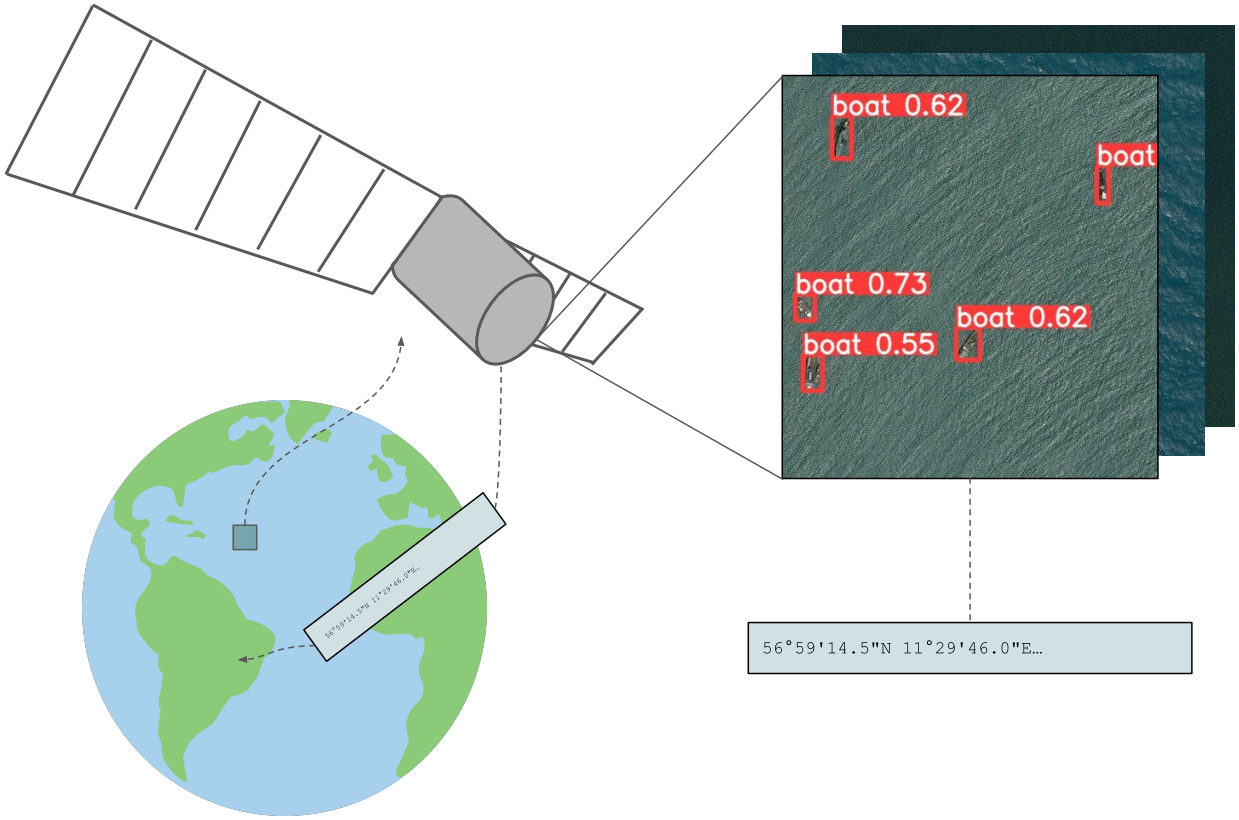
CHALMERS
UNIVERSITY OF TECHNOLOGY



Efficient deep learning in space

Knowledge distillation and optimization of resource usage in a satellite

Master's Thesis in Complex Adaptive Systems

Ebaa Asaad and Sara Larsson

DEPARTMENT OF MECHANICS AND MARITIME SCIENCES

CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2022
www.chalmers.se

Master's thesis 2022

Efficient deep learning in space

Knowledge distillation and optimization of resource usage in a satellite

Ebaa Asaad and Sara Larsson

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
Gothenburg, Sweden 2022

Efficient deep learning in space
Knowledge distillation and optimization of resource usage in a satellite
Ebaa Asaad and Sara Larsson

Supervisors:
Ola Benderius, Associate Professor at Mechanics and Maritime Sciences, Chalmers.
Alice Anlind, Software engineer, Unibap
Hannes von Essen, Deep learning researcher, EmbeDL
Wilhelm Tranheden, Deep learning researcher, EmbeDL

Examiner:
Ola Benderius, Associate Professor at Mechanics and Maritime Sciences

Master's Thesis 2022:33
Department of Mechanics and Maritime Sciences
Division of Vehicle Engineering and Autonomous Systems
Chalmers University of Technology
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Illustration of the pipeline for detecting locations of maritime vessels using object detection on-board a satellite.

Typeset in LaTeX
Printed by Chalmers Digitaltryck
Gothenburg, Sweden 2022

Efficient deep learning in space
Knowledge distillation and optimization of resource usage in a satellite
Ebaa Asaad and Sara Larsson
Department of Mechanics and Maritime Sciences
Chalmers University of Technology

Abstract

The development of micro-satellites and *machine learning* (ML) has increased drastically in recent years, which has unlocked new possibilities in the field of Earth observation. One of the applications is the tracking of maritime vessels since the current tracking systems such as the *automatic identification system* (AIS) can be eluded by simply switching it off. This project, therefore, investigates the possibilities of applying ML in a satellite, specifically with the aim of detecting maritime vessels.

An object detector (YOLOv5) was chosen for testing due to its speed, small size, and its user-friendly framework. For comparison with a simpler model, the classification models ShuffleNetV1 and a custom-built CNN model were chosen. Thereafter, for the purpose of optimization, knowledge distillation, as well as different methods for reducing resource usage, were tested.

The results show that it is feasible to implement ML on board a satellite to detect maritime vessels, where the best result for YOLOv5 was 2.1 min per 10,000×10,000 pixels RGB image on the target hardware using the GPU. Using the CPU with multiple threads achieved a result of 2.2 min for the same image. Increasing the batch size did not yield better results. ShuffleNetV1 was not supported by the TFLite framework, due to a network structure called group convolutions. Neither was quantization supported by the target hardware, but it did decrease the file size of the model by half.

Using knowledge distillation showed great results for the classifiers. Using ShuffleNetV1 to train the simpler CNN model yielded an increase of 12 % in accuracy. It also shows that it is possible to apply a non-supported network on the target device by distilling the knowledge to a supported network. Distilling knowledge using YOLOv5 was more difficult, due to the complexity of the network and the task of object detection. Two methods were therefore tested: using the teacher's output (logits) as a target and using the feature maps within the network as targets (feature imitation). However, only minimal improvements were reflected in the results.

Keywords: satellites, Earth observation, space, maritime vessels, YOLOv5, ShuffleNetV1, Tensorflow Lite, knowledge distillation, optimization, neural networks.

Acknowledgements

First, we want to send our deepest gratitude to our supervisor at Chalmers, Ola Benderius. Thank you for your great support and guidance during this project!

Further, we want to thank the people at EmbeDL and Unibap for giving us this wonderful opportunity! More specifically, we want to thank our supervisors at the respective companies. Wilhelm and Hannes, thank you for your enthusiasm and for being there for us every week with your guidance. Alice, we want to thank you for your never-ending excitement and your guidance throughout this project. We would also like to thank the people at AI Sweden, and especially Erik Svensson, for their support and for providing a fantastic work environment.

Artur and Domenic, thank you for all the feedback during the weekly meetings, and for making this journey just that bit easier.

Last but not least, we want to thank our families and friends for their unconditional love and support for us during this project! We would not have been able to do this without you.

Ebaa Asaad and Sara Larsson, Gothenburg, June 2022

Preface

This project is conducted at Chalmers University of Technology, in collaboration with Unibap AB and EmbeDL. Unibap AB is one example of an actor in the field of satellite hardware that develops computers and cloud services targeting machine learning onboard satellites. EmbeDL works with deep learning algorithms and is specialized in optimizing such algorithms for usage in embedded systems. The project is also included in a master's thesis collaboration organized at AI Sweden, which is an organization that facilitates the development of machine learning in Sweden.

Contents

List of Figures

List of Tables

1
Introduction

The development of *artificial intelligence* (AI) and *machine learning* (ML) has been rapidly increasing for the past few years. The increase in development in ML is mainly due to the increase in computation power as well as the efforts of different people around the world to gather and build large volume datasets, which are necessary to train ML models [1][2]. One challenge regarding ML models is how expensive it is to train them, especially in computation power and time. Moreover, to acquire higher accuracy, ML models have become larger, deeper, and more complicated [3][4][5]. Even after the model is put into production, it still requires a high amount of computation power, especially when striving for instant predictions on frequent inputs or large input sizes [6]. Therefore, ML models often need to be optimized in order to run on embedded and mobile systems [7]. Examples of such systems are smartphones, embedded vehicle systems, and satellites.

Recently, the interest in the usage and development of small satellites for Earth observations has grown rapidly [8][9]. The increasing number of satellites brings possibilities for more frequently acquired data, and more powerful sensors bring higher quality data, which is critical for the utility of ML models. Large volumes of data create the need for specific requirements from the hardware and high downlink and uplink capabilities between the satellite and the systems on Earth [8][9]. Particularly the size of images for *computer vision* (CV) increases drastically as their resolution increases. For example, one image from the Sentinel-2 satellite can be up to 3.2 GB in size (see 3.4.2.1), and the time for one captured satellite image to reach a computing model and give a result could take several days. The delay between taking an image and it reaching a computing model makes it unfeasible for satellite images to be used in applications where it is crucial to act quickly. Therefore, it would be advantageous to bring machine learning models to space to process the data on-the-fly and only downlink the important results.

One major application possibility is the monitoring of maritime vessels on the oceans [10]. The oceans are in critical condition due to overfishing [11]. Unfortunately, it is difficult to monitor activities on the oceans and therefore illegal fishing continues to exist. Also, human trafficking and the smuggling of goods near certain shores are not unusual activities [12][13]. During the time of writing this report, *automatic identification system* (AIS) is the main monitoring system for maritime vessels. It transmits information such as the location and occupation of the vessel. However, it is only required on vessels of a specific size, and it can be easily manipulated by for example, simply switching it off [14].

The onboard computers on satellites have in recent years become more powerful, which makes it possible to use ML models in space [8][9]. However, these computers are still limited in many aspects, such as computation power, memory, and storage compared to, for example, desktop computers. These limitations are mainly due to the shortage of electricity in space. Moreover, several other applications could be using the same platform, which increases the importance of optimizing different parts of the pipeline for a specific process.

There are several ML models for classification and detection on images, already optimized for small mobile systems as well as embedded systems. Examples of these are MobileNetV2 [7], ShuffleNet [15], SENets [16] and certain YOLO-models [3][6][17][18][19][20]. These models may or may not be applicable for a satellite computer, but in order to make efficient use of the expensive equipment in the satellites, the models should be developed to be compact in terms of storage, computing power and time. Another way to decrease the use of these resources required by the process is to change the surrounding processes around the ML model.

1.1 Research questions

This project focuses on benchmarking different machine learning algorithms for object detection and image classification for satellite images. The evaluation of the models includes researching different inference times and model sizes. The best-fitted models will then be tested on the target computer and evaluated in a similar matter. Furthermore, a review of the model as well as the processes surrounding the model will be done to find optimization possibilities and further decrease the consumption of computation power, memory usage, and storage. This results in two research questions for the project:

- What is the performance of a developed machine learning model in terms of accuracy, inference time, and storage when deployed on a satellite (for example by using Unibap's satellites)?
- What optimization methods can increase the performance of the machine learning model in terms of accuracy, memory usage, inference time, and storage when deployed on a satellite (for example by using Unibap's satellites)?

1.2 Limitations

No processing of raw satellite images was performed in this project. Hence, the dataset used in the training of the networks was the Kaggle's Airbus ship detection challenge dataset [21].

Moreover, the performance of the ML models is only tested on Unibap's hardware, more specifically a satellite computer called iX5-100 [22]. The results and optimization methods are therefore only meaningful for this, and similar, hardware. The satellite computer was not physically in space, since the focus of the project is the performance of the models themselves. Testing in a computer in space would be limiting, due to environmental factors such as radiation, but also due to the lack of freedom when testing in such an environment, as other programs are running there as well. Furthermore, the hardware is only used as a testing bed for the models, i.e. no training was conducted on the specified hardware.

1.3 Important metrics and acronyms

This section explains the different metrics and acronyms used throughout the project when measuring the performance of machine learning models. Table 1.1 explains the metrics used for measuring speed and complexity. Table 1.2 explains some metrics used for evaluating classifiers and object detectors. Table 1.3 lists some important acronyms relating to memory used by the *central processing unit* (CPU) and *graphics processing unit* (GPU).

FPS	Frames per second. The number of frames a model is able to process during one second.
FLOP	Floating point operations (commonly written as FLOPs). The number of operations required to run a single instance of a model. Not to be confused with FLOPS.
FLOPS	Floating point operations per second. Usually used for indicating the computing capacity for a hardware. Might be useful in the context of ML for determining a theoretical time for training on a specific hardware. Not to be confused with FLOP.

Table 1.1: Metrics used to indicate the speed and complexity of a ML model.

FN, FP, TP, TN	See Fig. 1.1.
Accuracy	The proportion of true results among all examined cases. It is defined as $\frac{TP+TN}{TP+TN+FP+FN}$.
IOU	Intersection over union (commonly written as IoU), given by the intersection of the predicted bounding box and the labeled bounding box, divided by the union of the predicted bounding box and the labeled bounding box.
P	Precision, given by the number of predicted labels that match the ground truth, divided by the total number of predictions ($\frac{TP}{TP+FP}$). TP and FP are in object detection given by a threshold of IOU.
R	Recall, given by the number of predicted labels which match the ground truth, divided by the total number of annotated labels ($\frac{TP}{TP+FN}$). TP and FN are in object detection given by a threshold of IOU.
AP	Average precision, given by the area under the precision vs recall curve.
mAP	Mean average precision (commonly written as mAP). The mean of all AP for each class. mAP.5 means that the threshold for IOU is 0.5. mAP0.5:0.9 means the average thresholds over IOU, from 0.5 to 0.95 with step 0.05.

Table 1.2: Metrics for evaluating the accuracy of classifiers and object detectors.

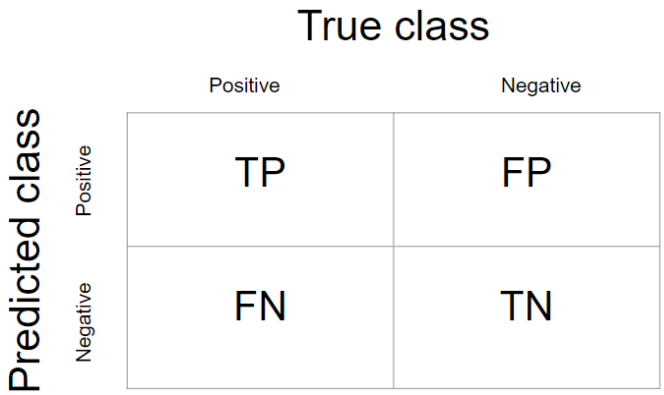


Figure 1.1: The confusion matrix. The abbreviations represent true positives (TP), true negatives (TN), false positives (FP), and false negatives (FN).

RAM	Random access memory. A fast memory allocation for active processes.
VRAM	Video RAM. RAM memory used by the GPU.
VIRT	Virtual memory. The memory allocated for a process to use, represented as a continuous memory but possibly mapped to noncontinuous memory locations.
RES	Resident size, RAM memory of the CPU used by a certain process.
SHR	Shared memory, memory accessible by multiple processes to allow for inter-process communication.

Table 1.3: Different memory types used by the CPU and GPU

1.4 Outline

Chapter 2 highlights the important and relevant background and previous research needed to understand the results. Sect. 2.1 gives an understanding of the satellites in the field of Earth observation and Sect. 2.2 explains the different datasets commonly used in the context of training and testing machine learning models. Sect. 2.3 introduces the relevant machine learning algorithms. Lastly, Sect. 2.4 explains common optimization methods, including knowledge distillation in Sect. 2.4.1.

To understand the processes applied in this project, a description of the steps and choices are explained in chapter 3, Method, including the datasets in Sect. 3.1, choice of models in Sect. 3.2, training setup in Sect. 3.3, evaluation in Sect. 3.4, and finally the choice of optimization methods in Sect. 3.5.

The results are then presented in chapter 4, where the evaluation of the ML models in relation to accuracy is presented in Sect. 4.1, and results for the optimization in 4.2 and Sect. 4.3.

Lastly, the discussion of the results is found in chapter 5 and the conclusion in chapter 6. In detail, the used dataset and model choices are discussed in Sect. 5.1 and 5.2 respectively. The performance of the models are then discussed in Sect. 5.3, the outcome of using knowledge distillation in Sect. 5.4, theoretical methods for optimizing the pipeline in Sect. 5.5, the applicability of machine learning in space in Sect. 5.6, ethical and sustainability challenges in Sect. 5.7, and possible future work in Sect. 5.8.

2
Background

Machine learning as we know it now has its origin in the late 1960s through the work of Rosenblatt [23][24]. He created a machine that was capable of recognizing the letters of the alphabet, which was called a "perceptron". The perceptron later became the prototype of modern artificial neural networks [24]. The real turning point for machine learning was at the beginning of the 21st century. The turning point is mainly due to certain trends, where the first one is the big data trend. Large volumes of data became available and new methods were necessary to deal with these large volumes of data [24]. The second trend was the breakthrough when GPUs became affordable in relation to their computational capabilities, in connection with a better and standardized software to use the devices. The rise in computational power and its affordability allowed the distribution of large amounts of data among different processing units as well as the processing of a large amount of data in the memory [24]. The third trend was the development of new algorithms in the machine learning domain, specifically deep machine learning [24].

This chapter presents some basic information to build a foundation for understanding and putting into context, the results of this project. Sect. 2.1 presents some background on different types of satellites and hardware that accompany them. Sect. 2.2 goes through a few of the different datasets used for training a machine learning model. Sect. 2.3 contains information about how machine learning is commonly used in the domain of image analysis by introducing a few models. Finally, a few different algorithms for optimizing deep learning models are explained as well in Sect. 2.4.

2.1 Satellites

Satellites have been used for many purposes, one of which is Earth observation. There are different types of satellites such as open data satellites and commercial satellites. Copernicus is an open-access earth observation program organized by the European space agency that aims to monitor changes in the planet and its environment [25]. The project provides information that can be used for a variety of applications, such as regional and local planning, agriculture, and fisheries to name a few [25]. The project provides six thematic streams of services, namely atmosphere, marine, land, climate change, security, and emergency [25]. The goals of the project are achieved through the Sentinel family of satellites which includes the Sentinel-1 to Sentinel-6 satellites [26]. Sentinel-1 and Sentinel-2 provide radar and optical

imagery respectively, for land and marine services, and are used in previous projects aimed at monitoring coasts [10][26][27]. The Sentinel-1 constellation consists of two satellites, which provide radar imagery with a resolution down to 5 m and coverage up to 400 km as well as a six days revisit period [28]. Having radar imagery makes it possible to generate images regardless of the weather and time of day [28]. Sentinel-2, which also consists of two satellites, on the other hand, has an optical camera providing optical images with a resolution down to 10 m and a swath width up to 290 km[29]. There are also commercial satellites such as WorldView, which can provide images down to 30 cm resolution [30].

2.2 Datasets

A dataset needs to be large and diverse to achieve good training on a machine learning model and to make the task of generalizing easier. One of these datasets, which is widely used as a benchmark in image classification, is the ImageNet dataset [31]. The dataset has around 14 million images consisting of 21,841 subcategories [32]. Another dataset widely used by the community of machine learning developers is the COCO dataset, which is a popular dataset used for object detection [33]. The COCO dataset contains 330,000 images where over 200,000 of them are labeled [34]. Another dataset for object detection is the PASCAL VOC2017 originated from one of the PASCAL visual object classes (VOC) challenges that aims to provide standard annotated datasets and evaluation procedures for CV and ML [35]. The dataset consists of 9,963 annotated images collected from the flickr2 photo-sharing website.

Images can also come directly from satellites such as the Sentinel-family, which is a part of the Copernicus program [25]. However, these images are considered raw data and are not directly usable in the context of machine learning, as they require some preprocessing steps. These images can be found in the Copernicus Open Access Hub [36]. This hub was used in several projects aimed at monitoring coasts, by combining satellite imagery with AIS data [10][27]. However, the datasets generated in those projects are not open to the public.

Another source for datasets relating to data science and machine learning is Kaggle [37], which contains many different datasets for different applications. However, the dataset of interest to this project is the Kaggle Airbus ship detection challenge [21]. This dataset contains around 200,000 satellite images prepared for usage in the context of machine learning.

2.3 Machine learning in image processing

This section goes through different machine learning models used for image classification (see Sect. 2.3.1) and object detection (see Sect. 2.3.2).

2.3.1 Image classification

Image classification has been an important topic for many years. It is a complex process that is affected by many factors. Many algorithms such as K-means, minimum distance, and maximum likelihood have been used for image classification [38]. However, artificial neural networks have become increasingly popular in the last few years because of the advances in computation power [24] and the networks' ability to approximate any function [39].

2.3.1.1 ShuffleNetV1

ShuffleNetV1 is a *convolutional neural network* (CNN) designed for mobile devices [15]. It utilizes pointwise grouped convolutions to reduce the computation complexity [15]. This is reflected by the low number of FLOP (approx. 140 MFLOP) the network has [15]. One drawback of using grouped convolutions however is that the output of a certain group only relates to the information of the inputs within that group [15]. The information is not shared across groups, which can affect the way the network learns, as the output feature maps lack inter-communication. Therefore, channel shuffle is used in ShuffleNetV1 as a way to mitigate the effect of grouped convolution, while conserving the advantages gained in computation complexity [15].

2.3.1.2 Other classifiers

There are lots of different networks that have been developed for image classification. One of those networks is MobileNetV2, which is an artificial neural network designed to run on mobile and embedded devices [7]. It uses depthwise separable convolutions, linear bottlenecks, as well as inverted residuals to achieve the goal of being able to run on mobile devices [7]. The idea is to reduce the computation complexity while maintaining good accuracy. Another network is Xception [40], which also makes use of depthwise separable convolutions. More detailed comparison of image classifiers can be found in the appendix, Table. A.1.

2.3.2 Object detection

Another more complicated field of computer vision is to detect and classify an object within an image. This process is however cumbersome since it consists of two tasks and it often comes down to a trade-off between speed and precision [3][6][17]. Therefore, two types of object detectors have been developed to be suitable for different tasks, mainly the *two stage detector* (TSD) and the *one stage detector* (OSD) [6]. For example, a fast detector is required if the prediction is used in a live video feed or in recommendation systems that detect certain events, while an accurate detector could be used in systems with non-frequent input where the accuracy is more important than speed.

TSD consists of two processes, one for detecting the location of an object and one for classifying the object [6][17]. The processes are trained separately which is an easier fitting task than training them simultaneously. However, the TSD contains

two models in a pipeline where the models need to have a high amount of parameters and sub-processes in order to achieve high accuracy. They are therefore less effective in time and computing power. In constrast, OSD was developed to create a fast model that would be able to detect and classify instantaneously, for example in a video feed. The trick is to have a single network for predicting both the localization and the classification of an object and transform the problem into a regression problem. It was first introduced in a network called YOLO (You only look once) in 2016 [17][41] and has triggered a massive exploration of fast object detectors. Both new versions of YOLO and other models have been developed since [3][6][18]. The increase of complexity in the multitask of localization and classification however results in less acurate predictions. Since this project targets a restricted computer where the model predicts vessels from satellite images directly from the camera, the OSD will be further investigated.

The history of YOLO and a literature review of different object detectors can be found in Sect. A.1 and Sect. A.2 respectively. Furthermore, a comparison of different object detectors can be found in Sect. A.3.

2.3.2.1 YOLOv5

YOLO "only looks once" at the image as it simultaneously classifies and detects objects within that image. The network consists of a *backbone* (feature extraction), *neck* (feature aggregation), and a *head* (prediction and regression). The reason why this network received massive attention within the community was due to the possibility of fast detection of objects within the frame rate of a video [17]. In addition, by using the image directly, an encoding of the global features could be obtained. This yields a strong generalization in the feature extractions and also reduces the error of detecting the background [42]. Several versions of YOLO have been developed since, which have increased the efficiency, accuracy, and applicability of the model. YOLO could even be implemented in small and mobile embedded systems despite being a deep neural network.

YOLOv5 uses a so-called "bag of freebies" that consisted of methods to decrease the training costs and a "bag of specials" that significantly improves the accuracy with a small increase in inference cost [6][20]. The backbone consists of *cross-stage partial* (CSP) blocks, which maintain fine-grained features by repeatedly sending half of the feature map through a dense layer where the output is then concatenated with the untouched half. This allows for a better gradient flow through the dense layers, saves the gradient changes in the feature map, and reduces the number of parameters. The neck then consists of a *spatial pyramid pooling block* (SPP) and a *path aggregation network* (PAN). SPP provides a fixed output size with a dynamic input size and it increases the receptive field using different sizes of pooling which are proportional to the input size, creating a spatial pyramid. PAN is used to preserve the spatial knowledge throughout the network by concatenating earlier feature maps with later feature maps. The head consists of convolutional layers that output the prediction values. YOLOv5 tackles the high variety of object sizes by extracting

feature maps from three different parts of the PAN, which facilitates the detection of objects with different sizes. A simplified overview of the specific network architecture used in this project is later shown in Fig. 3.2.

YOLOv5 is not implemented using the framework Darknet as the previous YOLO-versions. Instead, it is using the user-friendly framework PyTorch that is widely used in the machine learning community [43]. YOLOv5 is also constantly developing with new versions, and the framework gives a wide range of settings and possibilities to modify the network structure for customization. YOLOv5 also uses auto-learning bounding box anchors for easier choice of anchors.

2.4 Optimization

This section covers two methods for optimizing neural networks, knowledge distillation, and quantization. There is however a large collection of other methods that could be used for optimization of neural networks such as network pruning that will not be covered in this paper.

2.4.1 Knowledge distillation

A simple method to improve the performance of a model is to average the output of an ensemble of different models trained on the same data [44]. This is however computationally expensive since all models must be used for a single prediction. It is however possible to avoid this by distilling the knowledge of the trained ensemble of models into a single model, which is called *knowledge distillation* (KD).

Another difficulty in optimizing networks is knowing how to change the model structure without losing the trained knowledge. With KD however, the knowledge could be distilled into a smaller network by teaching the smaller network to match the features of the teacher network instead of trying to slim down the original network [44].

KD not only reduces the complexity of the model but also helps the model in generalizing [45][44]. The generalization ability of networks is often desired, but difficult to obtain since the model would predict cases it has not previously seen. A trained KD model could therefore achieve better generalization due to the fitting of knowledge into a less complex network.

KD defines a loss function from the difference between the outputs of the teacher model and the student model when running inference on the same data. The last layer of the classification CNN-models produces probabilities of all possible classes using a softmax layer to point out the most probable classes, depending on the differences between the output elements. The wrong classes in a confident classifier, therefore, obtains very small values (close to 0) and would almost not be counted into the KD loss at all. Hence KD uses a temperature T in the softmax function that reduces the differences and creates *soft labels* as shown in (2.1).

$$q_i = \frac{exp(z_i/T)}{\sum_j exp(z_j/T)} \tag{2.1}$$

where q_i is the probability given to class i obtained from comparing the logit (non-normalized value) z_i with all other logits and T is the temperature modifying the softmax. The total loss is then calculated using following (2.2).

$$L_{KD} = \frac{1}{m}\sum_{p=1}^{m}(s_p - t_p)^2 \tag{2.2}$$

where m is the number of elements in the predictions p, obtained from s (the student) and t (the teacher).

There are several ways to set up KD. One effective way is to use a labeled training set to train the student network where the prediction loss of the student network is combined with the KD loss, as seen in (2.3). This helps the student in the right direction and forces the student to match the soft targets of the bigger network [44].

$$L = L_{pred} + \lambda L_{KD} \tag{2.3}$$

where λ is the factor determining the grade of affect KD will have on the loss. This helps the smaller network in the right direction since it cannot completely match the bigger network.

Traditional KD with soft labels works great on simpler tasks such as image classification but is less effective on complex tasks such as object detection. Especially reliable localization knowledge is hard to distill [45]. The reason for this is that the classifier works on a global context in the image while detectors target local areas. Hence distilling the knowledge will result in noise given by the background. There is however another method for KD that focuses on the localization task that uses fine-grained feature imitation on local near object regions to calculate the loss, as shown in (2.4).

$$lij = \sum_{c=1}^{C}(f_{adap}(s)_{ijc} - t_{ijc})^2 \tag{2.4}$$

where i and j are the coordinates for each object anchor and c the class. s is the student feature map and t is the teacher's feature map. f_{adap} is a layer to adapt the student's layers to be compatible with the teachers. Using an imitation mask I, the estimated near anchor locations are included shown in (2.5).

$$L_{im} = \frac{1}{2N_p}\sum_{i=1}^{W}\sum_{j=1}^{H}\sum_{c=1}^{C} I_{ij}(f_{adap}(s)_{ijc} - t_{ijc})^2 \tag{2.5}$$

where $N_p = \sum_{i=1}^{W}\sum_{j=1}^{H} I_{ij}$ (number of positive points in the mask), W the width and H the height. L_{im} is then added to the training loss L_{det} with a balancing factor λ, as shown in (2.6).

$$L = L_{det} + \lambda L_{im} \tag{2.6}$$

This method improved the discrimination ability of the resulting network and provided a more reliable localization [45]. The errors of detecting the background as an object and duplicating or grouping the objects were decreased.

2.4.2 Quantization

Quantization reduces computation and model size by reducing the precision of the weight datatype from a higher number of bits to a lower number of bits [45][46][47][48][49]. It is an efficient method for optimization with some loss of accuracy in network calculations. Although, this requires that the device supports the low-bit operations.

3
Methods

This chapter presents the different methods, datasets, and frameworks used in this project. Sect. 3.1 presents the dataset used. Sect 3.2 goes through the models and reasoning behind the choice of them. Sect. 3.3 presents the training setup, such as hardware, dataset split, etc. Sect. 3.4 goes through the method for evaluating the results obtained, while Sect. 3.5 presents different optimization methods that are used in this project.

3.1 Dataset

A dataset consisting of 192,555 satellite images with the size of 768×768 pixels was chosen from the Kaggle Airbus challenge. Kaggle Airbus was chosen because of the already prepared and labeled images for training, meaning that no preprocessing is required for them to be utilized. Moreover, the dataset includes many images from different environments (water, clouds, land, etc.), which can help mitigate any bias that can arise in the dataset. Using unprocessed satellite images, such as Sentinel 1 and 2, would have required a massive amount of preprocessing. This would have taken unnecessary time since it would not have added any value from a research point of view, as the focus of this project is not the dataset. A sample of the images available in the dataset can be found in Fig. 3.1.

Figure 3.1: A sample of the images available in the Kaggle Airbus Challenge dataset.

3.2 Model choice

A classification model was chosen as a first step for classifying the images. It is also a good reference point that can be used for benchmarking the performance of other model types.

An object detector was also chosen above other methods due to the high interest in object detectors in the field of computer vision. An object detector is also suitable due to the recent improvements in speed as well as accuracy in the OSDs. Moreover, testing the feasibility of these more complex networks in a satellite would be interesting. Furthermore, an object detector is suitable because the target case is to detect maritime vessels. An object detector gives more rich information about the vessels such as the precise location and size of the vessel.

3.2.1 Classifiers

ShuffleNetV1 was chosen as a binary classifier based on its low number of FLOPs. It was implemented as described in the paper for ShuffleNetV1 [15]. Specifically ShuffleNetV1 1x, with three groups, was chosen. Moreover, the framework used for building the model was Tensorflow's Keras module. The model is relatively small as it only has 994,514 parameters.

A simple CNN model was also used for distilling knowledge from ShuffleNetV1, and for usage in the satellite computer. The model is small (240,722 parameters), and a description for the simple CNN model can be found in A.4.

3.2.2 Object detection: YOLOv5

YOLOv5 was chosen because of the simplicity of implementation and great performance, as explained in Sect. 2.3.2.1. The model is widely used due to the implementation in the framework Pytorch, which has a larger community compared to the frameworks of YOLOv4 (Darknet) and PP-YOLO versions (PaddlePaddle). YOLOv5 is also 90 % smaller in file size compared to YOLOv4 according to the literature study in Appendix A.3.2. Comparing the latest models for the smallest networks in each YOLO version, YOLOv4Tiny and YOLOv5n, (presented in Table 3.1) does also confirm that YOLOv5 is much smaller.

For stable training results, the released version 6.1 of the official GitHub repo for YOLOv5 was used [50]. A visualization of the specific version of model architecture that was used is shown in Fig. 3.2 which was the latest architecture for YOLOv5. The exact layer settings are saved in the YAML files in the project GitHub repository for YOLOv5 [51].

The loss function used is also the default one of YOLOv5. It includes the classification loss (class probabilities), the objectness loss (confidence of the detection), and the regression loss (IOU of the bounding box), also seen in (3.1).

$$L = L_{cls} + L_{obj} + L_{box} \tag{3.1}$$

YOLOv4Tiny	YOLOv5n
23.67 MB	4 MB

Table 3.1: The smallest models of YOLOv4 and YOLOv5 compared by their file-size, where YOLOv5n is approximately 83 % smaller.

YOLOv5 model	Large	Medium	Small	Nano
Number of parameters (M)	46.1	20.9	7.0	1.8

Table 3.2: The different YOLOv5 models tested in this project and their numbers of parameters.

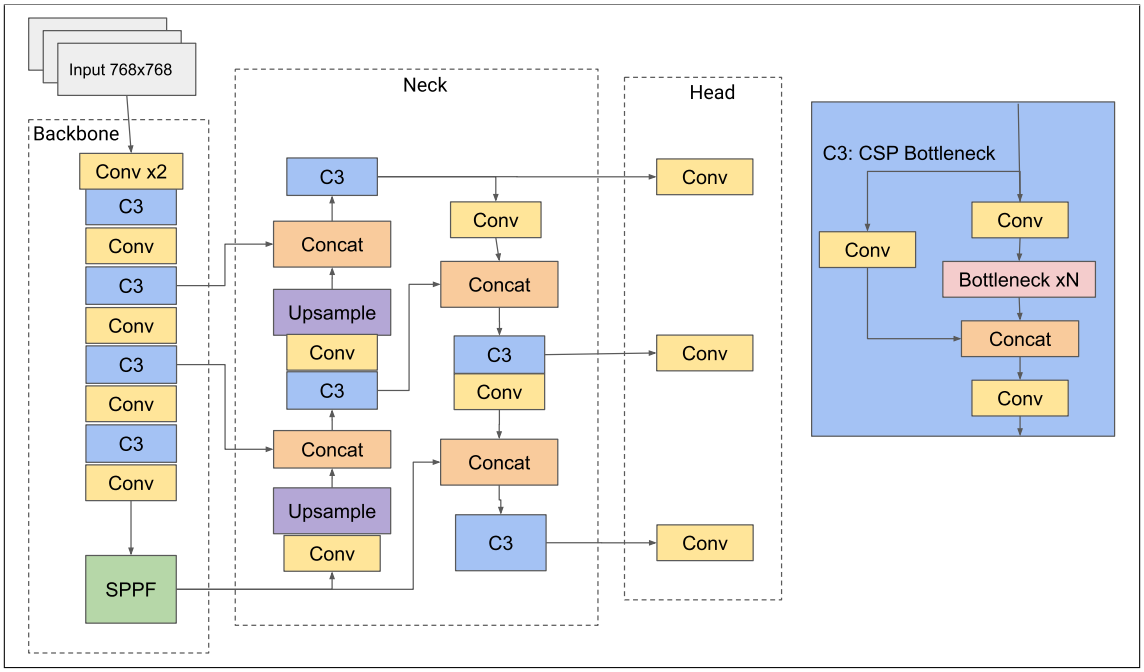


Figure 3.2: A simplified overview of the network architecture for all the YOLOv5 models used in this project. *Conv* indicates convolutions and *Concat* is where the layers are concatenated. *C3* is a particular CSP block shown in the blue box to the right, where the *bottleneck* reduces the dimensions. It is also uses the *SPPF* which is a faster SPP.

3.3 Training setup

The training of the networks was done using the Kaggle Airbus dataset mentioned in Sect. 3.1. For classification models, the dataset was randomly split into 60 % training, 20 % validation, and 20 % test. For YOLOv5, the dataset was divided into partitions 70 %, 15 %, and 15 %.

ShuffleNet was trained for two epochs due to the network overfitting easily, as the number of training images is large. Moreover, since validation is done at the end of an epoch, the model was trained for exactly two epochs. The code is found in the GitHub repository for this project [51].

The released version 6.1 of the official Github repository for YOLOv5 was used to train the YOLOv5 models. Multiple model sizes (see Table 3.2), pre-trained and without pre-trained weights were tested. The pre-trained weights were acquired from the original GitHub repository of YOLOv5, and are trained on the COCO dataset [50]. All model structure settings (saved in YAML files) and the pre-trained weights are furthermore saved in the GitHub repository containing the training setup for YOLOv5 [51].

Since the process of training the models is separate from doing inference, the training was done on a much more powerful computer. The training was performed using AI Sweden's cluster server which is accessed through an interface called *AiQu*. Docker images were used for training, where they were built locally and pushed to DockerHub to be accessible by the AiQu server. The specifications for the AiQu server can be found in Table 3.3a.

Kernel	Linux
Machine	AMD64
CPU cores	256
GPU	NVIDIA A100-SXM4-40GB
Memory	40 GB/GPU, 1008 GB RAM

(a) Specifications of AiQu. Note that all resources are shared between multiple projects at AI Sweden.

Kernel	Linux
Machine	AMD64
CPU cores	4
GPU	AMD R3E
Memory	2 GB (GPU/CPU)

(b) Specifications of Unibap's satellite computer iX5.

Table 3.3: Hardware used for training and evaluating the models.

3.4 Evaluation

This section provides an overview of the evaluation process. Firstly, it goes through the method for measuring the accuracy of the models in Sect. 3.4.1. Secondly, Sect. 3.4.2 goes through the different methods used when testing inference on the target hardware.

3.4.1 Measuring accuracy

AiQu was used to evaluate the accuracy since the process is computer-heavy due to the number of images used for prediction testing (38,511 images for classification and 28,883 images for object detection). Testing on another computer than the target computer is possible since the prediction results are not computer specific, they originate from math operations within matrices.

3.4.2 Measuring resource usage

For realistic and relevant results in the aspect of speed and memory usage, a satellite computer was used. Specifically, a satellite computer called iX5 was utilized for testing. The specifications can be found in Table 3.3b. The computer used was located in Uppsala at Unibap's office for stable testing and to follow the limitation of this project to not take into account the environmental effects in space. Measurements in space do also require more preparation and therefore restrict the number of testings. VPN and SSH were used to establish a connection from a distance. The testing required conversion of the model weights to one of the supported frameworks, *Tensorflow Lite* (TFLite), which is optimized for embedded devices such as the ix5 [9]. Specifically, the C++ version of TFLite was used for testing with CMake as the building tool. To validate if the conversion to TFLite was successful, a comparison of several predictions by the TFLite interpreter and the Pytorch model was done on the same input images. A link to the repository containing the complete code for the TFLite testing can be found in the project GitHub [51].

3.4.2.1 Input image

No dataset was used for the testing of resource usage in the satellite since no training is performed during an inference. Furthermore, the prediction is not important during the measurements of time and memory usage. Therefore, a dummy image was created for testing, with the same size as a large satellite image. Sentinel-2 user handbook states that a sentinel-2 camera samples 13 spectral bands: four bands at 10 m, six bands at 20 m, and three bands at 60 m spatial resolution with an orbital swath width of 290 km [52]. These resolutions mean that for a 10 m resolution and 290 km×290 km image, the number of pixels in such an image is 29,000×29,000 pixels, for 20 m it is 14,500×14,500 pixels, and for 60 m it is 4,834×4,834 pixels. Given that each pixel is in `float32` format, which takes up four bytes, such an image would have a size of roughly 3.2 GB for the finest resolution. This is the worst-case scenario for one of the best cameras available on satellites, therefore it is not feasible to test using these numbers. The resolution and size of the image chosen for testing in this project is, therefore, a made-up case of 10,000×10,000 pixels, which is a more reasonable worst-case scenario regarding smaller satellites. The precision of the image was also chosen to be `int8` to match the datatype of the satellite images in the Kaggle Airbus Challenge dataset.

3.4.2.2 Inference time

The inference time of the TFLite models were measured using an automated script in bash and executable programs in C++ targeting the GPU and the CPU. The time was measured using time functions in the C++ code:

```
time1 = std::chrono::high_resolution_clock::now()
\\ Code to measure
time2 = std::chrono::high_resolution_clock::now()
```

The resulting time taken was then calculated using the difference between two measured times. Several tests on each model were performed for stability in the testing. The timing tests were also measured assuming that the initialization of the GPU delegate is already done for fair testing results between the GPU and CPU. The initializing of the GPU increases the time significantly and can be reduced or eliminated, which is later described in Sect. 5.8.

3.4.2.3 Memory

Memory usage was measured with an automated bash script. External measuring tools for Linux systems called *top* and *radeontop* were used to track the memory usage of the built programs. These tools were used to capture the memory usage of all the surrounding functions as well as the overhead. *Top* measures the RAM (CPU), VIRT, RES, and SHR of a certain process, and *radeontop* measures the VRAM usage. The swap memory (memory the process borrows from the hard disk when running out of RAM) was also turned off during the tests, to get more accurate results.

3.5 Optimization methods

To optimize the speed and the memory usage of the models, a thorough investigation of the methods used for setting up the pipeline of the process was done. Knowledge distillation was also used, which is another indirect optimization of resource usage that targets a better accuracy using a smaller network. Other common methods such as network pruning were not investigated. Quantization is however explored since it is easily applied after the model is trained.

3.5.1 Optimizing resource usage

By evaluating several methods of structuring the code and using different methods in the frameworks, a mapping of the efficiency of each method could be obtained.

3.5.1.1 Memory management and conversion of datatypes

The image consisted of eight-bit integers, and the model required 32-bit floats as input. Therefore conversion of input datatypes was required. Initially, the image was converted to `float32` directly after loading. Then, a more efficient method of only converting the *tile* (a smaller part of the image) that is sent to the input tensor

was tested.

TFLite requires several steps for the values to actually reach the input tensor. The values have to first be copied over to an input tensor that is accessible by the program. Thereafter, when running the invoke-command where the prediction is obtained, it copies all values to the actual input tensor located in the delegate. In the beginning, the C++ function called *memcpy* was used for copying the value to the input tensor accessible by the program. Then, due to troubles related to the program crashing due to segmentation fault, another method was tested of simply copying every element value one by one to the accessible input tensor.

The above explained methods of building the code resulted in three different methods, *Method A*, *Method B* and *Method C* (see Table 3.4).

Method	Explanation
A	Convert the complete image to `float32` directly after loading and use `memcpy` to copy values into the accessible input tensor.
B	Convert only the tile to `float32` and use `memcpy` to copy values into the accessible input tensor.
C	Convert only the tile to `float32` and copy the values by indexing pixel by pixel to the accessible input tensor.

Table 3.4: The tested coding structuring methods for running inference on the target hardware using the proposed 10,000×10,000 pixels RGB image.

3.5.1.2 Inference options

Different inference options on the model were also tested. Different input sizes were tested with sizes divisible by 32 since YOLOv5 has this restriction for the input. Furthermore, different numbers of threads were tested on the CPU, since threads were not supported by the GPU. Note also that multithreading is only utilized during the invoking of the model. Lastly, different batch sizes were also tested.

3.5.2 Knowledge distillation

KD was used to increase the accuracy for smaller models, which is an indirect optimization of resource usage. The teacher model for the classifier was ShuffleNet where the knowledge was distilled to a smaller CNN model explained in Sect. 3.5.2.1. Sect. 3.5.2.2 explains the setup of using bigger YOLOv5 models as teachers for a YOLOv5n (nano) model.

3.5.2.1 Knowledge distillation for classification

ShuffleNetV1 was used to train a smaller CNN model. The model description can be found in Appendix A.2. The simpler model was not optimized for the best possible accuracy, it was rather taken as a reference for comparison before and after applying

KD. The soft outputs of the teacher model were used in the distillation loss as seen in (2.3). The model was trained in the same way as ShuffleNetV1, for two epochs and with the same dataset, in order to facilitate a direct comparison.

3.5.2.2 Knowledge distillation for object detection

KD was used for object detection by distilling larger models into a YOLOv5 nano model. Both methods of KD-loss were tested, the loss function for soft labels (2.2) and the loss function for the feature imitation (2.5).

The feature maps used for the feature imitation were obtained right after the concatenations occurs when merging earlier feature maps to later feature maps in the PAN structure (4 extra connections between the vertical paths in Fig. 3.2).

Moreover, it was not possible to use the soft labels in YOLOv5, since the Softmax-function is not present in YOLOv5. Therefore, only the direct outputs of the network (logits) were used instead with the same loss function (2.2). The KD-losses were then combined with the loss from the ground truth (2.3)(2.6).

The COCO dataset (see Sect. 2.2) was also used when testing KD in order to get a better understanding of the effects that the KD algorithm has on YOLOv5 when having one class (this case) compared to having multiple classes.

4
Results

This chapter presents the results obtained from different tests. Sect. 4.1 presents the results of training YOLOv5 and the classifiers, in regards to accuracy and other ML-related metrics. Thereafter, in Sect. 4.2, the results of running the models in a satellite computer are presented, relating to the usage of computational power and memory as well as inference time for different models. Finally, in Sect. 4.3, the results of applying knowledge distillation to the models mentioned in Sect. 3.5.2 are presented.

4.1 ML metrics

In this section, the results obtained from training the ML models are presented. In Sect. 4.1.2, the results of training different YOLOv5 models, in regards to precision, recall, and mean average precision (both mAP.5 and mAP0.5:0.95), are presented. Sect. 4.1.1, presents the results of training the classifiers, both ShuffleNetV1 and the simple CNN model, in regards to accuracy.

4.1.1 Ml metrics of the classifiers

This section presents the results obtained from the training of ShuffleNetV1 and the simple CNN model on the Kaggle dataset mentioned in Sect. 3.1. The results are in regards to the accuracy obtained from three datasets, i.e. training, validation, and testing. One result for ShuffleNetV1 can be seen in Table 4.1 and for the simple model in Table 4.2.

Set	Training	Validation	Testing
Accuracy	0.9338	0.8583	0.8586

Table 4.1: ShuffleNetV1 results in regards to accuracy after 2 epochs. The results show the accuracy on the three different datasets, i.e. training, validation and testing.

4.1.2 ML metrics of YOLOv5

The results of the training of different YOLOv5 models regarding several machine learning metrics are shown in Table 4.3. The difference in size between the different YOLOv5 networks is high compared to the difference between all ML metrics. The

Set	Training	Validation	Testing
Accuracy	0.7460	0.7471	0.7468

Table 4.2: Simple CNN model results in regards to accuracy after 2 epochs. The results show the accuracy of the three different datasets, i.e. training, validation, and testing.

focus of this project is the optimization and efficient utilization of resources, therefore, YOLOv5n was used to further generate the results in Sect. 4.2 and Sect 4.3.

Model	**P**	**R**	**mAP@.5**	**mAP@.5:.95**	**parameters (M)**
yolov5n pt	0.789	0.701	0.761	0.490	1.77
yolov5n	0.783	0.666	0.731	0.467	1.77
yolov5s pt	0.836	0.731	0.810	0.543	7.02
yolov5m pt	0.849	0.787	0.859	0.602	20.89
yolov5l pt	0.869	0.814	0.879	0.638	46.13

Table 4.3: Models with different sizes tested on a test set consisting of 28,884 images and 12,416 boat labels. All models are trained for exactly 80 epochs to facilitate a direct comparison. "pt" indicates a pre-trained model using the COCO dataset, obtained from the YOLOv5 repository.

The results YOLOv5n and YOLOv5s when built in two different frameworks, and their effect on all the metrics, are shown in Table 4.4. The results show that the conversion does affect the metrics but not notably.

Model	**Framework**	**P**	**R**	**mAP@.5**	**mAP@.5:.95**
yolov5n	pytorch	0.789	0.701	0.761	0.490
yolov5n	tflite	0.798	0.689	0.750	0.482
yolov5s	pytorch	0.836	0.731	0.810	0.543
yolov5s	tflite	0.823	0.737	0.798	0.532

Table 4.4: The comparison of two models before and after the conversion to TFlite. The two models are evaluated on a test set consisting of 28,884 images and 12,416 boat labels. Both models are trained for 80 epochs and have pre-trained weights obtained from the YOLOv5 repository.

4.2 Resource usage

This section presents the results obtained from running the different models on the target hardware. Sect. 4.2.1 goes through the results obtained from running YOLOv5n, while Sect 4.2.2 goes through these results obtained from the classifiers. The results are presented in regards to inference time and memory usage.

4.2.1 Resource usage using YOLOv5n

This section presents the results obtained from running YOLOv5n models on the target hardware in regards to inference time and memory usage.

4.2.1.1 Memory management and conversion of datatypes

The memory management methods and conversions of datatypes were analyzed according to the three methods described in Table 3.4. Table 4.5 shows the measured memory and inference time.

Method	time (min)	CPU memory (GB)
1	5.3	1.30
2	5.3	0.47
3	5.6	0.45

Table 4.5: Comparison of different methods of memory and data management described in Tab 3.4 with input size 1,440×1,440.

Method A, converting the image from `int8` to `float32` directly after loading it, killed the process since it run out of RAM. However, turning on swap memory made it possible to obtain a measurement. As seen in Table 4.5, a reduction of approximately 0.8 GB was obtained by simply using Method B instead, where only the tiles were converted to `float32`. Time was not affected when using Method B compared to Method A.

Using `memcpy` function in C++, to copy over values to the input tensor (Method B) instead of copying each pixel separately (Method C) reduced the time by approximately 16 seconds on average. The memory usage for `memcpy` (Method B) was almost the same as for Method C.

Testing the memory usage using the different methods was only done using the CPU since no model inference was done during this step. All other tests were measured using Method B due to the above results, except for the batch sizes that were measured using Method C, which is further explained in Sect. 5.3.1.2.

4.2.1.2 Breakdown of inference process

Table 4.6 compares the time used for pre-processing and inference over different input sizes. The pre-processing step, including loading and tiling up the image, always

takes the same amount of time and is negligible compared to the actual inference time. Looking further into the details of the inference time in Table 4.7, it is evident that the times measured for loading the input tensor and fetching the output are also negligible compared to the detection step.

Fig. 4.1 shows the measurements of memory usage during a run using GPU and CPU, both with the image loaded and without. The CPU starts the inference instantly since almost no time is used for the pre-processing steps (also seen in Table 4.6). Therefore it has almost constant use of memory throughout the run. When running on the GPU on the other hand, it first creates the GPU delegate (around 90% of the run in Fig. 4.1). After the initialization, the inference begins and lasts a much shorter time than the CPU (shown in Table 4.6). The same figure does also show a difference of approximately 300 MB in virtual memory and resident size between loading the input image and not loading the input image for both the GPU and the CPU. Lastly, there are no differences in the used VRAM for loading an image and not loading an image, as seen in Fig. 4.2.

Input size (pixels)	**Pre-processing (min)**	**Inference (min)**
192	0.031	5.52
288	0.031	5.31
480	0.031	5.32
768	0.031	6.12
1440	0.031	5.35
2080	0.031	5.73

(a) CPU

Input size (pixels)	**Pre-processing (min)**	**Inference (min)**
192	0.031	6.71
288	0.031	4.79
480	0.031	2.47
768	0.031	2.66
1440	0.031	2.13
2080	0.031	2.43

(b) GPU

Table 4.6: Comparing the pre-processing time and inference time for YOLOv5n, using different input sizes, GPU or CPU, and `float16` as the weight precision.

Input size (pixels)	**Load input to tensor (s)**	**Detect (s)**	**Fetching output (s)**	**Detect once (s)**
192	1.147	401.6	0.015	0.143
288	1.269	285.9	0.007	0.234
480	1.211	147.3	0.005	0.334
768	1.342	158.5	0.002	0.809
1440	1.139	126.6	0.001	2.584
2080	3.078	142.8	0.001	5.712

Table 4.7: YOLOv5n for different input-sizes with detailed timing using GPU with the precision of weights in `float16`. All columns are measured for the complete image except for the last column which measures the processing time for one tile.

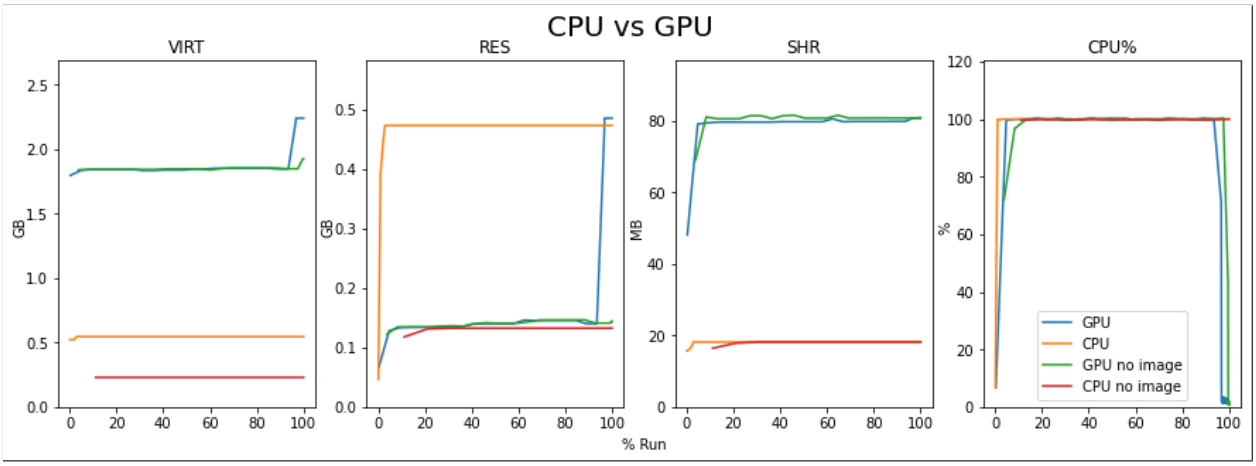


Figure 4.1: Memory used during inference for input size 1,440×1,440 pixels to the network, using GPU and CPU, with and without loading the input image. The changes in the horizontal axis must be taken with a grain of salt since 100% of a run corresponds to the entire measurement time and the number of tiles processed is not the same during the run with image and the run without image.

4.2.1.3 Different input sizes

The results of using different input sizes are shown in Fig. 4.3. The inference time when using the GPU decreases as the network's input size increases, while when using the CPU the time is almost constant. Fig. 4.3 also brings forward a connection between input sizes and the amount of overlap in the last tile when pre-processing the image (further explained in discussion, Sect. 5.3.2.2). This overlap is especially reflected in the inference time when using the CPU. The least time-consuming models are 1152, 1440, 1536, and 3360 using the GPU. The max size for the tiles was furthermore measured at 3712×3712 and failed at 3776×3776.

Fig. 4.4 shows the relation between input size and memory usage. The slope for the memory used (VIRT and RES) when running on the CPU is larger than the slope when using the GPU. A trade-off between the time taken and the VRAM used by the GPU is then shown in Fig. 4.5 and Table 4.8. It is visible that there exists a lower bound of approximately 2.1 - 2.5 min starting with an input size of 480×480.

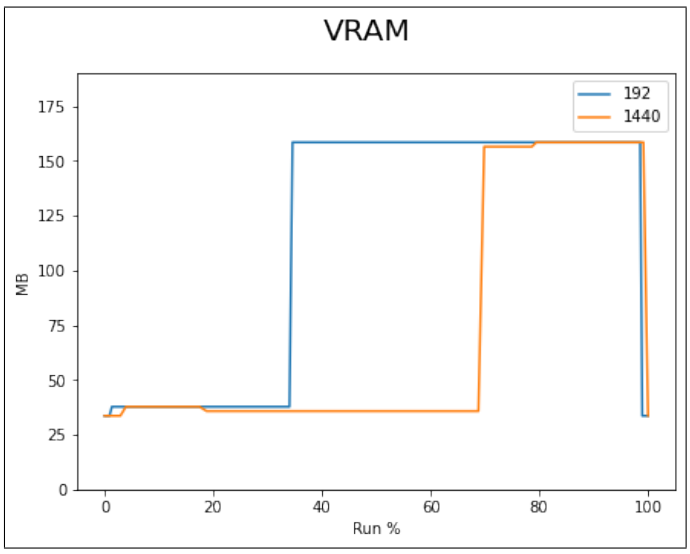


Figure 4.2: VRAM used by the GPU during inference for input size 1,440×1,440 pixels to the network, with and without loading the input image. The changes along the horizontal axis must be taken with a grain of salt since 100% of a run corresponds to the entire measurement time and the number of tiles processed is not the same during the run with image and the run without image.

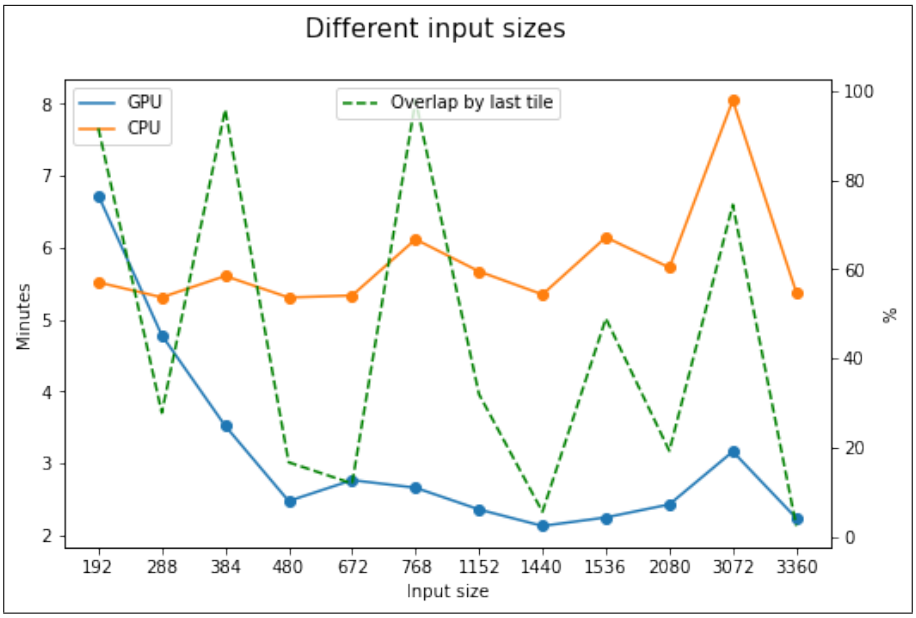


Figure 4.3: Time it takes for the program to load an image, tile it, and process all tiles for YOLOv5n models with different input sizes using GPU and CPU. The yellow dashed line shows the overlap needed in percent for the last tiles to the right and bottom.

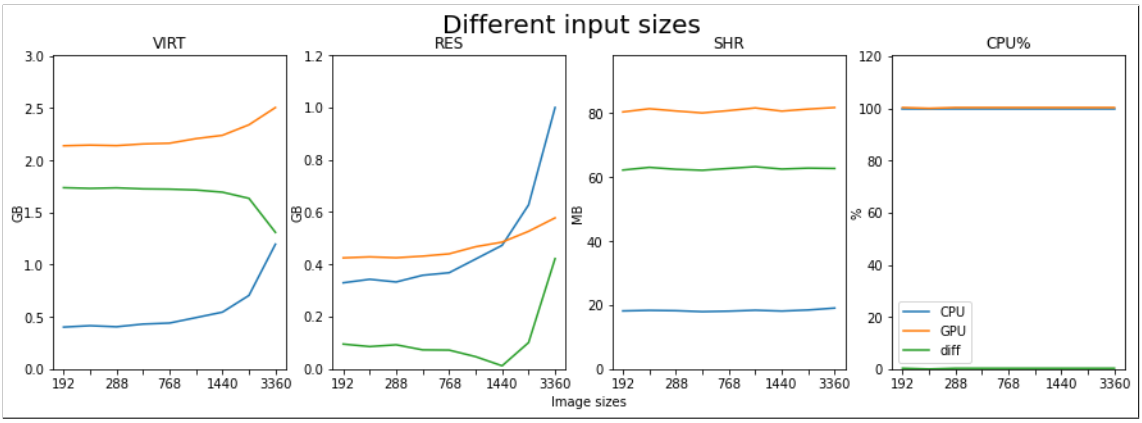


Figure 4.4: Memory used during inference for different input sizes using GPU and CPU. The green graph is showing the differences between the CPU and GPU

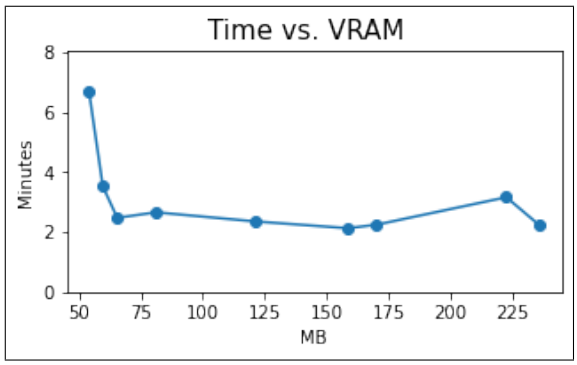


Figure 4.5: A trade-off between VRAM used by the GPU and inference time for the following network input sizes shown in Table 4.8.

Input size	**Time (min)**	**VRAM** (MB)
192	6.689	54.06
384	3.532	59.70
480	2.483	65.50
768	2.663	81.10
1152	2.368	120.82
1440	2.137	158.46
1536	2.254	170.10
3072	3.173	222.32
3360	2.287	235.50

Table 4.8: Values used in Fig. 4.5

Input size	**Inference `float32` (min)**	**Inference `float16` (min)**	**Inference `int8` (min)**
192	6.5	6.5	7.0
384	3.3	3.3	3.5
768	2.3	2.3	2.5
1152	2.2	2.2	2.4
1536	2.0	2.0	2.1

Table 4.9: Inference time for different precision of the weights using GPU for YOLOv5n.

4.2.1.4 Quantization

Using quantization does not decrease the inference time, but does decrease the size of the weight file drastically. Table 4.9 shows the differences in time when using weights in `float16`, `float32`, and `int8`. There is no difference in time for `float16` and `float32`, but `int8` does slightly increase in time. Furthermore, in Table 4.10 it is shown that saving the precision of the network in `float16` instead of `float32` and `int8` instead of `float16` reduces the file size by approximately half the size. The size of the input tensor for the model does however not affect the size of the weight file significantly.

Framework	Model	Input size	Precision	File size (MB)
tflite	yolov5s	768	`fp32`	26.91
		768	`fp16`	13.52
tflite	yolov5n	1152	`fp32`	6.99
		768	`fp32`	6.88
		32	`fp32`	6.78
		1152	`fp16`	3.56
		768	`fp16`	3.50
		32	`fp16`	3.45
		1152	`int8`	1.98
		768	`int8`	1.95
		32	`int8`	1.93

Table 4.10: Storage space taken by the model weights for different input sizes and precision of model weights.

4.2.1.5 Multithreading

Fig. 4.6 shows that four or five threads are the optimal number for all input sizes using the CPU. The best obtained time given in Table 4.11 shows that an input size of 3,360×3,360 pixels with nine threads is the best. However, all the measured times in the table are similar and around 2.2 min. These times show a similar pattern to the results of the input sizes (Sect. 4.2.1.3), which seem to indicate the existence of a lower bound, slightly above 2 min for the inference time. The time stops decreasing already around four threads.

Fig. 4.7 shows the memory measurements of using different numbers of threads. There are no differences in resident size (RAM), but the virtual memory increases. The more threads that are used, the higher percentage of the CPU processing power is used. The CPU power does however stop increasing after four threads. No measures were done for the GPU since multithreading is not supported by the GPU.

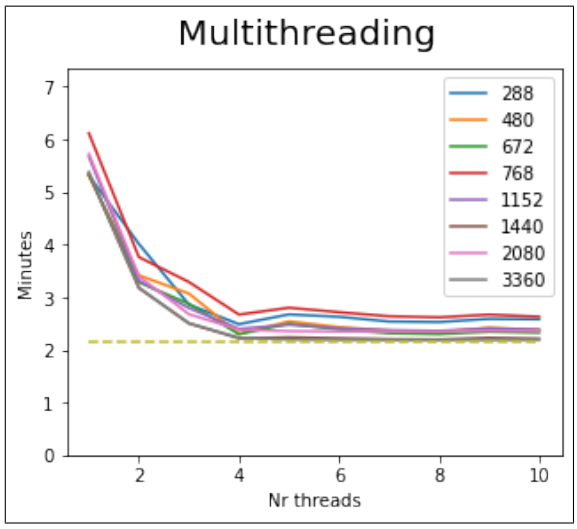


Figure 4.6: Inference time using CPU with different input sizes and multithreading. The dashed purple line shows the best time previously archived using an input size of 1,440×1,440 and batch size one at 2.14 minutes. The best values are shown in Table 4.11.

Input size	Threads	Time
3360	9	2.187
3360	6	2.188
3360	7	2.190
3360	8	2.190
3360	10	2.190
3360	5	2.197
1440	8	2.198
1440	7	2.204
1440	10	2.213
1440	4	2.219
1440	6	2.221

Table 4.11: Best values in time from Fig. 4.6

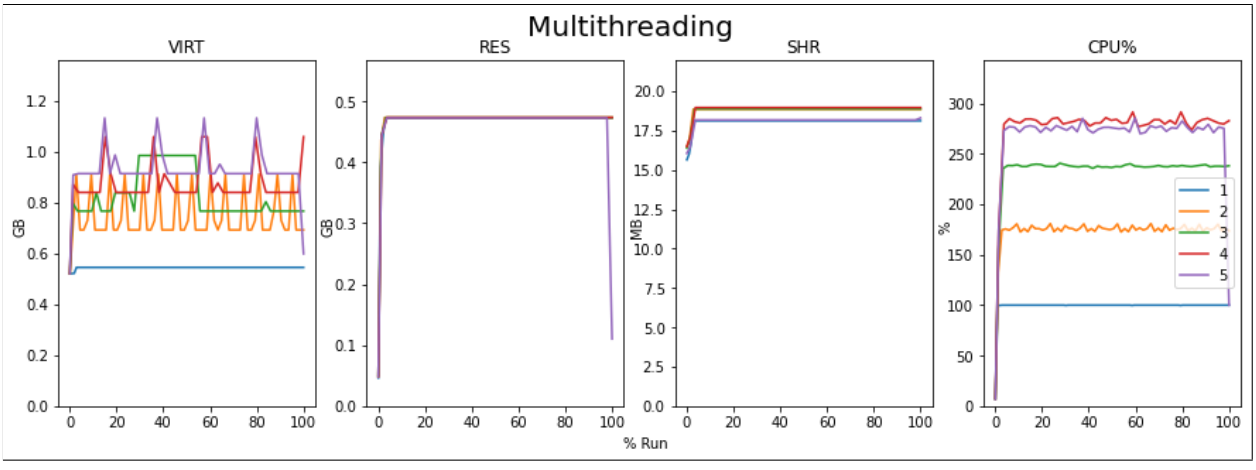


Figure 4.7: Memory and CPU usage for one to five threads. The horizontal axis represents the fraction of the time a process was running.

4.2.1.6 Batch size

Increasing the batch size was not possible with Method B where the C++ function `memcpy` was used to copy the values into the input tensor. Therefore, Method C was used where the pixel values were copied over one by one. Larger batch sizes than one were not tested for the CPU since computations on the CPU with larger inputs did not improve the time, as seen in Sect. 4.2.1.3.

A problem with larger batch sizes was that the input sizes were only able to be tested up to 768×768 since 928×928 failed. Different batch sizes were therefore tested with smaller input sizes and the results are shown in Fig. 4.8. The results show that increasing the batch size does decrease the inference time, but that the slope fades towards both larger batch sizes and larger input sizes.

The VRAM used by the process with increasing batch size is shown in Fig. 4.9. The increase appears to be linear and higher with larger input sizes.

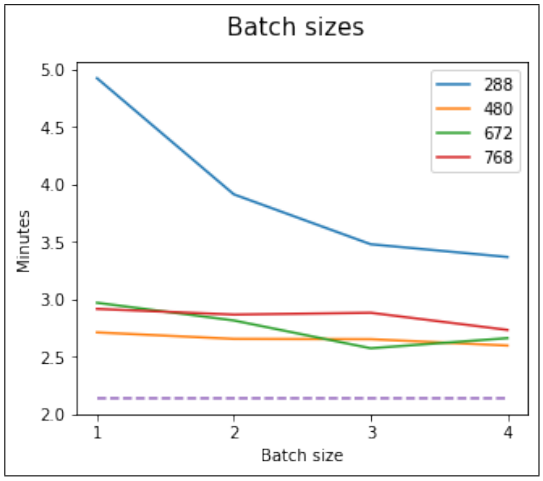


Figure 4.8: Time for different input sizes using different batch sizes and the GPU. The dashed purple line shows the best time previously archived using an input size of 1,440×1,440 and batch size one at 2.14 min.

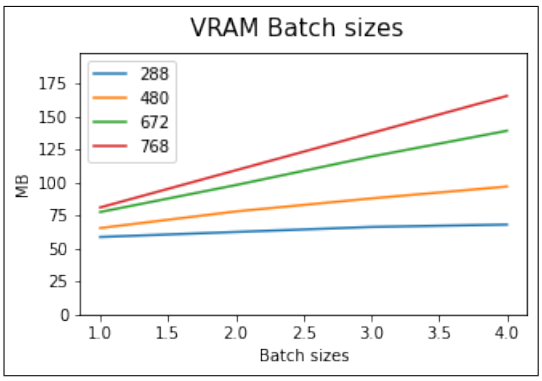


Figure 4.9: VRAM used during inference for different input sizes as the batch size increases.

4.2.2 Resource usage using the classifiers

This section presents the resource usage results for the classifiers. The results for ShuffleNet are presented in Sect. 4.2.2.1, but since it was not successful to run on the satellite computer, the simple CNN model is used for measurements instead. The detailed measurements of the simple CNN model are therefore shown in Sect. 4.2.2.2 and a comparison of YOLOv5n and the simple CNN model is presented in Sect. 4.2.3.

4.2.2.1 Resource usage using ShuffleNet

ShuffleNetV1 uses grouped convolutions which was not, at the time of writing this report, supported by TFLite. Converting the model to TFLite format was possible, but running an invoke using that model was not supported. Therefore, only the simple CNN model was used for running inference on the target hardware.

4.2.2.2 Resource usage using the simple CNN model

Results of the inference time for the pre-processing and inference steps, using different input sizes, are presented in Table 4.12. Furthermore, a breakdown of the inference time is shown in Table 4.13. All surrounding steps take much less time compared to the detection step, which shows a similar pattern to the breakdown for the YOLOv5n (Sect. 4.2.1.2). The model is however smaller and so is the time it takes for the detection step. Therefore, in this case, the loading of the input tensor step does make a difference. However, the time for loading the input tensor is almost constant.

Furthermore, multithreading did decrease the inference time for the CPU but did not improve beyond the best measured time for the GPU, which is seen in Table 4.10. Fig. 4.11 shows that using a larger batch size than one does not decrease the inference time at all.

Input size (pixels)	**Pre-processing (min)**	**Inference (min)**
192	0.031	1.07
288	0.031	1.02
480	0.031	1.02
768	0.031	1.16
1440	0.031	1.02

(a) CPU

Input size (pixels)	**Pre-processing (min)**	**Inference (min)**
192	0.039	0.39
288	0.034	0.28
480	0.034	0.28
768	0.034	0.30
1440	0.034	0.27

(b) GPU

Table 4.12: Comparing the pre-processing time and inference time for the simple CNN model, using different input sizes, GPU or CPU, and `float16` as the weight precision.

Input size (pixels)	Load input to tensor (s)	Detect (s)	Fetching output (s)	Detect once (s)
192	0.9	21.9	0.005	0.01
288	1.1	15.5	0.002	0.04
480	1.2	15.5	0.002	0.04
768	1.2	16.7	0.001	0.09
1440	1.1	15.1	0.000	0.31

Table 4.13: The simple CNN model for different input sizes with detailed timing using GPU with the precision of weights in `float16`. All columns are measured for the complete image except for the last column which measures the processing time for one tile.

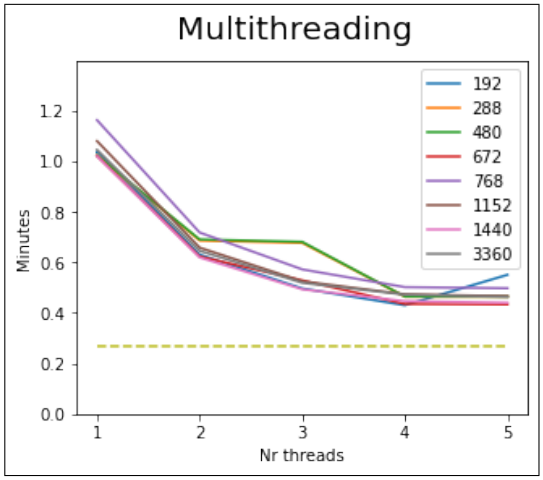


Figure 4.10: Comparing the inference time for different input sizes to the model with a different number of threads using the simple CNN model. The light green dashed line is the lowest inference time measured with the GPU (1,440×1,440 as input size at 0.27 min).

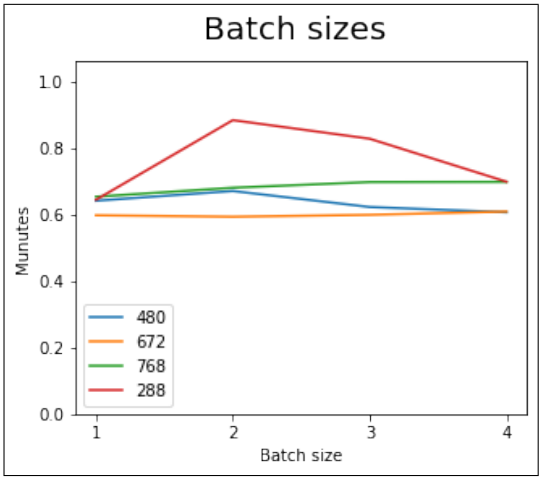


Figure 4.11: Comparing the inference time for different input sizes to the model with different batch sizes using the simple CNN model. The light green dashed line is the lowest inference time measured with the GPU (1,440×1,440 as input size at 0.27 min).

4.2.3 Simple model vs. YOLOv5n

An inference using the classifier for an image of 10,000×10,000 pixels resulted in about 16 s to 26 s for all tile sizes compared to 128 s when using the YOLOv5n model with 1440×1440 as input size on the GPU, which was the fastest YOLOv5 model. On the CPU, it took 1 min to 1.2 min for the classifier versus 5.3 min to 6.0 min for YOLOv5n. There was however not much of a difference in memory usage, as shown in Fig. 4.12 and Fig. 4.13.

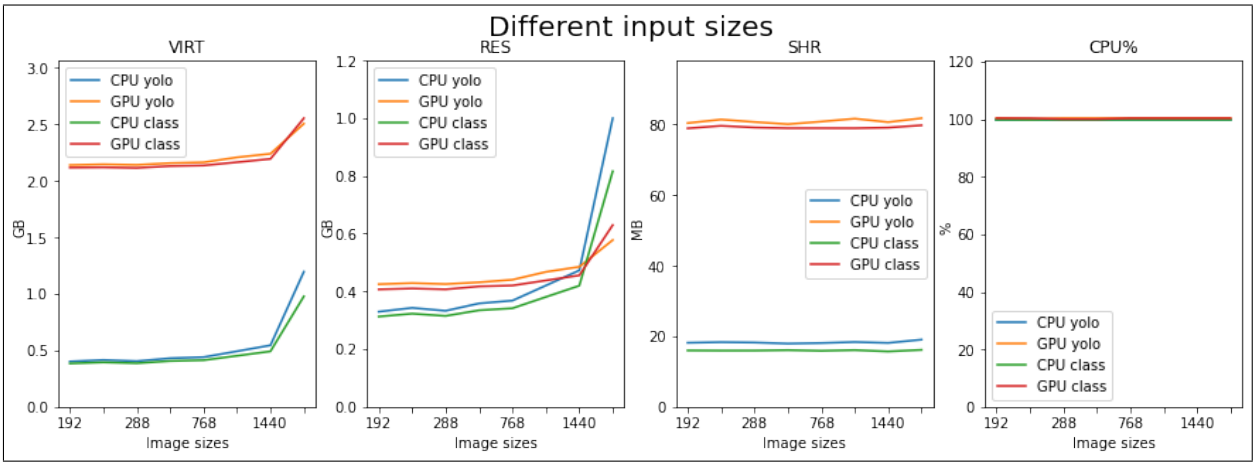


Figure 4.12: Comparing memory consumption for different input sizes for YOLOv5n and the simple CNN model.

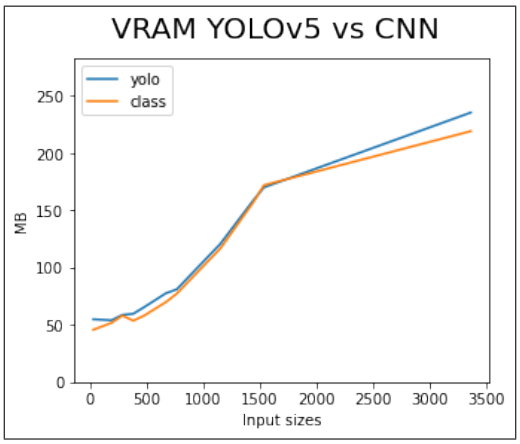


Figure 4.13: Comparing VRAM for different input sizes for YOLOv5n and the simple CNN model.

4.3 Experimental results of knowledge distillation

This section presents the results from running knowledge distillation on the object detector (see Sect. 4.3.2) and the classifier (see Sect. 4.3.1).

4.3.1 Results of knowledge distillation on the classifiers

The simple CNN model was trained using knowledge distillation for two epochs, in order to facilitate a direct comparison with the results of training the simple CNN model without KD. The results of using KD can be found in Table 4.14, which shows a 12% increase in the accuracy.

Set	Training	Validation	Testing
Accuracy (before KD)	0.7460	0.7471	0.7468
Accuracy (after KD)	0.8706	0.8619	0.8651

Table 4.14: The simple CNN model's results before and after using knowledge distillation from a pre-trained ShuffleNetV1, in regards to accuracy after two epochs.

4.3.2 Results of knowledge distillation on YOLOv5

Fig. 4.14 shows the results obtained by comparing the outcome when training a YOLOv5n normally to the outcome of training a YOLOv5n using knowledge distillation from a YOLOv5l. The tests include both using pre-trained weights for YOLOv5n and without. The precision was not affected by KD, while recall and the two mAP measures had all similar affections. Soft labels resulted in the best improvements followed by knowledge distillation using the first three feature maps while using only one feature map even worsened the training.

Fig. 4.15 shows the results of running knowledge distillation on the COCO dataset. These tests were including longer training since detecting multiple classes is a more complex problem and require more time to train. The results, in this case, show that knowledge distillation does not affect the training at all.

Appendix A.5 shows the complete set of results obtained in this project.

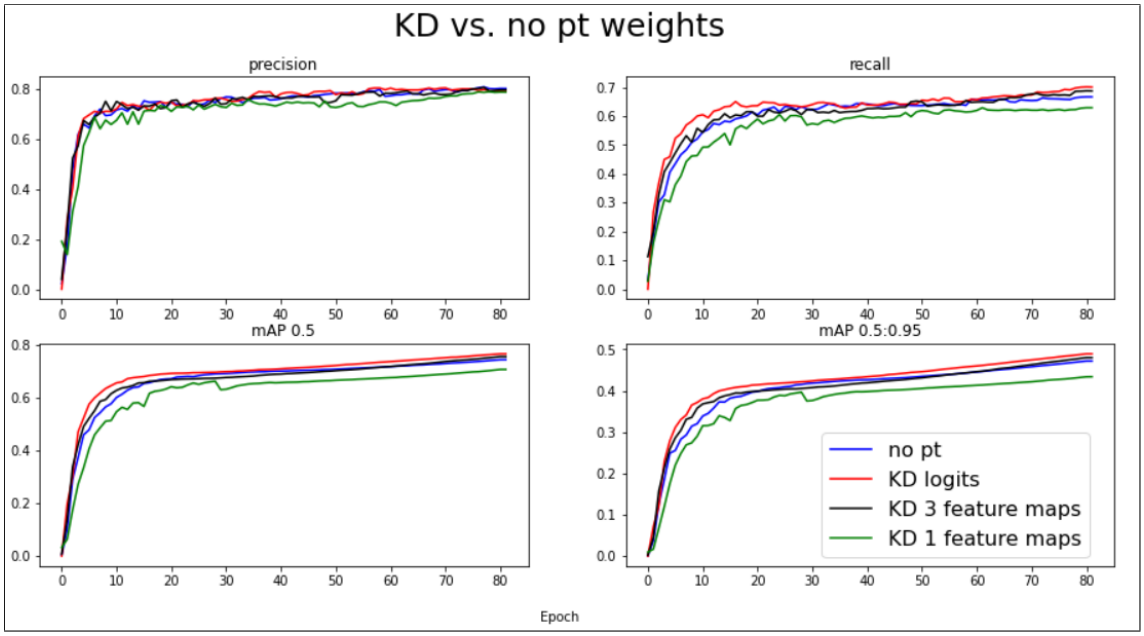


Figure 4.14: A comparison of the training metrics when running knowledge distillation using the logits and three feature maps from a YOLOv5l to train a YOLOv5n. The training was conducted using the Kaggle dataset mentioned in Sect 3.1.

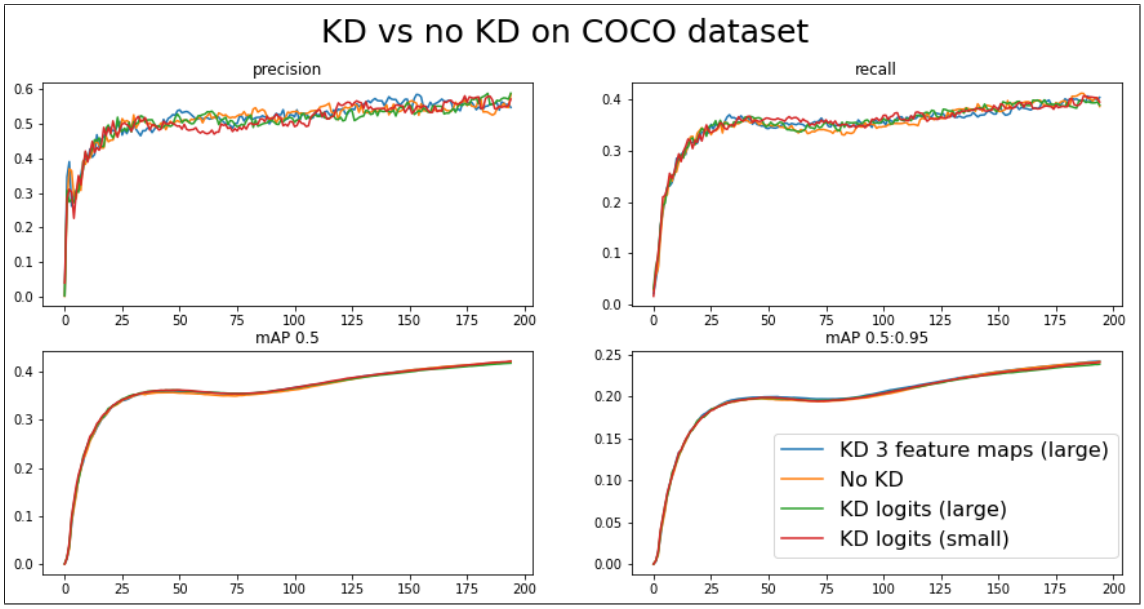


Figure 4.15: A comparison of the training metrics when running knowledge distillation using both feature maps and logits from a YOLOv5l to train a YOLOv5n on the COCO dataset for a higher number of epochs.

5
Discussion

The choice of dataset and models are discussed in Sect. 5.1 and 5.2. Thereafter, the results obtained from knowledge distillation are discussed in Sect 5.4. Both Sect. 5.5.1 and 5.5.2 contains discussions on the results obtained from the testings on the target computer. The applicability of ML in space is discussed in Sect. 5.6, while ethical and sustainability aspects are discussed in Sect. 5.7. Finally, possible extensions and future work is discussed in Sect. 5.8.

5.1 Choice of dataset

The dataset used was the Kaggle Airbus challenge dataset. While the dataset is great for training ML models, it does not reflect the nature of the process of using an ML model in space. Images taken using satellite cameras are not as neatly processed as the images found in the Kaggle dataset. For example, as mentioned in Sect. 2.1, the Sentinel-2 camera has at best a resolution of 10 m, which can make it difficult to detect vessels, compared to the images found in the Kaggle dataset (see Fig. 3.1). Consequently, since different satellite cameras will produce different images, the performance of an ML model in terms of accuracy cannot be generalized to different datasets. However, by focusing on the models' resource usage, and examining the relative differences before and after optimization, a general understanding of such models' behavior can be developed. Therefore, for such ML models to be applied in a live system, a different dataset, consisting of images taken with the camera on the target system (or similar) needs to be created and a new model needs to be trained on that dataset.

5.2 Choice of model

This section discusses the ideas behind the choice of the models used in this project. In addition, it names a few important points when choosing a model for embedded systems.

5.2.1 Choice of classifier

ShuffleNetV1 was chosen because it is a network designed for mobile and embedded systems. The network is based on grouped convolution, which is faster than normal convolution. The network was also chosen because of its relatively low number of FLOP. However, FLOP is an indirect metric that does not translate directly

to computational speed on specific hardware. This is something that needs to be taken into account when designing a model that is required to run on small mobile systems. Different factors such as memory access cost, degree of parallelism, and even element-wise operations can influence the actual time it takes to run inference on specific hardware. Also, it is important to make sure that the operations executed by the network are supported by the target framework.

5.2.2 Choice of object detector

YOLOv5 was chosen because of the simplicity of implementation and because it is created in the widely used and user–friendly framework, Pytorch. Moreover, YOLOv5 has proven its efficiency in mobile and embedded systems, which is more important in this project than other metrics such as accuracy. YOLOv5 is however a complex network that targets the detection of multiple classes with possibly a lot of background noise, which is not the case in this project. Satellite images over the sea often consist of mostly water where vessels are the only class. Therefore, other simpler models might be more efficient in resource usage. On the other hand, there might be more information about the vessel that YOLOv5 provides that could be interesting. An example of such information could be the size of the vessel. Since only larger vessels are legally required to report to AIS, the information about the vessel size is crucial to determine if the vessel should exist on the AIS or not. Moreover, YOLOv5 is good at detecting other classes, which can be beneficial when utilizing a satellite. For example, another application in the same satellite could be to detect airplanes. The flexibility of YOLOv5 allows for more information to be captured at once, which is much more resource-efficient than using several models.

The comparison of different YOLOv5 sizes in Table 4.3 shows that YOLOv5s is almost four times bigger than YOLOv5n in the number of parameters and does not show much better results in terms of ML metrics. The same goes for even bigger networks where YOLOv5m is three times bigger than YOLOv5s and YOLOv5l is more than two times bigger than YOLOv5m. YOLOv5n already gives relatively good results (0.789 in precision and 0.701 in recall), compared to its size and the size and results of other tested YOLOv5 models. Therefore, to reduce resource usage, YOLOv5n was chosen. However, the choice might not be as straightforward when it comes to more complex use cases, because the trade-off between size and performance of the different models would be larger. For example, in an application with multiple classes where accuracy is crucial, choosing YOLOv5n might not be the best course of action.

No extensive hyper-parameter tuning or modifications of the networks were made during testing since the focus of the project was to optimize resource usage. These limitations could have affected the results for precision negatively since the default settings might not be optimal for the used dataset. Changes in the model architectures could also affect both precision and resource usage.

5.3 Performance on the hardware

The results obtained from the different optimization methods are discussed in this section. The description of the methods are found in Sect. 3.5.1 and the results in Sect. 4.2. All sections include discussions of the results for YOLOv5n, except for Sect. 5.3.6 which discusses the results of the classifier.

5.3.1 Effects of code structure

The most basic optimizations such as structuring the code efficiently and allocating only the necessary memory for storing variables could decrease memory usage significantly. This section describes some of the essential steps to decrease memory consumption.

5.3.1.1 Converting image vs. converting tile to `float32`

It is clear that converting only the tile to `float32` (Method B) and not the complete image (Method A), is preferable when using a target device with restricted RAM, since the process crashed when not using any swap memory (see Sect. 4.2.1.1). However, the default state of swap memory is on and would most probably be used during production. It is, therefore, possible to use more memory than the available RAM during inference, but this is not preferable since the swap memory uses the hard disk which is slower than the RAM. Converting the entire image to `float32` does not provide any advantages time-wise and only has the drawback of filling up the memory.

5.3.1.2 Using `memcpy`

Using `memcpy` (Method B) instead of copying all values one by one to the input tensor (Method C) reduced the inference time by 16 seconds (see Sect. 4.2.1.1). These 16 seconds might not be significant in comparison to the two minutes it takes. However, it can accumulate to a big amount of time saved when processing a large number of images. It is also shown in Fig. 4.8 that the difference in time could be larger with other input sizes. On the other hand, Method B did not function for all batch sizes. It was discovered that `memcpy` requires that the memory allocation for the input data is contiguous, which is not the case when using `memcpy` on each tile in larger batch sizes. However, it would be advantageous to solve this issue so that `memcpy` can be used on bigger batch sizes.

5.3.1.3 Memory used by the image

Looking at the breakdown of the inference process in Sect. 4.2.1.2, a difference of 300 MB was measured when loading the input image versus not loading it, which matches the size of an input image plus the size of one tile. The size of an input image is 284 MB (10,000×10,000 image with three channels in `int8`) and the size of a single tile is 23MB (a 1440×1440 tile with three channels in `float32`), which equals approximately 300 MB. These results show that a large portion of the memory usage

is allocated for the input image. On the other hand, the GPU memory is allocated through the GPU delegate, and is therefore independent from the memory allocated for the image, which explains why the GPU memory remains constant both when loading and not loading an image.

5.3.1.4 How TFLite manages memory

There were several steps to allocate memory for all three methods, briefly mentioned in Sect. 3.5.1.1. The complete image was first loaded. Thereafter, a tile was extracted from the image and stored into a variable, which was then copied into the input tensor. Lastly, after calling the actual detection function in TFLite (`interpreter->invoke()`), the values are copied to the memory space of the input of the model, which is allocated in the GPU. The last part is shown by the process breakdown shown in Sect. 4.2.1.2, where the actual detection takes much more time than the surrounding processes. Furthermore, it is shown that using the GPU takes more memory than using the CPU. The increased memory usage of the GPU is sensible since the values must be copied to the memory of a completely separate processing unit, while the CPU is able to contain the values in the same memory space.

5.3.1.5 Better memory managements

One way of making better use of the memory would be to free the memory as soon as a tile is used as input to the network. Freeing up memory in this way will however require some low-level programming where it is possible to be closer to the hardware. Perhaps forgo the usage of some off-the-shelf libraries towards implementing a customized pipeline. Moreover, the model's input size should be customized in relation to the image size see (5.3.1.5).

$$\text{mod} \left(\frac{\text{image size}}{\text{input size}} \right) \approx 0.$$

Another method for preventing RAM overload is to use a separate program to tile up the image and save it locally on the disk. This can potentially slow down the process since the cost of loading the image from the disk is higher than RAM, but it can save a bit of memory. Moreover, this method allows for removing the loaded image from the disk as soon as it has been used in the model, which can be beneficial in some cases. For example, if the raw image from the camera is not in `int8` format, then it would be beneficial to have such a program that does the tiling in a separate step.

5.3.2 Effects of using different input sizes

Using different input sizes can affect the inference time significantly when using the GPU, which is shown in Sect. 4.2.1.3. The GPU has an efficient architecture for matrix multiplications, which decreases the inference time when increasing multi-dimensional input sizes. The CPU on the contrary does not have this ability and therefore the measurements are somewhat constant or even increasing. The timing

measures further revealed a connection between the input size and the overlap given by the last tile, which is further explained in (Sect. 5.3.2.2).

By further analyzing the memory usage, it was visible that the memory usage for the CPU is increasing at a higher rate than the memory usage for the GPU when measuring the resident size and virtual memory. This is seen in Fig. 4.4. The lower increase in memory usage by the GPU exists because the process of inference is performed in the GPU and is not included in the CPU measurements. It is instead visible by measuring the VRAM. The differences between the CPU and GPU are also high in Fig. 4.4 since it requires more memory to use two processing units than only the CPU.

5.3.2.1 Choosing an input size

In the aspect of increasing the input size, it could be concluded that the GPU is more efficient to use than the CPU. Which of the input sizes is most suitable when using the GPU is dependent on the most critical resource. It was discovered in the trade-off between time and VRAM (Fig. 4.5) that there was a lower bound in time (around 2.1 min to 2.5 min), starting with an input size of 480×480 pixels. Therefore, to get the best trade-off between time and memory, the input size of 480×480 pixels should be used. On the other hand, if time is more critical than memory usage, it would be more sensible to use the fastest model with an input size of 1,440×1,440.

5.3.2.2 Avoiding processing unnecessary pixels

Having an image size that is not divisible by the tile size will result in a remainder of pixels when tiling the image. The current implementation of tiling tackles the remaining pixels by simply counting from the edge, instead of starting where the previous tile ended. The current implementation results in unnecessary pixels that are processed, and an illustration of the problem is shown in Fig. 5.1. The problem is reflected by the matching peaks in the measured inference times and the percentage of overlap that the last tiles have, especially when using the CPU. This can be seen in Fig. 4.3. Therefore it is important to match the input size of the model to the image size so that the overlap in the last tile is as minimal as possible, i.e.

$$\mod \left(\frac{\text{image size}}{\text{input size}} \right) \approx 0.$$

5.3.3 Quantizing the weights

The results for quantization in Sect. 4.2.1.4 show that neither having the precision of the model weights in `int8` nor `float16` decreased the time of inference, which was contrary to the expectations. The model with a precision of `float16` gave the same time results as `float32` and `int8` even exceeded the inference time for `float32`. It was discovered that these results were because quantization was not accelerated by

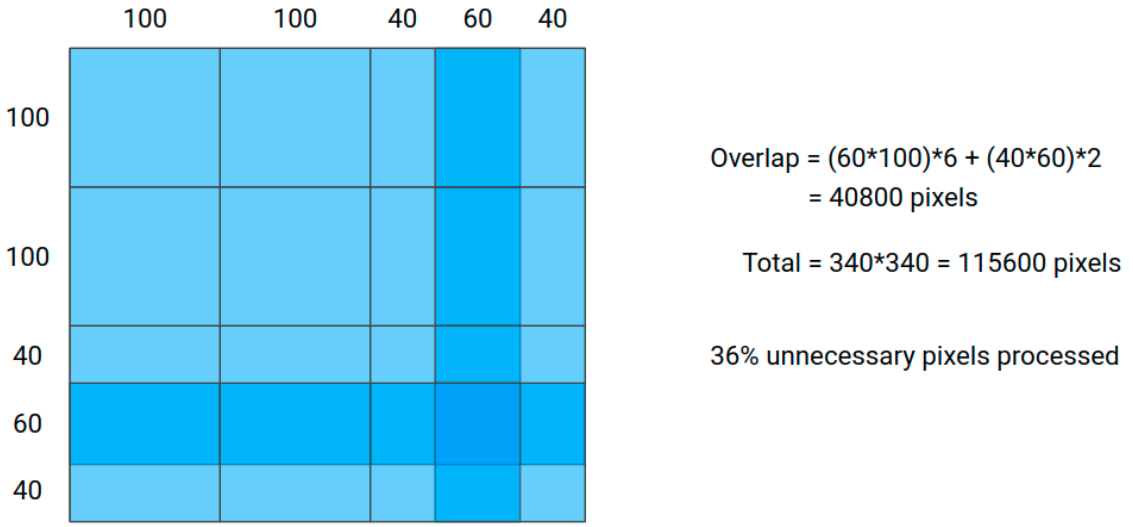


Figure 5.1: An example using a 340×340 pixels image with a tile size of 100×100 pixels. This illustrates the overlap created by the last processed tiles on each row and column.

the target computer, nor was it supported by the TFLite interpreter.

TFLite supports exporting models weights to `int8` format, but does not support it during the inference step. The environment works in `float32`. When running an `int8`-model, TFLite first dequantizes the input to `float32` and then quantizes the results back to `int8`. This is also the reason why the inference time increases when using `int8`. Using a model containing `float16` weights, the computation still occurs with `float32`. It is however possible to command the interpreter to do computations in `float16`, but it does not affect the time.

A positive outcome of quantizing the weights is that it decreases the size of the model, which is shown in Table 4.10 where the file sizes of different input sizes and datatypes are presented. All levels of quantization (from `float32` to `float16` and from `float16` to `int8`) decreased the file size of the saved weights by half. Judging by the increase of inference time for `int8`-models, the method of quantization to `int8` is not optimal to use in the target computer. However, `float16` is suitable, it decreases the file size significantly, and does not affect the inference time.

5.3.4 Multiple threads in the CPU

Having multiple threads can be a way of speeding up the process if the computation is done using the CPU. The results in Sect. 4.2.1.5 show that the inference time is decreasing as the number of threads increases. The optimal number of threads seems to be four threads, as the decrease in time is negligible with an increasing number of threads. The best time obtained using four threads is 2.2 min. Moreover, the CPU power similarly stops increasing after four threads. Resident size has the same memory usage for all number of threads, but both the virtual memory and the shared memory increase when using more threads than one. The increase in virtual

and shared memories occurs when using multiple processes that require communication between each other, which is the case in multithreading.

The results in Sect. 4.2.1.5 further show that the best time obtained was 2.187 min with nine threads on a 3,360×3,360 pixels input size, which almost reaches the same speed as using the GPU with 1,440×1,440 (2.137 min). The CPU could therefore be used instead of the GPU if necessary for this particular hardware with almost the same performance. Less memory is also used by using the CPU, but since the process is using the maximum processing power of the CPU, no other processes would be able to run. The satellite computer ix5 does furthermore have four CPU cores which might explain why the time stops decreasing after four or five threads.

5.3.5 Increasing the batch size

The results for different batch sizes in Sect. 4.2.1.6 show a decrease in time when using larger batch sizes but without any significant improvements. Larger input sizes crashed the program due to lack of memory, while the smaller input sizes did not reach enough low inference time to match the time of other optimization tests.

Four measurements were taken of different input sizes, where the smallest input size of 280×280 had the best trade-off between time and memory as the batch size increased. However, it started from a much higher time with a batch size of one, compared to other input sizes. The input sizes 480×480, 672×672, and 768×768 started at better inference time for a batch size of one but did not improve significantly by increasing the batch size. Moreover, the increase in memory usage for different batch sizes was higher when using larger input sizes. Therefore, the best input size to use for larger batch sizes is 480×480.

A comparison between the results for batch sizes and other results cannot be taken seriously. Different batch sizes were tested using Method C (copying pixel by pixel to the input tensor) while other tests were conducted using Method B (using the C++ function `memcpy`). Method C was slower, which not only affects a one-to-one comparison of the time values but also might affect the rate of change in time when using different batch sizes as well as different input sizes. Unfortunately, the problem of using a larger batch size with `memcpy` (explained in Sect. 5.3.1.2) was not solved during this project.

5.3.6 Using the classifier

The results for applying the simple CNN model (since ShuffleNet was not supported by TFLite) show that the only difference between the YOLOv5n model and the CNN model is the time used for detection (presented in Sect. 4.2.2). All other results in the inference breakdown such as time and memory were similar to the measurements for YOLOv5n. It can therefore be concluded that it is not the model itself that uses up the memory.

5.4 Network compression with knowledge distillation

This section discusses the findings of using knowledge distillation to do an indirect optimization of resource usage by improving the accuracy on a small network.

5.4.1 ShuffleNetV1 teaching the simple CNN model

Knowledge distillation is good for learning the distribution between the different target classes. The class distribution is especially important when the classifier has to distinguish between many different classes. However, it evidently also works in the case of binary classification, as the results in Sect. 4.3.1 show an improvement of 12%, which is an outcome that was somewhat unexpected. The probability distribution provided by the teacher model is valuable information that would otherwise be lost if only hard labels are used. The probability distribution, which is used in the loss function to train the student model, is a strength of knowledge distillation in the classification tasks.

5.4.1.1 Using unsupported networks

A problem that emerged when trying to use ShuffleNet is that grouped convolution was not supported by TFLite. Having non-supported operations meant that using ShuffleNet on the target hardware was not feasible. However, by using knowledge distillation, a smaller and less complex model was able to utilize the knowledge learned by ShuffleNet in order to better generalize to the data. Moreover, it meant that a model which was practically unfeasible for usage on the target computer could still be utilized to train a smaller model that is a better fit for the target hardware.

5.4.2 YOLOv5l teaching a YOLOv5n

Knowledge distillation for an object detector is more complicated because the task of object detection is more complex than a classifier. Therefore, the results in Sect. 4.3.2 were not as impressive as when using KD on the classifier. Two different cases were used for the training of YOLOv5 using KD. The first is using the Kaggle Airbus Challenge dataset, which only contains one class (see Sect. 5.4.2.1). The second is using the COCO dataset in Sect. 5.4.2.2.

5.4.2.1 The case of detecting maritime vessels

The first KD method was to take knowledge from different feature maps inside the neck of the network (see Fig. 3.2), namely four places which are the outputs of the concatenation layers. Using only one feature map worsened the training performance. By only focusing on teaching the network to learn from one feature map, the general knowledge might not be transferred correctly. Especially in the case of YOLOv5, as different parts of the network are responsible for detecting different sizes of the object. By adding more feature maps, the performance of training instead improved. The improvement was however small, but not due to fluctuations

since it was noticed during multiple tests, with and without pre-trained weights.

Using the prediction output directly (logits) as the transferring knowledge showed a higher improvement than using feature maps. The improvement is thought to be because logits can be directly compared between different YOLOv5 networks, while a series of actions need to be conducted to match the tensor shape of the student and the teacher models when comparing feature maps. This series of steps can mean that distilling knowledge from a YOLOv5l to a YOLOv5n can be problematic since the difference in size is too big, which can lead to a loss of information during training. Therefore, it was investigated to use a YOLOv5s (small) with the same setup, but no improvements were noticed. The small network, on the other hand, might have too small of a difference in performance compared to the nano to learn anything. To draw any conclusions about weather the networks have feasible size for knowledge distillation is therefore not possible from these results.

A trivial motivation to why the improvements only are small is that the testings in this project did not cover the most optimal combinations of feature maps or an optimal learning rate of the knowledge distillation. Furthermore, the performance of the models in the beginning is already high considering the complicated task of detecting and classifying and it might be hard to find another local optima that results in higher improvements.

5.4.2.2 The case of a large number of classes

The complexity of the task was thought to be a contributing factor in the minimal improvements observed when using knowledge distillation on YOLOv5. Detecting a maritime vessel in an image filled with blue water is perhaps not the most challenging task for an object detector such as YOLOv5. Therefore, it is thought that there is not much more the network can learn by using KD in such a case. That is why an experiment using COCO dataset was conducted to test this hypothesis. However, looking at Fig. 4.15, it can be seen that no improvements were detected when using the different setups of knowledge distillation. With these results, another thought arises, that the COCO dataset instead might be a too complicated task for knowledge distillation.

5.5 Theoretical optimizations

It is possible to further decrease the time when applying detection of maritime vessels by combining a classifier and an object detector, or by considering only using the classifier. This section includes a theoretical discussion of these two proposals, using the practical results of this project.

5.5.1 Combining models

One way of speeding up the process is using a small-sized classifier to filter the tiles before detecting the maritime vessels with the YOLO network, as illustrated

in Fig. 5.2. This would increase the inference time over multiple tiles, with a slight penalty of accuracy.

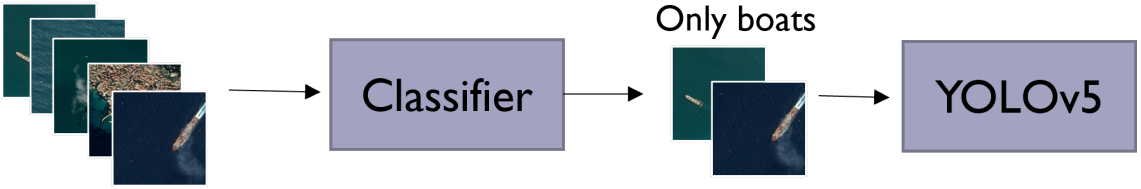


Figure 5.2: An illustration of the filtering process when combining a YOLOv5 with a classifier.

Assuming that a classifier has 87% accuracy, combined with the YOLOv5n network, a general understanding of the performance of such mixed model can be constructed. Based on the ratio of vessels in the Kaggle dataset, which has 42,556 images containing vessels and 150,000 without vessels. Also given the results obtained in time for processing one tile, shown in Table 4.7 and Table 4.13, where the classifier can go through an image in 0.09 s seconds and YOLO in 0.809 s.

With the 87% accuracy of the classifier, the worst-case scenario in time would be if the classifier classifies all the 42,556 images containing vessels correctly while classifying 13% of all the 150,000 images that do not contain vessels incorrectly. This would lead to 62,056 images going through the classifier into the YOLO model. This is however still considerably lower than the 192,556 images that otherwise would have been processed by the YOLO model without the filtering. It is 13% less accurate, but it is 56.65% faster which is a huge improvement.

Fig. 5.3 shows the above discussed worst-case scenario and the average-case scenario over different accuracy of the classifier. Even if the classifiers have less accuracy, and the worst case is experienced, it would still be a massive improvement in time with 50% saved time at 80% in accuracy. However, it would be a more inaccurate model since there would be two steps with uncertainty in the pipeline. As always, there would be a trade-off between time and accuracy.

It is important to note also that if all tiles contain vessels, for example in a harbor where maritime vessels are frequent, the time would instead increase. Both of the models would then be processing all tiles, which results in 0.899 s per tile and an 11% increase in time compared to only using the YOLO model. However, combining the models might not be a problem in the use case of detecting illegal activities since they most probably would not occur in areas with a lot of marine traffic. Moreover, high-traffic areas are usually covered by the tracking systems on earth. The only critical case would be if the hardware is heavily restricted in time so that not all tiles could be processed. Therefore one method could be to only use the detection algorithm when the satellite is orbiting over less trafficked areas and areas which are hard to surveillance.

The filtering process does not require both models to run in sequence. Therefore, it would be advantageous to run the classifier and YOLO in parallel, if the hardware has enough computing power and memory. Having these models running in parallel should in theory save more time.

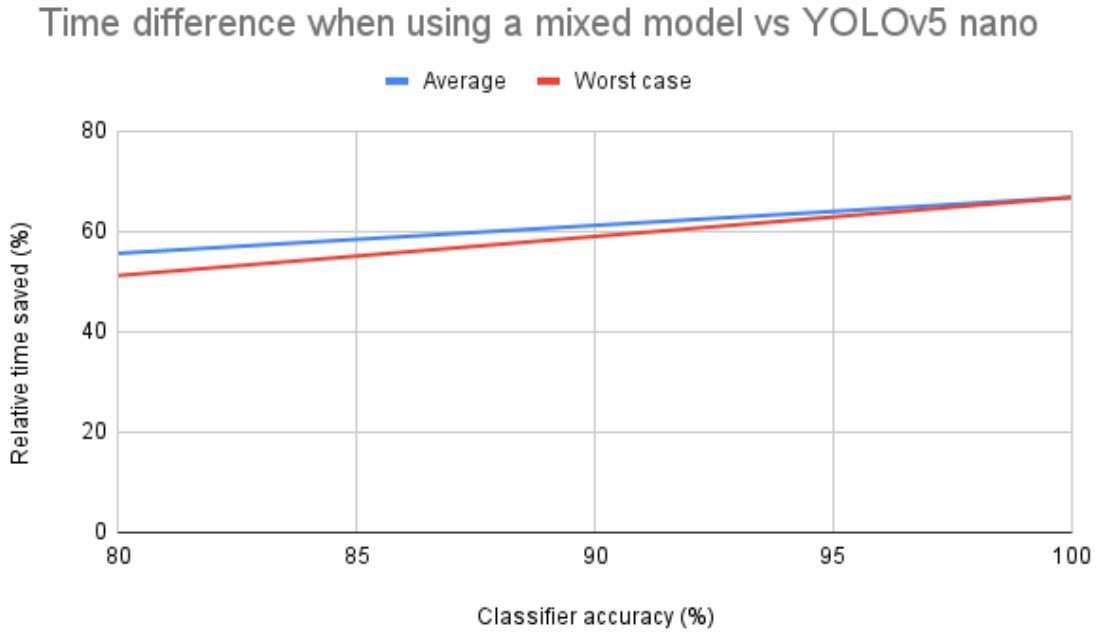


Figure 5.3: The relative time saved when using the mixed model structure compared to only using YOLOv5 nano. Both the average and worst-case scenarios can be seen in the figure.

5.5.2 Using only the classifier

Using only a classifier could be done by taking a relatively small input size into the network. The output of such a solution would be interpreted as: *this area that is covered by this tile potentially contains a vessel*. Such output is still valuable information, especially if the area is small enough. For example, an image with a resolution of 10 m/pixel and input size of 192×192 would correspond to an area of 3.7 km^2, which is a relatively easy area to cover [53]. More importantly, it would amount to 87% less time compared to using a YOLOv5n.

Another restriction of using only the classifier is that it cannot be used in areas where vessels are allowed to be. The tracking system AIS is not required for smaller vessels. Therefore if the classifier detects the presence of a boat within an area, there would be no information on how many boats there are or if they are boats that cheats the tracking system (i.e. larger boats). It could simply be a family on a vacation that is detected. This is avoided when using the YOLO model since it also gives a bounding box around the vessel that reveals the size and location of the vessel. To send the coast guard each time a vessel is detected would therefore

be unfeasible when using the classifier. However, covering protected areas where no vessel should exist, then it is highly effective.

5.6 Detecting vessels from space

This section discussed the applicability of using a ML model to detect maritime vessels from space.

5.6.1 Frame rate

The frame rate of 2 min/image is not fast enough to process a live feed (more than one fps), however, it is still fast enough to provide valuable information in Earth observation. Detection using a live feed of a single location would not be possible using a single satellite due to the orbiting state of most satellites. One solution would be to use geostationary satellites. A combination of several satellites could also provide an almost continuous feed of a certain location. This is practical in cases when small changes are relevant, such as tracing a vessel to detect, for example, fishing patterns. In the task of monitoring the oceans, a high frame rate is not required due to the long distances and large areas covered. A single vessel in the ocean would for example not have any significant changes in its location during the two minutes it takes to tun an inference. Moreover, the location between the frames could also be calculated or filled with the location given by the AIS.

5.6.2 Memory when deployed

Time is affected by the size of the RAM that the computer has. More memory will allow for parallel processes, which decreases time. Measurements will therefore be completely different on other computers. The swap memory was turned off during the testing, which is on by default. Having swap memory turned on enables the process to use more memory and possibly decrease the processing time if it enables parallelism.

5.6.3 The results depend on the camera

The camera and the type of images the camera produces also affect the measurements in regards to different frame rates, data types, and resolutions. An optic camera is not operable during the night and has noise such as clouds, while a radar camera is not affected by these aspects. The extremely large image of $10{,}000 \times 10{,}000$ was chosen for the test setup, which comes from the Copernicus project that uses one of the best cameras available on satellites. Therefore in most cases, there will be a smaller image that is processed.

5.6.4 Tiling the image

It is important to note that tiling an image presents a risk of accidentally cropping the vessels in two. The camera must therefore have a resolution where the vessels

are small enough to have a small probability of a vessel being on the edge of a tile. Fig. 5.4 shows examples of images obtained from the Kaggle Airbus Challenge with different vessel sizes in a 768×768 pixels image. With an input size of 192×192, there would be four tiles. For the vessels in the first image, the tiling would most probably not crop where any vessel is located, while in the second and third image, a cropping over a vessel would be probable.

Figure 5.4: Different vessel sizes for images with a resolution of 768×768 pixels.

5.6.5 Accuracy of the model

The object detector tested obtained 0.789 in precision and 0.701 in recall. These results would mean that the detector would give a false prediction 21 % of all times and it would miss a vessel 30 % of all times. In a real scenario of detecting vessels, it would still contribute by giving a hint of where vessels are located. If there is no match of a vessel in the existing tracking systems such as AIS, then the pipeline could still warn the coast guards that there is an untracked vessel at a certain location. In location aspects, YOLOv5n got 0.761 in mAP@0.5 and 0.49 in mAP0.5:0.95, which is not as important as the precision and recall. The exact location and size of a bounding box do not in this case matter since the exact location and size of the vessel can be backed up by the AIS and previous detection of the same boat.

5.6.6 Use cases

There are two different use cases for the models used in this project. The first one is when an object detector is used, with or without filtering at first, to complement the existing tracking system AIS. The complementing part is important, since it is inevitable to be captured by a satellite image, while it is easy to cheat the AIS. One possible pipeline would be to first detect the vessels and extract the location and size of each of them based on their bounding boxes. The smaller maritime vessels that are not required to have an AIS are then filtered out. Such information is much smaller in size, compared to a satellite image, which is easier and faster to downlink to Earth. These locations are compared with the information available in the AIS. If no match of the vessel is found, then the coast guard would be alarmed that there

is a detection of a potential illegal activity.

The other use case would be when detecting boats in protected areas where no vessels are allowed. In this case, using only the classifier would be the most efficient method. The pipeline would be similar, but the downlinked information would contain an area instead, which is directly alarmed to the coast guard.

5.7 Ethical and sustainability aspects

With Earth observation follows the complications of integrity and consent. Surveillance at certain locations could be seen as an act of espionage or even a threat between countries. The privacy of individuals is also affected since the images captured from above can unveil enclosed areas, which are taken without consent. Satellite images are large and usually only give an overview of an area, but some cameras on certain satellites could capture images down to a resolution of 30 cm, which is almost as big as a license plate of a car. Surveillance does however help in creating order and could be used to prevent disasters. Especially observing oceans is important due to their critical condition. The supervision of oceans is poor due to the huge covering areas and different laws in different water areas (international waters for example). Therefore illegal activities, such as dumping trash or oil, illegal fishing, and smuggling, are easier to carry out there.

There are several sustainability issues regarding the increase in the number of satellites in recent years. More satellites will increasingly pollute the higher layers above the atmosphere. The process of launching satellites requires a massive amount of resources, both for the rocket to leave the Earth's atmosphere and for developing the rocket itself. The space traffic is also not organized as the traffic on Earth and collisions do occur. There are no dangers of satellite parts reaching the Earth and satellite parts are small enough to burn before reaching Earth. However, as the number of satellites increases rapidly, the probability of collisions will increase creating a huge amount of space debris, which in its turn can lead to more collisions.

5.8 Future work

There are several tests that this project did not cover and plenty of extensions of the project. This section mentions some of the most important future work.

5.8.1 Further optimizations of memory and speed

A good deal of optimization could be done using low-level programming as mentioned in Sect. 5.3.1. Manually allocating and freeing memory to avoid unnecessary copies could potentially save more memory.

One method that can be worth looking into is testing whether the process can be split between both the CPU and the GPU, given that the CPU is running multiple

threads. In theory it could yield double the speed of only running on one of the devices. To note however is that two different interpreters then are required, which could increase memory usage.

5.8.2 Serialization of the GPU delegate

The duration of initializing the GPU delegate significantly increased the time and was therefore mitigated to get comparable results for the inferences using the GPU. The initialization can be avoided by continuously running the process, but it might not be possible in all applications. For example, when working with time-slots to manage several processes in the satellite, the delegate would be required to re-initialize every time the process starts. This can however be solved by investigating the serialization of the delegate from previous runs which increases the initialization time up to 90 % [54].

5.8.3 `memcpy` using batches

As discussed in Sect. 5.3.1.2, the function `memcpy` did not work when using a batch size larger than one. The implementation of copying values to the input tensor was to apply `memcpy` on each tile in the batch. This implementation failed since the tiles were not contiguous in memory, which is required when using `memcpy`. It would be advantageous to solve this since the implementation of `memcpy` is faster than other tested implementations. A possibility would be to create a matrix as a temporary place holder, consisting of all tiles in the batch. However, this would require an additional copy of the input containing all tiles which would increase memory usage. On the other hand, an additional copy would theoretically be equivalent to a larger input size.

5.8.4 The problem of overlapping tiles

It was mentioned in Sect. 5.3.2.2 that there exists an overlapping problem that causes the processing of unnecessary pixels if the input size to the network is not matched to the size of the input image. This is however not feasible if the input image is of dynamic size. Another solution that was not tested in this project, could therefore be to instead discard the leftover pixels if there are only a few leftover pixels. Discarding pixels, however, introduces a risk of missing a boat, if the boat is located on the right or bottom edges of the image. On the other hand, if the satellite images do overlap, then this would not be a problem.

5.8.5 Knowledge distillation for object detectors

There are plenty of explorations left for using knowledge distillation on complicated networks such as the YOLOv5. Testing more combinations of feature maps for the KD method of imitation masks would give a better understanding of how the feature maps affect the training. Moreover, finding an optimal learning rate for KD in relation to other learning rates could also improve the training. An investigation

of whether YOLOv5n is too small to additionally improve would also be interesting, or simply combine the losses of feature imitation and the logits.

5.8.6 Testing in space

A natural extension to the project is to test the model in the satellite in orbit. Radiation and other environmental issues could affect the model performance as well as the process around it. More work is also required to actualize it as a full product. For example, post-processing the output into geo-locations which can be sent back to earth where it is received and compared to AIS data.

5.8.7 Edge Learning

Edge Learning is a growing concept that would be interesting to investigate in this use case, where prediction and training occur on-board the satellite. By sending the AIS location and vessel size of all vessels inside the camera snapshot to the satellite, the labeling and training using camera images directly on the edge device could be possible. This would allow to continuously improve the model.

6
Conclusion

This research shows that machine learning can be implemented on board a satellite with aim of detecting and localizing maritime vessels. The best result obtained, using YOLOv5, is 2.1 min per 10,000×10,000 pixel image, which is considered to be very large. Therefore, in most cases, even less time and memory will be used.

There are certain limitations that need to be taken into account. One of these limitations is the target hardware, and especially the target camera, as the resolution and image size is vital for both model accuracy and applicability on board a satellite. Model choice is an important aspect when the target hardware is restrictive. Therefore, it is important to not only look at metrics such as FLOPs, which is an indirect metric when it comes to actual inference time. However, there is room for flexibility depending on the rate at which results need to be obtained from the application, as well as the rate at which images are captured.

The research also shows that *memory management* and the *data type* of the image are extremely important to memory usage. For example, having an image of `int8` and only converting the tile used in inference reduces the memory usage significantly. Also, `memcpy` was more efficient in time than copying pixels one by one but did not work for larger batch sizes. It is also shown that when tiling the image, it is important to match the tile size, or *input size*, to the image size to reduce the overlap created by the last tile on each row and column. Few pixels in the last tile will create a larger overlap that increases time due to more pixels being processed. *Quantization* was not supported by the target hardware but decreased the file size. The precision of `float16` was optimal since it reduced the file size by half and did not affect the inference time or memory usage. Using `int8` reduced the file size additionally by half, but increased the inference time since the process in TFLite is first to de-quantize the input and then quantize the output. Furthermore, using a GPU in TFLite yields a better inference time at a cost of higher memory usage. This is most likely due to the GPU delegate and the fact that the GPU and CPU have separate memory accesses. One way to mitigate this is to use the CPU with *multiple threads* at a cost of increased CPU usage. *Batch sizes* larger than one did not decrease the inference time to the same extent as larger input sizes. However, the measurements of batch sizes might not be comparable since `memcpy` does not work with batch sizes larger than one. Furthermore, by comparing the results of using the classifier and YOLOv5, it is evident that the model size does not matter in measuring memory, but does matter significantly in time.

Applying a smaller network comes with a penalty in accuracy, but accuracy could be increased using knowledge distillation from a larger network. This was shown successful for distilling knowledge from ShuffleNetV1 to a simple CNN model where the model accuracy increased by 12 %. Moreover, it enables models which are not supported by the target computer (ShuffleNetV1) to distill their knowledge to a supported model (simple CNN model). Knowledge distillation on YOLOv5 however was found difficult due to the much more complex nature of the network and task. A slight improvement of the training metrics was obtained were using the logits yielded the best results. Using three feature maps also yielded better training, but using only one feature map worsened the training. In a multi-label setup using the COCO dataset, there was no improvement when using knowledge distillation. There is however plenty of additional tests that could give a better understanding of the effect of knowledge distillation on an object detector.

One way of radically decreasing inference time (87 % less time) is to use a classifier instead of an object detector, combined with a small input size. This comes at a cost of having an estimation of an area in which a vessel is located, rather than an exact location. Another approach to decrease inference time is combining the object detector with a classifier that first filters out tiles with only water. This saves around 50 % of the time with a penalty of 13 % in accuracy and the risk of instead increasing time if all tiles contain vessels, for example near harbors.

Apart from the methods tested in this project, there are plenty more possibilities to optimize the process even more. Hyperparameter testing, network pruning, low-level programming to decrease redundant memory allocations, customizing interpreters and accelerators and other optimizations available in the frameworks are yet to be studied. The initialization of the GPU when starting the process takes time and is not included in the results of this project. Therefore an important future work in cases where frequent process startups are required would be to save the delegate to the disk using serialization (explained in Sect. 5.8.2).

The results of this project will hopefully lead to further developments in the field of maritime monitoring. More work is required before this can become an assisting application when stopping illegal activities, but it constitutes a good first step in that direction.

Bibliography

[1] H. Wu and Q. Dai, "Artificial intelligence accelerated by light," *Nature Publishing Group*, 2021.

[2] Y. Duan, J. S. Edwards, and Y. K. Dwivedi, "Artificial intelligence for decision making in the era of big data–evolution, challenges and research agenda," *International Journal of Information Management*, vol. 48, pp. 63–71, 2019.

[3] J. Redmon and A. Farhadi, "Yolo9000: Better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[4] K. He *et al.*, "Deep residual learning for image recognition," 2016.

[5] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," 2014.

[6] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," 2020.

[7] M. Sandler *et al.*, "Mobilenetv2: Inverted residuals and linear bottlenecks," 2018.

[8] A. D. George and C. M. Wilson, "Onboard processing with hybrid and reconfigurable computing on small satellites," *Proceedings of the IEEE*, vol. 106, no. 3, pp. 458–470, 2018.

[9] O. Petri, "Urban change detection on satellites using deep learning," M.S. thesis, 2021.

[10] K. Bereta, D. Zissis, and R. Grasso, "Automatic maritime object detection using satellite imagery," 2020.

[11] J. B. C. Jackson *et al.*, "Historical overfishing and the recent collapse of coastal ecosystems," *Science*, vol. 293, no. 5530, pp. 629–637, 2001. DOI: 10.1126/science.1059199. eprint: https://www.science.org/doi/pdf/10.1126/science.1059199. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.1059199.

[12] *U.s. wants ships to keep their tracking devices on -senior official*, https://www.reuters.com/article/mideast-iran-tankers-tracking-idUSL2N24T089, Accessed: 2022-03-07.

[13] M. Mukhayadi, A. Karim, and W. Hasbi, "Designing a constellation for ais mission based on data acquisition of lapan-a2 and lapan-a3 satellites," *Telkomnika*, vol. 17, no. 4, 2019.

[14] E. Ahlberg and J. Danielsson, "Handling and analyzing marine traffic data," M.S. thesis, 2016.

[15] X. Zhang *et al.*, "Shufflenet: An extremely efficient convolutional neural network for mobile devices," pp. 6848–6856, 2018.

[16] J. Hu, L. Shen, and G. Sun, "Squeeze-and-excitation networks," 2018.
[17] J. Redmon *et al.*, "You only look once: Unified, real-time object detection," 2016.
[18] J. Redmon and A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.
[19] J. Solawetz. (2020). "Yolov5 new version - improvements and evaluation," [Online]. Available: https://blog.roboflow.com/yolov5-improvements-and-evaluation/.
[20] X. Long *et al.*, "Pp-yolo: An effective and efficient implementation of object detector," *arXiv preprint arXiv:2007.12099*, 2020.
[21] *Kaggle: Airbus ship detection challenge*, https://www.kaggle.com/c/airbus-ship-detection/data, Accessed: 2021-02-23.
[22] *Ix5-100 spacecloud® solution*, https://unibap.com/en/our-offer/space/spacecloud-solutions/ix5100/, Accessed: 2022-02-23.
[23] F. Rosenblatt, "Perceptron simulation experiments," *Proceedings of the IRE*, vol. 48, no. 3, pp. 301–309, 1960.
[24] A. L. Fradkov, "Early history of machine learning," *IFAC-PapersOnLine*, vol. 53, no. 2, pp. 1385–1390, 2020.
[25] *About copernicus*, https://www.copernicus.eu/en/about-copernicus, Accessed: 2022-05-23, [Online].
[26] *Discover our satellites*, https://www.copernicus.eu/en/about-copernicus/infrastructure/discover-our-satellites, Accessed: 2022-05-23, [Online].
[27] K. Bereta *et al.*, "Monitoring marine protected areas using data fusion and ai techniques," 2019.
[28] *Sentinel-1*, https://sentinels.copernicus.eu/web/sentinel/missions/sentinel-1/overview, Accessed: 2022-05-23, [Online].
[29] *Sentinel-2*, https://sentinels.copernicus.eu/web/sentinel/missions/sentinel-2, Accessed: 2022-05-23, [Online].
[30] *Worldview legion*, https://www.maxar.com/worldview-legion.
[31] O. Russakovsky *et al.*, "Imagenet large scale visual recognition challenge," *International journal of computer vision*, vol. 115, no. 3, pp. 211–252, 2015.
[32] Devopedia. (2021). "Imagenet." Accessed: 2022-05-21, [Online]. Available: https://devopedia.org/imagenet%5C#:~:text=ImageNet%5C%20is%5C%20a%5C%20large%5C%20database,kind%5C%20in%5C%20terms%5C%20of%5C%20scale.
[33] T.-Y. Lin *et al.*, "Microsoft coco: Common objects in context," in *European conference on computer vision*, Springer, 2014, pp. 740–755.
[34] *Cocodataset.org*, https://cocodataset.org/#home, Accessed: 2022-05-23, [Online].
[35] M. Everingham *et al.*, "The pascal visual object classes (voc) challenge," *International journal of computer vision*, vol. 88, no. 2, pp. 303–338, 2010.
[36] *Copernicus open access hub*, https://scihub.copernicus.eu/dhus/#/home.
[37] *Kaggle*, https://www.kaggle.com/.

[38] D. Lu and Q. Weng, "A survey of image classification methods and techniques for improving classification performance," *International journal of Remote sensing*, vol. 28, no. 5, pp. 823–870, 2007.
[39] A. Krogh, "What are artificial neural networks?" *Nature biotechnology*, vol. 26, no. 2, pp. 195–197, 2008.
[40] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1251–1258.
[41] S. Li *et al.*, "Detection of concealed cracks from ground penetrating radar images based on deep learning algorithm," *Construction and Building Materials*, 2021.
[42] P. Jiang *et al.*, "A review of yolo algorithm developments," *Procedia Computer Science*, vol. 199, 2022.
[43] D. Thuan, "Evolution of yolo algorithm and yolov5: The state-of-the-art object detection algorithm," 2021.
[44] G. Hinton, O. Vinyals, J. Dean, *et al.*, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, vol. 2, no. 7, 2015.
[45] T. Wang *et al.*, "Distilling object detectors with fine-grained feature imitation," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2019, pp. 4933–4942.
[46] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented approximation of convolutional neural networks," *arXiv preprint arXiv:1604.03168*, 2016.
[47] S. Han *et al.*, "Learning both weights and connections for efficient neural network," *Advances in neural information processing systems*, vol. 28, 2015.
[48] J. Wu *et al.*, "Quantized convolutional neural networks for mobile devices," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 4820–4828.
[49] M. Rastegari *et al.*, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European conference on computer vision*, Springer, 2016, pp. 525–542.
[50] G. J. et al. (2022). "Yolov5 github repository," [Online]. Available: `https://github.com/ultralytics/yolov5/tree/v6.1`.
[51] E. Asaad and S. Larsson, `https://github.com/Sara980710/master_thesis`, Jun. 2022.
[52] *Sentinel-2 user handbook*, 1st issue, European Space Agency, Paris, France, 2015, p. 9.
[53] M. S. Essa, private communication, May 2022.
[54] *Gpu delegate serialization*, `https://www.tensorflow.org/lite/performance/gpu_advanced#gpu_delegate_serialization`.
[55] R. Girshick, "Fast r-cnn," 2015.
[56] D. Silver *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.
[57] J. Nelson. (2020). "Responding to the controversy about yolov5," [Online]. Available: `https://blog.roboflow.com/yolov4-versus-yolov5/`.
[58] R. Girshick *et al.*, "Rich feature hierarchies for accurate object detection and semantic segmentation," 2014.

[59] S. Ren *et al.*, "Faster r-cnn: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems*, vol. 28, 2015.

[60] W. Liu *et al.*, "Ssd: Single shot multibox detector," in *European conference on computer vision*, Springer, 2016, pp. 21–37.

[61] T.-Y. Lin *et al.*, "Focal loss for dense object detection," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2980–2988.

[62] T. Wang *et al.*, "Learning rich features at high-speed for single-shot object detection," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019, pp. 1971–1980.

[63] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," pp. 10 781–10 790, 2020.

[64] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," 2019.

[65] F. N. Iandola *et al.*, "Squeezenet: Alexnet-level accuracy with 50x fewer parameters and< 0.5 mb model size," *arXiv preprint arXiv:1602.07360*, 2016.

[66] B. Yan *et al.*, "A real-time apple targets detection method for picking robot based on improved yolov5," *Remote Sensing*, vol. 13, no. 9, 2021.

[67] X. Huang *et al.*, "Pp-yolov2: A practical object detector," *arXiv preprint arXiv:2104.10419*, 2021.

A
Appendix

This appendix shows the model architecture for the simple CNN model in Sect. A.4.

A.1 The history of YOLO

YOLO, "you only look once" at the formulated problem of object detection. YOLO is an object detector that simultaneously classifies and detects objects within an image. The model consists of a *backbone* (feature extraction), *neck* (feature aggregation), and *head* (prediction and regression). The possibility of fast detection of objects within the frame rate of a video is the reason why YOLO received massive attention within the community and has become one of the most used object detectors [17]. In addition, by using the image directly, an encoding of the global features could be obtained. The encoding yields a strong generalization when extracting features and also reduces the error of detecting the background [42]. Several versions of YOLO have been developed since, which have increased the efficiency, accuracy, and applicability of the model. The different YOLO versions have allowed YOLO to be implemented in different platforms, even small and mobile embedded systems.

The first version of YOLO was developed to decrease the inference time on object detectors. The architecture consisted of 24 convolutional layers followed by two fully connected layers, where the first twenty layers were pre-trained for classification [17]. Even if the model was fast, it suffered from errors in the localization and recall, and the effects of using different sizes of bounding boxes were not included in the loss [3][17].

YOLOv2 focused on the weaknesses of the first YOLO version, mainly low recall and high localization error. The published paper states that YOLOv2 was able to detect over nine thousand different objects with even greater speed than before when using a particular setup of the model which they called *YOLO9000*[3]. A new, faster architecture for the backbone called Darknet-19 consisting of 19 convolutional layers and five max-pooling layers were used, with batch normalization and a high-resolution classifier (double the resolution for YOLOv1). They further introduced anchor boxes (a prior set of bounding boxes given as input to the network) with location offsets relative to the grid cells instead of predicting the bounding boxes directly with global coordinates. The anchor boxes made the training of the network significantly easier and it was inspired by Faster R-CNN [3][55]. To obtain relevant anchor boxes, k-means clustering is used on the target dataset. Several ob-

jects could then also be predicted at once since several anchor boxes are used when sliding through the image. This increases the number of objects detected from 98 boxes to more than a thousand [3]. Moreover, a way of combining datasets was also developed since there was a large gap between the existing number of annotations for the classification and the detection. The method is using WordTree hierarchy for object detection to solve the gap which made it possible to train both tasks simultaneously. YOLO9000 is also trained with randomly selected input sizes to provide a smooth trade-off between speed and accuracy.

YOLOv3 increased the overall performance by adding a better feature extractor called Darknet-53, consisting of 53 convolutional layers with residual layers in between [18]. Residual networks use *skip connections* that skip several layers, which eases the optimization task and solves the degradation problem that emerges as networks become deeper [56][4]. YOLOv3 also predicts bounding boxes at three different scales, similar to a *feature pyramid network* (FPN) that concatenates feature maps during downsampling with features during upsampling. The original author of YOLO, Joseph Redmon, stopped developing the models after YOLOv3 [43].

YOLOv4 targeted training on conventional GPUs and did an investigation of different methods to increase efficiency and accuracy in the model architecture and pipeline. The authors used a so-called "bag of freebies" that consisted of methods to decrease the training costs and a "bag of specials" that significantly improves the accuracy with a small increase in inference cost [6][20]. The backbone consists of a CSPDarkNet53, which is a modified version of DarkNet53 where the RESNets are replaced by cross-stage partial (CSP) blocks. CSP maintains fine-grained features by repeatedly sending half of the feature map through a dense layer where the output then is concatenated with the untouched half. This will allow a better gradient flow through the dense layers, saves the gradient changes in the feature map, and reduces the number of parameters. The neck then consists of *spatial pyramid pooling block* (SPP) and a *path aggregation network* (PAN). SPP provides a fixed output size with a dynamic input size and it increases the receptive field using different sizes of pooling proportional to the input size, creating a spatial pyramid. PAN is used to preserve the spatial knowledge that is easily lost in deep networks and when down-sampling. The concept is the same as the FPN used in YOLOv3 but it additionally includes an extra bottom-up layer that is used to connect layers from the earlier layers. This thorough investigation increased the AP and FPS by 10 % and 12 % respectively compared to the results of YOLOv3.

YOLOv5 translated the network from the framework Darknet to the user-friendly and widely used framework PyTorch [43]. However, this model is similar to YOLOv4 and does not have any essential improvements [19][41]. Therefore, the name YOLOv5 has been controversial within the community [57][43] and there was no peer-reviewed publication connected to the release of YOLOv5. YOLOv5 is also constantly developing with new versions within YOLOv5 and the framework gives a wide range of settings and possibilities to modify the network structure for customization. YOLOv5 also uses auto-learning bounding box anchors for easier choice of anchors.

Other YOLO versions have been developed after YOLOv5 such as *PaddlePaddle YOLO* (PP-YOLO) which is based on another framework called PaddlePaddle, and there are plenty of additional models that are modifications of the original YOLO versions [20].

A.2 Literature review of object detectors

This section presents a literature review of different object detectors (other than YOLO) that were investigated during this project.

A.2.1 R-CNN, Fast R-CNN, Faster R-CNN

Regional based convolutional neural network (R-CNN) is an early developed *two stage detector* (TSD) from 2014. The model uses two thousand proposed regions obtained by a selective search algorithm, which groups image segmentations into larger regions by looking at different similarities [58]. The proposals of regions solved the problem of having infinite possibilities of locations that CNN object detectors previously had since the output must define all possible locations. The time required to detect is high per image since the CNN is fed by all two thousand region proposals. Therefore, the author created *Fast R-CNN* that instead runs the image directly as input to the CNN, where the feature map then was reshaped using a region of interest (ROI) pooling layer to fix the output image with the region proposals [55]. *Faster R-CNN* then increased the time even further by replacing the selective search algorithm with a separate network that instead predicts the region proposals [59].

A.2.2 SSD

Single shot multi-box detector (SSD) was developed in 2016 to skip the region proposal detector in R-CNNs and is, therefore, an OSD [60]. The model consists of a single network that extracts features with a set of default bounding boxes and a scoring of all classes per location. The bounding boxes and classes are then predicted for multiple resolutions to handle objects in different sizes.

A.2.3 RetinaNet

RetinaNet is also an OSD and was the first one to surpass the TSD in accuracy because it solved the class imbalance problem in OSDs [61]. This however results in a high dimensional output where the number of elements is given by the number of bounding boxes, locations, and classes (A.2.3).

$$nr_{elements} = nr_{box} \times nr_{loc} \times nr_{class}$$

The high dimensional output mostly contains "easy" negative predictions compared to the TSD where the region proposal detection already has filtered out most regions for the classification task. Classes that are considered "easy" to classify as negative

for the network are the ones that obtain values close to zero. To solve this, they introduced *Focal loss* that pushes all loss values near zero to zero to prevent small losses on frequent classes to accumulate beyond the bigger losses on less frequent classes. The structure is similar to other OSDs where they use a CNN structure called ResNet as the backbone and a FPN.

A.2.4 LRF

Learning rich features (LRF) is an OSD that uses a bi-directional network instead of the traditional top-down pyramid structure to preserve all levels of semantic information since the top-down method mainly focuses on passing high-level features to the bottom layers [62].

A.2.5 EfficientDet

EfficientDet is a family of detectors that use a *weighted bi-directional feature pyramid network* (BIFPN) which is similar to the concept of LRF, and *compound scaling* that scales the resolution, depth, and width of all stages of the network uniformly [63]. These networks cover a large scale of resource constraints which increasingly is requested.

A.3 Comparison of existing models

This section presents a comparison of the different models investigated in this project before deciding which was most suitable.

A.3.1 Image classifiers

There are multiple frequently used networks for image classification such as ResNet, SENet, AlexNet, DenseNet, Xception, and EfficientNet [64]. These networks are compared in the paper for EfficientNet, where the EfficientNet is the smallest net with the best accuracy.

Then there are networks specifically developed to be small and efficient for mobile and embedded systems. In the ShuffleNetV1 paper, it is shown that ShuffleNet achieves a better, or similar, classification error compared to other networks such as VGG-16, GoogleNet, AlexNet, and SqueezeNet [15]. This is achieved even though ShuffleNetV1 has down to ten times lower complexity (FLOP) than the networks mentioned above [15]. The paper further stated that compared to MobileNet-224 which is an efficient network for mobile applications, the ShuffleNetV1 gives better results in both complexity (FLOP), and classification error. However, another paper that uses MobileNet-224 and ShuffleNetV1 states that MobileNet-224 gives less error. On the other hand, in this test MobileNet-224 had four times more complexity (FLOP) and twice as many parameters [16]. MobileNetV2 then came along with better accuracy and less complexity than MobileNetV1, which achieved better results than ShuffleNetV1 2x in terms of accuracy, but FLOP was not mentioned in

that paper [7]. A compilation of the results found in these papers can be found in Table A.1.

Network	Accuracy (%)	MFLOP	Params (M)
ShuffleNetV1 x0.5 (g=4)	58.4	38	-
ShuffleNetV1 x1.0 (g=8)	67.6	140	1.8
ShuffleNetV1 x1.5 (g=3)	71.5	292	3.4
ShuffleNetV1 x2.0 (g=3)	73.7	524	5.0
MobileNet-224	70.6	569	4.2
MobileNetV2	72.0	300	3.4
EfficientNet-B0	77.1	390	5.3

Table A.1: A comparison of different classifiers that were tested on the ImageNet dataset. The numbers have been gathered from different papers [7][15][65][64].

A.3.2 Object detectors

To compare different object detectors, an overview of results achieved from several object detector papers is discussed in this section. The investigation does mostly focus on obtaining a model with high speed.

In the YOLOv2 paper, it is stated that YOLOv2 was able to predict with 73.4 mAP for the VOC 2007 dataset [3], which already performed better than the TSDs Fast R-CNN and Faster R-CNN as well as the OSD SSD as seen in Fig. A.1. The paper however mentions that it performs worse using the COCO dataset, which is also seen in the comparisons of models in the RetinaNet paper in Fig. A.2. The results furthermore state that RetinaNet performs at the same level as the TSD and it is the first OSD that has better accuracy than all TSD as mentioned earlier [61]. The paper for YOLOv3 then uses almost the same graph as RetinaNet but adds the results for YOLOv3. It shows the improvement in accuracy compared to YOLOv2, seen in Fig. A.3. YOLOv3 almost reaches the level of accuracy RetinaNet has while preserving the high speed of half the inference time RetinaNet requires.

The YOLOv4 paper is comparing its model to loads of other object detectors (YOLOv3 among them), and YOLOv4 performs better than all in the trade-off between precision and accuracy [6]. One of the more noticeable detectors mentioned is the *learning rich features* (LRF) since it could achieve a speed almost twice the speed of YOLOv4 as seen in A.4. However, LRF instead suffers from accuracy. EfficientDet is often seen in graphs for comparison since it is scalable in a steady and uniform way [6][20][63][66]. YOLOv4 has double the speed compared to EfficientDet, as seen in Fig. A.5.

YOLOv5 is not significantly different from YOLOv4 as mentioned in Sect. A.1. One paper compares YOLO versions for detecting invisible cracks in the ground using ground-penetrating radar [41]. Their results show that YOLOv4 has better robustness and overall detection of cracks since YOLOv5 also detects pseudo-cracks.

Another comparison paper which is exploring the evolution of YOLO versions, states that it is hard to compare the two versions since they are built in two different languages and two different frameworks [43]. However, in the end, the same paper state that YOLOv5 has been proven to be better in several circumstances. YOLOv5 is also more practiced due to the ease of use compared to YOLOv4 which might lead to more people using and testing YOLOv5. A third paper that uses object detection for computer vision in apple picking robots has examined that the size of the file containing the weights for YOLOv5 is 90 % smaller than the file for YOLOv4, which makes the model more suitable for small mobile and embedded systems [66].

In the paper covering PP-YOLO, it is mentioned that PP-YOLO outperforms YOLOv4 as seen in Fig. A.6. Similarly, as seen in Fig. A.7 from the paper for PP-YOLOv2 it is shown that PP-YOLOv2 outperforms YOLOv5. There are however not many other comparisons of the family of PP-YOLO and the framework PaddlePaddle is not as widely known as Pytorch.

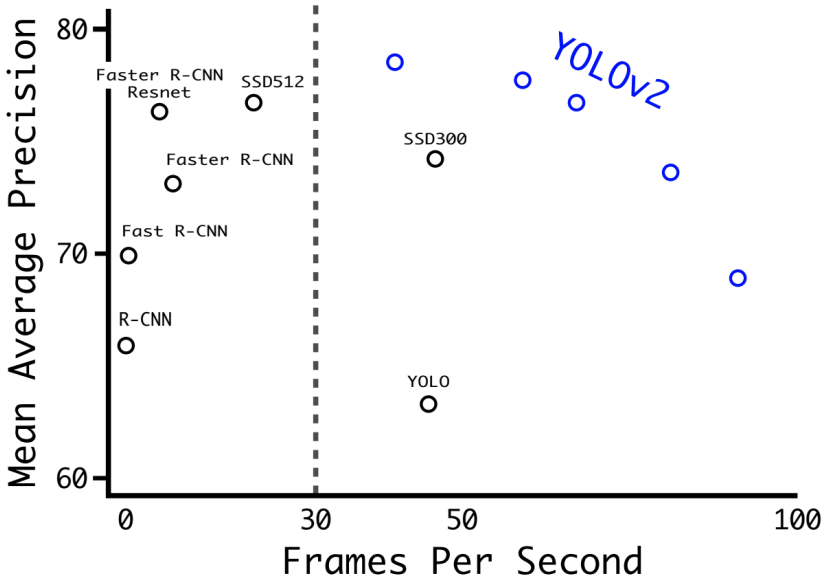


Figure A.1: Comparison of object detectors from the YOLOv2 paper [3]. Accuracy and speed on VOC 2007 dataset.

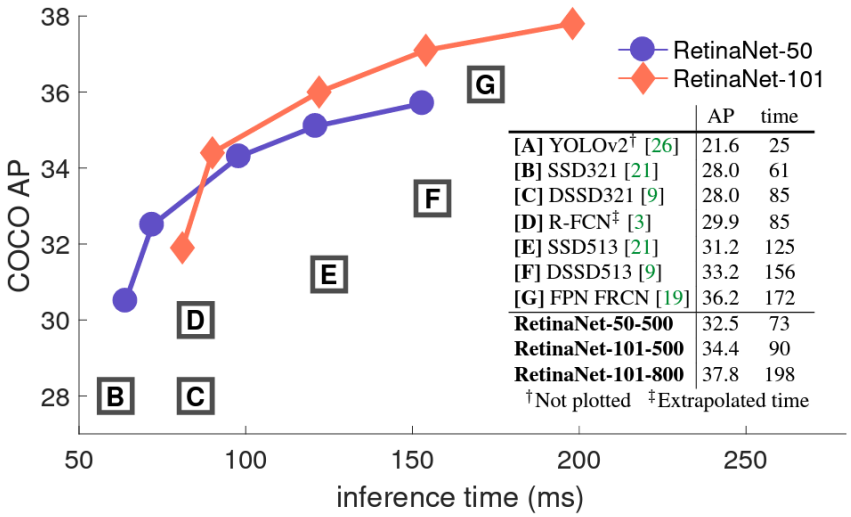


Figure A.2: Comparison of networks from the RetinaNet paper [61]. Average precision versus inference time on the COCO dataset.

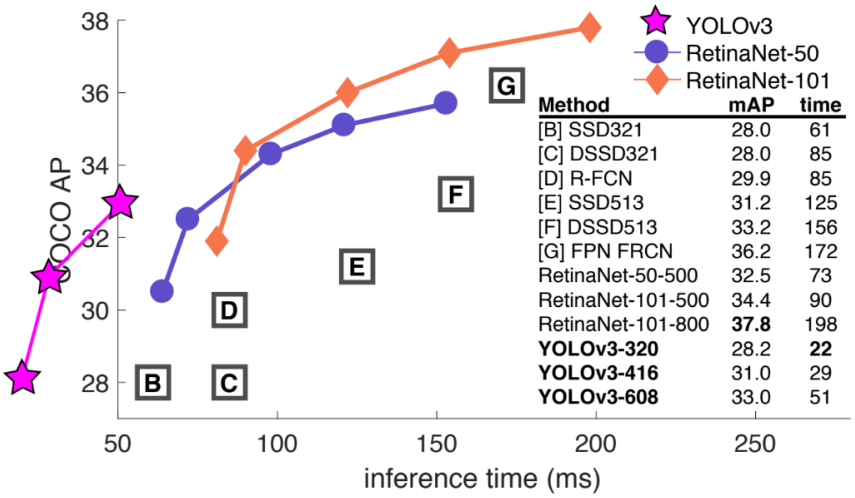


Figure A.3: Comparison of networks from the YOLOv3 paper [18]. Average precision versus inference time on the COCO dataset.

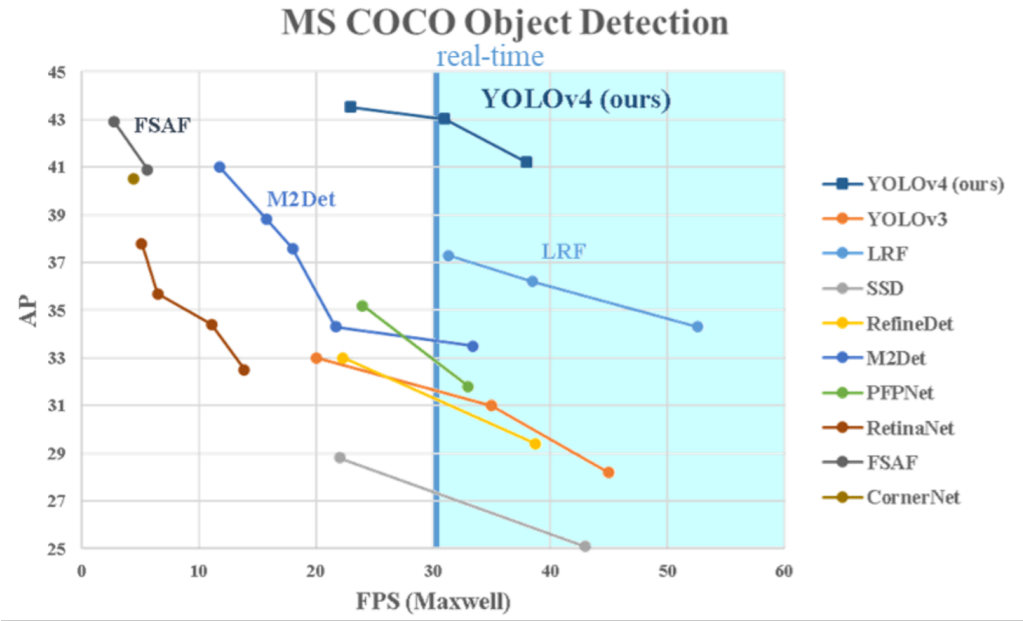


Figure A.4: One of the graphs comparing networks from the YOLOv4 paper [6]. Average precision versus inference time on the COCO dataset.

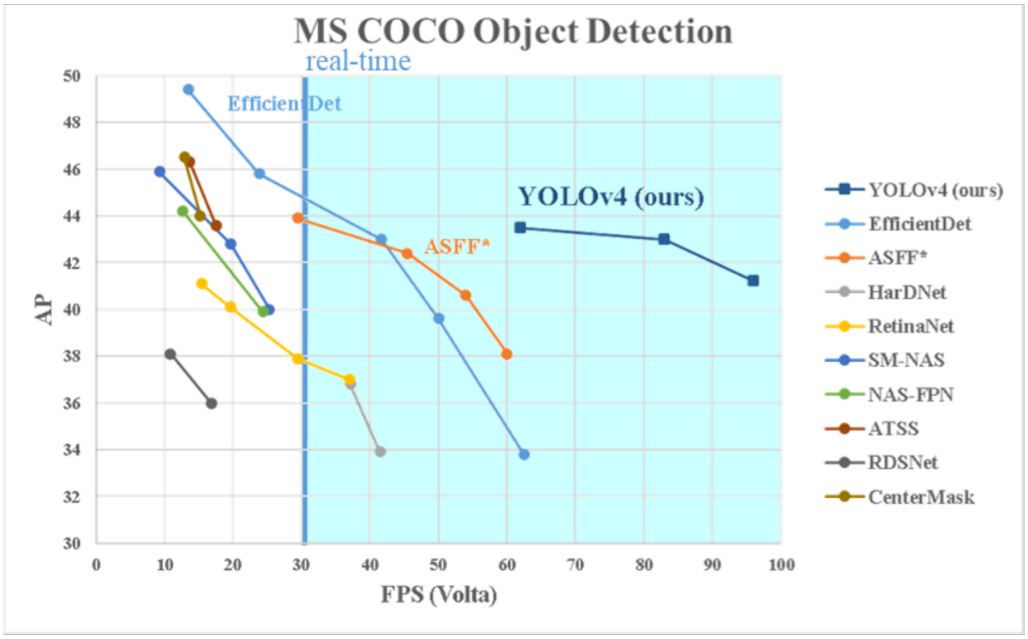


Figure A.5: Another graph comparing networks from the YOLOv4 paper [6]. Average precision versus inference time on the COCO dataset.

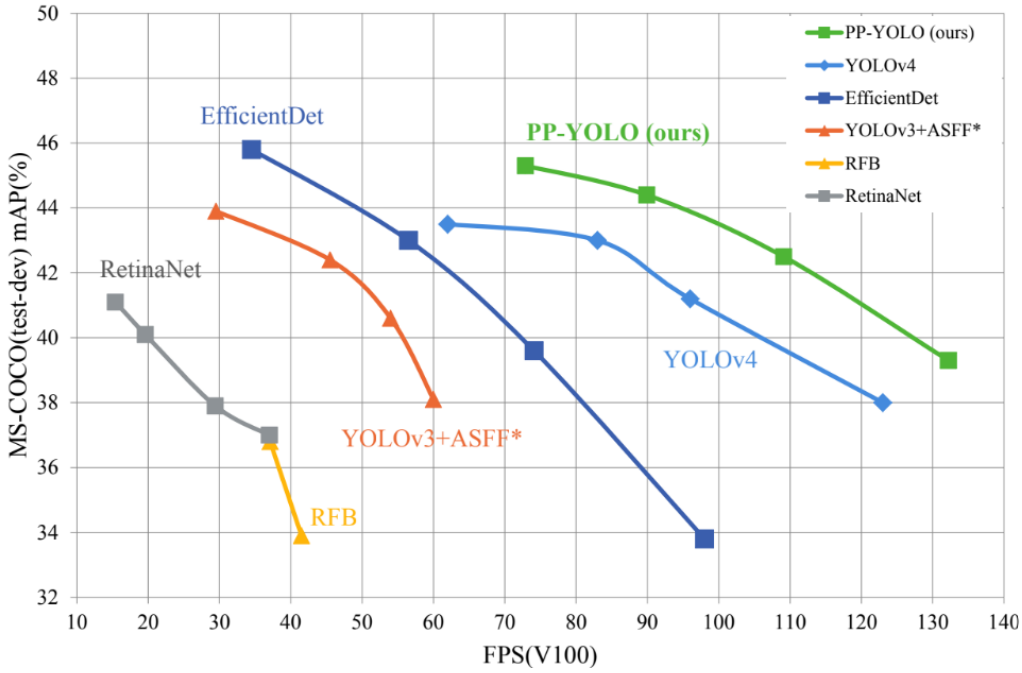


Figure A.6: Graph comparing networks from the paper covering PP-YOLO [20]. Average precision versus inference time on the COCO dataset.

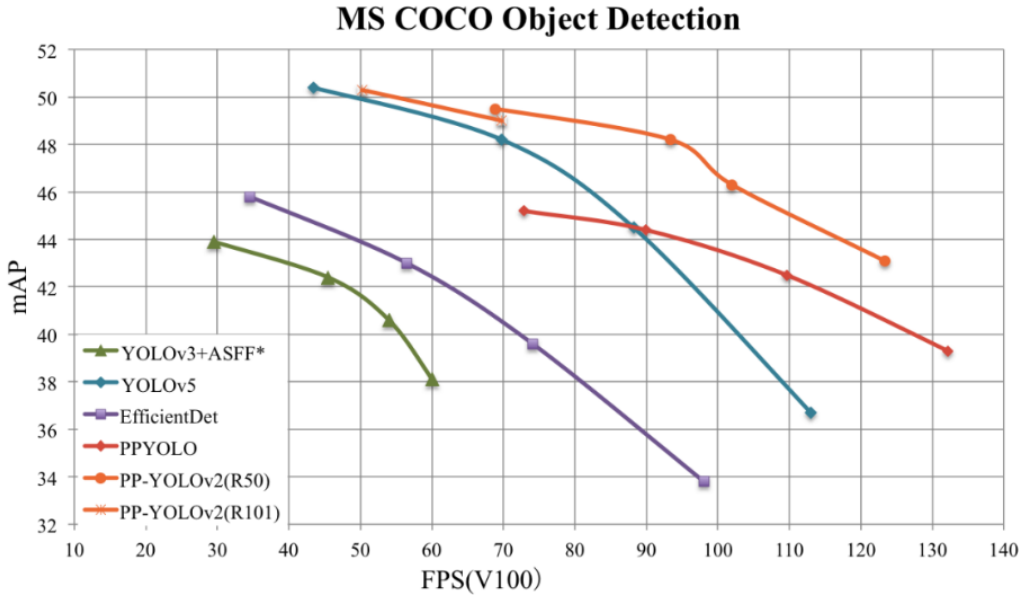


Figure A.7: Grapgh from the PP-YOLOv2 paper [67]. Average precision versus inference time on the COCO dataset.

A.4 Simple CNN model architecture

The architecture of the simple CNN model used in this project can be found in Table. A.4.

Layer
2D Convolution, filters: 24, kernel size: 3, strides: 2, activation: relu, padding: same
Max pool, pool size: 3, strides: 2, padding: same
Batch normalization
2D Convolution, filters: 144, kernel size: 3, strides: 2, activation: relu, padding: same
Max pool, pool size: 3, strides: 2, padding: same Batch normalization
Global average pooling
Dense, 512 neurons, activation: relu
Batch normalization
Dense, 256 neurons, activation: relu
Dense, 2 neurons, activation: None

Table A.2: The simple CNN model used as a student model during knowledge distillation. The number of parameters used in the model is 240,722.

A.5 Set of results for knowledge distillation using YOLOv5

A selected set of tests for knowledge distillation are listed in Table A.3 and their results in accuracy are shown in Fig. A.8.

i	PT	Teacher	KD-location	λ	w	P	R	mAP @.5	mAP @.5:.95
1	No	None	-	-	-	0.783	0.666	0.731	0.467
2	Yes	None	-	-	-	0.789	0.701	0.761	0.490
3	No	YOLOv5s	FM[0,1,0,0]	0.01	400	0.786	0.626	0.705	0.440
4	Yes	YOLOv5s	FM[0,1,0,0]	0.01	400	0.797	0.665	0.740	0.473
5	Yes	YOLOv5l	FM[0,1,0,0]	1	400	0.730	0.433	0.516	0.315
6	No	YOLOv5l	FM[0,1,0,0]	0.01	400	0.785	0.614	0.695	0.428
7	No	YOLOv5l	logits	0.01	400	0.788	0.697	0.761	0.488
8	Yes	YOLOv5l	FM[0,0,0,0]	0.01	400	0.685	0.417	0.470	0.289
9	Yes	YOLOv5m	FM[0,1,0,0]	0.01	400	0.788	0.658	0.728	0.459
10	Yes	YOLOv5l	FM[0,0,1,0]	0.01	400	0.773	0.615	0.695	0.428
11	Yes	YOLOv5l	FM[1,1,1,0]	0.01	0	0.792	0.709	0.771	0.499
12	No	YOLOv5l	FM[1,1,1,0]	0.01	0	0.798	0.669	0.744	0.478
13	No	YOLOv5s	logits	0.01	0	0.784	0.658	0.729	0.466
14	No	YOLOv5l	FM[1,1,1,0]	0.001	0	0.79	0.678	0.745	0.479
15	No	YOLOv5l	logits	0.1	0	0.77	0.597	0.675	0.406

Table A.3: A list of tests on knowledge distillation. Evaluated on a test set with 28,884 images and 12,416 boat labels from the Kaggle Airbus challenge. All models are trained for 80 epochs and evaluated with a batch size of one. i is the test index. "PT" stands for pre-trained weights from the YOLOv5 repository (not trained on the Airbus dataset) and the teacher weights are trained on the Airbus dataset. KD-location shows whether feature maps or logits were used. FM stands for feature maps where the four 1/0 indicates if the feature map is used or not. λ is the knowledge distillation multiplication factor and w is the number of warm-up steps (counted per batch).

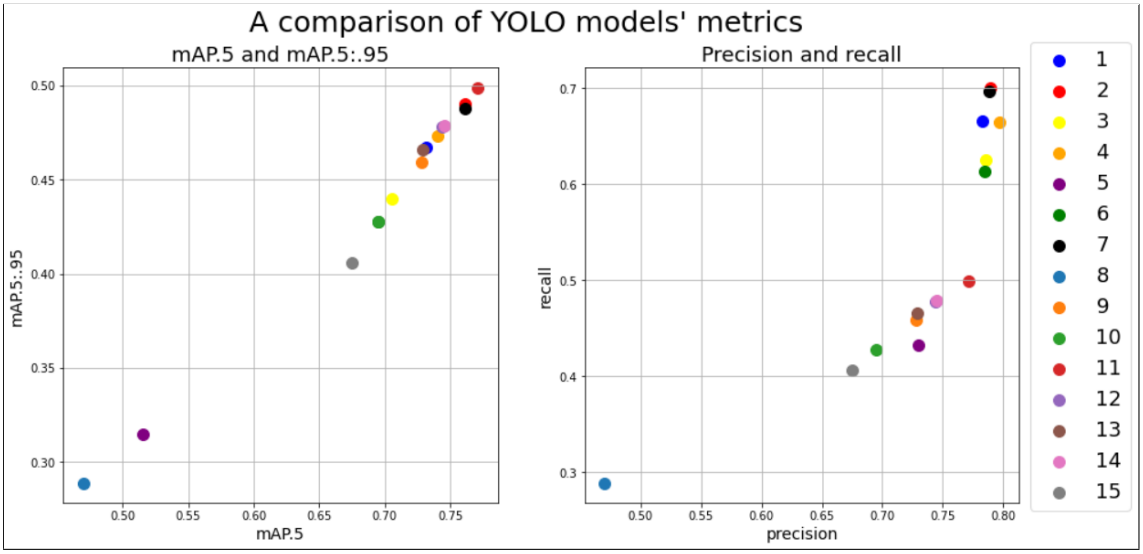


Figure A.8: A visualization of the results is found in Table A.3 for the comparison of different YOLOv5 models. The legend above corresponds to the index column in Table A.3.

DEPARTMENT OF MECHANICS AND MARITIME SCIENCES
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden
www.chalmers.se

CHALMERS
UNIVERSITY OF TECHNOLOGY