Contents

List of Abbreviations

AI	Artificial Intelligence
API	Application Programming Interface
CAD	Computer-Aided Design
CNN	Convolutional Neural Network
DoG	Difference of Gaussians
DPM	Deformable Part Models
FPS	Frames Per Second
GPU	Graphics Processing Unit
HOG	Histogram of Oriented Gradients
HSL	Hue, Saturation, Lightness
HSV	Hue, Saturation, Value
IO	Input/Output
IOU	Intersection Over Union
IR	Infrared
LDF	Linear Discriminant Function
Li-Po	Lithium polymer
LTS	Long Term Support
NCS	Neural Compute Stick
NN	Neural Network
R-CNN	Region-based Convolutional Neural Network
RANSAC	Random sample consensus
RGB	Red, Green, Blue

ROI	Region Of Interest
ROS	Robotic Operation System
SDK	Software Development Kit
SIFT	Scal-invariant feature transform
SoC	System on a chip
SVM	Support vector machines
TDNN	Time Delay Neural Network
USB	Universal Serial Bus
VPU	Vision Processing Unit
YOLO	You Only Look Once

Notation

Latin Alphabet

$C_{1,2}$	arbitrary linear discriminant function	1
$C_{1,3}$	arbitrary linear discriminant function	1
$C_{2,3}$	arbitrary linear discriminant function	1
g	magnitude of gradient	1
g_x	vertical gradient	1
g_y	horizontal gradient	1
$H(i,j)$	arbitrary function	1
$I'(u,v)$	output gray-scale image	1
$I(u,v)$	input gray-scale image	1
x_1	Cartesian coordinate	1
$x_2(x_1)$	arbitrary linear discriminant function	1

all detections	number of all detections	1
all ground truths	number of all ground truths	1
Area of intersection	area of intersection	$pixel^2$
Area of union	area of union	$pixel^2$
Averaged latency	averaged latency	s
Est. FPS	estimated performance	FPS
FN	number of false negative cases	1
FP	number of false positive cases	1
IoU	intersection over union	1
IOU^{truth}_{pred}	box accuracy	1
Performance	performance	FPS
Precision	precision	1
Processed frames	number of processed frames	1
Pr(Class$_i$|Object)	conditional class probability	1
Pr(Class$_i$)	class-specific confidence score	1
Pr(Object)	probability box conatins object	1
Recall	recall	1
System uptime	system uptime	s
TP	number of true positive cases	1

Greek Alphabet

β	threshold weight	1
β_0	weight vector	1
θ	direction of gradient	rad

List of Figures

List of Tables

Listings

Abstract

This project will consider different approaches for object detection on the Turtlebot3 platform. Identifying objects with the specific platform widens the scope of operations the system can fulfill and enables it to react and interact with its environment. Due to its modular design it can be equipped with a camera. Therefore, objects should be identified by visual analysis. Methods for detecting objects based on visual data should be implemented in the most general way and for evaluation the following should be considered: a local Neural Network, another running on specialized hardware, outsourcing computation to the cloud and a classical machine learning approach. These will be compared based on their performance and detection results.

Kurzzusammenfassung

Dieses Projekt beschäftigt sich mit Methoden zur Objekterkennung auf der Turtlebot3 Plattform. Die Fähigkeit, Objekte zuerkennen, würde die Möglichkeiten des Systems beträchtlich erweitern. Insbesondere die Reaktion und Interaktion des Systems mit seiner Umgebung würde von dieser Möglichkeit profitieren. Da die modulare Plattform leicht mit einem Kamera-Modul erweitert werden kann, soll die Objekterkennung auf visuellen Informationen basieren. Methoden zur Erkennung sollen möglichst allgemein implementiert werden und zur Beurteilung wurden folgende Ansätze ausgewählt: ein lokales neuronales Netzwerk, ein Weiteres auf spezialisierter Hardware, ein cloud-basierter und ein klassischer Maschinen-Lern-Ansatz.

1 Introduction

Autonomous robots have a great variation of tasks. An autonomous system should be capable to behave autonomously and purposeful in the real world, described by [CB99]. The primary prerequisite for interaction of a system with its environment is that relevant objects placed within the spatial scope of operation should be recognized. The reliability of recognition is highly use-case-dependent, e.g. for detecting obstacles with an autonomous car a distinction between real objects on the road and symbols drawn at it can be sufficient. Determining the nature of an object can be a difficult task and the most intuitive solution for this problem seems to be utilizing optical sensors. Furthermore, objects can differ in various ways while sharing the same nature. For example, a blue and a red bucket are still buckets while have distinctive differences in color, or two chairs can be built rather differently, but still share the same nature of being a chair. Expanding the meaning of interacting with the environment some beneficial and purposeful tasks can be created. Examples for those objectives are observation, exploration and recovery, fulfilled by projects like REMUS[1] or Bat[2]. These can be seen in a more military context. As well applications in civilian life exist. In most objectives additional information about the environment are helpful to estimate threats or even improve cooperation by recognizing other systems. Additionally, it could enable the system to transport specific objects in its environment.

As previously described the scope of applications for detecting specific objects is widely spread. Therefore, it has to be limited to a specific system and task. An exemplary task would be detecting a finite set of objects in a labyrinth while moving through it. This environment can be easily setup and reproduced in a laboratory. Additionally, the task simulates a real operation quite well, while offering predictable results and a simple evaluation. As system the Turtlebot3 platform is chosen due to its ability to fulfill such an operation, while being inexpensive and easy to use.

Consequently the goal of this project is set to determine an acceptable method to detect objects with the Turtlebot3 platform using a camera module. This would enable the Turtlebot3 to generate more information about its environment and improve its interaction with it. Additionally, introducing object detection would widen the variety of tasks the system can fulfill. The given hardware limits the range of methods drastically considering their local performance needs. Outsourcing heavy workloads to specialized hardware or to a remote service should be considered while concerning the perhaps lower runtime limited by battery capacity. To achieve this goal, criteria for an object detection method for the Turtlebot3 Waffle will be determined. Afterwards, available methods will be checked whether they suit the platform. Suitable object detection methods will be implemented into ROS. Lastly, implemented methods will be evaluated and compared based on similar inputs with previously determined criteria.

[1] a series of autonomous underwater vehicles [Sto01]
[2] unmanned air vehicle designed for the US Army [Gru11]

2 Literature review

The following section considers existing approaches for object recognition and detection on specific systems. It describes the used systems and summarizes each method including its results. The methods are chosen due possible adaptation to this project and relevant of aspects each method are emphasized. The main tasks is to extract simple and light, in context to their resource demand, approaches for object detection under heavy limitation of the Turtlebot3 platform with the best possible performance.

2.1 Object recognition on the Turtlebot2

The goal of the project described in [Hof17] is the development of a modular framework for 2D and 3D object recognition and the creation of an adaptable navigation program. According to [Hof17] the faculty for mechanical engineering of the Westphalian University of Applied Sciences worked on implementing object recognition algorithms on the predecessor of the Turtlebot3 platform, the Turtlebot2 platform. The specific system they used was a Turtlebot2 equipped with a Kobuki differential-drive mobile base and an Astra 3D sensor, which should enable the system to record 2D color images and 3D point clouds. The system was operated with ROS Kinetic on top of Ubuntu 16.04. It utilizes three specific object recognition systems, HSV detection, feature detection and shape detection.

The idea of the HSV detection system is to identify an object based on noticeable colors. Due to the strong effect of overall brightness on RGB values in an image a method for easier filtering is to transform the image pixel-wise into a set of hue, saturation and value. After the transformation lower and upper limits for each channel are determined and only pixels within these boundaries are considered. This leads to a binary one-channel image where pixels only fulfill the conditions or don't. The resulting image reveals some noise within it which can be removed by opening and closing in terms of mathematical morphology. Opening describes erosion followed by dilation and leads to the removal of fragments from the image. Opposing to that, closing means dilation followed by erosion transformations and removes small holes. Based on the contours of the transformed one-channel image compared with an reference image an object might be recognized.

The feature detection system is supposed to identify objects using key-points and descriptors. A key-point is defined as a part of an image that contains a lot of local information. Whereas a descriptor holds information about a key-point in a way that described key-points are comparable. The main idea of this system is to simplify an object that it has similar features in different views by utilizing robust and invariant descriptors. Matching descriptors requires that the matcher is trained with a set of training descriptors of a known object. There are different algorithms available to match descriptors. This project uses a brute force method for matching descriptors. That means it compares all descriptors with each other. If enough matches are found, the system assumes that the test image contains an object with the similar features as the training image and therefore shows the same object.

The shape detection system identifies objects based on their similarity to common shapes. A typical example for this detection method is to recognize a mug based on its cylindrical form. RANSAC[3] (random sample consensus) is used to estimate a mathematical model from a set of data points containing outliers given by the Astra 3D sensor. The resulting estimated model is compared with an expected model given by the user.

The project shows that HSV, feature and shape detection are applicable on low resource platforms, but these methods require a specific use-cases, which is primarily a single object with the following characteristics on an easily distinguishable background. HSV detection requires a high color difference between object and background and between different objects. Feature detection depends on clear object-specific patterns. Shape detection necessitates similarities between real object and a model. Therefore, all three methods should be usable for object detection on the Turtlebot3 platform. The success rate of each depends highly on its configuration which is based on the objects. This project is primarily aimed to detect one object and tune each method specifically to this object. In conclusion, it is not directly suited for detecting multiple objects, but shows an interesting approach for low-resource object detection.

2.2 General Object Detection Method by On-Board Computer Vision with Artificial Neural Networks

Another approach focused on real-time application aims to find a solution based on object detection for a collision avoidance system. The approach described in [VI17] should be able to detect obstacles in front of a car and a critical factor is to detect so called fake obstacles like text and signs on the road. This leads to the need of recognizing different features of objects and differentiating between two-dimensional and three-dimensional obstacles. According to the paper, a highly effective feature extraction method is the Histograms of Oriented Gradients (HOG) method presented by [McC86]. This method was originally used for detecting humans based on the features of shape not considering size or color of the object. It divides an image into parts and counts the occurrences of gradient orientation in each part. The image can be divided into rectangles or into circles. The number of gradient orientations is stored in a histogram.

Due to different shape variation ratio of real and fake obstacles, when a vehicle approaches them, it is possible to differentiate them. Therefore, a Time Delay Neural Network (TDNN) is designed to learn the differences between two-dimensional and three-dimensional objects and classify these. TDNN describes an architecture which is specialized to work with sequential data. It should recognize features of time-shifting and consists of a large pattern recognition system. Commonly the TDNN concept is known in speech-recognitions tasks. The main feature of the TDNN is its ability to express relations between time-delayed inputs and it is structured as a feed-forward network.

In their proposed method [VI17] collected 150 samples of obstacles, where 90 are real obstacles and 60 are fake ones. The process is divided into three parts, preprocessing, feature extraction and classification. The preprocessing phase should prepare data recorded by the front-facing camera

[3] widely used robust estimator for mathematical models in vision problems [CM08]

of the vehicle for further computation. The brightness and contrast of the images is adjusted and a Gaussian blur is applied to remove noise. Later the image is converted from RGB to gray scale and reduced to the region of interest (ROI). Finally, the Canny edge detector method[4] is applied to find edges of objects in the image. They used the HOG method described in section 3.5 to extract features from video images and for real-time obstacle detection TDNN. The method showed after testing an accuracy for object classification of 96.67% with a false positive error of 3.33% and is assessed as not suitable for an automatic braking system due to the possibility of a sudden emergency braking in case of a false positive detection and a vehicle following close by crashing into the first one. Therefore, it is only considered as a warning system.

Hence, [VI17] offers a viable approach for object detection. Although a differentiation between real and fake obstacle is not necessarily needed for this project. Therefore, utilizing a TDNN would create irrelevant bulk. Even though, extracting features from a training data set and comparing these with testing data seems to be an interesting way to identify objects on the Turtlebot3 platform locally.

2.3 You Only Look Once: Unified, Real-Time Object Detection

A new approach for object detection is offered by [Red16] with You Only Look Once (YOLO). Instead of repurposing a classification method for detecting objects, it utilizes a single neural network to predict bounding boxes and class probabilities directly from an image within a single evaluation. Therefore, it promises being extremely fast and to outperform other detection methods, like Deformable Part Models (DPM) and Regional Convolutional Neural Network (R-CNN), described in section 3.4. YOLO demands less computation power and memory than these methods, but requires a decent GPU, e.g. a Nvidia Titan X, to generate real-time predictions. Downside of its approach is a higher localization error rate but less likely resulting in false positives on background. The network architecture of YOLO is inspired by the GoogLeNet model for image classification [Sze15] and consists of 24 convolutional layers followed by 2 fully connected ones, shown in figure 2.1.

The YOLO design works as shown in figure 2.2. The input image is divided into a $S \cdot S$ grid. Each grid cell is responsible for detecting the object which center is positioned in the cell and consequently, it predicts bounding boxes and according confidence scores. This results in five predictions for each bounding box, which are the (x,y) coordinates of the center, relative to the grid call, the width and height, relative to the image, and a confidence score. Additionally, each grid cell predicts a class probability, conditional on the cell containing an object. Equation 2.1 shows how the class-specific confidence scores $\text{Pr}(\text{Class}_i)$ are computed by the conditional class probabilities $\text{Pr}(\text{Class}_i|\text{Object})$ and the individual box confidence predictions defined as $\text{Pr}(\text{Object}) \cdot \text{IOU}^{\text{truth}}_{\text{pred}}$, where $\text{Pr}(\text{Object})$ is the probability that the box contains an object and $\text{IOU}^{\text{truth}}_{\text{pred}}$ is the accuracy of that box.

$$\text{Pr}(\text{Class}_i|\text{Object}) \cdot \text{Pr}(\text{Object}) \cdot \text{IOU}^{\text{truth}}_{\text{pred}} = \text{Pr}(\text{Class}_i) \cdot \text{IOU}^{\text{truth}}_{\text{pred}} \tag{2.1}$$

[4] multi-staged method to detect edges in images [Can86]

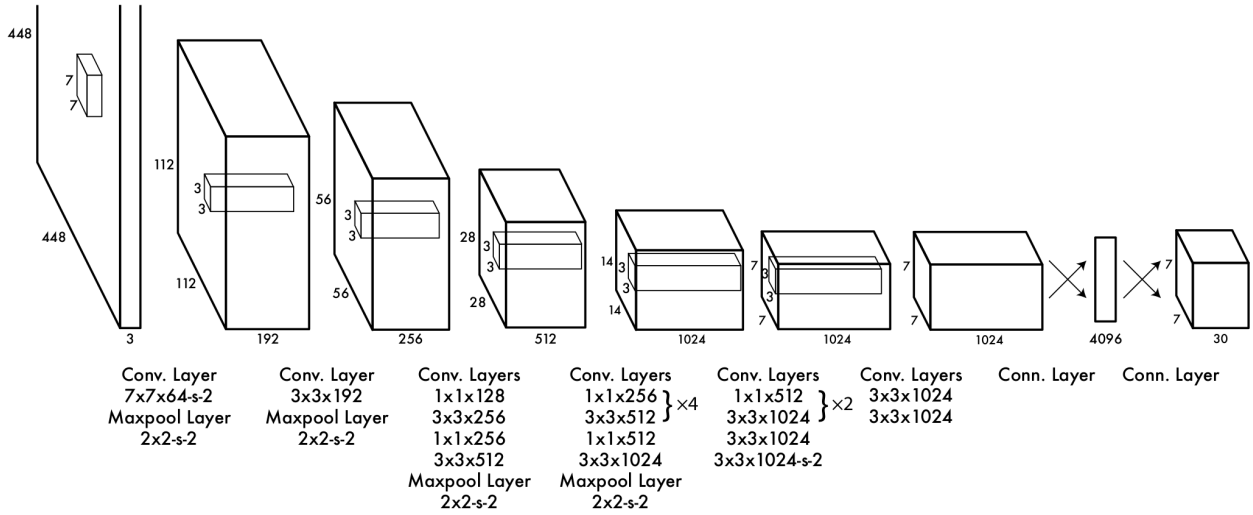

Figure 2.1: Layer graph of YOLO model [Red16]

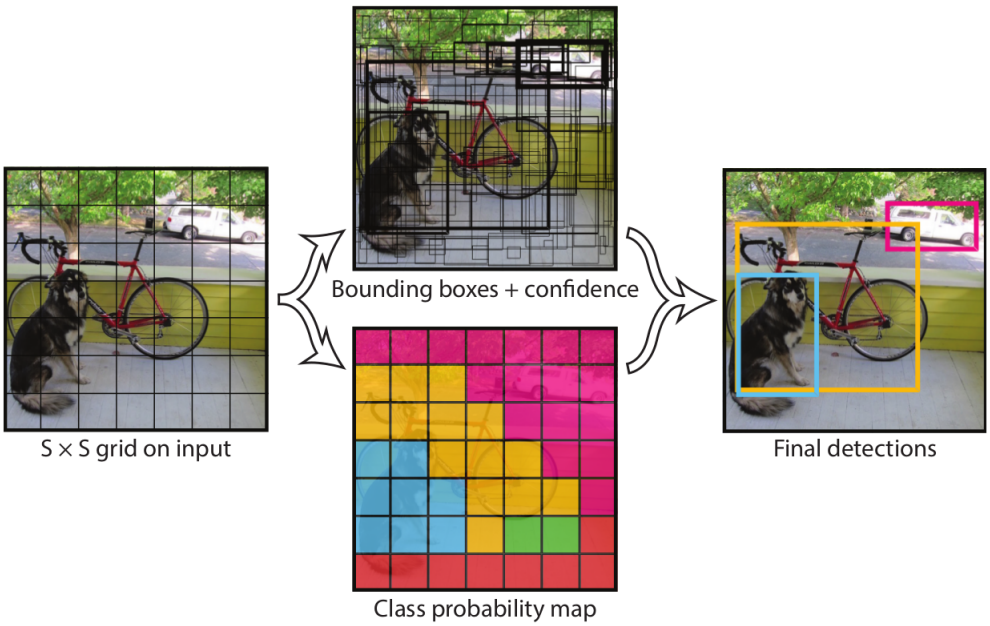

Figure 2.2: Visualization of Darknet YOLO model [Red16]

Due to the high resource demand of YOLO, which results in the necessity of computing on a GPU to gain real-time performance according to [Red16], it is not feasible to run on a SoC like it is used on the Turtlebot3 platform, but an in complexity reduced version of YOLO called *Tiny YOLO* is available. This variation seems to match the limitations of the platform, further described in section 4.1.

2.4 Cloud-based robot grasping with the google object recognition engine

The trend of steadily increasing transfer speeds has allowed many enterprises to bring their services to the cloud for a better scalability and availability. In the same sense [Keh13] considers a cloud-based data and computation approach for 3D robot grasping. It offers a prototype based

on a Willow Garage PR2 robot with on-board color and depth cameras using Google's proprietary object recognition engine.

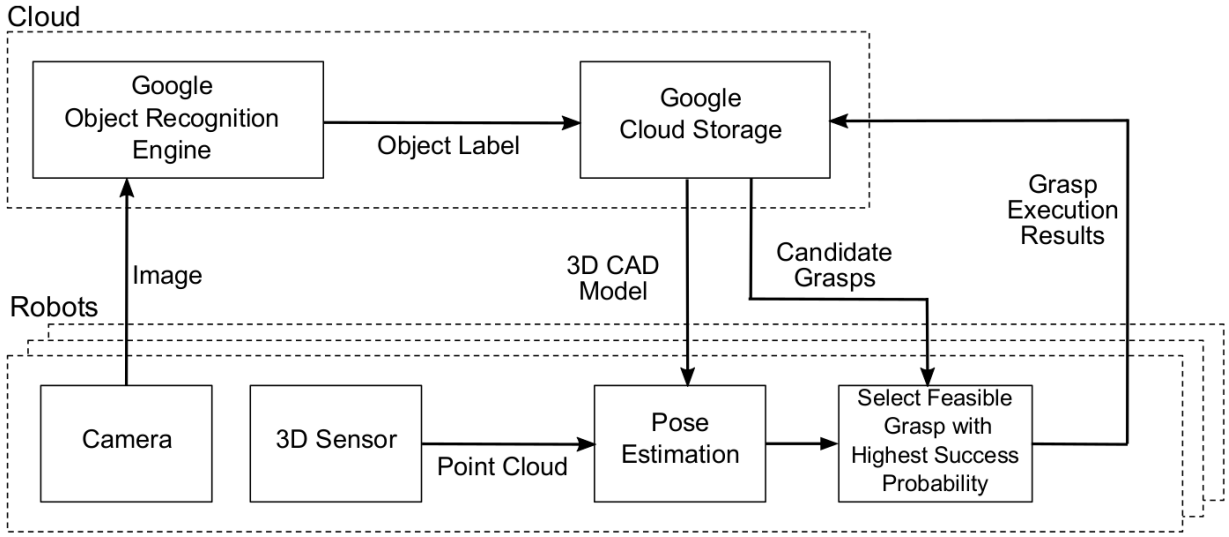

Figure 2.3: Model for cloud-based robot grasping with the google object recognition [Keh13]

As shown in figure 2.3, the robot takes a picture of an object and sends it via Internet to Google's Object Recognition Server. If the engine identifies an object, it forwards the object label to a cloud storage bucket which responds to the robot with a matching 3D CAD model and possible grasp candidates. Using the model and the point cloud from the depth camera the position of the object is estimated locally. Lastly, the grasp candidate with the highest success probability is chosen and performed. After execution the result is assessed and stored in the cloud storage bucket. This approach claims a high object recognition rate with a recall rate of 80% in their test set. Additionally, it results in a pose estimation failure rate of under 14% and a grasping failure rate of under 23%. These results seem to be quite low, but grasping specific objects properly is a high complex task.

The described approach seems promising as it outsources the heavy workload of object recognition to Google's cloud services. This offers high computation power while requiring an Internet connection constantly. Although, [Keh13] can not be directly ported to be used in this project, because it is only capable to classify objects in the cloud and requires CAD models of all items to localize them. These are not available for this project and have to be created with a lot of effort. Therefore, adapting its concept and creating a suitable approach is required. The opportunity of outsourcing computation workloads would enable the Turtlebot3 platform to utilize resource-heavy methods to detect objects, which were not be able to run locally. Additionally, [Keh13] integrates perfectly into the ROS philosophy, described in section 3.2. Furthermore, it uses asynchronous messaging, like ROS nodes do while communicating over topics.

3 Basics

In the following chapter basics required for this project are explained. It will consider the Turtlebot3 platform as target system. Furthermore, the Robot Operation System (ROS) driving Turtlebot3 is introduced. Additionally, methods used for object classification and detection in images will be described.

3.1 Turtlebot3

According to [Bra18] the Turtlebot series is designed as standard platform robots for learning and prototyping. They are driven by Robot Operating System (ROS), which is further described in section 3.2. The series finds its origin in the *Turtle* robot driven by the educational programming language Logo in 1967. Logo [Pea83] is a general purpose language and widely known for its turtle graphics. Figure 3.1 shows the current generation of the Turtlebot3 series with three models available. The smallest model, the *Burger*, is not in the scope of the project because it is not available with a camera, although it could be equipped with one if needed. Both bigger models *Waffle* and *Waffle Pi* are equipped with a camera. In fact the *Waffle* is provided with an Intel® Realsense™R200 and the *Waffle Pi* with a Raspberry Pi Camera Module v2.1.

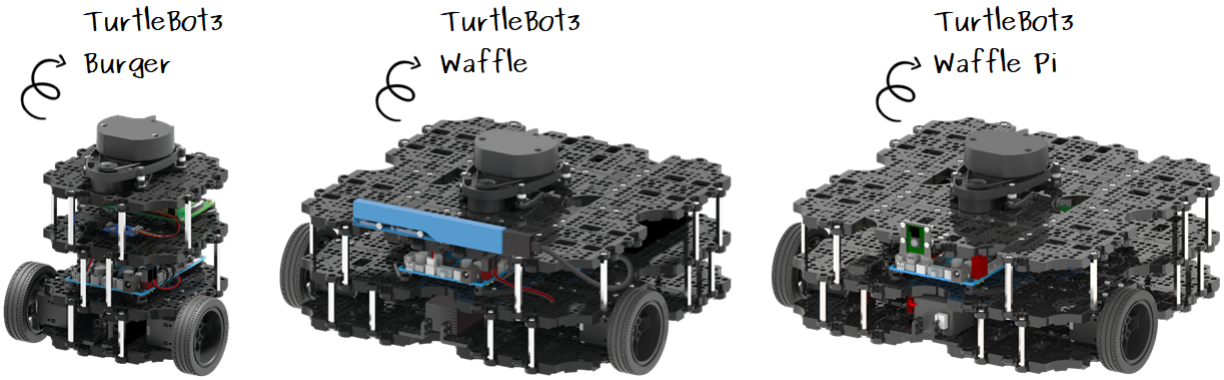

Figure 3.1: Turtlebot3 models [Rob18]

The Turtlebot3 is, according to [Bra18], designed as a small, affordable, programmable and ROS-based robot platform for educational, research, hobby and product prototyping uses. It is designed to minimize the size of the robot platform and reduce costs without sacrificing functionality and quality by adapting a modular concept and using established components. Due to availability in the faculty the Turtlebot3 *Waffle* is used in this project although it is discontinued.

As shown in figure 3.2 the Turtlebot3 *Waffle* is a three level platform robot impelled by two electric drives directly connected to one rubber tire each. Additionally, it is equipped with two omni-directional steel balls on the bottom side to ensure stability. As main processing board it is delivered with the Intel® Joule™570x and for lower level tasks with an OpenCR board. The Joule can be operated by basically any Linux system, but an Ubuntu 16.04 LTS *Xenial Xerus* is recommended by the manufacturer, Robotis. Its visual sensor is an Intel® Realsense™R200, which is capable of recording RGB images with a maximum resolution of 1920 by 1280 pixels and 30 FPS. Moreover

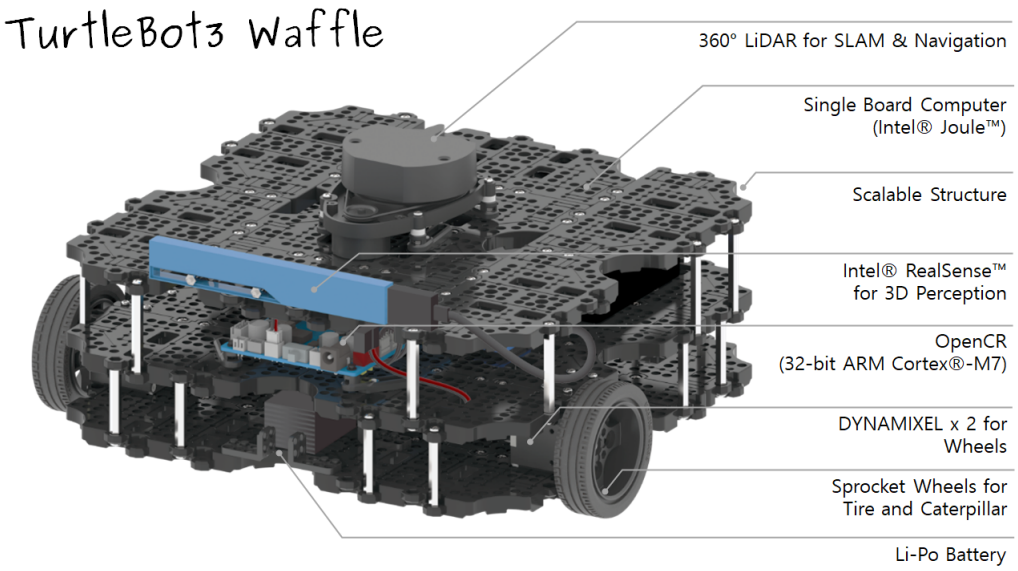

Figure 3.2: Turtlebot3 *Waffle* [Rob18]

the R200 can take Infrared (IR) images with a maximum resolution of 640 by 480 pixels with 60 FPS for depth recordings. The common approach for detecting depths described by [And12] is to project a dot-grid in the IR-spectrum, recording the projected grid and calculating depths from distances of the dots. Additionally, the *Waffle* is equipped with a 360 Laser Distance Sensor LDS-01. For mobility it is delivered with a 11.1V 1800 mAh lithium polymer (Li-Po) battery.

3.2 Robot Operating System

Robot Operating System (ROS) is described as an open-source robot operating system by [Qui09], but not as an operating system in its traditional sense for process management and scheduling. It offers a structured communications layer on top of the host operating system and is structured like a peer-to-peer network of processes. ROS refers these processes as nodes. Nodes have a variety of options to communicate with each other, e.g. synchronous message sending over services, asynchronous data streaming over topics and storing data on parameter servers, which can be described as shared dictionaries accessible via API calls. Nevertheless, ROS is not a real-time framework, but offers some functions for real-time operations.

The design goals of ROS are described by [Qui09] as follows. First and foremost, it should offer the possibility to handle a number of processes, which are probably distributed over different hosts and therefore, are structured as a peer-to-peer topology. Additionally, this should also prevent from bottlenecking due to a slow wireless link to a central server. To enable processes to find each other at runtime it is necessary to introduce a lookup mechanism. This essential node is called master and all nodes have to register at it. Second, ROS should be multi-lingual. That means it is designed language-neutral in way the user can access it with his preferred programming language. At the current state ROS supports C++, Python, Octave and LISP. Third, ROS should be tool-based. So it is not necessary to build and run ROS components in a massive development and runtime environment, instead due to its microkernel design it is possible to choose needed tools from a variety of small tools depending on the component the user aims to build and run.

Fourth, ROS is structured as a thin framework with drivers and algorithms usable even outside the project. Dependencies of components will be reduced as much as possible to ensure easier code extraction, testing and reuse. Lastly, ROS is free and open-source. It is distributed with a BSD license which permits use and development in non-commercial and commercial contexts. This should encourage every user to participate in the ROS project.

3.3 Intel® Movidius™Neural Compute Stick

The Movidius Neural Compute Stick (NCS), shown in figure 3.3, is a fanless deep learning device in the form factor of a common USB stick sold by Intel. According to [Int18b] it is powered by the Intel® Movidius™Myriad™2 Vision Processing Unit (VPU), which is used in millions of security cameras, gesture controlled drones and industrial machine vision equipment. The NCS should enable the user to develop and use AI applications without requiring a network connection to the cloud. The built-in low-power VPU offers in combination with the Intel Movidius Software Development Kit (SDK) the ability to rapidly profile, tune and deploy convolutional neural networks (CNN) easily and with low-power demands.

Figure 3.3: Intel® Movidius™Neural Compute Stick [Int18b]

Using the NCS requires USB 2.0 connectivity although USB 3.0 is recommended. Furthermore, [Int18b] specifies a x86 64-bit computer running Ubuntu 16.04, Raspberry Pi 3 Model B running Stretch desktop or Ubuntu 16.04 virtual box instance as host system, fitting the Turtlebot3 lineup perfectly. After installing the SDK, it is necessary to compile a pre-trained Caffe or TensorFlow model of the network. This can be done on a different, more powerful system as the target one. To deploy the NCS, it can be plugged into the target system and accessed via the API delivered by the SDK as Python and C library. Example projects using these libraries are widely available and can be easily adjusted. Furthermore, the available libraries could be easily used with ROS due to its multi-lingual design, described in section 3.2.

3.4 Neural networks

To understand neural networks (NN) it is necessary to explain the idea of perceptrons because NNs consist of multiple perceptrons. A perceptron is an artificial model of a neuron in the human

brain. According to [Ros58] a perceptron can have multiple inputs and one output. All inputs are weighted and combined in a transfer function. The result of the transfer function is mapped through an activation function to the output of the perceptron. It can be trained by adjusting the weights. [Ros58]

To create a NN it is necessary to connect at least two neurons in any way. Most NN are built in a specific structure depending on the use-case. Commonly NN are divided into layers, which can be of an input, output or hidden type. Usually the user interacts only with the input and output layers. Characteristic structures are feed-forward NN and recursive NN. Characteristic structures for NN are the feed-forward and recurrent type. Where feed-forward NN have only in one direction connected neurons and recurrent NN have neurons, which are either connected to a lower level neuron or to themselves. More relevant for object classification and detection are feed-forward NN. [Goo16]

An important class of feed-forward NN are Convolutional Neural Networks (CNN) They are specialized on two-dimensionally arranged input data like images. In contrast to traditional feed-forward NN not all neurons in a CNN are directly connected to each neuron of the next layer. This increases the efficiency of the network in the mentioned use-case. A CNN consists in a classical way out of these parts. The first one is responsible to extract features from the input data and the second one classifies on object based on the input features. The first part mostly consists of convolutional and pooling layers, where the second part consists of fully-connected layers.

Convolutional layers basically apply a convolution operation to the input $I(u,v)$ with the arbitrary function $H(i,j)$ and pass the result $I'(u,v)$ to the next layer, shown in figure 3.4. The convolution is essentially defined by the filter size, which determines the size of the convoluted region, and the step size, which specifies how much the filter should move after every operation.

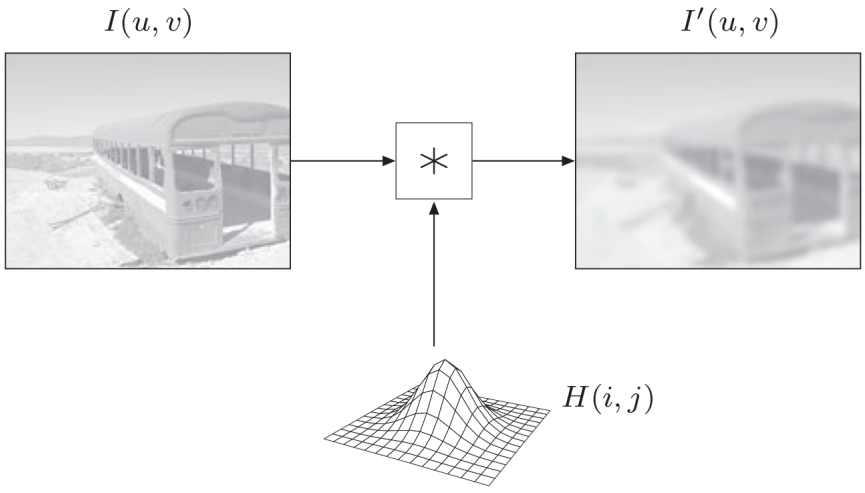

Figure 3.4: Example for convolution on a gray-scale image [BB15]

The pooling layers reduce the amount of information while preserving relevant features. Essential parameters of the pooling layer are again filter and step size. There are three different methods for pooling: Max-, Average- and Sum-Pooling. Max-Pooling describes passing the maximum value in the filtered region through to the subsequent layer. Average-Pooling describes computing the average value of all values in the filtered region and passing it. And lastly Sum-Pooling describes

computing the sum of all values in the filtered region and passing it. An example for the application of Max-pooling is shown in figure 3.5.

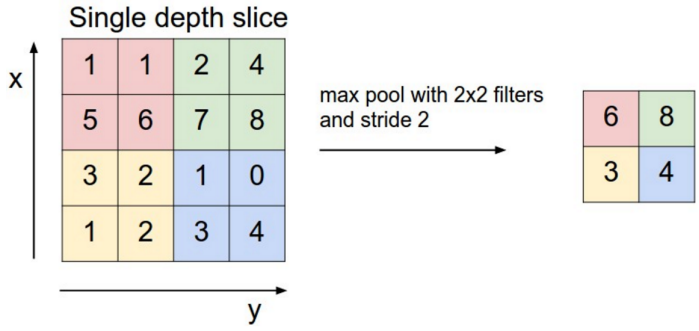

Figure 3.5: Visualization for Max-pooling [Kar18]

The fully-connected layers are simply described as layer, where every neuron is connected to every neuron in the next layer. They should process all inputs from the previously described layers. [Hil18]

CNN are quite capable of classifying objects in images, but object detection is a way more challenging task. Due to its additional localization aspect which is not necessary in a classifier. The solution for achieving this task is to introduce a regional aspect to CNN. This leads to a Region-based Convolutional Neural Network (R-CNN), which is divided into three modules. First, region proposals are generated, which determine Regions of Interest (ROI) for detection. Second, a CNN extracts features from each ROI to generate a feature vector. Finally, for each class a binary SVM, further described in section 3.7, that was previously trained for its class, computes from the vector a class-specific result. [Gir14]

The general approach of training a NN is in case of supervised training to feed its input layer with data which was previously classified or labeled with an expected output. The training data is classified if a classifier should be trained or labeled for a detector. By predicting an output based on its internal weights. The prediction is compared with the expected output and based on the result the weights are adjusted. This is achieved by defined a cost function. Due to the fact that the optimal solution has the smallest possible cost it can be treated as optimization problem.

The successful application of a NN for object detection can be seen in the YOLO architecture described in section 2.3, which claims to outperform the R-CNN approach. Even though, there exist a lot of different approaches for NN. [Red16]

3.5 Histogram of oriented gradients

The histogram of oriented gradients (HOG) introduced by [McC86] is a method for extracting features from images. It is characterized as dense, that means it extracts features from the whole image or the ROI. This contrasts to the SIFT method in section 3.6. At first, the meaning of a feature descriptor is to be described. It is essentially the representation of an image or ROI simplified to characteristic information cleaned from unnecessary information. A HOG feature descriptor uses

the distribution of gradient directions as features. The gradients contain characteristic information because their magnitude is large at edges and corners in the image and these areas are more distinctive than flat ones for the object.

The HOG methods works as follows. First, the image is preprocessed by dividing it into patches with a fixed aspect ratio. According to [DT05] a typical patch size for pedestrian detection is $64 \cdot 128$ pixels, which results in an aspect ratio of $1:2$.

Second, the horizontal g_y and vertical gradients g_x of each patch are computed, which are combined afterwards to belonging magnitude g and direction of the gradient θ by equation 3.1 and 3.2.

$$g = \sqrt{{g_x}^2 + {g_y}^2} \tag{3.1}$$

$$\theta = \arctan \frac{g_y}{g_x} \tag{3.2}$$

The magnitude of gradient highlights rapid changes in intensity in the image. Most of the information in the image are removed, but the characteristic information is still preserved. The magnitude and direction of the gradient for every pixel is now known. For RGB colored images the gradients for every channel are computed and only the maximum magnitude with its corresponding direction considered.

Third, to calculate the HOG the image patch is divided into $8 \cdot 8$ pixel cells. Each cell is considered with a fixed number of gradient orientation bins. Now each pixel votes depending on its gradient direction for a bin in its cell with a weight proportional to its gradient magnitude.

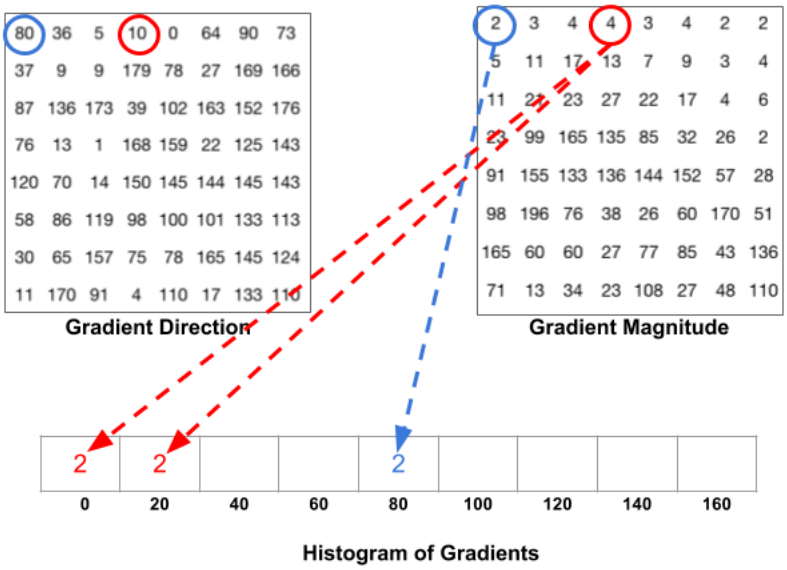

Figure 3.6: HOG voting visualization [Mal18a]

Figure 3.6 shows the voting process. The blue marked pixel has a direction of 80 degree, therefore it adds its magnitude into the corresponding bin. The red marked pixel has a direction of 10 degree and adds therefore half of its magnitude to the neighboring bins 0 and 20 degree. After processing all pixels in the same manner, a corresponding histogram is generated.

Fourth, the histogram is highly depending on the overall lightning of the image. By normalizing the histogram vector a lighting independent descriptor is created. Therefore, blocks of $4 \cdot 4$ cells are created and normalized. Now the region to be normalized is moved by one cell and again normalized. This process is repeated over the whole image patch.

Finally, all normalized vectors are combined to one vector and the HOG descriptor is created. [DT05]

3.6 Scale-invariant feature transform

Scale-invariant feature transform (SIFT) is a method to extract local features from images and was introduced by [Low99] in 1999. It is inspired by Harris' and Stephens' corner detection algorithms [HS88] and tries to implement additionally to the rotation-invariance of its predecessors a scale-invariance. This is required because a corner does not necessarily stay a corner if the image is scaled.

The SIFT algorithm is divided into five steps. First, the ROI has to be adjusted to the scale of the image. This can be done by scale-space filtering the image. Here the Laplacian of Gaussian [MH80] can be used as blob detector resulting blobs in a variety of sizes indicated by different σ-values, which can be seen as a scaling parameter. Or as a more performance efficient approach the Difference of Gaussians (DoG) [DA06] which is an approximation of it. Afterwards local extrema are searched, these are potential keypoints and give information about their scale in which they are found.

Second, the accuracy of potential keypoint locations has to be increased. The Taylor series expansion of scale space is used to calculate a more accurate location. Afterwards the intensity at the extrema is filtered with a threshold. Additionally, low-contrast keypoints are removed.

Third, the keypoint are made rotation-invariant. Depending on the scale the environment of the keypoint is considered and gradient magnitude and direction are computed locally. The different orientations are ordered in a histogram with 36 bins considering 360 degrees. Now the orientation of the keypoint is derived from the highest peak in the histogram and any peaks above 80%.

Forth, the keypoint descriptor is created by considering a $16 \cdot 16$ area around the point. This is divided into 16 blocks with a $4 \cdot 4$ size. For each block orientation histogram with 8 bins is created. These 128 bins for each keypoint are represented as vector to build the keypoint descriptor.

Finally, to match two images the keypoints of both are compared, e.g. using a brute force approach.

3.7 Support vector machines

Support vector machines (SVM) are in the context of machine learning a tool for supervised pattern classification. Additionally, SVM can be used for clustering and regression tasks. They achieve this by generating an optimal hyperplane from labeled training data and categorizing new data with the hyperplane. To visualize following basic example shown in figure 3.7 is given, two classes of points are drawn into a two-dimensional Cartesian coordinate system. These can be easily separated by a straight line, although multiple lines offer a solution. Now the line with highest margin to all data points is chosen. This example describes a linear, binary SVM, the most common type of SVMs, because the points are separated by a linear discriminant function (LDF) and the feature space is separated into two halves which means only binary classification is possible. The LDF can be described by a weight vector β^T and a threshold weight β_0, shown in equation 3.3. LDFs can be used to separate nonlinearly separable data by using a suitable mapping of the data points from the input space to a feature space with higher dimension (kernel trick). [MR16]

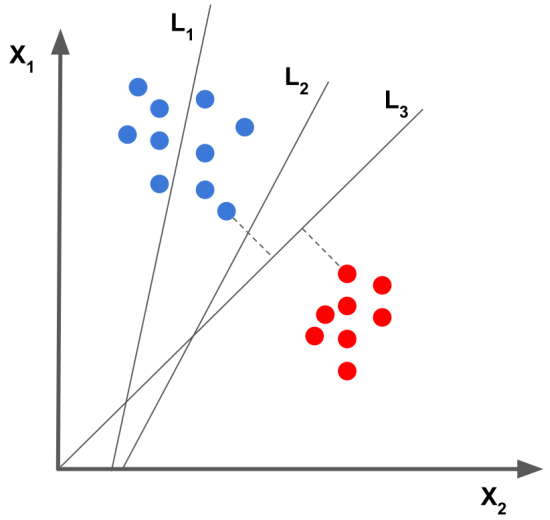

Figure 3.7: Basic example for SVM [Mal18b]

$$x_2(x_1) = \beta_0 + \beta^T x_1 \tag{3.3}$$

Furthermore, multiclass classifiers can be built from combing multiple binary classifiers, shown in figure 3.8. For each combination of classes, $C_{1,2}$, $C_{1,3}$ and $C_{2,3}$, a discriminating function is generated. This results in an ambiguous region, where points are hard to classify. [MR16]

SVMs reformed machine learning and pattern recognition in terms of classification over more than two decades and are now used as state-of-the-art model for classification tasks. They offer two characteristic properties:

1. Training the LDF of a SVM can be seen as a convex optimization problem. This is done by maximizing the between classes. Therefore, it is called *maximum-margin classifier*.

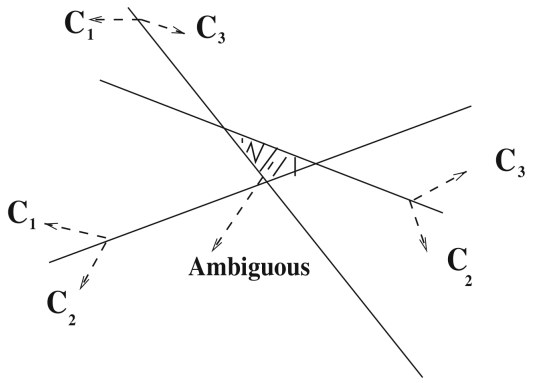

Figure 3.8: Multiclass classification with multiple binary classifiers [MR16]

2. To increase efficiency the kernel trick can be applied which performs computations in the input space instead of the feature space.

Contributing to the popularity of SVMs are widely spread libraries like LIBSVM [CL11], which will be later used in the project. [MR16]

3.8 Bag of Visual Words

Combining the Bag of Words method with visual categorization leads to the Visual Categorization with Bags of Keypoints or Bag of Visual Words approach taken by [Csu04]. This methods tries to solve the problem of identifying objects in natural images while generalizing across variants of the object class. It is based on clustering invariant descriptors of image patches to generate high-dimensional feature vectors and can be divided into the following steps.

First, images patches are detected and described. Due to their robustness to partial visibility and clutter local descriptors are chosen. Using the Harris affine region detector [MS02] leads to an affine region, which is mapped circular. Now SIFT descriptors are computed for this region, described in section 3.6.

Second, the descriptors of the image patches are assigned to a vocabulary, which consists of predetermined clusters. Due to the circumstance that it would take a lot of resources to compare the descriptors of the image to be classified to all descriptors in the training data and comparing only a small number of training descriptors would lead to a low accuracy, an intermediate step of clustering is chosen. To reduce the size k-means clustering [DHS01] is applied multiple times with different k and the cluster with the lowest risk in categorization is taken.

Finally, a multiclass classifier like a SVM is applied to the image using the bag of keypoints as feature vector and selects a class for the image.

4 Methods

In the following section criteria for analyzing a variety of methods for object detection methods on the Turtlebot3 platform will be described. Additionally, a general way of implementing different methods will be considered. Furthermore, four methods for object detection are chosen and their usage is described.

4.1 Criteria analysis

The most important step to evaluate different approaches for detecting objects with a specific system is to set up criteria which can be used to compare these approaches. So according to [WJ06] the following criteria should be considered:

- Performance
- Precision
- Recall
- Intersection over Union (IoU)

Performance should be described with equation 4.1 as the average number of frames from the camera input the system can process and should be measured in FPS.

$$\text{Performance} = \frac{\text{Processed frames}}{\text{System uptime}} \tag{4.1}$$

According to [DG06] the following four cases of detection have to be defined. First, True Positive (TP) means a correct identified object. Secondly, False Positive (FP) means a wrong identified object. Thirdly, True Negative (TN) describes a case where no object is detected and that is true. Finally, False Negative (FN) means there is an object but it is not detected. Precision is defined as the ability of the method to detect only relevant objects and is therefore described with equation 4.2.

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} = \frac{\text{TP}}{\text{all detections}} \tag{4.2}$$

Recall should be described as the ability of the method to detect all relevant cases and is therefore defined by equation 4.3 with the number off positively identified objects among all properly identified cases.

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} = \frac{\text{TP}}{\text{all ground truths}} \tag{4.3}$$

The IoU describes the accuracy of the bounding boxes marking their corresponding object. It is computed by dividing the intersecting area of the estimated and real bounding box by the overlapping area, shown in equation 4.4. [Eve10]

$$\text{IoU} = \frac{\text{Area of intersection}}{\text{Area of union}} \tag{4.4}$$

These criteria should be in reasonable boundaries, suitable for the use-case of the system. Additionally, the implementation of each object detection method should be efficient enough in memory and IO operations to run on the SoC-board used by the Turtlebot3 platform. Furthermore, the process running object detection should leave reasonable system resources available to run other processes that invoke operations on the system. Outsourcing heavy, resource demanding workloads to a cloud service or specialized hardware could be possible. This would require a stable connection to the specific cloud service or in case of specialized hardware the possibility of integrating these components into the Turtlebot3 platform.

The capability of the Joule 570x, used on the Turtlebot3 *Waffle*, is rather limited for resource demanding processes. Due to [Int18a] it is driven by a 1.7 GHz - 2.4 GHz quad-core Intel Atom T5700 processor with 4 GB of RAM and 16 GB of flash memory, more detailed specifications are listed in table A.2. Therefore, the capability RAM-intensive processes is quite limited. Moreover, the integrated GPU is not suitable for heavy computation. In conclusion, all heavy computational tasks have to be performed by its CPU. This means local real-time object detection will be really challenging because it is not efficiently computable on CPUs.

4.2 Implementation

In the following section the implementation of four different methods for object detection or classification shall be described. The approach of implementation is as general as possible solved to enable the usage of other methods.

4.2.1 Export images

Most methods for object detection are expecting single images as input instead of a video stream. This results in the need to access the video stream recorded by the camera of the robot and export single images out of it. For ease of use these images should be exported to a local filesystem and from there further processed. ROS is delivered with a package suited for this task, *image_view*. As shown by in the launch-file in listing 4.1, this package can be easily used to export images from a topic, here */camera/color/image_raw* with a specified framerate in line 4, here set as seconds between two frames to a directory accessible by ROS.

```
<launch>
  <node name="export_images" pkg="image_view" type="extract_images" cwd="ROS_HOME">
    <remap from="image" to="/camera/color/image_raw" />
    <param name="sec_per_frame" value="1.0" />
    <param name="filename_format" value="obj_detection_frame%04d.jpg" />
  </node>
</launch>
```

Listing 4.1: Launch file for exporting images

The recorded resolution can be easily adjusted by changing the default resolution in the launch-file of the camera bringup as shown in listing 4.2 or by setting the arguments while calling the bringup.

```
  <arg name="color_width"          default="1920" />
  <arg name="color_height"         default="1080" />
```

Listing 4.2: Adjusting camera resolution

To prevent bottlenecking due to slow write speeds while writing exported images to the filesystem the ROS_HOME directory is mounted as temporary filesystem using the RAM of the system. This can be archived by adding the entry shown in listing 4.3 to */etc/fstab*. Using a temporary file system instead of the internal flash memory is based on experience multiple times faster in read and write speeds.

```
tmpfs /home/psawaffle1/.ros tmpfs rw,noexec,nodev,nosuid,uid=psawaffle1,gid=psawaffle1
    ,mode=1700,size=500M  0 0
```

Listing 4.3: Temporary filesystem from /etc/fstab

4.2.2 Forwarding exported images to the desired method

To use a variety of methods the in section 4.2.1 exported images have to be forwarded to each framework in the most general way. This should be solved by referencing each image in a single terminal command in context of the method that should be used. This way of implementation leads to a simple, understandable and easily adaptable approach matching PEP20 [Pet04]. The ROS_HOME directory should be now watched for new files in the naming scheme, given by listing 4.1. This can easily achieved by using a simple filesystem watcher, here the implementation in Python is chosen due to its script-based language style and widely spread, shown in listing B.1. After the watcher detects a new file, its path should be added to a predefined command depending on the object detection method that should be used and executed in shell. This design enables multi-threaded computation by decoupling the process of analyzing one frame from its parent process, here a Python instance. Using multi-threaded computation allows the usage of all four cores the Joule is equipped with, here this means four threads are available due to lacking hyper-threading support. The implementation of calling the method can simply be added to the watcher script. Furthermore, the execution time for each call can be measured here. Even though it is not considered in this project, it is quite simple to process the feedback of each object detection method, combine it with a time stamp from the image export node and write the processed information into

a ROS parameter server or make these in any other way available to ROS. This would enable the possibility to map detected objects into the environment of the robot and enable interaction.

4.2.3 Darknet YOLOv2 Tiny

As local approach utilizing a NN the Darknet frame work published in [Red16] is chosen, previously described in section 2.3. Although the YOLO network structure offers great results, it is quite resource-intensive. Therefore, the YOLOv2 Tiny structure is used, shown in table 4.1. It consists of 16 layers with 9 convolutional and 6 max-pooling layers. Using such a thin network offers suitable performance for object detection on the used SoC, the Joule 570x.

Table 4.1: Network sturcture of YOLOv2 Tiny

	layer	filters	size	input	output	
0	conv	16	$3 \times 3/1$	$416 \times 416 \times 3$	$416 \times 416 \times 16$	0.150 BFLOPs
1	max		$2 \times 2/2$	$416 \times 416 \times 16$	$208 \times 208 \times 16$	
2	conv	32	$3 \times 3/1$	$208 \times 208 \times 16$	$208 \times 208 \times 32$	0.399 BFLOPs
3	max		$2 \times 2/2$	$208 \times 208 \times 32$	$104 \times 104 \times 32$	
4	conv	64	$3 \times 3/1$	$104 \times 104 \times 32$	$104 \times 104 \times 64$	0.399 BFLOPs
5	max		$2 \times 2/2$	$104 \times 104 \times 64$	$52 \times 52 \times 64$	
6	conv	128	$3 \times 3/1$	$52 \times 52 \times 64$	$52 \times 52 \times 128$	0.399 BFLOPs
7	max		$2 \times 2/2$	$52 \times 52 \times 128$	$26 \times 26 \times 128$	
8	conv	256	$3 \times 3/1$	$26 \times 26 \times 128$	$26 \times 26 \times 256$	0.399 BFLOPs
9	max		$2 \times 2/2$	$26 \times 26 \times 256$	$13 \times 13 \times 256$	
10	conv	512	$3 \times 3/1$	$13 \times 13 \times 256$	$13 \times 13 \times 512$	0.399 BFLOPs
11	max		$2 \times 2/1$	$13 \times 13 \times 512$	$13 \times 13 \times 512$	
12	conv	1024	$3 \times 3/1$	$13 \times 13 \times 512$	$13 \times 13 \times 1024$	1.595 BFLOPs
13	conv	512	$3 \times 3/1$	$13 \times 13 \times 1024$	$13 \times 13 \times 512$	1.595 BFLOPs
14	conv	130	$1 \times 1/1$	$13 \times 13 \times 512$	$13 \times 13 \times 130$	0.022 BFLOPs
15	detection					

The network should be trained with a minimal but sufficient dataset. This should decrease the training time. Therefore, Hemanth Venkateswara's *Office-Home Dataset* [Ven17] is chosen. It consists of 65 object categories split in four domains: Art, Clipart, Product and Real-World. These domains contain several images in a variety of sizes for each class. Table 4.2 shows the minimum number of images, minimum and maximum size and the accuracy of a linear SVM trained with the data for each domain. To create a data set with a suitable minimum number of images per class the domains Product and Real-World are combined, shown in table 4.3. The domains Art and Clipart are excluded from the considered dataset because they consist of primarily abstract images of the supposed objects. Additionally, the number of classes is reduced from 65 to 21 categories: *Backpack, Batteries, Bottle, Bucket, Candles, Drill, Flipflops, Hammer, Helmet, Keyboard, Knives, Marker, Monitor, Mug, Pan, Scissors, Screwdriver, Sneakers, Toys, TrashCan* and *Webcam*. These classes are chosen due to availability, suitable size to deploy in a testing environment and visual differences.

To prepare the dataset for training labels for each image have to be added, according to the scheme shown in listing 4.4. This scheme is composed as follows: *<object-class>* describes the category

Table 4.2: Data collection of Hemanth Venkateswara's *Office-Home Dataset* [Ven17]

Domain	Min: #	Min: Size	Max: Size	Acc.
Art	15	117×85 pix.	4384×2686 pix.	44.99 ± 1.85
Clipart	39	18×18 pix.	2400×2400 pix.	53.95 ± 1.45
Product	38	75×63 pix.	2560×2560 pix.	66.41 ± 1.18
Real-World	23	88×80 pix.	6500×4900 pix.	59.70 ± 1.04

Table 4.3: Composition of resulting dataset

Class	Number of images
Backpack	302
Batteries	228
Bottle	345
Bucket	253
Candles	336
Drill	189
Flipflops	279
Hammer	255
Helmet	322
Keyboard	294
Knives	258
Marker	197
Monitor	320
Mug	267
Pan	183
Scissors	307
Screwdriver	203
Sneakers	288
Toys	242
TrashCan	259
Webcam	217

of the object as integer starting with 0, *<x_center>* the center in x-direction of the object's bounding box relative to the image's width, *<y_center>* the relative center in y-direction, *<width>* the relative width of the bounding box and *<height>* the relative height. One image can have multiple label entries in its corresponding label file separated by line breaks.

```
<object-class> <x_center> <y_center> <width> <height>
```

Listing 4.4: Labeling scheme for Darknet Object Detection

After preparing the labels, a meta file describing the dataset has to be created. This file is shown in listing 4.5 and consists of the number of classes (line 1), the path to a list of all training images (line 2), the path to a list of images to validate the learning process (line 3), the path of a list of all class names and the relative path, where generated weight should be saved.

```
classes = 21
train = <path>/90.list
valid = <path>/10.list
names = <path>/labels.txt
backup = custom/OfficeHomeDataset_smallv2
```

Listing 4.5: Meta file of the prepared dataset

The training process now outputs information, e.g. iteration number, an averaged error and an averaged recall. Based on this data it can be estimated how long the training process should be continued. Computing the training on a GPU using Compute Unified Device Architecture (CUDA) [SK10] offers extraordinary high performance boosts compared with training on a CPU.

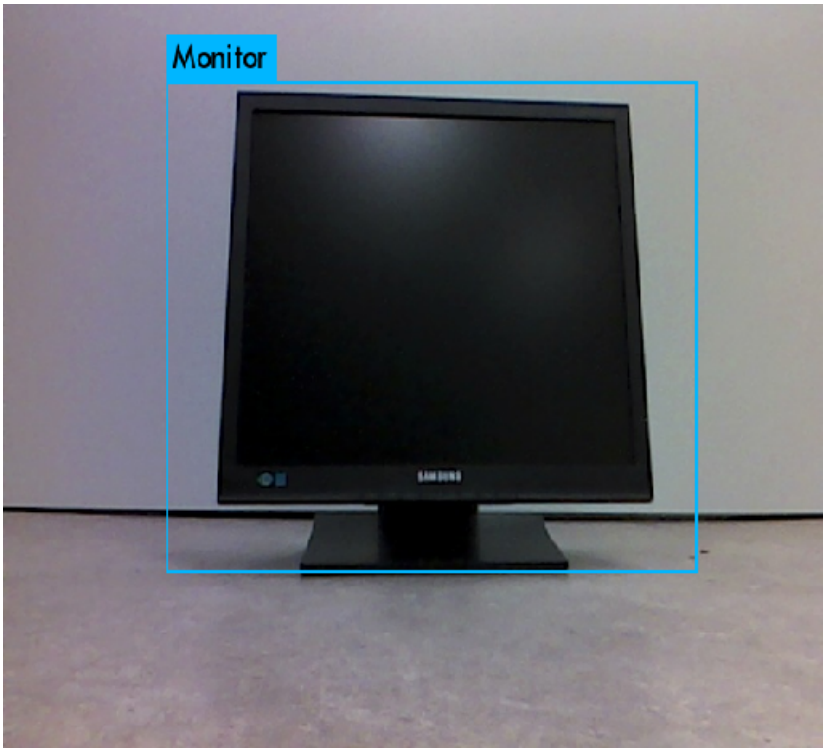

Figure 4.1: Example for a detection using Darknet YOLOv2 Tiny

Using the computed weights after approximately one week of training, the NN can be fed with them and images to be processed. This results in detections, one of these is shown in figure 4.1. Due to the output of Darknet which consists only of an image with the prediction and the object category as text output, it is required to modify Darknet to output the prediction including its bounding box as text. This ensures that the predictions can be easily recorded and evaluated without the need of extracting information about each bounding box from the prediction images causing unnecessary bulk and error.

4.2.4 Intel NCS

As second method using the NCS is chosen, because it can outsource the computation-heavy task of analyzing images in a NN from the Joule to itself. This should lead to a vast performance boost and due to the NCS' low power consumption to an increase in runtime of the system. To use the

previously in section 4.2.3 trained NN with the NCS, mentioned in section 3.3, the network and its weights have to be exported into a Caffe model [Jia14]. This is basically a different way to save the network structure and its weights. The export can easily achieved by using openly available conversion scripts and results in a *prototxt* file describing the structure and a *caffemodel* containing the weights. The model is now compiled to the NCS using preferably more powerful host than the SoC and can be easily accessed with the SoC using the SDK and example code snippets provided by Intel [Int18b].

4.2.5 Google Cloud Vision

Another approach could be to outsource the detection process to a cloud service, if an Internet connection is available. The Google Cloud Vision service offers the possibility to localize objects in images. This feature is at the moment still in beta. Therefore, it is not possible to provide an own dataset the Google NN should be trained with. Google offers a network able to detect a huge number of classes pretrained on Google's own huge dataset. This would enable the Turtlebot3 platform to utilize the computation power of Google's cloud to detect objects while being connected to the Internet. Examples for this use-case are provided by Google and could be easily adjusted, shown in listing B.2. Google offers with its Cloud Vision service an easily configurable approach with high usability and performance which is of course not free. It is billed according to the number of request that should be processed.

4.2.6 Minimal bag of visual words classifier

As last approach another local one should be considered. Completely local and not requiring specialized hardware methods offer a higher level of autonomy. Due to high resource demands a rather simple and light on resources one is chosen. Therefore, using a minimal framework the Bag of Visual Words approach should be implemented. Using the same dataset derived from [Ven17] and used for training the NN in section 4.2.3 the bag of keypoints was generated with SIFT, described in section 3.6, as feature extractor. Afterwards images can be classified using LIBSVM [CL11], described in section 3.7. Unfortunately, this approach is only suitable for classification, but it could be enhanced for object detection with e.g. the sliding-window method [LBH09]. Using such a minimal approach is beneficial for its performance, but nevertheless it can result in lower detection rates.

5 Evaluation

The previously in section 4.2 described methods should be compared based on their performance and detection rate. To compare the different methods, objects from the in section 4.2.3 described 21 categories are placed within a testing area and a Turtlebot3 *Waffle* is driven through it while recording all available topics. ROS delivers a command-line tool specifically for this application. It is called *rosbag* and is able to record and play back messages from ROS topics while being designed to be highly performant [Kou16]. This results in a *rosbag* file with a duration of 366 seconds and amongst various other messages consisting 1049 messages of the type *sensor_msgs/Image*, which hold images of a resolution of 640×480 pixels and will be exported with 1 frame per second (FPS). Resolution and framerate are chosen considering the resource demand of the local methods. Furthermore, the input data have to be identical to compare different methods based on their results. In the following section this data should be used to compare the methods and referred as test dataset. This dataset has to be labeled by hand to create comparable results.

5.1 Performance-based comparison

To evaluate the performance of the various approaches the response time for analyzing frames of the test dataset are measured and recorded in table 5.1. The estimated FPS are derived by equation 5.1. These should give an approximation of the maximum FPS the method can handle.

$$\text{Est. FPS} = (\text{Averaged latency})^{-1} \tag{5.1}$$

Table 5.1: Latencies per image of methods for object detection

Method	Average [ms]	Min [ms]	Max [ms]	Est. FPS
Darknet YOLOv2 Tiny 4.2.3	6049	5239	6860	0.17
Intel NCS 4.2.4	1303	1201	1699	0.77
Google Cloud Vision 4.2.5	1661	1466	2288	0.60
Minimal bag of visual words classifier 4.2.6	2074	660	3569	0.48

The resulting table 5.1 shows that the approach using the Intel NCS delivers the most performance, followed by the Google Cloud Vision method. Closely followed by minimal bag of visual words classifier, which offers the lowest minimal latency and highest range of latencies. The slowest approach is obviously YOLOv2 Tiny running a NN completely local.

To investigate further the averaged system loads of the last minute are recorded while running the different methods separately. Using the Linux terminal tool *uptime* [GJ12] the system CPU load should be interpreted as the system is completely used with value above $1 \cdot$ system cores and below resources are still available. This results in the figures 5.1, 5.2, 5.3 and 5.4. These show that the system is on the lowest load while using the NCS, an averaged load using Cloud Vision and the highest loads using local methods, like the classifier and YOLOv2. Additionally, the system is overloaded while running the classifier and YOLOv2. Therefore, tasks are queued up, which

can the result in higher latencies. Whether these high system loads effect other processes was not further investigated, but it can be assumed due to the fact that the system reacted subjectively slower.

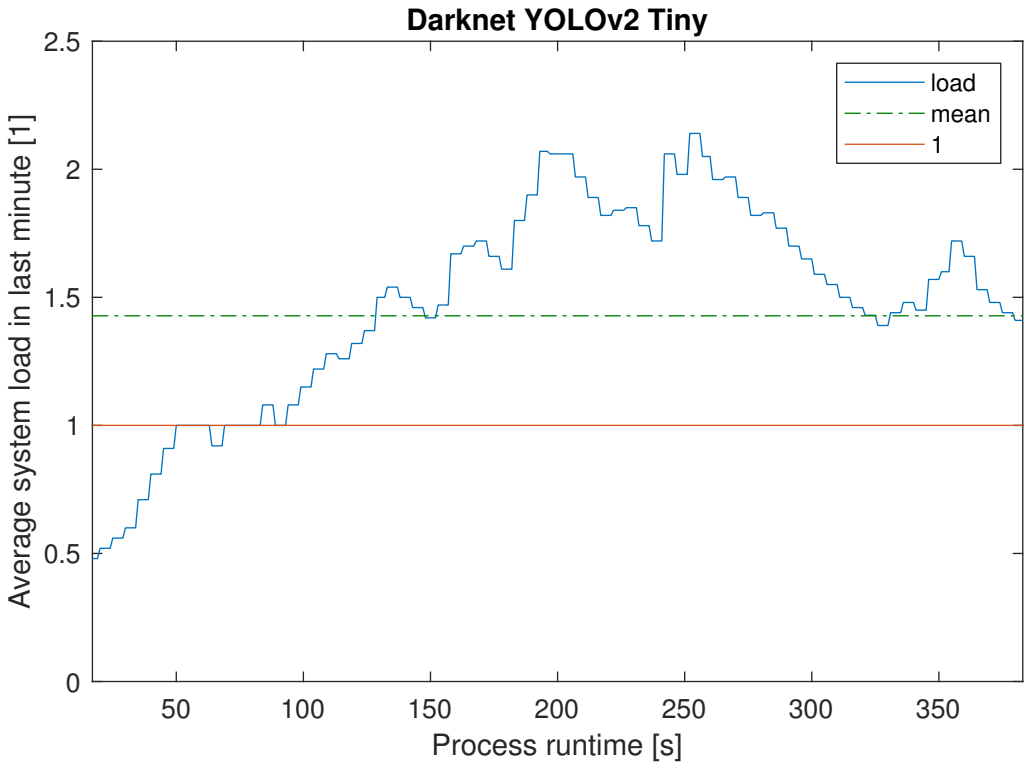

Figure 5.1: Average system load of Darknet YOLOv2 Tiny 4.2.3

Figure 5.1 shows the second highest averaged system load. The rapid changes in load indicate that the detection process uses multiple cores, caused by a longer computation time per frame than time between two frames. This is probably due to too slow boost clock to compute one frame in-time. Furthermore, using multiple cores limits the usage of the maximal boost clock which delays the processing time even more. This could explain the relatively high latency according to table 5.1 for each frame.

Using the NCS for object detection yields the lowest system load, shown in figure 5.2. Indicated by the load always below 1 raises the assumption that none of the available cores is fully loaded and the system is capable of handling the process. This is exactly the aim of this method. Most of the load should be sourced to the NCS. The capability of the system is confirmed by the averaged latency in table 5.1.

The minimal bag of visual words classifier causes the highest averaged system load, shown in figure 5.3. Rapid changes in load can be observed which indicate multi-core usage. Although its lower than the YOLOv2 approach. This assumption is confirmed by the averaged latencies in table 5.1. The CPU is obviously more capable to process images with this method than with YOLOv2. Furthermore, the load is more stable which indicates a more beneficial usage of the system.

Figure 5.4 shows a rather unsteady and unexpected high system load in case of the Google Cloud Vision service. It was assumed that the load is quite low and steady because the system has only to forward images via the network to Google's cloud and to await a response. This is a

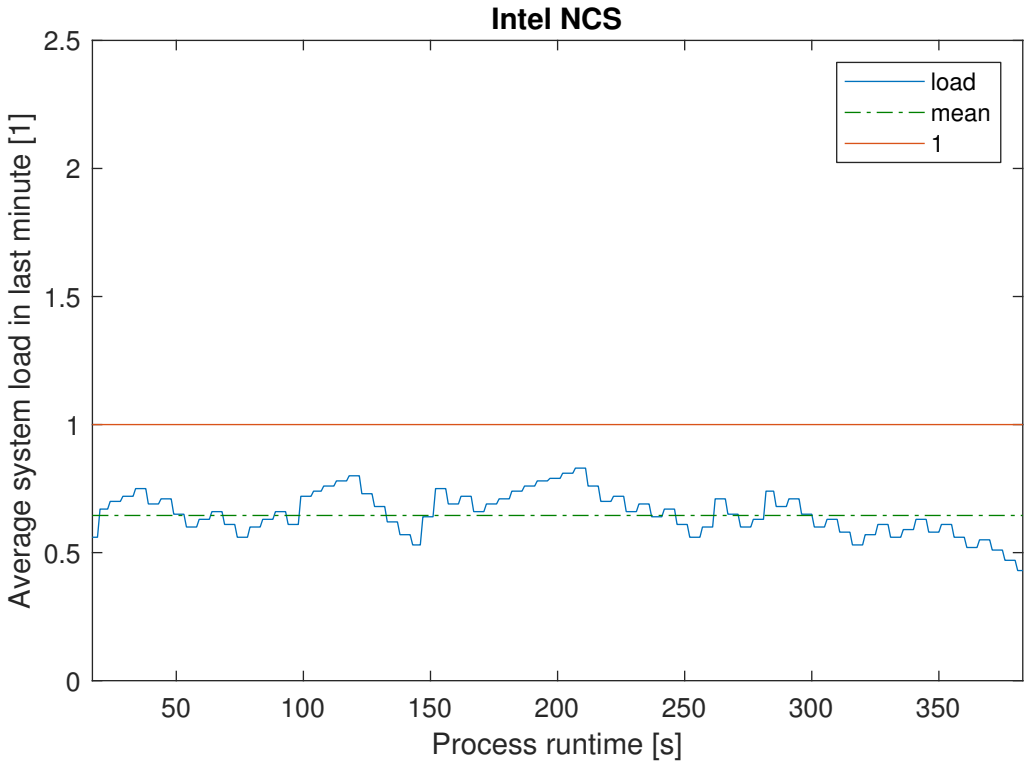

Figure 5.2: Average system load of Intel NCS 4.2.4

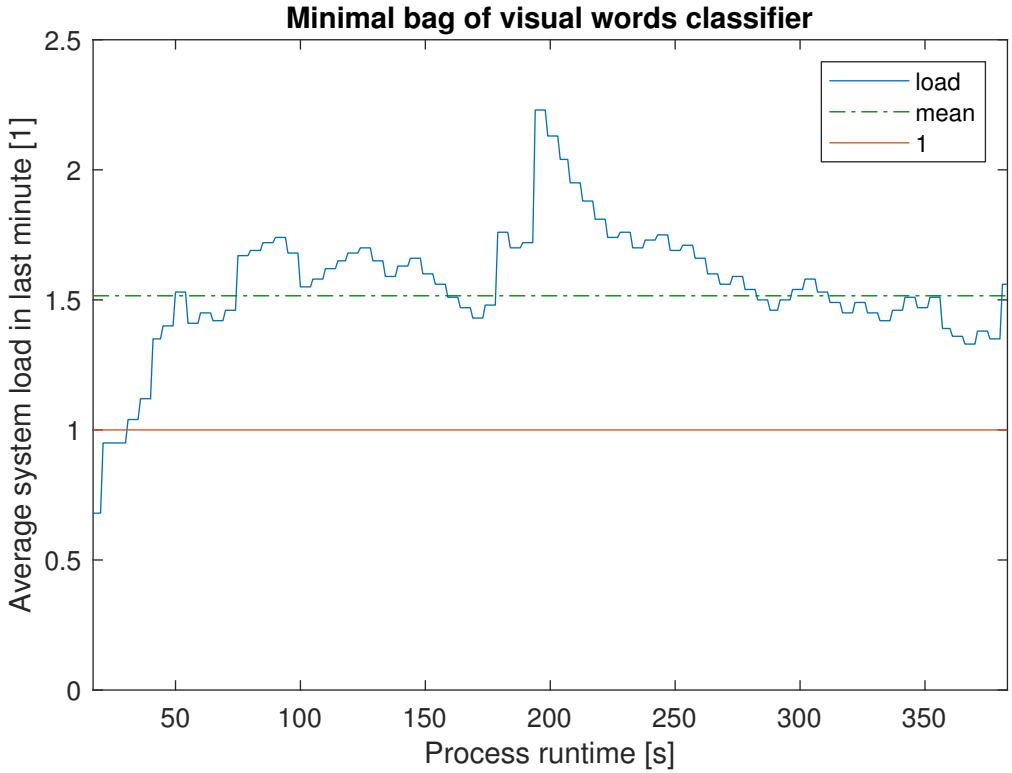

Figure 5.3: Average system load of Minimal bag of visual words classifier 4.2.6

quite low resource demanding task. The high load could be caused by a high number of listeners waiting for a response, but this is unlikely due to the low averaged latency, shown in table 5.1. Or another process running on the system produces such a load which is unlikely too due to

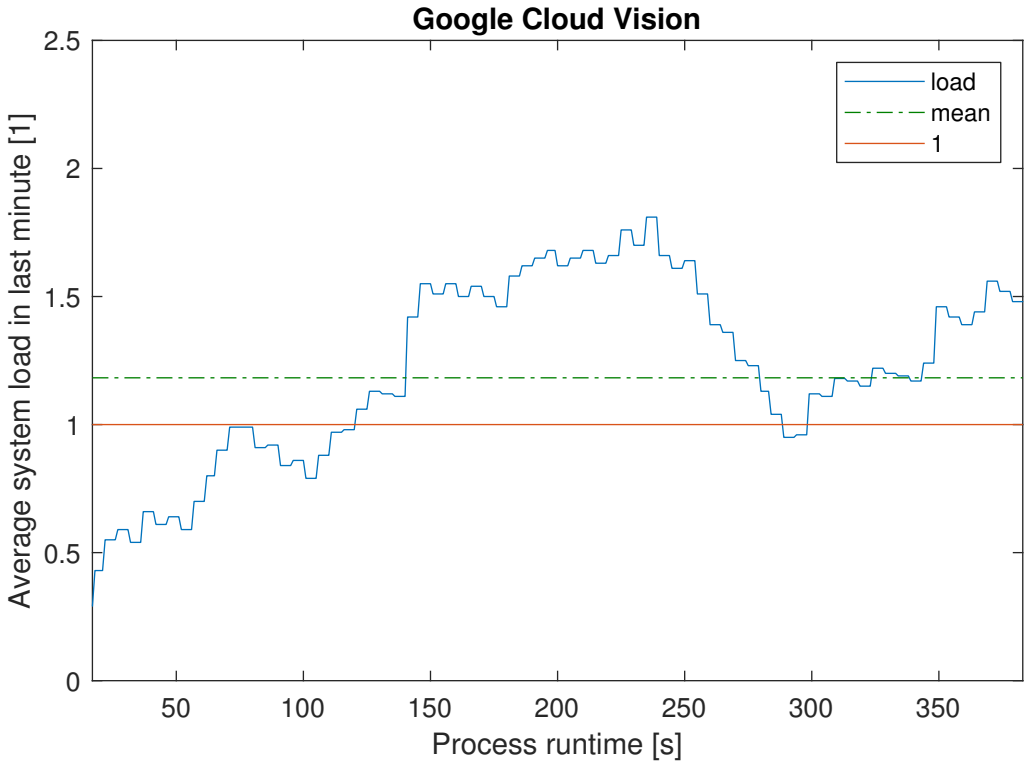

Figure 5.4: Average system load of Google Cloud Vision 4.2.5

the effect's reproducibility. It is more likely that the implementation of the method creates a high overhead. Another cause could be problems, like overheating or overloading, with the on-board network controller. This should be further investigated on other platforms, maybe using a different implementation or even the available C library.

Furthermore, the current draw of the whole system is measured on the battery while running the methods on the test dataset and shown in the figures 5.5, 5.6, 5.7 and 5.8. The current draw should ideally be in some proportional correlation with the system load and should illustrate differences in runtime due to the limited battery capacity. These show high draws for the classifier and YOLO while the current draw of the Cloud Vision approach is extremely high and the NCS low.

Overall it can be stated that the current measurements confirm assumptions made using the system load. The NCS method creates the lowest current draw due to the averaged low system load and the negligible low draw of the NCS. Furthermore, the current draws while using the classifier and the YOLOv2 method are quite high as expected due to their heavy CPU usage. The measurement of the cloud approach results an unexpected high current draw with relatively to the other measurements high fluctuations. This strengthens the assumption that something with the networking hardware is causing the high system load.

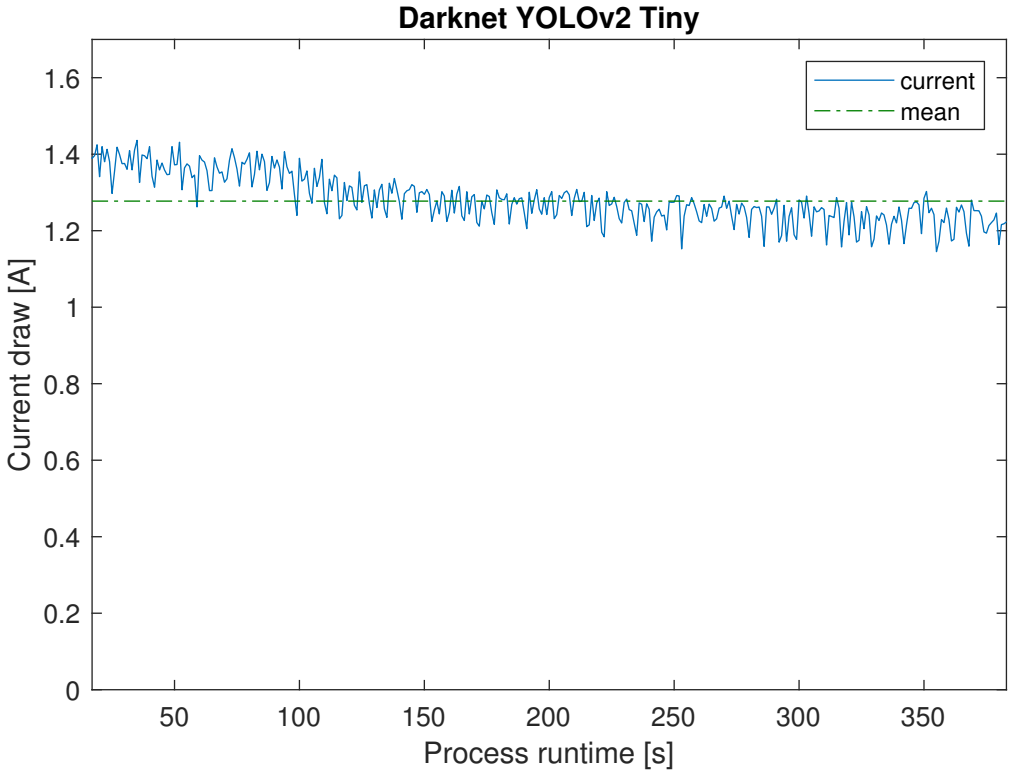

Figure 5.5: Current draw of the system using Darknet YOLOv2 Tiny 4.2.3

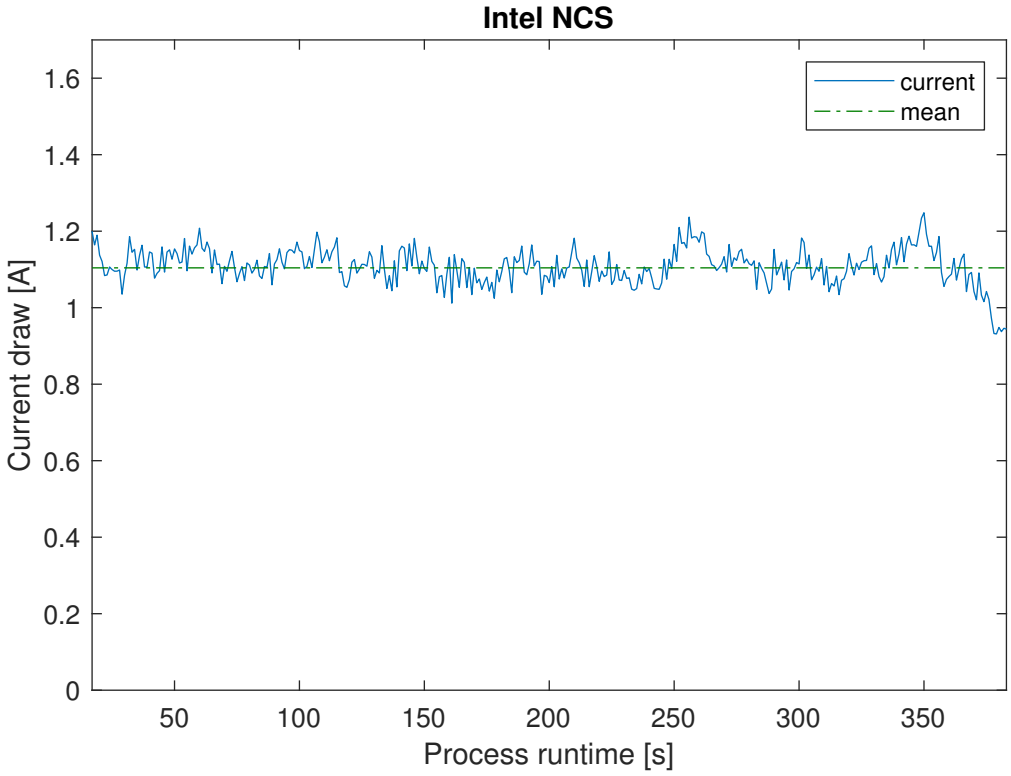

Figure 5.6: Current draw of the system using Intel NCS 4.2.4

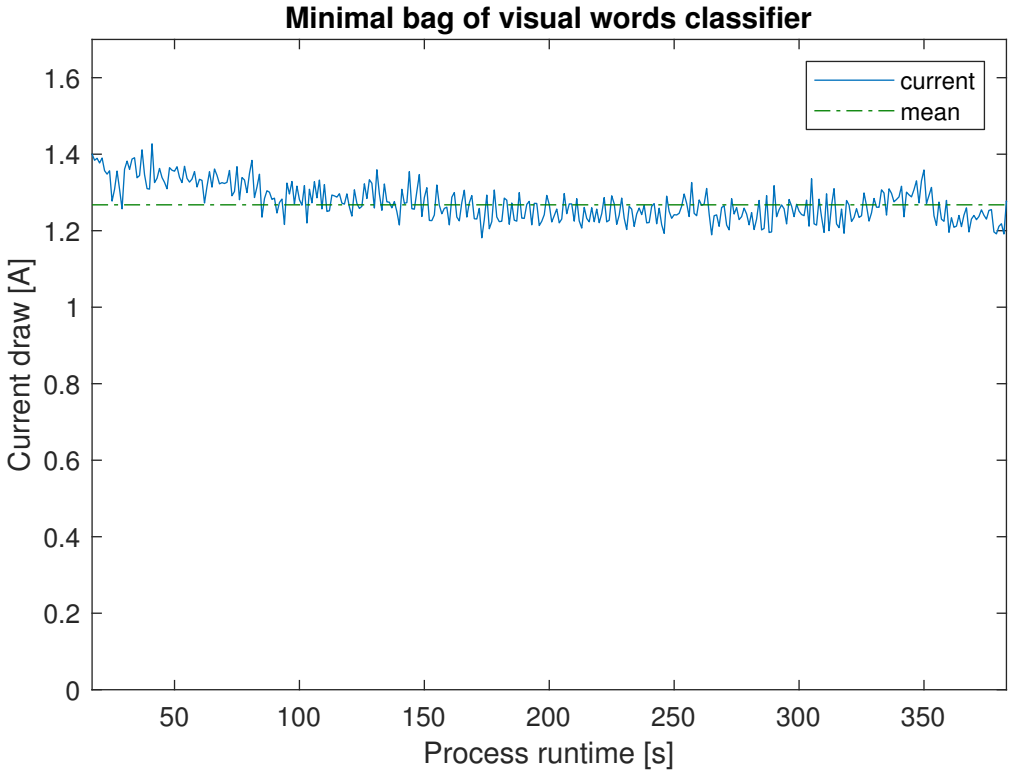

Figure 5.7: Current draw of the system using the minimal bag of visual words classifier 4.2.6

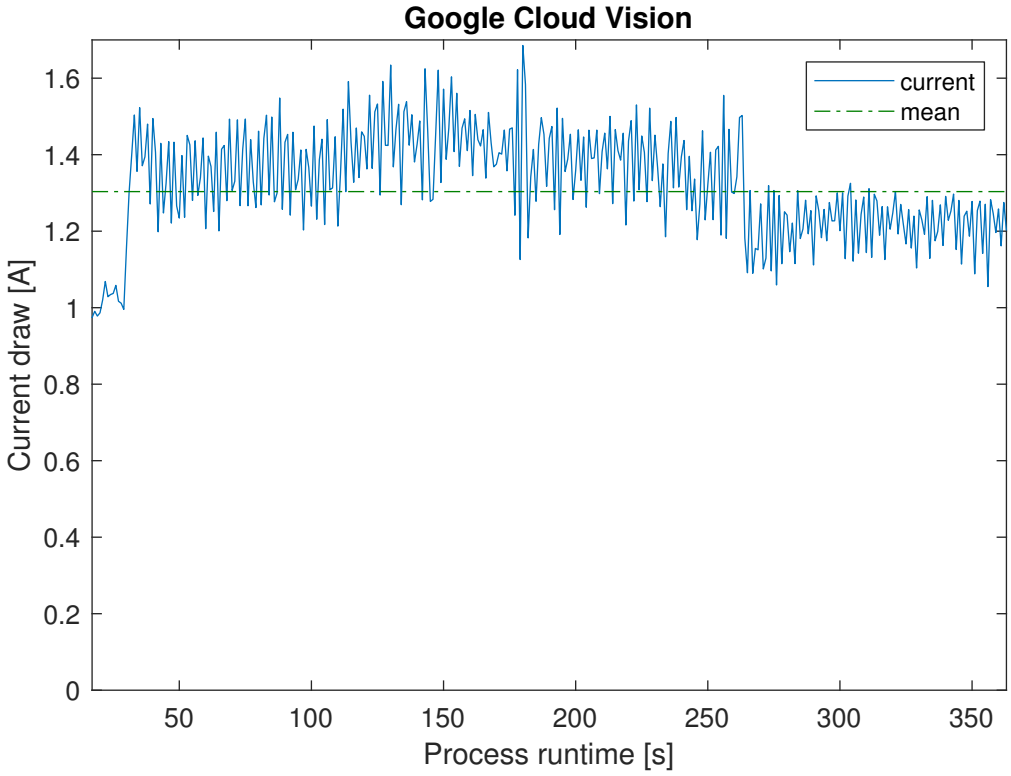

Figure 5.8: Current draw of the system using Google Cloud Vision 4.2.5

5.2 Precision-based comparison

The methods should be compared based on their detection rates. Therefore precision, recall and IoU are computed for each object class and shown in table 5.2. YOLOv2 Tiny and NCS can be combined because they use an identical weighted NN.

Table 5.2: Precision-based comparison of methods for object detection

Category Precision Recall IoU	**YOLOv2 Tiny / Intel NCS**	**Cloud Vision**	**Bag of visual words**
Backpack	0.31 0.29 0.68	0.31 0.83 0.83	0.48 0.35
Batteries	0.00 0.00	0.00 0.00	0.00 0.00
Bottle	0.00 0.00	0.00 0.00	0.00 0.00
Bucket	1.00 0.67 0.82	0.00 0.00	0.00 0.00
Candles	0.33 0.67 0.92	1.00 1.00 0.86	0.00 0.00
Drill	0.00 0.00	0.00 0.00	0.00 0.00
Flipflops	0.00 0.00	0.00 0.00	0.17 0.05
Hammer	0.00 0.00	0.00 0.00	0.14 0.03
Helmet	0.00 0.00	0.39 0.63 0.87	0.00 0.00
Keyboard	0.00 0.00	1.00 0.20 0.90	0.00 0.00
Knives	0.00 0.00	0.00 0.00	0.05 0.04

Marker	0.00	0.00	0.00
	0.00	0.00	0.00
Monitor	0.05	0.58	0.47
	0.08	0.76	0.21
	0.03	0.83	
Mug	0.00	0.00	0.00
	0.00	0.00	0.00
Pan	0.00	0.00	0.00
	0.00	0.00	0.00
Scissors	0.50	1.00	0.32
	0.14	0.57	0.25
	0.78	0.85	
Screwdriver	0.00	0.00	0.00
	0.00	0.00	0.00
Sneakers	0.25	0.5	0.07
	0.01	0.45	0.06
	0.84	0.79	
Toys	0.25	0.43	0.38
	0.02	0.72	0.65
	0.87	0.69	
TrashCan	0.00	0.00	0.08
	0.00	0.00	0.04
Webcam	0.00	0.00	0.00
	0.00	0.00	0.00
Average	0.13	0.25	0.10
	0.09	0.25	0.08
	0.71	0.72	

Comparing the results of the different methods based on their precision, recall and IoU, shown in table 5.2 yields that from all 21 object classes 8 are not detected at all, these are *Batteries*, *Bottle*, *Drill*, *Marker*, *Mug*, *Pan*, *Screwdriver* and *Webcam*. *Batteries* and *Marker* are probably to small considering resolution of the recording and the distance between camera and object. *Bottle*, *Webcam* and *Pan* weren't clearly visible in the exported frame and appear distorted. *Mug* and *Screwdriver* should be big and prominent enough. Due to their distinctive shape and color a

detection should be possible. It has be assumed that the training data was not variant enough to detect those.

Considering only the number of detected object categories the classifier approach recognized the most with 9 classes, followed by the cloud approach with 8 classes and the YOLO / NCS approach with 7. An unexpected result, if it is taken into account that the classifier gives only one prediction per frame and the input images mostly consist of multiple objects. Therefore, this result has to be treated as coincident and it has to be stated that the classifier is not optimal designed for such an use-case.

Considering precision, recall and IoU the cloud gives the best result. This is not too surprising due to the enormous training dataset Google provides for its service. Furthermore, Google offers a greater variety of labels which were for the use in this project condensed to the required 21 object categories.

An interesting result can be observed looking at the IoUs of YOLO / NCS and Cloud Vision. The values are quite similar which shows that the estimated bounding boxes of TP detected objects are identical. Furthermore, rendering a video containing the bounding boxes with their respective object classes shows that most of the bounding boxes match real objects but refer to the wrong class. This indicates that the dataset used for training the network was not sufficient. Although, a larger dataset would have required a significant longer training time.

Additional, it is to argue that the manual test data was rather restrictively labeled. The criteria for labeling an object in one frame was if it is detectable for a human. This causes high TN rates. Furthermore, a video stream can consist motion blur which can effect TN or even FP rates.

In general it can be stated that objects with distinctive contours or colors are better detected than others.

6 Discussion

The aim of this project was to find a suitable method for detecting objects using the Turtlebot3 platform. To achieve this an example scenario was created with the Turtlebot3 *Waffle* and criteria for comparing different methods were specified. After considering existing approaches for object detection and the limitations of the system four different methods were chosen. Two of these approaches should run completely local, one utilizing a NN, Darknet YOLOv2 Tiny, and another as classical machine learning approach, here a minimal bag of visual words design. Using specialized hardware for working with NNs, an additional approach was chosen with the Intel NCS. Last but not least, the performance increase exploiting cloud services should be taken into account using the Google Cloud Vision service.

Ideally a method for detecting objects with the desired platform should be light in its resource demands and simple to use. It should be able to process input data with real-time performance without overloading the system while having a nearly perfect detection rate. It's quite obviously that such a method won't exists, but some method may have the potential to approach the ideal method.

As local approach Darknet YOLOv2 Tiny offers an autonomous design. The utilized NN can be, specifically for its use-case, trained and its structure easily adjusted. Furthermore, with GPU accelerated hardware performance available it delivers real-time performance while providing excellent detection rates. Nevertheless, such resources are not available on the considered platform. Therefore, Darknet can deliver acceptable results, but its performance is way too slow for a moving system. Although, Darknet YOLOv2 could be potentially used for object detection on the Turtlebot3 platform, the system has be enhanced to deliver a higher performance. This could be achieved by simplifying the network structure or improving the computation capabilities of the platform.

Implementing the usage of specialized hardware like the Intel NCS offers the same detection rates like Darknet while lacking some if its usability due to the need of converting models and dealing with an additional API. Although, it offers the ability to process TensorFlow and Caffe models which enables the user to use sophisticated and established models. Furthermore, Intel's documentation is quite application-oriented. Its main point is the specialization for NN computation resulting in a nearly five times faster performance than Darknet. Additionally, the NCS can be used in a clustered setup. Even though, just one NCS delivers an acceptable performance on the Turtlebot3 platform.

The recent development of network transfer speeds enable outsourcing heavy-computational loads to the cloud. Using the Google Cloud Vision service is a good example for this. It offers an enormous dataset while being quite cost-efficient. Furthermore, its performance was acceptable for a slow moving system. The main strength of such a service is its detection rate while maintaining a high usability.

As classical machine learning approach the minimal classifier based on bag of visual words is quite resource demanding, but is able to deliver acceptable results. Due to its nature as pure classifier its

use-case is rather limited. An imaginary use-case could be to pre-analyze images before forwarding them to a detection method.

Concluding the most promising methods for object detection on the desired platform are using the Intel NCS, here clustering multiple sticks should deliver the desired result, or using the Google Cloud Vision service. Using the service should be way more beneficial considering the introduction of 5G. Nevertheless, security concerns during the transfer of the requested images, their corresponding responses and of course the processing through an external company with financial interests have to be made. Therefore, hosting a similar service like a TensorFlow serving instance should be considered. Furthermore, processing the feedback of object detection methods should be implemented as a ROS node to use these information efficiently.

A Data sheets

Table A.1: Turtlebot3 Hardware Specifications [Rob18]

Items	**Burger**	**Waffle (Discontinued)**	**Waffle Pi**
Maximum translational velocity	0.22 m/s	0.26 m/s	0.26 m/s
Maximum rotational velocity	2.84 rad/s (162.72 deg/s)	1.82 rad/s (104.27 deg/s)	1.82 rad/s (104.27 deg/s)
Maximum payload	15kg	30kg	30kg
Size (L x W x H)	138mm x 178mm x 192mm	281mm x 306mm x 141mm	281mm x 306mm x 141mm
Weight (+ SBC + Battery + Sensors)	1kg	1.8kg	1.8kg
Threshold of climbing	10 mm or lower	10 mm or lower	10 mm or lower
Expected operating time	2h 30m	2h	2h
Expected charging time	2h 30m	2h 30m	2h 30m
SBC (Single Board Computers)	Raspberry Pi 3 Model B	Intel® Joule™570x	Raspberry Pi 3 Model B
MCU	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)	32-bit ARM Cortex®-M7 with FPU (216 MHz, 462 DMIPS)
Remote Controller	-	-	RC-100B + BT-410 Set (Bluetooth 4, BLE)
Actuator	Dynamixel XL430-W250	Dynamixel XM430-W210	Dynamixel XM430-W210
LDS(Laser Distance Sensor)	360 Laser Distance Sensor LDS-01	360 Laser Distance Sensor LDS-01	360 Laser Distance Sensor LDS-01
Camera	-	Intel® Realsense™R200	Raspberry Pi Camera Module v2.1
IMU	Gyroscope 3 Axis Accelerometer 3 Axis Magnetometer 3 Axis	Gyroscope 3 Axis Accelerometer 3 Axis Magnetometer 3 Axis	Gyroscope 3 Axis Accelerometer 3 Axis Magnetometer 3 Axis
Power connectors	3.3V / 800mA 5V / 4A 12V / 1A	3.3V / 800mA 5V / 4A 12V / 1A	3.3V / 800mA 5V / 4A 12V / 1A
Expansion pins	GPIO 18 pins Arduino 32 pin	GPIO 18 pins Arduino 32 pin	GPIO 18 pins Arduino 32 pin
Peripheral	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4	UART x3, CAN x1, SPI x1, I2C x1, ADC x5, 5pin OLLO x4
Dynamixel ports	RS485 x 3, TTL x 3	RS485 x 3, TTL x 3	RS485 x 3, TTL x 3
Audio	Several programmable beep sequences	Several programmable beep sequences	Several programmable beep sequences
Programmable LEDs	User LED x 4	User LED x 4	User LED x 4
Status LEDs	Board status LED x 1 Arduino LED x 1 Power LED x 1	Board status LED x 1 Arduino LED x 1 Power LED x 1	Board status LED x 1 Arduino LED x 1 Power LED x 1
Buttons and Switches	Push buttons x 2 Reset button x 1 Dip switch x 2	Push buttons x 2 Reset button x 1 Dip switch x 2	Push buttons x 2 Reset button x 1 Dip switch x 2
Battery	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C	Lithium polymer 11.1V 1800mAh / 19.98Wh 5C	Lithium polymer 1800mAh / 19.98Wh 5C
PC connection	USB	USB	USB
Firmware upgrade	via USB / via JTAG	via USB / via JTAG	via USB / via JTAG
Power adapter (SMPS)	Input : 100-240V, AC 50/60Hz, 1.5A @max Output : 12V DC, 5A	Input : 100-240V, AC 50/60Hz, 1.5A @max Output : 12V DC, 5A	Input : 100-240V, AC 50/60Hz, 1.5A @max Output : 12V DC, 5A

Table A.2: Intel® Joule™570x Developer Kit Specifications [Int18a]

Essentials	
Product Collection	Intel® Joule™Kits
Status	Discontinued
Launch Date	Q3'16
Expected Discontinuance	06/16/2017
Board Form Factor	24mm x 48mm
Socket	2 - 100 pin Hirose
Embedded Storage	16 GB
Lithography	14 nm
DC Input Voltage Supported	3.6-5.25V
Recommended Customer Price	N/A
Pre-Installed Operating System	Linux
Memory & Storage	
Included Memory	Yes

Max Memory Size (dependent on memory type)	4 GB
Memory Types	LPDDR4
Max # of Memory Channels	4
Max Memory Bandwidth	25.6 GB/s
ECC Memory Supported	No
Processor Graphics	
Integrated Graphics	Yes
Graphics Output	Intel® HD Graphics: HDMI 1.4B
Intel® Clear Video Technology	No
# of Displays Supported	2
Expansion Options	
PCI Support	Yes
PCI Express Revision	PCIe Gen2.0
PCI Express Configurations	PCIe - 1 x1 mux'ed with USB3
Max # of PCI Express Lanes	1
PCIe x4 Gen 3	0
PCIe x8 Gen 3	0
PCIe x16 Gen 3	0
PCIe x1 Gen 2.x	1
PCIe x4 Gen 2.x	0
PCIe x8 Gen 2.x	0
PCIe x16 Gen 2.x	0
PCIe x1 Gen 1.x	0
PCIe x4 Gen 1.x	0
PCIe x8 Gen 1.x	0
PCIe x16 Gen 1.x	0
Removable Memory Card Slot	n/a
I/O Specifications	
# of USB	2
USB 2.0 Configuration (External + Internal)	2
USB 3.0 Configuration (External + Internal)	1 + 1
Total # of SATA Ports	0
Max # of SATA 6.0 Gb/s Ports	0
# of eSATA Ports	0
RAID Configuration	0
# of PATA Ports	0
# of Parallel Ports	0
Integrated LAN	No
Integrated Wireless	802.11ac 2x MIMO
Integrated Bluetooth	Yes

B Code samples

```
import sys
import time
import subprocess
import timeit
import getpass
from watchdog.observers import Observer
from watchdog.events import FileSystemEventHandler

username = getpass.getuser()


class Watcher:
    DIRECTORY_TO_WATCH = "/home/" + username + "/.ros"

    def __init__(self):
        self.observer = Observer()

    def run(self):
        event_handler = Handler()
        self.observer.schedule(
            event_handler, self.DIRECTORY_TO_WATCH, recursive=False)
        self.observer.start()
        try:
            while True:
                time.sleep(5)
        except:
            self.observer.stop()
            print "Error"

        self.observer.join()


class Handler(FileSystemEventHandler):

    @staticmethod
    def on_any_event(event):
        if event.is_directory:
            return None

        elif event.event_type == "modified" and event.src_path.find("
    obj_detection_frame") > -1:
            dir = "/home/" + username + "/catkin_ws/src/obj_detection"
            cmd = "md5sum " + event.src_path
            cmd_darknet = "cd " + dir + \
                "/darknet && ./darknet detect custom/yolov2-tiny-deploy.cfg custom/
    yolov2.weights " + event.src_path
            cmd_google = "python cloud_vision.py " + event.src_path
            cmd_bagofwords = "cd " + dir + \
                "/Minimal-Bag-of-Visual-Words-Image-Classifier && python classify.py -
    c custom_obj_codebook.file -m custom_obj_trainingdata.svm.model " + event.src_path
            cmd_ncs = "cd " + "/home/" + username + \
```

```
                "/ba_schraven/YoloV2NCS && python3 ./detectionExample/Main.py --image
    " + event.src_path

            if(len(sys.argv) > 1):
                if sys.argv[1] == "darknet":
                    cmd = cmd_darknet
                elif sys.argv[1] == "google":
                    cmd = cmd_google
                elif sys.argv[1] == "bagofwords":
                    cmd = cmd_bagofwords
                elif sys.argv[1] == "ncs":
                    cmd = cmd_ncs

            start = timeit.default_timer()
            subprocess.call(cmd, shell=True)
            stop = timeit.default_timer()
            print(cmd + ": " + str((stop - start) * 1000) + " ms")


if __name__ == "__main__":
    w = Watcher()
    w.run()
```

Listing B.1: Simple implementation of the watcher in Python

```
import io
import os
import sys

# Imports the Google Cloud client library
from google.cloud import vision
from google.cloud.vision import types

# Instantiates a client
client = vision.ImageAnnotatorClient()

# The name of the image file to annotate
file_name = os.path.join(
    os.path.dirname(__file__),
    sys.argv[1])

# Loads the image into memory
with io.open(file_name, 'rb') as image_file:
    content = image_file.read()

image = types.Image(content=content)

# Performs label detection on the image file
response = client.object_localization(image=image)
objects = response.localized_object_annotations

output = file_name
for obj in objects:
    if(obj.score >= 0.5):
```

```
30          output += "," + obj.name + "," + \
31              str(round(obj.score, 2))
32
33          for vertex in obj.bounding_poly.normalized_vertices:
34              output += (',{},{}'.format(vertex.x, vertex.y))
35
36  print output
```

Listing B.2: Implementation of Google's Object Localizer service

References

[And12] Michael Riis Andersen, Thomas Jensen, Pavel Lisouski, Anders Krogh Mortensen, Mikkel Kragh Hansen, Torben Gregersen, and Peter Ahrendt. “Kinect depth sensor evaluation for computer vision applications.” *Technical Report Electronics and Computer Engineering*, vol. 1(6) (2012).

[BB15] Wilhelm Burger and Mark J. Burge. *Digitale Bildverarbeitung: eine algorithmische Einführung mit Java*. German. 3., vollst. überarb. und erw. Aufl. Berlin [u.a.]: Springer Vieweg, 2015.

[Bra18] Howard Brand, Max Diekel, Yuhao Liu, Mayukh Sattiraju, and Ramraj Segur-Mahadevaraja. “Autonomy Science and Systems Project Report.” Vol. (2018).

[Can86] John Canny. “A computational approach to edge detection.” *IEEE Transactions on pattern analysis and machine intelligence*, vol. (6) (1986), pp. 679–698.

[CB99] Jose C. Brustoloni. “Autonomous Agents: Characterization and Requirements.” Vol. (Oct. 1999).

[CL11] Chih-Chung Chang and Chih-Jen Lin. “LIBSVM: a library for support vector machines.” *ACM transactions on intelligent systems and technology (TIST)*, vol. 2(3) (2011), p. 27.

[CM08] Ondřej Chum and Jiří Matas. “Optimal randomized RANSAC.” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 30(8) (2008), pp. 1472–1482.

[Csu04] Gabriella Csurka, Christopher Dance, Lixin Fan, Jutta Willamowski, and Cédric Bray. “Visual categorization with bags of keypoints.” *Workshop on statistical learning in computer vision, ECCV*. Vol. 1. 1-22. Prague. 2004, pp. 1–2.

[DA06] W Davidson and Mortimer Abramowitz. “Molecular expressions microscopy primer: Digital image processing-difference of gaussians edge enhancement algorithm.” *Olympus America Inc., and Florida State University*, vol. (2006).

[DG06] Jesse Davis and Mark Goadrich. “The relationship between Precision-Recall and ROC curves.” *Proceedings of the 23rd international conference on Machine learning*. ACM. 2006, pp. 233–240.

[DHS01] Richard O Duda, Peter E Hart, David G Stork, et al. “Pattern classification.” *International Journal of Computational Intelligence and Applications*, vol. 1 (2001), pp. 335–339.

[DT05] Navneet Dalal and Bill Triggs. “Histograms of oriented gradients for human detection.” *Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on*. Vol. 1. IEEE. 2005, pp. 886–893.

[Eve10] Mark Everingham, Luc Van Gool, Christopher KI Williams, John Winn, and Andrew Zisserman. “The pascal visual object classes (voc) challenge.” *International journal of computer vision*, vol. 88(2) (2010), pp. 303–338.

[Gir14] Ross Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. “Rich feature hierarchies for accurate object detection and semantic segmentation.” *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587.

[GJ12] Larry Greenfield and Michael K. Johnson. *uptime(1) - Linux man page*. 2012.

[Goo16] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*. Vol. 1. MIT press Cambridge, 2016.

[Gru11] Northrop Grumman. *Bat Unmanned Aircraft System (UAS)*. 2011.

[Hil18] Tobias Hilbert. "Einsatz von Convolutional Neural Networks in der Bildverarbeitung zur Klassifikation von Leiterplatten." Vol. (2018), p. 98.

[Hof17] Janosch Hoffmann. "ROS-Based Object Recognition Framework." Gelsenkirchen, Germany: Westphalian University of Applied Sciences, 2017.

[HS88] Chris Harris and Mike Stephens. "A combined corner and edge detector." *Alvey vision conference*. Vol. 15. 50. Citeseer. 1988, pp. 10–5244.

[Int18a] Intel Corporation. *Intel® Joule™570x Developer Kit*. Specifications. 2018. URL: https://ark.intel.com/products/96414/Intel-Joule-570x-Developer-Kit (visited on 12/24/2018).

[Int18b] Intel Corporation. *Intel® Movidius™Neural Compute Stick*. Intel® Software. 2018. URL: https://software.intel.com/en-us/movidius-ncs (visited on 12/23/2018).

[Jia14] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. "Caffe: Convolutional architecture for fast feature embedding." *Proceedings of the 22nd ACM international conference on Multimedia*. ACM. 2014, pp. 675–678.

[Kar18] Andrej Karpathy. *CS231n Convolutional Neural Networks for Visual Recognition*. 2018. (Visited on 03/01/2019).

[Keh13] Ben Kehoe, Akihiro Matsukawa, Sal Candido, James Kuffner, and Ken Goldberg. "Cloud-based robot grasping with the google object recognition engine." *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE. 2013, pp. 4263–4270.

[Kou16] Anis Koubaa, ed. *Robot Operating System (ROS): the complete reference. Volume 1.* Studies in computational intelligence 625. Cham Heidelberg New York: Springer, 2016. 728 pp.

[LBH09] Christoph H Lampert, Matthew B Blaschko, and Thomas Hofmann. "Efficient subwindow search: A branch and bound framework for object localization." *IEEE transactions on pattern analysis and machine intelligence*, vol. 31(12) (2009), p. 2129.

[Low99] David G Lowe. "Object recognition from local scale-invariant features." *Computer vision, 1999. The proceedings of the seventh IEEE international conference on*. Vol. 2. Ieee. 1999, pp. 1150–1157.

[Mal18a] Satya Mallick. *Histogram of Oriented Gradients*. Learn OpenCV. 2018. URL: https://www.learnopencv.com/histogram-of-oriented-gradients/ (visited on 12/25/2018).

[Mal18b] Satya Mallick. *Support Vector Machines (SVM)*. Learn OpenCV. 2018. URL: https://www.learnopencv.com/support-vector-machines-svm/ (visited on 01/03/2019).

[McC86] Robert K McConnell. *Method of and apparatus for pattern recognition*. US Patent 4,567,610. 1986.

[MH80] David Marr and Ellen Hildreth. "Theory of edge detection." *Proc. R. Soc. Lond. B*, vol. 207(1167) (1980), pp. 187–217.

[MR16] Maddipati Narasimha Murty and Rashmi Raghava. *Support Vector Machines and Perceptrons: Learning, Optimization, Classification, and Application to Social Networks*. Springer, 2016.

[MS02] Krystian Mikolajczyk and Cordelia Schmid. “An affine invariant interest point detector.” *European conference on computer vision*. Springer. 2002, pp. 128–142.

[Pea83] Roy D Pea. “Logo Programming and Problem Solving.[Technical Report No. 12.].” Vol. (1983).

[Pet04] Tim Peters. *PEP 20*. Python Developer’s Guide. Aug. 19, 2004. URL: https://www.python.org/dev/peps/pep-0020/ (visited on 05/01/2019).

[Qui09] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y Ng. “ROS: an open-source Robot Operating System.” *ICRA workshop on open source software*. Vol. 3. 3.2. Kobe, Japan. 2009, p. 5.

[Red16] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. “You only look once: Unified, real-time object detection.” *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[Rob18] Robotis. *Turtebot 3 e-Manual*. 2018. URL: http://emanual.robotis.com/docs/en/platform/turtlebot3 (visited on 06/11/2018).

[Ros58] Frank Rosenblatt. “The perceptron: a probabilistic model for information storage and organization in the brain.” *Psychological review*, vol. 65(6) (1958), p. 386.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[Sto01] Roger Stokey, Tom Austin, Ben Allen, Ned Forrester, Eric Gifford, Rob Goldsborough, Greg Packard, Mike Purcell, and Chris von Alt. “Very shallow water mine countermeasures using the REMUS AUV: a practical approach yielding accurate results.” *OCEANS, 2001. MTS/IEEE Conference and Exhibition*. Vol. 1. IEEE. 2001, pp. 149–156.

[Sze15] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. “Going deeper with convolutions.” *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2015, pp. 1–9.

[Ven17] Hemanth Venkateswara, Jose Eusebio, Shayok Chakraborty, and Sethuraman Panchanathan. “Deep hashing network for unsupervised domain adaptation.” *Proc. CVPR*. 2017, pp. 5018–5027.

[VI17] Jittima Varagul and Toshio Ito. “General object detection method by on-board computer vision with Artificial Neural Networks.” *International Journal of Automotive Engineering*, vol. 8(4) (2017), pp. 149–156.

[WJ06] Christian Wolf and Jean-Michel Jolion. “Object count/area graphs for the evaluation of object detection and segmentation algorithms.” *International Journal of Document Analysis and Recognition (IJDAR)*, vol. 8(4) (2006), pp. 280–296.