Benjamin Ramberg Møklegård

People Detection using Transfer learning on Deep Convolutional Neural Networks

Master's thesis in Electronic Systems Design
Supervisor: Snorre Aunet
June 2020

Master's thesis

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

Benjamin Ramberg Møklegård

People Detection using Transfer learning on Deep Convolutional Neural Networks

Master's thesis in Electronic Systems Design
Supervisor: Snorre Aunet
June 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Acknowledgement

I want to extend my thanks to my project supervisor Professor Snorre Aunet for his assistance and guidance during my work on this thesis. Your feedback and aid have been greatly appreciated. I am also grateful to the team at Disruptive Technologies AS and would like to extend thanks to both Øystein Moldsvor for giving me an exciting assignment and Sigve Tjora for giving me feedback on my thesis your help has been much appreciated. Finally, I must thank my family and especially my parents, for supporting me during my studies. I could not have done this without your continuous encouragement and support.

Benjamin R. Møklegård

Abstract

Convolutional neural networks have been established as one of the most efficient ways of applying machine learning to computer vision. The purpose of this thesis has been to investigate the concept of transfer learning and how it can be utilized to retrain pretrained neural network models to increase detection accuracy. The fine-tuned networks in this thesis have been trained for the task of "People Detection." That is detecting and giving an estimate of how many people are present in an image. A subset of images from Open Image Database, a database curated by Google, has been used to train the custom detectors. In this thesis, a set of 20000 images is used for the training phase and 4000 images for the test phase. The images belong to the object class, "Person." The neural networks explored in this thesis are Mobilenet V2 + SSD (Non-quantized and Quantized), YOLOv3 and YOLOv3-Tiny.

Applying *transfer learning* increases the mean Average Precision (mAP) and average Recall (AR) scores for most of the models. mAP for Mobilenet V2 + SSD increases from 0.49 to 0.62. Mobilenet V2 + SSD Quantized increases from 0.004 to 0.61. YOLOv3 suffers a slight performance reduction, where the mAP reduces from 0.66 to 0.65. YOLOv3-Tiny sees an increase from 0.25 to 0.51.

The models have undergone further testing by being deployed on the Google Coral Dev Board, which features an accelerator. Every model has been tested on the Dev Board CPU, while the Quantized version of the Mobilenet V2 + SSD model has also been tested on the TPU accelerator. Results from the testing shows that the Mobilenet V2 + SSD (Non-quantized) model runs at a frame per second (FPS) of 1.35. The quantized model performs better at 3.61 FPS on CPU and 131.82 FPS on the TPU. YOLOv3 and YOLOv3-Tiny performs poorly with an FPS of 0.02 and 0.23, respectively. Estimations on the energy consumption per operation have been performed, to give a better overview on the energy efficiency of each model. Since the models are to be deployed in system for detection of people and that it is likely that it will run on battery power, the energy consumed per network becomes vital to determine which should be deployed to ensure longevity of such a system. In this thesis it was found that the Quantized Mobilenet V2 + SSD consumes approximately 8pJ/FLOPS when running on the TPU,increasing to 266pJ/FLOPS when running on CPU. The non-quantized Mobilenet model consumes 474pJ/FLOPS (CPU).

YOLOv3 consumes 1210pJ/FLOPS(CPU), and YOLOv3-Tiny uses 1397pJ/FLOPS (CPU). In terms of detection, the model has been applied to an example image containing four people. Both Mobilenet models and YOLOv3 manages to properly detect 4 people, while YOLOv3-Tiny only manages to detect three people. The conclusion reached in this thesis is that transfer-learning can help boost a pre-trained model's performance and fine-tune the models for custom tasks, such as "People detection." The recommendation one can provide from the results of deploying the neural networks is to use the Quantized Mobilenet V2 + SSD model. This model is shown to be the most energy-efficient model when deployed on the Edge TPU, which is vital in deploying such a system on resource-constrained devices such as the Google Coral Dev Board.

Sammendrag

Convolutional neural networks har blitt etablert som den mest effektive metoden for å anvende deep learning på datasyn. Denne oppgaven har som mål å utforske hvordan man kan bruke konseptet transfere learning for å øke deteksjons effektiviteten til allerede trente nevrale nett og deretter analysere disse trente modellen ved å anvende de på ressurs begrensede maskinvare. Oppgaven disse trente nettverkene er anvendt på er person deteksjon, det vil si å detekter og gi et estimat på hvor mange personer som befinner seg i et bilde. Nettverkene som er anvendt i oppgaven er trent på en del av et større datasett laget av Google, kalt Open Image Database. I oppgaven ble 20000 bilder tilhørende klassen "Person" brukt for å fine-tune de allerede trente modellene. Modellene som er utforsket i denne oppgaven er Mobilenet V2 + SSD (Non-quantized og Quantized) for Tensorflow, YOLOv3 og YOLOv3-Tiny for Darknet.

Ved å bruke transfer learning så kan man observere en forbedring i modellenes mean Average Precision (mAP) og Average Recall (AR). For Mobilenet V2 + SSD så øker mAP fra 0.49 til 0.62. Mobilenet V2 + SSD Quantized ser en økning fra 0.004 til 0.61. YOLOv3 ser en liten reduksjon fra 0.66 til 0.65 og YOLOv3-Tiny øker fra 0.25 til 0.51.

Videre så er modellene blitt testet på Google Coral Dev Board som har en innebygget akselerator for nevrale nett. Alle modellene har blitt testet på utviklingsbrettets CPU, hvorav Mobilenet V2 + SSD (Quantized) også har blitt testet på TPUen. Resultatene fra testingen ga at Mobilenet V2 + SSD kjører med en bilde per sekund (FPS) på 1.35. Den kvantifiserte modellen kjører raskere med 3.61 FPS på CPU og 131.82 FPS på TPUen. YOLOv3 og YOLOv3-Tiny har en FPS på 0.02 og 0.23 respektivt. Videre ble det funnet at energy per operation for Quantized Mobilenet V2 + SSD kunne estimeres til 8pJ/FLOPS når den kjørte på TPUen og 266pJ/FLOPS ved inferens på CPU. I motsetning så bruker den ordinere Mobilenet modellen 474pJ/FLOPS (CPU). YOLOv3s energi per operasjon ble estimert til 1210pJ/FLOPS (CPU) og YOLOv3-Tiny bruker ca 1397pJ/FLOPS (CPU). I forhold til deteksjon så klarer alle modellene bortsett fra YOLOv3-Tiny å detektere mennesker i et eksempel bilde som inneholder fire personer. Hvor YOLOv3-Tiny kun klarer å detektere tre personer. Konklusjonen i oppgaven blir at transfer learning kan hjelpe ved å gi modellene en økning i deteksjons nøyaktighet på egendefinerte datasett. videre så er anbefalingen at den kvantifiserte modellen anvendes. Denne kan kjøres på Corals TPU, hvilket er bevist å være den mest energi effektive effektive måten å kjøre modellene, noe som vil være viktig hvis modellen skal kjøres på hardware med ressurs begrensinger, slik som Google Coral Dev Board.

Contents

Acknowledgement . . . . . **iii**
Abstract . . . . . **v**
Sammendrag . . . . . **vii**
Contents . . . . . **ix**
Figures . . . . . **xiii**
Tables . . . . . **xv**
Acronyms . . . . . **xvii**
1 Introduction **1**
1.1 Problem description 1
1.2 Background and Motivation 2
1.3 Relevant Work 3
1.3.1 People Detection using CNNs 3
1.4 Thesis Contribution 4
1.5 Thesis structure 4
2 Theory **7**
2.1 A brief history of machine learning 7
2.2 What are Artificial Neural Networks 8
2.2.1 Biological Neuron 9
2.2.2 Artificial Neuron 9
2.2.3 Fully-connected Artificial Neural Network 10
2.3 Training a Neural Network 11
2.3.1 Loss Function 11
2.3.2 Backpropagation and Gradient Descent 13
2.4 Convolutional Neural Networks 14
2.4.1 Convolution Operation 14
2.4.2 Convolution Layers 16
2.4.3 Activation Layers 17
2.4.4 Pooling Layers 18
2.5 Transfer learning 19
2.6 Object Detection 20
2.6.1 R-CNN 20
2.6.2 Fast R-CNN 20
2.6.3 Faster R-CNN 21
2.6.4 SSD 21

2.6.5 YOLO . . . 22
3 Experimental Setup . . . 25
3.1 Models . . . 25
3.2 Datasets . . . 25
3.2.1 MSCOCO . . . 25
3.2.2 Open Image Dataset . . . 25
3.3 Software Implementation . . . 26
3.3.1 Tensorflow-based Models . . . 26
3.3.2 YOLO . . . 26
3.3.3 COCO API . . . 27
3.3.4 Evaluation of Darknet based Models . . . 27
3.3.5 Network Configuration . . . 27
3.4 Hardware Setup . . . 27
3.4.1 Google Coral Dev Board . . . 28
4 Methodology . . . 29
4.1 Detection Metrics . . . 29
4.1.1 IoU . . . 29
4.1.2 Detection outcomes and the confusion matrix . . . 29
4.1.3 Precision . . . 31
4.1.4 Recall . . . 31
4.1.5 Mean Average Precision . . . 31
4.2 Inference Time . . . 31
4.3 Number of FLOPS . . . 32
4.4 Frames per second . . . 32
4.5 Power Consumption . . . 32
4.6 Energy per Operation . . . 32
5 Results . . . 35
5.1 Transfer learning on pretrained models . . . 35
5.1.1 Mobilenet V2 + SSD . . . 35
5.1.2 Quantized Mobilenet V2 + SSD . . . 37
5.1.3 YOLOv3 . . . 39
5.1.4 YOLOv3 Tiny . . . 41
5.1.5 Combined performance figure . . . 42
5.2 Energy per operation during Inference Run on Google Coral . . . 44
5.3 Inference Time on different devices . . . 46
5.4 Applying Detector for Estimating People . . . 46
6 Discussion . . . 51
6.1 Applying Transfer-learning to boost model performance . . . 51
6.2 Evaluating the neural network performance in terms of Energy per Operation . . . 52
6.3 Inference Time on different devices . . . 53
6.4 Detecting People using the trained models . . . 54
7 Conclusion . . . 57
Bibliography . . . 59

A Additional Material **63**
B Detection on multiple Images **65**
C Pascal VOC Format **69**
D Copy of Specialization Report **71**

Figures

2.1 An example of a biological neuron. The image has no given name, by Unknown artist, Licensed as "Free to Use" 9

2.2 An example of an artificial neuron with three inputs 10

2.3 Artificial Neural Network with an input layer using two input neurons, a hidden layer with four neurons and a three neuron output layer 11

2.4 An example of a simple neural network containing two connected neurons 14

2.5 Example of Convolution Operation on a 4x4 input matrix with a 3x3 kernel. Reproduced from [29] 16

2.6 An illustration of a Convolutional Neural Network with two convolutional layers. Not shown on the image is the pooling layer in between the convolutional layers 17

2.7 Pooling is performed in one of two types either max-pooling or average-pooling. Figure reproduced from [29] 19

3.1 The Google Coral Dev Board 28

4.1 IoU can be visualized by looking at the intersect of two boxes over the union of the same boxes 30

4.2 An example of a confusion matrix 30

4.3 The UM32C USB Meter 33

5.1 Classification, Localization and Total Loss during Training of non-quantized model in blue and quantized model in orange. 37

5.2 Average and Total loss during training of Yolov3 39

5.3 Average and Total loss during training of YOLOv3-Tiny 41

5.4 Summarized difference in mean Average Precision before and after training per model. Where MB is Mobilenet, (Q) means quantized and (T) denotes YOLOv3-Tiny 43

5.5 Energy per operation given in J/FLOPS for each neural network model running on the Google Coral Dev Board 45

5.6 Detection result using the Non-Quantized Mobilenet model. Top: using the frozen graph exported from Object Detection API. Bottom: using the exported TFLite formatted model. Both tested with a threshold set to 0.4, Image is "Group of People Sitting Inside Room" by JopWell, Licensed as "Free to Use" 47
5.7 Detection result using the Quantized Mobilenet model with at threshold set to 0.4, Image is "Group of People Sitting Inside Room" by JopWell, Licensed as "Free to Use" 48
5.8 Detection result using YOLOv3 with at threshold set to 0.4, Image is "Group of People Sitting Inside Room" by JopWell, Licensed as "Free to Use" 49
5.9 Detection result using YOLOv3-Tiny with at threshold set to 0.4, Image is "Group of People Sitting Inside Room" by JopWell, Licensed as "Free to Use" 49

B.1 Detection generated by Mobilenet before and after TFLite conversion. Notice that after converting the model to TFLite Format, no detection is generated. Image is "People Having Meeting Inside Conference Room", by Christina Morillo, Licensed as Free to Use . . 65
B.2 Detection generated by the Quantized Mobilenet model. Image is "People Having Meeting Inside Conference Room", by Christina Morillo, Licensed as Free to Use 66
B.3 Detection generated by YOLOv3 and YOLOv3-Tiny. YOLOv3 manages to detect two people in the image, while YOLOv3-Tiny only manages to detect one. Image is "People Having Meeting Inside Conference Room", by Christina Morillo, Licensed as Free to Use . . 67

Tables

3.1 Configuration options for each network. 28

5.1 Metrics generated by evaluation on the test set with the pretrained Mobilenet model and the custom fine-tuned Mobilenet model 36
5.2 Metrics generated by evaluation on the test set with the quantized pretrained Mobilenet model and the quantized fine-tune model . . 38
5.3 Metrics generated by evaluation on the test set with the pretrained YOLOv3 model and the fine-tune model 40
5.4 Metrics generated by evaluation on the test set with the pretrained YOLOv3-Tiny model and the fine-tune model 42
5.5 Minimum Power Consumption during Inference Run on Each Model. Both the Total recorded power and the difference between the power during inference and idle is shown 44
5.6 Minimum Energy Per Operation during Inference Run on the Google Coral using each neural network model. Here MB denotes mobilenet and (Q) denotes Quantized. Y and Y-T is for YOLOv3 and YOLOv3-Tiny respectively 44
5.7 Average Inference Time during run on a subset of the test dataset on Desktop CPU, Desktop GPU, Edge CPU and Edge TPU for different frameworks. The Quantized Model is denoted with (Q) 46

Acronyms

ANN Artificial Neural Network.

AP Average Precision.

CNN Convolutional Neural Network.

DNN Deep Neural Network.

IoU Intersect over Union.

LSTM Long-short Term Memory.

mAP mean Average Precision.

OID Open Image Database.

RNN Recurrent Neural Networks.

SoM System-On-Module.

TPU Tensor Processing Unit.

VLIW Very Long Instruction Word.

VPU Vision Processing Unit.

Chapter 1

Introduction

1.1 Problem description

Disruptive Technologies Research AS [1] creates tiny wireless sensors that can be deployed in a wide variety of scenarios. One of these scenarios is "presence detection," which is detecting whether a person is in proximity to a temperature sensor by sensing the heat emitted from the person. The sensor's data can then be used to determine if a person is present at his workspace or not or whether a meeting room is empty or currently in use. Disruptive is interested in looking at alternatives to the temperature sensor-based approach. The alternative is presented as an exploration task. It is to look at the use of a wide-angle camera-based system, where deep learning can be applied to approximate the number of people present in an area. Also, to look at alternatives to the hardware unit given in the problem description. This problem was previously investigated in a specialization report as a prerequisite before starting the work on this thesis. In the specialization report, the Google AIY Vision Kit V2 [1] was evaluated, a hardware kit designed for learning machine learning. The report is attached to this thesis in Appendix A. In this thesis, the focus changes to the exploration of the concept of transfer learning and apply this to train custom neural network models for the detection of people. In addition to this, the use and deployment of the custom models on accelerator hardware are explored and analyzed. Analyzing the models helps give an overview of differences between models in terms of both energy efficiency and speed. The choice of accelerator for this thesis landed on the Coral Dev Board, an accelerator for machine learning developed by Google. The following summary describes the modifications to the original problem description.

1. Train one or more custom detectors using transfer learning.
2. Evaluate the performance of a custom detector.
3. Test the custom detector on the Hardware kit and evaluate performance in terms of power consumption, inference times, FPS, energy, and energy per operations.

[1]https://www.disruptive-technologies.com/

4. Analyze the feasibility of using the custom trained detector for detecting and counting people.

1.2 Background and Motivation

The use and deployment of deep neural networks for tasks such as computer vision has gained traction during the last decade. The driving factor behind this development has primarily due to an increase in computational capability due to advances in GPU architecture, as well as research into specialized accelerator architectures using FPGA and ASIC. Another key factor that has led to making training and deployment of Deep Neural Network feasible is due to an increase in the amount of data available for training, this comes as a result of the internet. The reason why Deep Neural Network can take advantage of a highly parallel architectures as can be found in GPU and custom ASIC, is due to the fact that DNN the majority of operations performed in a neural network can be decomposed down into matrix multiplications. An interesting example which highlights the difference between GPU and CPU performance when training a neural networks was shown as an example in Baji [2]s paper. Training a network such as AlexNet using a batch of 64 images took 64s to perform on a CPU, while running the same method on a GPU yielded a result in 7.5s. This gap in performance have seen further increase as GPU manufacturers such as NVIDIA has developed custom APIs for running the training process efficiently on their GPUs. In terms of ASIC the focus has been on creating small accelerator architectures that can run neural network models in an efficient manner. In recent years two new categories of devices have seen the light of day, these are Tensor Processing Units and Vision Processing Units. The TPU is a term used to describe hardware accelerators primarily developed and used in Google's data-centers to accelerate their AI infrastructure. They have also developed an Edge TPU for running inference on edge devices, such as their development board Google Coral[3]. These ASICs utilizes systolic arrays to perform matrix multiplications [4] VPUs uses different architectures, a notable example is the Intel Myriad VPU, this ASIC utilizes 16-SHAVE cores which are based on an VLIW architecture[5].

Applying deep neural networks to solve problems is a relatively recent trend, throughout the late 90s, and early to mid-2000 conventional machine learning methods were used instead of DNNs. These methods required careful construction of algorithms and systems to do tasks such as face detection or object classification. Some successful attempts had been made using Artificial Neural Network and Convolutional Neural Network on problems such as digit recognition[6]. The main challenge that came with DNNs was that they required extensive training to do recognition, and as such, creating larger and more complex models was seen as infeasible at that time.

Interest in applying Deep Neural Network for computer vision task flared up again

in 2012, when a DNN was used in the ImageNet Large Scale Visual Recognition Challenge (ILSRC). AlexNet an 8-layer Convolutional Neural Network outperformed the competitors with an top-5 error rate of 15.3% with the runner up achieving an error rate of 26.2%[7]. After AlexNet, many CNNs were developed, trying to further improve upon accuracy. Increasing the accuracy of a model can be achieved by adding additional convolutional layers to the CNN model. The problem with this approach is that network complexity increases. Although leading to increased accuracy, the sheer size and complexity of the models more powerful hardware to run the network and led to increases in inference time, power- and energy consumption. As a result, research has also been focused on creating smaller and more energy-efficient networks. The models need to be light enough to be deployed to resource-constrained devices like smartphones or smaller embedded devices used in robotics or surveillance.
This thesis explores the concept of transfer learning, and looks at how this can be applied to improve neural network performance. To do this four different neural network models are trained on a custom dataset and deployed on the Google Coral Dev Board to analyze the networks in terms of power, energy and energy per operation.

1.3 Relevant Work

This section presents literature on previous work done concerning the topics of this thesis. The primary goal of this thesis is to explore the use of transfer learning to train Convolutional Neural Networks to detect and generate estimates on people present in data such as an image. The secondary goal of this thesis is to deploy the trained CNNs on accelerator hardware and evaluate with metrics such as inference time, energy consumption, and energy per operation.

1.3.1 People Detection using CNNs

Studies that apply Convolutional Neural Networks to detect and count the number of people in an image comes in a wide variety of forms, and terms such as crowd-estimation and pedestrian detection are often used in literature. These terms are relevant in relation to the content for this thesis as the methods employ CNNs to count or detect people. Since there seem to be multiple methodologies for estimating the number of detections in an input image, this section presents regression and detection-based methods. *Counting by detection* is a method for estimation that is done by applying regression to the output of an object detector (which can be implemented by an CNN). The second method uses regression to do crowd density estimation; that is, it uses a Convolutional Neural Network to extract high-level features from an image and then applies regression to the extracted features to generate an estimation. In Chahyati *et al.* [8], the authors applied Faster-RCNN to predict the gender of individual persons in a video and then use this information in conjunction with tracking methods to detect people between frames. In their study, they test the use of Euclidean distance and Siames Neural Networks to improve upon tracking performance. In their study, they do not use this system to count the number of people directly, but it applies to this problem in this thesis since it would only require counting each detection generated by the network. Stewart *et al.* [9] used GoogLeNet and recurrent neural networks to create a network that generates bounding boxes that are used to detect people in crowded scenes. Their design addresses some of the challenges in generic object detection implementations based on CNNs. These networks often use some form of non-maximum suppression, which is used to reduce the number of boxes generated by the CNN. This results in the problem that objects close in proximity might not be detected, leading to poor detection results in crowded scenes. This problem is addressed by applying long-short term memory (LSTM), which feeds information in the RNN part of the network. The LSTM and RNN combination allows each consecutive detection to be generated with prior knowledge of earlier detections. This removes the need for non-maximum suppression, which can help boost performance. Hu *et al.* [10] applies CNNs to estimate the number of people in a crowded input image. The authors created a multi-scale CNN, which is trained to detect crowd features, and this is then used with a feature-count regression network that takes into account crowd count and crowd density. The model is designed for estimating density in very dense scenes containing hundreds or even thousands of people. In Nikouei *et al.* [11] developed a lightweight CNN architecture for real-time detection as an edge service. They employ depthwise separable convolutions, which separate the convolution operation into two stages, which reduces the number of necessary calculations. They also deployed their network to a Raspberry Pi 3 Model B, where they achieved a framerate of 1.79 FPS, which would correspond to an inference time of 558ms. Sam *et al.* [12] uses a concept of switching convolutions. That is, the model uses multiple independent crowd density regressors with different sized receptive field and field of view. This helps the model perform better since a camera might capture the information differently in a crowded scene, depending on its placement and field of view

1.4 Thesis Contribution

The main contributions of this thesis is to shed light on the application of transfer learning and look at how this can be used to improve accuracy of pretrained neural network models. Analysis of different neural network framework and models has also been performed, where emphasis on energy consumption and specifically energy consumed per operation has been one of the primary focuses.

1.5 Thesis structure

The thesis is structured with the following chapters:

- **Chapter 2 - Theory** which explains the theory of CNNs, object detection and how object detection can be used to detect people.
- **Chapter 3 - Experimental Setup:** which details the dataset used in this thesis. Explain the process of preparing the dataset to work with the different frameworks. How the different neural network models are trained and what tools are used to evaluate the different implementations.
- **Chapter 4 - Methodology:** explains the metrics that are used to evaluate the trained models and presents tools used to acquire said metrics.
- **Chapter 5 - Results:** Presents the acquired results after applying transfer learning and deploying the networks to the Google Coral.
- **Chapter 6 - Discussion:** where the results are analysed and advantages and disadvantages in relation to the CNN, hardware and methodology are discussed.
- **Chapter 7 - Conclusion:** which summarizes the findings in this thesis.

Chapter 2

Theory

This chapter will give the theoretical basis which the work of this thesis is based on. In the following sections, the functionality of Artificial Neural Network will be explained in Section 2.2, and the process of training a neural network will be described Section 2.3. Theory in regards to Convolutional Neural Network, which are applied in this thesis, will be elaborated in Section 2.4. Section 2.5 explains the concept of transfer learning. Section 2.6 will present object detection architectures and will show how the older object detection architectures have impacted on the development of newer architectures. Mathematical theory in the following sections are primarily based on the books *Deep Learning* by Ian Goodfellow [13] and *Ml Algorithms* by Bonaccorso [14].

2.1 A brief history of machine learning

The beginning of machine learning and neural networks can be traced back to neuroscience in the early 1940s. In 1943, McCulloch and Pitts [15] presented a model of an artificial neuron, where the neurons were made to function like first-order Boolean logic. The inputs to the neuron could be either 0 or 1. Depending on how the threshold of the output of the neuron is configured, the neuron could implement functions like, *AND*, *OR*, *NAND*, and *NOR*, however, the McCulloch and Pitts neuron was incapable of modeling the XOR and XNOR. Later in 1949, Hebb [16] helped to improve upon the model by addressing how the connectivity between the neurons changed depending on how often the cells were interacting. In his book, they postulated that when a cell repeatedly fires on another cell, the output of the firing cell would change to improve on the efficiency of firing on the receiving cell. The McCulloch and Pitts neuron was modified to allow weighting on the inputs. This allowed the model to consider certain inputs over others, which allowed the model to learn.

Some of the earliest attempts at implementations of neural networks came with research into single and multilayered networks (now often referred to as perceptrons) done by Rosenblatt [17] and his work in 1961 [18]. Rosenblatt's work explored a network of interconnected neurons, in which the neurons could learn new representations after being trained. However, the perceptron faced criticism by Minsky and Papert [19] for not being able to learn non-linear data. Pattern recognition in more complex data, like images or sound, was still a challenge for earlier networks. In 1980, Fukushima [20] presented an architecture which allowed for pattern recognition, and was an extension to earlier work done in 1975 [21]. The Neocognitron is recognized as the inspiration of modern-day convolutional neural networks; the architecture allowed for the extraction of data from images. One of the challenges with finding patterns in images is that objects are positional dependant in the image. A network that does not feature positional invariance will in most cases be incapable of detecting or recognizing features. Fukushima [20] addressed this by having two types of cells in the network, an S-cell which worked as the feature extractors and C-cells which correspond to the complex cells in the visual cortex, these cells respond to and corrects the positional error in the input stimulus adding positional invariance to the model. A huge breakthrough in neural network research came in the mid-1980s. Rumelhart and McClelland [22] in their report "Learning Internal Representations By Error Propagation" presented a method for training artificial neural networks using gradient descent and was a "rediscovery" of methods researched during the 1970s, the method is called backpropagation, and it is still used to this day when training modern neural networks. Armed with this knowledge, LeCun *et al.* [6] created a convolutional neural network that used backpropagation to learn to recognize handwritten digits. This architecture is often credited as the first definition of the modern implementation of convolutional neural networks. The 90s and early 2000s saw less focus on research of deep neural networks, and there are multiple reasons for this, training deep networks using backpropagation was expensive in terms of hardware and the development of other methods for pattern recognition and classification like support-vector machines [23] gained popularity. Research into artificial neural networks flared up again during the late 2000s, in 2012 Krizhevsky *et al.* [7] achieved breakthrough performance outperforming the competition in the *2012 ImageNet Large Scale Visual Recognition Challenge*. Their paper resulted in a new wave of research into convolutional networks.

2.2 What are Artificial Neural Networks

The term artificial neural network or ANN describes a set of interconnected nodes that form a connected network or graph. The structure of the network is loosely based on how the neurons in a biological brain are structured. An ANN can be applied to a wide variety of problems, such as regression and classification. However the primary purpose of an ANN is to find an approximation to an objective function **f(x)**[13]. The network does this by utilizing optimization methods that are used to reconfigure the network's internal parameters. This is used to find the best possible configuration of the network's parameters, such that the network's loss function is minimized.

2.2.1 Biological Neuron

Before the artificial neural network is explained, a brief explanation of how a biological neuron works and how it is structured will be described. This should give some idea of the similarities and inspirations concerning the structure of a neural network.
A biological neuron is a type of cell which is found in the brain. These cells are responsible for a lot of different functions in the body. Each neuron is connected to other neuron forming pathways in the brain, which are used to make memories, control motion, process things such as; thought, auditory signals from the ears, or visual signals from the eye. Since the neuron is a cell it features a *cell body* which is also called *soma*. The cell body houses the core or nucleus of the cell. The connections between neurons are formed using two types of connections, the *dendrites*, which acts as the input paths to the cell. The output path of a neuron is called *axon*. To explain how a biological neuron operates the functionality can be simplified and generalized. A neuron works by summing each input signals received on the dendrites. Each of these inputs may be weighted by the cell, giving certain inputs a higher priority than others. Depending on the value of this sum, the cell might fire a signal on its output, this is determined by how the neuron has been configured. [24]. Figure 2.1, shows an illustration of how a biological neuron can be visualized[1].

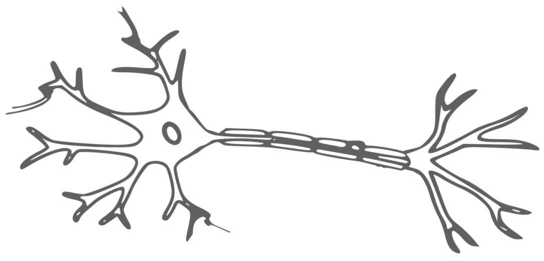


Figure 2.1: An example of a biological neuron. The image has no given name, by Unknown artist, Licensed as "Free to Use"

2.2.2 Artificial Neuron

The artificial neuron is the basic building block used in artificial neural networks. It shares similar features and functionalities to a biological neuron. An artificial neuron takes in a fixed set of inputs x_j for $j = 1, ..., n$, where each input corresponds to a neuron's activation in a previous layer. Each input is scaled by a set of weights w_i and summed in the neuron. The result of the summation is passed to an activation function *f(x)*[24]. Figure 2.2 show how an artificial neuron can be visualized.

[1]https://pixabay.com/vectors/brain-neuron-nerves-cell-science-2022398/

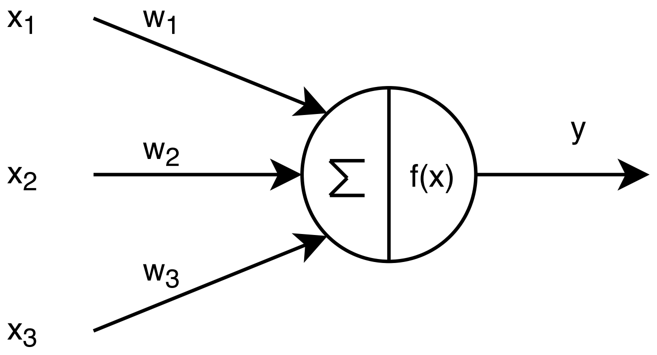


Figure 2.2: An example of an artificial neuron with three inputs

The activation y_i of an artificial neuron i can be described as in Equation 2.1

$$y_i = f\left(\sum_{i=0}^{N} w_i x_i + b\right). \tag{2.1}$$

It can also be modeled using vectors where the output of the neuron is the dot product between the input feature vector $\mathbf{x}$ and the weight vector $\mathbf{w}$. In this case, the bias term is incorporated into the feature and weight vector. Equation 2.2 describes this relation

$$y_i = f(\mathbf{w}^T \cdot \mathbf{x}). \tag{2.2}$$

2.2.3 Fully-connected Artificial Neural Network

When a set of artificial neurons are stacked together to form layers, a neural network is formed. Here each layer contains a fixed set of neurons. Each neuron is connected either sparsely, i.e., a neuron is connected to a subset of neurons in the next layer, or it is fully-connected, meaning that each neuron is connected to every neuron in the next layer. An artificial neural network has an input layer, one or multiple hidden layers, and the output layer. To designate a neuron i in a given layer L, each neuron can be given a designation as $a_i^{(L)}$. An example of an artificial neural network that has an input layer, one hidden layer, and an output layer is given in Figure 2.3.

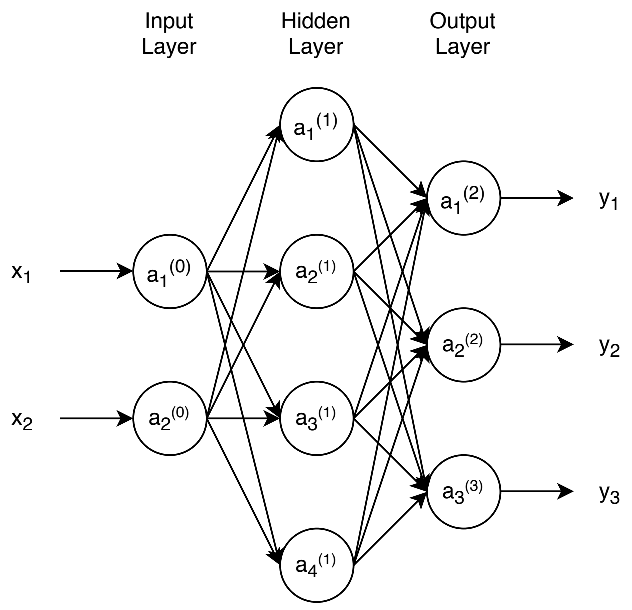


Figure 2.3: Artificial Neural Network with an input layer using two input neurons, a hidden layer with four neurons and a three neuron output layer

2.3 Training a Neural Network

2.3.1 Loss Function

To evaluate how well a neural network model performs, the notion of a loss function must first be explained. There exist conflicting views on the difference between the term loss function and cost function [13], some use the term loss function to describe the difference between a single input sample in relation to the generated output prediction, and that the cost function is the average loss over the entire dataset. In this thesis, the terms are used synonymously, the reason for this is that many papers refer to the cost function simply as the loss [25–27].

A loss function calculates the error of an ANNs generated prediction, and is found by looking at the difference between the predicted output, that is $f(x_i;\theta)$, and the expected output y_i, which is often denoted as a groundtruth. Here x_i is used as a notation to imply a single sample from a dataset $X = \{x_0, x_1, x_2, ..., x_n\}$ and labels y_i from a dataset $Y = \{y_0, y_1, y_2, ..., y_n\}$. Almost all modern neural networks are trained using some form of *maximum likelihood estimation*, a statistical method that tries to minimize dissimilarity between the empirical probability distribution of a dataset against the probability distribution generated by the model[13]. One of the most common loss functions derived using this methodology is defined as the negative of the log-likelihood, which is often described as the cross-entropy between the prediction and the training data. The general form of cross-entropy is called categorical cross-entropy loss and is defined in Equation (2.3)[14]

$$L(Y,X;\theta) = -\frac{1}{N}\sum_{i=0}^{N-1} y_i \log[f(x_i;\theta).] \tag{2.3}$$

Where θ is the network parameters, in the case of a generic fully-connected ANN, θ is simply the weights (w) and bias (b) terms. The choice of loss function depends on the type of task the network is supposed to perform, but it can be categorized into two kinds of losses: regression loss and classification loss.

Regression Loss

Regression deals with the problem of creating a model that predicts numerical values. An example of a regression model could be to predict housing prices from data such as lot size, number of rooms, and location. An easy way to evaluate the performance of a regression model would be to look at the difference between the numerical value predicted by the model and compare this to the expected output. Two of the most common loss functions are the mean absolute error (MAE) and mean square error (MSE). These losses equate the mean absolute difference or the mean squared difference between the predicted output and the correct output value. The losses are sometimes also called L1 and L2 losses[28]. The mean absolute error or averaged L1 loss is given in Equation (2.4) while the mean square error or averaged L2 loss is shown in Equation (2.5)

$$L(Y,X;\theta) = \frac{1}{N}\sum_{i=0}^{N-1} |y_i - f(x_i;\theta)|. \tag{2.4}$$

$$L(Y,X;\theta) = \frac{1}{N}\sum_{i=0}^{N-1} (y_i - f(x_i;\theta))^2. \tag{2.5}$$

Classification Loss

Classification deals with the problem of accurately labeling data into one or more classes. An example can be a model that tries to classify photos of fruit. A classification model should then correctly identify the class that each image belongs to. For models tasked with classifying data into one of two categories, the binary cross-entropy loss is the most commonly used function. Binary cross-entropy is a particular case of the categorical cross-entropy and is defined in 2.6[14].

$$L(Y,X;\theta) = -\frac{1}{N}\sum_{i=0}^{N-1} [-y_i \log(f(x_i;\theta)) + (1-y_i)\log(1-f(x_i;\theta))]. \tag{2.6}$$

Cross-entropy loss is also applicable to multi-class classification models. In that case, the categorical cross-entropy, as defined in Equation (2.3) is used.

2.3.2 Backpropagation and Gradient Descent

The process of training a neural network can be viewed as an optimization problem, where the goal of training is to minimize the network's loss function by adjusting the network's parameters, that is the weights and biases. This is achieved through the use of a process known as backpropagation, which is used in conjunction with a gradient-based optimization technique called *Gradient Descent*. This process is used to modify the weights and biases in the network, which in turn changes the output of the loss function.

Gradient Descent

Gradient Descent is an algorithm that is used to calculate the gradient of the loss function in relation to the network's current parameters and use this gradient to modify the network's weights and biases to improve the network's loss. Minimizing the loss is achieved by moving in the negative direction of the gradient using a limiting factor known as the learning rate ϵ. The parameters are updated by applying the following equation [14]

$$\theta \longleftarrow \theta - \epsilon \nabla_{\theta} L. \tag{2.7}$$

One of the major challenges in calculating the gradient is to calculate the partial derivatives of the loss since these depend on the partial derivatives for each weight and bias in the network. This is shown in Equation 2.8.

$$\nabla_{\Theta} L = \left[\frac{\delta L}{\delta w_i}, \dots, \frac{\delta L}{\delta w_n}, \frac{\delta L}{\delta b_i}, \dots, \frac{\delta L}{\delta b_m} \right] \tag{2.8}$$

Where n is the number of weights in the network, and m is the number of bias terms.

Backpropagation

Backpropagation is the method which solves the problem of calculating the loss gradients. It allows for efficient calculations of the derivatives in the loss gradient. The method works by first calculating the gradient of the last layer in the network. This is done by applying the chain rule to the loss function, simplifying some of the calculations. The resulting derivatives are passed backward through the network, which allows the previous layers to calculate the derivatives in relation to the loss function efficiently. To clearly illustrate the point, let's take a look at the output of a single layer in a neural network. To simplify matters, let's consider a network containing only two connected nodes and analyze this. Such a network can be seen in Figure 2.4. The activation on the output can be given as in Equation (2.9)

$$y^{(L)} = f(a^{(L)}). \tag{2.9}$$

Where $y^{(L)}$ is the activation as explained in Equation (2.1) of the last neuron in the network. $a^{(L)}$ denotes the weighted sum of the input and the bias of the

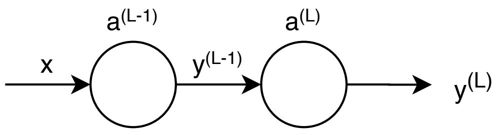


Figure 2.4: An example of a simple neural network containing two connected neurons

neuron in layer L. Since the loss is a function of the activation of the last layer and the activation is a function of the weight of the node, bias, and activation on the input from the previous node in the network. The derivative of the loss in the last node can be found by looking at the weight, bias, and activation by splitting the calculation up into parts using the chain rule. In Equation (2.10) the derivative is calculated with respect to the last node's weight, but the method is identical when calculating for bias and node activation for previous nodes.

$$\frac{\delta L}{\delta w^{(L)}} = \frac{\delta L}{\delta y^{(L)}} \frac{\delta y^{(L)}}{\delta a^{(L)}} \frac{\delta a^{(L)}}{\delta w^{(L)}} \tag{2.10}$$

We can apply this method to calculate the derivative in relation to the previous layers weight. This gives the relation in Equation (2.11).

$$\frac{\delta L}{\delta w^{(L-1)}} = \frac{\delta L}{\delta y^{(L)}} \frac{\delta y^{(L)}}{\delta a^{(L)}} \frac{\delta a^{(L)}}{\delta y^{(L-1)}} \frac{\delta y^{(L-1)}}{\delta a^{(L-1)}} \frac{a^{(L-1)}}{\delta w^{(L-1)}} \tag{2.11}$$

The interesting thing to notice here is that both equations share similar derivatives; this is the critical observation that makes up the foundation of backpropagation. *That is that we can calculate all the derivatives of the last layer and then propagate these back to the previous layers, reducing the number of derivatives that need to be calculated, thereby reducing the required number of operations when calculating the loss gradient*

2.4 Convolutional Neural Networks

Convolutional neural networks are a sub-type of artificial neural networks. These types of networks are inspired by how the receptive field in an eye captures information. In a CNN, the neurons in the convolutional layers perform a discreet convolution between data provided to the network and pretrained filter kernels. The CNN implements receptive fields by utilizing different sized convolutional filters. CNN is most often used on images, to do things like object classification or detection.

2.4.1 Convolution Operation

A convolution is an operation using two functions with real-valued arguments. The conventional mathematical definition of a convolution is given in Equation

2.12[13]

$$s(t) = (k * x)(t) = \int_{\tau} k(\tau)x(t-\tau)d\tau. \tag{2.12}$$

The function **x(t)** is often referred to as the input or feature map when it is used in neural networks, while **k(t)** (also often represented as **f(t)**) is referred to as a filter or kernel. The result of the continuous convolution defined in Equation 2.12 is a new function **s(t)**. **s(t)** can be described as the weighted average between an input function and a time-shifted weight function.

The time-continuous convolution in Equation 2.12 differs from the types that are applied in neural networks. In a convolutional neural network, the input to the network is often either a 2D image (single-channel monochrome image) or a 3D volume (multi-channel image, where each channel represents a color channel). In this case, the convolution is applied using discreet convolution between the input data **x** and the filter/kernel **k**.
This operation can be performed using a sliding-window technique; in this case, the output is a sum of element-wise multiplication between the values in the kernel with a subset of values in the input. This can be described as in Equation 2.13

$$s(i,j) = \sum_m \sum_n k(m,n) * x(i-m, j-n). \tag{2.13}$$

In Equation 2.13 the value of m and n is derived from the size of the kernel **k**. If the kernel is of size $w * h$ then the range of m and n can be found from:

$$\begin{aligned} -(w-1) \leq m \leq (w-1) \\ -(h-1) \leq n \leq (h-1). \end{aligned} \tag{2.14}$$

In machine learning libraries one often applies another method called cross-correlation which is similar in nature to a convolution [13], libraries like Tensorflow[2] and PyTorch[3] uses this implementation method. The equation for cross-correlation is given in Equation 2.15

$$s(i,j) = \sum_m \sum_n k(m,n) * x(i+m, j+n). \tag{2.15}$$

The output of the convolution over the input image is a new "image" called feature map. The size of a feature map generated by a convolution depends on multiple factors. These are the size of the kernel w, h. The stride of the kernel; that is how many pixels the kernel is displaced per calculation. Padding on the input data also impacts on the size of the output. The size of the generated feature map can be calculated using Equation (2.16)

$$\begin{aligned} W_f &= \frac{W_i - w + 2 * P}{S} + 1 \\ H_f &= \frac{H_i - h + 2 * P}{S} + 1 \end{aligned} \tag{2.16}$$

[2]https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/layers/convolutional.py
[3]https://pytorch.org/docs/stable/nn.html#convolution-layers

Where W_i, H_i is the size of the input image, w, h is kernel width and height, S is the stride of the kernel, and P is the padding applied on the input. Figure 2.5 shows a convolution with input of size 4x4, a kernel of size 3x3 with a stride of 1 and no padding. In this case the width and height of the output feature map can be calculated to be $\frac{4-3+2*0}{1} + 1 = 2$.

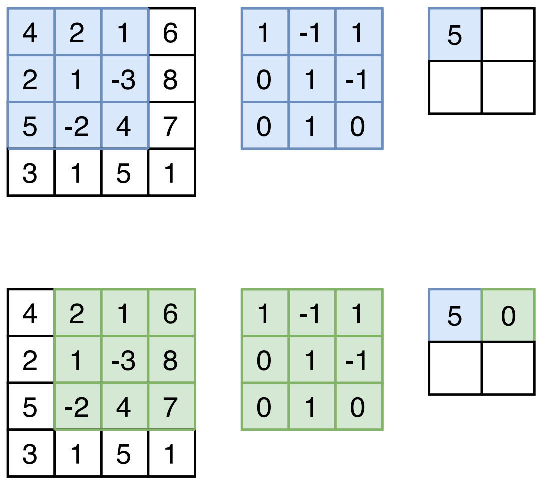


Figure 2.5: Example of Convolution Operation on a 4x4 input matrix with a 3x3 kernel. Reproduced from [29]

2.4.2 Convolution Layers

In a convolutional neural network, a convolutional layer consist of a configurable amount of nodes often referred to as filters. Each filter is applied to the input data using the convolution operation given in Equation 2.12. The filters operate on each channel/layer in the input data, so each filter must have the same depth as the number of channels in the input. The output of each filter is a feature map. Each node/filter has its own set of weights and biases, and are capable of extracting different, distinct features based on how they have been configured during training. These features might be simple like edges, lines, or curves, or it can be more complex like the contour of an eye. The input to the convolution layers is an N-dimensional volume. A color image can be seen as a 3-dimensional volume due to being composed of three color channels. Each of the filters in a convolution layer produces a feature map, and the feature maps are combined into a volume which is passed as the output of the layer. The volume would be an M-dimensional volume, where M corresponds to the number of filters/kernels in a convolutional layer.

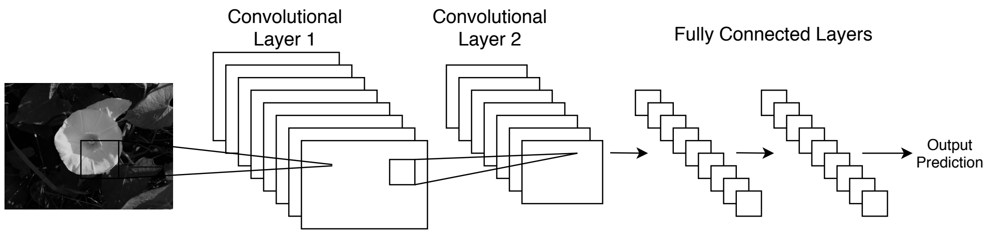


Figure 2.6: An illustration of a Convolutional Neural Network with two convolutional layers. Not shown on the image is the pooling layer in between the convolutional layers

2.4.3 Activation Layers

Following a convolutional layer, there is added an activation layer. The purpose of this layer is to introduce non-linearity into the data on the output of the convolutional layer. Convolution is a linear transformation, and as such, it is not capable of creating non-linear separations between different classes[24]. There exist a lot of different activation functions, the choice of activation function largely depends on the application on the network, and layers it is applied to. For CNN, the primary types of activation functions are the Sigmoid functions, ReLU and LeakyReLu, and Softmax for the fully connected layers. The activation functions are briefly explained in the following subsections.

Sigmoid function

A Sigmoid function is a class of non-linear function featuring a value between -1 and 1 or between 0 and 1. An S-shaped curve characterizes the shape of the sigmoid functions. Two common Sigmoid functions are the logistic function as defined in Equation 2.17 or the hyperbolic function defined in Equation 2.18.

$$f(x) = \frac{1}{1 + e^{-x}} \tag{2.17}$$

$$f(x) = \tanh x \tag{2.18}$$

ReLu

ReLu or rectified linear units are one of the most common activation functions to date. The output of the function is 0 when the input is less than zero and is equal to the input when greater than zero. It is defined in Equation 2.19.

$$f(x) = \max(0, x) \tag{2.19}$$

Leaky ReLu

The Leaky ReLu is a modified activation function based on the parametric ReLu and is defined in Equation 2.20.

$$f(x) = \max(0, x) - \beta \min(0, x) \tag{2.20}$$

In the ordinary ReLu, the network can stop training if the gradient becomes zero, which happens if the input is negative. Leaky ReLu solves this problem by setting β to a small value. This allows the gradient to hold a non-zero value since a small negative value can "leak" through the activation function[24].

Softmax

The softmax function is used in the last fully-connected layer in a classifier to generate a probability distribution over the detectors classes. It is defined in Equation 2.21

$$f(x) = \frac{e^x}{\sum_{j=1}^{n} e^{x_j}} \tag{2.21}$$

2.4.4 Pooling Layers

Pooling layers are used as downsampling layers and are used to reduce the size of a feature map. Doing this helps in reducing the number of computations needed per convolutional layer, but it also serves another purpose in that it provides some translational invariance to the input[24].

Pooling is often implemented in one of two fashions, either max-pooling or average-pooling. Not all CNN architectures use pooling, recent CNNs (like Mobilenet V2) replaces the pooling layers with a strided convolution. Max-pooling is the type of pooling layer which is most often used. It works by dividing the input data into regions. For each number in the region, the number with the largest value is selected. Average-pooling is similar to max-pooling. The difference is that the returned value is the average value of each region. Figure 2.7 shows the difference between the two types of pooling methods.

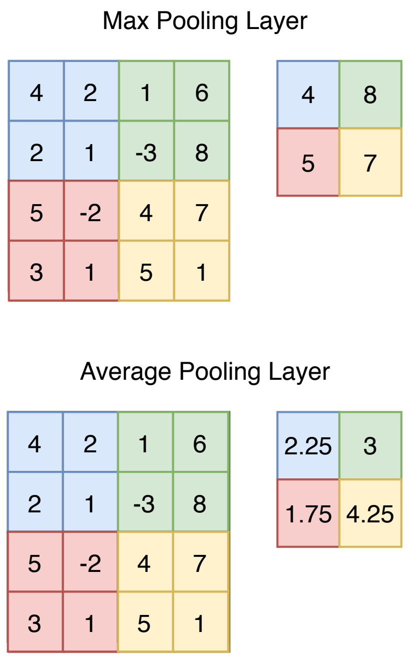


Figure 2.7: Pooling is performed in one of two types either max-pooling or average-pooling. Figure reproduced from [29]

2.5 Transfer learning

One of the major challenges when training a neural network model is the requirement of a large scale dataset. There is a need for such a dataset since it allows the model to properly generalize to the classes it should be able to classify or detect. To overcome this problem it is better to use an already trained model as a starting point and then fine-tune it to your specific dataset. This concept is what is known as transfer learning. Transfer learning allows an already existing model that has been trained on large scale datasets such as Imagenet[30], Pascal VOC[31] or COCO[32] to be fine-tuned to a custom dataset, this can help increase the overall accuracy of the model, but can also be used to retrain the model to detect classes that is not present in the original dataset.

Applying transfer learning is not always the best way to improve model performance depending on the size of your custom dataset and how similar this dataset is to the datasets used during training of the pretrained models [24]. If the custom dataset contains a considerable amount of images and is similar to the original dataset the model has been trained on, transfer learning can be applied. Since both datasets are similar, many features in each dataset are shared, making it feasible to use the pretrained model as a starting point and then fine-tune the model's weights and biases. Another option is to retrain the model from scratch, which can be done since the dataset is large. This sentiment is also shared in the case that the custom dataset is small. As long as the datasets are similar in terms of sharing data with similar features, transfer learning and fine-tuning can be applied to "re-train" and improve the model's performance. In contrast, a large custom dataset that is too dissimilar to the original dataset will not benefit much from applying fine-tuning. Since both datasets contain features not shared by the other, the point of using the pretrained model as a good starting point becomes moot. In this case, it would be better to train the model from scratch, something that can be done since the dataset is sufficiently large. The most problematic situation is a dissimilar dataset that is also small. In this case, doing training from scratch would yield a model with poor performance. There is a possibility to apply fine-tuning by using the pretrained model. However, the performance of the model would suffer since the already learned features would not translate well to the custom dataset[24].

2.6 Object Detection

This section will take a look at how an object detector using CNN works. It starts by looking at the earlier attempts at doing object detection, as these have had an impact on the design of modern architectures. One of the first object detection networks based around CNN is the R-CNN, which was later improved upon in Fast-R-CNN and Faster R-CNN, up to some of the more modern detectors such as the ones used in this thesis; SSD and YOLO. Object detectors are usually divided into two types, single-stage, and two-stage detectors. A two-stage detector uses a region proposal network to generate a set of possible detection proposals. These proposals are sent to the second stage, which uses a classifier to generate class probabilities. The one-stage detector differs from the two-stage by implementing both the region proposal and classifier into one network.

2.6.1 R-CNN

Regions with CNN features or R-CNN is an object detector based on the use of a convolutional neural network and was developed by Girshick *et al.* [33] in 2014. The detector is a two-stage detector. In R-CNN, a selective search algorithm is first used to generate 2000 region proposals from an input image. Each proposal is warped; that is, the spatial dimensions of the proposal region are sized to the input size of the R-CNN network. The classifier used in the network is a combination of a CNN implementation based on Alexnet [7], which extracts features from the proposed regions, the features are passed to a support vector machine (SVM) for predicting the class scores and offsets for the bounding boxes.

2.6.2 Fast R-CNN

One of the significant challenges with R-CNN is that training is expensive. The process of training was a multi-stage process. It needs to train for CNN feature extraction, then fit the SVM to the features and, at the end of the training, learn bounding-box regressors. It also required a lot of storage to cache features generated during training [33]. Fast R-CNN [27] remedied the shortcomings in R-CNN by presenting a new algorithm which performs training in one-stage rather than multiple stages. The architecture of Fast R-CNN is based on the work previously done in the R-CNN paper. However, the selective search algorithm used to generate region proposals for the input is removed. Instead, Fast R-CNN takes an entire image as input and generates features using a Backbone-CNN. This can be any pretrained CNN. In the Fast R-CNN implementation, VGG16 is used[27]. From these features, regions of interest (RoI) are identified using a region proposal algorithm like selective search. Each RoI is passed to an RoI pooling layer, followed by a set of fully connected layers that feed into two output layers, one softmax layer used for classification and a second layer for bounding box regression. The RoI pooling layer reshapes the input feature to a fixed size feature map of configurable size.

2.6.3 Faster R-CNN

Faster R-CNN improves upon Fast R-CNN by replacing the selective search algorithm with a second convolutional neural network, which generates region proposals called Region Proposal Networks or RPN. The input to the RPN is an image of arbitrary size; its output is rectangular object detections and an objectness score per detection[34]. The objectness score defines whether an object is believed to be present in a specific bounding box, or not. The RPN shares a set of its convolutions layers with the object detection network to further increase the efficiency of proposal generation. Fast R-CNN is used as the object detection network in Faster R-CNN [34].

2.6.4 SSD

SSD is a one-stage network developed by Liu *et al.* [25]. The network differs from the aforementioned R-CNN networks by replacing the region proposals and feature re-sampling with a deep feed-forward convolutional network. This generates a fixed-size collection of bounding boxes and the score associated with the presence of an object in each bounding box[25].

Architecture-wise, the network can be seen as a compound of two parts; the base network and the auxiliary structure, which adds extra feature layers to increase detection accuracy. The base network is simply a repurposed CNN-based classifier, where the last layers of the network (the fully-connected layers used for generating class probabilities) are removed.

The auxiliary structure is added to the end of the base network. This structure adds multiple convolutional layers. Each layer decreases in size progressively and is used to allow for the detection of objects at different scales. Feature maps generated in the layers earlier in the detection pipeline will be large and as such the filters being applied to the feature map will detect smaller objects. In comparison, the smaller feature maps generated by the layers later in the network will primarily detect larger objects.

The way SSD generates an object detection is by applying a set of convolutional filters to each feature map produced by the layers in the auxiliary structure and the last layer of the base network. For each feature map, two convolutional kernels of size 3x3 with p channels are used to generate bounding box offsets and class scores [25].
A set of k default boxes are associated with each cell in the feature map. These boxes have prespecified aspect rations and have their position fixed in relation to each cell. A cell, in this case, refers to a position in the feature map. For a feature map of size m * n, the number of cells is equal to the size m * n. So the number of default boxes applied to each feature map is of size m * n * k. For each of these default boxes, the offset and class scores are calculated. The offset contains four values, one for x offset, one for y offset, a value for width, and the last for height. The network's hyperparameters determine the number of classes. This implies that for each feature map a total of $(c + 4)kmn$ output values are produced for a total of m * n * k boxes [25]. These are fed to a non-maximum suppression layer, which reduces the number of generated predictions to produce the final output.

2.6.5 YOLO

You only look once, or YOLO is a one-stage network developed by Redmon and Farhadi [35]. YOLOv3 features a 53-layer feature extraction network called Darknet-53, followed by 53-extra layers for implementing object detection.

Detection in YOLOv3 is performed over three different scales, which allows the network to better detect smaller objects, which was a challenge in earlier implementations[26, 36]. YOLOv3 also utilizes the concept of anchor boxes to reduce the time needed to compute bounding boxes. Bounding boxes in YOLOv3 are generated on the dataset rather than using pre-computed boxes, as is the case for SSD and Faster R-CNN. This is done to increase the network's ability to learn to produce good detections [36].

The way YOLOv3 generates detections is by first dividing the input image into three grids of three different sizes. For each cell in each grid, a fixed set of bounding boxes are generated using anchor boxes. Each bounding box contains a set of predicted values. Five values predict box attributes, 4 of which are values for computing bounding box offsets (x, y, width, height) and the last value for box confidence value. The box confidence value gives a measure of the probability of an object being present in the generated bounding box. In addition to these 5 values, there are c class probabilities attached to each box, where c corresponds to the number of classes. Multiple feature maps are concatenated into the different scaled layers. Each of the scaled layers takes advantage of this concatenation to enhance the semantic information in the image and to generate more accurate detections. For the scaled detection layers, the earlier layers are up-scaled and added to the later layers to enhance the features [35].

In terms of the number of generated boxes, YOLOv3 generates k boxes per cell in each of the three feature extraction layers, assuming each layer is of size n * n the number of boxes generated by the network is equal to n * n * 3 * k, where the size of the networks output tensor is of size n*n*3(k*(5 + c))[35].

Chapter 3

Experimental Setup

3.1 Models

In this thesis four object detection models are explored. These are based on Single Shot Detection and YOLO. The SSD models uses Mobilenet V2[37] as its feature extraction and the model is acquired from the Tensorflow Model. Zoo[1]. For YOLO, YOLOv3[35] and YOLOv3-Tiny[35] was chosen.

3.2 Datasets

3.2.1 MSCOCO

Common Objects in Context or *COCO*[2][32] is a dataset released by Microsoft. It features over 300 thousand images, with over 1.5 Million object instances and 80 object categories. The dataset is not used directly for the experimental part of the thesis, however, the pretrained models used in this thesis have been pretrained on this dataset.

3.2.2 Open Image Dataset

The dataset used for the experimental part of this thesis is based on data from the Open Image Dataset[38][3]. Open Image Dataset is a dataset created and curated by Google. It features more than 15 Million+ bounding boxes on 600 class categories, 2,7 Million+ segmentation masks for Image segmentation, and around 60 Million image-level labels.
In this thesis, a tool called OIDv4-ToolKit[4][39] is used to download a subset of the dataset. The tool allows the user to specify the subset of the data to download (i.e., whether to download train, test, and validation datasets) and what kind

[1]https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/detection_model_zoo.md
[2]`http://cocodataset.org/#home`
[3]`https://storage.googleapis.com/openimages/web/index.html`
[4]`https://github.com/EscVM/OIDv4_ToolKit`

of classes should be downloaded. In general, for applying transfer learning, the dataset does not need to be very large, one can achieve relatively good results using a few images. However, to supply the models with a decent basis to do transfer learning in this thesis, a set of 20000 images was chosen for the training set, and 4000 images were chosen for the test and validation set. The class label selected is Person, since this is the object of interest.

3.3 Software Implementation

In this thesis, two types of frameworks are used to train and deploy neural network models. The Tensorflow Object Detection API[5] is used to train and test two object detection models; Mobilenet V2 + SSD and a quantized Mobilenet V2 + SSD model. The Tensorflow Object Detection API is used to do the training. While for YOLO[6] which is based on the darknet framework, training, and testing is handled by the Darknet API. The following subsections explains how the data must be processed, to convert it from Pascal VOC format to formats used by the different frameworks. An example of the Pascal VOC format is supplied in Appendix C. Code used to process is available at GitHub, links are given in Appendix A.

3.3.1 Tensorflow-based Models

The process of structuring the dataset and training a TensorFlow model can be divided into the following steps:

- Generate a CSV file, by cycling trough each label file. Each label file is structured in the Pascal VOC format and contains information about the image (filename and size), and information about detections (bounding box coordinates, object class).
- From the CSV file, generate Tensorflow Records.
- Configure the relevant configuration file for the network. These are based on the samples provided in the Tensorflow Object Detection API.
- Start the training process by using the model_main.py in the object detection folder in the Object Detection API.

These steps can be applied to multiple models in the model zoo.

3.3.2 YOLO

For YOLO, the process is somewhat similar, however, instead of generating a CSV file that contains all the information about every image and detections. YOLO requires a text file containing the class and bounding box coordinates in the format **(object_class x y w h)** for each image in the dataset. Another caveat is that the bounding box coordinates used for YOLO use the bounding box center for x, y

[5]https://github.com/tensorflow/models/tree/master/research/object_detection
[6]https://github.com/pjreddie/darknet/wiki/YOLO:-Real-Time-Object-Detection

coordinates, and width and height. This differs from the Tensorflow based models, which uses the top-left corner of the bounding box x1, y1, and the bottom-right corner of the bounding box x2, y2. So the coordinates must be transformed into this form. The process of preparing data and model for training is as follows:

- Generate a text file containing one object ground truth per line. Each image in the dataset requires one text file.
- Configure the YOLO architecture by modifying the yolov3.cfg configuration file
- Start training by using the darknet command **./darknet detector train**

3.3.3 COCO API

Pycocotools[7] is a tool-chain used to evaluate the Tensorflow based networks during and after training. The API is set as the standard evaluation method in the Tensorflow Object Detection API and is, at this time, one of the more commonly used evaluation methods. The performance metrics generated by the tool-chain include mean Average Precision (mAP) and Recall for a variety of different Intersect over Union values. These metrics will be further explained in Methodology.

3.3.4 Evaluation of Darknet based Models

There is no direct way of evaluating the Darknet based network using Pycocotools. The results from running an inference over the test set must be translated to a form that is accepted by the Pycocotools API. This is done by using a conversion script created by Github user ydixon[8]. The program takes results generated by a darknet model as an input. It parses this input so that the results can be passed to Pycocotools and then evaluated using the COCO evaluation model. The purpose of doing this conversion is to produce results using the same tool for each of the frameworks, which in turn makes it easier to compare the different detectors.

3.3.5 Network Configuration

Each network features different configuration options. This is a result of using two different frameworks. The following Table 3.1 outlines the training configurations. It also states the estimated number of flops required to run inference on each network.

3.4 Hardware Setup

To train and test the networks different hardware architectures have been used. Training of the network was performed using the following hardware specifications:

[7] https://github.com/cocodataset/cocoapi/tree/master/PythonAPI/pycocotools
[8] https://github.com/ydixon/mAP_eval

Parameter	Mobilenet	Mobilenet (Quantized)	YOLOv3	YOLOv3-Tiny
Batch Size	24	6	24	64
Input Shape	300 x 300	300 x 300	320 x 320	320 x 320
Learning Rate	0.004	0.004	0.001	0.001
Number of FLOPS	1.28 GFlops	1.29 GFlops	38.6 GFlops	3.2 GFlops

Table 3.1: Configuration options for each network.

- Intel I7-7700K CPU @ 4.2GHz
- 16GB of RAM
- Nvidia GTX1080 (8GB of Video memory)

.

While testing has been performed on Desktop CPU, GPU and Google Coral Dev Boards CPU and TPU.

3.4.1 Google Coral Dev Board

The Google Coral Dev Board is a hardware platform designed specifically for running low-power, low-inference machine learning models. The board is divided into two modules, the carrier board featuring the peripherals such as audio, HDMI, USB connections, and GPIO. The second part is the Coral SoM. The System-On-Module incorporates an NXP iMX 8M SoC, Google Edge TPU ML accelerator, Cryptographic coprocessor, Bluetooth, Wi-fi, and more. A full list of features are available in the datasheet of the Coral SoM[40] and the datasheet of the Google Coral Dev Board [3].

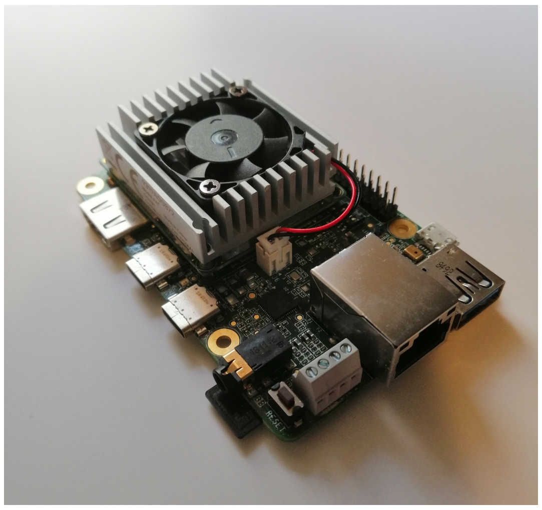


Figure 3.1: The Google Coral Dev Board

Chapter 4

Methodology

4.1 Detection Metrics

In this section, the metrics used to evaluate the performance of the retrained object detectors are discussed. It starts by presenting the metrics: IoU and definition of detection outcomes, then Precision and Recall will be explained as these metrics needs to be understood to make sense of mean Average Precision (mAP) which is the current de-facto standard for evaluation of object detection networks. The generation of these metrics is in relation to the MS-COCO and follows this evaluation standard[1]. In addition to these metrics, the inference time of the network and the number of floating-point operations are of interest in this thesis.

4.1.1 IoU

Intersect over Union or IoU, also known as the Jaccard Index[2] is a common metric used in the process of evaluating the accuracy of an object detector. This metric measures how well the position and aspect ratio of two boxes correspond and is defined mathematically in Equation (4.1).

$$IoU = \frac{Area\ of\ Overlap\ of\ Boxes}{Area\ of\ Union\ of\ Boxes} \tag{4.1}$$

Here the area of overlap between the two boxes is related to the total area formed by the compound of the boxes. This can be visualized, as shown in Figure 4.1.

4.1.2 Detection outcomes and the confusion matrix

An essential part of detection systems is to generate accurate and correct predictions. But what constitutes a proper detection, and how can we quantify this? In both classification and detection systems, this is done by using a concept of positives and negatives. A positive indicates that there exists an object in a given box.

[1]http://cocodataset.org/#detection-eval

[2]https://deepai.org/machine-learning-glossary-and-terms/jaccard-index

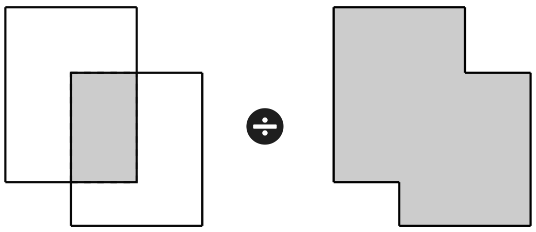


Figure 4.1: IoU can be visualized by looking at the intersect of two boxes over the union of the same boxes

The prediction of the detection system might generate erroneous detections in relation to the actual ground truth. This results in the concept of true and false positives and negatives. These can be defined as follows.

- **True Positive** - The object detection system predicts a box that overlaps to a certain extent with a ground truth box, thereby correctly recognizing the presence of an object.
- **False Positive** - The object detection system generates a prediction that does not correspond to any box in the ground truth. This implies that there might be an object in the detected box, but there exists no ground truth to validate this
- **True Negative** - The object detection system does not generate any prediction, and the ground truth does not contain any objects.
- **False Negative** - The object detection system does not generate any prediction, while the ground truth states that there exists an object in a box.

When these metrics are structured into a matrix, it is known as a confusion matrix and can be seen in Figure 4.2. So a detection generated by a CNN constitutes a

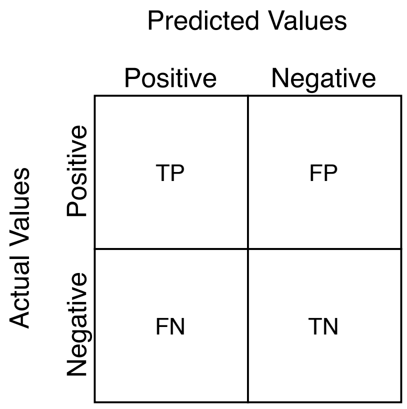


Figure 4.2: An example of a confusion matrix

true positive if the predicted box belongs to the ground truth box class and that the IoU overlap is greater than a set threshold. If a bounding box has an IoU that is less than a set threshold, the box is labeled as a **false positive**.

4.1.3 Precision

The precision of a detection measures the ratio of correctly predicted results (i.e., True Positives) in relation to the total amount of generated detections. Equation (4.2) shows this relation

$$Precision = \frac{True\ Positives}{True\ Positives + False\ Positives}. \tag{4.2}$$

4.1.4 Recall

Recall looks at the relation between the number of correct predictions (i.e., True Positives) and the number of missed predictions (i.e., False Negatives) and is given by Equation (4.3)

$$Recall = \frac{True\ Positives}{True\ Positives + False\ Negatives}. \tag{4.3}$$

4.1.5 Mean Average Precision

The mean Average Precision is a metric used to measure how "well" an object detector manages to generate bounding box predictions over a set of classes and can be mathematically described as in Equation (4.4)

$$mAP = \frac{1}{Num\ Classes} \sum AP(class). \tag{4.4}$$

Average Precision can be calculated in multiple ways, using 11-point interpolation, all-point interpolation, integration, to mention a few. In the COCO evaluation metric, a 101-point interpolation of precision values over a set of evenly spaced recall values are used.

4.2 Inference Time

The inference time of the Tensorflow-based models is measured using the Python Time module and specifically, the **perf_counter**, which returns a value from a performance counter / the clock with the highest resolution in the system. This should lead to relatively concise measurements. For the Darknet models, the Darknet API reports the time it uses to generate a prediction per image. This is logged to a report file and is then processed through a custom Python script that extracts the numerical values. The timing numbers reported in this thesis are based on the average of inference run on a subset of the test dataset, specifically 1 sample per image with a subset consisting of 400 images.

4.3 Number of FLOPS

The number of floating-point operations is estimated using the Tensorflow Profiler module in the Tensorflow API. This gives an approximation of the number of floating-point operations that are needed to execute a network. One challenge in the estimation of the number of flops is that Tensorflow allows variable-sized input tensors to the neural networks, which makes it impossible to get a correct approximation of the number of FLOPS. As such, this method only works for networks where the input size has been fixed. However, this is generally the case for neural networks used on a hardware accelerator. Estimation on the number of FLOPs in regards to Darknet models is reported each time Darknet is called. The first thing the API does is load the model weights, and it calculates the number of FLOPs per layer from the configuration file. Estimation on the number of floating operations per model in this thesis has already been presented in the configuration Table 3.1.

4.4 Frames per second

Frames per second denote the number of frames processed by the neural network per second and can be approximated using the network's inference time. This is a valid method since the inference time is a measure of how fast the network processes an image on its input. Equation (4.5) gives this relation.

$$FPS = \frac{1}{Inference\_time} \tag{4.5}$$

For the Tensorflow models, the FPS is reported by measuring the inference time during the invocation call to the Tensorflow Lite Interpreter. Darknet reports prediction-time per processed image, which can be stored in a log. To get the estimated FPS a custom python program is used to extract inference time also reports back the FPS of the model.

4.5 Power Consumption

Power consumption is recorded using an inline multimeter, the UM32C USB tester. The UM32C is a USB-based multimeter, which allows for the recording of voltage, current, and power consumed by devices connected to the meter. It also features Bluetooth, which allows for remote read-out and control. The meter can be controlled through the use of the UM32C application, which can be run on Android and iPhone. Figure 4.3 shows the UM32C meter.

4.6 Energy per Operation

The *energy per operation* is a useful metric when evaluating the different models. Since the use case of the trained model is in a system for the detection of people

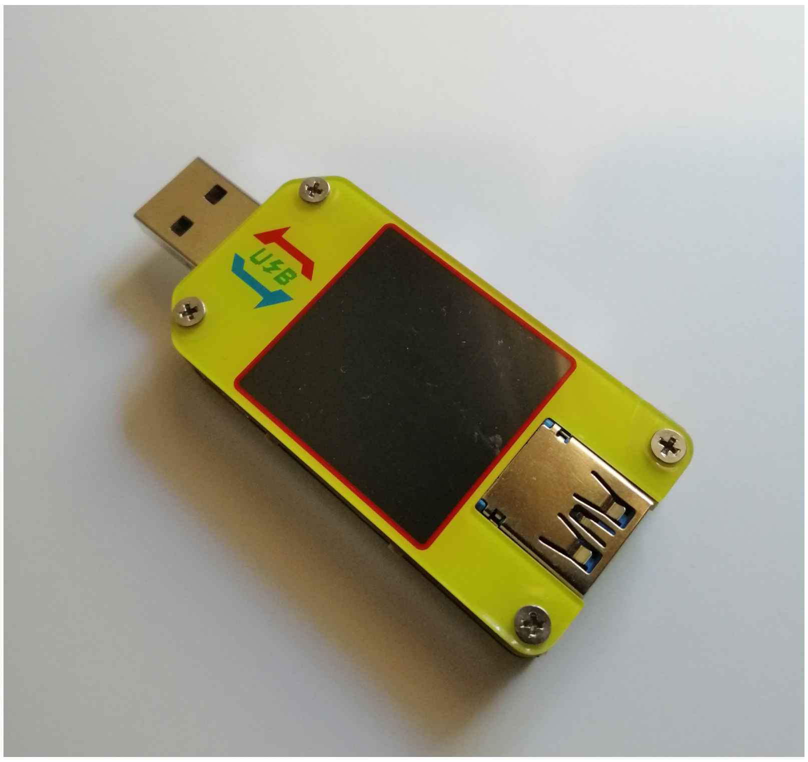


Figure 4.3: The UM32C USB Meter

and assuming the system is to be deployed on battery-powered hardware, the energy consumed is an important consideration. To estimate the energy required per operation, one needs to first calculate the energy consumed during inference. This can be done by converting the recorded power consumption into consumed energy using either the inferences time or FPS as shown in Equation (4.6).

$$Energy = Power * Inference\_time = \frac{Power}{FPS}. \tag{4.6}$$

The energy per operation is then found by taking the calculated energy and divide that by the number of FLOPS as in Equation (4.7).

$$EnergyperOperation = \frac{Energy}{FLOPS} \tag{4.7}$$

Chapter 5

Results

In this chapter, the results from each of the fine-tune models are presented. To make sure that Tensorflow and Yolo results are consistent, they are validated on the same dataset, the 4000 test images. The inference time and the number of operations required for a pass are recorded for each detector. Results from running inference on the Google Coral Edge TPU are presented both in terms of power consumption, inference time, and frames per second, and from this energy and energy per operation is estimated. Finally, the last subsection shows a detection result using each of the fine-tune models.

5.1 Transfer learning on pretrained models

5.1.1 Mobilenet V2 + SSD

The blue lines in Figure 5.1 shows the improvement in terms of classification and localization loss during fine-tuning of the model. The total loss, which consists of the classification, localization, and regularization (omitted due to minor contribution) losses drops from 7.8 down to 5.4, a 30% improvement over the base model. Loss is an interesting way to visualize that the network is learning, but it does not paint the entire picture. A model may have low loss, but still make poor detections, another factor is that loss is widely different for different architectures. This discrepancy is addressed when utilizing mean Average Precision, IoU, and Recall as these are calculated the same way on each detector. Table 5.1 shows the COCO evaluation metrics when evaluating over the training data using the last saved checkpoint. The model achieves a mAP of 0.36 for an IoU of 0.50:0.95 and 0.62 for IoU of 0.50.

			Pretrained	Custom	Pretrained	Custom
IoU	Area	Max Det	mAP		Average Recall	
0.50	all	100	0.495	0.622	-	-
0.75	all	100	0.309	0.362	-	-
0.50:0.95	all	100	0.295	0.358	0.416	0.542
0.50:0.95	small	100	0.002	0.001	0.012	0.016
0.50:0.95	medium	100	0.047	0.067	0.108	0.222
0.50:0.95	large	100	0.366	0.441	0.509	0.644
0.50:0.95	all	1	-	-	0.278	0.292
0.50:0.95	all	10	-	-	0.415	0.496

Table 5.1: Metrics generated by evaluation on the test set with the pretrained Mobilenet model and the custom fine-tuned Mobilenet model

5.1.2 Quantized Mobilenet V2 + SSD

The results which are generated by using a quantized version of Mobilenet should, in theory, produce similar results to the original architecture. Since the quantized model converts the original 32-bit float weights into unsigned 8-bit integers, a drop in accuracy is to be expected. The orange lines in Figure 5.1 displays the loss during the training of the quantized model. The loss drops from 7.3 to 5.2, which is a 29% drop, close to the 30% for the non-quantized architecture. Table 5.2 displays the evaluation of the Quantized model before and after doing fine-tuning. The Quantized model achieves a mAP of 0.35 for an IoU of 0.50:0.95 and 0.61 for IoU of 0.50, slightly less than its non-quantized counterpart.

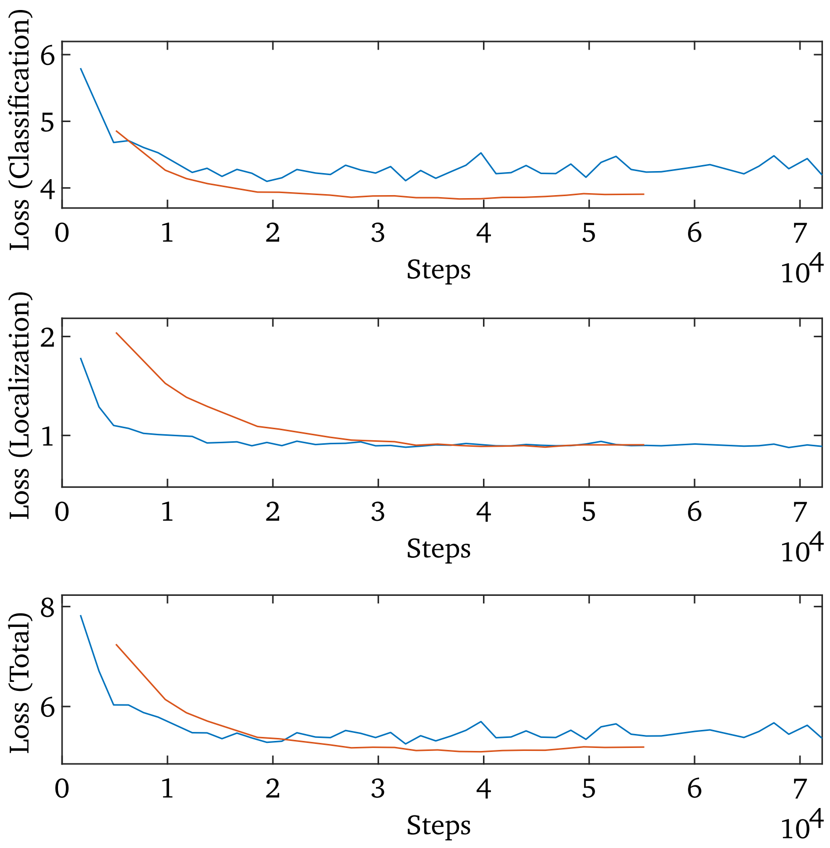


Figure 5.1: Classification, Localization and Total Loss during Training of non-quantized model in blue and quantized model in orange.

			Pretrained	Custom	Pretrained	Custom
IoU	Area	Max Det	mAP		Average Recall	
0.50	all	100	0.004	0.614	-	-
0.75	all	100	0.001	0.356	-	-
0.50:0.95	all	100	0.001	0.353	0.227	0.540
0.50:0.95	small	100	0.000	0.003	0.000	0.028
0.50:0.95	medium	100	0.000	0.066	0.002	0.227
0.50:0.95	large	100	0.002	0.438	0.292	0.640
0.50:0.95	all	1	-	-	0.000	0.291
0.50:0.95	all	10	-	-	0.002	0.493

Table 5.2: Metrics generated by evaluation on the test set with the quantized pretrained Mobilenet model and the quantized fine-tune model

5.1.3 YOLOv3

The results obtained when training YOLOv3 is interesting; overall, the trained detector is performing slightly worse than the pretrained model. The loss displayed in Figure 5.2 shows that the model converges around 500 iterations. Table 5.3 shows YOLOv3 before and after applying fine-tuning. With a mAP of 0.66, the pretrained YOLOv3 model performs the best. The custom detector performs slightly worse with a 2.1% reduction in the mAP score, going from 0.66 to 0.65.

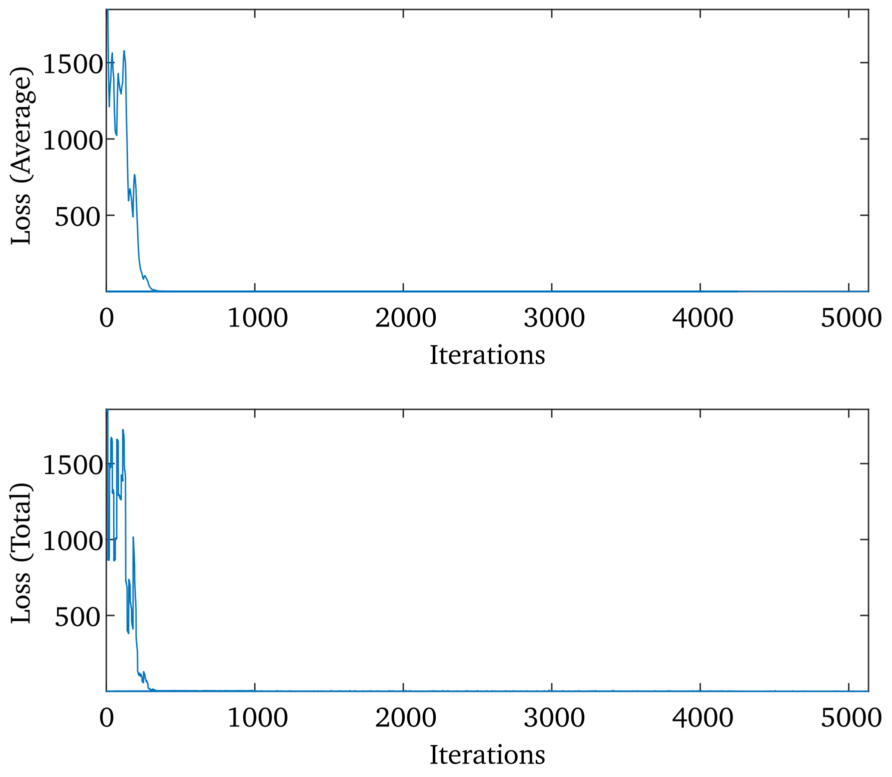


Figure 5.2: Average and Total loss during training of Yolov3

			Pretrained	Custom	Pretrained	Custom
IoU	Area	Max Det	mAP		Average Recall	
0.50	all	100	0.666	0.652	-	-
0.75	all	100	0.402	0.334	-	-
0.50:0.95	all	100	0.385	0.344	0.599	0.490
0.50:0.95	small	100	0.030	0.011	0.142	0.054
0.50:0.95	medium	100	0.212	0.096	0.375	0.232
0.50:0.95	large	100	0.444	0.414	0.624	0.573
0.50:0.95	all	1	-	-	0.305	0.283
0.50:0.95	all	10	-	-	0.536	0.468

Table 5.3: Metrics generated by evaluation on the test set with the pretrained YOLOv3 model and the fine-tune model

5.1.4 YOLOv3 Tiny

YOLOv3 Tiny is a scaled-down version of YOLOv3. It uses fewer layers, which makes the model much faster and more suitable for low-end embedded devices. This will have an impact on the overall performance of the detector. In Figure 5.3 one can observe that the loss drops from 400 downto 8 after 400 iterations. YOLOv3-Tiny is one of the detectors which has shown significant improvement when training on a custom dataset. Table 5.4 shows the performance of the model when using the weights pretrained on COCO. The model achieves a mAP of 0.249, which is relatively poor compared to the other pretrained models. However, after training on the custom dataset, the model has seen an improvement in the mAP from 0.25 to 0.51, as given in Table 5.4. This is a 104% improvement over the pretrained model.

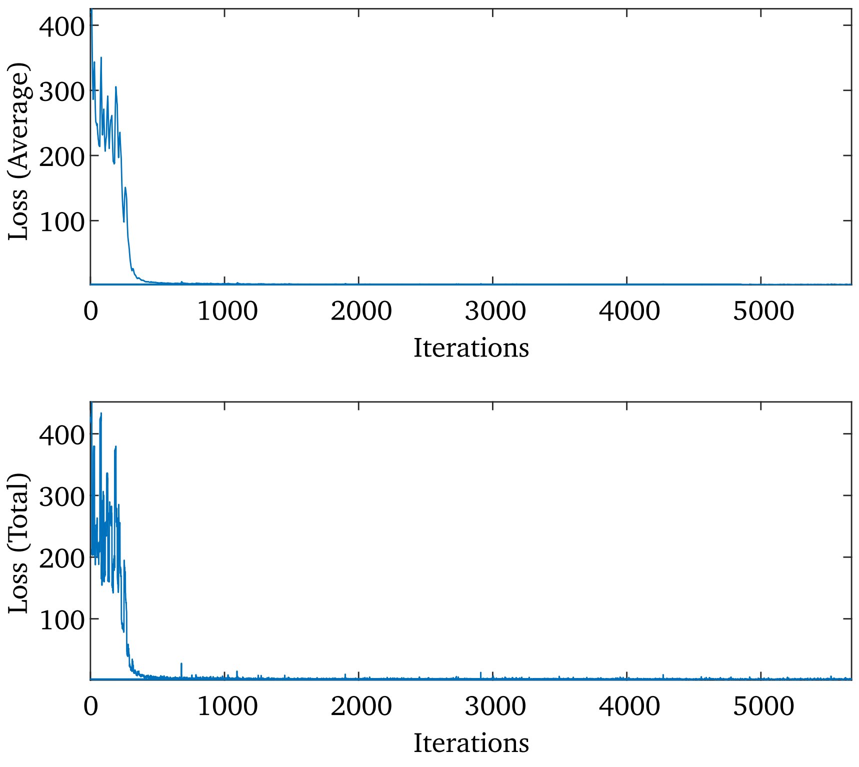


Figure 5.3: Average and Total loss during training of YOLOv3-Tiny

			Pretrained	Custom	Pretrained	Custom
IoU	Area	Max Det	mAP		Average Recall	
0.50	all	100	0.249	0.512	-	-
0.75	all	100	0.091	0.131	-	-
0.50:0.95	all	100	0.115	0.211	0.308	0.370
0.50:0.95	small	100	0.000	0.000	0.010	0.011
0.50:0.95	medium	100	0.000	0.031	0.025	0.136
0.50:0.95	large	100	0.190	0.265	0.390	0.444
0.50:0.95	all	1	-	-	0.167	0.206
0.50:0.95	all	10	-	-	0.279	0.342

Table 5.4: Metrics generated by evaluation on the test set with the pretrained YOLOv3-Tiny model and the fine-tune model

5.1.5 Combined performance figure

Figure 5.4 displays the change in the mAP using an IoU of .5 for each model, which makes it easier to compare the performance differences between models. The blue bar is the evaluation done using the pretrained models, and orange is the evaluation result after applying fine-tuning.

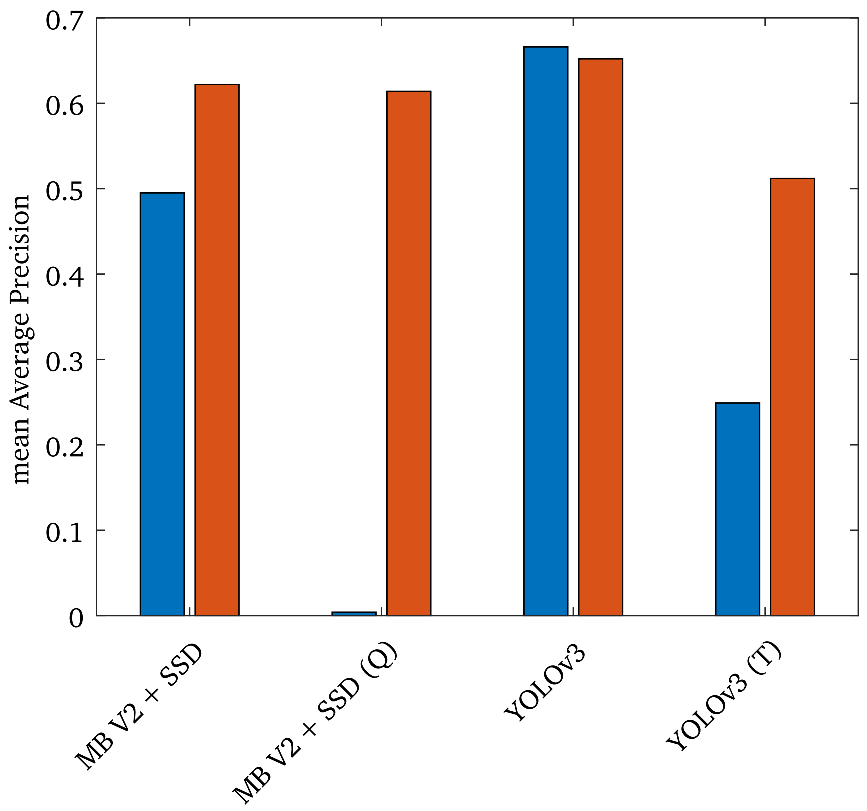


Figure 5.4: Summarized difference in mean Average Precision before and after training per model. Where MB is Mobilenet, (Q) means quantized and (T) denotes YOLOv3-Tiny

5.2 Energy per operation during Inference Run on Google Coral

Table 5.5 gives the minimum recorded inference power of each network, and the power consumed when the Coral is IDLE, i.e., not executing any inference. The difference between the power during IDLE and during model execution is also shown, giving an overview of power consumed primarily by the frameworks running each of the neural network models on the board. For each of the models, the frames per second (FPS) is also reported. The FPS is reported for each inference when running the Tensorflow Lite models. For the Darknet models, the FPS is calculated using the extracted inference time in the results log generated after running the network. Table 5.6 shows the calculated energy and energy per operation. The estimated FLOPS are also provided in the table and are equal to the values reported in the configuration Equation (2.12). Figure 5.5 displays a graph of each model's energy per operation for ease of comparison. This indicates the difference between the neural networks and the methods used for inference.

Recorded Power and Frames per Second			
	Minimum Power	w/o Idle Power	Frames per second
Coral IDLE	3.17W	-	
Mobilenet V2 Non-Quantized (CPU)	3.99 W	0.82W	1.35FPS
Mobilenet V2 Quantized (CPU)	4.41W	1.24W	3.61FPS
Mobilenet V2 Quantized (EDGETPU)	4.30W	1.13W	131.82FPS
YOLOv3 (CPU)	4.11W	0.94W	0.02FPS
YOLOv3-Tiny (CPU)	4.20W	1.03W	0.23FPS

Table 5.5: Minimum Power Consumption during Inference Run on Each Model. Both the Total recorded power and the difference between the power during inference and idle is shown

Energy per Operation				
	FLOPs	FPS	J	J/FLOPs
Mobilenet V2 Non-Quantized (CPU)	1.28GFLOPS	1.35FPS	606mJ	474pJ/FLOPS
Mobilenet V2 Quantized (CPU)	1.29GFLOPS	3.61FPS	343mJ	266pJ/FLOPS
Mobilenet V2 Quantized (EDGETPU)	1.29GFLOPS	131.82FPS	8.6mJ	7pJ/FLOPS
YOLOv3 (CPU)	38.6GFLOPS	0.02FPS	46.7J	1210pJ/FLOPS
YOLOv3-Tiny (CPU)	3.2GFLOPS	0.23FPS	4.5J	1397pJ/FLOPS

Table 5.6: Minimum Energy Per Operation during Inference Run on the Google Coral using each neural network model. Here MB denotes mobilenet and (Q) denotes Quantized. Y and Y-T is for YOLOv3 and YOLOv3-Tiny respectively

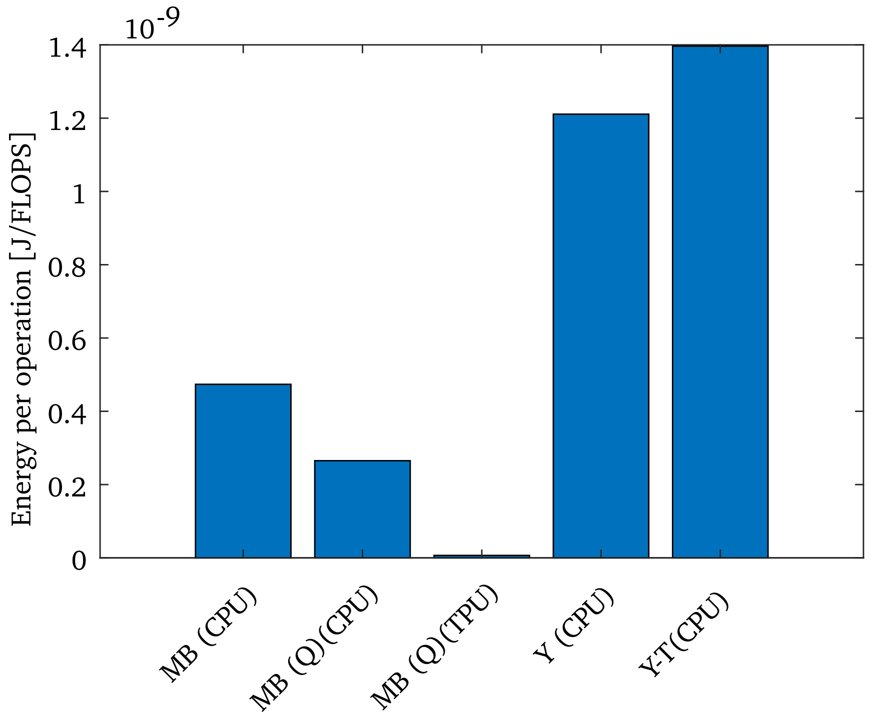


Figure 5.5: Energy per operation given in J/FLOPS for each neural network model running on the Google Coral Dev Board

5.3 Inference Time on different devices

Table 5.7 shows the average inference time recorded during inference on different hardware for each of the trained networks. Results for Desktop GPU for Tensorflow Lite models and Edge TPU for the Darknet Models are omitted due to lack of compatibility with GPU and Edge TPU for Tensorflow and Darknet respectively.

Tensorflow (Lite) Models				
Model	Desktop CPU	Desktop GPU	Dev Board CPU	Edge TPU
Mobilenet V2	100ms	-	740ms	-
Mobilenet V2 (Q)	79ms	-	276ms	8ms

Darkenet Models				
Model	Desktop CPU	Desktop GPU	Dev Board CPU	Edge TPU
YOLOv3	2949ms	13ms	51444ms	-
YOLOv3-Tiny	265ms	2m	4564ms	-

Table 5.7: Average Inference Time during run on a subset of the test dataset on Desktop CPU, Desktop GPU, Edge CPU and Edge TPU for different frameworks. The Quantized Model is denoted with (Q)

5.4 Applying Detector for Estimating People

In the following section, the detection results when applying images to each detector are displayed. This is interesting since it shows how well each detector performs and gives some indication on the feasibility of implementing a custom object detector for detecting people. Figure 5.6 shows detection results when applying the trained mobilenet model using both the frozen graph exported directly from the trained model checkpoint and the result from the converted TFLite model. Figure 5.7 shows the detection result using the Quantized Mobilenet model. Figure 5.8 and Figure 5.9 shows the detection results when applying YOLOv3 and YOLOv3-Tiny to the same image.

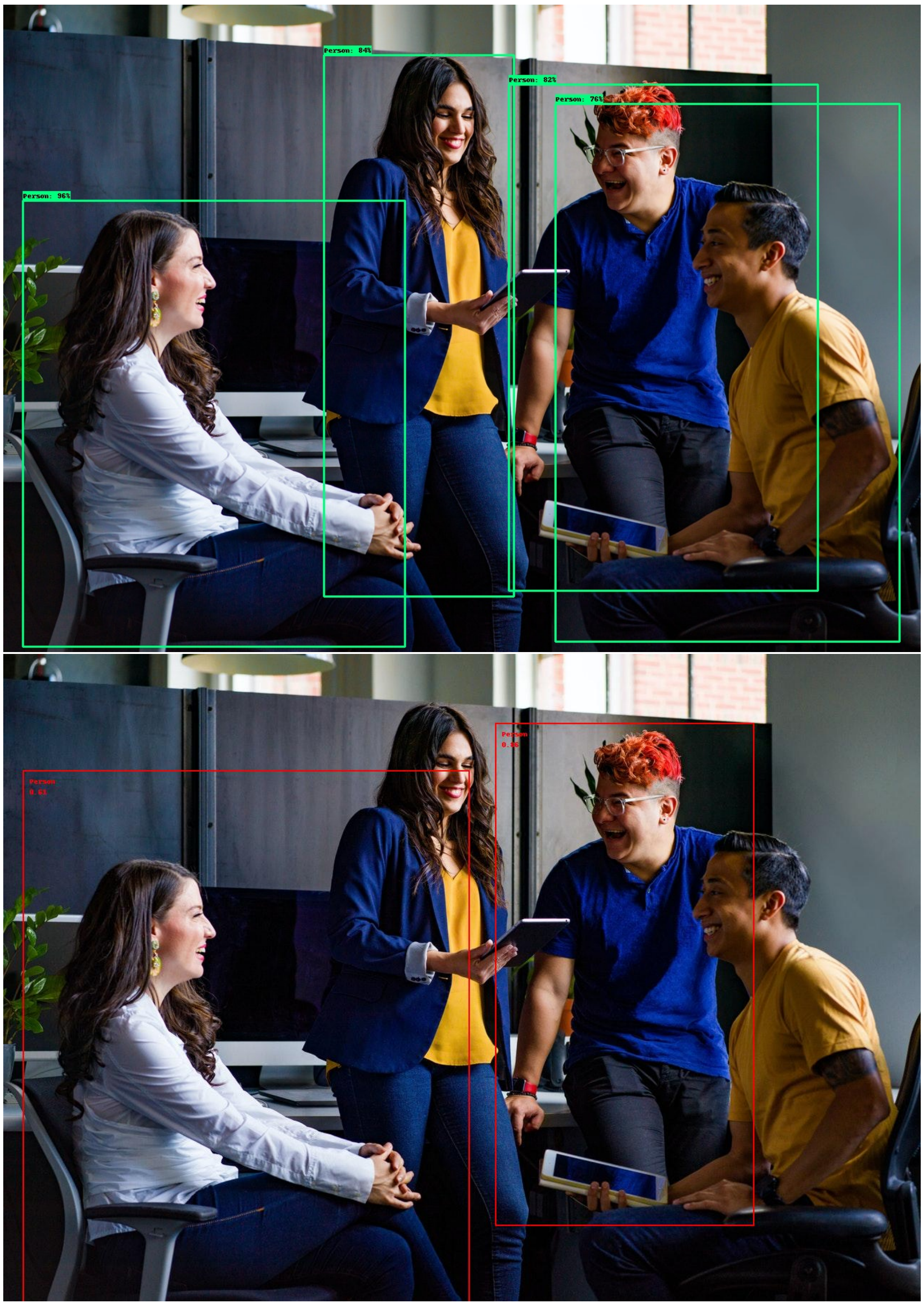


Figure 5.6: Detection result using the Non-Quantized Mobilenet model. Top: using the frozen graph exported from Object Detection API. Bottom: using the exported TFLite formatted model. Both tested with a threshold set to 0.4, Image is "Group of People Sitting Inside Room" by JopWell, Licensed as "Free to Use"

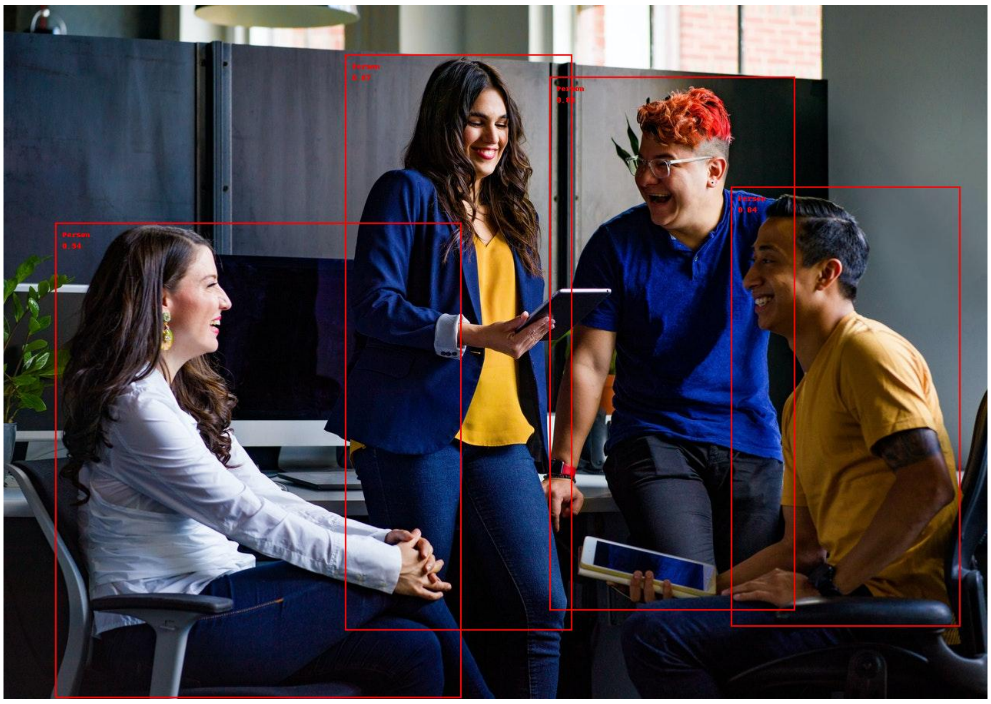


Figure 5.7: Detection result using the Quantized Mobilenet model with at threshold set to 0.4, Image is "Group of People Sitting Inside Room" by JopWell, Licensed as "Free to Use"

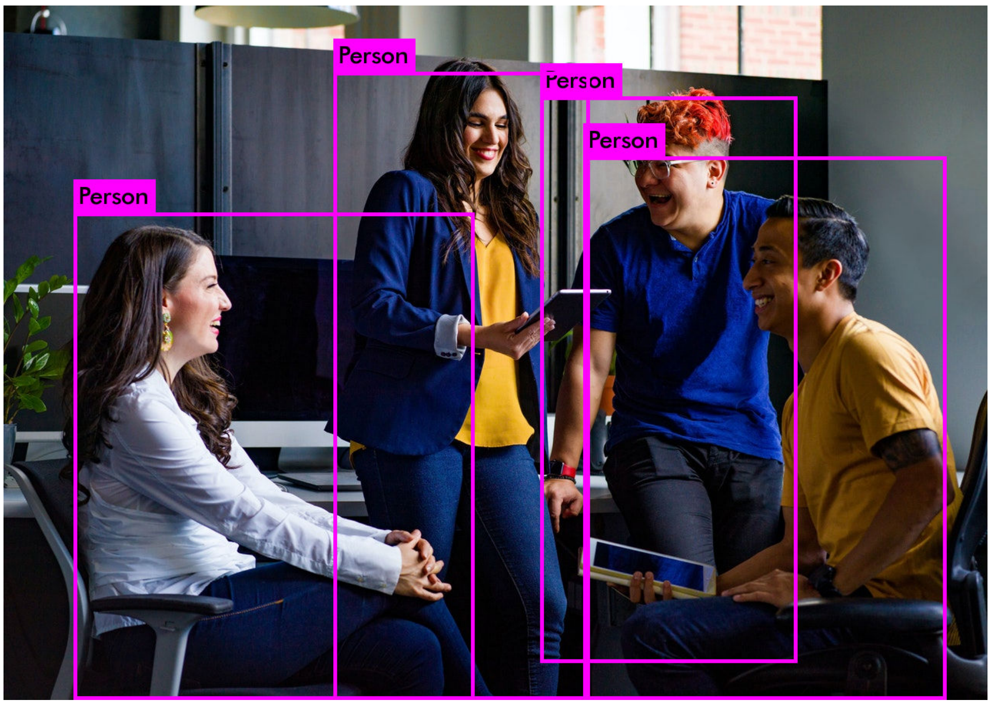


Figure 5.8: Detection result using YOLOv3 with at threshold set to 0.4, Image is "Group of People Sitting Inside Room" by JopWell, Licensed as "Free to Use"

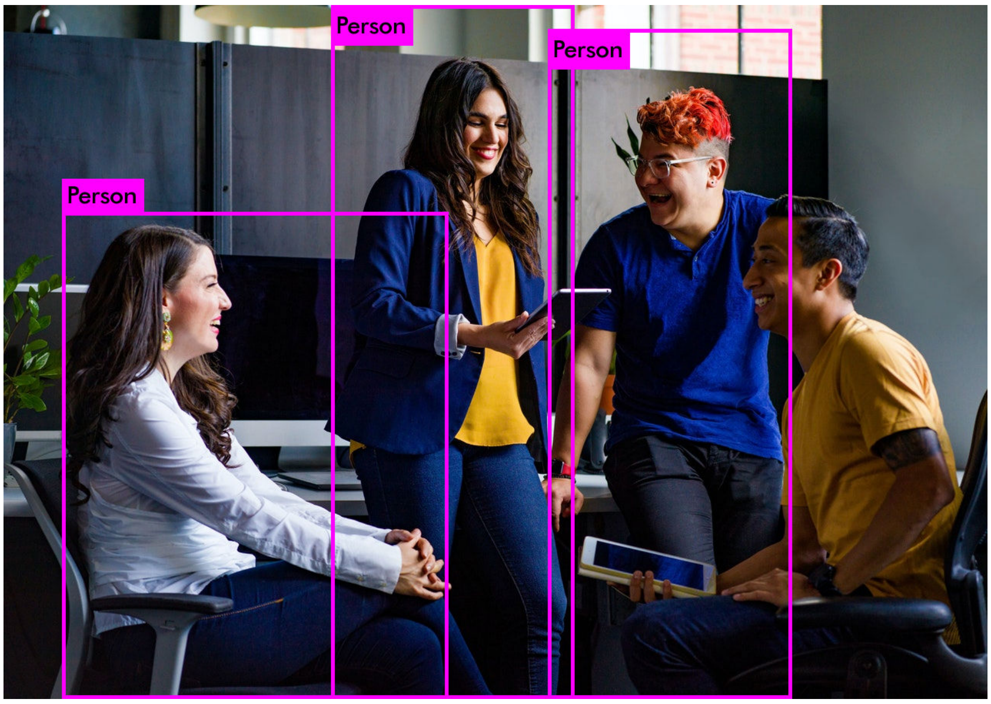


Figure 5.9: Detection result using YOLOv3-Tiny with at threshold set to 0.4, Image is "Group of People Sitting Inside Room" by JopWell, Licensed as "Free to Use"

Chapter 6

Discussion

6.1 Applying Transfer-learning to boost model performance

Utilizing transfer-learning to boost the performance of the pretrained models was found to increase the overall mean Average Precision for each detector except in the case of YOLOv3. In terms of improvement, both the YOLOv3-Tiny and the Quantized Mobilenet V2 SSD model saw major improvements compared to their pretrained performance. During the evaluation of the non-quantized model, there was found that there were some issues with the model leading to poor evaluation results on the pretrained checkpoint. The results when evaluating the non-quantized model first showed similar performance in terms of in mAP to the results of the pretrained quantized model, which are shown in Table 5.2. However, this problem was resolved after downloading a fresh copy of the non-quantized detector. After running an evaluation of the new copy, the pretrained model performed much better and produced the results, as shown in Table 5.1. This appeared to be the problem in the case of the quantized model as well. However, after downloading a fresh copy of the model and evaluating it, the results in the case of the original pretrained model and the fresh model remained the same. YOLOv3 saw a slight performance drop of 2.1% after being subjected to fine-tuning, one reason might be that the model has been trained a bit too much, leading to a slight overfit to the dataset, which would lead to a poorer mAP score. Using an earlier weight checkpoint could potentially yield a better score. However, the model performs generally very well on the test set both before and after training. This isn't that surprising given that the model is rather large compared to the other tested models, so the capability of the network to learn features is better than for the other detectors. YOLOv3-Tiny achieves the lowest mAP score at 0.51 compared to the other detectors, however in comparison to the pretrained model, it sees a 104% increase boost to its performance. During the training of the of YOLOv3-Tiny, it was found that the optimal training point was after 4000 iterations when training, at 5000 iterations the model had overfitted the training set and was incapable of generating detections on other data. From the results, one can conclude that transfer-learning and fine-tuning *can* help improve the model detection accuracy, as observed in the increased average precision and the average recall of the models.

6.2 Evaluating the neural network performance in terms of Energy per Operation

The power consumption reported in Table 5.5 where extracted from a set of 300 samples recorded by the UM32C meter. Three hundred samples were found to be the best suitable number of samples. This was decided so that one could remove the impact that the onboard fan has when it turns on, which happens when the SoM becomes hot. From this, the minimum power consumed was extracted since this should better represent the energy consumed by the neural networks, minimizing the impact of other processes running on the Coral, as well as heating fluctuations due to internal heating of components such as the cache.

To get an estimate on the power consumption, which could be contributed to the models, the power that the Coral Dev board consumes when its IDLE is subtracted from the total recorded power for each model. The Coral development board was found to consume around 3.45W while running 'IDLE.' This makes the contribution attributable to each network to be between 0.8-1.2W of power. Each of the network frames per second is also reported in Table 5.5.

The quantized Mobilenet model running on the Edge TPU far outperforms every other network, running almost at 132FPS. This is not that surprising given that the accelerator is specifically designed to run tensor operations. One can also notice that the quantized Mobilenet model running on the Dev Board CPU is 2.6 times faster than the non-quantized model, this is probably due to the differences in the data types used by each model. The quantized model utilizes 8-bit unsigned integers, and 32-bit floating-point is used in the regular model. Both of the YOLO models perform significantly worse at FPS of 0.02 for the YOLOv3 model and 0.2FPS for the smaller YOLOv3-Tiny model. This can be attributed to differences in the runtime used by Tensorflow and Darknet, as well as differences in the model architectures. YOLOv3 performs the worst, but this model is also the largest in terms of the required number of FLOPS, so it is to be expected that the model performs the worst.

Where the difference between the models comes to light is in Table 5.6, here the energy and energy per operation is calculated using the power consumption and FPS in Table 5.5. Each of the Tensorflow Lite models consumes less than 1J when running on the Google Coral Dev Board. The quantized Mobilenet model is the model that performs the best. While running on the Edge TPU, the model consumes the lowest energy at roughly 8.6mJ. Running the quantized model on the Dev boards CPU increases the energy consumption significantly to 343mJ a

40x increase over the TPU case. The non-quantized Mobilenet consumes 606mJ, which is 70x more than the quantized model on the Edge TPU and 1,7x higher than the quantized model running on CPU. YOLOv3 and YOLOv3-Tiny perform poorly on the Dev board consuming 62.0J and 5.4J, respectively. This is a massive difference in comparison to the Tensorflow models.

In terms of energy per operation, the model running on the Edge TPU performs best using only 7pJ/FLOPS. In comparison running it on the CPU increases it to 266pJ/FLOPS. For the non-quantized model, this further increases to 474pJ/FLOPS. YOLOv3 consumes 1210pJ/FLOPS while YOLOv3-Tiny consumes 1397pJ/FLOPS. The separation between the two YOLOv3 models is not that far off, which can be explained by observing that the YOLOv3-Tiny model is almost ten times smaller than the corresponding YOLOv3 model, something which is observable in the recorded FPS as well. So for YOLOv3, there seems to be a direct connection between the choice of accuracy and number of flops vs. the inference time or frame per second.

It is essential to note, that these are estimates, in terms of the number of FLOPS for the Tensorflow models, the number is generated before the models are converted to TFLite format. This is since the TFLite API does not feature any profiling options for FLOPS estimation. Since TFLite aims to be lightweight, one should expect to see a significant decrease in the number of FLOPS required to run the converted models. As such, the energy per operation would be impacted by this and would, in all likelihood, increase. However, they still serve as a general guideline in determining which of the model performs the best.

6.3 Inference Time on different devices

The results presented in Table 5.7 display the time that each model takes uses when running an inference, which is generating a prediction on the input data. The Darknet framework reports back the prediction time automatically when calling ./darknet detector test, while the Tensorflow Lite Runtime API is void of such an option. This was solved by measuring the start and stop times when the TFLite Interpreter was called.

From Table 5.7 one can clearly see that all of the models perform very well, depending on which hardware they are executed on. In the case of the Tensorflow models, the model has been converted to TFLite format and is executed using the *TFLite Runtime API*, the problem with this API is that there is no direct support for CUDA which is required to run the model efficiently on an NVIDIA GPU, as is the case in the standard version of Tensorflow. While for the Darknet API, there exists no support for running the models using the TFLite Runtime and as such no way to run the models on the TPU. One possible way to resolve this issue is to convert the YOLO models to Tensorflow models using Keras or similar libraries and then do a conversion to TFLite format. Doing this would lead to a fairer comparison between the models. However, one should expect that the YOLO models perform worse than the Mobilenet models on the basis that both YOLO models require more FLOPS to run a single inference, which implies that one should expect a higher inference time.

Testing of YOLOv3 and YOLOv3-Tiny utilizes the Darknet framework during inference run. The framework records the output of the detector in addition to the recorded inference time per image. Using a custom python script the values were extracted, and the average inference time was calculated from these values and given in Table 5.7. From Table 5.7, we can see that both of the YOLOv3 models perform very well when utilizing the Desktop GPU, both runs close to the Quantized Mobilenet model running on the Coral Dev Board's TPU. Where the YOLO models struggle is when running inference using only CPU, here the YOLO-Tiny model is almost 2-3 times slower than the corresponding Mobilenet models. There exist multiple possible reasons why there is such a significant difference in inference time between the different models, which boils down to differences innate in the frameworks, as well as variation in the required number of necessary FLOPS per model. Another caveat by using the Darknet models is that there is no direct support for running these models on the Corals TPU. There exist versions of YOLOv3 and YOLOv3-Tiny, which are implemented in Tensorflow. However, at the moment, these implementations utilize operations which are currently not supported by the TPU.

6.4 Detecting People using the trained models

Each model manages to generates detections when presented with an Image containing a fixed number of people, as could be seen in Figure 5.6, Figure 5.7, Figure 5.9. The non-quantized Mobilenet model performs worse than the quantized model. This seems to be due to the conversion to Tensorflow TFLite file format. As running the model using the frozen graph generated by Object Detections API, the model works as intended. This is shown in Figure 5.6. The quantized model performs better than the non-quantized model and works correctly after being converted to TFLite, as can be observed in Figure 5.7. In the case of YOLOv3-Tiny, the detector performs slightly worse than the Mobilenet models, managing only to detect three out of four people. Of all the fine-tuned models, the YOLOv3-Tiny is the one with the lowest mAP of 51.2, so this might explain the difference. YOLOv3 manages to detect all four individuals in the image accurately. One should note that this image represents a best-case scenario; there are few obstructions. This should make it easier for the network to detect and label the people in the image correctly. In images where there are clear obstructions or where people are positioned far away from the camera, the models are less likely to generate correct detections. An alternative example which displays this discrepancy is shown in Appendix A, these image seems more challenging than the previous example, both due to lighting conditions and the position of the individual persons in the room. In that case each model performs significantly worse than in the previous example. This warrants some more exploration, some of these problems can possibly be fixed using better or more specialized datasets, switching to methods using regression or by using a larger more complex model.

Chapter 7

Conclusion

In this thesis, two object detection frameworks have been used to perform fine-tuning on four CNN-based object detection models. The Tensorflow based models Mobilenet V2 + SSD (non-quantized and quantized) and Darknet based models Yolov3 and Yolov3-Tiny. The models have successfully been trained on a custom dataset containing a subset of images from the Open Image Database where the training set features 20k images, and the test set features 4k images. Fine-tuning the models has shown an increase in the overall performance of each model on the custom dataset, except in the case of Yolov3. Each model has been tested on different hardware alternatives such as desktop CPU and GPU, embedded CPU, and an accelerator edge TPU. The top contenders where Yolov3 and Yolov3-Tiny when running inference using a GPU both which perform very well achieving inference times of 13ms and 8ms, respectively. The Quantized Mobilenet model using the Google Coral's Edge TPU, which achieves an inference time of 8ms. In terms of power, each model was tested on the dev board's integrated CPU and TPU. It was found that in terms of consumed power, each network consumed approximately the same amount when running on the embedded hardware on the Coral. Evaluating the models in terms of the consumed energy showed a clear difference between them when they were deployed on the Google Coral. The quantized Mobilenet model performs the best when running on the Edge TPU. The model runs at 131.82 FPS consuming only 8pJ/FLOPS. On the other side of the performance scale, one can find the YOLO models where YOLOv3 is the worst contender running at 0.02 FPS and consuming 1210pJ/FLOPS on the Dev Boards CPU. In terms of applying each detector, the example image showed that every model except the YOLOv3-Tiny model managed to predict the correct number of people. The image should be seen as the best-case scenario. It contains few obstructions which could impact on the overall accuracy of the detections. In other cases the models perform poorly as was discussed previously with the given example in the appendix. Overall the Mobilenet models perform best in terms of FPS, energy efficiency and when generating predictions. The YOLOv3 models outperform every model in terms of mean Average Precision, however, when looking at the generated predictions it seems to perform slightly worse than the Mobilenets. It is also the largest model, leading to poor FPS when running on embedded hardware. Since one of the goals of Disruptive is to investigate methods of applying neural networks on resource constraint devices, that might be battery-powered, energy efficiency becomes one of the critical design considerations. In that case, the choice of the model falls on using the Quantized Mobilenet V2 + SSD model as this is currently the only model that is capable of running on Googles Edge TPU which has been shown in the case of this thesis to be the most energy-efficient method for deploying and running a neural network.

Bibliography

[1] Google Inc, *Google AIY Vision Kit Webpage*, 2018.

[2] T. Baji, “GPU: the biggest key processor for AI and parallel processing,” *Photomask Japan 2017: XXIV Symposium on Photomask and Next-Generation Lithography Mask Technology*, vol. 10454, no. July 2017, p. 1 045 406, 2017.

[3] Google LLC, “Coral Dev Board datasheet,” vol. 2, no. August, 2019.

[4] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P. L. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. Richard Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar, S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon, “In-datacenter performance analysis of a tensor processing unit,” *Proceedings - International Symposium on Computer Architecture*, vol. Part F1286, pp. 1–12, 2017.

[5] Intel Corporation, *Intel® Movidius™ Myriad™ X VPUs*.

[6] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, *Backpropagation applied to digit recognition*, 1989.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet classification with deep convolutional neural networks,” in *Advances in Neural Information Processing Systems*, vol. 2, 2012, pp. 1097–1105.

[8] D. Chahyati, M. I. Fanany, and A. M. Arymurthy, “Tracking People by Detection Using CNN Features,” *Procedia Computer Science*, vol. 124, pp. 167–172, 2017.

[9] R. Stewart, M. Andriluka, and A. Y. Ng, “End-to-End People Detection in Crowded Scenes,” *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-Decem, pp. 2325–2333, 2016.

[10] Y. Hu, H. Chang, F. Nian, Y. Wang, and T. Li, “Dense crowd counting from still images with convolutional neural networks,” *Journal of Visual Communication and Image Representation*, vol. 38, pp. 530–539, 2016.

[11] S. Y. Nikouei, Y. Chen, S. Song, R. Xu, B.-Y. Choi, and T. R. Faughnan, “Real-Time Human Detection as an Edge Service Enabled by a Lightweight CNN,” in *2018 IEEE International Conference on Edge Computing (EDGE)*, IEEE, Jul. 2018, pp. 125–129.

[12] D. B. Sam, S. Surya, and R. V. Babu, “Switching Convolutional Neural Network for Crowd Counting,” in *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, vol. 2017-Janua, IEEE, Jul. 2017, pp. 4031–4039.

[13] A. C. Ian Goodfellow Yoshua Bengio, *The Deep Learning Book*. The MIT Press, 2016.

[14] G. Bonaccorso, *Ml Algorithms*, 1st ed., 1. Packt Publishing Limited, 2018.

[15] W. S. McCulloch and W. Pitts, “A logical calculus of the ideas immanent in nervous activity,” *The Bulletin of Mathematical Biophysics*, vol. 5, no. 4, pp. 115–133, Dec. 1943.

[16] D. O. Hebb, *The Organization of Behavior; A Neuropsychological Theory*. John Wiley & Sons, Ltd, 1949.

[17] F. Rosenblatt, *The Perceptron - A Perceiving and Recognizing Automaton*, 1957.

[18] F. Rosenblatt, “Principles of Neurodynamics.,” 1961.

[19] M. L. Minsky and S. A. Papert, *Perceptrons (1988 ed)*. MIT Press, 1988.

[20] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980.

[21] K. Fukushima, “Cognitron: A self-organizing multilayered neural network,” *Biological Cybernetics*, vol. 20, no. 3-4, pp. 121–136, 1975.

[22] D. E. Rumelhart and J. L. McClelland, *Parallel Distributed Processing*. The MIT Press, 1986, pp. 318–362.

[23] C. Cortes and V. Vapnik, “Support-Vector Networks,” *Machine Learning*, vol. 20, no. 3, pp. 273–297, Sep. 1995.

[24] S. Pattanayak and S. Pattanayak, “Introduction to Deep-Learning Concepts and TensorFlow,” in *Pro Deep Learning with TensorFlow*, Apress, 2017, pp. 1–392.

[25] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Y. Fu, and A. C. Berg, “SSD: Single shot multibox detector,” in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9905 LNCS, Dec. 2016, pp. 21–37.

[26] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, vol. 2016-Decem, pp. 779–788, 2016.

[27] R. Girshick, "Fast R-CNN," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2015 Inter, pp. 1440–1448, 2015.

[28] B. Planche and E. Andres, *Hands-On Computer Vision with TensorFlow 2*. Packt Publishing, 2019, pp. 1–306.

[29] B. R. Møklegård, "(Unpublished) Specialization Project: AI on device visual occupancy detection," NTNU, Trondheim, Tech. Rep., 2019.

[30] J. Deng, W. Dong, R. Socher, L.-J. Li, Kai Li, and Li Fei-Fei, "ImageNet: A large-scale hierarchical image database," in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, vol. 20, IEEE, Jun. 2009, pp. 248–255.

[31] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman, "The Pascal Visual Object Classes (VOC) Challenge," *International Journal of Computer Vision*, vol. 88, no. 2, pp. 303–338, Jun. 2010.

[32] T.-Y. Lin, M. Maire, S. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick, "Microsoft COCO: Common Objects in Context," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, PART 5, vol. 8693 LNCS, 2014, pp. 740–755.

[33] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 580–587, 2014.

[34] S. Ren, K. He, R. Girshick, and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, 2017.

[35] J. Redmon and A. Farhadi, "YOLOv3: An Incremental Improvement," Apr. 2018.

[36] J. Redmon and A. Farhadi, "YOLO9000: Better, faster, stronger," *Proceedings - 30th IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2017*, vol. 2017-Janua, pp. 6517–6525, 2017.

[37] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L. C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Jan. 2018, pp. 4510–4520.

[38] A. Kuznetsova, H. Rom, N. Alldrin, J. Uijlings, I. Krasin, J. Pont-Tuset, S. Kamali, S. Popov, M. Malloci, A. Kolesnikov, T. Duerig, and V. Ferrari, "The Open Images Dataset V4: Unified Image Classification, Object Detection, and Visual Relationship Detection at Scale," *International Journal of Computer Vision*, 2020.

[39] A. Vittorio, *OIDv4_ToolKit*, 2018.

[40] Google LLC, "System-On-Module Datasheet," vol. 5, no. February, 2020.

Appendix A

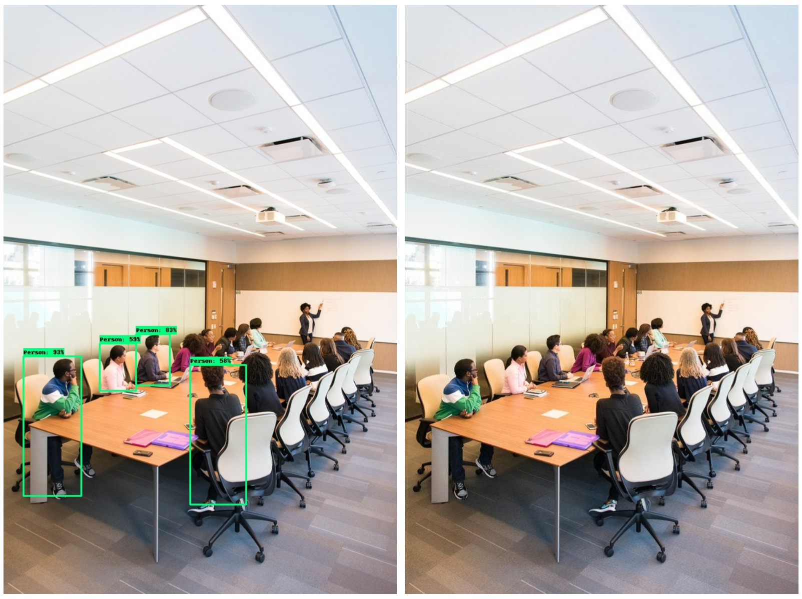
Additional Material

The code created to prepare the dataset for training using the Tensorflow Object Detection API and using Darknet is supplied on GitHub trough the following link: `https://github.com/benrammok/master_thesis`. The saved model checkpoint and tensorflow lite files in addition to the trained Darknet weights are available on Google Disk trough the following link: `https://drive.google.com/drive/folders/1dQqfVXXdPqMxZ7HTLgcNFVO83WIO3oN8?usp=sharing`

Appendix B

Detection on multiple Images

This section shows some of the detection result when applying the different detectors on a slightly more difficult image.



(a) Result from Frozen Mobilenet Model **(b)** Result from TFLite Converted Mobilenet Model

Figure B.1: Detection generated by Mobilenet before and after TFLite conversion. Notice that after converting the model to TFLite Format, no detection is generated. Image is "People Having Meeting Inside Conference Room", by Christina Morillo, Licensed as Free to Use

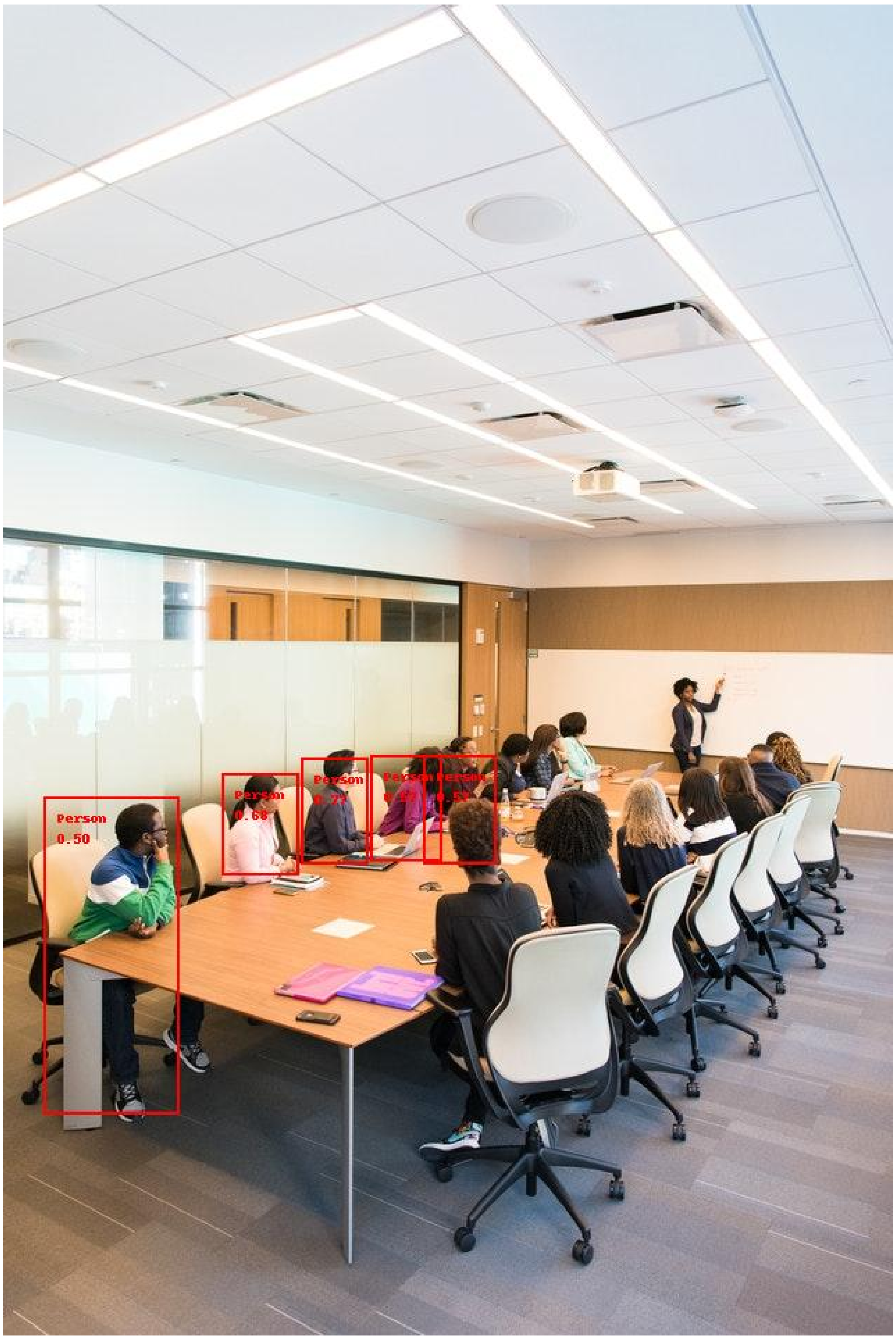
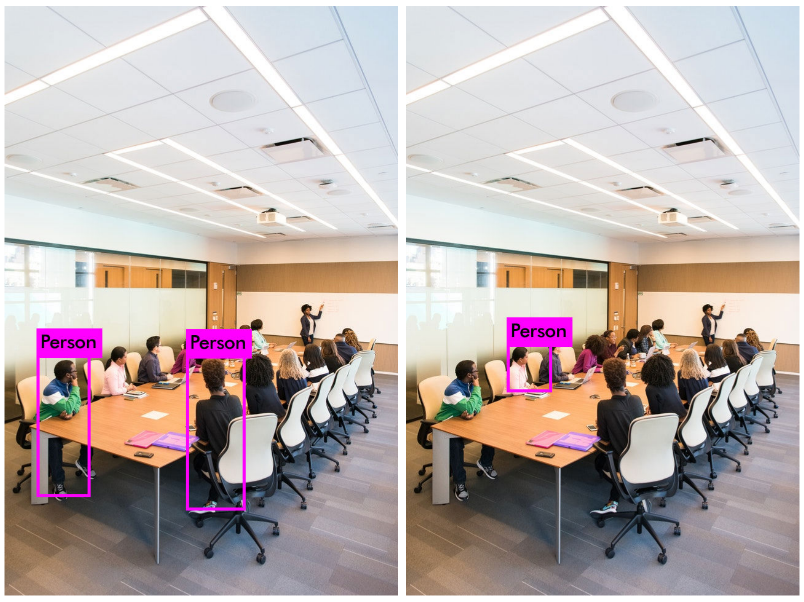


Figure B.2: Detection generated by the Quantized Mobilenet model. Image is "People Having Meeting Inside Conference Room", by Christina Morillo, Licensed as Free to Use



(a) Result from YOLOv3 **(b)** Result from YOLOv3-Tiny

Figure B.3: Detection generated by YOLOv3 and YOLOv3-Tiny. YOLOv3 manages to detect two people in the image, while YOLOv3-Tiny only manages to detect one. Image is "People Having Meeting Inside Conference Room", by Christina Morillo, Licensed as Free to Use

Appendix C

Pascal VOC Format

The following listing displays an example of how the Pascal VOC XML format is structured.

Code listing C.1: An example of the Pascal VOC XML format. This example contains a single object with name/class of Person and a bounding box position.

```
<annotation>
  <folder>Person</folder>
  <filename>filename.jpg</filename>
  <path>Path_to_image_location</path>
  <source>
    <database>Unknown</database>
  </source>
  <size>
    <width>1024</width>
    <height>683</height>
    <depth>3</depth>
  </size>
  <segmented>0</segmented>
  <object>
    <name>Person</name>
    <pose>Unspecified</pose>
    <truncated>0</truncated>
    <difficult>0</difficult>
    <bndbox>
      <xmin>78</xmin>
      <ymin>0</ymin>
      <xmax>989</xmax>
      <ymax>566</ymax>
    </bndbox>
  </object>
</annotation>
```

Appendix D

Copy of Specialization Report

The following pages is a copy of the project report created during the specialization project performed during the Fall of 2019. The report serves as some of the previous work done in relation to the content of this thesis and as such is included as additional material.

NTNU – Trondheim
Norwegian University of
Science and Technology

4590 Specialization Project

AI on device visual occupancy detection

Benjamin Ramberg Møklegård
Kandidatnr: 10002

December 19, 2019

Summary

This report is the culmination of work done in the 7,5 credit course TFE4590 Specialization Project at NTNU. The project assignment chosen for this course is "AI on device visual occupancy detection." The purpose of the assignment is to test and evaluate the Google AIY Vision Kit and to look at the feasibility of implementing the kit into a solution for detecting and give an approximation to how many people are present in a room. The AIY kit features a Myriad 2 Visual Processing Unit (VPU) for accelerating machine learning tasks. With the software supplied with the kit comes a set of pre-trained convolutional neural network models that can execute on the VPU. There are models for things like face detection, object detection, dish classification, and image classification. Both the supplied object detection model and the face detection model has been tested and verified on the AIY kit. To evaluate the performance of the kit, metrics related to the power consumption and network inference time is recorded and a test to approximate the accuracy of the models have been performed. The object detection uses on average **2.4W** at 5VDC, and has an average inference time of **96ms** during a Camera Inference run. The power consumption drops when using Image Inference; in this case, only **1.3W** is consumed and the inference time increases to **3.18s**. Face detection consumes slightly lower power at **2W** and an inference time of **76ms** for Camera Inference, while for Image Inference, the model consumes **1.3W** and has an inference time of **3.16s**. The average accuracy of the face detection model is **35.4%** and **48.1%** for the object detection model. A simple tracker is tested with the object detector to try to see if this could help improve detection in situations where there is a high level of occlusion, the tracker has been successfully tested. However, it isn't straightforward to generate metrics that verify this implementation; this needs further exploration. Testing of the object detection and tracking uses Camera Inference, so no accuracy has been recorded for this system. The system consumes **2.5W**. It executes at an average inference time of **98ms**. The AIY Vision Kit is easy to use and allows for quick deployment of machine learning-based visual processing. It manages to execute the models at a decent speed and consumes a reasonably low amount of power. It is difficult to make a definite conclusion of whether this product should be a part of a person detection system. One of the reasons is the low accuracy of the neural network models. The other is the constrained hardware of the Raspberry Pi Zero. The lack of a high-speed communication option between the Pi and the VPU limits the overall performance of the kit. Further more the limited memory on the VPU puts constraints on the network complexity, a slightly more complex network could potentially achieve higher average accuracy. For a high accuracy detection system running in real-time, the AIY Kit would not be a good choice. For a low-powered system that can accept a lower accuracy, the AIY Vision Kit could be a suitable system.

Contents

1. Introduction **1**

2. Theory **3**
2.1. Google AIY Vision Kit . . . 3
2.1.1. Intel Myriad V2 . . . 4
2.2. Convolutional Neural Networks . . . 4
2.2.1. Convolution Layer . . . 5
2.2.2. Pooling Layer . . . 5
2.2.3. Activation Layer . . . 6
2.2.4. Fully connected layers . . . 8
2.3. MobileNet + SSD . . . 8
2.3.1. MobileNet . . . 8
2.3.2. Single Shot Detection . . . 9
2.4. Object Tracking . . . 10
2.4.1. Tracking of Bounding Box Centroid . . . 10

3. Implementation **11**
3.1. Privacy and Regulations . . . 12

4. Methodology **14**
4.1. Equipment . . . 14
4.2. Metrics . . . 14

5. Results **17**

6. Discussion **21**
6.1. The AIY Vision Kit . . . 21
6.2. Object Detection and Tracking implementation . . . 22
6.3. Measurements and results . . . 22

7. Relevant Work **24**

8. Conclusion **25**
8.1. Future Work . . . 25

A. Appendix **29**
A.1. Object Detection using Camera Inference . . . 29
A.2. Object Detection using Image Inference . . . 31

A.3. Program for measuring Accuracy - Object Detection 33
A.4. Object Detection with Centroid Tracker 35
A.5. Face Detection using Camera Inference Inference 37
A.6. Face Detection using Image Inference 39
A.7. Program for measuring Accuracy - Face Detection 41
A.8. Multitracking using OpenCV and Object Detection 43

1. Introduction

The use of machine learning to do visual processing has grown in interest over the years, with the reintroduction of the convolutional neural network, there has been an incredible increase in performance on visual processing tasks. One of the primary focus of research has been on creating deeper and more advanced neural network architectures to improve accuracy. This approach has resulted in an increase in computational intensity, memory requirement, and overall energy needed to run the networks efficiently. Networks such as the VGG16 require, on average, 130 - 140 million parameters[1], it would also need a massive amount of multiplication and additions operations, in the range of 10^9 operations[2]. As such, these large and complex networks are unsuitable for applications set in an embedded setting where the hardware resources generally are limited, such as robotics or low-powered handheld devices. With the introduction of architectures like MobileNet[2], the focus has shifted to creating smaller networks that perform similar to the more computationally heavy networks at a fraction of the needed memory and computational cost. As a result of this, there has been invested time in creating hardware accelerators to accelerate and further improve the performance of these devices. This makes it possible to run networks relatively effectively on smaller low-end devices. Which in turn has spawned a new category of devices based on the concept of AI on edge. AI on edge is a relatively new concept; the purpose is to move the computations which have previously been done on remote servers or infrastructure to application-specific hardware on the edge device. This leads to a wide range of benefits, such as increased security as the computation is performed on the device, reduction in communication overhead since no connection to remote servers is needed and lower latency due to on-device computation. These devices can be applied to a wide variety of applications and problems like facial detection and recognition, object classification or scientific simulation. Google has been one of the significant contributors to these types of devices, introducing educational kits like the Google AIY Voice Kit (launched in 2017) and the Google AIY Vision Kit (launched in 2018). It also hosts a set of development kits called Google Coral, which are devices specifically designed to run Google's machine learning library, Tensorflow. The purpose of this report is to test the Google AIY Kit, evaluating the computational speed of the kit by measuring the inference time, and try to evaluate the feasibility of using the kit in a system that detects and approximates the number of people that are currently present in a room.

The AIY Vision Kit has been successfully tested and metric in relation to power consumption and inference time has been recorded for face detection, object detection and object detection with tracking. When using the supplied object detection model the kit consumes on average 2.4W (at 5VDC) of power and has an average inference speed of 96ms when using the Camera Inference. It consumes on average 1.3W and has an average inference speed of 3.18s when running Image Inference. For the Face detection model the kit consumes on average 2W and has an average inference time of 76ms for the Camera Inference, and 1.3W with average inference time of 3.16s for Image Inference. The combination of object detection and tracking uses on average 2.5W and has a average inference time of 98ms. The face detection model performs overall better than the object detection model when looking at average power consumption and inference time. However for the average accuracy of both models the object detection model have an average accuracy of 48.1%, and 35.4% for face detection when tested on a set of 100 images, containing people in different positions, rotations and lighting conditions.

The report is divided into the following sections; Theory, which will give a general description of neural networks and the components which is applied in the system. Implementation, where suggestions on how a system can be implemented by utilizing vision processing and object detection to estimate the number of people in a meeting room or office. Methodology where the equipment and the method used to test the system and to register the results are discussed. Results, which will give some details on the performance of the implemented system. Discussion where the advantages and disadvantages of the system will be discussed and explored. Relevant Work where similar solutions will be explored. And the conclusion which summaries the discoveries made in this report

2. Theory

2.1. Google AIY Vision Kit

The AIY Vision Kit is a kit developed by Google. The purpose of the kit is to make it easier to learn and apply knowledge of artificial intelligence and machine learning. The primary focus of the vision kit is implementations regarding vision-based machine learning, such as object detection, classification, facial detection, to name a few. Included in the kit is a Raspberry Pi Zero WH, a Pi Camera V2 module, a custom extension board called the Vision Bonnet, privacy LED, piezo buzzer, push-button, and the cardboard which forms the housing[3]. Supplied alongside the kit is a custom version of Raspbian, a Linux based OS. The OS contains libraries for interfacing with the components of the kit like the Vision Bonnet and the Pi Camera module. It also has pre-trained models which can be deployed to the Vision Bonnet and used for tasks such as object detection and classification. The Vision Bonnet is a custom extension board for the Raspberry Pi Zero. The board features an Intel Myriad V2 VPU, connections to the Pi Camera, buttons, LED's and piezo.

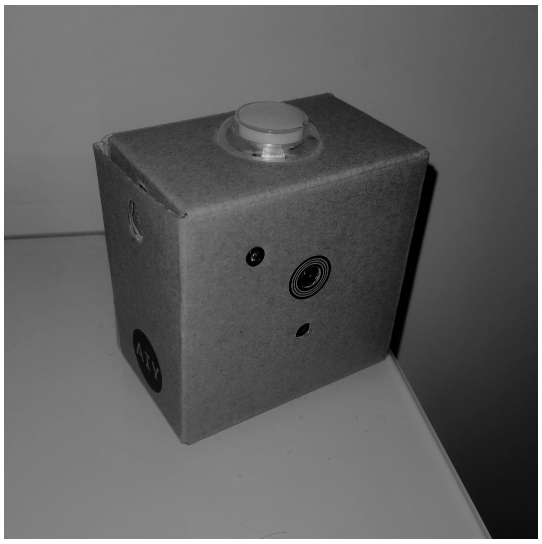


Figure 2.1.: Fully Assembled Google AIY Vision Kit

2.1.1. Intel Myriad V2

Intel Myriad V2 is a Visual Processing Unit (VPU). It's designed for low powered image and vision processing[4]. The VPU uses 12 VLIW 128-bit vector processors called "shave" processors to accelerate machine vision. The VPU features a broad set of I/O and supports I2C, I2S, SPI, USB3. It also has a 12 lane MIPI interface with a speed of 1.5Gbps per lane. The MIPI lanes can be either configured for CSI-2 or DSI, which are interface standards for communicating with a camera or a display, respectively. For the storage of machine learning models, the VPU features an on-board storage of 2MB[4].

2.2. Convolutional Neural Networks

Convolutional Neural Networks (CNN) forms the backbone of most modern object detectors as well as other image processing and classification systems and is the basis for the models found on the AIY kit. This section gives a brief overview of how a CNN is structured, and in the following section, a general description is given on how the pre-trained object detection model on the AIY kit is structured and how it functions.

The structure of a Convolutional Neural Network consists of an input layer, a set of hidden layers, and an output layer. Each layer consist of a set of nodes often referred to as either neurons or nodes, each node can apply a wide variety of functions. When used in Convolutional Neural Networks, each node in a layer perform the same operation. The layers which are common in a Convolutional Neural Network are convolutional, pooling layers, activation layers, or fully connected layers. The input and output of each layer are usually referred to as a feature map[2].

Generally, a neural network is divided into two different types, either shallow neural networks or deep neural networks, the difference is simply the number of hidden layers used. A shallow network usually only has one hidden layer, where a deep neural network has more than one layer[5]. In the context of the convolutional neural network used for image processing, these are almost exclusively configured as deep neural networks.
The configuration of convolutional neural networks can be achieved in two different ways. The first way works by modifying the parameters of the nodes in each layer. These parameters are only tweaked and changed during the training of the network. The way these parameters change is by a process known as back-propagation. Back-propagation is an algorithm performed by software or libraries one uses when designing a convolutional neural network model and is not something the user needs to configure themselves. The other way of configuring the network is through the network's hyperparameters. Hyperparameters are parameters that are set by the network designer. These determine the structure of the network, and so these are always set before the training of the model. Examples of parameters can be the number of filters per convolutional layer, the size of the input layer, size of the output layer, scaling of each subsequent layer[5].

2.2.1. Convolution Layer

The convolutional layer performs a discrete convolution between an input image or feature map and convolves this with a filtering kernel, which consists of a set of weights. A layer can consist of multiple kernels where each kernel produces a distinct feature map. Each of these feature maps may extract specific features from the input image. The extracted features in the feature map, depends on the configuration of the kernel of each filter. The process of convolving the input image or feature map uses a sliding window technique, which works by moving the kernel over the image with a configurable stride. A stride of 1 implies that the kernel moves one pixel at a time, while a stride of K would move the kernel K pixels at a time. Each pixel in the output feature map is a sum of element-wise multiplication between the pixel values in the currently selected position in the input feature map with the weights in the convolutional kernel. Equation 2.1 defines this process mathematically and is a compounded version based on the same formula in [6] and is similar to the one given in [2].

$$o\_feature(x,y) = \sum_{i,j=0}^{N} i\_feature(x+i, y+j) * weights(i,j) \tag{2.1}$$

Where $o\_feature$ is the output feature map, $i\_feature$ is the input feature map, and $weights$ is the kernel weights.
When doing convolution on a multilayered feature map or image, the feature map produced on the output of the filter is a linear combination of the feature maps[2].
As a result of the convolution with the kernel, the output feature map experiences a reduction in resolution. The output resolution of the feature map largely depends on the size and the stride of the kernel. One can also add padding to the input data, which modifies the size of the output of the convolution. There exist two different types of padding; the "valid" and the "same" padding. With "valid" type padding, there are no additional values applied to the edges of the input data, the output of the convolution with "valid" padding is a feature map with a reduced resolution in respect to the original input. The "same" type padding adds extra data to the edge of the input data, which results in that the feature map on the output of the convolution having the same size as the original map on the input[7].
An example of the convolution process is given in Figure 2.2, here the kernel is of size 3x3 and is convolved over a 4x4 image, for the first pixel in the output feature map, each weight is multiplied by the pixel in the corresponding position in the currently selected region marked with the colors blue and green. The resultant feature map is of size 2x2.

2.2.2. Pooling Layer

To reduce the number of calculations needed, when applying convolution to larger feature maps, pooling layers are added to downscale the resultant feature map produced by each convolutional layer. The pooling operation works by dividing the input feature map into regions, where each region corresponds to a pixel in the downscaled feature map. The value of the pixel on the output depends on the configuration of the pooling layer[5].

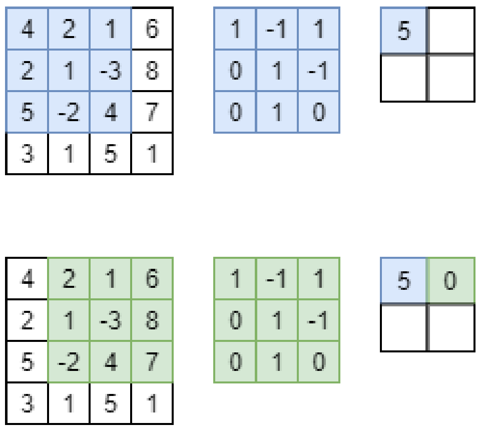


Figure 2.2.: Example of Convolution on a 4x4 Input Feature Map and a 3x3 filtering kernel

The operation can also be viewed and implemented as a sliding window moving over the input, similarly to how convolution is applied[8]. A pooling layers is usually either configured as a max-pooling layer or as an average-pooling layer, the difference between the two types is simply the value they produce when moved over a section of the data. In a max-pooling layer, for each selection of pixels in the input data, it returns the pixel with the highest value. While for the average-pooling layer, the average of all pixel values is calculated and returned. Figure 2.3 shows an example of the two pooling types, here an input feature map of size 4x4 gets reduced to a new feature map of size 2x2 by using a 2x2 selection region. This example uses a sliding window to select a region; the window has a stride of 2. The size of the selected region and the stride determine the output size of the new feature map. For convolutional neural networks, max-pooling is the type of pooling-layer, which is most often used[9, 8]. One of the other significant advantages of doing pooling is that it manages to maintain the extracted features in previous layers and it also allows the model to be more resistant to invariance in images such as rotation and translation[7].

2.2.3. Activation Layer

The purpose of the activation layer is to determine which nodes in the network ("neurons") should activate and which should remain inactive. For a convolutional layer, each output pixel from the convolution operation is fed trough an activation layer. Here the value is modified by an activation function. The resultant output of this operation gets passed to the pooling layer.

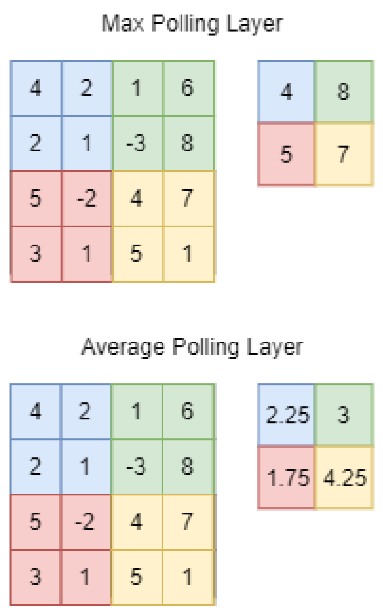


Figure 2.3.: Example of pooling Layer Types

There exist many different types of activation functions, and the choice of activation function affects both the speed and the learning ability of the networks. The choice of function also depends on what purpose the system should serve. For modern convolutional neural networks the two most commonly used activation functions are the Sigmoid function and the ReLU function. Another commonly used activation function is the Softmax classification function, this function is often used at the end of the network to convert the values of the last layer to a set of probabilities for the object classes the network is trained to recognize[5].

Sigmoid

The Sigmoid function is an activation function characterized by an S-like shaped curve. The output of the function is bounded between 0 and 1, or -1 and 1, depending on the implementation. There exist multiple different mathematical implementations of the function, but one of the commonly used is the logistic function, which is defined in equation 2.2.

$$f(x) = \frac{1}{1+e^{-x}} \tag{2.2}$$

The challenge in using the Sigmoid based functions is that they suffer from saturation. Sigmoid functions saturate when the input grows larger or smaller, which inevitably leads to a decrease in the network's ability to learn. This problem is known as the 'vanishing gradient' problem, and the result of this problem is that the network completely stops learning[5].

ReLU

Another type of activation function is the ReLU or Rectified Linear Unit, the function is a non-linear function defined by equation 2.3

$$f(x) = max\{0, x\} \tag{2.3}$$

The function can't saturate as it is bounded by 0 when x is less or equal to zero and is linear when x is greater than zero. So it does not suffer from the 'vanishing gradient' problem as the Sigmoid does. ReLU is the most common activation function in modern convolutional neural networks. The reason is that it significantly improves network training speed over functions that face the problem of saturation[10, 5]. In some networks the ReLU functions are supplied with an upper bound like in the newer version of MobileNet, MobileNet V2[11]

2.2.4. Fully connected layers

A fully connected layer differs from a convolutional and pooling layer. Each node in a fully connected layer connects to every node in the subsequent layer. The purpose of the fully connected layer is to combine the output of the last convolutional layer and interpret meaning of the extracted features, this is often done by applying soft-max as the activation function[12]. In some networks, there is no need for a fully connected layer. This is mostly dependent on the purpose and configuration of a network[2]. But for cases such as image classification, it is usual to see one or more fully connected layers at the end of the convolutional network, acting as feature classifier[5, 1].

2.3. MobileNet + SSD

The model used for object detection present on the Vision kit is a re-trained model of the MobileNet V1 + SSD architecture developed by Google. It allows for the detection and generation of bounding boxes outlining objects for which the model has been trained to detect. The version on the Vision kit is restricted to detecting people, cats, and dogs. MobileNet V1 + SSD is an architecture built on two types of networks. The MobileNet V1 architecture which functions as a feature extractor and the Single Shot Detection (SSD) network, which generates bounding boxes and object-classification probabilities for the features that MobileNet has extracted. The following subsections give a general description of how these two networks work and operate.

2.3.1. MobileNet

MobileNet is a convolutional neural network model created by Google for use in embedded systems. The model utilizes a process known as Depthwise Separable Convolutions[2].

In a regular convolution, an input image with multiple channels gets filtered and combined into a feature map of one layer. In Depthwise Separable Convolutions, this process is split up into two different operations a depthwise convolution and a 1x1 pointwise convolution. The depthwise convolution works similarly to a standard convolution, however where a regular convolution would do both filtering and combination of the input channels, the depthwise convolution applies a filter for each input channel, producing a filtered output for each channel. Each of these outputs is then combined to form the final feature map. The 1x1 pointwise convolution is applied to the feature maps; the layer computes a linear combination of the input layers, which results in the formation of a feature map with one channel after the operation[2].
The advantage of depthwise separable convolutions over the standard convolution is that the separation of the two operations, filtering, and combination, results in a reduction of the necessary computations, in comparison to a standard convolution. The network also removes the pooling layers in favor of strided depthwise convolutions. This helps by reducing the number of layers in the model, which in turn makes the model smaller[2].
MobileNet introduced two new hyperparameters for configuring this new network architecture. These are the width multiplier (often referred to as depth multiplier) and a resolution multiplier. These hyperparameters give the designer the ability to customize the size and performance of the network.
The width multiplier aims to reduce the size of the computational model by reducing the number of channels/filters in each subsequent layer making the model smaller and faster. This is done by scaling each layer by a constant α. It manages to reduce the number of parameters in a network by the square of α.[2]
The resolution multiplier works similarly as the width multiplier. Instead of targeting the number of channels, it reduces the resolution of the input layer and the hidden layers. This leads to a quadratic reduction in parameters similar to the width multiplier[2].
The trade-off from changing these parameters comes in a reduction in accuracy, however depending on the purpose of the network, this might be acceptable, and as such effectively applying these hyperparameters to a network architecture can lead to a drastic reduction in computational intensity, saving power and computation time.

2.3.2. Single Shot Detection

Single Shot Detection is a convolutional neural network that produces bounding-boxes, as well as a score that indicates the likelihood that an object is present and the type of object detected. The network is structured in such a way that each convolutional layer is smaller than the previous one. This is done to allow for the detection of objects at different sizes. Since each convolutional filter decreases the resolution of the feature map, the higher resolution feature maps allow for the detection of smaller objects while the lower resolution feature maps can discern larger objects[13]. Each feature layer in the SSD, as well as the output of the base network, produces a set of predicted detections. These predictions specify the shape offset in relation to a set of default bounding boxes. The default bounding boxes are specified for each feature map, it uses the combination of shape offsets and default boxes to produce a set of bounding boxes in relation to each detection[13]. The end result is fed to a non-maximum suppression layer, which creates the finalized result[13].

2.4. Object Tracking

Object tracking is the process of correlating the position of moving objects in a set of frames, like a video. Tracking can be implemented in multiple different ways; one of the more straightforward ways of creating a tracker is to implement a tracker which correlates the change in the center position of a set of supplied bounding boxes.

2.4.1. Tracking of Bounding Box Centroid

A centroid based tracker works by looking at the distances between the center of one or more bounding boxes and tries to correlate this position to the position of a bounding box which has previously been detected[14]. Assuming that the motion of the detected object between subsequent detections is low, one can correlate the old position p_1 and the new centroid p_2 by looking at the euclidean distance between the points the distance is calculated using the Pythagorean theorem 2.4.

$$d(p_1, p_2) = \sqrt{(p_{2,x} - p_{1,x})^2 + (p_{2,y} - p_{1,y})^2} \tag{2.4}$$

Figure 2.4 shows two points marked in red which relates to centroid of the bounding box and green which is the centroid of the bounding boxes which are detected in the next frame. A simple assumption is that the shortest distance between two points indicate that the two points correlate. To limit the correlation between points which are far apart, one can set a limit on the maximum distance that is allowed between two points. If there exists a point that does not correlate to any previous position, this gets registered as the position of a new detection. Since a person can move out of the camera frame, there needs to be a way for removing 'dead' detections, this can be done by assigning a limit on how long the detection is allowed to stay registered. So each detection is assigned a value corresponding to how many frames it has been since the last successful correlation. If no new detection which can correlate the current position to an older detection has appeared in a fixed set of frames, the old detection is removed from the tracker [14].

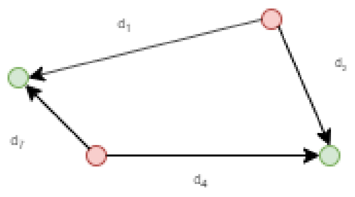


Figure 2.4.: Centroid Tracking

3. Implementation

This section takes a look at possible implementations using the Google AIY Vision Kit as a part of a person counting system. Three possible ways of implementing this and some consideration of their strengths and weaknesses for each possibility is discussed. There are potentially many ways of implementing a system for counting people. One method can be based on the detection of faces and count the number of detected occurrences. However, this could suffer from poor detection due to obstructions blocking the view of the area. Also, the position and rotation of a person in the image can limit the face detection ability to detect a face. A better way would be to utilize the built-in object detection model to detect people since the model supplied alongside the Vision Kit is trained to detect cats, dogs, and people. The detection of a person should be trivial. This model should work better than a face detector since it does not matter if the person is facing the camera or not, it infers meaning by analysing shapes of the objects detected in the image and classify thereafter. This system would also suffer due to obstructions in the area of detection. It can also suffer somewhat from the position of the object in the room.
A better way to utilize these systems would be by including a tracker to predict where the person might appear next. So if a person disappears behind an obstruction, the system can approximate and correlate where the person might appear. This can potentially give a better approximation of how many people have been detected. Since the tracker stores the last position of currently detected objects, the approximation might be closer to the actual number of people who are present in the frame.

Three configurations seem feasible, one involving the face detection model, one which only uses the object detection model and a configuration that uses the object detection model and a simple centroid based tracker. Face detection and object detection is done by using the supplied models on the Vision Kit. As previously explained, the object detection model is based on the MobileNet V1 + SSD model and differentiates between detections of a person, a cat, or a dog. The following is the configuration of the model on the kit, the input supports an image of size 256x256, and the network has a depth-multiplier of 0.125, this is one of the larger networks that is allowed to run on the Vision kit. The model returns a bounding box for each detected object, in addition to class probability. A centroid tracker module is used to implement the tracking functionality. The module in the program is a pre-created tracker created by Rosebrock [14]. The centroid of each bounding box is calculated by the generated box from the object detection model. The calculated centroid gets passed to the centroid tracker. This is performed on each result that is returned from the VPU. The face detection and the object detection programs use the corresponding examples supplied on the Vision Kit as the base. These programs have been rewritten and extended to generate a CSV file with the inference timing results. For the system using the object detection and the centroid based tracker, the system extends the object detection program to include the tracker. It counts the number of people detected by counting the number of detections that have been registered by the tracker. The object detection and centroid tracking program has been tested, and the program code is supplied in Section A.4 in the Appendix. It isn't very easy to give results to corroborate the efficiency of the tracker, so the primary focus remains on the exploration of the object detection model. An attempt to do tracking using OpenCV has been tested. However, due to the limited hardware on the Raspberry Pi Zero, issues with the multitracker implementation in OpenCV, and limited time of the project, this approach is not explored further in this report. The test program has been supplied in Section A.8 in the Appendix. The programs for Face- and Object detection using Camera and Image Inference can be found in Section A.1, A.2, A.5, and Section A.6 in the Appendix.

3.1. Privacy and Regulations

Disclaimer: I am not a lawyer, this section is not legal advice, it summarises relevant information which is of interest when dealing with camera systems.

Since the purpose of the system is to survey an area and approximate the number of people present. It is interesting to take a look at how laws and regulations regarding the collection and use of data would apply to such a system.

The system applies a camera to capture raw video from an area and then processes this information to make detections. Relevant laws and regulations which governs the use of camera systems and privacy considerations are *"lov om behandling av personopplysninger"* which regulates the collection and storage of personal information, and *"forskrift om kameraovervåkning i virksomhet"* which regulates the use of surveillance systems in businesses, as well as *"lov om arbeidsmiljø, arbeidstid og stillingsvern mv. (arbeidsmiljøloven)"*, which regulates the use of forced control measures in businesses (Chapter 9).
Since the current implementation of the system only reports on the number of people it has detected and stores no video which could be used to identify individuals, paragraphs regarding retention and deletion of stored surveillance material does not apply in this case. § 4. in *"forskrift om kameraovervåkning i virksomhet"* states that one must give notice that a surveillance system is in use and who is responsible for the system, this is a relevant paragraph and would come into consideration if the system is applied in a business or similar manner, where the goal is to survey. *"lov om arbeidsmiljø, arbeidstid og stillingsvern mv. (arbeidsmiljøloven)"* Chapter 9, clearly states that the use of a surveillance system can only be deployed if the employer of the business has a legitimate reason to deploy it, examples of reasons could be for use in security purposes or to restrict access. Given that the purpose of the system is to detect the number of people

present in an area and report this back as aggregated data, it should have a marginal impact on the privacy of people being detected by it, as no other relevant data is stored like video or images. As this is the case, the law and regulations should most likely not pose significant limitations when deploying a system like this. The only potential problem could be in the reasoning of the deployment of the system. If one cannot give a good enough reason for why the system is needed, this could constitute an unnecessary intrusion into the employee's privacy. This problem should be addressed by contacting the governing agency to make sure that the laws and regulations are followed. Other than this limitation, the only relevant action an employer would be required to take would be to inform and give notice that the system has been put into use.

4. Methodology

This section presents the equipment which are used during testing of the different models, and gives a brief overview of the metrics which are used to present the results.

4.1. Equipment

The object and face detection models, as well as the implementation using object detection and tracking is tested using the Google AIY Vision kit V2, which features a Raspberry Pi Zero W with a 1GHz Single Core CPU, 512MB of RAM, the Vision Bonnet with the Myriad 2 VPU. Python 3.5 is used for the creation of program scripts, and the supplied pre-trained convolutional neural network models and Python modules found on the kit are used to evaluate the kit. Multiple tests have been devised to test the impact of inference time when applying the different models to the VPU on the Vision Bonnet.
Power measurement is done by using the USB Charger Doctor from Adafruit. This is an inline current and voltmeter which can be connected to USB ports. The meter has a resolution of 10mV and 10mA, and it supports an input dc voltage of 3.5V-7V and a current draw in the range of 0-3A. The USB Charger Doctor does not feature digital readout, so the results are obtained by doing multiple tests with the different models (face and object), observing and taking note of the measured voltage and current. The Charger Doctor is shown in Figure 4.1

4.2. Metrics

The metrics used in this report to quantify the models on the AIY kit are the inference time, i.e., the time it takes for an image to be sent to the neural network to a result is produced on the output. Inference time is recorded by calling python's built-in time function, before and after the inference has been started. The recorded inference times are stored in a CSV file. Average inference time is calculated using normal averaging, as shown in equation 4.1.

$$A = \frac{1}{N}\sum_{i=0}^{N} x_i \tag{4.1}$$

Power is the second important consideration. The AIY Vision Kit is based around a Raspberry Pi Zero, which is designed to consume low-power. Measuring the power of the kit in the different scenarios can lead to a better understanding of why it can be favorable to utilize a Visual Processing Unit to accelerate machine learning, as well as

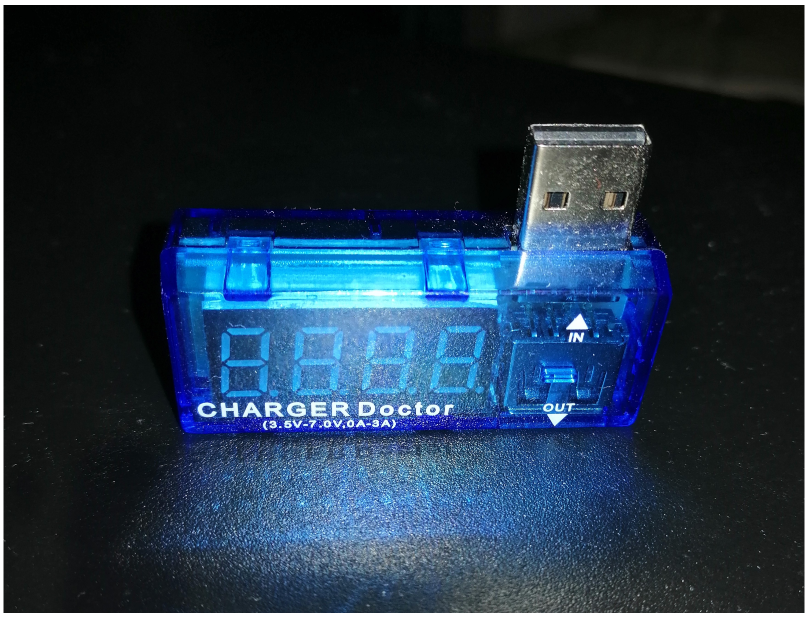


Figure 4.1.: Adafruit Charger Doctor, an Inline Multimeter

to understand what factors that lead to a reduction in power when writing programs for the system. Since the current drawn by the processor can fluctuate during processing, multiple measurements are taken and averaged during each test of the different models (object, face) and configurations (tracking) to get an approximation on the power consumption. A metric which often is used when classifying neural network is the mean average precision (mAP). The mean average precision is a metric used to classify how well the object detection models generate class probability and bounding boxes, and how well these predictions correspond to pre-specified bounding boxes (also referred to as ground-truth). The metric is used during the training and validation of the model and is usually validated on a large scale data set such as ImageNet and COCO[2, 1].
Since the Google AIY kit features a custom API and inference engine. It's a bit more difficult to evaluate the performance of object detection based on this method.
In this report, a more straightforward approach to approximate the accuracy of both the face and object detection models is used. The accuracy metric for the implemented system measures the accuracy by relating the number of correctly detected people in an image versus the total number of people present in the image, which has been counted beforehand. A custom python program has been written to do the accuracy testing. The program works by loading a set of images, where each image has an approximate width of 1280 and height of 720. The name of each image has been structured in the following format: **t,n.jpg**. Here t is the number of people in the frame, and n is a number for separating images that have an equal amount of people present in them.
The accuracy is calculated by using the total number of people in the image (which is part of the image name), with the number of people detected by the detection model as

given in equation 4.2.

$$Accuracy = \frac{Predicted\ Detections}{Total\ Number\ of\ People} \tag{4.2}$$

To produce a decent approximate result, a set of 100 images has been tested. The approximation can be even more accurate if the number of images that are tested are increased used. However, due to the limitation on transfer speed due to the use of SPI as the communication channel between the Raspberry Pi Zero and the Vision Bonnet, the number of images was constrained to a lower number to save time. The test programs for determining accuracy for the Face- and Object Detection Models can be found in Section A.7 and Section A.3 in the Appendix, respectively.

5. Results

This section summarises the results obtained from doing tests on the AIY Vision Kit. The tests for inference times are separated into two different figures. This is due to the massive difference in inference time for Camera Inference versus Image Inference.

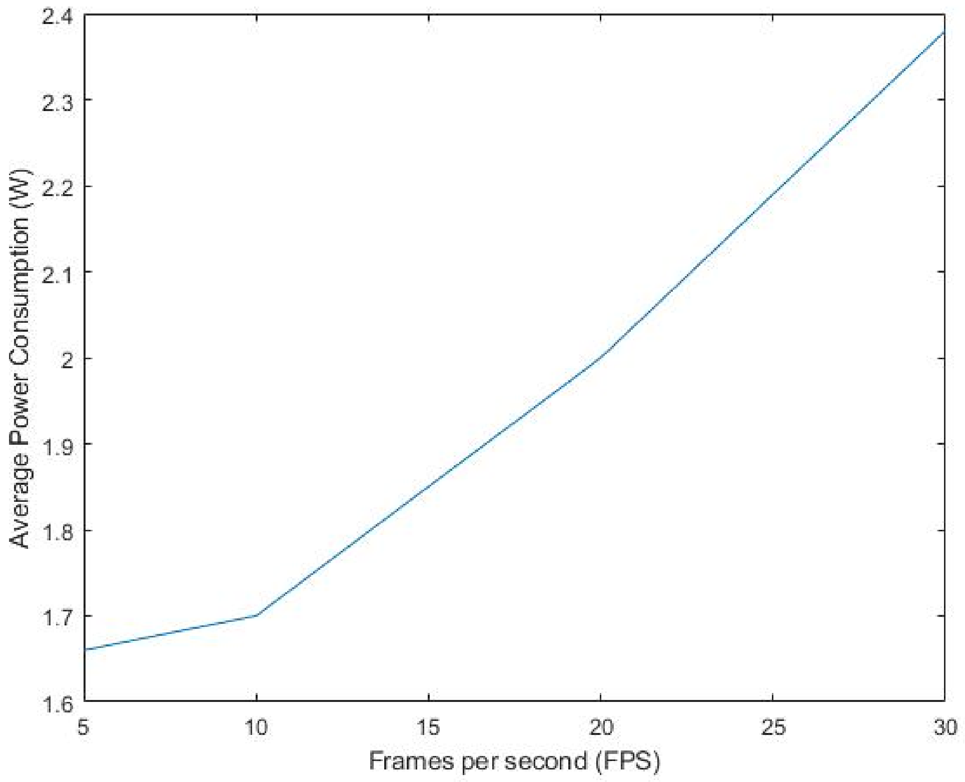


Figure 5.1.: Average Power Consumption during Object Detection

The programs that use Camera Inference uses the Pi Camera to capture images. This image stream gets transferred to the Raspberry Pi Zero, who processes this stream on the built-in GPU, which then feeds the output to the HDMI. The GPU consumes power for each operation it needs to perform. Reducing the number of frames the Pi Camera produces should help in reducing the overall power consumption since the GPU now needs to process fewer images. This should also reduce the power needed by the camera when filming. Figure 5.1 shows how the average power is affected by reducing the number of frames per second. The object detection model is used with the Camera Inference in the program when generating the results, and the frame-rate of the camera is tested with a varying configuration of frames per seconds of 5, 10, 20, and 30. By reducing the frame-rate from 30FPS to 5FPS, a 30% reduction in average power consumption can be observed. This is likely due to the Pi Camera having to produce fewer frames per second, which also leads to a reduced load on the GPU. It seems like the Pi Camera is the primary source of power consumption, this can also be observed in Figure 5.3, where Image Inference is performed, during this program the camera is not used. The GPU of the Raspberry Pi is active during testing in both the Camera- and Image Inference programs. For the Image Inference programs, the GPU is not actively used by the program, but the Raspbian OS is using the GPU to display the desktop, and as such, the consumption due to the GPU is likely similar for both Camera and Image Inference. Figure 5.2 looks at the average inference time versus the

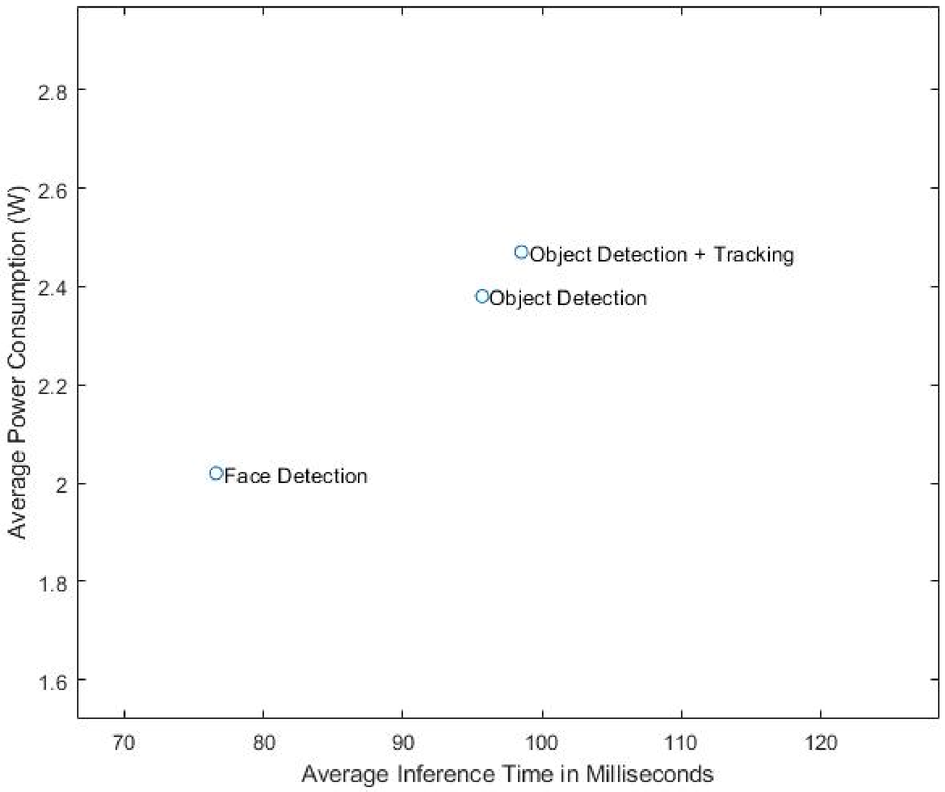


Figure 5.2.: Average Power Consumption versus Average Inference Time of Different Models using Camera Inference Engine

power consumed using the different models when Camera Inference is performed. The face detection model performs the best with the lowest average inference time. This makes sense, given that the face detection model is probably using a simpler neural model, which might require fewer calculations than the object detection model. Both object detection and object detection with tracking perform similarly. There is a small change in power, most likely due to adding more load on the CPU in the form of the tracking module. Some of the power consumption can potentially also be attributed to the use of the annotator API. The API allows for annotation to be displayed on top of the image stream generated by the camera. The annotator is implemented in software and, as such, puts a load on the CPU when it's used. This is done after each inference result. Table 5.1 shows the obtained accuracy results. This is obtained by running a set of 100 images on the Vision Kit. As can be seen from Table 5.1, the object detection model performs better overall. It is not surprising, given that the face detection should only produce a result when there are one or more faces present in the picture. People who are turned away from the camera or where the faces are partially obstructed will probably not give a detection. Both models suffer somewhat from low average accuracy, so depending on the setting in which the models are used, one will most likely see very different results using either of the models.

Model	Average Accuracy
Face Detection	35.4%
Object Detection	48.1%

Table 5.1.: Average Accuracy of Face- and Object detection models when running on 100 images

Figure 5.3 shows the average power and inference time of the face and object detection model when used with the Image Inference engine, both models have approximately the same average inference time, and the average power consumption is also similar in both cases. It is also clear that the power consumption when using Image Inference is lower than the equivalent models run using Camera Inference. This corresponds to the earlier assumptions since the Image Inference does not utilize the camera versus the Camera Inference which does. It is likely that this is the main reason why the two differ Inference types differ. There should also possibly be some reduction due to the program not using the Annotation API, this should lower the CPU utilization which could potentially lead to some reduction in power, however this reduction might be small. Image Inference has a longer inference time in comparison to the Camera Inference. The reason why the two differ is that in the case of Image Inference, the image has to be transferred to the Vision Bonnet. Since SPI is used to transfer the image, this leads to a much longer total inference time. This is one of the major limitations of the kit is configured to communicate with the Raspberry Pi Zero, and it makes Image Inference unsuitable for real-time detection.

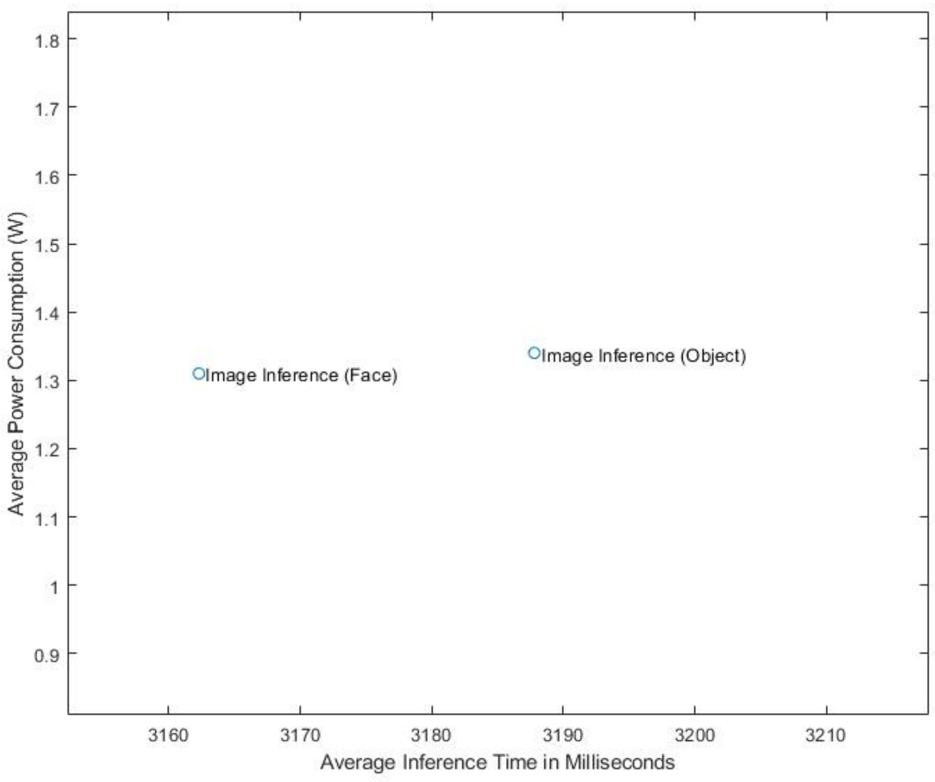


Figure 5.3.: Average Power Consumption versus Average Inference Time on Different Models using Image Inference Engine

6. Discussion

In this section, the results, as presented in the last section, are revisited and discussed further. Both the strength and weaknesses of the kit, the implemented solution, and the measurements are also discussed. The discussion divides into three parts. The first discusses the hardware kit. The following discussion is on the implemented solution, and the last section is a discussion regarding the acquisition of the measured results.

6.1. The AIY Vision Kit

The AIY Vision Kit is a relatively inexpensive and easy way to start learning and applying machine learning to visual processing. Google has made a serious attempt to make the kit easy to set up and easy to use. The supplied pre-trained models, in addition to their Inference Engine and their API for doing annotation, manipulate the hardware, and extracting information from each of the trained models makes it a breeze to work with this kit. The approach to ease-of-use is one of the kit strengths. This does however come at a cost, since the API and inference engine is written to interact with the Myriad VPU closely this makes it more difficult to load the pre-built models in other visual processing libraries, like OpenCV 2. The model files that are read by Google's inference engine and are used on the VPU are binaryproto files, there seems to be no easy way to convert these models back into Tensorflow, and OpenCV is not capable of opening these files. This makes it difficult to test the models in software, something which would be interesting. Since it would clearly show the division between a software-based solution and the accelerated solution. There are other downsides with the AIY hardware, the communication between the Raspberry Pi Zero and the vision bonnet is done by using SPI, this severely limits the performance when loading images to the Vision Bonnet. This is not a major problem when the Camera Inference Engine is used. In this mode, the MIPI bus on the Myriad is used to do high-speed capture of information between the Pi Camera and the Raspberry Pi GPU. As a result, this leads to a reduction in the total communication overhead due to only needing the SPI to deliver the final result of the Inference. This leads to a much lower inference time, as can be seen in Figure 5.2 were Camera Inference is used, versus the inference time when using Image Inference in Figure 5.3. The hardware of Raspberry Pi Zero is limited as well. With 512MB of RAM, which is split between the OS and the GPU. And a single-core 1.0GHz ARM Cortex processor, the use of highly accurate tracking algorithms is infeasible due to computationally heavy calculations. There is also difficulty doing compilation on the device. However, this can be somewhat alleviated by doing cross-compilation on another machine.

6.2. Object Detection and Tracking implementation

As we saw in the results, the overall accuracy of the face detection model and the object detection model is relatively low. This is somewhat expected due to the model used for the object detection, in their paper Google managed to achieve an accuracy of 50.6% when running only MobileNet as a classifier on ImageNet, in this case, the input was slightly smaller 224x224[2], and the network had a depth-wise multiplier of 0.25. This is similar to the average accuracy that was found when testing the system. It should also be noted that the network in the paper is trained on a larger data-set. Another challenge is that the accuracy calculated in this report does not take into consideration the generation and position of the bounding boxes, which is normally considered when testing an object detector. Possible mispredictions are not accounted for when calculating the average accuracy. This implies that the result might actually be slightly lower since mispredictions might occur during testing. There is also the problem with low accuracy of the models, the face detection model has a low accuracy simply due to it not being capable of detecting people when no face is present in an image. The simple centroid based tracker works fine when running the object detection model, and by counting the currently tracked objects which have been registered by the tracker, it seems like it manages to give numbers that correspond a bit more accurately than when only doing the object detection. Since it is a bit more difficult to produce metrics for this setup, more work needs to be done to verify how the accuracy is actually impacted. There are also challenges in regard to the tracker implementation. Since the tracker keeps a detection stored for a custom number of frames, the number generated by the system does not always correspond to the actual number of people in the frame. Another challenge is id swapping; if two people are close together, the tracker might assign an ID already assigned to detection to another detection, since only the number of people present is interesting, the problem of whether a person has the correct id or not doesn't matter. The main problem as a result of id swapping in regards to this system would be that detection of a new person can potentially be linked to an already existing detection, and as such, the number of detections can remain the same. Some of these problems can be addressed by using a more powerful tracker, but this might require more powerful hardware as the Raspberry Pi Zero is not capable of doing accurate real-time tracking with its limited hardware specifications. More research needs to be done on this type of system.

6.3. Measurements and results

There is also room for improvement regarding the acquirement of measurements. The USB Charge Doctor works great as a tool for recording both voltage and current, but a system with a digital readout would be a better solution. This could have been directly integrated with the timing readouts in the different programs. The way these measurements have been acquired for this report is by observing the voltage and current values, during multiple runs of each program. Since the current can fluctuate during the program, a digital readout would yield a more accurate profile of the actual power consumption, rather than the average power consumption based on observation. However, for the sake of making a comparison between the different network models and compare Camera Inference to Image Inference, the average power is sufficient to show the difference, as could be seen in Figure 5.2 and 5.3. The measurements also clearly show how varying the frame-rate of the camera affects the overall power consumption, as shown in figure 5.1. There are also some program configurations which could have been tested, that would have yielded an even better understanding of the average power consumption due to the utilization of the Pi Camera and the camera preview using the GPU. Testing how the average power changes when using the Pi Camera versus not using the camera could help explain why there is a major difference in average power when doing Camera Inference, the main challenge here is that the GPU of the Raspberry Pi is in use regardless of what program is running, since it is used to display the Raspbian OS desktop. One of the significant challenges in regards to measuring neural network performance is the lack of a standardized system. Since networks are configured differently with a broad set of configurable parameters, actually comparing different network types, sizes, and configurations are challenging.

7. Relevant Work

There are few papers where the work has involved the use of a Google AIY Vision kit directly as a "people counter" or as a part of the system in general. This section will present a some few articles that have successfully applied the Google AIY Kit or the Myriad 2 VPU, an article that tries to do detection of people using CNN's. There exist some articles where the Google AIY Vision kit have been used or tested, but for completely different tasks. An example is a poster published by Fermilab, Hawks et al. [15] compared performance of the Vision kit, Google's Edge TPU Coral and a Intel i7-6700 processor when running a simulation on gravitational lens finding, they found that the AIY kit had the worst average inference time at 73ms, this corresponds well with the observations made in this report, the face detection model averages around 76ms. Orjuela-Canon et al. [16] used the Movidius NCS (which features the Myriad VPU) and a Raspberry Pi B 3 to test a detection system for mobility actors (cars, motorcycles, busses and pedestrians). The managed to get an average FPS of 11 using a re-trained MobileNet + SSD architecture, however few details is given in regards to the specifications of the network other than what architecture and it is therefore difficult to compare their performance against the performance of the MobileNet + SSD architecture running on the Vision Kit. Cojocea and Rebedea [17] looks at a multiple ways of doing people counting in crowded places, one by doing video capture and doing the detection on a server, and the other way is done on the embedded system. Their system is similarly structured as the system presented in this report, it is built around the use of an object detector for generating the initial detections then it uses an object tracking algorithm to estimate the number of people that is present in a frame. The problem is that they state in their report that they only have managed to test their system on a computer running two GTX1080-Ti Graphics cards which makes it infeasible to try to compare performance metrics.

8. Conclusion

This report has explored and tested the Google AIY Vision kit and presented a couple of different possibilities for the implementation of a system for detecting and approximating the number of people in a room or office space. The Vision Kit shows promise. It is easy to get up and running. The supplied models and the implemented inference engine makes it easy to test and deploy pre-trained / pre-built models. However, there are hardware limitations both on the Raspberry Pi Zero and the Vision Bonnet, which limits the overall performance and puts constraints on which network architectures can be used and applied to the kit. The models suffer from a low average accuracy of 48.1% for object detection and 34.8%. This makes it somewhat problematic to utilize the kit in a person detection system. It cannot be completely guaranteed that the system is capable of accurately detecting how many people are present in a frame. This can potentially be alleviated by using tracking and continuously adding new persons of interest as they are detected, and continue tracking people who have already been seen. The limitation on the performance of the Raspberry Pi Zero and the lack of high-speed communications options imposes limits on how well the system can perform. Tracking implementations with low computational footprint are capable of running decently on the Raspberry Pi Zero, running more advanced tracking algorithms seems infeasible due to the performance of the Pi. There is also the performance problems imposed by the memory size limitation on the Myriad 2 chip on the Vision Bonnet, which effectively limits the size and complexity of which network architectures can be run on the board. As this is the case, using the kit in a system for detecting people is possible. However, this system would suffer due to low accuracy and performance. The deployment and use of AI on edge is still a relatively new topic of research, and more work and other hardware platforms need to be tested to find the best way of implementing a people detection system.

8.1. Future Work

This project has been an interesting journey, it has introduced me to the concept of machine learning and computer vision. This is an enormous field of research and as such I have barely scratched the surface of what one can achieve when applying machine learning to different vision processing problems. There are multiple things which could be improved upon or tested to further enhance my own understanding, but also to improve upon the solution presented in this report. It would be interesting to test the Vision Bonnet on a more powerful raspberry pi 3B+ or raspberry pi 4. Testing the Coral TPU from Google could also make an interesting case, these runs Tensorflow on edge and it would be interesting to see how well this device would stack up to the AIY kit.

There is also the possibility of improving upon the tracking solution, there exists papers which explains relatively fast offline tracking and implementing or using these to improve upon the detection result, however these were not focused on in this report. OpenCV tracking has been tested on the implemented system, however due to relatively limited hardware, the tracker it self uses multiple seconds to calculated and generate a result, making it unsuitable to be run on resource constraint devices such as the Raspberry Pi Zero, however running this on more powerful devices should be feasible and this is something which also would be of interest, when developing these kinds of systems.

Bibliography

[1] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. In *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 9 2015. URL `http://arxiv.org/abs/1409.1556`.

[2] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. 4 2017. URL `http://arxiv.org/abs/1704.04861`.

[3] Google Inc. Vision Kit Webpage, 2018. URL `https://aiyprojects.withgoogle.com/vision/`.

[4] Intel Corporation. Myriad 2 MA2x5x Vision Processor Transforming Devices Through Ultra Low-Power Machine Vision. Technical report, 2016. URL `www.movidius.com`.

[5] Hamed Habibi Aghdam and Elnaz Jahani Heravi. *Guide to Convolutional Neural Networks: A Practical Application to Traffic-Sign Detection and Classification.* 2017. ISBN 978-3-319-57550-6. doi: 10.1007/978-3-319-57550-6.

[6] Steinar Thune Christensen, Snorre Aunet, and Omer Qadir. *A Configurable and Versatile Architecture for Low Power, Energy Efficient Hardware Acceleration of Convolutional Neural Networks.* ISBN 9781728127699.

[7] Tom Hope, Yehezkel S Resheff, and Itay Lieder. *Learning TensorFlow : a guide to building deep learning systems.* O'Reilly Media Inc, 2017. ISBN 978-1491978511. URL `https://www.oreilly.com/library/view/learning-tensorflow/9781491978504/`.

[8] Dan C. Cireşan, Ueli Meier, Jonathan Masci, Luca M Gambardella, and Jürgen Schmidhuber. Flexible, high performance convolutional neural networks for image classification. In *IJCAI International Joint Conference on Artificial Intelligence*, pages 1237–1242, 2011. ISBN 9781577355120. doi: 10.5591/978-1-57735-516-8/IJCAI11-210.

[9] Yingying Zhang, Desen Zhou, Siqin Chen, Shenghua Gao, and Yi Ma. Single-Image Crowd Counting via Multi-Column Convolutional Neural Network. In *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, volume 2, pages

589–597. IEEE, 6 2016. ISBN 978-1-4673-8851-1. doi: 10.1109/CVPR.2016.70. URL `http://ieeexplore.ieee.org/document/7780439/`.

[10] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. ImageNet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, volume 2, pages 1097–1105, 2012. ISBN 9781627480031. URL `http://code.google.com/p/cuda-convnet/`.

[11] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang Chieh Chen. MobileNetV2: Inverted Residuals and Linear Bottlenecks. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 4510–4520, 1 2018. ISBN 9781538664209. doi: 10.1109/CVPR.2018.00474. URL `http://arxiv.org/abs/1801.04381`.

[12] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review, 9 2017. ISSN 1530888X.

[13] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng Yang Fu, and Alexander C. Berg. SSD: Single shot multibox detector. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 9905 LNCS, pages 21–37, 12 2016. ISBN 9783319464473. doi: 10.1007/978-3-319-46448-0{_}2. URL `http://arxiv.org/abs/1512.02325http://dx.doi.org/10.1007/978-3-319-46448-0_2`.

[14] Adrian Rosebrock. Simple Object Tracking with OpenCV, 2018. URL `https://www.pyimagesearch.com/2018/07/23/simple-object-tracking-with-opencv/`.

[15] Benjamin Hawks, Pradeep Jasal, Michael Wang, Brian Nord, and Iiser Pune. Real-Time Machine Learning Inferencing with Edge Computing Devices from Google and Intel. Technical report. URL `https://coral.withgoogle.com/docs/dev-board/datasheet/`.

[16] Alvaro David Orjuela-Canon, Institute of Electrical and Electronics Engineers. Colombia Section, Institute of Electrical and Electronics Engineers. Colombian Caribbean Section, IEEE Computational Intelligence Society. Colombia Chapter, and Institute of Electrical and Electronics Engineers. *2019 IEEE Colombian Conference on Applications in Computational Intelligence - ColCACI : conference proceedings : June 5-7th 2019, Barranquilla, Colombia.* ISBN 9781728116143.

[17] Eduard Cojocea and Traian Rebedea. Counting People in Crowded Places using Convolutional Neural Networks. Technical Report 2, 2019.