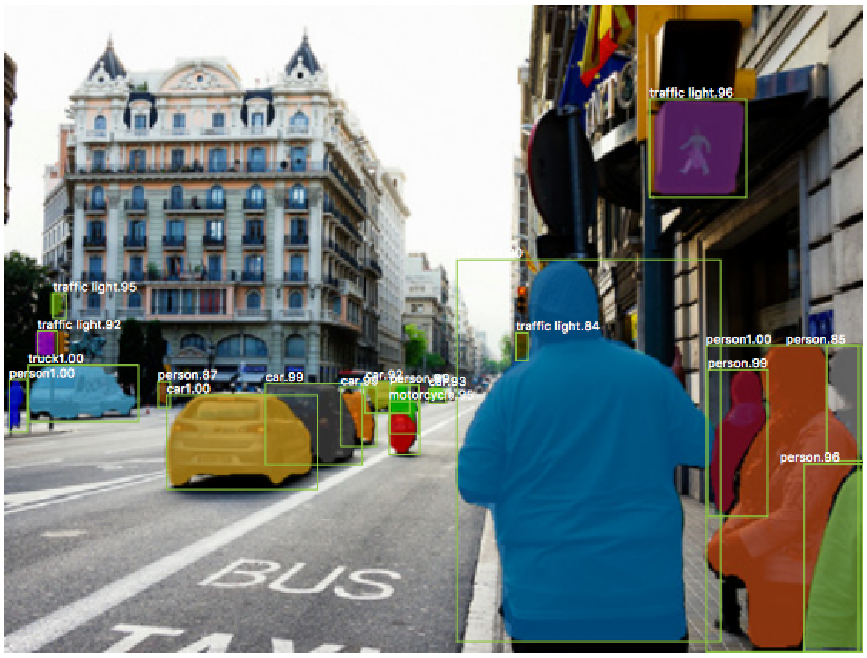


Figure 2.16: Pixel-level object detection using Mask R-CNN [28].

Later work extends this approach by detecting which pixels belong to each of the detected objects, as shown in fig. 2.16 [28], a method named *Mask R-CNN*. Here, in parallel to the region proposal network, a network is branched off which simultaneously computes a binary mask for each object. The binary map identifies which pixels belong to the object, as shown in fig. 2.16.

You Only Look Once (YOLO)

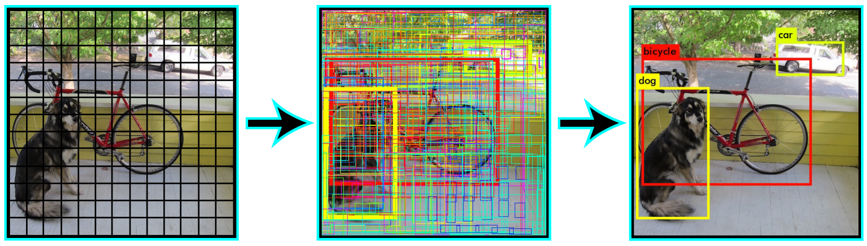


Figure 2.17: Object detection using YOLO [29].

The network used by this project, named You Only Look Once (YOLO), takes a novel approach. Rather than making a first pass to detect shapes, and a second pass to classify them, these operations are performed by a single neural network. The image is divided into a grid, and the probability that each cell in the grid forms part of a bounding box, as

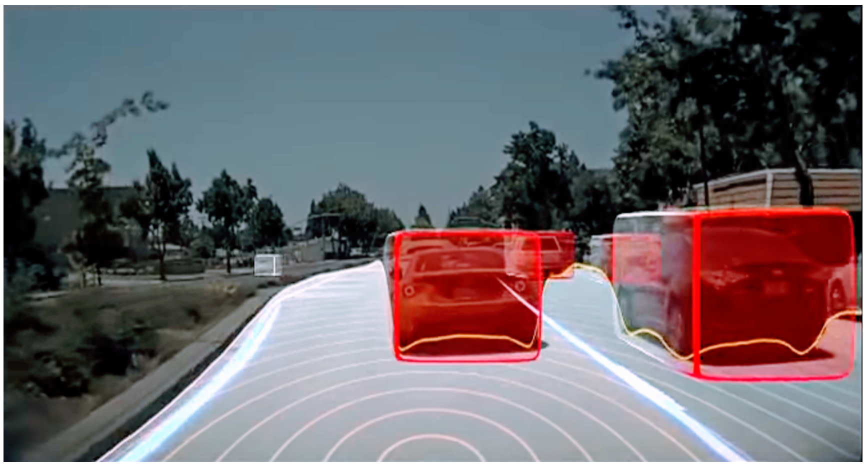


FIGURE 2.18: Free-space detection and 3D object detection for autonomous driving [31].

well as the probability that the contents of the box belongs to an image class, is computed simultaneously. These values are then combined to obtain a class-specific confidence score which encodes both the probability of the particular class appearing in the box, and how well the box fits the object. This allows the network to reason globally about the contents of the image, where the R-CNN approach treats each selected shape separately from the others. Furthermore, performing detection in a single pass provides greater speed [30].

2.4.4 Transfer learning

The level of performance discussed in the previous section requires the training of very large convolutional neural networks. Due to problems of overfitting in such large networks, it is necessary to train the network on very large datasets. In [31], researchers from NVIDIA describe a process by training such a network on the ImageNet dataset, consisting of approximately 1.2 million images. These are in turn augmented by various transformations to a total dataset of 22 million images. By training on such a large dataset, it is possible to create a very robust feature detector. However, the amount of training required can take weeks or months, as in the example described by the researchers.

The features learned by a convolutional neural network, as shown in fig. 2.10, have been found to be similar across many different applications. Observe in fig. 2.9, that the majority of a network performs feature detection while only the last few layers use these features to classify the image. One application of transfer learning is adapting an existing network to a new domain. This can be done by replacing erasing the weights of the classification part of the network, or replacing these layers with new layers if required. By keeping the weights of the feature learning layers of the network fixed, the network can be retrained to a new purpose. In the case of the work done at NVIDIA, a general image classifier trained on the ImageNet dataset was repurposed to identify objects in the autonomous vehicle domain. This can then be done much faster, and with a much smaller dataset, than what is required to train a new network. Using this technique resulted in a network able to detect pedestrians and vehicles, and identify which sections of the road are safe for driving, as shown in fig. 2.18[31].

2.5 Finite State Machines

At the outset of the work on this thesis, the cyborg was running the decision making software outlined in [32].

While state machines are a well-proven approach to flow control in many applications, there was a desire to investigate alternative approaches. This and the following section will provide some background on these two approaches. These concepts are explained in detail in [33].

A *finite state machine* is an abstract model of computation, that allows for actions to be encoded into a finite number of states. Changes from one state to another are referred to as *transitions*. The transition from an initial state to a new state is governed only by the starting *state*, and the *event*, as shown in fig. 2.19. Whenever the conditions for a transition are met, the system will perform that transition and execute the *action* associated with the new state. State machines typically have actions associated with entering and exiting a state.

2.5.1 Events

The type of state machines described here is termed an event-driven state machine. As the ROS system needs to perform many different actions, it is necessary that we do not block the system by polling for new events. Instead, events arrive asynchronously and are consumed by the state machine before it goes back to sleep.

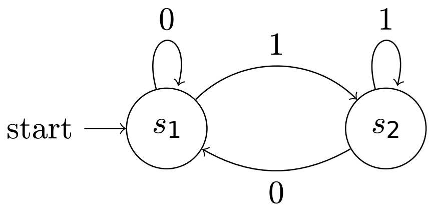

$$\begin{tikzcd}
\text{start} \arrow[r] & s_1 \arrow[loop above, "0"] \arrow[rr, bend left, "1"] & & s_2 \arrow[loop above, "1"] \arrow[ll, bend left, "0"]
\end{tikzcd}$$

FIGURE 2.19: An example of a simple state machine.

2.5.2 Actions

Actions in finite state machines are associated with transitions. In actual use, actions are typically modularized by splitting into an entry action and an exit action for each state as well as component executed repeatedly while the state is active. Often, it is desirable to specify behavior that should trigger when the state machine enters or leaves a state. This granularity achieved by splitting each actions simplifies behavior reuse, for example if every transition into a states share behavior that can be placed in an entry action.

2.5.3 Strengths and weaknesses

An overview of the use of state machines for AI in computer games is given in. In general, the concerns outlined here overlap with the concerns relevant in the sort of high-level decision making that is necessary in the Cyborg project.

State machines have work well for structured behavior, as they make it simple to implement a sequence of behaviors and the conditions for transitioning between them. They work particularly well for behavior where it is necessary to interrupt the current behavior due to external events.

One major weakness of state machines is that as the control system grows, the number of states and transitions between states quickly balloons and can be come unmanageable. Take as an example a control system for the Cyborg, where it should act in different ways depending on the charge level of its battery. If the battery is in good condition, the Cyborg should behave as normal. If the battery is in a medium condition, the Cyborg should attempt to minimize its energy use and should recharge as soon as is convenient. And, if the battery is in poor condition, the Cyborg should do whatever it can to recharge as soon as possible. Using a state machine, there are two ways to implement this. Either, would be possible to use a hierarchy of state machines, one for each battery state. Alternatively, each state in the state machine can be a state machine in itself, with a state for each battery state. In both cases, the number of states would be three times the number of states in the original control system. Furthermore, the number of transitions that must be added when including a new state grow with the size of the state machine, and the process can quickly become cumbersome.

Another weakness, which has been an important consideration for this project, is that state machines fall short when implementing unstructured behavior. While they are well suited for structured behavior of the type that a robot may do in an industrial setting, they are less suited for a robot which is intended to roam freely and react in a life-like way to its surroundings [33].

```
class Foo(smach.State):
    def __init__(self):
        smach.State.__init__(self, outcomes=['outcome1','outcome2'])
        self.counter = 0

    def execute(self, userdata):
        rospy.loginfo('Executing state FOO')
        if self.counter < 3:
            self.counter += 1
            return 'outcome1'
        else:
            return 'outcome2'
```

LISTING 2.2: State example from the SMACH documentation.

2.5.4 Implementation in ROS

The existing control system at the outset of this project was implemented using SMACH, a task-level state machine architecture for ROS. SMACH allows for fast prototyping and implementing complex state machines. Since it is a task-level architecture, it is not suitable for low-level control but rather for high-level decision making. As mentioned in section 2.5.3, state machines are not well suited to handle unstructured tasks. This is also stated by the SMACH developers [34].

States in SMACH are implemented as individual classes, which allows for reuse. This also includes reuse of behavior between states, by using object composition. Actions to be performed by the robot are associated with states in the state machine, and the robot will carry out that action for as long as it is in the particular state.

2.6 Behavior trees

A behavior tree is a model for task execution commonly used for artificial intelligence in video games, but that is also applicable to robotics and other types of control systems. Much of the strength of the behavior tree approach lies in the ability to compose complex behavior from simple building blocks, without needing to consider the implementation of each building block. In this sense they are similar to finite state machines. However, while the fundamental building block in a state machine is the state, the fundamental building block in behavior trees is the task. These concepts are explained extensively in [33].

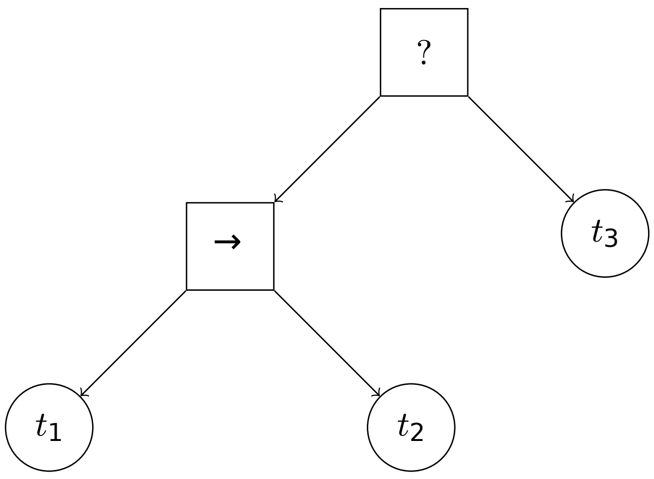


FIGURE 2.20: An example of a simple behavior tree.

2.6.1 Types of tasks

All nodes in a behavior tree share certain similarities. At their core, tasks are small execution units which are run for a certain length of time by the program which executes the behavior tree. At the end of execution, the task returns a status code, typically success, failure, or running. There are three types of tasks commonly used in behavior trees, *actions*, *conditions*, and *composites*.

Actions

Actions are behaviors that change the state of the system. For example, an action may cause the Cyborg to move to a given location or check the status of its battery. Actions can be long running, and will return a running status code to the calling function while they are performing their task.

Conditions

Conditions are checks made to test a property of the system. These can be used to do checks before executing an action, for example to check if the action is wanted or possible. To continue the example given above, the battery status that was updated may be compared to a critical value. In this way, an action can be executed if and only if it is found that the battery state permits it, or the system may execute behavior to recharge the battery if not.

```python
for node in children:
    status = node.tick()

    if status != False:
        return status

return False
```

Listing 2.3: An example implementation of the Priority node in Python.

```python
for node in children:
    status = node.tick()

    if status != True
        return status

return True
```

Listing 2.4: An example implementation of the Sequence node in Python.

Composites

Composites are nodes which tie the actions and conditions together. While actions and conditions are implemented by the programmer, there are a fixed and very small set of typical composite nodes. Generally there are two composite nodes that are used, the *Priority* node and the *Sequence* node.

In the simple example shown in fig. 2.20 the first control node, with the *?* symbol, is a Priority node. This node will execute, or tick, its children from left to right until one of the children returns successfully, as shown in listing 2.3.

The first child of this node, with the $\rightarrow$ symbol, is a Sequence node. This node will execute its children from left to right until one of the children returns failure, as shown in listing 2.4. By combining these, the tree shown in fig. 2.20 will first attempt to tick nodes *t1* and *t2*, and only if either of those fail will it tick node *t3*.

This approach can be repeated to create increasingly complex behavior, without consideration of the actual behavior performed by the tasks *t1*, *t2*, and so on. If behavior reuse is required, it is possible to compose the behavior in a sub-tree which can be included as a task in a parent tree.

2.6.2 The blackboard

In order to create more complex behavior, it is generally necessary to be able to share data between tasks in a tree. For example, the action which queries for the Cyborg's battery charge level would need somewhere to store this value, so that it can be used by the condition which compares the charge level to a critical value. The *blackboard*, as it is commonly called, is a data store used for this purpose. Generally, a blackboard is a key-value store for data exchange between tasks. The blackboard allows for storage of data that is available to any node in the tree. Depending on the implementation, it may also allow for data to be available only to other nodes in a sub-tree or data that is only available to the node which stored the data.

2.6.3 Strengths and weaknesses

When using behavior trees, complex behavior arises from composition of simple behavioral components. Where the state machine approach may encourage the developer to encode a large amount of behavior in each state, when using behavior trees it is useful to break tasks into their smallest useful parts. By combining the behavior tree approach with a graphical interface for composing the tree, this allows for non-programmers to create trees from basic building blocks without needing to write code.

One commonly encountered limitation of behavior trees is that they make it difficult to implement state-based behavior. For example, while it is simple to encode behavior that causes the Cyborg to dock with its charging station if the battery runs low, it is more difficult to make the Cyborg generally act in a way that conserves energy if it is in a state of having a somewhat low battery. Doing so would require either that two similar behavior trees are implemented, one for a normal state and one for an energy-depleted state, or that each task in the tree is a state machine with these two states. In this way, behavior trees share some of the weaknesses found in state machines. While this is possible to implement this functionality, it adds significant design complexity and is therefore generally avoided [33].

2.6.4 Implementation in ROS

Unlike finite state machines, ROS does not have a go-to implementation like SMACH. Therefore, to reach the objective of this thesis different approaches are evaluated. These will be described in the following chapter.

3 Implementation of the behavior tree control system

To replace the existing state machine-based control system, a behavior tree control system was implemented. This chapter will provide details on the goals of this system, the choices that were made in planning the system and a description of the implementation of the system itself.

3.1 Goals

At the outset of this thesis, the existing solution which used a state machine approach, was evaluated. SMACH is a task-level state machine architecture for the Robot Operating System (ROS). There are some intrinsic issues with large state machines, in particular concerning maintainability and scalability. As the cyborg state machine grew in size and complexity, these became increasingly apparent.

As outlined in chapter 2, the events that govern transitions between states in a state machine are tightly coupled. Because of this, when adding or removing a state, it is necessary to update the transitions between this state and potentially every other state in the state machine. Furthermore, a state machine with many states can be difficult to grasp as it becomes cluttered, and the many states and transitions require the user to maintain a complex mental model. Reusability of behaviors can also be difficult, as the states are tightly coupled to their transitions. Finally, as outlined in the SMACH documentation, SMACH is less well suited for unstructured tasks [34]. As one goal of the NTNU Cyborg project is life-like behavior, it was found that a decision making approach more suited for unstructured behavior was desirable.

3.2 Evaluation of alternatives

Before implementing a behavior tree control system for the Cyborg, several existing implementations and libraries were evaluated. Among these implementations were two existing implementation in ros called *ROS-Behavior-Tree* [35] and *pi_trees* [36]. Also, a behavior tree library not specifically for ROS is evaluated, called behavior3 by the author of [37]. The main considerations in choosing an implementation are that it needs to be written in C++ or Python, as those are the available languages in ROS. Furthermore, it would be beneficial if it were possible to specify the behavior tree in a configuration file.

3.2.1 Pi Trees

Pi Trees is implemented by the Pi Robot project. The library is implemented in Python, and is designed specifically for use with ROS [36]. Behavior trees are implemented in code, rather than being specified using a configuration file, which adds some complexity to the design process. As an example, consider a simple tree implementation from the Pi Robot project [38] shown in listing 3.1.

The Iterator node used here has not been discussed previously, as it is less commonly used. It iterates through its children in the same way as done by the other types of composite nodes, but it ignores their return value. As can be seen in the example, each composite node needs to be named. Properly naming things is considered one of the hardest things in computer science [39], and it was found that the requirement to name composite nodes added significant developer overhead.

Also, as Python lacks the ability to do forward declarations of objects, there is a significant amount of jumping back and forth when configuring the tree. Note for example that the *CLEAN_ROOM* task is referred to throughout the example. This is necessary as its children need to be created before they can be assigned to the parent node. It was found that this requirement to essentially build the tree from the bottom up was harder to do than to build the tree from the top down.

```python
for room in black_board.task_list.keys():
    # Convert the room name to upper case for consistency
    ROOM = room.upper()

    # Initialize the CLEANING_ROUTINE selector for this room
    CLEANING_ROUTINE[room] = Selector("CLEANING_ROUTINE_" + ROOM)

    # Initialize the CHECK_ROOM_CLEAN condition
    CHECK_ROOM_CLEAN[room] = CheckRoomCleaned(room)

    # Add the CHECK_ROOM_CLEAN condition to the CLEANING_ROUTINE selector
    CLEANING_ROUTINE[room].add_child(CHECK_ROOM_CLEAN[room])

    # Initialize the CLEAN_ROOM sequence for this room
    CLEAN_ROOM[room] = Sequence("CLEAN_" + ROOM)

    # Initialize the NAV_ROOM selector for this room
    NAV_ROOM[room] = Selector("NAV_ROOM_" + ROOM)

    # Initialize the CHECK_LOCATION condition for this room
    CHECK_LOCATION[room] = CheckLocation(room, self.room_locations)

    # Add the CHECK_LOCATION condition to the NAV_ROOM selector
    NAV_ROOM[room].add_child(CHECK_LOCATION[room])

    # Add the MOVE_BASE task for this room to the NAV_ROOM selector
    NAV_ROOM[room].add_child(MOVE_BASE[room])

    # Add the NAV_ROOM selector to the CLEAN_ROOM sequence
    CLEAN_ROOM[room].add_child(NAV_ROOM[room])

    # Initialize the TASK_LIST iterator for this room
    TASK_LIST[room] = Iterator("TASK_LIST_" + ROOM)

    # Add the tasks assigned to this room
    for task in black_board.task_list[room]:
        # Initialize the DO_TASK sequence for this room and task
        DO_TASK = Sequence("DO_TASK_" + ROOM + "_" + task.name)

        # Add a CHECK_LOCATION condition to the DO_TASK sequence
        DO_TASK.add_child(CHECK_LOCATION[room])

        # Add the task itself to the DO_TASK sequence
        DO_TASK.add_child(task)

        # Create an UPDATE_TASK_LIST task for this room and task
        UPDATE_TASK_LIST[room + "_" + task.name] = UpdateTaskList(room, task)

        # Add the UPDATE_TASK_LIST task to the DO_TASK sequence
        DO_TASK.add_child(UPDATE_TASK_LIST[room + "_" + task.name])

        # Add the DO_TASK sequence to the TASK_LIST iterator
        TASK_LIST[room].add_child(DO_TASK)

    # Add the room TASK_LIST iterator to the CLEAN_ROOM sequence
    CLEAN_ROOM[room].add_child(TASK_LIST[room])

    # Add the CLEAN_ROOM sequence to the CLEANING_ROUTINE selector
    CLEANING_ROUTINE[room].add_child(CLEAN_ROOM[room])

    # Add the CLEANING_ROUTINE for this room to the CLEAN_HOUSE sequence
    CLEAN_HOUSE.add_child(CLEANING_ROUTINE[room])
```

Listing 3.1: Implementation of a behavior tree using Pi Trees, from the Pi Trees documentation.

```
#include <actions/action_test_node.h>
#include <conditions/condition_test_node.h>
#include <behavior_tree.h>
#include <iostream>

int main(int argc, char **argv)
{
    ros::init(argc, argv, "BehaviorTree");

    try
    {
        int TickPeriod_milliseconds = 1000;

        BT::ActionTestNode* action1 = new BT::ActionTestNode("Action1");
        BT::ConditionTestNode* condition1 = new BT::ConditionTestNode("Condition1");
        BT::SequenceNodeWithMemory* sequence1 = new BT::SequenceNodeWithMemory("seq1");

        action1->set_time(5);
        condition1->set_boolean_value(true);
        sequence1->AddChild(condition1);
        sequence1->AddChild(action1);

        Execute(sequence1, TickPeriod_milliseconds);
    }
    catch (BT:BehaviorTreeException& Exception)
    {
        std::cout << Exception.what() << std::endl;
    }

    return 0;
}
```

Listing 3.2: Implementation of a behavior tree using ROS-Behavior-Tree, from the ROS-Behavior-Tree documentation.

3.2.2 ROS-Behavior-Tree

ROS-Behavior-Tree is implemented in C++ , and like Pi Trees it is created specifically for use with ROS. The tree is implemented in code, as shown in listing 3.2. As the tree is implemented in code the same added complexity is seen here as described above. Furthermore, as C++ is a compiled language, any change to the tree requires that the project is rebuild.

3.2.3 behavior3

behavior3 is implemented as both a Python library, *behavior3py*, as well as a JavaScript library, *behavior3js*. Additionally, the project includes an editor, *behavior3editor*. Of the options evaluated, behavior3 was the only one that allows for specifying the tree in a text file. An attempt has been made to implement a tree using one of the other libraries, but it was quickly discovered that specifying the tree in code, either C++ or Python, became unwieldy as the tree size increased.

After evaluating the described options, it was decided to use behavior3. The editor provides a graphical interface for creating behavior trees, and allows the user to export these as JSON configuration files. The editor is shown in fig. 3.2. The configuration files can be parsed and run using behavior3py or behavior3js. As ROS allows for nodes to be implemented in C++ or Python, this project will use behavior3py.

3.3 Implementation

As described in chapter 2, a behavior tree is build from standard composite nodes, as well as nodes that implement decorators and actions. To use behavior3py a ROS node was made, called *cyborg_bt*, as well as a node for organizing custom decorators and actions called *cyborg_bt_nodes*. cyborg_bt imports the required decorators and actions needed by the Cyborg control system from cyborg_bt_nodes and runs the tree produced with behavior3editor. Much of the functionality required to implement these decorators and actions has been placed in ROS services or topics, in order to make behavior as reusable as possible.

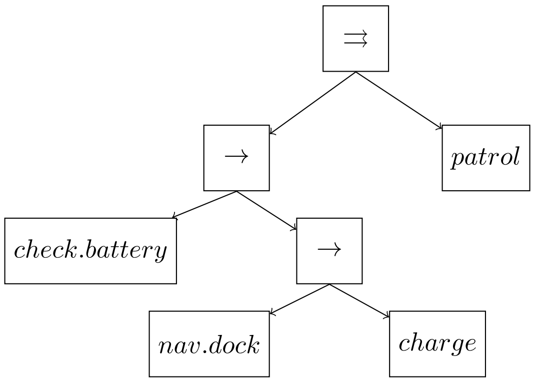
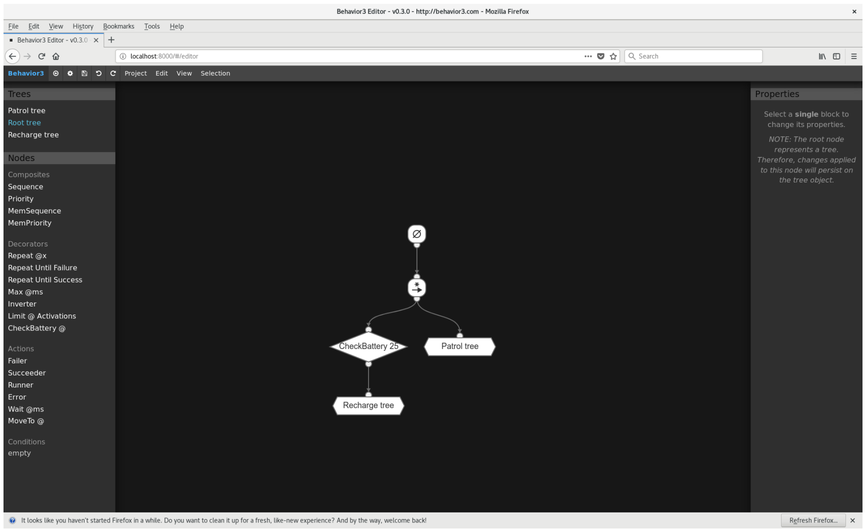


FIGURE 3.1: Behavior tree that checks battery state, and charges if necessary, while patrolling.



Select a **single** block to change its properties.

NOTE: The root node represents a tree. Therefore, changes applied to this node will persist on the tree object.

It looks like you haven't started Firefox in a while. Do you want to clean it up for a fresh, like-new experience? And by the way, welcome back!

FIGURE 3.2: Screenshot of behavior3editor showing implementation of a simple behavior tree.

An example of such a node, the action node MoveTo, which moves the Cyborg to a requested location, is included with the attached source code. The node issues a movement request to the navigation subsystem, and then monitors the progress of the Cyborg as it moves along the path to its target. If progress stalls for longer than an allowed timeout, the movement request is aborted. In this case, we monitor the movement along the path to the target and not merely the Euclidean distance to the target. This is important, as the Cyborg may have to navigate around obstacles along its path which can involve moving further away from the target.

A simple demonstration of a working behavior tree is shown in fig. 3.1. This behavior tree causes the Cyborg to move in a patrol pattern between several locations while it monitors the state of the Cyborg's battery charge, and recharges as needed. Figure 3.2 shows the same functionality being implemented in behavior3.

4 Implementation of the control system monitoring application

To monitor the state of the control system described in chapter 3, a monitoring application was implemented. This chapter will provide details on the goals of this system, the choices that were made in planning the system and a description of the implementation of the system itself.

4.1 Goals

In addition to the control system, a main objective of this thesis is to present the development of an application that allows the user to monitor the status of the Cyborg. One shortcoming of behavior trees is that it can be hard to troubleshoot behavior, as the behavior stems from complex interaction of simple components [33]. For this reason, it was found necessary to create an application that made it possible to visualize the execution of the tree in real time.

The application should show the structure of the behavior tree, as well as highlight those nodes in the tree that are being executed at the current time. It is also useful for the application to give the user the ability to halt and resume the execution of the tree.

Initially, the ability to interact with the tree in order to manually change its execution was also investigated. The original idea was for the application to include the ability to trigger execution of a particular node of the behavior tree. However, the necessary increase in the complexity of the application was found not to be justified. Description of the feature is included, as it significantly influenced the planning stage of the monitoring application before being dropped from consideration.

4.2 Evaluation of alternatives

There are several libraries and applications available that allow for visualization of the relevant data, and a choice had to be made regarding which option to choose.

4.2.1 Plotting framework

One approach that was evaluated was to create a standalone monitoring application, or even a simple dashboard that could be presented in a web browser. Among the options that have been investigated are d3, which is a visualization library for Javascript, Bokeh, and Plotly, which are visualization libraries for Python. One of the desired features of the application was that the visualization should be interactive, so that the user could influence the state of the Cyborg. All three libraries enable rich visualizations in the browser, but Bokeh and Plotly were found to offer a higher degree of interaction. As interaction was originally a desired feature in the monitoring application, a Python solution using either Bokeh or Plotly was investigated further.

ROS nodes can be implemented in either C++ or Python, as explained in chapter 2. For implementing visualizations as a web service, Python was deemed to be a good choice. The monitoring application might have been implemented as a ROS node, were it not for the fact that ROS uses Python 2.7. Python 2.7, unfortunately, predates the introduction of asynchronous support in Python. This presented a challenge. Three ways to overcome this were identified that will be explained in the following sections.

4.2.2 Rosbridge

In order to access information from ROS in an outside application, the Robot Web Tools project has developed a protocol called *rosbridge* [40]. The rosbridge protocol exports data from ROS using websocket which is a two-way TCP protocol, and uses a simple JSON API for communication. An application, including a web application, can access information from the Cyborg by communicating with ROS using websocket. This solution makes the interface easily accessible without requiring that the user installs software locally, and it allows for the Cyborg to be monitored from any computer with a web browser.

Existing frontends for rosbridge include *roslibjs*, which is written in Javascript. As it was found that visualization should be done with Plotly or Bokeh, this would have required porting of roslibjs to Python.

4.2.3 ROS 2.0

As porting roslibjs to Python would involve a significant amount of work, ROS 2.0 was evaluated as an alternative. ROS 2.0 is a new project by the creators of ROS intended to improve on some of the fundamental shortcomings of ROS. In doing so, it also solves the same problems that rosbridge intends to solve, by making it far easier for outside applications to communicate with an ROS control system.

As outlined in [12], the development of ROS involved the from-scratch implementation of a publish-subscribe system. Since the start of the ROS project, many new technologies have become available which provide features beyond the scope of the implementation used in ROS. Crucially for the purpose of this project ROS 2.0 uses Python 3, which is a requirement for the visualization libraries discussed above.

While the core ROS 2.0 implementation is considered stable and ready for production use at the time of writing, the Cyborg project depends on ROS nodes that are not yet available for ROS 2.0. Specifically, the project depends on nodes developed by MobileRobots, the manufacturer of the Pioneer LX, and it is outside the scope of this project to port these to ROS 2.0. However, ROS 2.0 does include the ability to interoperate with ROS, meaning that it is possible to leave existing ROS functionality as is, while new functionality could target ROS 2.0. Unfortunately, many important libraries are not yet ported, including underlying functionality such as nav_core and move_base, the navigation and movement subsystems respectively. Once these subsystems are ported it would be feasible to transition the Cyborg project to ROS 2.0, and certainly to target new functionality at ROS 2.0.

4.2.4 rqt

After evaluating both rosbridge and ROS 2.0, it was found that these solutions complicated the architecture of the Cyborg control system more than necessary. Furthermore, using off-the-shelf solutions wherever possible minimizes necessary work and maintenance for the Cyborg project, which allows for efforts to be focused on improving core functionality.

An alternative approach to the solutions described above, and the one decided upon by Martinius Knudsen (the author's co-advisor) and the author, was to implement the required functionality as one or more plugins for rqt. rqt is a Qt-based framework for Graphical User Interface (GUI) development for ROS, and comes with a rich set of plugins already available. rqt provides much of the functionality the Cyborg project requires, such as visualization of time series data, a rich logging interface and command input. For these reasons, the choice was made to implement the behavior tree visualization as a plugin for rqt, and use existing rqt plugins for the other visualizations needed by the project.

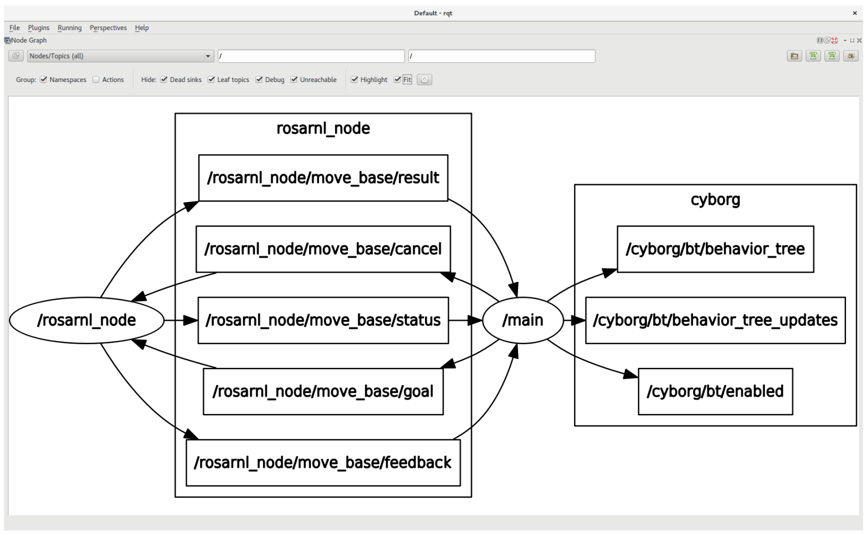

Default - rqt
File Plugins Running Perspectives Help
Node Graph
Nodes/Topics (all)
Group: Namespaces Actions
Hide: Dead sinks Leaf topics Debug Unreachable
Highlight Fit
rosarnl_node
/rosarnl_node/move_base/result
/rosarnl_node/move_base/cancel
/rosarnl_node
/rosarnl_node/move_base/status
/rosarnl_node/move_base/goal
/rosarnl_node/move_base/feedback
/main
cyborg
/cyborg/bt/behavior_tree
/cyborg/bt/behavior_tree_updates
/cyborg/bt/enabled

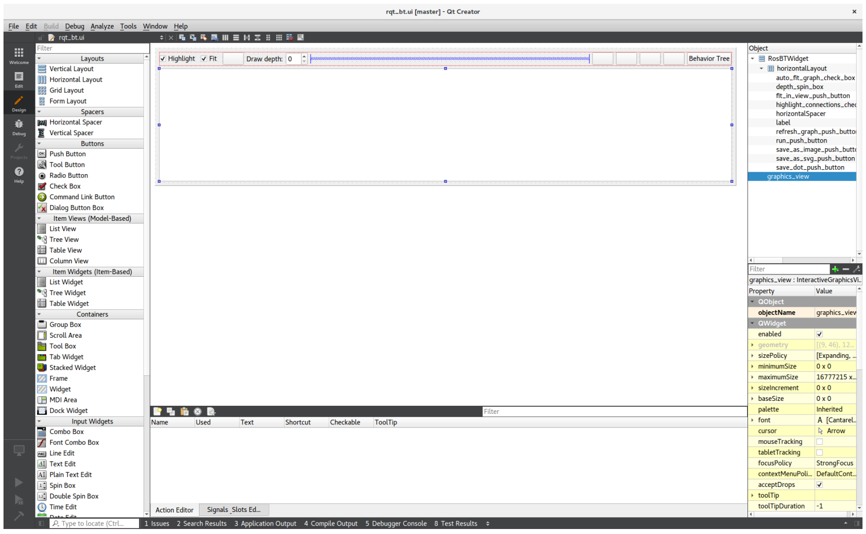
FIGURE 4.1: Screenshot of rqt-graph, showing the currently running ROS nodes and the communication pathways between them.

The behavior tree visualization plugin will be based on the existing rqt graph visualizer *rqt_graph*. This plugin visualizes the nodes that make up the ROS control system, and the communication paths between these nodes as shown in fig. 4.1. The use case for rqt_graph and the behavior tree plugin are somewhat similar, as they both generate DOT code that describes the graph which will then be passed to a Qt library for drawing. Due to this overlap, the rqt_graph served as valuable inspiration for the behavior tree plugin.

4.3 Implementation

4.3.1 Graphical User Interface

Implementing the GUI using Qt provides a range of graphical widgets, which can be composed using the Qt Creator software. The resulting UI text file can then be loaded in Python which automatically creates objects for each widget specified in the UI file. Interactive widgets provide signals which are emitted upon actions such as a button click, a value change and so forth. These signals may be connected to functions implemented by the developer, in order to perform actions when the user interacts with the graphical interface. Actions may be performed immediately, or they may be deferred in situations where this is required. For example, long running tasks may cause the interface to become unresponsive if they are executed indiscriminantly, and it may be necessary to defer them for the sake of a better user experience. Where required, widgets may be subclassed in order to add additional signals, but this has not been necessary for this project as the default signals sufficed to implement the needed functionality.


rqt_bt.ui [master] - Qt Creator
File Edit Build Debug Analyze Tools Window Help
Highlight
Fit
Draw depth: 0
Behavior Tree
Action Editor
Signals_Slots Ed...

Object
- RosBTWidget
  - horizontalLayout
    - auto_fit_graph_check_box
    - depth_spin_box
    - fit_in_view_push_button
    - highlight_connections_chec
    - horizontalSpacer
    - label
    - refresh_graph_push_butto
    - run_push_button
    - save_as_image_push_butt
    - save_as_svg_push_button
    - save_dot_push_button
  - graphics_view

graphics_view : InteractiveGraphicsVi...

Property	Value
QObject	
objectName	graphics_view
QWidget	
enabled	✓
geometry	[(9, 46), 12...
sizePolicy	[Expanding, ...
minimumSize	0 x 0
maximumSize	16777215 x...
sizeIncrement	0 x 0
baseSize	0 x 0
palette	Inherited
font	A [Cantarel...
cursor	Arrow
mouseTracking	☐
tabletTracking	☐
focusPolicy	StrongFocus
contextMenuPoli...	DefaultCont...
acceptDrops	✓
toolTip	
toolTipDuration	-1

FIGURE 4.2: Screenshot of Qt Creator.

The behavior tree visualization plugin was named *rqt_bt*. The plugin was designed using Qt Creator. The interface of Qt Creator is shown in fig. 4.2. The entire GUI is composed of a widget created by the author, *rqt_bt_widget*, which acts as a container for the other components of the GUI. At the top of the interface, the layout is made up of a *QHBoxLayout* widget, which is a container widget for laying out widgets horizontally. This widget contains the control buttons for interacting with the application. Below this the layout contains an *InteractiveGraphicsView* widget. This is a widget created by the ROS project, which extends the stock *QGraphicsWidget*. The widget is responsible for drawing the actual graphics view to the screen. All of the listed widgets subclass *QWidget*, which is the base class for all user interface objects in Qt.

The buttons in the interface allow the user to control the behavior of the plugin in the following ways:

- The *Highlight Connections* check box check box will cause the interface to highlight the connections from a node to its parent and its children, when the node is hovered over by the mouse.
- The *Fit* check box will cause the graph to be automatically drawn to fit within the view port, so that the whole graph can be seen at the same time. The graph will resize to fit even if the graph changes.
- The *Fit in View* push button will cause the graph to be resized to fit in the view port, but only a single time. That is, it will not be automatically updated.
- The *Draw Depth* spin box allows the user to select how deeply nested behavior trees should be drawn. Trees deeper than the selected level are collapsed to a single node, as shown in fig. 4.4.
- The *Save as DOT*, *Save as SVG* and *Save as Image* push buttons allow the user to save the graph in a variety of formats.
- The *Refresh Graph* push button forces redrawing of the graph.
- The *Run* push button enables or disables the behavior tree node running in ROS.

The run push button deserves some further explanation. This button allows the user to send an enable or disable service request to the behavior tree node, which pauses execution of the behavior tree. The plugin monitors the status of the behavior tree, so that the plugin sends the correct service request depending on the current status of the behavior tree. If the behavior tree node is running, the button displays a pause symbol and the plugin will send a request to halt execution of the behavior tree. Conversely, if the behavior tree's execution is paused, the button displays a play symbol and the plugin will send a request to resume execution of the behavior tree.

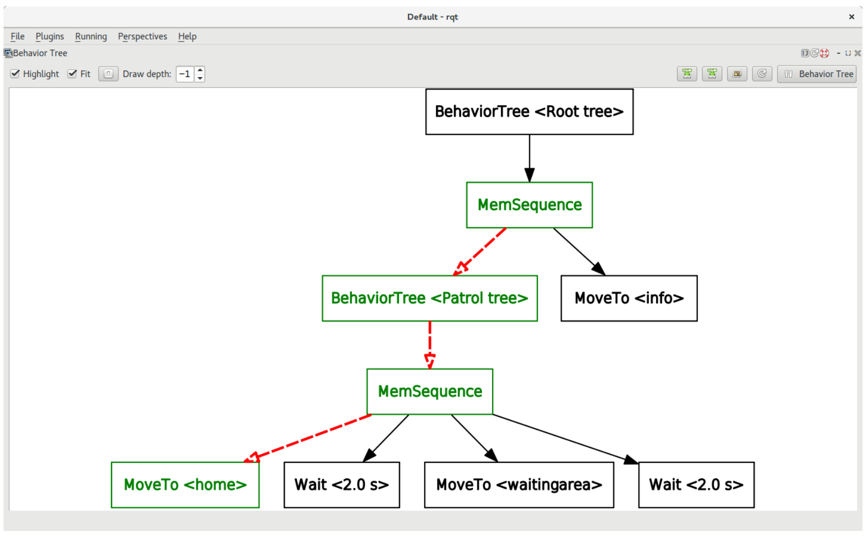


FIGURE 4.3: Screenshot of rqt_bt.

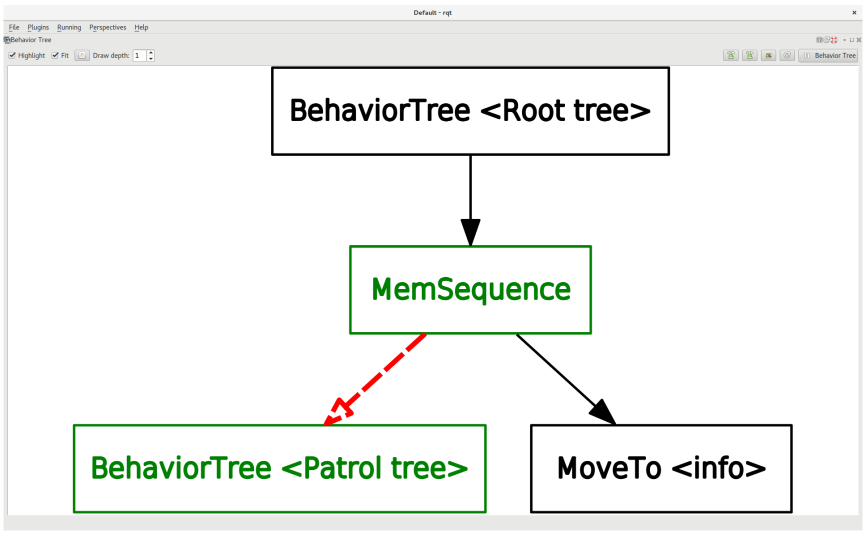

Default - rqt
File Plugins Running Perspectives Help
Behavior Tree
Highlight Fit Draw depth: 1
Behavior Tree
BehaviorTree <Root tree>
MemSequence
BehaviorTree <Patrol tree>
MoveTo <info>

FIGURE 4.4: Screenshot of rqt_bt with reduced drawing depth.

4.3.2 Displaying the tree

In order to display the behavior tree, as well as which nodes are currently active, two pieces of information are required to be published by the behavior tree node cyborg_bt. Firstly, we require that it publish the structure of the tree, and secondly we require that it publishes a list with the IDs of the currently active nodes. When the cyborg_bt node is created, the behavior tree is traversed and a NetworkX graph is created that represents the tree. NetworkX provides a method for converting these graphs to and from JSON data, which allows for transmitting the structure of the tree as a string message on a topic, and recover it upon receipt. The generated data is made available in two separate topics, as a JSON string, and as a list of ID strings, respectively.

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

import json
import networkx as nx
import pygraphviz
import rospy

from cyborg_msgs.msg import BehaviorTree, BehaviorTreeNodes
from networkx.utils.misc import make_str

class BTData(object):
    def __init__(self, data_sub_name=None, update_sub_name=None):
        if data_sub_name:
            rospy.Subscriber(data_sub_name, BehaviorTree, self._bt_cb)

        if update_sub_name:
            rospy.Subscriber(update_sub_name, BehaviorTreeNodes, self._bt_update_cb)

        self.tree = None
        self.active_nodes = []

    def get_graph(self):
        # Create a graph from the JSON data, or fall back to an empty graph
        try:
            graph = tree_graph(self.tree)
        except:
            graph = nx.OrderedDiGraph()

        G = nx.drawing.nx_agraph.to_agraph(graph)

        return G, self.active_nodes

    def _bt_cb(self, msg):
        self.tree = json.loads(msg.tree)

    def _bt_update_cb(self, msg):
        self.active_nodes = msg.ids
```

Listing 4.1: Implementation of the BTData class for rqt_bt.

In order to consume the provided data, the rqt_bt_widget creates a *BTData* object, which listens to these two topics, as shown in listing 4.1. The BTData instance is provided to the DOT code generator class *BTDotcodeGenerator*, shown in listing 4.2. The BTData instance will continue to listen to the provided topics, and maintain an updated representation of the behavior tree graph, which can be retrieved using the *get_graph()* method. The topics published by cyborg_bt are latched, meaning that the most recently transmitted message will be retransmitted to any new subscribers. This means that the BTData instance receives updated information as soon as it is created. If this is not the case, BTData falls back to providing an empty graph so that consumers of the graph avoid having to add special cases for invalid data.

When rqt_bt_widget requires a redraw of the behavior tree graph, either due to periodic redrawing or due to user interaction, the *generate_dotcode()* method in BTDotcodeGenerator is called. This method retrieves the current graph and the list of currently active nodes from its instance of BTData. The method performs processing of the graph, such as culling nodes that are deeper than the requested drawing depth, and adds highlighting to indicate the currently active nodes. Finally, DOT code is generated to represent the graph, which is passed back to rqt_bt_widget so that it can be drawn by the Qt framework. If the DOT code has changed since the last time it was drawn, it is passed to an instance of *DotToQtGenerator*. This class is provided by *qt_dotgraph*, which is a library created by the ROS project that provides helper functions for drawing DOT graphcs in Qt.

Using the *dotcode_to_qt_items()* method provided by DotToQtGenerator, we create a list of nodes and edges. These are added to a *QGraphicsScene* which is passed to the QGraphicsView. The QGraphicsView widget draws the scene on the user's screen.

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

import itertools
import pydot
import rospy

def pairwise(iterable):
    "s -> (s0,s1), (s1,s2), (s2, s3), ..."
    a, b = itertools.tee(iterable)
    next(b, None)
    return itertools.izip(a, b)

class RosBTDotcodeGenerator(object):
    def __init__(self, data_provider):
        self.data_provider = data_provider

    def generate_dotcode(self, max_depth=-1):
        G, active_nodes = self.data_provider.get_graph()

        # Remove nodes that are too deep
        for node in G.nodes():
            node.attr['shape'] = 'box'

            # Remove nodes deeper than the given draw depth
            if max_depth != -1 and int(node.attr['depth']) > max_depth:
                G.remove_node(node)

        G.layout(prog='dot')

        # Highlight active nodes
        for node in active_nodes:
            if node in G.nodes():
                n = G.get_node(node)
                n.attr['color'] = 'green'

        for (start, end) in pairwise(active_nodes):
            if start in G.nodes() and end in G.nodes():
                try:
                    e = G.get_edge(start, end)
                    e.attr['style'] = 'dashed'
                    e.attr['penwidth'] = 2

                    e.attr['colorR'] = 255
                    e.attr['colorG'] = 0
                    e.attr['colorB'] = 0
                except KeyError:
                    pass

        dot = G.string()

        return dot
```

Listing 4.2: Implementation of the BTDotcodeGenerator class for rqt_bt.

5 Implementation of object detection and classification

To provide the Cyborg's control system with information about objects in the Cyborg's surroundings, an object detection and classification system was implemented. This system uses data from the Cyborg's depth-sensing stereo camera to both identify and locate objects in the Cyborg's surroundings. It continues the work done by Experts in Teamwork (EiT) students working for the Cyborg project, who evaluated various object detection approaches and produced a demonstration using the You Only Look Once (YOLO) algorithm [41]. Throughout the time spent on this thesis, the author has collaborated with and guided this group in their work. This has allowed the author to focus on proven ideas, unlike previous parts of this thesis where much effort was spent identifying promising solutions. While partially based on these ideas, the following work is done by the author. This chapter will provide details on the goals of this system, the choices that were made in planning the system and a description of the implementation of the system itself.

5.1 Goals

As described in chapter 2, the ros-zed-wrapper from StereoLabs exposes the left and right images of the Cyborg's stereo camera, as well as a depth map calculated from these images. Here, we intend to analyze the images from the stereo camera to detect objects in the images, and then calculate the distance to each object. The output from the system will be twofold. First, an image with bounding boxes drawn around each detected object is published as an ros topic. Furthermore, the object's class and distance from the Cyborg also drawn on the image. Secondly, a list of detected objects, along with a header containing the time stamp of the prediction message, and a copy of the header from the input image is also published as an ROS topic. Each detected object will be represented by a structure containing its classification, a polygon structure describing its bounding box, its distance from the Cyborg, and the classification confidence outputted by the detection algorithm.

```
string label
float64 confidence
float64 distance
geometry_msgs/Polygon bounding_box
```

```
Header header
Header image_header
Prediction[] predictions
```

LISTING 5.1: Prediction and Predictions message formats.

5.2 Evaluation of alternatives

As described in chapter 2, there are many possible algorithms that perform object detection and image classification. Key to this project is using an approach that performs well using the limited available computational resources on the Cyborg. It was also desirable to use off-the-shelf software, to minimize the work needed to create and maintain the solution.

As described, the YOLO algorithm allows for object detection and classification using a single pass through the neural network, which is an efficient approach [30]. As the object detection part of this thesis continued existing work by [41], the choice was made to continue to use the YOLO algorithm. Consideration was given to which implementation of this algorithm to use.

The demo implementation by [41] used a library called PyYOLO. PyYOLO is a Python wrapper around the original CUDA implementation by the author of YOLO, and therefore performs very well. The downside to this is that the underlying YOLO implementation must be compiled for PyYOLO to be used, which adds to the maintenance burden of the project members.

As of late 2017 there is support for YOLO in the *Deep Neural Network (DNN)* module of OpenCV, and it was decided that this merited further research. While OpenCV lacks *CUDA* support, and was therefore expected not to perform as well in timing benchmarks, this is planned for inclusion in future versions of OpenCV. Furthermore, OpenCV's DNN module is able to use a number of neural networks, which allows for different object detection algorithms to be run with only minimal changes to the Cyborg software.

5.3 Implementation

In this section, we will detail to separate implementations using this algorithm, written by the author. These implementations make use of a reference implementation of the YOLO algorithm, to meet the goals outlined above, but are subject to different considerations. The weights used in the neural network are published by the creator of YOLO [30].

5.3.1 C++ implementation

First, a C++ implementation using OpenCV was developed. There are two significant advantages to implementing the object detection node in C++. First, it is possible to run the detection algorithm as a nodelet in the same process as the ros-zed-wrapper. A nodelet is a variation on the concept of an ROS node, as described in chapter 2. Rather than running the zed-ros-wrapper node and the object detection node in separate processes, which would require transporting raw image data over TCP, these are run in the same process. This allows us to use the same publish/subscribe interface as in a normal node, but have message transport handled using shared memory. This change is handled seamlessly by the Nodelet class, which nodelets inherit from. The only caveat the developer must keep in mind is that messages cannot be modified, once published to a topic, as their memory is now shared with all nodelets subscribing to the topic.

In order to make up the object detection node, there are two approaches to launching the component nodelets that make up the node. It is possible to implement a node in C++ in the usual manner, instantiate a nodelet Loader and use this to load each nodelet. This is shown in listing 5.2. The alternative approach is to load a nodelet manager, and each nodelet, using a launch file. This accomplishes the same result, but allows for changing the launch procedure without requiring a recompilation. The approach is shown in listing 5.3.

The second advantage to using C++ is that we are able to use the implementation of the YOLO algorithm created by the OpenCV project. As of recent versions of OpenCV, a wide range of object detection algorithms are available using the same programming interface. This provides a level of flexibility, and also allows for the Cyborg project to benefit from improvements done by the OpenCV project, simply by upgrading to new versions of OpenCV in the future.

The full implementation can be found in the attachments, but we will give a run-through of the steps taken. After the nodelet manager creates an instance of the class, the OnInit() function is run, which performs the steps normally found in the class constructor. In the case of the ObjectDetectorNodelet, shown in listing 5.4, this function fetches parameters

```cpp
#include <ros/ros.h>
#include <nodelet/loader.h>

int main(int argc, char** argv) {
    ros::init(argc, argv, "ros_dnn");

    nodelet::Loader nodelet;
    nodelet::M_string remap(ros::names::getRemappings());
    nodelet::V_string nargv;

    nodelet.load(ros::this_node::getName(),
            "zed_wrapper/ZEDWrapperNodelet",
            remap, nargv);

    nodelet.load(ros::this_node::getName(),
            "ros_dnn/ObjectDetectorNodelet",
            remap, nargv);

    ros::spin();

    return 0;
}
```

Listing 5.2: Nodelet instantiation using a C++ node.

```xml
<launch>
  <node pkg="nodelet" type="nodelet" name="ros_dnn"  args="manager"/>

  <node pkg="nodelet"
        type="nodelet"
        name="zed_wrapper"
        args="load zed_wrapper/ZEDWrapperNodelet ros_dnn"/>

  <node pkg="nodelet"
        type="nodelet"
        name="object_detector"
        args="load ros_dnn/ObjectDetectorNodelet ros_dnn"/>
</launch>
```

Listing 5.3: Nodelet instantiation using a launch file.

```
class ObjectDetectorNodelet: public nodelet::Nodelet {
    public:
        virtual void onInit();

    private:
        ros::NodeHandle nh;
        ros::NodeHandle nh_ns;

        /* Dynamic reconfigure */
        dynamic_reconfigure::Server<ros_dnn::ObjectDetectorConfig> server;
        dynamic_reconfigure::Server<ros_dnn::ObjectDetectorConfig>::CallbackType f;
        void dyn_reconf_cb(ros_dnn::ObjectDetectorConfig &config, uint32_t level);

        /* Neural network */
        cv::dnn::Net net;
        double conf_threshold;
        std::vector<std::string> class_labels;
        int frame_height;
        int frame_width;

        /* Draw a list of predictions on an image.
         * This adds a bounding box, a label and a confidence.
         */
        cv::Mat draw_predictions(
                std::vector<ros_dnn::Prediction> predictions,
                cv::Mat& frame) const;

        /* Pass an image through the neural net and return a list of predictions */
        std::vector<ros_dnn::Prediction> get_predictions(
                cv::Mat& frame,
                const cv::Mat& out,
                cv::dnn::Net& net) const;

        /* Publish/subscribe */
        image_transport::Subscriber sub_img;
        image_transport::Publisher pub_img;
        ros::Publisher pub_pred;

        /* Callback for receiving an image.
         * This comes in two flavors, with and without depth map. */
        void camera_cb(
                const sensor_msgs::ImageConstPtr& img,
                const sensor_msgs::ImageConstPtr& depth);
        void camera_cb(const sensor_msgs::ImageConstPtr& img);
};
```

Listing 5.4: ObjectDetectorNodelet class definition.

from the ROS parameter server, such as the names for the input and output topics as well as parameters required to load the neural network. This includes a configuration file describing layers of the network, a file with weights for the network as described in chapter 2, and so forth.

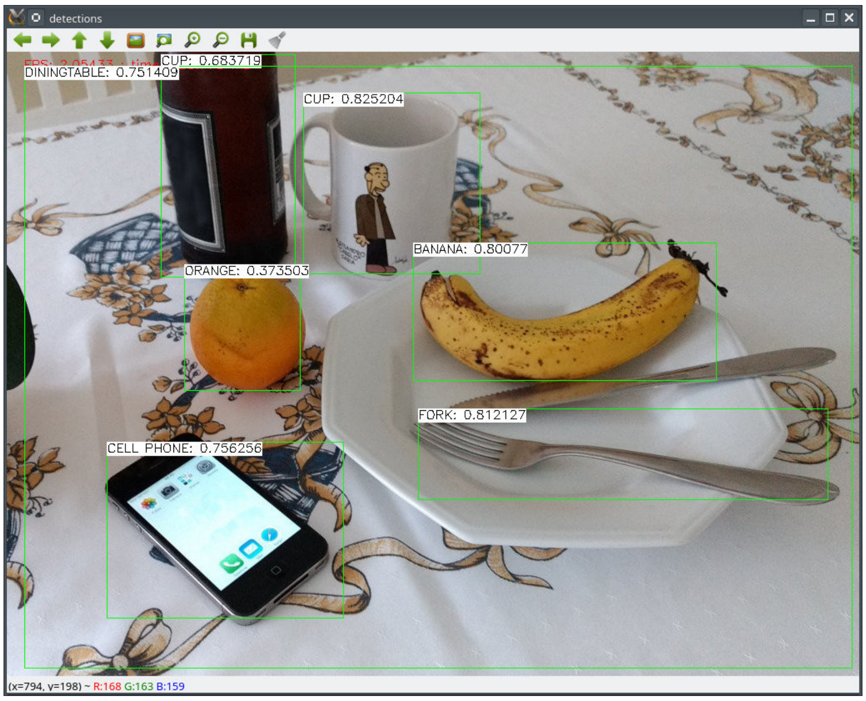


FIGURE 5.1: Screenshot of OpenCV object detection, courtesy of opencv.org.

Once all the required parameters are retrieved, the class loads the neural network and subscribes to the image and depth map topics published by zed_ros_wrapper. The image and depth map topics are subscribed to using a SubscriberFilter, which ensures that we receive both images in the same callback function, so that we can easily process them together. This also ensures that the received image and depth map have matching time stamps, by using the ExactSyncPolicy from the message_filters library. If the node is configured to run without subscribing to a depth map, this filtering step is not performed. Finally, once topic subscriptions are configured, the output topics created so that the node can publish the object detection results. We also set up dynamic reconfigure, so the user can change the classification threshold – the minimum confidence required for a detected object to be published – while the node is running.

On reception of an image, and optionally a depth map, the callback function *camera_cb()* is called. Here, we check if there are any subscribers listening to the output of the node, and abort if there are not. Then, the received image is converted to OpenCV format and fed through the neural network. The returned output from the network, for each detected

```cpp
class Prediction {
    public:
        Prediction(std::string label, int confidence, cv::Rect bounding_box)
            : label(label),
              confidence(confidence),
              bounding_box(bounding_box)
        {
        }

        /**
            * \brief Convert the prediction to a ros_dnn_msgs::Prediction message.
            */
        ros_dnn_msgs::Prediction to_prediction_msg() const;

        /**
            * \brief Get the distance from the camera to the prediction.
            */
        double get_distance(cv::Mat& depth_map) const;

        /**
            * \brief Draw a prediction on the provided frame.
            */
        void draw(cv::Mat& frame) const;

    private:
        std::string label;
        int confidence;
        cv::Rect bounding_box;
};
```

Listing 5.5: Prediction class definition.

object, includes the object class label, the label confidence and coordinates for a bounding box around the object. These are processed and a vector of Prediction objects, as shown in listing 5.5, are created to represent each detected object in the image. If the depth map is available, this can also be included.

Finally, the predictions are drawn on the image, in the same way as shown in fig. 5.1, and the image is published to the output image topic. The predictions are also published as a list, in the form shown in listing 5.1.

5.3.2 Python implementation

As described previously, the main advantage to using C++ over Python was believed to be in the ability to run the zed_ros_wrapper in the same process as the object detection algorithm. In order to test this hypothesis, it was necessary to also implement the same functionality as a Python node. This initially met a roadblock, as up until the new version of ROS released in May 2018 ROS used its own version of OpenCV. The version of OpenCV included with ROS has typically lagged the latest OpenCV release by several months. As the DNN functionality in OpenCV has been rewritten very recently, this meant that this functionality was unavailable in ROS until recently. For this reason, we initially implemented one version in Python without using OpenCV, and a second version after ROS released their new version.

Initial version

Initially, object detection was implemented in ROS using the PyYOLO library, which is a simple Python interface to the original C implementation of YOLO. Just as the C implementation, the performance of this implementation is significantly higher than the OpenCV implementation described above. This is hypothesized to be due to CUDA support in the C implementation, which is lacking in OpenCV.

The demonstration application created by [41] was modified to output the detection image and prediction list described earlier. The program flow in the Python version of the object detection node is generally similar to the C++ implementation, but implementing in Python yields significantly more compact code.

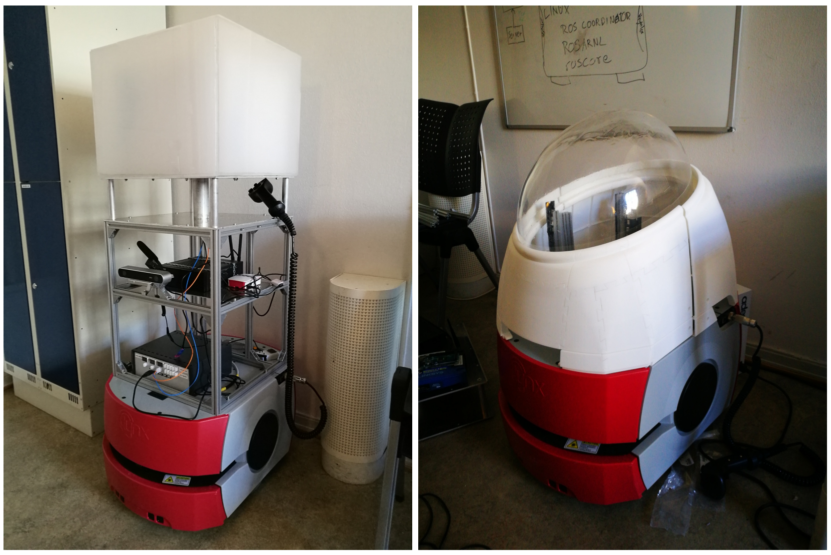
OpenCV version

With the ability to use a custom compiled version of OpenCV with ROS, it became possible to use a version with support for cutting edge features, including the Deep Neural Network (DNN) module. While OpenCV has CUDA support for much of its functionality, the DNN module is limited to GPU acceleration through OpenCL which currently only supports Intel GPUs. As the Cyborg project uses an Nvidia Jetson TX2 for graphics processing, this is a limitation. As GPU acceleration is key to achieving usable performance when running neural networks, backend support for this platform is key to making an OpenCV solution practical. Even so, this makes it possible to perform an apples to apples performance comparison between a C++ and a Python implementation using OpenCV, which is interesting in itself. Though this was only made possible shortly before the deadline for this report, it was interesting enough for the Cyborg project that this second Python implementation was created.

6 Other tasks

In addition to the major tasks presented previously, some smaller tasks have been performed in the process of writing this thesis. These tasks are described in this chapter.

6.1 Rack mounting system



(A) Before. (B) After.

FIGURE 6.1: The Cyborg, showing equipment mounting before and after installation of the 10 inch rack.

At the outset of the project, the Cyborg was equipped with a mounting frame, as shown in fig. 6.1a. A 10 inch rack mounting system was planned for this mounting frame, inspired by the use of 10 inch equipment enclosures used for home and small business networking equipment. An example of such an enclosure is shown in fig. 6.2a.

10 inch racks, or network enclosures, are a common equipment standard for small network equipment such as network patch panels, switches and so forth. By using standard dimensions for enclosure width and height, as well as standard placement for mounting holes, equipment can easily be swapped in and out of the rack. This modularity was found to simplify work on the Cyborg. Both the existing power supply and the enclosure for the Jetson

(A) Example of a 10 inch cabinet.

(B) Rack rails fitted on the Cyborg.

TX2 computer were mountable in the rack, while leaving ample space for one or two extra modules in the future. By fitting the Cyborg with four rack posts, as shown in fig. 6.2b, it is possible to mount the power supply facing backwards. This enables neater routing of power cables, as most computing and networking equipment have power connectors at the back. The Cyborg can be seen before and after installation of the new rack mounting system in fig. 6.1. The rack itself was constructed by the authors of [42], as part of their Experts in Team project, with guidance from the author of this thesis.

6.2 Convenience functionality for future ROS development

To support collection functionality used by all nodes in the Cyborg software environment, two nodes were created called *cyborg_types*, *cyborg_util* and *cyborg_nav*. These will be described in this section.

6.2.1 cyborg_types

Common data types that are useful in multiple parts of the project are included in this node, so that they may be easily imported where needed. Generally, these provide useful convenience functions for processing received ROS messages.

Point

The *Point* class represents a point in space. The data used to create a Point is generally received as a geometry_msgs/Pose or geometry_msgs/Pose2D message, both of which include a geometry_msgs/Point message from the navigation subsystem in ROS. In addition to exposing the x, y and z coordinates of the 3D point, the class also provides a convenience method for computing the Euclidean distance to a second Point.

Quaternion

The *Quaternion* class represents an orientation in space. The geometry_msgs/Pose message includes a geometry_msgs/Quaternion message, which contains the x, y, z and w coordinates used to instantiate the class. The class also includes a method for instantiating from a geometry_msgs/Pose2D message, which includes the orientation as an Euler angle.

Pose

The *Pose* class represents a point and orientation in space, represented by the previous two classes. In addition to providing class methods allowing the class to be instantiated from any of the Pose messages in geometry_msgs, the class also provides a convenience method for computing the Euclidean distance to a second Pose.

Path

A *Path* is an ordered collection of Poses, which typically represents a path between two locations in the world. In the Cyborg control system, it is used to keep an updated representation of the current path the Cyborg is moving along, so that progress can be tracked. To do this, it includes methods for adding and removing Poses from the Path, as well as advancing along the Path to a given Pose – so long as the given Pose is within a certain distance from a Pose along the Path. The class also contains a method to retrieve the total distance from the first to the last Pose along the Path.

6.2.2 cyborg_util

cyborg_util currently contains a single class, *Locations*. The Locations class retrieves a list of available goals from the *AvailableGoals* service, which is described below. The class acts as a persistent cache for the locations, exposing them as Pose objects.

6.2.3 cyborg_nav

For simplifying navigational tasks, a set of services was created that can be used by e.g. nodes in the behavior tree. These provide functionality which might be useful to multiple parts of the Cyborg software environment, and to simplify code reuse are created as individual services.

AvailableGoals service

For managing a set of known locations, and exposing these to other parts of the control system, a service was created called AvailableGoals. The service is shown in listing 6.1. The service is instantiated with a map file, which contains a list of named waypoints or goals. Upon request of the list of available goals, the service will parse this map, and create a response message containing the list. This list will be cached, so that it can be returned in the future without needing to recreate it.

DistanceToGoal service

For requesting the distance to a goal or waypoint measured from the current location of the Cyborg, a service was created called *DistanceToGoal*. As shown in listing 6.2, the service will receive a goal, from which a Pose object is created. The Pose class is included with the attached source code. Using the created Pose object, the service calls the *MakePlan* service, which is exposed by the ros_arnl node from MobileRobots, described in chapter 2. The DistanceToGoal service will then receive a plan, consisting of a list of positions, which show the path from the Cyborg to the desired final position. The list of positions is used to create a Path object. The Path class, which is also included in the attached source code, acts as a wrapper for this list of positions, and exposes useful functions for interacting with the list. The DistanceToGoal service will retrieve the length of the Path object, which is then returned to the user. The length is found by calculating the Euclidean distance between each pair of coordinates along the path.

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

import csv
import rospy
from collections import namedtuple
from itertools import ifilter, imap
from geometry_msgs.msg import Pose2D
from cyborg_msgs.msg import GoalWithHeading
from cyborg_msgs.srv import AvailableGoals, AvailableGoalsResponse

NAME = 'available_goals_server'

GoalRecord = namedtuple('GoalRecord', 'cairn type x y theta comment icon name')


class AvailableGoalsHandler():
    def __init__(self):
        rospy.init_node(NAME)

        srv_name = '/cyborg/nav/get_available_goals'
        rospy.Service(srv_name, AvailableGoals, self.goals_cb)

        # Will be created on demand
        self.response = None

        rospy.spin()

    def create_response(self, filename):
        """ Create AvailableGoalsResponse from map file """
        response = AvailableGoalsResponse()
        with open(filename) as f:
            # Get only lines with the keyword 'Goal' or 'GoalWithHeading'
            lines = ifilter(lambda line: 'Cairn: Goal' in line, f)

            # Parse lines, and create a GoalRecord, then create a
            # GoalWithHeading message for each waypoint
            reader = csv.reader(lines, delimiter=' ')
            for rec in imap(GoalRecord._make, reader):
                pose = Pose2D()
                pose.x = float(rec.x) / 1000
                pose.y = float(rec.y) / 1000
                pose.theta = float(rec.theta)

                goal = GoalWithHeading()
                goal.name = rec.name
                goal.comment = rec.comment
                goal.position = pose

                response.goals.append(goal)

        return response

    def goals_cb(self, data):
        """ Return list of available goals """
        if self.response is None:
            map_file = rospy.get_param('~map_file')
            self.response = self.create_response(map_file)
        return self.response


if __name__ == "__main__":
    rospy.set_param('~map_file', '/home/mortenmj/maps/maps/glassgarden.map')
    AvailableGoalsHandler()
```

Listing 6.1: Implementation of the AvailableGoals service.

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

from __future__ import print_function

import rospy
from cyborg_msgs.srv import DistanceToGoal, DistanceToGoalResponse
from cyborg_types import Path, Pose
from rosarnl.srv import MakePlan

NAME = 'distance_to_goal_server'


class DistanceToGoalHandler():
    def __init__(self):
        rospy.init_node(NAME)

        srv_name = '/cyborg/nav/get_distance_to_goal'
        rospy.Service(srv_name, DistanceToGoal, self.__distance_cb)

        srv_name = '/rosarnl_node/make_plan'
        rospy.wait_for_service(srv_name)
        try:
            self._plan_svc = rospy.ServiceProxy(srv_name, MakePlan)
        except rospy.ServiceException as e:
            rospy.logerr("Service call to %s failed: %s" % (srv_name, e))

        rospy.spin()

    def __distance_cb(self, data):
        # Create a Pose message to validate our input data
        goal = Pose.from_pose(data.goal)

        # Get a path to the given goal
        path = Path.from_posearray(self._plan_svc(goal).path)

        # Calculate distance if a path was found, or return inf
        distance = path.length if path else float('inf')

        return DistanceToGoalResponse(distance=distance)


if __name__ == "__main__":
    DistanceToGoalHandler()
```

Listing 6.2: Implementation of the DistanceToGoal service.

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

from __future__ import print_function

import rospy
from cyborg_msgs.srv import DistanceToGoal
from cyborg_msgs.srv import ClosestGoal, ClosestGoalResponse
from cyborg_util import Locations

NAME = 'closest_goal_server'


class ClosestGoalHandler():
    def __init__(self):
        rospy.init_node(NAME)

        # Distance to goal service proxy
        srv_name = '/cyborg/nav/get_distance_to_goal'
        rospy.wait_for_service(srv_name)
        try:
            self._dist_svc = rospy.ServiceProxy(srv_name, DistanceToGoal)
        except rospy.ServiceException as e:
            rospy.logerr("Service call to %s failed: %s" % (srv_name, e))

        # Closest goal service
        srv_name = '/cyborg/nav/get_closest_goal'
        rospy.Service(srv_name, ClosestGoal, self.goal_cb)

        self.locations = Locations()

        rospy.spin()

    # Get the closest point of interest
    def goal_cb(self, data):
        # Get the distances to each location.
        # If no path is found, the path will be empty
        dist = dict((k, self._dist_svc(v).distance) for k, v in self.locations)

        # Get entry with the shortest distance
        name, dist = min(dist.iteritems(), key=lambda p: p[1])

        return ClosestGoalResponse(name=name, distance=dist)


if __name__ == "__main__":
    ClosestGoalHandler()
```

Listing 6.3: Implementation of the ClosestGoal service.

ClosestGoal service

For requesting the closest goal, or waypoint, from the current location of the Cyborg, a service was created called *ClosestGoal*. Upon creation, this service instantiates a Location object. The Location class is included in the attached source code. When the Location class is instantiated, it will call the AvailableGoals service, and create a cache to hold the list of available goals. The ClosestGoal service iterates through this list, calling the DistanceToGoal service for each available goal, and return the goal with the shortest distance from the Cyborg.

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

import rospy
from rosarnl.srv import LoadMapFile


def load_map():
    filename = rospy.get_param('~map_file')

    rospy.wait_for_service('/rosarnl_node/load_map_file')
    s = rospy.ServiceProxy('/rosarnl_node/load_map_file', LoadMapFile)

    try:
        s(filename)
    except rospy.ServiceException as e:
        print("Service failed: %s" % e)


if __name__ == "__main__":
    rospy.set_param('~map_file', '/home/mortenmj/maps/maps/glassgarden.map')
    load_map()
```

LISTING 6.4: Implementation of the LoadMap service.

LoadMap service

The ros_arnl service LoadMap provides the ability to load a new map file to the Cyborg. Unfortunately, the service does not allow for automatically loading a given map on startup. In order to automatically load this map on startup, a simple service was created to call the ros_arnl LoadMap service. The Cyborg LoadMap service takes a single startup parameter, *map_file*, which is used to load the map using the ros_arnl LoadMap service. The service implementation is shown in listing 6.4.

7 Results and discussion

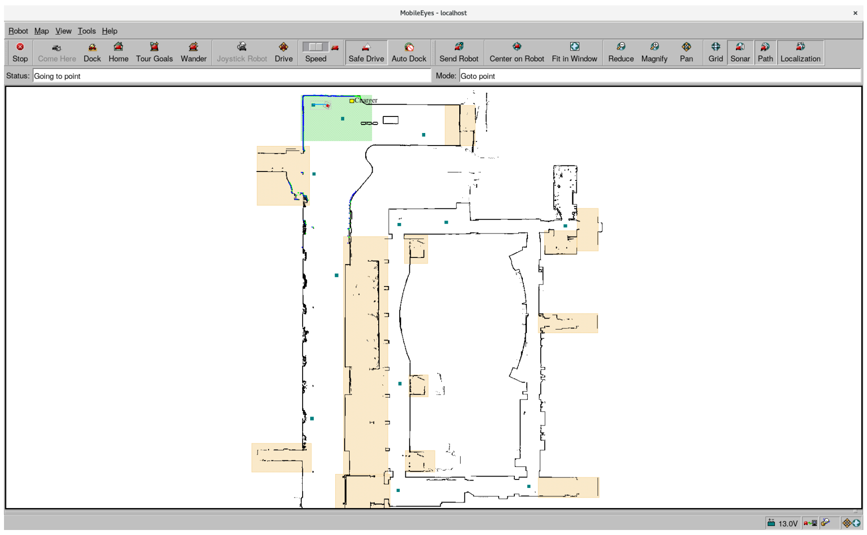
The objective of this thesis was to bring the Cyborg closer to a state where it could function as a mascot for the university. In order to do this, the Cyborg needed a robost and extensible control system as well as an awareness of the objects in the world around it. Here, the results of the work that has been done towards these goals, described in chapters 3, 4 and 5 is presented. The following also includes a discussion of the outcome of the work that has been done and how it may have been improved. Furthermore, some interesting avenues of future work are presented.

7.1 Behavior tree control system

A behavior tree control system was implemented using the behavio3 library, described in chapter 3. In addition to the main goals of being reliable and developer-friendly, the system was found to be easily scalable. In particular, the ability to create behavior trees using configuration files and load them using ROS launch files was found to be a significant advantage. The graphical editor behavior3editor, shown in fig. 3.2, simplified behavior tree creation further. Compared to the other approaches that were evaluated, where the behavior tree was specified in code, this was found to simplify the design process.

The control system was tested using the MobileSim and MobileEyes software applications from MobileRobots, described in chapter 2. A map of the university campus was loaded into the simulation software, and the Cyborg was tested in this environment. This can be seen in fig. 7.1. In combination with the behavior tree visualization application, this setup allowed for thorough testing of the control software.

It was found that behavior trees are well suited for splitting behavior into their smallest components, and composing more complex behaviors from sub-trees of these units. However, it was found that composing complex control systems from basic blocks can lead to behavior which can be hard to understand without the ability to visualize the flow of control in the system.



Status: Going to point

Mode: Goto point

FIGURE 7.1: Screenshot of the Cyborg running in simulation, shown in MobileEyes.

As described in chapter 2, a shortcoming in both finite state machines and behavior trees is the difficulty in representing alternative behavior in different modes, e.g. a low power mode and a normal mode. This is a known problem with both approaches, and it would be useful to investigate possible solutions, as it could lead to more life-like behavior in the future. As described by [33] there are two obvious ways to fix this, by combining finite state machines and behavior trees. Firstly, one could implement one behavior tree per mode and use a state machine to select which tree to run, or secondly one could implement each task in the behavior tree as a state machine.

In addition to the current composite nodes that exist in behavior3, it would be useful to have a form of parallel node. Such a node would make it possible to execute multiple branches at the same time, e.g. in order to monitor the condition of important status variables in one branch while running the Cyborg's behavior in another.

It would also be useful to implement services that can attach to composite nodes, that are run in the background when the composite node's branch is running. This would make it possible to retrieve status variables and write them to the blackboard, behind the scenes, which would simplify condition checking in the behavior tree nodes themselves.

Unfortunately, these features were not investigated further since they were beyond the scope of this thesis. As the Cyborg as a research platform is continuously subject to improvement and modification, this could be implemented in the course of future projects.

7.2 Control system monitoring application

The visualization software, described in chapter 4, was developed in order to simplify creation and debugging of the behavior trees. The software has shown itself to be a valuable diagnostic tool for this purpose. The end result can be seen in fig. 4.3 and fig. 4.4. The software performed well, and as part of the rqt framework it could be used in the same graphical interface as other useful plugins for rqt. This made it possible to create a custom monitoring application by selecting relevant rqt plugins and combining them into a complete monitoring application.

In the course of developing behavior trees for testing the control system, it was found that being able to visualize the control flow of the system greatly aided in the design process. On some occasions the behavior of the system was difficult to understand initially, but once visualized was simple to debug. It is believed to be useful to the project when increasingly complex behavior is implemented in the scope of future projects.

The initial idea of being able to force the system to go to a particular state, described in chapter 4, was dropped from consideration after consultation with co-advisor Martinius Knudsen. However, for the purpose of testing during the development of behavior trees, the possibility to enable and disable parts of the tree from from the visualization interface would have been interesting. For example, the user might click on a node to disable this node and any children below the node, and click again to re-enable. As each node and edge in the visualization is a Qt object it should be possible to execute callback functions when these are interacted with. Communication between the visualization interface and the behavior tree node would need to be planned thoroughly, making this a considerable undertaking.

7.3 Object detection visualization

The third objective of this thesis was to enable the Cyborg to detect and classify objects in the surrounding world, as described in chapter 5. Two parts of the image processing pipeline were considered. This was evaluated running the same functionality as both C++ and Python implementations. The C++ implementation was tested both running the modules as nodes, and as nodelets, as described in chapter 5.

Firstly, the time required for image transmission from the zed_ros_wrapper module to the object_detection module In comparing the execution times of the different implementations, some interesting results were found. Firstly, the difference when the C++ implementation was run in the nodelet configuration compared to the node configuration was found to be small. While the difference in runtime measured in percent was almost 20%, this amounts to a total difference of only 2.5 milliseconds. Furthermore, the image transmission step was found to be largely insignificant compared to the time it took to process the image by running it through the neural network. Even for the Python implementation, transmission time increases by only 30% compared to the nodelet implementation in C++ , and accounts for about 1% of the total processing time.

Task	C++ (nodes)	C++ (nodelets)	Python
Image transmission	13.24	15.75	17.08
Object detection	2147.48	2147.48	1524.02

TABLE 7.1: Execution times of object detection implementations, in milliseconds.

Secondly, as expected, the object detection took the same amount of time for both C++ implementations. However, the Python implementation took a shorter amount of time clocking in at only 70% of the runtime of the C++ implementations. This difference was confirmed not to be due to any differences in the pre- or post-processing done to the results, but rather be due to differences in OpenCV itself. Similar results have also been confirmed by others [43]. Other parts of the image processing, such as calculating the average depth of the object within the image, were performed using the same underlying libraries in both C++ and Python and as such achieve nearly identical performance.

It should be noted that these results are from running the neural network on a CPU, and that improved performance for the object detection phase should be expected when running on a Graphical Processing Unit (GPU). While OpenCV supports OpenCL as a backend for the DNN module, this is only available on Intel GPUs, while the Cyborg project uses an Nvidia GPU for image processing. Support for a CUDA backend for the DNN module is under active development, it is expected that this will greatly improve performance with minimal changes required to our software.

7.4 Other future work

A logical next step would be to combine the work described above. The behavior tree control system and the object detection system could be integrated, in order to make the Cyborg control system aware of objects in its surroundings. Making a control system that is aware of objects in the surrounding world would make it possible to enable advanced life-like behavior. The ros_arnl node that is used to communicate with the Cyborg does not provide the necessary low-level access to the map server that would be needed to place objects on the map. Placing detected objects on the map would allow for the path planning system to take them into account, thereby making this a useful area of research.

The object detection implementations using OpenCV are fully functioning, but would benefit from CUDA support in OpenCV. As shown by [41], CUDA support greatly accelerates object detection on Nvidia platforms such as the one used by the Cyborg. This would also be a useful avenue of research, as any contributions to OpenCV would benefit not only the Cyborg project but the larger computer vision research community.

8 Conclusion

In summary, the following tasks have been planned and implemented:

- a control system based on behavior trees
- a visualization application for the control system
- an object detection system using neural networks

Additionally, the author has guided and assisted EiT students contributing to the Cyborg. The outcome of the major tasks has been described and discussed.

The behavior tree control system functions well, and is easier to extend than the state machine system used previously. Both approaches have shortcomings in expressing multi-modal behavior. This is expected, and is described in chapter 2 and chapter 7.

The visualization application proved to be a useful debugging tool when creating behavior trees. Extending this software to provide more interaction with the behavior tree would be useful, although technically challenging.

The object detection algorithm works well, but would benefit from improved GPU support in the underlying library.

When developing software, it is easy to look back and criticize the architectural choices that have been made. During the development of this thesis, revisiting previous code and design choices has been a continuous process throughout the work that has been carried out. In the end this has been worth the effort, as it has led to robust new functionality implemented in the Cyborg, which yield considerable improvements to its control system.

Bibliography

[1] A. M. Knudsen, "NTNU Cyborg: A study into embodying Neuronal Cultures through Robotic Systems," *131*, 2016. [Online]. Available: https://brage.bibsys.no/xmlui/handle/11250/2414034.

[2] J. Waløen, "The NTNU Cyborg v2.0: The Presentable Cyborg," Norwegian University of Science and Technology, Tech. Rep., 2017. [Online]. Available: https://brage.bibsys.no/xmlui/handle/11250/2457153.

[3] MobileRobots, "Pioneer LX Mobile Research Platform," Tech. Rep. [Online]. Available: http://www.mobilerobots.com/PDFs/Pioneer%20LX%20datasheet.pdf.

[4] Stereolabs, *Stereolabs ZED Documentation*. [Online]. Available: https://docs.stereolabs.com/overview/getting-started/introduction/ (visited on 06/17/2018).

[5] Nvidia, "Jetson TX2 Developer Kit User Guide," Tech. Rep., 2017.

[6] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "ROS: an open-source Robot Operating System," in *ICRA Workshop on Open Source Software*, 2009.

[7] ROS, *Documentation - ROS Wiki*. [Online]. Available: http://wiki.ros.org/ (visited on 06/17/2018).

[8] MobileRobots, *ARIA Documentation*. [Online]. Available: http://robots.mobilerobots.com/wiki/ARIA (visited on 06/17/2018).

[9] ——, *ARNL Documentation*. [Online]. Available: http://robots.mobilerobots.com/wiki/ARNL/MOGS%7B%5C_%7DNavigation%7B%5C_%7Dand%7B%5C_%7DLocalization%7B%5C_%7DSoftware (visited on 06/17/2018).

[10] ——, *MobileEyes Documentation*. [Online]. Available: http://robots.mobilerobots.com/wiki/MobileEyes (visited on 06/21/2018).

[11] ——, *MobileSim Documentation*. [Online]. Available: http://robots.mobilerobots.com/wiki/MobileSim (visited on 06/21/2018).

[12] B. Gerkey, *Why ROS 2.0?* 2017. [Online]. Available: http://design.ros2.org/articles/why%7B%5C_%7Dros2.html (visited on 02/08/2018).

[13] B. D. Perry, "The memories of states: How the brain stores and retrieves traumatic experience," *Splintered reflections: Images of the body in trauma.*, pp. 9–38, 1999. [Online]. Available: `http://search.ebscohost.com/login.aspx?direct=true%7B%5C&%7Ddb=psyh%7B%5C&%7DAN=1999-02692-001%7B%5C&%7Dsite=ehost-live`.

[14] G. Cybenko, "Approximations by superpositions of sigmoidal functions," *Approximation Theory and its Applications*, 1989, ISSN: 10009221. DOI: `10.1007/BF02836480`.

[15] K. Hornik, "Approximation capabilities of multilayer feedforward networks," *Neural Networks*, vol. 4, no. 2, pp. 251–257, Jan. 1991, ISSN: 0893-6080. DOI: `10.1016/0893-6080(91)90009-T`. [Online]. Available: `https://www.sciencedirect.com/science/article/pii/089360809190009T?via%7B%5C%%7D3Dihub`.

[16] T. M. (M. Mitchell, *Machine Learning*. McGraw-Hill, 1997, p. 414, ISBN: 0070428077. [Online]. Available: `http://www.cs.cmu.edu/%7B~%7Dtom/mlbook.html`.

[17] Y. LeCun and C. Cortes, "MNIST handwritten digit database," *AT&T Labs [Online]. Available: http://yann. lecun. com/exdb/mnist*, 2010. [Online]. Available: `http://yann.lecun.com/exdb/mnist/`.

[18] Mathworks, *Convolutional Neural Network - MATLAB & Simulink*. [Online]. Available: `https://www.mathworks.com/content/mathworks/www/en/discovery/convolutional-neural-network.html` (visited on 04/07/2018).

[19] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going Deeper with Convolutions," *arXiv:1409.4842*, 2014, ISSN: 10636919. DOI: `10.1109/CVPR.2015.7298594`. arXiv: `1409.4842`. [Online]. Available: `https://www.cv-foundation.org/openaccess/content%7B%5C_%7Dcvpr%7B%5C_%7D2015/papers/Szegedy%7B%5C_%7DGoing%7B%5C_%7DDeeper%7B%5C_%7DWith%7B%5C_%7D2015%7B%5C_%7DCVPR%7B%5C_%7Dpaper.pdf%20https://arxiv.org/abs/1409.4842`.

[20] L. Brown, *GPU Accelerated Deep Learning for CUDNN V2*, 2015. [Online]. Available: `https://www.slideshare.net/NVIDIA/gpuaccelerated-deep-learning-for-cudnn-v2` (visited on 04/07/2018).

[21] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, 1943, ISSN: 00074985. DOI: `10.1007/BF02478259`. arXiv: `arXiv:1011.1669v3`. [Online]. Available: `http://www.cs.cmu.edu/%7B~%7D./epxing/Class/10715/reading/McCulloch.and.Pitts.pdf`.

[22] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," *Advances In Neural Information Processing Systems*, pp. 1–9, 2012, ISSN: 10495258. DOI: http://dx.doi.org/10.1016/j.protcy.2014.09.007. arXiv: 1102.0183. [Online]. Available: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[23] M. D. Zeiler and R. Fergus, "Visualizing and Understanding Convolutional Networks," Nov. 2013. arXiv: 1311.2901. [Online]. Available: http://arxiv.org/abs/1311.2901.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016, ISSN: 1664-1078. DOI: 10.1109/CVPR.2016.90. arXiv: 1512.03385. [Online]. Available: http://ieeexplore.ieee.org/document/7780459/.

[25] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," Nov. 2013. arXiv: 1311.2524. [Online]. Available: http://arxiv.org/abs/1311.2524.

[26] R. Girshick, "Fast R-CNN," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2015 Inter, pp. 1440–1448, Apr. 2015, ISSN: 15505499. DOI: 10.1109/ICCV.2015.169. arXiv: 1504.08083. [Online]. Available: http://arxiv.org/abs/1504.08083.

[27] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017, ISSN: 01628828. DOI: 10.1109/TPAMI.2016.2577031. arXiv: 1506.01497. [Online]. Available: http://arxiv.org/abs/1506.01497.

[28] K. He, G. Gkioxari, P. Dollar, and R. Girshick, "Mask R-CNN," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2017-Octob, pp. 2980–2988, Mar. 2017, ISSN: 15505499. DOI: 10.1109/ICCV.2017.322. arXiv: 1703.06870. [Online]. Available: http://arxiv.org/abs/1703.06870.

[29] Joseph Redmon, *Darknet: Open Source Neural Networks in C.* [Online]. Available: https://pjreddie.com/darknet/ (visited on 04/07/2018).

[30] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2015, ISSN: 01689002. DOI: 10.1109/CVPR.2016.91. arXiv: 1506.02640. [Online]. Available: http://arxiv.org/abs/1506.02640.

[31] NVIDIA, *NVIDIA Drive PX2 self-driving car platform visualized - YouTube*. [Online]. Available: https://www.youtube.com/watch?v=URmxzxYlmtg%7B%5C&%7Dt=9s (visited on 04/08/2018).

[32] T. R. Andersen, “Controller Module for the NTNU Cyborg,” 2017. [Online]. Available: https://brage.bibsys.no/xmlui/handle/11250/2441148.

[33] I. Millington and J. D. Funge, *Artificial intelligence for games*. Morgan Kaufmann/Elsevier, 2009, p. 870, ISBN: 9780123747310.

[34] J. Bohren, *SMACH Documentation*. [Online]. Available: http://wiki.ros.org/smach (visited on 06/17/2018).

[35] M. Colledanchise, R. Santomo, and P. Ögren, *behavior_tree - ROS Wiki*. [Online]. Available: http://wiki.ros.org/behavior%7B%5C_%7Dtree (visited on 06/16/2018).

[36] P. Goebel, *pi_trees - ROS Wiki*. [Online]. Available: http://wiki.ros.org/pi%7B%5C_%7Dtrees (visited on 06/16/2018).

[37] R. d. P. Pereira and P. M. Engel, “A Framework for Constrained and Adaptive Behavior-Based Agents,” Jun. 2015. arXiv: 1506.02312. [Online]. Available: http://arxiv.org/abs/1506.02312.

[38] Pi Robot, “Programming with Behavior Trees and ROS,” Tech. Rep. [Online]. Available: http://www.pirobot.org/ros/pi%7B%5C_%7Dtrees.pdf.

[39] M. Fowler, *TwoHardThings*. [Online]. Available: https://martinfowler.com/bliki/TwoHardThings.html (visited on 06/17/2018).

[40] R. Toris, J. Kammerl, D. V. Lu, J. Lee, O. C. Jenkins, S. Osentoski, M. Wills, and S. Chernova, “Robot Web Tools: Efficient messaging for cloud robotics,” *IEEE International Conference on Intelligent Robots and Systems*, vol. 2015-Decem, pp. 4530–4537, 2015, ISSN: 21530866. DOI: 10.1109/IROS.2015.7354021. [Online]. Available: http://robotwebtools.org/pdf/paper.pdf.

[41] T. Opheim, F. Vatsendvik, A. Moltumyr, and E. Henriksen, “Prosjektrapport - Gruppe 4B,” Tech. Rep., 2018. [Online]. Available: https://ntnu.app.box.com/v/robot/file/294980938949.

[42] A. Johansen, M. Lervik, A. Moan, T. Myrvang, A. Johannessen, and F. Samdal Solberg, “Prosjektrapport - Gruppe 1B,” Tech. Rep., 2018. [Online]. Available: https://ntnu.app.box.com/v/robot/file/294979313498.

[43] D. Liu, *OpenCV DNN speed compare in Python, C#, C++*. [Online]. Available: http://www.died.tw/2017/11/opencv-dnn-speed-compare-in-python-c-c.html (visited on 06/11/2018).