Dominik Wieser, BSc

Tests and Evaluations of Neural Networks on Embedded Hardware

Master's Thesis

to achieve the university degree of

Master of Science

Master's degree programme: Information and Computer Engineering

submitted to

Graz University of Technology

Supervisor

Ass.Prof. Dipl.-Ing. Dr.techn. Friedrich Fraundorfer

Institute of Computer Graphics and Vision
Head: Univ.-Prof. Dipl-Ing. Dr.techn. Horst Bischof

Graz, May 2019

This document is set in Palatino, compiled with pdfLaTeX2e and `Biber`.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Affidavit

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly indicated all material which has been quoted either literally or by content from the sources used. The text document uploaded to TUGRAZonline is identical to the present master's thesis.

Date	Signature

Abstract

Object detection and image classification based on Convolutional Neural Networks are state of the art and deliver very good results. However these networks need a lot of computing power for inference. It is still a challenge to run these networks in real-time on small embedded devices. In this thesis a Vision Processing Unit (VPU) is used to run a SSD MobileNet object detection network and a MobileNet classification network on a Raspberry PI. Concretely the Intel Movidius Neural Compute Stick (NCS) is used, which achieves an inference time of 216 ms on the used object detection network. A prototype of a small portable device is build which gives image descriptions of objects via text-to-speech output and vibrations on a smartphone. The surroundings are captured by a camera connected to the Raspberry PI. The device can be controlled over a web application, which also shows the detection results in real-time. Tests with the device showed, that the system works well and that the trained objects are detected reliable. The thesis shows that computer vision can be used on embedded devices for robotics or mobile assistance systems, generating real-time image descriptions. Further improvements of the work could be a object detection in 3D, which also includes the distance of the objects and based on that a better filtering of the spoken image descriptions.

Contents

Abstract iv

1 Introduction 1

2 Models 3
- 2.1 MobileNet . . . 3
  - 2.1.1 Depthwise Separable Convolutions . . . 4
- 2.2 Object Detection . . . 4
- 2.3 Classification . . . 6

3 Intel Movidius Neural Compute Stick 7
- 3.1 NCSDK . . . 7
- 3.2 OpenVINO . . . 9
- 3.3 Comparison of the NCS vs. NCS 2 . . . 11
- 3.4 Different form factors of Myriad VPUs . . . 11

4 Implementation 13
- 4.1 Hardware Setup . . . 13
- 4.2 Converting the models . . . 14
  - 4.2.1 Classification . . . 14
  - 4.2.2 Object Detection . . . 16
- 4.3 Application . . . 17
  - 4.3.1 Camera Publisher . . . 20
  - 4.3.2 Inference Publisher . . . 22
  - 4.3.3 Web Server . . . 26
  - 4.3.4 Web Application . . . 29
  - 4.3.5 Autostart . . . 37

5 Evaluation **39**
5.1 Object Detection Model 39
5.1.1 Time for Inference on VPU vs CPU 40
5.1.2 Custom Trained Assistant System Network 43
5.1.3 Evaluation of the Perspective of the Camera 46
5.2 Classification Model 50
5.2.1 Time for Inference on VPU vs CPU 50
5.2.2 Evaluation of Detection Results of the Classification Network 50

6 Conclusion and Future work **53**

Bibliography **56**

List of Figures

2.1 Regular Convolution, Image from [3] 4
2.2 Depthwise Convolution, Image from [3] 4
2.3 Pointwise Convolution, Image from [3] 4
2.4 SSD default boxes, Image from [7] 5

3.1 Intel Movidius Neuronal Compute Stick (NCS), Image from [6] 8
3.2 NCSDK Workflow, Image from [9] 8
3.3 OpenVINO Workflow, Image from [10] 10
3.4 AAEON UP AI Core X 12

4.1 Hardware in comparison with a pen 13
4.2 Overview of the application 19
4.3 Screenshot of the Application 31
4.4 Screenshot of Application with opened Menu 31
4.5 Screenshot of the Application on a PC 32

5.1 Inference result on the CPU 42
5.2 Inference result on the VPU 42
5.3 Car and Person Inference with SSD Mobilenet V1 Coco network 43
5.4 Car and Person Inference with assistant system network . . . 43
5.5 Car and Person inference with SSD Mobilenet V1 Coco network 44
5.6 Car and Person inference with assistant system network . . . 44
5.7 Dumpster inference with SSD Mobilenet V1 Coco network . . 45
5.8 Dumpster inference with assistant system network 45
5.9 Motorcycle inference with SSD Mobilenet V1 Coco network . 45
5.10 Motorcycle inference with assistant system network 45
5.11 Street with cars. Picture taken from view of a walking person 46
5.12 Street with cars. Picture taken from ground level. 46
5.13 Car from above. Inference with assistant system network. . . 48

5.14 Car from below. Inference with assistant system network. . . 48
5.15 Car from above. Inference with SSD Mobilenet V1 Coco. . . . 48
5.16 Car from below. Inference with SSD Mobilenet V1 Coco. . . . 48
5.17 Person from above. Inference with assistant system network. 49
5.18 Person from the middle. Inference with assistant system network. 49
5.19 Person from below. Inference with assistant system network. 49
5.20 Person from above. Inference with SSD Mobilenet V1. 49
5.21 Person from the middle. Inference with SSD Mobilenet V1. . 49
5.22 Person from below. Inference with SSD Mobilenet V1. 49
5.23 Pavement edge further away 51
5.24 Wall behind pavement edge 51
5.25 Pavement/Wall Edge from above 52
5.26 Pavement/Wall Edge from below 52

Listings

4.1 Camera Publisher initialization 21
4.2 Camera Publisher Main Loop 21
4.3 ZMQ receiver setup for image 23
4.4 ZMQ receiving of image 24
4.5 Build Script for Object Detection Inference 25
4.6 Script to run Object Detection Inference 26
4.7 Camera Class for Flask Video Streaming 27
4.8 Websocket Server 28
4.9 Websocket Client 33
4.10 Speech Synthesis with Web Speech API 35
4.11 Vibration Feedback with Vibration API 37

List of Tables

3.1 Comparison of NCS vs NCS2 11
4.1 Files of the saved model for inference 23
5.1 Comparison of inference time 40

1 Introduction

In the last few years several problems in computer vision have seen a huge enhancement in terms of quality. Many of these improvements are due to the rise of powerful neuronal networks. Computer vision related tasks are part of many products we use daily. At the same time more and more people are using mobile devices such as smartphones, tablets, drones or other embedded devices. The inference of neuronal networks is a computational expensive process. Especially mobile devices have limited computing power and the power consumption of the chips should be as low as possible to extend battery life.

The Intel Movidius Neural Compute Stick (NCS) is a chip designed to make the inference of neuronal networks fast and energy-efficient. In this thesis this chip will be used to run a classification and object detection network on a Raspberry PI. A small camera is attached directly to the Raspberry PI to record the images.

The object detection network is able to detect the following categories: car, person, bike, motorcycle, dumpster and bus. Using this network more than one object can be detected at the same time. In the case of objects covering the whole image, a classification network is used. In that case, the network is trained to detect walls, stairs and the edge of a pavement.

For robotics and mobile assistant systems it is important that potentially dangerous objects can be detected. As a showcase in this thesis spoken image descriptions will be used. Other ways to detect objects in an environment can be by ultrasonic sensors. However, this ultrasonic sensors fail with certain types of obstacles. For example, stairs going down, or moving objects that are further away such as cars, motorcycles, buses or bicycles cannot be reliably detected with the current systems.

In this thesis these limitations will be solved with the help of Convolutional Neuronal Networks (CNNs). CNNs already proved that they are very powerful detecting objects. For this specific task and the categories mentioned above a network has already been trained in a project executed at the Technical University of Graz. This network is converted so that it is able to run on the Neural Compute Stick (NCS) on the Raspberry PI.

The user can connect a phone to the Raspberry PI and receive feedback over the phone. The communication with the user is done with a text-to-speech output and vibration feedback. For testing purposes, the detection results are also shown in real-time on the live image on the phone.

In chapter 2 the models used for the object detection 2.2 and the classification 2.3 are described in detail. Chapter 3 gives details about the structure and the use of the Intel Movidius NCS. Chapter 4 describes the implementation of the project. Especially the hardware used, the transformation of the network and the web application with text-to-speech functionality. In chapter 5 the results are evaluated and compared with similar models.

2 Models

The models used in this thesis need to run in real-time on an embedded device, to generate real-time spoken image descriptions.

The base network used for the object detection (section 2.2) as well as for image classification (section 2.3) is the MobileNet network [4]. Compared to other Convolutional Neural Networks (CNNs) the inference is faster. The architecture of MobileNet is described in detail in section 2.1.

Both networks were trained in a foregoing project at Graz Technology of University. The implementation is done in Tensorflow. These models were trained on training data gathered specially for a mobile assistance system. In this thesis these existing networks were transferred to run on the Intel Movidius Neural Compute Stick.

2.1 MobileNet

Convolutional Neural Networks (CNNs) are very popular. The networks tend to get deeper and more complicated to achieve higher accuracy. However on limited hardware it is hard to run time critical applications.

MobileNet V1 is designed to achieve a significant speedup with similar accuracy as regular CNNs. The main idea behind MobileNet are depthwise separable convolutions.

2.1.1 Depthwise Separable Convolutions

Most images have 3 input channels (RGB). Regular convolutions apply a filter over all 3 input channels and output a weighted sum of the input pixels over all 3 channels. See figure 2.1.

The depthwise separable convolution instead consists of two separate steps. The depthwise convolution and the pointwise convolution.

The depthwise convolution performs a 2D convolution on every channel separately. See figure 2.2. In a second step these 3 channels get combined with a pointwise convolution. This is a regular convolution with 1x1 kernel. See figure 2.3.

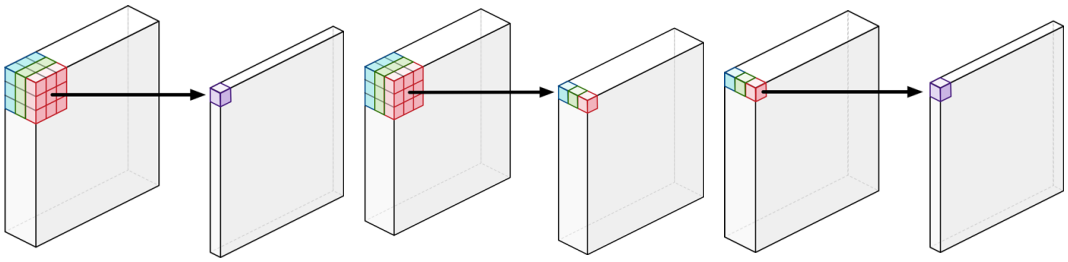

Figure 2.1: Regular Convolution, Image from [3]

Figure 2.2: Depthwise Convolution, Image from [3]

Figure 2.3: Pointwise Convolution, Image from [3]

Splitting these two operations have a similar result, but the computation is much faster. In [4] it is shown that the speedup for 3x3 kernels is about the factor of 9, with similar results on accuracy.

The MobileNet architecture consists of 14 layers in total. The first layer is a regular 3x3 convolution with 32 filters. The following 13 layers are depthwise separable convolutions with 3x3 filters and a stride of 1 or 2.

2.2 Object Detection

For object detection the Single Shot MultiBox Detector (SSD) architecture is used. [7]

The SSD architecture is able to run in real-time and achieves similar accuracy like other object detection networks. In [7] it is shown that SSD is faster than Region-CNN (R-CNN), Fast-R-CNN and Faster-R-CNN, reaching similar accuracy.

The name of the SSD architecture comes from:

Single Shot
: Classification and object localization are done in a single forward pass network.

Multibox
: Is a technique used for bounding box regression. In SSD this technique is enhanced by choosing fixed priors.

Detector
: The detected objects also get classified into certain classes.

In SSD fixed priors are chosen. Two examples for a 8x8 grid and 4x4 grid can be seen in figure 2.4. Every feature map cell has default bounding boxes of different aspect ratios and size associated. For every prior the delta to the bounding box together with the confidence of the predicted class is calculated.

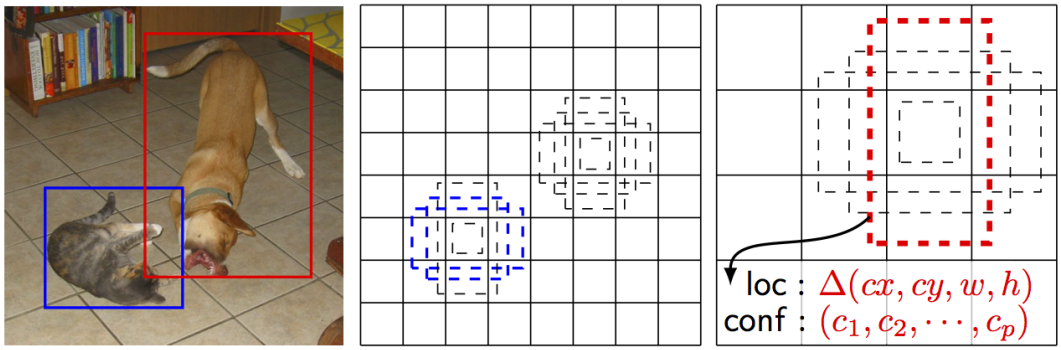

Figure 2.4: SSD default boxes, Image from [7]

The bounding boxes get calculated at 6 different scales. For the first scale 3 different aspect ratios are used. For the following scales 6 different aspect ratios are used.

As a base network to generate the feature maps also MobileNet described in section 2.1 is used.

2.3 Classification

For the classification also the MobileNet network, described in section 2.1 is used. The output of the MobileNet is a vector of dimension 1024. The classification is done by placing a fully connected layer with 3 output classes as the last layer. The classes that can be detected are stair, pavement edge and wall.

During training the sigmoid activation function was used. Therefore for each class the output can be between 0 and 1. A single image can therefore be classified in multiple classes.

Often in classification also the softmax activation function is used. In that case the sum of all values add up to one. There is only one class that has high values at a time. To be able to not detect any class, a ”empty” class is added.

3 Intel Movidius Neural Compute Stick

Intel Movidius Neuronal Compute Stick (NCS) is a chip specialized for the inference of Deep Neural Networks (DNN). It offers high performance and very good power efficiency. With the help of the NCS state-of-the-art computer vision applications can run on embedded devices. [6]

The Intel Movidius is a so called Vision Processing Unit (VPU) which allows highly parallel programmable computing with workload-specific hardware acceleration. The chip consists of 12 SHAVE cores. SHAVE cores support a mixture of different types of instructions. [19]

In contrast to the GPU the VPU does not contain specialized hardware like rasterization or texture mapping.

The chip is available in regular chip packages to include it into custom applications. For fast prototyping the NCS is perfect. It includes the chip with the needed peripherals in the form of a USB stick. The NCS is shown in figure 3.1.

Before an inference can run on the NCS, existing network typologies, including the weights and biases, need to be compiled. Currently there are two different frameworks available to do that. These frameworks are described in section 3.1 and 3.2.

3.1 NCSDK

The Neural Compute SDK (NCSDK) consists of tools which allow you to build custom applications with the Neural Compute Stick. A graphical

Figure 3.1: Intel Movidius Neuronal Compute Stick (NCS), Image from [6]

overview of the tool set is given in figure 3.2.

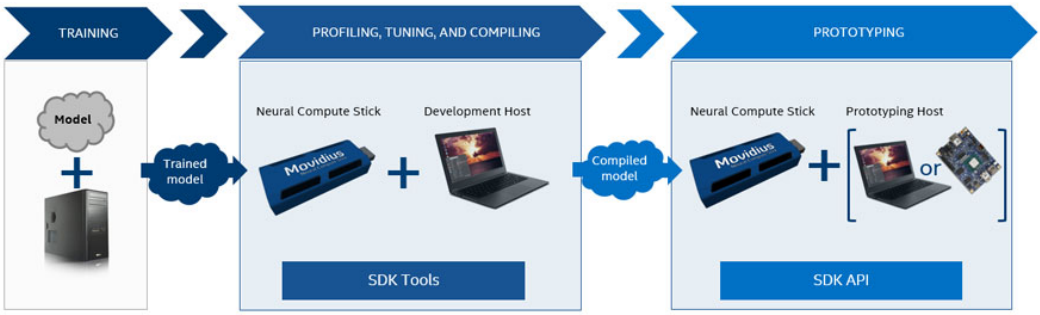

Figure 3.2: NCSDK Workflow, Image from [9]

The training of the network is done on the GPU of a host PC. It can be done with the frameworks Caffe or Tensorflow. The model definition together with the corresponding weights are then converted into an intermediate format for the stick. This is done with the mvNCCompile tool. The output is a .graph file which can be later loaded on the stick for inference. Using mvNCCompile one can check if the network behaves the same on the NCS as on the host PC. mvNCProfile can check how long each layer takes to compute.

As soon as the network is compiled the NCAPI can be used for inference on the stick. The API is available for Python an C++. The .graph file can be loaded to the Neural Compute Stick and an inference can be started. At run

time all the original models and weights are not needed anymore, as all the information is included in the .graph file. If the network is updated and the input and the output layers stay the same, only the .graph file needs to be replaced.

The model of the classification, described in section 2.3, was converted with the NCSDK compiler.

3.2 OpenVINO

During the realization of this master thesis Intel offered a new framework which is compatible with the NCS. [5]

Open Visual Inference and Neuronal Network Optimization toolkit (OpenVINO) is a tool which allows developers to achieve improved neural network performance on a variety of Intel processors to allow real time vision applications. The toolkit enables deep learning inference among different hardware.

The general idea of the OpenVINO Toolkit is that models from different deep learning frameworks such as Caffe, Tensorflow, MXNet, Kaldi and more, are converted into an intermediate format. This intermediate format can then be used for inference on all kinds of different hardware. Among the supported hardware the Neural Compute Stick is also included. At the time of the writing of this thesis the following hardware is supported. [10]

- Intel® CPU
- Intel® Processor Graphics
- Intel® FPGAs
- Intel® Movidius™ Neural Compute Stick
- Intel® Gaussian Mixture Model

The workflow is similar to the method used by the NCSDK (see section 3.1). A graphical model is shown in figure 3.3.

First a model is trained with one of the supported frameworks. For this thesis Tensorflow models which were already trained for this specific problem

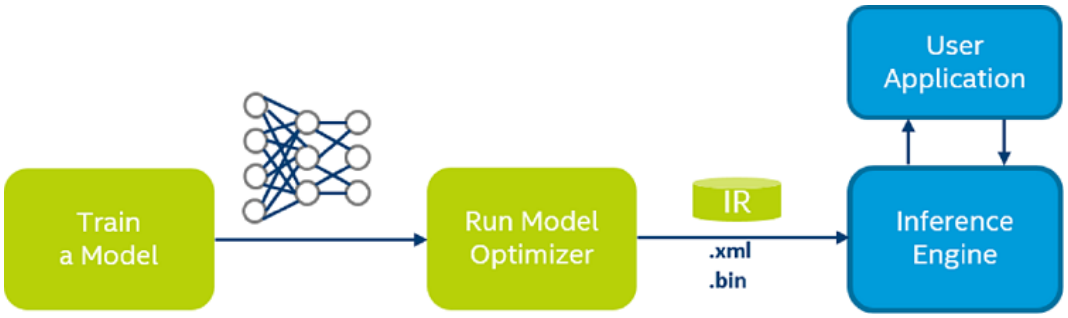

Figure 3.3: OpenVINO Workflow, Image from [10]

were used. The OpenVINO framework then offers two tools, the model optimizer and the inference engine.

Model Optimizer

The model optimizer creates the intermediate representation of the trained model. It takes the network topology and the weights as input and automatically adjusts the deep learning models for optimal execution. The output of the model optimizer is independent of the hardware used for inference. The layers used by the model need to be supported by the model optimizer. The model optimizer supports the following frameworks.

- Caffe
- TensorFlow
- MXNet
- Kaldi
- ONNX

The intermediate representation describes the model with two files:

.xml describes the topology of the network.
.bin contains the weights and biases in form of binary data.

Inference Engine

The inference engine takes the intermediate representation, which was generated with the help of the model optimizer as an input. The inference engine offers a unified API to integrate with the application logic. Furthermore, it optimizes the inference for the execution at the

targeted hardware. The inference engine also runs on a Raspberry PI, and is optimized for embedded platforms.

The OpenVINO framework is used for the object detection model, which is described in section 2.2.

3.3 Comparison of the NCS vs. NCS 2

During the writing of the thesis Intel released the successor of the Neural Compute Stick (NCS), the Neural Compute Stick 2 (NCS 2).

The NCS 2 is based on the new Intel® Movidius™ Myriad™ X VPU, whereas the NCS was based on the Intel® Movidius™ Myriad™ VPU.

Intel claims that the NCS 2 is up to 8 times faster than the NCS. To see if a speedup for the networks used in this thesis can be achieved, the NCS 2 was purchased. However, no real speedup for the used networks could be seen.

	Myriad 2	Myriad X
SHAVE Cores	12	16
Enhanced ISP with 4K support	no	yes
Vision Accelerators including Stereo Depth	no	yes

Table 3.1: Comparison of NCS vs NCS2

3.4 Different form factors of Myriad VPUs

The Myriad VPUs are built to enable energy-efficient and fast inference on embedded devices. However, the inference on this special VPUs are more energy-efficient than on a GPU or a CPU. This could also be a reason to use the VPUs in regular computers or servers, where the inference is usually done on the GPU or on the CPU.

The company AAEON UP focuses on bringing the Myriad VPUs into regular computers. They offer one or more chips on an expansion board which can be easily connected over mPCIe, M.2 2230, M.2 2242, and M.2 2280.

Figure 3.4: AAEON UP AI Core X

My personal opinion is, that in future we will see VPUs like the Intel Movidius Myriad integrated into regular computers or smartphones by default. As Intel already has frameworks like OpenVINO to support efficient inference on different Intel devices and they also own the Movidius VPUs, it would be easy for them to package an Intel VPU together with an Intel CPU into a single chip. This could be an additional boost for the deep learning hype.

4 Implementation

4.1 Hardware Setup

The hardware is shown in figure 4.1. It is a small portable device powered by a rechargeable battery. In figure 4.1 it is shown in comparison with a pen to get a feeling for the size of the device. Due to the small size, it would be possible to integrate this hardware directly into a mobile system like a small robot. However, that is not part of this thesis.

Figure 4.1: Hardware in comparison with a pen

The basis of this device is a Raspberry PI (RPI) with an Intel Movidius Neural Compute Stick connected to the USB port of the RPI. The RASP CAM2 camera is directly connected to the Raspberry PI over a serial MIPI-Interface. The camera is able to take images with a resolution of 1080p @ 30fps, although this high resolution is not needed for the current use case. As the object detection runs with about only 5 frames per second, the image is also recorded with about 5 frames per second.

The Raspberry PI and the camera are mounted in a black case. The camera sits on top of the case. The angle of the camera can be adjusted and turned approximately 45 degrees.

The whole setup is powered by a 5 volt rechargeable battery. This makes it much easier to carry the setup around and use it to create test pictures for the evaluation in chapter 5. It also shows that the power consumption is low enough to power the device for several hours. When connecting the power pack to the Raspberry, it is important that a short and sufficiently thick cable is used. When starting the inference the Neural Compute Stick shows a high current flow. If the resistance of the cable is too high, the voltage drops and the Raspberry PI reboots. This happened when a regular USB cable, usually used to charge a phone, was used. However, when using the short and thick cable no problems occurred anymore. In addition, make sure that the power pack in use can deliver a current of at least 2A.

The hardware can be built even smaller when creating a custom Printed Circuit Board (PCB). Doing this was not on the scope of this thesis. However, a first start would be using a Raspberry Zero together with the Myriad^{TM} 2 or Myriad^{TM} X VPU. The software created in this thesis should run on this setup without major modifications.

4.2 Converting the models

4.2.1 Classification

The classification network is based on MobileNet architecture, which is described in section 2.3. To convert the network the NCSDK toolkit is used

which is described in section 3.1.

The mvNCCompile tool needs a frozen model file to create the graph file which then can be used for inference.

To create the frozen model for inference the graph definition file containing the network architecture and the checkpoint file containing the weights and biases of the model is needed.

To get the graph definition file the *export_inference_graph.py* function from the Tensorflow repository is used. The following command was used to create the graph definition file.

```
export_inference_graph.py --alsologtostderr --
    model_name=mobilenet_v1 batch_size=1 dataset=
    imagenet --image_size=224 --output_file
    mobilenet_v1_1.0_224.pb
```

In the next step the model definition file together with the checkpoint file is used to generate the frozen model. The following command was used to create the frozen model.

```
python3 ~/ncappzoo/tensorflow/tf_src/tensorflow/
    tensorflow/python/tools/freeze_graph.py --
    input_graph=mobilenet_v1_1.0_224.pb --input_binary=
    true --input_checkpoint=model.ckpt --output_graph=
    mobilenet_v1_1.0_224_frozen.pb --output_node_name=
    MobilenetV1/Predictions/Reshape_1
```

This frozen model can then be used to compile it for the NCS with the mvNNCCompile tool.

```
mvNCCompile -s 12 mobilenet_v1_1.0_224_frozen2.pb -in=
    input -on=MobilenetV1/Predictions/Reshape_1
```

The result is a graph file which contains the model definition and the weights. This graph file can be loaded on the Neural Compute Stick and used for inference.

4.2.2 Object Detection

The object detection network is based on SSD MobileNet, which is described in section 2.2. To convert the network the model optimizer of the OpenVINO toolkit described in section 3.2 is used.

The original model, which was trained by Dipl.-Ing. Christian Ertler, bases on the SSD MobileNet Coco network from the Tensorflow detection model zoo. [8] The pre-trained frozen model was taken from his last experiment from the 14th of June of 2018. It can be found in the following folder of his project.

`/walkassist_object_detection/experiments/2018-06-14_17-44-15/evaluation/frozen_infreence_graph.pb`

The OpenVINO model optimizer supports converting models from the official Tensorflow detection model zoo. [12] The command line tool of the model optimizer supports many parameters that need to be set correctly to convert the model:

–input-model
: Is the path to the pre-trained frozen model file. The file was used without modifications from Dipl.-Ing. Christian Ertler.

–tensorflow_use_custom_operations_config
: This file describes the rules how to convert Tensorflow topologies. For several common topologies files have been recreated by Intel. It was not able to convert the network using the files created for SSD topologies from the model zoo. Later it was found out that it was not working due to the use of an old version of SSD MobileNet Coco.

–output
: Allows to cut off the topology. The output of this network will be the node names *detection_boxes*, *detection_scores*, *num_detections*.

–data_type
: This parameter needs to be set to *FP16* when compiling for the Intel Movidius NCS. When running the inference on the CPU, this parameter must not be set.

After changing several parameters in the custom operations configuration file, the network still did not compile without any errors. The network used

by Dipl.-Ing. Christian Ertler is based on the *ssd_mobilenet_v1_coco_2017_11_17* version. The conversion was possible without any problems for the slightly newer version *ssd_mobilenet_v1_coco_2018_01_28* with the current framework at that time OpenVINO 2018 R3.

After returning to the OpenVino 2018 R2 (2018.2.300) version and using the corresponding custom operation files for the SSD typology the conversion was possible.

The conversion on a regular computer takes about 10 seconds. A *frozen_inference_graph.bin*, *frozen_inference_graph.xml* and *frozen_inference_graph.mapping* is generated, which will be used for inference then.

The following command was used to convert the model. All paths and parameters are included to easily reproduce it. In order to compile for the CPU, the *data_type* parameter needs to be removed.

```
~/intelr2/computer_vision_sdk_2018.2.300/
    deployment_tools/model_optimizer/mo_tf.py --
    input_model ~/walkassist_object_detection/object-
    detection/experiments/2018-06-14_17-44-15/evaluation
    /frozen_inference_graph.pb --
    tensorflow_use_custom_operations_config ~/intelr2/
    computer_vision_sdk_2018.2.300/deployment_tools/
    model_optimizer/extensions/front/tf/ssd_support.json
     --output="detection_boxes,detection_scores,
    num_detections" --data_type FP16
```

4.3 Application

The application running on the Raspberry PI consists of four modules written in several programming languages. The different modules, running in separate processes, communicate over ZeroMQ (ZMQ) [20].

ZeroMQ allows to connect code over any platform and supports messaging patterns like publisher-subscriber. ZMQ carries messages across inproc, IPC, TCP, TIPC or multicast. In this thesis TCP is used.

The four main parts of the software are the following:

Camera Publisher
: The camera publisher reads the frames from the RPI camera and publishes them.

Inference Publisher
: The inference publisher reads the frames from the camera publisher and runs the object detection on the Neural Compute Stick.

Web Server
: The web server reads the results from the inference publisher and serves a web page to the user. It publishes the inference results over a websocket connection and creates a video stream of the processed images.

Web Application
: The user can open the web application on a phone, PC or tablet. The web application shows the live image stream already with the inference results and gives feedback to the user through vibrations as well as spoken notifications. To save bandwidth the built-in text-to-speech engine of the browser is being used.

In figure 4.2 a graphical overview of the application is given. The communication between the three processes running on the Raspberry PI is done with ZMQ TCP connections. The communication between the web application and the web server is done with a websocket connection.

The camera and the NCS are directly connected to the Raspberry PI. The web application runs on an external device (smartphone, tablet or PC) and is the interface to the user. The communication with the web server is done over WiFi. There are basically 3 ways how the WiFi connection between the mobile device and the Raspberry PI can be established. Each of them has different advantages and disadvantages.

Raspberry creating WiFi
: The Raspberry creates a WiFi network and every mobile device can connect to this network to establish a connection with the web server. The main advantage is the very easy setup, as the user just needs to connect to the WiFi, and the IP of the Raspberry can be static. The disadvantage is that the WiFi of the Raspberry does not have

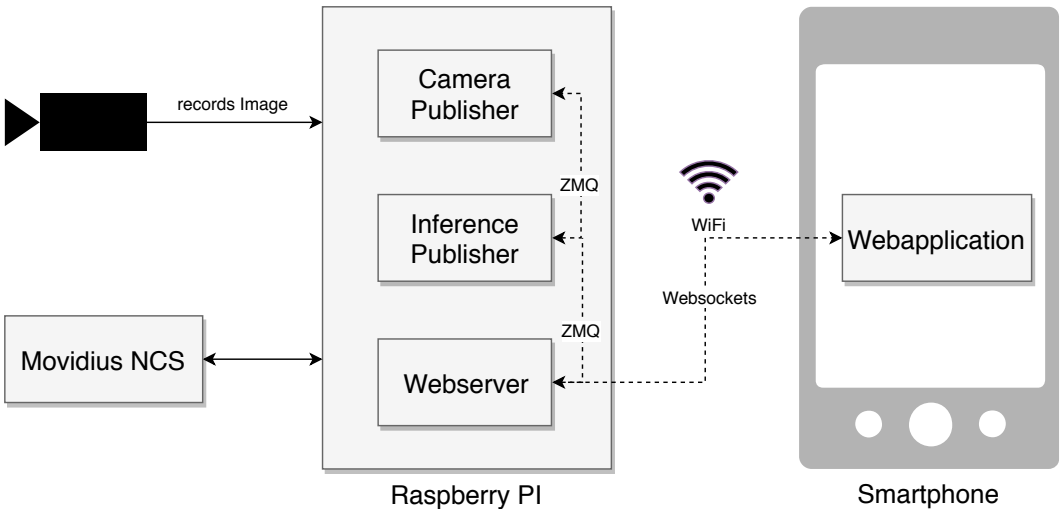

Figure 4.2: Overview of the application

a connection to the Internet. Most smartphones, when connected to a WiFi, try to route all the traffic through the WiFi. Therefore the smartphone also loses connection to the internet. Other services such as WhatsApp, Google Maps or others will stop working.

Smartphone creating Hotspot

The smartphone or a different mobile device creates a WiFi network. The Raspberry connects to it. The user then browses to the IP address of the Raspberry to start the web application. The advantage in that case is that the Raspberry, as well as the smartphone have internet connection. The disadvantage is that the setup process is a bit more complex. First of all, the WiFi credentials of the hotspot need to be set on the Raspberry PI. And secondly the IP-address that the Raspberry got over DHCP needs to be known. However, most smartphones show the IP-addresses of the connected devices.

Connection through the Internet

The Raspberry PI and the smartphone could also have independent connections to the Internet. For example, by having the Raspberry connected to a public WiFi or connecting a modem with a SIM card. Both, the Raspberry PI as well as the smartphone, can connect to a relay server. All the traffic is routed through this relay server. The

advantage is that remote computers and smartphones can also connect easily. The disadvantage is that a lot of mobile data and good cellular connectivity is needed, as also the image of the camera is transferred via the Internet.

In this thesis the second option - smartphone creating hotspot - was chosen. The reason for this is that it makes sense to have a direct connection between the smartphone and the Raspberry. In the first case - Raspberry creating WiFi - the smartphone as well as the Raspberry would not have an internet connection. This was considered as a big disadvantage.

If the Raspberry would also have internet connection, the system could be enhanced with some additional features. In case the object detection fails or a person or robot needs assistance on the road, a connection to a helpline can be established. This can be triggered by shaking the smartphone or some other easy command. The helpline can access the image of the Raspberry camera and guide the robot or person over voice. As long as computer vision methods are not perfect yet, this hybrid mode is a very interesting one.

4.3.1 Camera Publisher

The camera publisher records the image of the Raspberry's camera and publishes the frame so that the inference publisher can read it and perform the inference on it. This module is written in C++.

For this application it is very important that the user or robot gets a notification in real time, for example, when a car appears in the picture. It is, therefore, critical that there is no delay introduced anywhere in the system. The next processing step needs the frame in the format of an OpenCV frame to pass it on for inference.

When capturing the frames with the standard python library *picamera* in the tests, the image was always 2-3 seconds delayed. As mentioned above this is not suitable for this application. [13]

Several other approaches and libraries were tried out, but they all had similar issues. The open source C++ library RaspiCam is used in this thesis

to read the Raspberry's camera. This library works best and does not add any delays, which is critical for the purpose of this project. [14]

In listing 4.1 the *Raspicam_CV* object is created and the ZMQ publisher to publish the frame on TCP port 5559 is set up. The object detection network was trained for a resolution of 500 x 500 pixels. Therefore, the inference is done with a similar resolution. Moreover, the final stream transferred to the web application should not have a resolution which is too high. Otherwise, too much bandwidth is needed to transfer the image and the application lags. Therefore, the image is captured at a resolution of 640 x 480 pixels.

```
raspicam::RaspiCam_Cv Camera;
Camera.set(CV_CAP_PROP_FRAME_WIDTH, 640); //320
Camera.set(CV_CAP_PROP_FRAME_HEIGHT, 480); //240

Mat frame;

zmq::context_t context(1);
zmq::socket_t publisher(context, ZMQ_PUB);
publisher.bind("tcp://*:5559");
```

Listing 4.1: Camera Publisher initialization

In listing 4.2 the main loop of the camera publisher is shown. It grabs an image from the camera, displays it for debugging reasons and publishes the frame with ZMQ. The frame object is copied with memcpy and transferred in binary format. The total length of the object is calculated by multiplying the number of pixels by the number of channels.

```
for(;;) {
    Camera.grab();
    Camera.retrieve (frame);
    imshow("Camera Publisher Frame", frame);

    int len = frame.total() * frame.channels();

    zmq::message_t message(len);
    memcpy (message.data(), frame.data, len);
    bool rc = publisher.send(message);
```

```
        int k = waitKey(10);
        if(k == 'q'){
            break;
        }
    }
```

Listing 4.2: Camera Publisher Main Loop

The Raspberry camera is not conncted like a regular USB camera. So there is no `/dev/video0` or similar device file. If regular USB cameras were used, the standard OpenCV functions to read from a camera could be used. In case one wants to exchange the camera, only the camera publisher needs to be modified and the image will be published to Port TCP 5559. The rest of the software stays identical.

In chapter 5 different viewing angles of the camera are evaluated. It turns out that the viewing angle from the bottom is better for some objects, whereas the viewing angle from the top is better for other objects and networks. In further steps more than one camera could be connected. For example, one from above and another one from underneath. One camera could even be placed from behind, to warn people or robots of bikes, cars or other objects coming from behind.

The camera publisher always needs to be started. Otherwise the input for the other modules is missing. The module is started by running the script *run.sh*. To compile the module use cmake. The provided *CMakeLists.txt* file links OpenCV, RaspiCam and ZMQ to the executable. The according libraries need to be installed first.

4.3.2 Inference Publisher

The inference publisher does the actual inference on the Neural Compute Stick. It receives the recorded image from the camera publisher on port 5559. It processes the image and publishes it together with the bounding boxes of the detected object on Port TCP 5558. The meta data of the detected bounding boxes (coordinates, probability and class) are also published on

port TCP 5557. All this information will be available in the web application afterwards and the user can decide for which objects and for which minimum confidence level they want to be warned.

For the inference the model files generated in section 4.2 are needed. The original model is not needed at run time. The content of the model files used is described in table 4.1.

File	Content
frozen_inference_graph.bin	The weights and biases of the trained model.
frozen_inference_graph.labels	The names of the labels of the model.
frozen_inference_graph.xml	The network definition of the model.

Table 4.1: Files of the saved model for inference

The code is based on the *object_detection_demo_ssd_async* example from the OpenVINO inference engine samples. [11]

The code is written in C++ and the inference is processed asymmetrical. This boosts the performance as the second image is loaded while the inference of the first image is done.

In listing 4.3 a OpenCV Matrix is created and a subscriber listening on TCP port 5559 is opened. The *ZMQ_CONFLATE* option ensures that only the last message is kept in the receiving queue. So always the newest available frame gets processed. This is important to keep the latency low, to achieve a real time notification of dangerous objects.

```
cv::Mat curr_frame(480, 640, CV_8UC3);
cv::Mat next_frame(480, 640, CV_8UC3);

zmq::context_t context(1);
zmq::socket_t subscriber(context, ZMQ_SUB);
int conflate = 1;
subscriber.setsockopt(ZMQ_CONFLATE, &conflate,
    sizeof(conflate));
subscriber.connect("tcp://localhost:5559");
subscriber.setsockopt(ZMQ_SUBSCRIBE, "", 0);
```

Listing 4.3: ZMQ receiver setup for image

In listing 4.4 the actual image is received and copied over to the OpenCV matrix. No conversation is needed. The binary data can be used as it is, as the used data structure is the same. Tests showed that in this case this is the best way of sending an image. Converting it to JPEG and decoding it again does not only lower the quality of the image, but also makes the application slower.

```
        zmq::message_t message;
        subscriber.recv(&message);

        memcpy(curr_frame.data, message.data(), message
            .size());
```

Listing 4.4: ZMQ receiving of image

In the code all the setup needed to prepare an inference is done. The most important steps are the following:

Load Plug-in for Inference Engine
The OpenVINO Framework supports different hardware for inference. This code works with different hardware. In the standard case the plug-in for the NCS is loaded.

Read IR Generated by Model Optimizer
In the next step the intermediate representation generated by the model optimizer is read. The files read are also described in table 4.1.

Configure Input and Output
The input and output blobs are configured to accept an image and return the results of the SSD.

Loading Model to the Plug-in
The model is loaded to the plug-in. When the plug-in used is the NCS, the intermediate representation is transferred to the Neural Compute Stick. One could say that the Neural Compute Stick is now programmed to do the inference.

Create Infer Request
The inference request is generated.

Do Inference
In an endless loop the next image will be taken and the inference will be started. The result of the inference is written in an array which is

then published. Also the image gets edited so that the bounding boxes can be seen together with the labels in the image. This edited image is also published via ZMQ.

The output of the inference for each frame is a list of possible objects. Each result in the list includes the following data.

- Image ID
- Label (car, bus, person, bike, motorcycle, dumpster)
- Confidence
- Coordinates (xmin, ymin, xmax, ymax)

All possible objects are published. Later on, in the web application the interesting classes can be filtered. Also the user can set from which confidence level onward one would like to be notified of objects.

Compiling for the Raspberry

The code should be running on the Raspberry. While running the compiler it is important to set the *DCMAKE_CXX_FLAGS* and to set the target platform to *armv7-a*, which is the chipset that powers the Raspberry.

The listing 4.5 shows a bash script with which the code can be compiled for the Raspberry.

```
#!/bin/bash

build_dir=$PWD/build
mkdir -p $build_dir
cd $build_dir
cmake .. -DCMAKE_BUILD_TYPE=Release -DCMAKE_CXX_FLAGS="
    -march=armv7-a"
make -j2 object_detection_demo_ssd_async
```

Listing 4.5: Build Script for Object Detection Inference

Also this module uses cmake to generate the make file.

Running the Module

For inference different hardware can be used. In this thesis we focus on the Intel Movidius Neural Compute Stick. However, using the OpenVINO Framework the inference can also be run on the GPU or on the CPU.

In listing 4.6 the bash script to run the inference on the Neural Compute Stick is shown. The option -d MYRIAD is important to run the inference on the Neural Compute Stick. Other options for -d would be:

- CPU
- GPU
- FPGA
- HDDL
- MYRIAD

```
#!/bin/bash

./build/armv7l/Release/object_detection_demo_ssd_async
    -i cam -m model/frozen_inference_graph.xml -d MYRIAD
```

Listing 4.6: Script to run Object Detection Inference

4.3.3 Web Server

On the raspberry a web server is also running. The user can connect to this web server to open the web application which is described in section 4.3.4.

The web server serves the HTML, CSS and JavaScript files, but also the dynamic content like the livestream of the camera image and the data of the current inference results.

This module is written in python, and Flask is used as the web server.

Videostream

This web server receives the image from the inference publisher on port TCP 5558. Also in this module ZMQ with the conflate option is used, to only take the latest available inference result.

From the received image a live video stream is generated which can then be played by the user with an HTML5 player in the web application.

The concept of building a video streaming server with flask was taken from [2]. A separate camera class was implemented that takes the single images from the ZMQ queue and returns them. This can be seen in listing 4.7.

```
class Camera(BaseCamera):
    """An that streams images from ZMQ"""

    @staticmethod
    def frames():
        context = zmq.Context()
        socket = context.socket(zmq.SUB)

        socket.connect("tcp://localhost:5558")

        socket.setsockopt(zmq.SUBSCRIBE, b'')
        socket.setsockopt(zmq.CONFLATE, 1)

        for i in range(1,3000000):
            msg = socket.recv(0, True, False)
            buf = buffer(msg)
            A = np.frombuffer(msg, dtype="uint8")
            B = A.reshape((480, 640, 3))

            yield cv2.imencode('.jpg', B)[1].tobytes()

        print "Done"
        cv2.destroyAllWindows()
```

Listing 4.7: Camera Class for Flask Video Streaming

Static Pages

The web server will also serve the static content for the web application, like the images, CSS and JavaScript files. In section 4.3.4 it will be discussed more about these files and their functionality.

The file index.html is the start page of the web application. It is also a static page which gets served by the flask web server.

websocket

All the inference results get published to the web application over a websocket connection.

Websockets are based on TCP and are a bidirectional connection between the web browser (web application) and the web server. Websockets allow the web server to push new information to the web client, without the web client asking for new information or reloading the page. This feature is extremely useful on this project to provide the user with a live connection.

In listing 4.8 every received message from port 5557, gets published on the websocket connection and can then be received by the web application. The websocket part is running in a separate thread, so that the application can still serve the static content at the same time.

All the clients connected to the websocket server receive the same messages.

```
def update_thread():
    while True:
        #socketio.emit('message', 'was gekommen')
        #time.sleep(5)
        socket = context.socket(zmq.SUB)
        socket.connect("tcp://localhost:5557")
        socket.subscribe("")

        while True:
            update = socket.recv_string()
```

```
        print update
        socketio.emit('message', update)
```

Listing 4.8: Websocket Server

Shutdown Event

When powering the device (connecting it to the power pack) the operating system starts and all the modules of the application get started automatically. It is described in more detail in section 4.3.5.

As there are no buttons or other input methods on the Raspberry itself, there is no way to tell the Raspberry that it should switch itself off again. In the case of just disconnecting the power bank, the file system can get corrupted, so that a boot would then not be possible anymore.

For this reason, in the web application there is a designated button to switch of the Raspberry. By browsing to the IP of the device and making a GET request to the route /shutdown, the device will switch off.

4.3.4 Web Application

The web application is the interface to the user. In the web interface the user gets all the important information and feedback, that the network generates for him. An spoken description of the image is given. Optionally a vibration can be generated as soon as certain class appears.

Additionally, the web application serves as a configuration tool for the user. The user can activate and deactivate the speech and vibration output for separate classes. Also, the minimum time between notifications can be set dynamically. The default value for a notification of the presence an object is a confidence level of 50%. This can be lowered by people who want to be notified earlier, but it can also have the risk of getting more false notifications.

User Interface

The user interface is very clean and easy to use. It is responsive and, therefore, can be used on every screen size. Most of the time the interface will be most probably used on a smartphone.

In figure 4.3 a screenshot of the application on a smartphone can be seen. In the center the live image, recorded by the camera, can be seen. In this example there is a person and two cars in the picture and the objects are clearly identified and marked with a red bounding box.

On top of the image there are icons of all the possible classes (bus, car, bike, motorcycle, person and dumpster). As soon as one of these objects appears in the image, the corresponding icon turns green, indicating that at least one object of this class has been recognized in the image. If more than one object of the same class is in the image, the number below the icon indicates it accordingly.

Below the image there are messages in the form of a chat. It shows the history of the object detections. When a new object is detected, a new message describes what can be seen in the image. For instance, in this case the message says: "Be careful there are two cars and a person." This message will also be spoken in the case that the corresponding settings is enabled.

In figure 4.4 the settings sidebar is shown, in this case, on a smartphone screen. At the top the user can set the confidence level sliding the bar. Only bounding boxes with a higher confidence than the set one are shown to the user. Also notifications (text-to-speech output and vibrations) will only be given if the confidence level is high enough.

The user is also able to switch on and off the text-to-speech output and the vibration output for all the different categories.

When walking in a busy or crowded place, a lot of objects will be detected. In order to avoid giving too many notifications in very a short time, hindering the user experience, the user can set a minimum time between that must elapse between notifications. This time can be set between 2 and 20 seconds.

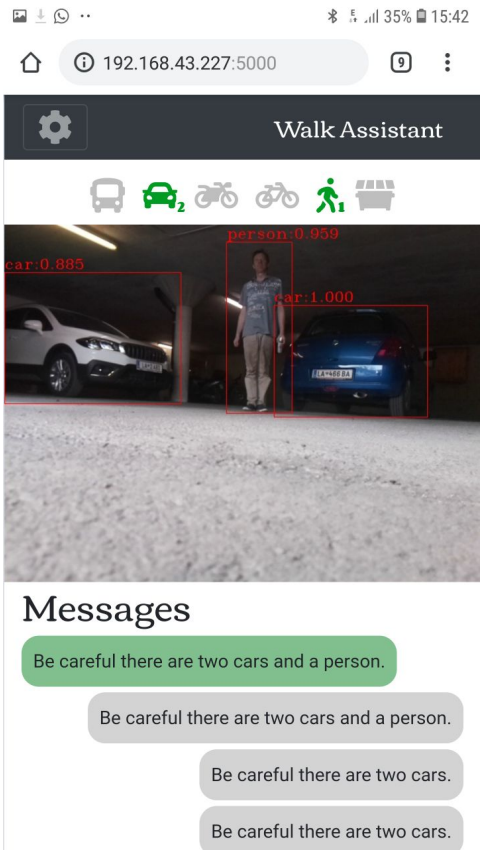

Figure 4.3: Screenshot of the Application

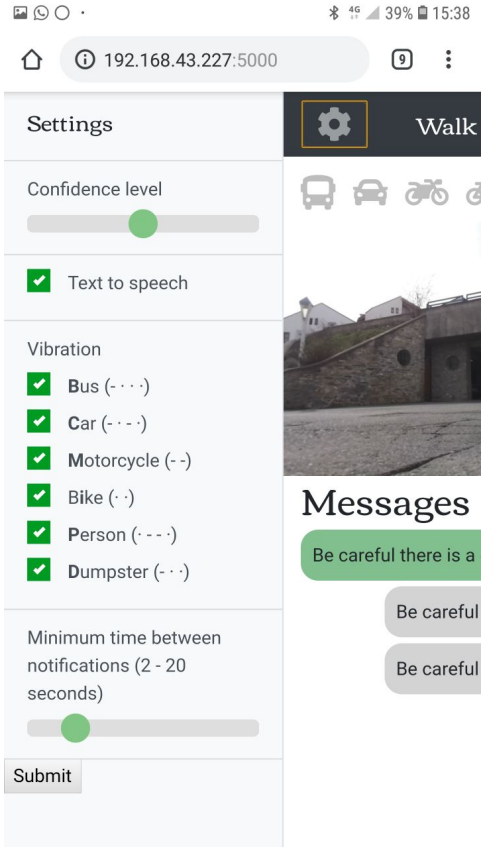

Figure 4.4: Screenshot of Application with opened Menu

In figure 4.5 the same web application is shown on a PC with bigger screen size. In that case the settings navigation is always visible to the user. Moreover, the messages are shown at the right side of the image, instead of below the image. Thanks to this setup, more messages are visible at the same time and the livestream image is bigger, using the space in an optimal way.

The whole menu is also enabled with keyboard navigation. More about this is written in the section about accessibility below.

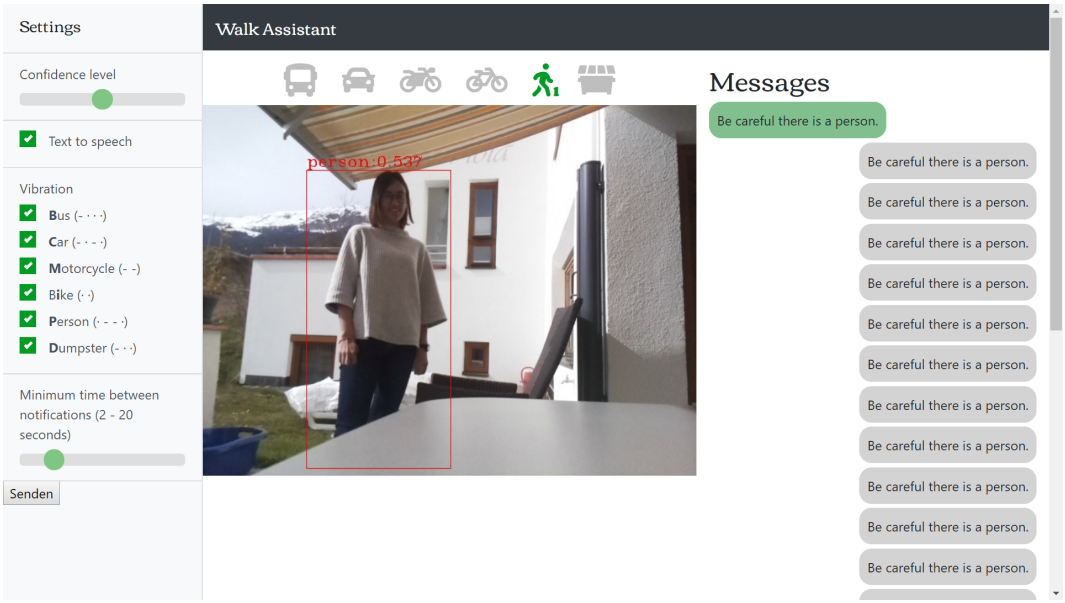

Figure 4.5: Screenshot of the Application on a PC

CSS and Bootstrap

The application is styled with CSS. The library bootstrap is used to create a responsive design, allowing the web page to adapt to every screen-size. See figure 4.2 and figure 4.5 for a comparison.

Websocket Client

The web server described in section 4.3.3 also implemented a web server socket with socket.io. [15] This websocket connection is used to push new inference data to the web application. This way it is possible that the icons are updated in real time and that the user gets immediate notifications over vibrations or text-to-speech, without any delay.

Socket.io also offers a JavaScript File to include into the web application. The web application then connects to the web server.

In listing 4.9 code is shown which receives the proposals over the websocket connection. As soon as the JavaScript file is completely loaded, the browser connects to the websocket which is running on the same domain and port as the current window is opened. When the connection is established the browser sends "I am connected" to the server. As soon as a message of type "message" is received the received content is parsed as JSON. In the field "proposals" the proposals are found.

```
$(function () {
  var socket = io.connect('http://' + document.domain +
      ':' + location.port);

  socket.on('connect', function() {
        socket.emit('my event', {data: 'I\'m connected
           !'});
  });

  socket.on('message', function(message) {
        console.log("message received");
        console.log(message);
        var obj = JSON.parse(message);
        console.log(obj['proposals']);
  });
}
```

Listing 4.9: Websocket Client

In the real code this proposals are not printed to the console log, but they are used to update the icons and generate the notifications. By doing so the configurations (confidence, vibrations enabled and text to speech output) will be taken into consideration.

Accessibility in Webdesign

As this web application should also be usable as a mobile assistant system, it is especially important that the web application is optimized for accessibility. Accessibility is key in the design of products for people with disabilities and in general helps to interact with the computer.

The user output in this thesis will be done with speech and vibrations. This will be described in detail in the sections below.

But for the users it is also important to be able to use the settings of the web application as well as understand them. Therefore, the web application and the menu especially needs to be accessible. In this thesis the web application was done according to the Web Content Accessibility Guidelines (WCAG). This guidelines were published by the Web Accessibility Initiative (WAI) of the World Wide Web Consortium (W3C).

The WCAG is organized in 4 principles with several guidelines. [18] This principles are met in this thesis. The web application can, therefore, be used by an impaired person.

Perceivable information and user interface components must be presentable to users in ways they can perceive those.

- Provide text alternatives for any non-text content so that it can be changed into other forms people need, such as large print, braille, speech, symbols or simpler language.
- Provide alternatives for time-based media.
- Create content that can be presented in different ways (for example simpler layout) without losing information or structure.
- Make it easier for users to see and hear content including separating foreground from background.

Operable user interface components and navigation must be operable.

- Make all functionality available from a keyboard.
- Provide users with enough time to read and use content.
- Do not design content in a way that is known to cause seizures.
- Provide ways to help users navigate, find content, and determine where they are.

Understandable information and the operation of user interface must be understandable.

- Make text content readable and understandable.
- Make web pages appear and operate in predictable ways.
- Help users avoid and correct mistakes.

Robust content must be robust enough that it can be interpreted reliably by a wide variety of user agents, including assistive technologies.

- Maximize compatibility with current and future user agents, including assistive technologies.

Text-to-speech

The text-to-speech output will be done with the HTML5 Web Speech API [17]. This is a HTML5 standard which is built into the browser. It is completely offline ready and,therefore, it also works, when the smartphone has no connection to the internet.

The Web Speech API is currently not supported by Safari or Internet Explorer. In this thesis the system was always tested in a Chrome browser, either on the PC or on an android smartphone. In future the Web Speech API should be supported by all major browsers.

The Web Speech API does not only support speech synthesis but also speech recognition.

In listing 4.10 the sentence from the variable sentence is synthesised into speech and played over the speakers.

```
var msg = new SpeechSynthesisUtterance(sentence);
var voices = window.speechSynthesis.getVoices();
msg.voice = voices[0];
```

```
msg.rate = 1;
msg.pitch = 1;
speechSynthesis.speak(msg);
```

Listing 4.10: Speech Synthesis with Web Speech API

The sentence which is spoken describes the image in a grammatically correct way. It includes the count and the name of the classes seen in the picture. The structure of the sentence is the following:

”Be careful there (is/are) (a/two/three/.../many) (name of class in singular or plural), (a/two/three/.../many) (name of class in singular or plural), ..., (and) (a/two/three/.../many) (name of class in singular or plural)”.

Examples of sentences spoken by the device.

- Be careful there is a person.
- Be careful there are two persons, three cars and a dumpster.
- Be careful there is a person and a bus.

As mentioned before there is a minimum time between two notifications, not to swamp the user with notifications. The time between notifications can be configured in the web application.

Vibrations

Another way to notify the person about objects of the certain class, is by vibrations. This may be more suitable than text-to-speech notifications, especially if the environment is noisy or the person’s hearing is impaired. Of course, the device in use needs to be able to produce vibrations. Vibrations usually do not work on computers or tablets, but they work on most smartphones.

As we are using a web application, the vibration API will be used. [16] The vibration API allows to set the time of a vibration and the pause between the vibrations in milliseconds. This way vibration patterns can be generated.

In this thesis every class can be activated and deactivated for vibration feedback. See the menu in figure 4.4. Morse code is used as vibration patterns to communicate one letter per category.

The morse code is an international standard. The length of a dot (•) is one unit. A hyphen (-) is three units. The space between parts of the same letter is one unit. One unit in this thesis was defined with 200 milliseconds.

The following letters and vibration patterns are used for the classes:

- **Bus** (- • • •)
- **Car** (- • - •)
- **Motorcycle** (- -)
- **Bike** (• •)
- **Person** (• - - •)
- **Dumpster** (- • •)

In listing 4.11 an example for the pattern of the category 'bus' can be seen. The first number in the array is the duration of the vibration, the next number is the break, the next one a vibration again. The numbers are in ms.

```
navigator.vibrate([600,200,200,200,200,200,200]);
```

Listing 4.11: Vibration Feedback with Vibration API

The vibration API works on Firefox, Chrome, Android Browser and Chrome for Android. Again it does not work on Safari. Also, this function will not work on iPhones.

4.3.5 Autostart

The device created in this thesis should be very easy to use. To start the application only the power needs to be plugged. A start script will then start all the modules needed. Firstly, the camera publisher will be started, secondly, the inference publisher, and lastly, the web server.

X-Windows-System is the window manager running on linux. The X-Server handles all the communication accessing the computer screen. The application will only start when the X-Server is already running. The X-Server also has a startup script. This script is located in:

`/etc/xdg/lxsession/LXDE-pi/autostart`

The following line is added to the script in order to start the services:

```
@sh /home/pi/start.sh
```

Around 1 minute later, the Raspberry should be fully started and users can connect with their smartphones or PCs to the web server.

5 Evaluation

In this chapter the results of the object detection and the classification networks are evaluated for quality and inference time. Therefore, different networks are compared on different hardware (CPU and VPU). Moreover, the influence of the camera position on the detection results are evaluated.

5.1 Object Detection Model

The network, which was trained for the object detection task in this thesis, is described in detail in section 2.2. It is based on a the SSD Mobilenet V1 network. The network was pre-trained with the Coco dataset and afterwards trained with specific images, which were made with a camera positioned at ground level. The network was trained to detect the following categories:

- person
- bicycle
- car
- motorcycle
- bus
- dumpster

The Common Objects in Context (Coco) dataset [1] is a large-scale object detection, segmentation, and captioning dataset. It has over 80 object categories and over 200.000 labeled images. All the classes used in this thesis, except the class dumpster, are already included in the Coco dataset.

In the evaluation also different networks will be used. The networks are described here to know which exact networks were used and compared with each other.

Assistant System
This network is the network which was trained by Dipl.-Ing. Christian Ertler for this specific task. He used the SSD MobileNet V1 which was already pre-trained on the Coco dataset as a base network and used a custom dataset to further train the network.

SSD MobileNet V1 Coco
The network is the same as the network used for the assistant system. Only the weights are different, as it was not trained with the images from the assistant system dataset. It also knows 80 classes, instead of the 6 classes used for the assistant system. In contrast to the assistant system network, the class dumpster is not known by this network. The used version is the version from 17th November 2017. The frozen model including the weights can be downloaded here: http://download.tensorflow.org/models/object_detection/ssd_mobilenet_v1_coco_2017_11_17.tar.gz

5.1.1 Time for Inference on VPU vs CPU

As this application gives notifications about objects for assistant systems or robotics, one very important characteristic of this thesis was to create a system which is capable of running in real time. Therefore, the inference time for the different networks was tested on the CPU and on the Vision Processing Unit (VPU). The results are summarized in table 5.1.

Network	Run time in ms		FPS	
	CPU	VPU	CPU	VPU
Assistant System	104 ms	216 ms	9,61	4,63
SSD MobileNet V1 Coco	340 ms	99 ms	2,94	10,10

Table 5.1: Comparison of inference time

In the table 5.1 it can be seen that the standard assistant system network runs with about 4,6 fps on the VPU. This means that the application can run at a reasonable frame rate even on small embedded systems like the

Raspberry PI thanks to the help of the Intel Movidius Neural Compute stick.

For applications such as self driving cars or similar ones, 4,6 fps are not enough. However, for people walking on the street this seems a reasonable frame-rate.

The CPU used for the tests in this thesis is an Intel Core i5-3317U running at 1.70 GHz. Interesting to see is that the assistant system network is running faster on the CPU than on the VPU by about the factor 2. Of course, when using a weaker CPU identical to the one from the Raspberry 3+, the ARM Cortex-A53 running at 1.4 GHz, the inference would probably take longer. Exact numbers cannot be determined, as the OpenVINO framework currently only supports Intel CPUs.

Another interesting fact is that the SSD MobileNet V1 Coco network runs faster than the assistant system model on the VPU, although the assistant system model supports less classes and the rest of the architecture is the same. The inference is twice as fast and takes only 99 ms, compared to 216 ms.

On the CPU the behaviour is exactly the opposite. The inference of the SSD MobileNet V1 Coco network takes 340 ms compared to 104 ms for the assistant system network. A reason for that could be that the network is supported by default by the Intel Movidius Neural Compute stick. The compiler is probably optimized for this kind of networks, and that can be the reason for that such a fast speedup can be achieved with the VPU.

The results of the detections are very similar on the CPU and the VPU. Because the VPU and the CPU do not use the same floating point precision they are not 100% the same. However, these little deviations do not make a difference, especially for this use case.

In figure 5.1 and in figure 5.2 a car and a dumpster are detected correctly. A container to store gravel is detected as a dumpster. Whether the inference is done on the CPU or the VPU does not matter. The results look the same in both images (see figure 5.1 and 5.2).

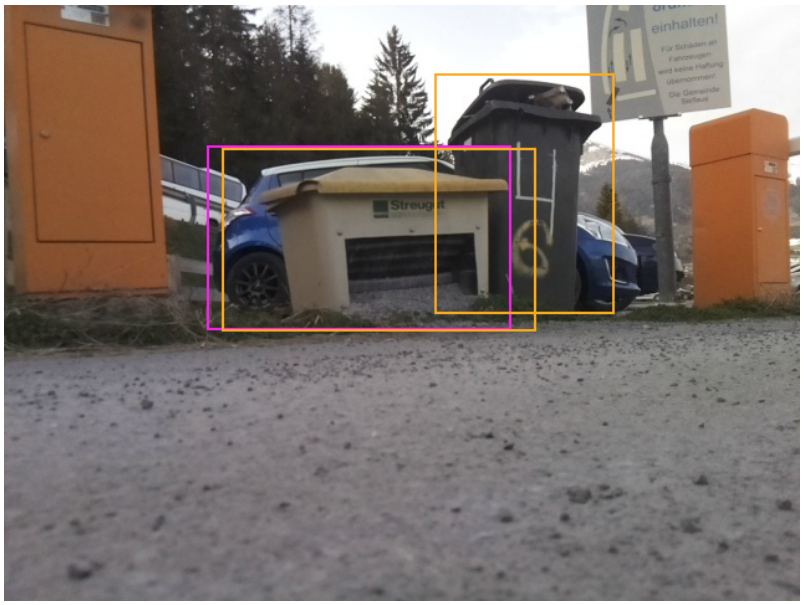

Figure 5.1: Inference result on the CPU

Figure 5.2: Inference result on the VPU

5.1.2 Custom Trained Assistant System Network

Dipl.-Ing. Christian Ertler trained a custom network with images which were recorded specially for the assistant system. In this section the custom trained network will be compared to the SSD MobileNet V1 Coco network, a network which was only trained on the Coco dataset. The network architecture of the networks are the same and, therefore, should not make too much difference. The weights and biases, of course, are different.

Other network architectures are also available trained on the Coco network. It only makes sense to compare results between different network architectures if the exact same data set is used. The class dumpster is not available in the MobileNet V1 Coco network and, therefore, it cannot be compared.

In figure 5.3 the inference was done with the SSD Mobilenet V1 network trained on Coco. In figure 5.4 the inference was done with the network which is trained specific for the assistant system. Both results look the same. One could conclude that the custom training did not improve the result, as the same result could be achieved with a standard pre-trained SSD MobileNet network.

Figure 5.3: Car and Person Inference with SSD Mobilenet V1 Coco network

Figure 5.4: Car and Person Inference with assistant system network

In figure 5.5 and 5.6 a very similar image is shown. However, this image was captured with a camera at ground level and not from the perspective of a walking person. In that case the networks do not perform the same.

The network trained specially for the assistant system (figure 5.6) does not only recognize the car, but also the person standing next to the car. The standard SSD Mobilenet V1 Coco network (figure 5.5) also recognizes the car, but does not recognize the person standing next to the car.

The training images for the assistant system network were all captured from the perspective of the ground. So from this perspective the custom trained network seems to perform better.

Figure 5.5: Car and Person inference with SSD Mobilenet V1 Coco network

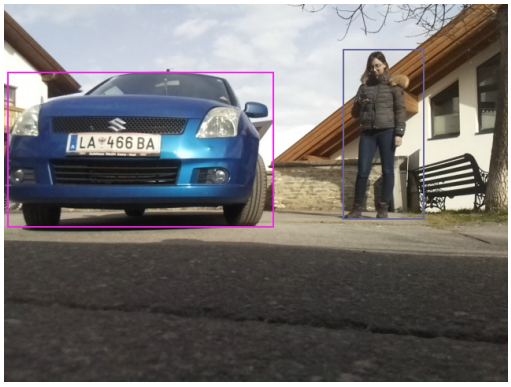

Figure 5.6: Car and Person inference with assistant system network

In figure 5.7 and 5.8 an image of a dumpster can be seen. The inference was again done with both networks. As the dumpster is not a known class for the standard SSD MobileNet V1 Coco network, the dumpster is not detected with this network. On the other hand, the assistant system network does not have any problems detecting this dumpster.

Nevertheless, there are also examples where the standard SSD MobileNet V1 Coco network performs better than the custom assistant system network. In figure 5.9 and 5.10 a motorcycle is shown and the inference is done for both networks. The motorcycle gets detected correctly by the standard SSD MobileNet V1 Coco network, but does not get detected by the custom assistant system network. In figure 5.9 also two other small objects are detected by the SSD MobileNet V1 Coco network. However, these object categories are not in the defined list of classes which are interesting for the assistant system. So they can be ignored.

Figure 5.7: Dumpster inference with SSD Mobilenet V1 Coco network

Figure 5.8: Dumpster inference with assistant system network

Figure 5.9: Motorcycle inference with SSD Mobilenet V1 Coco network

Figure 5.10: Motorcycle inference with assistant system network

5.1.3 Evaluation of the Perspective of the Camera

The camera position certainly has a great influence on the detection results of the system. It is obvious that for some cases i makes sense to position the camera at the level of the ground. For example, it is hard to detect pavement edges when the camera is not positioned at ground level and the perspective is, hence, a low one.

However, it also has several disadvantages. One of the disadvantage is certainly that the perspective is a bit unusual and, therefore, more training data would be needed to achieve better results. Another disadvantage is that the camera moves a lot when it is fixed at a low position, like a leg. The camera would need to be stabilized, which makes the device much more complicated. When the image moves a lot the images get blurry. Also the constant change of perspective makes it hard or even impossible to exactly detect where a object is located. Moreover, objects further away are detected when the leg is facing upwards. These objects that are far away, however, are not too interesting for most use cases.

In this example, a street with some parking cars as obstacles are recorded from the view of the ground and from the view of a walking person. See figure 5.11 and 5.12 for a comparison of the two scenes.

Figure 5.11: Street with cars. Picture taken from view of a walking person

Figure 5.12: Street with cars. Picture taken from ground level.

Both scenes (from up and down) are evaluated with the assistant system and the SSD Mobilenet V1 Coco network.

It is very interesting to see that when the picture is taken from above (position of a walking person) the SSD Mobilenet V1 Coco network performs better than the assistant system network. In figure 5.13 no car is detected with the assistant system network, whereas in figure 5.15 the car right in front of the person is detected correctly by the SSD Mobilenet V1 Coco network. The cars further away in the back are not detected by any network.

When making an inference of the same scene but with a picture taken from below the networks behave opposed. In this case the assistant system network performs better than the SSD Mobilenet V1 coco network. With the assistant system network the car right in front of the person and also one car in the back is detected (see figure 5.14). With the SSD Mobilenet V1 Coco network, no car is detected (see figure 5.16).

Special trained networks for specific viewpoints seem to make the performance better for this specific viewpoint, but do not detect objects from the other viewpoint that well anymore. So if the camera is mounted on ground level, more training data for the ground level certainly makes the overall results better.

In the next example 2 people are detected again with the assistant system network and the SSD Mobilenet V1 coco network. This time there are 3 different viewpoints used. The viewpoints are from below, from the center and from above. The images are shown from figure 5.17 to 5.22.

The SSD MobileNet V1 network can detect all persons, independently from the viewpoint. The assistant system network did not detect any person in the image from the middle. In the image taken from below the assistant system network only detected one person, although this is the position that the assistant system network is trained on.

Despite the fact that the weights of the SSD MobileNet V1 network were used to fine-tune the weights with the new training data for the assistant system, the overall results in this specific example got worse.

Figure 5.13: Car from above. Inference with assistant system network.

Figure 5.15: Car from above. Inference with SSD Mobilenet V1 Coco.

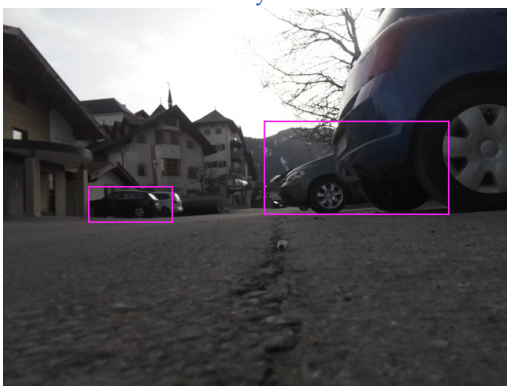

Figure 5.14: Car from below. Inference with assistant system network.

Figure 5.16: Car from below. Inference with SSD Mobilenet V1 Coco.

Figure 5.17: Person from above. Inference with assistant system network.

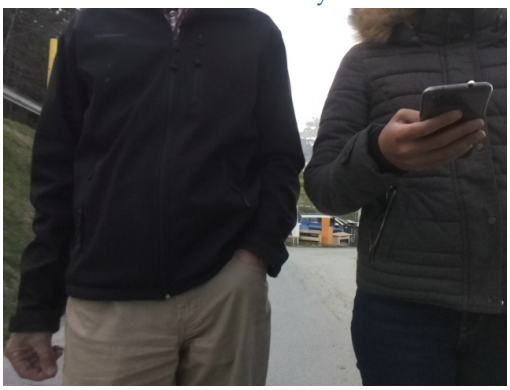

Figure 5.18: Person from the middle. Inference with assistant system network.

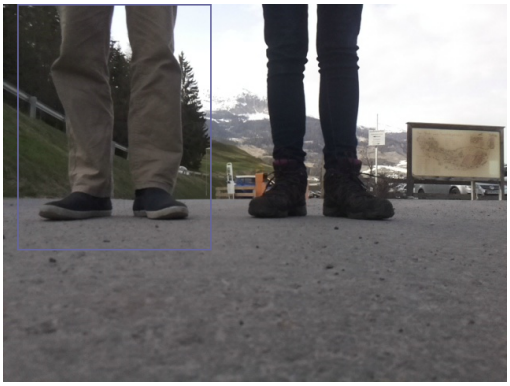

Figure 5.19: Person from below. Inference with assistant system network.

Figure 5.20: Person from above. Inference with SSD Mobilenet V1.

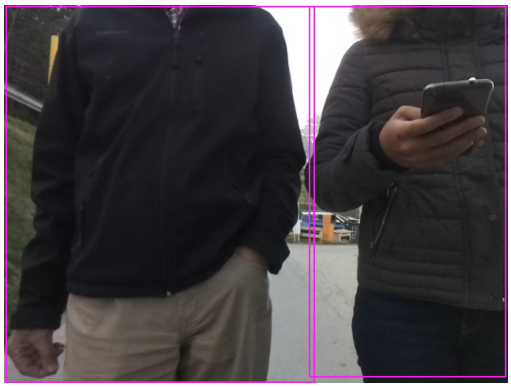

Figure 5.21: Person from the middle. Inference with SSD Mobilenet V1.

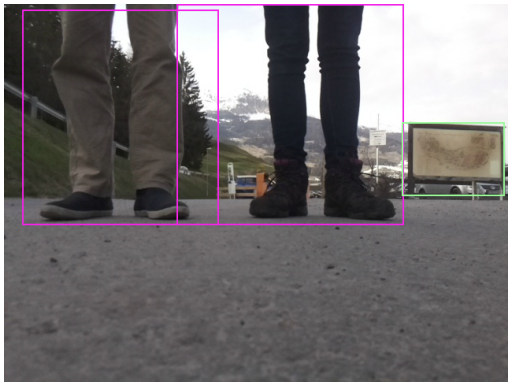

Figure 5.22: Person from below. Inference with SSD Mobilenet V1.

5.2 Classification Model

In this section the classification model is evaluated. The model architecture is a MobileNet V1 Convolutional Neural Network. It was trained to detect 3 different classes: wall, pavement edges and stairs. In this example multi-label classification is used. That means that for each class a value is given between 0 and 1. One image can therefore be classified as more than one class.

5.2.1 Time for Inference on VPU vs CPU

The Intel Movidius NCS enables fast inference on the Raspberry PI and therefore brings Convolutional Neural Networks running in real time to embedded systems. The time needed for inference of a single image was tested on the CPU and on the VPU.

The CPU used was again an Intel Core i5-3317U processor running at 1.70 GHz. On this CPU a single inference takes 334 ms.

On the Intel Movidius NCS the inference for the same network and the same image only takes 39 ms. So the speedup in this example is more than factor 8. At the same time the Intel Movidius NCS is much more energy efficient.

5.2.2 Evaluation of Detection Results of the Classification Network

In figure 5.23 a pavement edge is seen. The pavement edge is taken with a camera positioned at ground level, but the pavement edge is still a bit further away. The network detects 84.9% a wall and 15.1% a pavement edge. Behind the pavement edge even further away there is a wall. It seems when the pavement edge is to small in the image, the wall gets detected stronger than the pavement edge.

In comparison in figure 5.24 99.5% pavement edge is detected, whereas only 0.5% wall is detected. In this image a wall can be seen very clearly. However in this image the pavement edge is closer and more prominent.

Figure 5.23: Pavement edge further away

84.9% wall
15.1% pavement edge
0.0% stairs

Figure 5.24: Wall behind pavement edge

99.5% pavement edge
0.5% wall
0.0% stairs

In the figure 5.25 and 5.26 the same scene is shown. Once the image is taken from the perspective of the ground and once the image is taken from the perspective of a walking person.

Interesting to see is that when the image is taken from the position of the walking person (see figure 5.25) the network gives a 84.9% probability that the recorded image is a wall and only a 15.1% probability that the image is a pavement edge.

In the image which is taken from the perspective of the ground (see figure 5.26) the network behaves opposite. The probability of being a pavement edge is 90.2%, whereas the probability for being a wall is only 9.7%.

It seems that pavement edges can only be detected when they are close and seen from the perspective of the ground. This also makes sense, as all the training data was taken from the perspective of the ground.

Figure 5.25: Pavement/Wall Edge from above

84.9% wall
15.1% pavement edge
0.0% stairs

Figure 5.26: Pavement/Wall Edge from below

90.2% pavement edge
9.7% wall
0.0% stairs

6 Conclusion and Future work

In this thesis a small embedded device was developed, that notifies persons or robots from potentially dangerous objects in their surroundings. This was done with the help of Convolutional Neural Networks (CNNs). In particular two different neural networks were used.

The first network is an object detection network which was trained on the classes car, bus, bike, person, motorcycle and dumpster. The base for this object detection network is a SSD MobileNet architecture which was trained with special training data for the assistant system. This network is used to detect objects which are further away and only fill part of the image.

The second network is a classification network which detects stairs, walls and pavement edges. When close to these objects they usually fill the whole image. In that case image classification is used instead of object detection. This network is also based on the MobileNet architecture.

Both networks already existed and were created in a previous project at Graz University of Technology. In this thesis these networks were translated so that they can run on a Vision Processing Unit (VPU). The Intel Movidius Neural Stick together with the Raspberry PI was used to create an embedded system that notifies persons or robots from obstacles. A camera was connected directly to the battery powered Raspberry PI, to deliver the image. A web server was developed to which the user can connect with a smartphone or a PC. The inference results of the network is sent in real time over a websocket connection to the browser of the user. When an object appears a notification is given via text-to-speech commands or vibrations. Different settings in the web application allow the user to customize the experience.

The system works pretty reliable, although in crowded places there are far too many notifications as objects are detected all the time. This prototype

is a good base which proves that it is possible to create a computer vision powered embedded device as an assistant system or for robotics. Using better networks and more training data the results are expected to get even better.

The system also has clear limitations which can be improved in the future.

The device does not detect many classes which can be very dangerous for a person or robot. Among these objects are street sign or street lights. These classes and more should be also added to the network.

It is helpful to know how many objects there are in an image. But for a person or robot it would be an important information if these objects are close to the person or far away. Currently the distance of the object cannot be measured as the orientation between the camera and the obstacle is unknown. The orientation would need to be very precise and therefore this approach was not used.

This problem could also be solved using a stereo vision camera or another depth sensor. In that case a complete 3D model of the surrounding can be generated. The next step would be to filter which objects are in the walking direction and close to the person. Only if there is really an object in the way of the person or robot a notification should be generated. This would be much more helpful as less notifications but of better quality are generated. Another useful enhancement could be to improve further the web application with a feature that allows remote help for the person or the robot. In difficult situations a remote person could be called over the internet. The remote person could see the surroundings over the camera and guide the robot or person over voice.

While there is still much room for improvements the principal concept looks very promising. This thesis showed that computer vision application using CNNs can be embedded in small and portable devices. This is very interesting for many different use cases.

Appendix

Bibliography

[1] CocoDataset. *Coco - Common Objects in Context*. 2019. URL: http://cocodataset.org/#home (visited on 04/01/2019) (cit. on p. 39).

[2] Miguel Grinberg. *Video Streaming with Flask*. 2019. URL: https://blog.miguelgrinberg.com/post/video-streaming-with-flask (visited on 04/01/2019) (cit. on p. 27).

[3] Matthijs Hollemans. *Google's MobileNets on the iPhone*. Dec. 2011. URL: https://machinethink.net/blog/googles-mobile-net-architecture-on-iphone/ (visited on 12/10/2011) (cit. on p. 4).

[4] Andrew Howard et al. *MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications*. 2016. URL: https://arxiv.org/abs/1704.04861 (cit. on pp. 3, 4).

[5] INTEL. *Intel Open Source*. 2019. URL: https://01.org/openvinotoolkit (visited on 04/01/2019) (cit. on p. 9).

[6] Intel. *Intel Movidius Webpage*. 2019. URL: https://www.movidius.com/ (visited on 04/01/2019) (cit. on pp. 7, 8).

[7] Wei Liu et al. *SSD: Single Shot MultiBox Detector*. 2016. URL: https://arxiv.org/abs/1512.02325 (cit. on pp. 4, 5).

[8] ModelZoo. *Tensorflow detection model zoo*. 2019. URL: https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md (visited on 04/01/2019) (cit. on p. 16).

[9] Movidius. *Movidius NCSDK*. 2019. URL: https://movidius.github.io/ncsdk/index.html (visited on 04/01/2019) (cit. on p. 8).

[10] Openvino. *Open Vino Toolkit Docs*. 2019. URL: http://docs.openvinotoolkit.org (visited on 04/01/2019) (cit. on pp. 9, 10).

[11] Openvino-SSD-Sample. *Object Detection SSD C++ Demo, Async API Performance Showcase*. 2019. URL: https://docs.openvinotoolkit.org/latest/_inference_engine_samples_object_detection_demo_ssd_async_README.html (visited on 04/01/2019) (cit. on p. 23).

[12] Openvinotoolkit. *Converting TensorFlow Object Detection API Models*. 2019. URL: https://docs.openvinotoolkit.org/latest/_docs_MO_DG_prepare_model_convert_model_tf_specific_Convert_Object_Detection_API_Models.html (visited on 04/01/2019) (cit. on p. 16).

[13] Picamera. *Picamera Online Documentation*. 2019. URL: https://picamera.readthedocs.io/en/release-1.13/ (visited on 04/01/2019) (cit. on p. 20).

[14] RaspiCam. *RaspiCam: C++ API for using Raspberry camera with/without OpenCv*. 2019. URL: http://www.uco.es/investiga/grupos/ava/node/40 (visited on 04/01/2019) (cit. on p. 21).

[15] Socket.io. *Socket.io*. 2019. URL: https://socket.io/ (visited on 04/01/2019) (cit. on p. 33).

[16] W3C. *Vibration API (Second Edition)*. 2019. URL: https://www.w3.org/TR/vibration/ (visited on 04/01/2019) (cit. on p. 36).

[17] W3C. *Web Speech API Specification*. 2019. URL: https://w3c.github.io/speech-api/ (visited on 04/01/2019) (cit. on p. 35).

[18] WCAG. *WCAG Quickreference*. 2019. URL: https://www.w3.org/WAI/WCAG21/quickref/ (visited on 04/01/2019) (cit. on p. 34).

[19] Wikichip. *SHAVE v2.0 Microarchitecture Movidius*. 2019. URL: https://en.wikichip.org/wiki/movidius/microarchitectures/shave_v2.0 (visited on 04/01/2019) (cit. on p. 7).

[20] ZMQ. *ZeroMQ Distirbuted Messaging*. 2019. URL: http://zeromq.org/ (visited on 04/01/2019) (cit. on p. 17).