Master's thesis

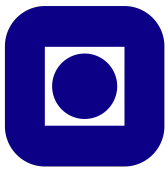
NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Computer Science

Jørgen Boganes

Accelerating Object Detection for Agricultural Robotics

Master's thesis in Electronic Systems Design and Innovation
Supervisor: Magnus Jahre
June 2020

NTNU
Norwegian University of Science and Technology



NORWEGIAN UNIVERSITY
OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Accelerating Object Detection for Agricultural Robotics

BY Jørgen BOGANES

SUPERVISED BY:
Magnus JAHRE

June 6, 2020

Acknowledgements

The author would like to thank Wiig Gartneri AS, specifically Frode Ringsevjen, for the tour of their grounds, for showing the ropes of tomato growing and harvesting, and for the great help with gathering the data needed. Their generosity laid a solid foundation for developing a realistic model for object detection. Thanks are also given to the High Performance Computing (HPC) Group at the Norwegian University of Science and Technology (NTNU), for giving me access to their supercomputer, Idun [1]. Finally, I would like to thank my advisor in this project, Magnus Jahre.

Assignment Description

The assignment description is as follows:

> Conventional farming relies on human labor for a variety of tasks – many of which are time-consuming, poorly paid, and physically strenuous. An alternative approach would be to automate these tasks with robots or other computing systems. A critical component of such agricultural systems is machine learning models that analyzes video feed(s) to gather information or take action (i.e. find tomatoes and pick the ripe ones).
>
> In this thesis, the student will use data sets generated in prior work to investigate complexity versus accuracy trade-offs in agricultural applications. First, the student should identify machine learning models that achieve acceptable accuracy. Then, the student should assess the computational and storage overhead of these models during inference and reason about how well the models fit with the computational capabilities of suitable embedded systems. If time permits, the student should implement and evaluate a proof-of-concept system on an FPGA-accelerated platform.

Abstract

In agricultural technology – or *agritech* – harvesting ripe fruit is a costly and time consuming process. This is usually done by human laborers, and agritech is thus a field where automation has a lot of potential. However, there is currently a lack of efficient and cheap ways for greenhouse farmers to automate these types of processes. Relevant literature describes a plethora of ways to detect ripe fruit on and off the vine – often employing advanced techniques, utilizing non-conventional equipment and massive amounts of computational power. For the average farmer, a cheaper and more manageable system is desired. But the most advantageous way of going about developing such a system is not always apparent – finding it can take a lot of time, and can get very expensive.

In this thesis, we attempt to create an accurate machine learning model for an agritech scenario, with the aim of accelerating it on suitable embedded systems. This is achieved by first using transferred weights from a pre-trained neural network architecture, and then training the model further on custom data. This data consists of ripe clusters of Piccolo tomatoes, and was gathered in a greenhouse under controlled light conditions. The methods presented in this thesis achieve a maximum object detection accuracy of 90%.

Four different hardware solutions are then theoretically examined, with an end goal of deploying the model to the most suitable of them. The model is able to run comfortably on all of them, according to reasonable requirements that were set based on the particularities of the task at hand, including metrics such as inferences per second, power consumption, and complexity of development. The thesis concludes that running such a model on a Field-Programmable Gate Array (FPGA) would likely result in the least amount of latency, but the tremendously complex development required when mapping such models to FPGAs suggests that deploying the model on a simpler System-on-Chip (SoC) solution, such as one from the NVIDIA Tegra series, would give a satisfying result, while remaining less complex.

Sammendrag

Innenfor agrikulturell teknologi – eller *agritech* – er det å høste inn frukt en dyr og tidkrevende prosess. Dette er vanligvis utført av menneskelig arbeidskraft, og agritech er derfor et felt hvor automatisering har stort potensiale. Per i dag ser man en mangel på effektive og billige måter man kan automatisere denne typen arbeid på. Relevant litteratur beskriver en mengde metoder som kan brukes for å gjenkjenne frukter. Disse er bruker vanligvis svært avanserte metoder og ukonvensjonelt utstyr, og bruker massive mengder datakraft. For gjennomsnittsbonden er et billigere og mer overkommelig system derfor ønskelig. Det er dessverre vanskelig å finne den absolutt beste måten man kan utvikle et lignende system på, og det å undersøke dette videre kan ta mye tid, og kan bli usedvanlig dyrt.

Denne avhandlingen prøver derfor å lage en nøyaktig maskinlæringsmodell for bruk innen agritech, med et mål om å aksellerere den på et passende innvevd system. Dette er oppnådd ved å overføre parameterene fra et ferdig trent nevralt nettverk, for å så trene videre på egen data. Denne dataen består av modne klaser med Piccolo-tomater, og ble filmet i et drivhus under kontrollerte lysforhold. Metodene som blir presentert i denne avhandlingen oppnår en maksimal nøyaktighet i objektdeteksjon på 90%.

Etter dette blir fire forskjellige maskinvareløsninger undersøkt teoretisk, med et mål om å kjøre modellen på den mest passende av dem. Modellen klarer å kjøre uten problemer på alle fire, og oppfyller som regel alle krav som ble satt, basert på oppgavens omstendigheter. Disse inkluderer antall bilder analysert per sekund, effektforbruk, og hvor kompleks utviklingen er. Avhandlingen konkluderer med at å kjøre en slik modell på en FPGA mest sannsynlig ville resultert i minst mulig latens i objektdeteksjon. Den utolig kompliserte utviklingen som kreves for FPGAer impliserer dog at å heller kjøre modellen på en SoC, som f.eks en fra NVIDIAs Tegra-serie, vil gi et like tilferdsstillende resultat, uten å være for kompleks.

Contents

List of Figures

List of Tables

List of Acronyms

ABARES Australian Bureau of Agricultural and Resource Economics and Sciences

ASIC Application-Specific Integrated Circuit

CCTV Closed-Circuit Television

CNN Convolutional Neural Network

COCO Common Objects in Context

CPU Central Processing Unit

CVAT Computer Vision Annotation Tool

DCNN Deep Convolutional Neural Network (CNN)

FAIR Facebook AI Research

FINN Framework for Fast, Scalable Binarized Neural Network Inference

FN False Negative

FP False Positive

FPGA Field-Programmable Gate Array

GPU Graphics Processing Unit

HPC High Performance Computing

IC Integrated Circuit

MPSoC Multi-Processor System-on-Chip

NI Near-Infrared

NMS Non-Maximum Suppression

NTNU Norwegian University of Science and Technology

mAP Mean Average Precision

OOR Out of Reach

OpenCV Open Source Computer Vision Library

PoC Proof-of-Concept

PR Precision/Recall

PYNQ Python Productivity for Zynq

R-CNN Regions with CNN Features

RGB Red/Green/Blue

SoC System-on-Chip

SSD Single Shot Detection

STHEM Supporting Utilities for Heterogeneous and Embedded Image Processing

TD Threshold of Detection

TDP Thermal Design Power

TN True Negative

TP True Positive

TULIPP Toward Ubiquitous Low-Power Image Processing Platforms

Chapter 1

Introduction

1.1 Motivation

The utilization of modern technology in agriculture is a rapidly growing sub-field of what can be considered one of the most integral industries of present-day society [2]. Agriculture can involve massive amounts of manual labor, usually consisting of menial and repetitive tasks. These are not only boring and strenuous for the worker, but they are also time-consuming, and present a large cost. According to the Australian Bureau of Agricultural and Resource Economics and Sciences (ABARES), sourcing skilled human laborers is one of the most costly aspects of the agricultural industry [3]. Realizing a robot that is able to replace some of the many agricultural workers could therefore be extremely lucrative. Such a robot presents great potential for reducing the cost of operations, and to increase the general efficiency of the processes [4, 5].

Due to the varying nature of the tasks presented, creating an all-purpose robot capable of completing all of them – much like the human laborers of today are doing – would likely be infeasible. However, realizing a subsystem capable of doing one, or just a part of one of these tasks could still be very lucrative. If a trained human eye can recognize ripe fruit intuitively and almost effortlessly, a machine learning model could be able to do the same, and perhaps even do it better than its human counterpart.

Such a system could potentially lead the way to an improved workflow, and could reduce the need for skilled human workers. Due to the fact that it simply requires a one-time investment (naturally including some additional maintenance costs), rather than a constant expenditure, this type of system could save money in the long run,

One interesting path to explore is the recognition and harvesting of ripe fruit and vegetables. The point of ripening is often quite obvious, and can, in most fruits, be gauged by examining the size and color of the growth. Some times, however, for example with cucumbers, weighing is needed to accurately say whether they are ripe or not. Nevertheless, a system that could recognize and harvest some of these fruits could be very profitable for the agritech industry.

A prerequisite of this type of system would be a reliable machine learning model, that can accurately detect the fruits and vegetables that are ripe for picking. Such a system would need to run on a computer with sufficient capabilities – being able to decode a video stream, having enough memory space and computational resources, et cetera – on a platform that can accommodate its energy needs.

1.2 Interpretation of the Assignment

The Assignment Description is quite straight-forward, and there is a logical split into five different tasks. A natural and linear progression can be established from these tasks, and some are more important than others. Creating a model, and evaluating what hardware it could be deployed on, are requirements of the project. The *if time permits* part is made optional. The tasks are set up as follows:

($\boldsymbol{M_1}$) Find a model with an acceptable complexity vs. accuracy trade-off

($\boldsymbol{M_2}$) Assess computational storage and overhead of the model

($\boldsymbol{M_3}$) Reason how well the chosen model works with the capabilities and restrictions of suitable embedded systems

Also, if time permits:

($\boldsymbol{O_1}$) Implement a Proof-of-Concept (PoC) on an FPGA-accelerated platform.

($\boldsymbol{O_2}$) Evaluate this system

1.3 Research Contributions

The research presented here shows how it is possible to create a fairly accurate machine learning model that can be run on an embedded system with certain computational restrictions.

Realistic data, consisting of clusters of Piccolo tomatoes on the vine, was gathered, and a plethora of models were trained on this very dataset. The many models show empirically that object detection requires a somewhat complex model to be sufficiently accurate, especially when irrelevant tomato clusters (i.e. clusters that should not be detected by the model for various reasons) are visible in every single frame of data.

The research can show that basing ones model on a neural network architecture that is already trained on data including pictures similar to the object one wants to detect, followed by continuing the training with ones own relevant data, can be very beneficial. Such transfer learning, on a pre-trained architecture, can mitigate the negative effects of training on a relatively small dataset, and greatly reduce the training time required for achieving an acceptable accuracy. The research concludes that a complex model based on the ResNet101 backbone [6], with transferred weights from a model trained on the Common Objects in Context (COCO) dataset [7] gives the best accuracy/complexity trade-off of all the models surveyed. This then meets M_1. We also evaluate the storage and overhead required to run the chosen model, in regard to M_2.

Furthermore, the research of this thesis shows that it is indeed possible to run such a model on an embedded system, with low computational power (and thus a low power consumption), even without using the many methods of reducing the model size and increasing its inference speed. This is based on the evaluation of four different platforms, their theoretical abilities, and their previous usage in related literature. This thus meets M_3

The optional tasks, O_1 and O_2 were not attempted, due to constraints beyond the authors control in relation to the COVID-19 pandemic.

1.4 Project Outline

The project is structured as follows:

1. **Introduction**
   The first chapter presents the motivation for the assignment, how the assignment text was interpreted, and the contributions made by the research.

2. **Background**
   The background chapter contains an exploration of previously done work in the same vein as the thesis itself. Firstly, we explore what has been done in agricultural technology. Then we look at some state of the art solutions for object detection. Finally we present an overview of some efforts that have been made regarding the mapping of machine learning models to embedded systems.

3. **Experimental Setup**
   This chapter expands upon how the general experiment of this project was set up. This includes how the data was gathered and annotated, how the model was trained, what hardware and software was used, et cetera.

4. **Machine Learning Model**
   Descriptions of the models that were trained, their attributes, and their evaluation. Finally, the chapter also describes which of the models that were tested had the best accuracy/complexity trade-off.

5. **Model Results and Discussion**
   The final result of the machine learning models, and a discussion on their functionality, their complexity, and their faults.

6. **Model in Practice**
   An analysis of four possible hardware realizations for the chosen model: Two SoCs from NVIDIA, an FPGA from Xilinx, and a general industrial computer.

7. **Conclusion and Future Work**
   The work is concluded, followed by some suggestions as to how it could potentially be continued in the future.

Chapter 2

Background

This chapter is meant to give a context for the current state of the fields related to our project. First, we look on the newest developments in agritech. Then, we analyze some of the state-of-the-art techniques for object detection, that have emerged in recent years. Finally, we examine the literature to see how other researchers have tried to deploy image recognition and object detection models to hardware.

2.1 Current State of Agricultural Technology

Agritech, as previously mentioned, is an umbrella term for the technology used to improve crop yield, efficiency in growing, harvesting, and profitability in conventional farming [2]. Currently, the field is dominated by labor performed by humans, as this is cheaper and more versatile and reliable than the automated options. Modern technology can however produce robotic systems that perform all the steps needed to do simple agricultural tasks – such as harvesting, planting, watering, or administering pesticide – without necessarily sacrificing efficiency or resulting in high costs. As this part of the industry is underdeveloped, agritech is definitely a lucrative candidate for automation.

With agritech being a market in such growth, advances are continuously being made. For example, L. Grimstad and P. From [8] developed a system enabling farmers to automatize the recognizing and harvesting of various kinds of plants. The system is modular, meaning it is composed of many smaller parts that can add up to a functioning system. A farmer can put together exactly the components that suit their projects best, and change it up if need be. So if the farmer wants its robot to work in the greenhouse instead of in a field, they can change some of the modular components

instead of building an entirely new robot. The system is purely mechanical, and no recognition model was developed. Thus, their paper attempts to tackle a different problem than we aim to solve here. Our thesis is focused on the recognition of fruits on the vine, and not the logistical challenges of maneuvering in a plowed field. However, in the future, these systems could perhaps be merged together to create a fully functioning harvesting robot.

Using Deep Convolutional Neural Networks (DCNNs), I. Sa et al. developed a fruit detection model, which achieved state-of-the-art performance using advanced techniques [9]. They went with a multi-modal fusion approach, and combined the information received from many different sensors and cameras, specifically cameras supporting Red/Green/Blue (RGB) and Near-Infrared (NI) imaging. Their model was based on the Faster R-CNN methodology [10], and was trained using transfer learning with pre-trained weights generated with the ImageNet [11] dataset. The model is a general fruit detection model, and is meant to lay a foundation for further training in specific cases. The paper shows that acceptable accuracies are achieved training with as little as 54 image samples. This paper does not discuss any optimization of the computational resources, nor the memory space needed, for the model to run. The smallest model the present in the paper contains over 138 million parameters, which is a considerable amount. If we were to accelerate such a model on hardware, we would have to consider if the amount of weights is excessive, since our memory space could be very limited.

A less complex model, made to recognize tomatoes on the vine for yield estimation, was made by K. Yamamoto et al. [12]. The paper separates the life of the tomatoes into three separate stages – young, immature, and mature (ripe) – and develops an object detection model around this idea. They used a standard digital camera, supporting only RGB colors, to capture images of tomatoes for their dataset. They did not pre-process the data at all before training. The dataset gathered consisted of a total of 154 images, and using k-means clustering they achieved a precision of 0.88.

Using an RGB Closed-Circuit Television (CCTV) camera, L. Zhang et al. developed a three-layer neural network model for the recognition of cucumbers [13]. They also pre-processed the dataset, by for example removing superfluous parts of the images (the parts not containing any cucumbers), and by extracting color features. The paper only focused on cucumbers, and specifically tried to accurately find the stem of the fruit,

where it is to be cut off when harvesting. With a final accuracy of 76%, the results of Zhang et al. are questionable. In addition to this, they only tested their model on 40 images.

2.2 Object Detection

In the ever-changing scene of object detection, it is irrealizable to determine *exactly* what method and meta-architecture is best for a given purpose. Today, a lot of the top scoring architectures are often based on the project of Girshcik et al. on Regions with CNN Features (R-CNN) [14]. With the classic R-CNN, we see that training is computationally expensive, both in memory space and time taken.

Fast R-CNN [15] was then introduced, and it managed to reduce the training and testing speed, and to increase the detection accuracy. As they continued to further build this project, out came Faster R-CNN [10], an even better version of this architecture. This was improved on once again, and the most recent development from the work of Girshick et al. is Mask R-CNN. Mask R-CNN is very similar to Faster R-CNN, but it proves to slightly out-preform its predecessor, and includes a small overhead that introduces mask-segmentation [16]. Segmentation would introduce a lot of complexity in the annotation process, and could possibly cause the model to be trained so specifically that it cannot properly recognize new objects of the same kind that it is trained to detect. This is often referred to as over-fitting.

2.3 Image Processing and Object Detection on Embedded Systems

Effectively mapping a neural network – or any machine learning model for that matter – on an embedded system can be very beneficial. Optimizing the system and method of mapping can reduce cost across the board, by minimizing redundant use of hardware, and thus reducing the overall unnecessary power consumption. Possible hardware types to consider are Application-Specific Integrated Circuits (ASICs) and FPGAs. S. I. Venieris et al. show that FPGAs could *bridge the gap between power-hungry programmable architectures and fixed-function power-efficient ASICs* [17]. FPGAs are often

more energy-efficient than their counterparts, and have the ability to better utilize parallelism [18].

In the case of a robot analyzing images for objects, we desire hardware that consumes a minimal amount of power, while still being able to effectively detect relevant objects in a timely manner. We are not looking for a maximum amount of computations per second. The crux of the matter is to minimize the latency of the system, i.e. the time taken to analyze an image for objects.

In 2016, T. Kalb et al. [19] presented TULIPP, short for Toward Ubiquitous Low-Power Image Processing Platforms. Their work aims to develop a platform that can define implementation rules and interfaces to mitigate issues related to power consumption, while guaranteeing performance. This is specifically made for applications specialized for image processing. This will allow developers to adhere to the big three system requirements for embedded systems: size, weight, and power consumption. Their aim is to set up an ecosystem and continue work with the organizations responsible for standardization, and to derive suggestions for new industry standards.

The tool-chain component of the Toward Ubiquitous Low-Power Image Processing Platforms (TULIPP) platform, developed by A. Sadek et al., is called Supporting Utilities for Heterogeneous and Embedded Image Processing (STHEM) [20]. STHEM presents a set of components that aim to increase the productivity of the programmer, by making the development of low-power image processing system easier.

A comprehensive analysis of practical application of neural networks running on embedded systems was in 2017 done by A. Canziani et al. [21]. They concluded with four major findings. Among these, they showed that the power consumption of such a system is independent of the batch size and architecture used. This means that in a creating such a model, it is in ones interest to choose an architecture that suits ones workflow, and not necessarily to take it in to account based on computational restrictions in hardware. They also showed that the energy constraint in such a scenario is an upper bound on the maximum achievable accuracy and model complexity. The paper also presented a hyperbolic relationship between accuracy and inference time. This entails that a small increase or decrease in accuracy can change the computational time needed substantially.

H. Mao et al. proposed, in 2015, an energy efficient implementation for

real-time object detection [22]. By analyzing and mitigating the bottlenecks in the process, they developed a pipelined system, which was realized by using the Central Processing Unit (CPU) and Graphics Processing Unit (GPU) in tandem. With their set-up, they were able to run Fast R-CNN with 1.85 inferences per second.

In the more recent years, improvements have of course been made. Although it has been shown that solutions based on FPGAs and ASICs are more energy efficient [23, 24], they are often harder to set up than on a normal computer. Mapping a complex neural network to an FPGA can be incredibly difficult [25], and because of this, many tool-chains and methodologies are currently being developed to simplify the process.

In 2019, A. Sharma et al. implemented different convolutional neural networks on a Zynq-based FPGA [26], specifically to be used for real-time object detection. The paper showcases two different meta-architectures, Single Shot Detection (SSD) and Faster R-CNN. This is realized using Python Productivity for Zynq (PYNQ) [27]. PYNQ is an open-source project by Xilinx, facilitating python development on their Zynq line of embedded systems. The paper confidently concludes that a Faster R-CNN architecture can run comfortably on such systems, with good accuracy, analyzing around 17 images per second, utilizing a model consisting of approximately 10 million parameters.

A methodology to facilitate the mapping of machine learning models to hardware was developed by M. Wielgosz et al. in 2019 [28]. They especially focused on FPGAs, and their main focus for the mapping was latency reduction. They managed to run a very simple three-layer neural network with a latency of only 210ns. FPGAs will most likely give better results regarding latency than most other types of hardware, but it can be very difficult to program. As D. Bacon et al. clearly conclude in their paper on FPGA programming: *the programmability of FPGAs must improve if they are to be part of mainstream computing* [25].

Another interesting development to consider is the projects of Y. Umuroglu et al. They present a Framework for Fast, Scalable Binarized Neural Network Inference (FINN) [29, 30]. This framework aims to automate the creation of inference engines on FPGAs. One can simply input a description of a neural network, and FINN will optimize it based on platform, design target, and specific precision. So even if FPGAs are much harder to program than the alternatives, many tools exist that can simplify

deploying a model to an FPGA.

Chapter 3

Experimental Setup

The following chapter gives an in-depth overview of the factors that remain constant for all the experiments conducted. This includes what data will be used to train the models, how the data will be prepared for training, and on what hardware the models are to be trained.

3.1 Dataset

3.1.1 Data Gathering

All the data used for this project was gathered by the author on location, at the premises of Wiig Gartneri, in October of 2019. The work related to data gathering and annotation was done by the author for a preceding project, in the autumn of 2019 [31]. The dataset used to trained the models is a video consisting of 950 frames, showing an aisle of plants growing clusters of Piccolo tomatoes. The video was filmed on a trolley going through the aisles of the greenhouse. This trolley moves along on rails, while the workers continually perform their many tasks on the plants. For example, they remove superfluous basal shoots[1], and aid the plants in coiling themselves around a sort of wire hanging from above. All this is done to allow the plants to grow optimally. These tasks, including the harvesting of ripe tomatoes, happen while the trolley moves at a constant speed along the 55 meter long aisles. They take about 50 seconds to drive along one aisle, so we can assume an approximate velocity of 0.91 meters per second. The different tasks are not done simultaneously, but the trolleys move at the exact same

[1] Basal shoots are various kinds of stems that grow from adventitious buds on the base of a tree or shrub, or from adventitious buds on its roots.

(A) Frame 200 (B) Frame 250 (C) Frame 300

FIGURE 3.1: Three Arbitrary Frames From the Dataset

velocity during the harvesting as during the trimming and tending of the plants. The trolley is approximately 50cm wide.

The camera was kept in the exact same position in all frames, to retain the same angle. The potential variations of lighting in the greenhouse over this distance is negligible. The velocity itself was also constant over all frames. This speed is however not all that important, as long as it is constant. It must also be slow enough for it to be possible to distinguish all the individual tomato clusters in every frame. If the camera moves too fast, the frame could experience motion blur, and it would be difficult for both for the annotator and the model to accurately find tomatoes. Figure 3.1 shows three arbitrary frames from this video, where each of them are 50 frames ahead of the one before it. Every single frame from the dataset contains *at least* one tomato cluster, so no frames are without objects.

The data is sorted chronologically, based on the video stream, so that the first picture in the dataset is the first frame of the video, et cetera. The first 750 pictures make up the training set, and the remaining 200 frames make up the test set. This is about a 20/80 split. All the videos were filmed using an Apple iPhone XS [32], at a frame-rate of 30 frames per second, with a resolution of 1080×1920.

3.1.2 Data Annotation

The goal of annotation is effectively to separate the objects we want to recognize into a set of features, which can vary based on what fruit we are examining, and at what stage of ripeness it is. Such features usually include size, shape, color, et cetera. The methodology of the annotation was to select clusters that were in the same aisle as the trolley, and that were ripe enough to harvest. In Appendix A one can find more info on what determines whether a tomato gets annotated of not.

In annotating the clusters of tomatoes, we have two options: either drawing a complete polygonal mask around them, or just selecting them with a rectangular bounding box. These boxes are shown in Figure 3.2, and the polygonal masking is shown in Figure 3.2b. The more detailed masking method would significantly increase the time it takes to complete the annotation, and it would present a much greater risk of over-fitting the model [31]. For our project, the exact dimensions of the clusters are not necessarily what we are after. We care more about whether of not there are any relevant tomato clusters on the frame that is being analyzed, and approximately where on the frame they reside. The tomatoes in this project are therefore annotated using rectangular boxes, as shown in Figure 3.2a.

When referring to object detection throughout this thesis, we are referring to the object detection using bounding boxes, and not object segmentation using masks. All frames are annotated using Computer Vision Annotation Tool (CVAT) from Open Source Computer Vision Library (OpenCV) [33].

3.2 Model Training, Evaluation, and Analysis

All the training, evaluation, analysis, and generation of images, was done on Idun – A cluster consisting of NVIDIA Tesla V100 GPUs [34]. The usage of this powerful machine is courtesy of the NTNU Department of Computer Science [35].

3.2.1 Training

The models presented in this thesis are to be trained on one GPU, as was presented in Section 3.2, for 168 hours each. This will result in them training for a specific amount of epochs. Every such epoch is a single forward and

(A) Rectangular selection (B) Polygonal masking

FIGURE 3.2: Different Types of Annotating for Objects

backward pass of all the training examples. After each epoch, a checkpoint model is generated. It will be on these checkpoints that we run the evaluations. Since training time is not constant between the models, we will evaluate them at the same epoch checkpoint.

As the model is training, it will output a number after every step, representing the cumulative weight of the errors made by the model during its validation. This number is know as the loss number. It is favorable to train our model until this number converges, i.e. until the difference in loss between two sequential steps approaches zero. As we do not have unlimited resources or time, we have to set a maximum cap for training time, and it is very possible that this cap is reached before the loss number converges completely.

3.2.2 Evaluation

There are many different ways to evaluate the performance of a machine learning model. It is common to check how the model holds up to other types of models that are already made on general datasets [15, 10, 16]. However, creating a general model is not our goal in this project. Our aim is a very specialized model, and the performance we seek is necessary only for our specific dataset. Therefore, we are only attempting to fine-tune it based on our specific case. We must therefore evaluate our model on our own data. A good method of doing this is via cross-validation. This means, as we

leave part of the dataset out of our training (per the discussion found in Section 3.1.1), the remaining data will be used to test the performance of our model. We will then calculate the Mean Average Precision (mAP).

The mAP is a metric that is based on the *precision* and the *recall* of a model. In this context, precision refers to the models ability to predict only the objects that it is trained to detect: It is the percentage of True Positive (TP) predictions. If our model were to always be correct when detecting tomatoes, the precision value would end up being 1. Subsequently, if it only detects tomatoes that are not there, i.e. False Positives (FPs), the value would be 0. The precision is calculated as shown in Equation 3.1. Here, a TP is the proportion of True Positives in all the data that is predicted to be positive. The proportion of False Positives are subsequently referred to as FP. Appendix B contains more information on the relation between true or false positives and negatives.

$$\texttt{Precision} = \frac{TP}{TP + FP} \tag{3.1}$$

The recall can be described as the fraction of objects present in the image that are correctly predicted as actual objects. It is calculated as shown by Equation 3.2.

$$\texttt{Recall} = \frac{TP}{TP + FN} \tag{3.2}$$

Plotting these two – the precision and recall – against each other results in the so-called Precision/Recall (PR)-curve. The graph in Figure 3.3, shows an example plot of the precision versus the recall.

If we calculate the area under the PR-curve we get the Average Precision. The evaluation program performs its analysis by examining an image, looking at both its actual bounding boxes and the predictions of the model, and it computes the mAP across all these images. This metric will vary from epoch to epoch. As described in Section 3.2.1, evaluations will be done on the checkpoint at the end of every training epoch. Out of these evaluated epochs, we select the checkpoint that presents the highest resulting mAP. By analyzing the progression of the mAP over the epochs, we can also see whether the model has reached it level of maximal performance. This happens if the mAP starts decreasing.

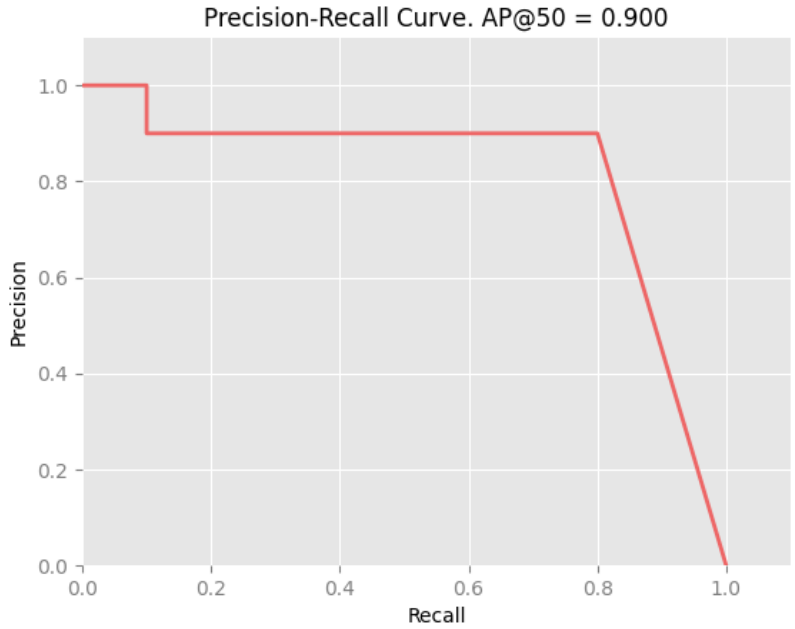


FIGURE 3.3: A Precision/Recall-Curve

3.2.3 Error Analysis

Analysis of outliers and problematic tomatoes will be done by generating an image with both the true bounding boxes, and the predicted ones drawn on it, followed by analyzing the result. We can gather a lot of useful information this way. Depending on how the predictions of the models are erroneous, different tweaks can be made to avoid making the same error in the next iteration. We utilize a slightly altered version of the methodology proposed by Hoiem et al. [36], and separate into three main types of detectional errors:

Type I Error:
A Type I error is a False Positive (see Table B.1), where the model detects a cluster of tomatoes in a place where there in reality is none. Figure 3.4a shows a Type I error.

Localization Error:
A localization error is not so much an error *per se*, but rather an inaccuracy that is almost impossible to remove completely. The model finds the tomato cluster, but does not get the exact bounding co-ordinates right. This can be seen in Figure 3.4b. This is not a serious error, and does not really affect our end result that much.

(A) Type I Error (B) Localization Error (C) OOR Error

FIGURE 3.4: Error Types in Model Prediction. Actual Annotations Are Shown by Green Bounding Boxes, and the Erroneous Predictions of the Model Are Shown in Red

Out of Reach (OOR) Error:

An OOR error is when the model detects an object, which *is* a tomato, but which is also out of the reach of the robot. These tomatoes are much smaller than the ones closer to the camera. Thus the detection is incorrect. This is shown in Figure 3.4c. An OOR error is a sub-type of the Type I error. Hoiem et al. refer to it as a *confusion with the background* [36]. We separate the two here, since detecting an actual tomato that is out of reach is less severe than detecting a tomato where there is none.

Chapter 4

Machine Learning Models

The following chapter examines five possible solutions for tomato-detecting machine learning models, with the goal of finding the one that has the most favorable accuracy versus complexity trade-offs.

4.1 Finding an Accurate Model

We want to create a model that can detect clusters of Piccolo tomatoes with high accuracy, but using only as much complexity as is strictly necessary. We want a model that is suited to be deployed on an embedded system, and so we must aim to reduce the storage space needed and keep the number of computations to a minimum. We can gain a lot from studying the research that already has been done in literature related to object detection, and also from utilizing previously made datasets and model architectures.

4.1.1 R-CNN

R-CNNs are a collection of convolutional neural network models that are designed for object detection, developed by R. Girshick, et al [14]. An R-CNN generates region proposals based on selective search, and then processes each proposed region, one at time, using Convolutional Networks to output an object label and its bounding box. One of the most recent addition to this is the Mask R-CNN [16], which was developed as an extension of the Faster R-CNN method [10]. This was done for Facebook AI Research (FAIR) [37].

Faster R-CNN has been compared with other modern convolutional object detectors, and the results showed that it requires more GPU time for training, but usually ends up with a better accuracy [38]. Given that we are using a supercomputer [1], GPU time is not necessarily a hindrance. Mask

R-CNN takes Faster R-CNN one step further, by adding object mask prediction in parallel with the existing branch for bounding box recognition [16].

The method also supports object segmentation. The segmentation involves localizing objects to the point of pixel accuracy. This will however not be necessary for our project. In addition to this, as discussed in Section 3.1.2, annotating with a polygonal mask rather than just a rectangle would take more time than is feasible for this project.

4.1.2 Implementation of Mask R-CNN

Developing our own implementation of the methodology presented in the Mask R-CNN paper would also take up far too much time, so we intend to find an already implemented version. This will save time, and be more reliable, as an open-source version will have withstood the test of time, after being used in countless different projects before this one.

There are many such implementations of Mask R-CNN, and the one that will be used for this project is the one developed by Waleed Abdulla [39], for Matterport [40]. This open-source implementation contains scripts for training and evaluating, as well as scripts to be used for visualization – for example drawing bounding boxes on images. This is convenient and very suitable for our purpose.

4.1.3 Transfer Learning

Previous work on object detection leads us to use an already developed architecture, which has been pre-trained on a dataset similar to our own, as a foundation for our model. Continuing the training of this same model, this time on our own dataset, can give us a jump-start in our quest for a sufficient accuracy.

Continuing training in this way is often referred to as *transfer learning*, since we are transferring already generated weights into a new model. It has been shown that transferring even very distant features is often better than just using random ones [41]. Since our data is sparse, and since the individual frames of our dataset do not differ very much from one another, basing our training on pictures of tomatoes from various other angles, and in different

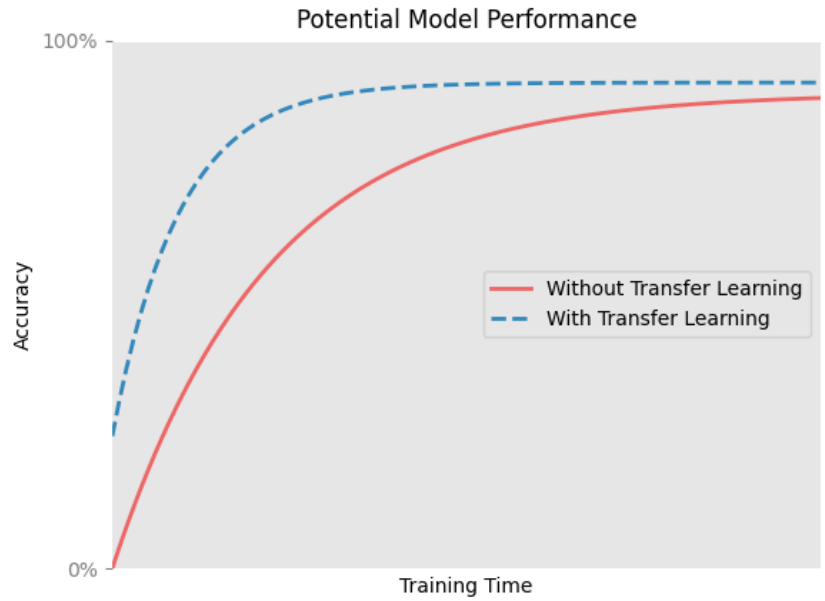


FIGURE 4.1: Idealized Theoretical Accuracy Over Time, With and Without Transfer Learning

modes of lighting, will give the deep network a greater understanding of what a tomato can look like.

There is also a potential to greatly reduce the training time needed by using a pre-trained architecture such as this – one that is already able to recognize a tomato, or a similar object (red apple, basketball, etc) very well – and fine-tune it for our specific case. This is shown by N. Kimura et al., who reduced training time by 80% by using transfer learning [42]. As the tomato is a very common object, and it is probably well-represented in many pre-made model architectures, this is a route that makes sense. As shown by the idealized plot in Figure 4.1, training using transfer learning can give the model a kick-start, so it will not have to start from scratch. The transfer learning can steepen the slope and increase the value of the asymptote (final accuracy) of the model performance.

The previously mentioned COCO dataset [7] seems suitable for our project. It contains tomatoes, as well as fruits that are similar to tomatoes, on and off branches. It also includes basketballs and other round red objects. We intend to use this architecture as the basis of most of our experiments. Figure 4.2 showcases three pictures from the dataset, all containing tomatoes among other objects. This will be a good starting point for our model.

(A) (B) (C)

FIGURE 4.2: Three Sample Images From the COCO Dataset

4.2 Hyperparameter selection for Mask R-CNN

The Mask R-CNN implementation we are using for this project presents many hyperparameters available for configuration. The complete list of these can be found in Appendix C. It is not feasible to tweak them all, in our limited allotted time, so we are only considering a few of them. These will be modified carefully as the model training goes on, and we will empirically choose the ones we find to be the most advantageous.

4.2.1 Batch size

It has been observed in practice, by N. Keskar et al., that when using a larger batch there is a significant degradation in the quality of the model as measured by its ability to generalize [43]. The authors of the Mask R-CNN implementation we are using set their batch size quite low. They used 2 images per GPU, over 8 GPUs, making the batch size 16 [16]. We will only be using one GPU in this project, so we also choose a small batch size, and set *BATCH_SIZE* to 4.

4.2.2 Max Ground Truth Instances

This parameter refers to the amount of ground truth instances to use in one image. One such instance would be one relevant tomato – i.e. ripe and ready for harvest – present in the frame. In the implementation of Mask R-CNN we will be using, this is originally set to 100. However, from analyzing our dataset, we can find the actual maximum amount of actual tomatoes in a frame, and set this parameter to whatever value that maximum is.

4.2.3 Detection Threshold

The detection threshold signifies how confident the model has to be, in that it has found an object, before it actually predicts it to be there. For the sake of simplicity, we will refer to the threshold as the Threshold of Detection (TD). Let us for example imagine that we set TD $= 0.75$. If the model then finds a cluster of Piccolo tomatoes, and is 73% sure that it is correct in its prediction, the threshold would not be met, and the model would not place a bounding box over that assumed tomato cluster. If, however, it were to be $\geq 75\%$ confident in that the cluster in question actually exists, the prediction will be set.

We will attempt to train the model with some different TDs, and see which ones work the best. The models are most likely much less confident when examining the tomatoes residing in the wrong aisles, and as the certainty of the model directly decides whether a tomato is predicted or not, an increase of the TD might discourage the model from erroneously detecting tomatoes in the wrong aisles.

4.2.4 Non-Maximum Suppression

When the model attempts to detect an object, it can sometimes propose a cluster of different boxes in the almost same location. These boxes have different degrees of certainty. If the model is very uncertain whether the box it has proposed is a True Positive (See Table B.1) or not, the model will remove the box and not predict it to be a tomato after all. To counter the clustering of bounding boxes on one object that sometimes happens, we can utilize Non-Maximum Suppression (NMS). If the resulting detections of our model present any such box-clustering errors, increasing the NMS threshold with a small increment should remove them. Figure 4.3 shows an example of what an increase of the NMS threshold can result in.

However, it has been shown, by W. Liu et al., that performing an NMS analysis can potentially increase the inference rate for an image by up to 10% [44]. We must keep this in mind, as we are attempting to minimize the inference latency of our model. The higher the NMS threshold, the more calculations will have to be done. Yet, with a high threshold one would also probably see an increase in accuracy. It is therefore desirable to run multiple

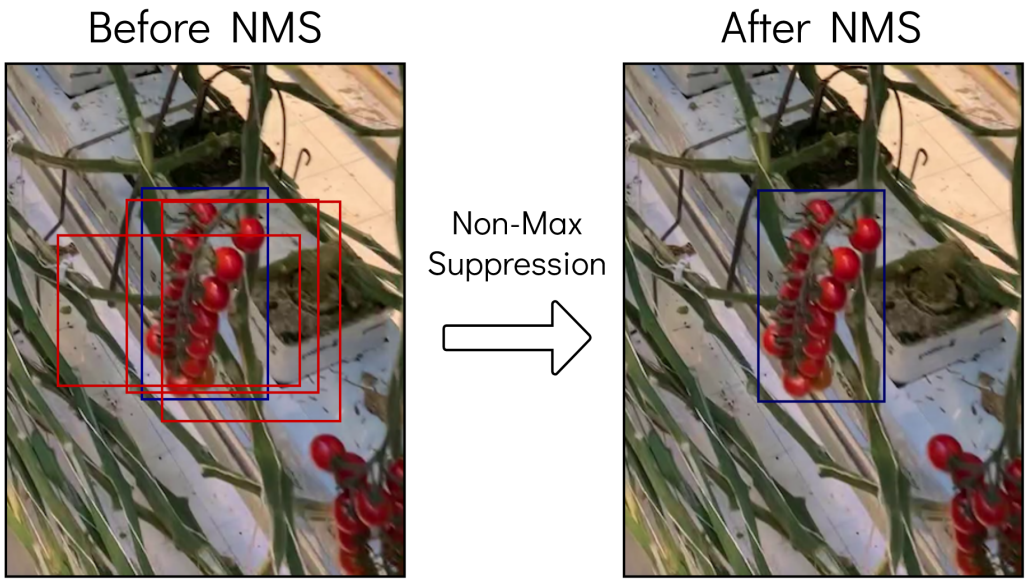


FIGURE 4.3: Example of Non-Maximum Suppression

iterations of the model, with varying NMS thresholds, to try and find a sweet-spot.

4.3 Simplifying the Model

As the Mask R-CNN based model described previously in this chapter could get quite complex, we will now attempt to simplify it. Being based on Mask R-CNN, our model is bound to have a large amount of overhead which could potentially make it difficult to store and run on a machine with limited computing power. We will therefore attempt to reduce its complexity, and see whether the reduction in size and computations needed is worth the decrease in accuracy.

It is difficult to predict ahead of time whether attempts at simplification will work to our advantage or not. It largely depends on the individual model being optimized, and cannot be done non-empirically. Therefore, we have to keep doing alterations and optimizations during the development of our model, and see which ones work the best.

This can be done in a multitude of ways, and trying all of them is beyond the scope of this paper. First, we will look at how we can reduce the overhead of the already existing complex mode, using three common

techniques. Afterwards, we will present four alternative models and see if they can be better options than the most complex one.

4.3.1 Pruning, Compression, and Quantization

Given an already-made model, we can still tweak it to run faster without necessarily losing that much accuracy. One can prune the network, which means removing superfluous or redundant parts of the network through thorough analysis. Compressing the parameters of the model can also be beneficial in reducing memory needs.

The third method, quantization, consists of reducing the decimal-point accuracy of the weights and activations of the neural network model. Quantization can greatly reduce the storage size and computational costs of the model. This is very relevant when porting the model to embedded hardware. Previous work has shown that reducing the weights all the way down to binary numbers doesn't necessarily reduce accuracy that much. Using their framework FINN, Y. Umuroglu et al. managed in some cases to achieve 10-100 times better performance, in terms of classification rates, at only a 3% (maximum) accuracy loss [29]. If we can reduce the amount of computation required to run an image analysis, we can subsequently lower the latency of our model.

S. Han et al. [45] showed that with a combination of these three methods (pruning, quantization, and compression), they managed to achieve a 35× to 49× reduction in size, without significant loss in accuracy. Using their procedure, they manages to reduce a baseline model with 240MB of parameters to 6.9MB, with only a 1% loss in accuracy. Due to time constraints, none of these techniques will be attempted in this project.

4.4 Proposed Alternative Models

In addition to the original Mask R-CNN model described above, which we will henceforth refer to as the **R101 Model**, we will train four supplementary models, in an attempt to either reduce the memory space or inference time needed, or to increase the accuracy.

The first one of these is a very simple sequential model. We will see whether it is able to detect tomatoes with a reasonable accuracy. We will then try a reduced version of the complex model, removing as much

overhead from the model as possible while while remaining sufficiently accurate. After this, we are going to train a model from scratch, similar to the R101 Model, which is without transfer learning, in an attempt to again reduce the size of the weights and activation layers without sacrificing accuracy. Finally, if none of these prove to improve upon the original model, we will attempt to fine-tune the it into a refined model, and see if this one outperforms its predecessor.

4.4.1 Sequential Model

A sequential model, with only a few layers, would take up much less storage space in comparison to the (up to) hundreds of megabytes needed by the more complex ones. One can realize a successfully functioning model while still requiring very little memory. F. Iandola et al. [46], using models as small as 4.2MB[1] achieved accuracies as high as 80.3% on the ImageNet [11] dataset. But, sequential models do not have a localization module. They can only perform classification – i.e. finding whether there is a tomato cluster in the picture or not – and not find exactly *where* in the frame the cluster is localized. This makes it difficult to train on our dataset, seeing as every frame contains at the very least 4 clusters.

However, localization is not strictly necessary. We can for example select a certain part of every frame, and analyze that part only – trying only to find whether or not there is a tomato cluster in the section, and not localizing it. An example of a possible selection could be the bottom right corner (this is the part of each image that is closest to the camera). It could also be possible to restrict the model to only analyze the parts of the image that actually can contain tomatoes. This model would naturally be significantly faster than any of the other, more complex models. If it were to work as expected, we would still end up with a model that could predict as needed, and recognize ripe tomato clusters – although only one at a time.

We thus attempt to create a somewhat accurate sequential model, with storage requirements in mind. For this project, we decide to base it on a tutorial from Keras specifically designed for small datasets with $100 - 1000$ samples [47]. Here they present a simple sequential image classification network, with an architecture as shown in Figure 4.4.

[1] The model was also compressed down to 0.5MB without significant loss of accuracy

```
model = Sequential()

model.add(Conv2D(32, (3, 3), input_shape=(512, 512, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))

model.add(Flatten())
model.add(Dense(4))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(1))
model.add(Activation('sigmoid'))

```

FIGURE 4.4: Sequential Model Layer Overview

4.4.2 Reduced Model

We can radically reduce the memory requirements of the R101 Model by further changing some of the meta-architecture. Before training on or analyzing an image, Mask R-CNN scales the image down. The standard de-scaling is down to 1024×1024. We can for example half this number, down to 512×512, to reduce the amount of pixels the model has to evaluate. Naturally, this will negatively affect the accuracy of the model. Less pixels to evaluate also means less detail and features to extract.

We can also change the pre-trained backbone used. In Mask R-CNN, the standard backbone used is the ResNet101 [6]. This is the second-largest ResNet version that is supported by the implementation, and may contain potentially superfluous overhead. We can attempt to reduce this by changing the backbone to be ResNet50 instead. Utilizing ResNet50 reduces the amount of memory needed by the model drastically by approximately 42%. More on this, and the different versions of ResNet, can be found in Table 4.1. The original paper shows that downgrading to ResNet50 only shows a slight reduction in accuracy: They lost around 0.5% to 1% depending on dataset they tested on [16]. We will refer to this model as the **R50 Model**.

TABLE 4.1: Three Versions of ResNet [6] and Their Attributes.

Version	Layers	Parameters	Memory Required
ResNet50 [48]	177	25.610.216	104MB
ResNet101 [49]	347	44.654.504	180MB
ResNet151 [50]	517	60.344.232	244MB

4.4.3 Complex Model Without Transfer Learning

Another option in the simplification of our setup is making a Mask R-CNN model from scratch, developing it without any transfer learning or pre-generated architecture, but still training it on the same dataset as before. This will most likely result in a slight loss of accuracy, in addition to a substantial increase in training time. It will however also possibly reduce the storage space and computations needed. As shown by A. Jodeiri et al., the impact of transfer learning on Mask R-CNN is significant [51]. The loss curve takes much longer to converge, and the convergence value is not as low as when using transfer learning. However, if the decrease in storage space and computations needed is lucrative enough, perhaps it is worth it after all. We will refer to this model as the **R101-Scratch Model**.

4.4.4 Refined Complex Model

If none of the alternative models presented above prove to out-perform the complex one, in terms of the trade-off between accuracy and computational time, we can, as a last resort, attempt to fine-tune the hyperparameters of the first model. If we have to use a complex model with a lot of overhead, it should be as accurate as possible. We will refer to this model as the **R101-Refined Model**. Such a refined version of the original R101 Model, that presents a higher accuracy – yet has the same amount of overhead as the first one – is definitely an improvement.

Chapter 5

Model Results and Discussion

This chapter presents an evaluation of all the five models proposed, and compares them. The chapter also includes an analysis of their errors – both errors specific to each model, and systematic errors present in all of them.

5.1 Summary of Models

The evaluated models are as shown below:

- **R101 Model**
  As described in Section 4.1.2, this first model is trained using the original Mask R-CNN set-up, without any alterations, using the ResNet101 backbone, and transferred weights.

- **Sequential Model**
  A simple and sequential model as described in Section 4.4.1, consisting of 15 layers, trained without transfer learning.

- **R50 Model**
  A reduced version of the R101 Model, but with the ResNet50 backbone, and further de-scaled input images. This is described in Section 4.4.2.

- **R101-Scratch Model**
  A version of the R101 Model, but this time starting from scratch, without transferring any weights from previously built models. Section 4.4.3 contains the description of this model.

- **R101-Refined Model**
  A fine-tuning of the R101 Model, presented in Section 4.4.4. The hyperparameters that are to be refined will be chosen empirically, based on the results obtained from the R101 Model.

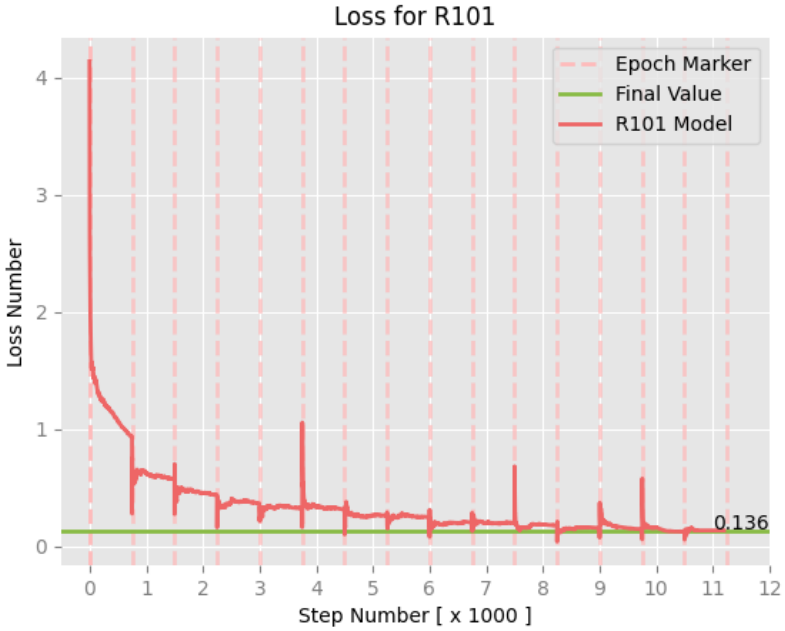


FIGURE 5.1: R101 Model Loss

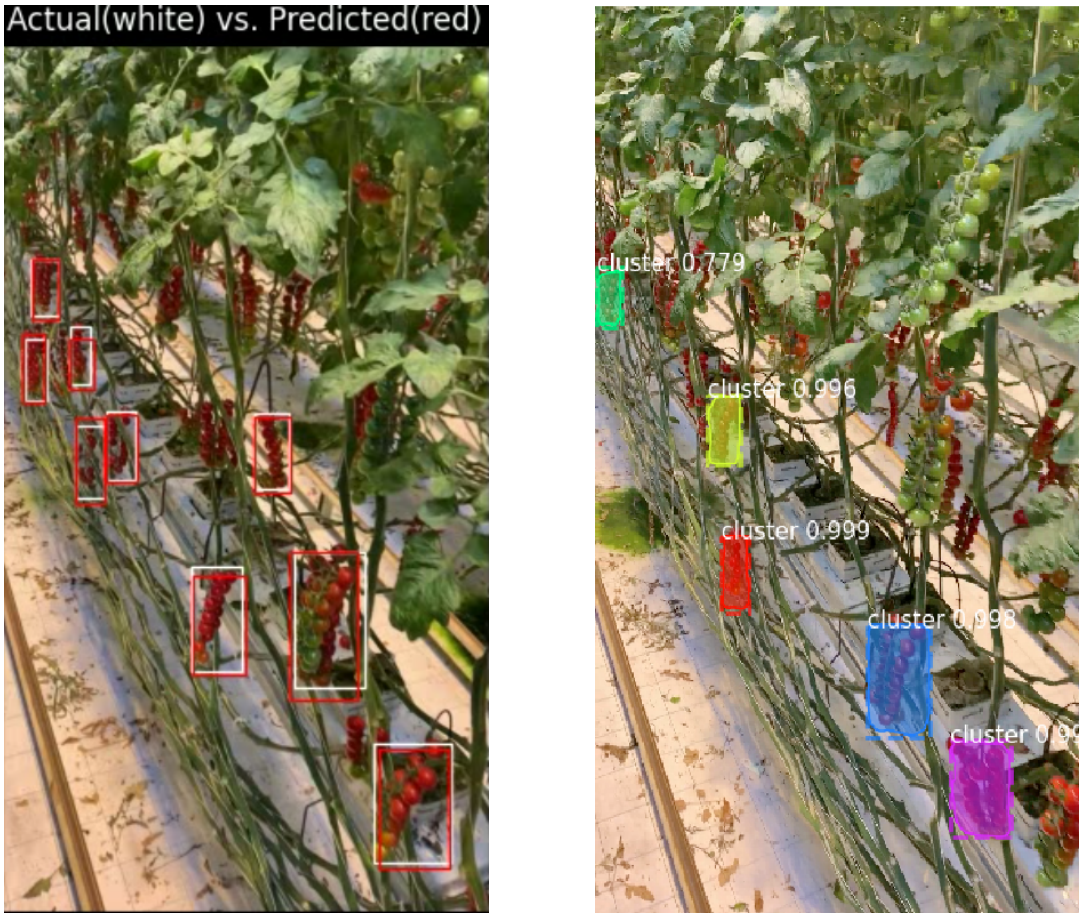
5.2 R101 Model

As described in Chapter 4, all the models except the Seqential Model are based on Mask R-CNN [16]. As was designated in Section 4.4, all the configurations of the R101 Model, other than a few select hyperparameters, are the same as the default ones, given by Matterport in their implementation of Mask R-CNN [52].

The R101 Model was trained for the aforementioned 15 epochs, and ended up taking 59 seconds per step, or 737 minutes per epoch. The resulting model seems to have a very high confidence in clusters that are alone (i.e. those not in close proximity to other clusters), and close to the camera. It does however still make mistakes.

The masking it performs, as shown in Figure 5.2b, is far from optimal. This makes sense, based on the fact that the entire annotation was done as rectangles, and not by masking, as previously mentioned in Section 3.1.2. However, the crude masking it is able to perform could still slightly increase the confidence of the model, by making the exact border of the tomato cluster a bit clearer.

The R101 Model started with a loss of 4.136, and the final loss value was 0.136, as shown in Figure 5.1. The model ended up reaching a maximum



(A) Actual Annotations versus the Predictions of the Model

(B) Masked Objects with the Degree of Confidence for Each Cluster Above

FIGURE 5.2: R101 Model Predictions of Two Arbitrary Frames

accuracy of 90%. This was achieved using a total of 63.733.406 weights. The R101 Model is quite large in size, and needs 255.9MB to store all its parameters.

5.2.1 R101 Model Error Analysis

After thorough analysis of the R101 Model and the way it predicts, some of the reasons for its non-zero rate of error become obvious. The model will occasionally erroneously detect clusters of tomatoes that are either not yet ripe, or that are on the other side of the aisle – i.e. out of the robots reach. Also, in a few instances, it seems to cluster multiple boxes on one single tomato. Based on this, we wish to alter the hyperparameters of the model to further increase the accuracy, and reduce the occurrences of the errors it displays.

FIGURE 5.3: Detection of a Tomato in the Wrong Aisle

5.2.1.1 Systematic Errors

Based on the three types of systematic error defined in Section 3.2.3, we can see that our model is working quite well. Following is an analysis of the errors inherent to the trained model.

No Type I Errors of any kind were found in the entirety of the predictions done by the finalized model. However, as expected, the finalized model repeatedly commits Localization Errors. These are, as mentioned, not very detrimental to the overall accuracy. As shown in Figure 5.2a, practically every single bounding box is a localization error to some extent. It is not feasible for a model to select the exact rectangle that the annotator selected, unless the objects themselves were to be perfect rectangles. Again, this does introduce a minuscule loss of accuracy, but it is a loss that can be neglected.

There were a couple of OOR Errors throughout the predictions, which probably are the main cause of reduced accuracy. These errors are usually tomatoes that are ripe and ready for being harvested, but that reside in the wrong aisle. Figure 5.3 shows a such a tomato being detected.

It makes sense that the greenhouses visited by the author are made with the human worker in mind. They are not quite suited for automation using robot workers yet. Between each aisle, there is a small gap of around 1 meter, where the workers drive their trolleys back and forth, doing the many tasks that need to be completed. There is, however, no visual separation set up in between these aisles. Examples of such a visual separator could for instance

be an opaque tarpaulin or large piece of cloth, or a solid wall. If the model can see through each of the aisles, we could potentially see the model make erroneous attempts in harvesting clusters residing in a different aisle than the aisle the model is currently examining.

This behavior makes sense: The size of a tomato cluster far away on the correct aisle could be around the same size as a cluster close by on a different (and wrong) aisle. As seen in Figure 5.4, a tomato on the correct aisle (A) is the same height in pixels as a tomato on an incorrect aisle (B). So, for such a model to work optimally, an element set up between the aisles, blocking the robots vision, could be advantageous. Without such a blocking element, the model would need to be much more accurate in order to work properly. If we were to alter the angle at which the robot is aiming its camera, as shown in Figure 5.5, this problem becomes even more apparent.

5.2.1.2 Non-Maximum Suppression

After meticulously looking through the images of the dataset with the prediction boxes drawn on them, we can see that the clustering of boxes described in Section 4.2.4 happens from time to time. The same section describes how this most likely will increase inference times. However, the the amount of clusters that could benefit from a higher NMS threshold are few, and therefore the increase in latency is negligible. Analysis of the clusters, and the certainty of the model in each of them, showed that the clusters usually consist of two or three boxes, where one of them has a certainty ≥ 0.9, and the others have a certainty of ≤ 0.7. This implies that setting the `DETECTION_NMS_THRESHOLD` parameter to 0.7 is a reasonable approach. No further testing of different threshold values and their results beyond that of one refined model was done, due to timing restrictions.

5.3 Alternate Models

5.3.1 Sequential Model

The sequential model, set up as it was described in Section 4.4.1, did not do very well. With its 15 layers, the model amassed a total of 4.030.441 parameters. The size of the model ended up being 15.8MB, and the maximum accuracy achieved was only 20.4%. Even with a 93.6% reduction

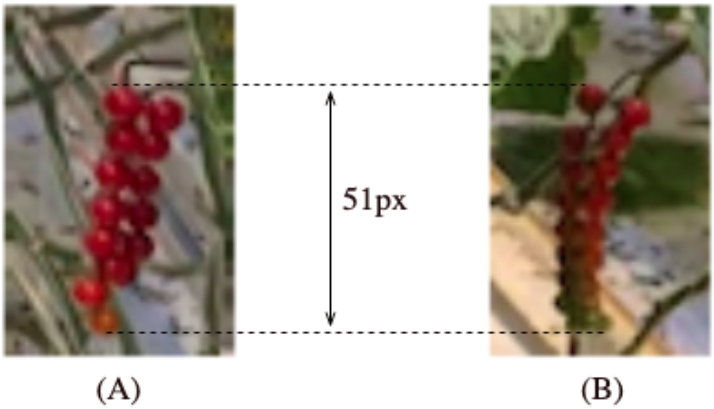


FIGURE 5.4: Equally Sized Tomato Clusters From Different Aisles

FIGURE 5.5: See-Through Aisles

in size, and thus a similar decrease in computations needed, this is still not enough to make the low accuracy worth it.

The poor results do not necessarily come as a surprise. One of the many reasons this image classifying model performing so unsatisfactorily can be the fact that there are no frames without tomato clusters present. The bad accuracy can also be explained by the lack of depth of the model. The problem we are facing, with multiple tomato clusters in each frame, is then a problem more suitable for a deeper network, capable of localization. We scrap the Sequential Model, and consider the others instead.

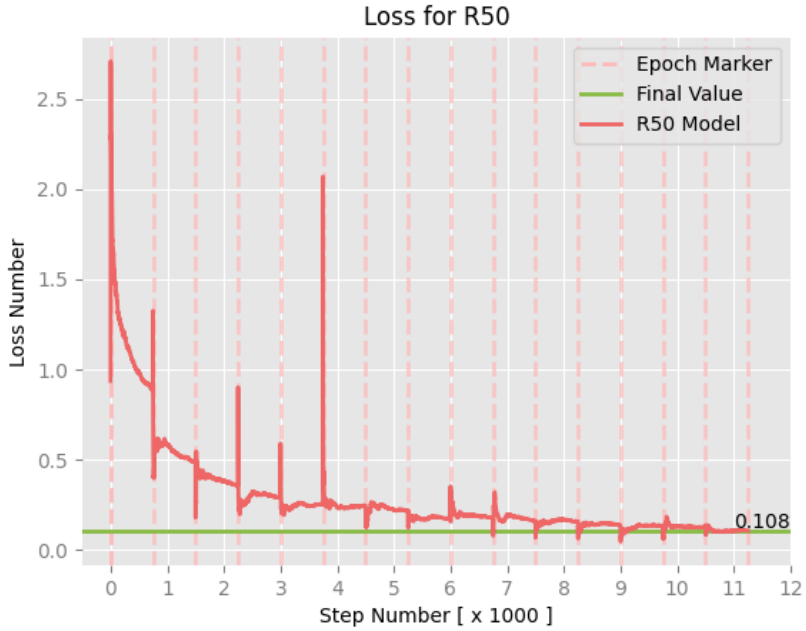


FIGURE 5.6: R50 Model Loss

5.3.2 R50 Model

The R50 Model, set up with a more light-weight backbone, as described in Section 4.4.2, reduced the storage space needed down to 179.2MB, compared to the 255.9MB required by the R101 Model. The loss of the R50 Model both starts and ends at a lower point than the R101 Model, as shown in Figure 5.6. It ended up reaching as low as 0.108. However, as previously mentioned, a lower loss number does not necessarily mean a higher accuracy. Its results support this notion, as its reduction in size decreased the accuracy down to 16.52%. This drastic reduction may be due to a far too aggressive reduction in the resolution of the input image, as was also presented in Section 4.4.2. This accuracy is far less than what we require, and thus we remove the R50 Model from our line-up as well. The R50 Model did however train at a much quicker rate than the R101 Model, most likely due to its reduced amount of weights – it trained at 39 seconds per step, or 487 minutes per epoch.

5.3.3 R101-Scratch Model

The R101-Scratch Model – which was trained without transferring weights, as described in Section 4.4.3 – ended up with a very low accuracy of 2.12%. This may be due to the lack of training time. When no weights are

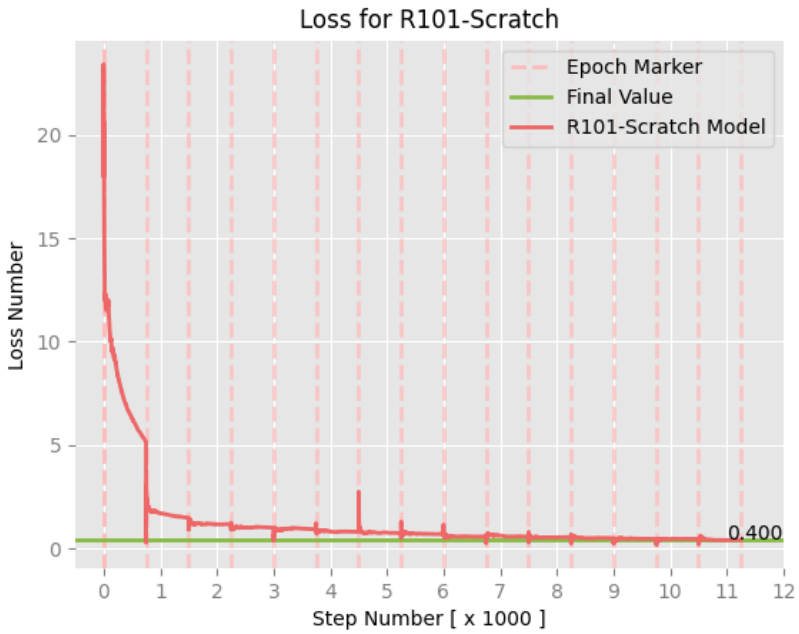


FIGURE 5.7: R101-Scratch Model Loss

transferred, it is obvious that the model needs a lot more time to train. The accuracy of the model was 0% for a lot of the first epochs, and around epoch $9 - 10$ it started gaining some (although insignificant) accuracy. Since the R101-Scratch Model is based on the same architecture as the R101 Model, the size of the model and the amount of weights are the same. It trained at approximately the same speed as the R101 Model, clocking in at 58 seconds per step, or 725 minutes per epoch.

As is shown in Figure 5.7, the loss number for the R101-Scratch Model starts at a much higher initial point than the others. The loss number for the R101-Scratch Model started at 23.42, and converged to 0.4 at the termination of training. Even though it seems to have converged based on the loss function, the R101-Scratch Model has an abysmal accuracy, and is thus also out of the question.

5.3.4 R101-Refined Model

Based on the results of the analysis previously done in Section 5.2.1, we are now able to alter the hyperparameters of the R101 Model, as was described in Section 4.4.4. We increase the threshold of the NMS significantly, from 0.30 to 0.70. In analysis of model performance, we saw an sizeable amount

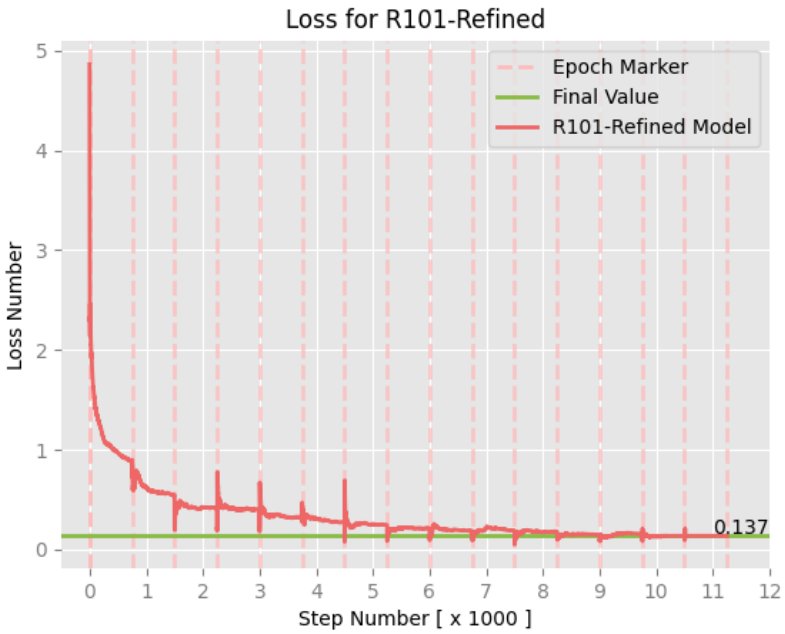


FIGURE 5.8: R101-Refined Model Loss

of bounding box clusters, similar to the ones of Figure 4.3, that justify this stark increase. In addition to this, we increase the confidence needed for a detection of a tomato cluster to count. We see that this is currently at 70%, and that some of the detections at this level of certainty are wrong. Therefore, we increase the confidence needed from 70% to 85%. We are still training the refined version of the R101 Model by transferring weights from the same source as before.

5.3.4.1 R101-Refined Model Results

Due to the massive increase in NMS operations needed, we see an increase in training time needed. The time taken per step ended up being 62 seconds, adding up to 775 minutes per epoch. This is a 5% increase from the R101 Model. The loss number started a bit higher than the R101 Model, but ended up at approximately the same level, with a loss of 0.137. The maximum accuracy achieved by the R101-Refined Model was 7.33%. This might be due to the increasing of the NMS threshold and the detection confidence parameters being too severe. It may also be due to lack of training time, or the lack of variety in our dataset. Altering these hyperparameters to lesser extents than before may prove to be more beneficial.

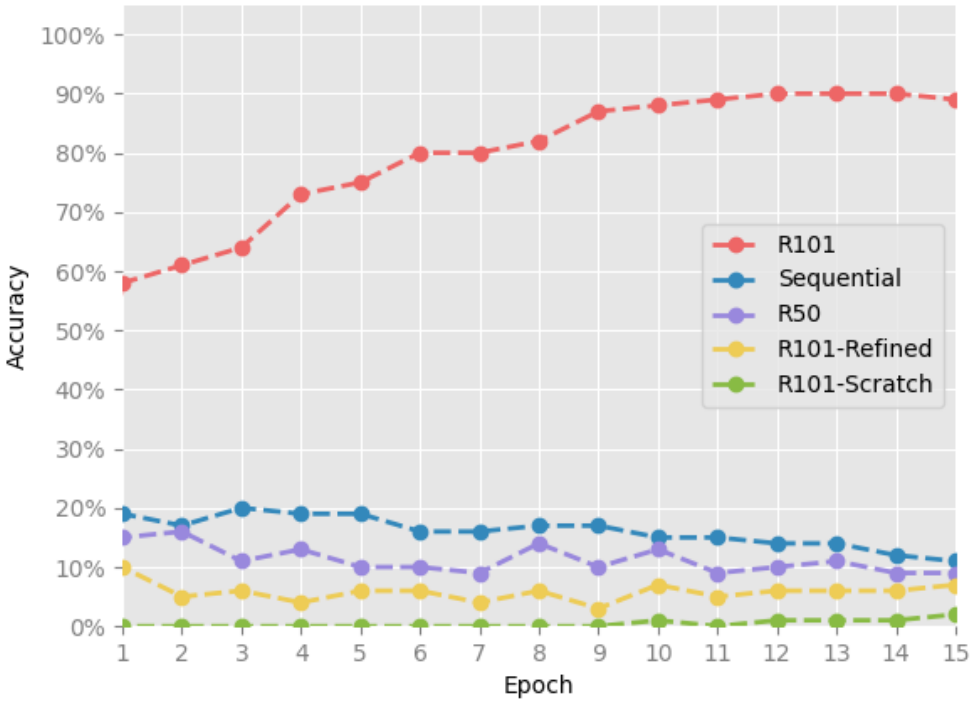


FIGURE 5.9: Comparison of the Five Models, Showing Each Models Accuracy For Every Epoch

5.4 Model Comparison

A graphed display of the accuracy of the five models over 15 epochs can be found in Figure 5.9. None of the models managed to match the R101 Model, which was based only on the Mask R-CNN architecture with its standard configurations, and which used the ResNet101 backbone. At a superlative accuracy of 90% it is by far our best option, accuracy wise. Judging by the comparison graph, it is apparent that the R101 Model reached its peak accuracy around the 12^{th} epoch, and that any further training might have been unnecessary.

A complete comparison of the five models can be found in Table 5.1. Here we can see the final accuracy of each model, their training time, their amount of weights, and the memory space needed to run them. This gives us a good overview that can be used when evaluating which one of them is the best candidate for deployment on hardware.

TABLE 5.1: Detailed Comparison of the Models.

	Time/Epoch	Final Loss	Accuracy	Size	Weights
R101	$737m$	0.136	90%	255.9MB	$63.7M$
Sequential	$31m$	0.003	20.4%	15.8MB	$4.0M$
R101-Scratch	$725m$	0.400	2.12%	255.9MB	$63.7M$
R50	$487m$	0.108	16.5%	179.2MB	$63.7M$
R101-Refined	$775m$	0.137	7.33%	255.9MB	$63.7M$

5.5 Potential Sources of Error

A source of error could be due to the annotation. The tomatoes that were selected by the annotator were not always *obviously* correct or incorrect. There is a small discrepancy in some frames that could have contributed to the model being less than optimally trained. The annotator may also have missed some perfectly harvestable tomatoes without realizing.

The lack of data is another possible source of error. However, other similar models have achieved satisfactory results using less images than the model presented in this paper. These models may also have had more varied images than those our dataset consists of. L. Zhang et al. do not disclose the amount of images their model trained on, but their test set only consisted of 40 images [13], which can imply a training set of around 150-200 images. L. Zhang et al. also tested on cucumbers, not on tomatoes. The contrast between the deep red tomatoes and the green color of the verdure surrounding them might facilitate the process of accurately identifying them.

This leads us to the lack of variation in the dataset. Almost all our frames are identical in angle, lighting, et cetera, and all the clusters that are annotated are at the same approximate stage of ripeness. There are very few clusters that are completely green. This could present a high risk of ending up with a model that has been fit too well to the training data, that only knows how to recognize an extremely specific representation of a tomato cluster. Any clusters with even just a slight variation from this exact image could then escape the model in its attempt to detect them. Over-fitting is a problem often present in projects with small and somewhat homogeneous datasets, and therefore this could be very relevant possibility for this project as well.

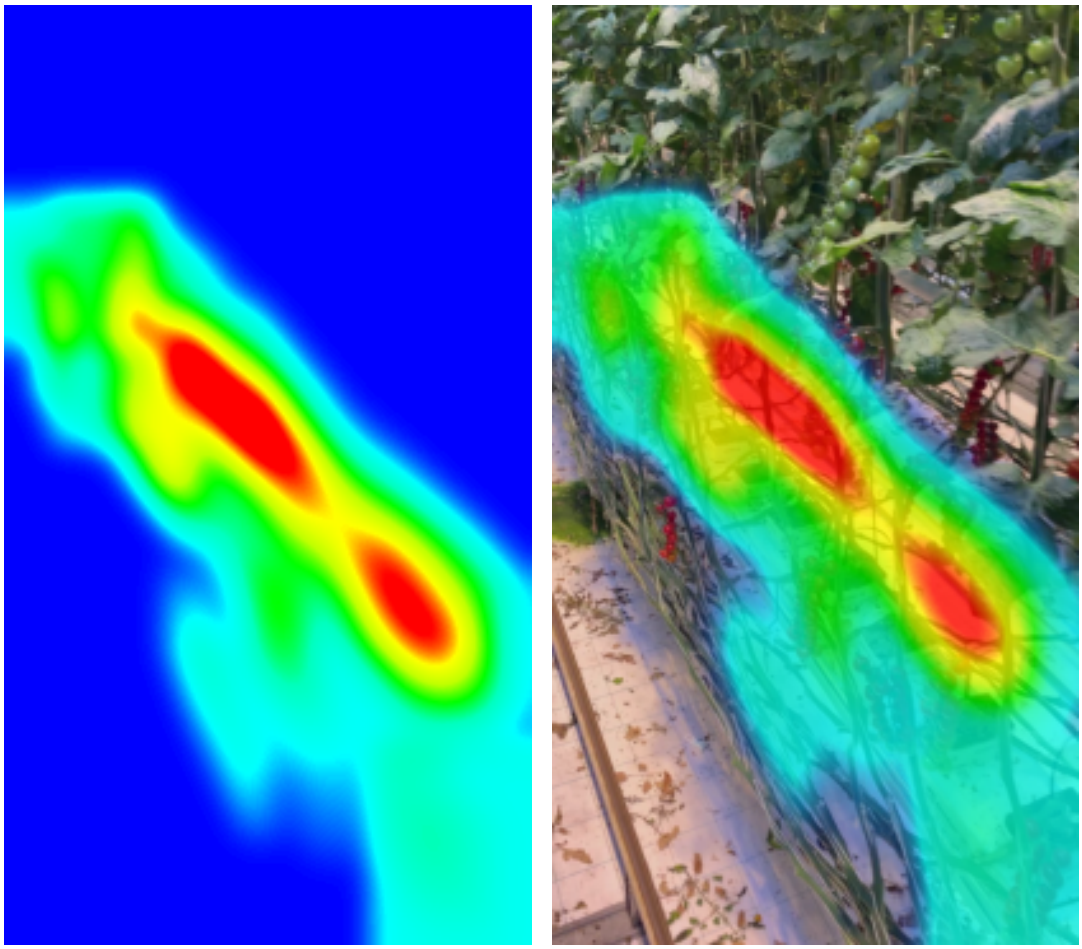
Chapter 6

Model In Practice

At this point, as discussed throughout Chapter 5, we have determined which of the five models that were trained for this project we wish to attempt to deploy to hardware. The R101 Model ended up being the most favorable one. This decision was based on the trade-off between accuracy and model size. We will now see how the chosen model could work in a real life scenario.

A robot moving around in a greenhouse would have to be battery-driven, since dragging a cable around could get very complicated. A battery is a limited source of energy, and due to this, we need to control our power consumption. The energy is defined as the power consumed over time, and with limited energy resources, we must aim for as low a power consumption as possible. Other solutions could include tapping in to the powered rails that drive the trolleys, but for this system to be general, we base our assumptions on it being battery-driven. We also want high mobility, and rails or a cable would definitely make it less mobile. In the interest of small-scale farmers, to allow them to implement similar systems, we also want to keep the costs low.

An embedded system is thus a very suitable candidate for our purpose. As mentioned in Chapter 1, utilizing such systems for deploying machine learning models is a rapidly growing field. Many of these systems have built-in video decoders, and some even have hardware accelerators dedicated to the very purpose of running neural networks. To find which such system would fit our purpose the best, we must first find the requirements of the model. We will set up a baseline, some minimum requirements, and judge the different options based on these. By keeping our assumptions pessimistic, or at the very least realistic, we can take a first step in finding the hardware platform that is most suited for testing models such as the ones we have presented.



(A) Heat-map of Annotation Placement (B) Overlain on Arbitrary Frame

FIGURE 6.1: Heat-maps of Annotation Placement

6.1 Requirements

6.1.1 Observations on the Annotated Data

There are some observations that can be gathered from analyzing the annotations and their properties – mainly regarding frequency, placement, and amount. We can generate a heat-map of the placement of annotations, to see what parts of the screen we need to focus on. In the heat-map shown in Figure 6.1, one can see that certain areas (shown in deep blue) never contain ripe clusters of tomatoes. Analyzing these parts of the frame is thus not strictly necessary.

Another interesting metric here is the average amount of annotations per frame. We can analyze the annotation files to find this number. A histogram of the annotation data can be found in Figure 6.2. On the x-axis we can see

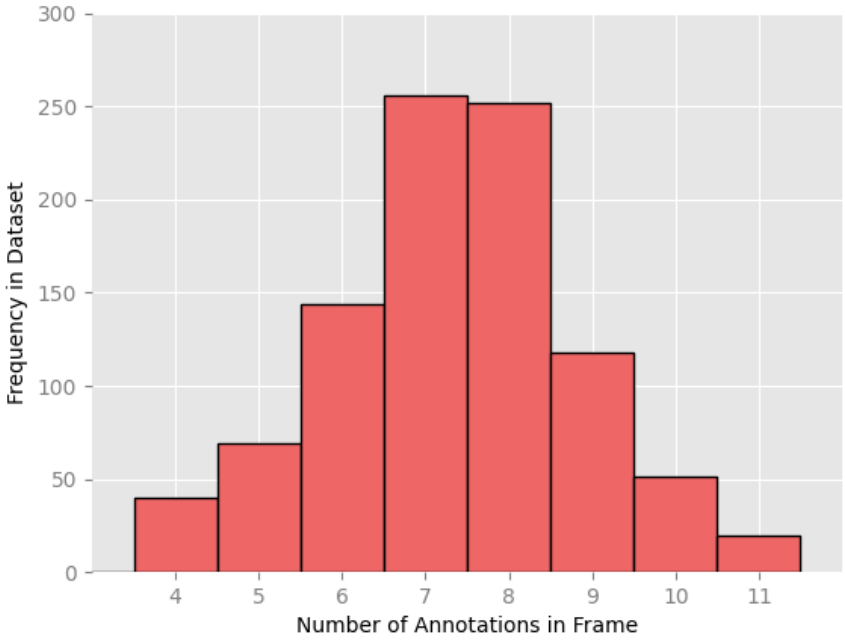


FIGURE 6.2: Histogram Showing Annotation Frequency

the different amount of objects per frame – in our case, amount of tomato clusters per frame. The y-axis shows the frequency of frames with that certain number of tomatoes. Further analysis of this data shows that the average count of tomatoes per frame is 7.335. It also tells us that the maximum number of predictions needed per frame is 11. The number of tomato clusters per meter is a fixed amount, set by the overseer of the greenhouse, and the frequency of tomatoes is thus unlikely to exceed 11[1]. Therefore, we can set the *MAX_GT_INSTANCES* parameter, as discussed in Section 4.2.2 to 12.

From this we can also find numbers on how big of an increase in tomatoes we can expect per frame. Figure 6.3 gives some insight into this. From the stack-plot in red we can see the cumulative amount of relevant tomato clusters in the dataset, which ended up being 60. From this, and the fact that there are 6 clusters in frame as the video footage starts, we can deduce the average increase per frame:

$$\frac{\texttt{number of increases}}{\texttt{total frames}} = \frac{54}{950} = 0.0568$$

[1]This was the case at the specific greenhouse where the data for this project was gathered.

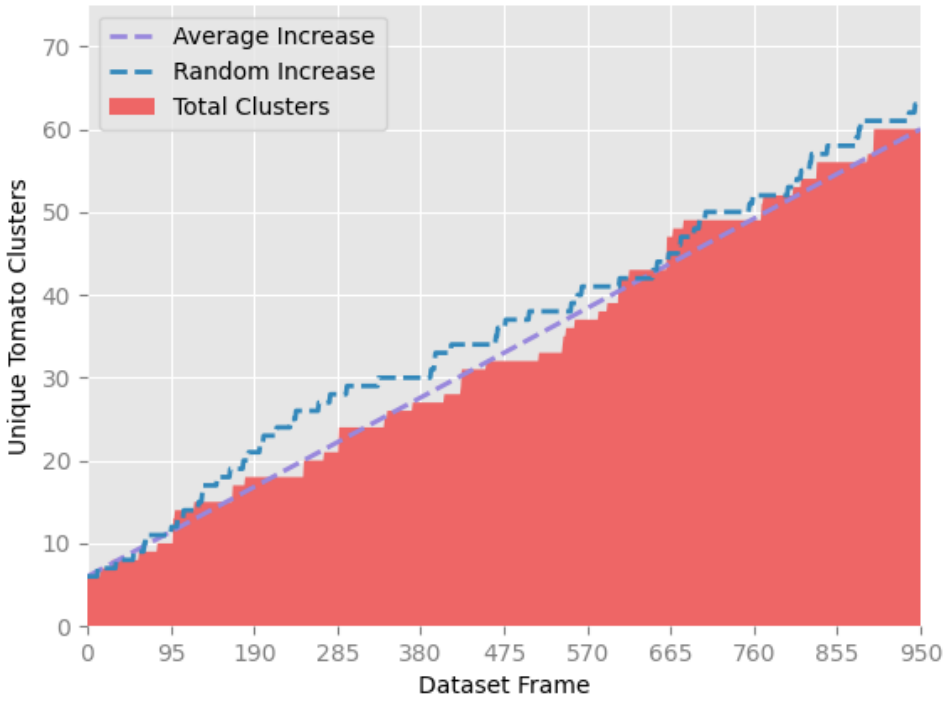


FIGURE 6.3: Cumulative Tomato Clusters in Dataset

This implies an estimated increase every 18 frames. Analysis was then done on the probabilities of new tomatoes appearing on screen. The chance of one tomato appearing ended up being 2.95%, whereas the chance of two appearing simultaneously was 0.52%. More information on this can be found in Table 6.1. A randomly generated sequence following these probabilities can be seen plotted in blue in Figure 6.3.

Based on the dataset we are working with, and how it is annotated, we can create a baseline for what the model we are to build has to be able to handle. Figure 6.2 shows that is needs to at least be able to handle 11 clusters per image. And Figure 6.3 shows that there is a non-zero probability of as many as 4 new tomatoes appearing in one frame.

When setting up the minimal requirements for our model, we want to know what level of accuracy is good enough. It could therefore be interesting to observe the probabilities involved with whether each model is able to perform the work we need it to do. This can be estimated with conventional statistics.

Using the formula for cumulative binomial distribution, shown in

TABLE 6.1: Probabilities for n New Tomato Clusters Appearing in the Subsequent Frame

n	p$(N = n)$
0	95.89%
1	2.947%
2	0.526%
3	0.421%
4	0.105%
≥ 5	0.000%

Equation 6.1, we can find the probability for a model of a certain accuracy to find a tomato with a given amount of predictions done per second.

$$P(X \geq x) = \sum_{x=X}^{n} \binom{n}{x} \cdot (p)^x \cdot (p)^{n-x} \tag{6.1}$$

Here we have the variables x, n, and p. The variable x is the amount of tomatoes we aim to find. The n signifies the amount of attempts we use in order to find them. And the p represents the accuracy of the model in question. As an example, let us imagine a model with and accuracy of $p = 30\%$, and an inference rate of 2 per second. It attempts for 7 seconds, and thus analyzes $b = 14$ frames, to find a tomato cluster at least $x = 1$ time, the probability of it actually finding the tomato would be:

$$P(X \geq 1) = 0.9932 \approx 99.3\%$$

This looks very promising, even with the low model accuracy. However, there is always more than one cluster per frame. If we then plot the different probabilities gathered from Equation 6.1 for our example case, as shown in Figure 6.4, we see that this model would not work very well in practice. It is obvious that this example model would not be able to handle much more than two or three cluster at a time, and definitely not the average which was previously in this section showed to be 7.335. Naturally, we want the model to perform better than simply being able to handle the average case.

It is also important to remark that this number is only valid given there being no systematic errors. With such systematic errors, the model would continue doing the same mistakes over and over, and no matter how many passes it has over the aisles, some tomatoes will never be found.

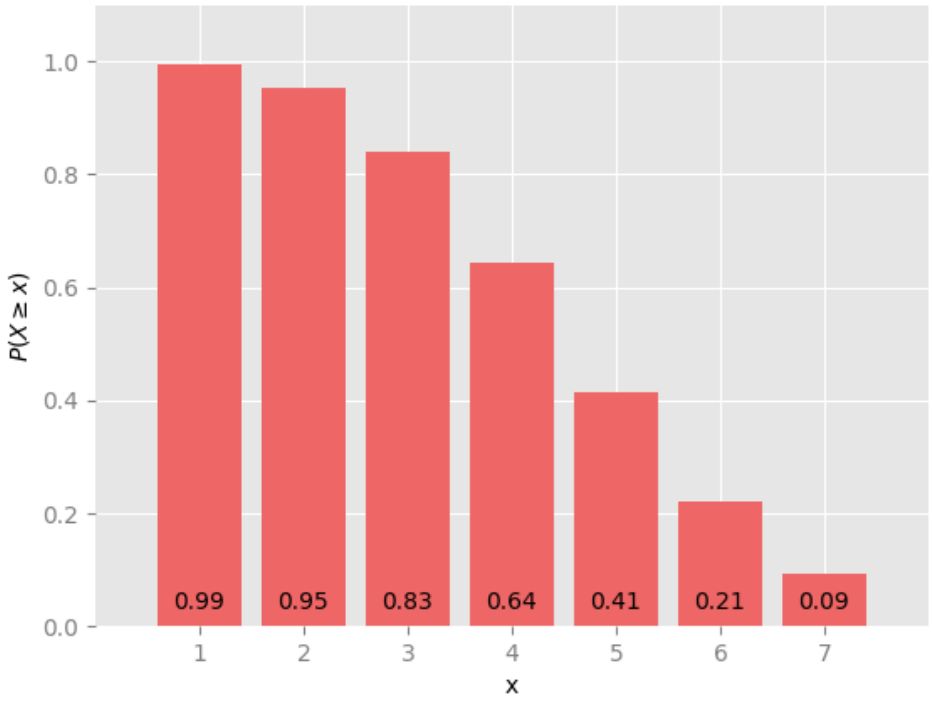


FIGURE 6.4: Probabilities for the Example Model Detecting x Tomato Clusters

As we know how many clusters of tomatoes on average are on screen, and how often new clusters appear, we can set a baseline on what the model needs to be able to handle in the minimum case. We have also presented a convenient way of naïvely estimating whether a model is accurate enough for our purpose, utilizing Equation 6.1.

6.1.2 Model Requirements

As previously mentioned in Section 3.1.1, the data for this project was gathered by the author while standing next to a worker on a moving trolley, going at a velocity of 0.91 meters per second. This is therefore a good estimation for the minimal speed an actual a harvesting robot would have to work. However, the robot would not necessarily need to work faster than the human workers. Humans need rest and nourishment, while the robots can work continually. So to make investing in such a harvesting robot, it must be able to work at a sufficient speed, enough to at least match the

human workers. From this basis, we can create some general requirements for our model.

We can empirically deduce from the source video that a tomato cluster stays in frame for an average of about 7 seconds, or 210 frames. This means that the model has to analyze the video stream and find each tomato in at least one of the 210 frames it is visible, otherwise the tomato will definitely not be harvested. With an accuracy of 90.0%, this is trivial. All this is, of course, in the ideal case.

But even with a much less accurate model – such as the Sequential Model, with its 20.4% accuracy – this can be done without much uncertainty. If we say our amount of inferences is 3 per second, and a tomato is on-screen for 7 seconds, we then have 21 tries to find the tomato. Using Equation 6.1, as shown in Equation 6.2 we get a probability of 99.2%.

$$P(X \geq 1) = 0.9916 \approx 99.2\% \tag{6.2}$$

Which is quite high. With only one pass of each aisle one would have a high chance of finding all relevant tomatoes. However, there are more clusters than just one in each frame, and our model will obviously have to do a bit more work. As mentioned in Section 6.1.1, analysis of the annotation data presents an average of 7.335 relevant tomatoes – i.e. ripe tomatoes situated in the correct aisle – per frame. So, to keep track of all relevant tomatoes, the model will have to be able to localize a total of $\sim$ 220 tomato clusters, divided over 30 frames, per second. If we then see how well (statistically) the Sequential Model would do, as shown in Equation 6.3, we get that its probability of finding all of them at least once is 11.8%, which is definitely unacceptable.

$$P(X \geq 1) = 0.1179 \approx 11.8\% \tag{6.3}$$

We are however not testing the Sequential Model, we are testing the R101 Model. With its superior accuracy, it would need less inferences per second to be reliable than the Sequential Model would. We will now attempt to estimate this minimal number of inferences per second needed.

In the name of being realistic, and to consume as little power as possible, we should set a minimal standard. Every tomato is, as discussed previously in this section, on-frame for an estimated 7 seconds. The last $0.5 - 1$ of these is where the clusters are the closest to the camera. These frames are the most

critical. By analyzing the predictions done on the dataset, we can also see that in these last $15 - 30$ frames, it is easier for the model to recognize its target, and the confidence the model presents for its detections increase noticeably. This is most likely due to the tomatoes being bigger, more detailed, et cetera. It is also in these last $15 - 30$ frames that *detection* is of utmost importance; the tomato in question is about to disappear off camera.

Thus, in a realistic scenario, all that is needed is a couple of correct predictions, for all present tomato clusters, per second. For example: if we are able to get three predictions per second, of which each prediction is on average 90.0% correct, there is no reason to believe that the model would miss any of the relevant tomato clusters, given no systematical errors.

To further improve the chances of all tomatoes being detected, one could make the robot take a second pass of the same aisle after it has completed the first one. But even with a second or third pass, a human worker would be needed to perform a control check. The proposed three predictions per second are thus a viable, and very power-conservative estimate. A higher frequency of predictions every second would of course result in a lesser chance of missing clusters, but the returns here are diminishing, and it is likely not worth it. Figure 6.5 supports this. It presents the cumulative probabilities – as defined in Equation 6.1 – of some different models, as they attempt to detect all the tomato clusters in one frame over one second. This is plotted over varying rates of inferences per second. In addition to graphing the R101 Model and the Sequential Model, we also graph two theoretical models with accuracies of 30% and 50%, for comparison. From this analysis, since we are using a model of 90% accuracy, we can confidently set the minimum viable number of inferences per second to 3.

6.1.3 Computational Requirements

As previously discussed, our model does not only find whether there is a tomato in frame, or how many of them there are. It also gives a prediction for a bounding box around each of the tomatoes, with exact co-ordinates. This can increase model size and amount of weights needed by quite a lot.

The R101 Model consists of a total of 63.733.406 weights, resulting in an allocation of 255.9MB of memory. This is not that much if one is considering running it on an industrial computer. However, if we are aiming for low

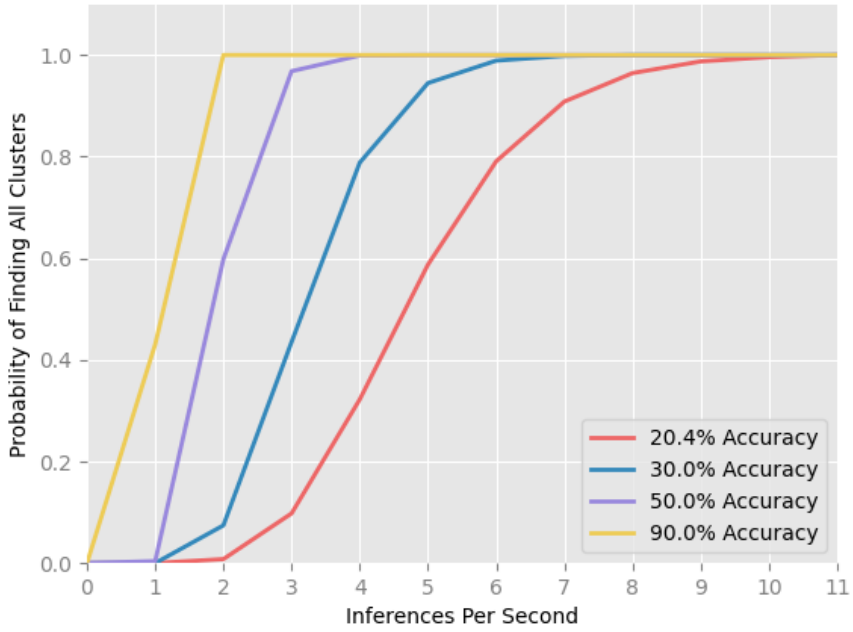


FIGURE 6.5: Probabilities for Detecting All Clusters Over One Second

power consumption, this may be far too much. To achieve both low power-consumption and a high inference rate, while still remaining accurate, we might need a model that is smaller in size. In this case, would have to resort to the methods described in Section 4.3.1.

6.1.4 Hardware

As the requirements for the model have been presented, we can now go on to find a suitable embedded system solution to run it on. Such a system would be evaluated based on the following factors:

1. Inferences per Second
2. Power Consumption
3. Size and Weight
4. Hardware Cost
5. Complexity of Development

This list is loosely sorted in an order of decreasing importance. If the system we choose cannot do the amount of inferences we need per second, it is useless to us – no matter how minuscule the power consumption. Furthermore, if a system does everything we want it to, a bit of a tedious setup is not necessarily a deal-breaker. We will adjust all values found in literature naïvely, that is to say linearly, to align more with our actual model.

An example of this could be if we know that a certain device can run a model with $1M$ weights at 20 inferences per second, and we know that our model has $2M$ weights, we linearly scale the numbers and estimate that the same device can run our model at 10 inferences per second.

As determined in Section 6.1, the amount of **inferences per second** has to at the very least be above 3 to meet our requirements. A specific limit for **power consumption** is not set, but here we can simply say that lower is always better. This, of course, also depends on the system being evaluated. An industrial computer is not expected to consume power in the same order of magnitude as an FPGA. To evaluate this metric, we look at similar cases already tested in literature, and we look at the theoretical maximal power the system can sustain over time. This is not the peak power consumption overall, but rather the maximum consumption during the running of normal applications. This is known as the Thermal Design Power (TDP) of a component. The peak power is often around 1.5 times higher than the TDP, and the actual average power consumption is often slightly lower [53]. Therefore, the TDP of a system is a conservative and viable indicator its energy needs, and we will use this in our evaluation.

For **Size and weight**, we do not set any specific limitations, but naturally they have to be within reason. An industrially sized computer could exceed 1 meter in width or breadth, which would make it bigger than a tomato aisle and thus not viable. The trolley used for the data gathering was, as mentioned in Section 3.1.1, 50`cm` wide. The computer can subsequently not be any wider than this. How much weight the trolley can support was not ascertained, but it comfortably and without issue bore two adult humans while driving. We also know that the robot would need a basin to store its gathered fruit, and thus we can surmise that this will most likely not be an issue. However, here as well, smaller is better.

Hardware cost is quite low on the list. A robotic system consists of many modules, and of these the object detection system is probably one of the cheaper ones. A tomato harvesting robot would, in addition to detecting tomato clusters, need a way to perform the actual harvesting. This would probably be done using a robotic arm of sorts. To withstand the wear and tear of constant use, and the high humidity of the greenhouses it would work in, industrial-level machinery would be preferred. In 2017, the IEEE Globalspec [54] established that this type of industrial robotic arm would cost between \$25 K-\$400 K[2]. So even with the lowest estimate, the hardware for the arm would cost more than ten to twenty times the price of an industrial computer system.

Development complexity is not necessarily a deal-breaker in any case, but given two equally good options, where one has a less complex development process than the other, the former will naturally be chosen over the latter. And systems with amazing results but with an overly complex setup might be considered less attractive than a lesser system with a simpler deployment. Many systems perform better based on how much work is put into development. Especially when working with FPGAs, one sees that the more efficiently one maps the models in hardware, the lesser the power consumption [55]. All this will be considered when evaluating which system is the most suitable for our purpose.

6.2 Hardware Survey

It may be infeasible to find the most advantageous hardware system to run our model on whilst remaining in the theoretical domain. We would therefore rather seek to rule out the suboptimal ones – i.e. the ones that fail to meet our requirements. With the factors presented in Section 6.1 in mind, we will now analyze four of these types of systems, shown in Figure 6.6:

- ◇ A Standard Industrial Computer
- ◇ Two NVIDIA Tegra-series SoCs
- ◇ A Xylinx Zynq-series MPSoC

A more concise comparison of the systems can be found in Table 6.2.

[2]Per 22.05.2020 this range was approximately between 251K NOK - 4.02M NOK

(A) Industrial Computer

(B) NVIDIA Tegra

(C) Xilinx Zynq

FIGURE 6.6: Three Potential Hardware Solutions

6.2.1 Industrial Computer

The least specialized of the three options is a standard industrial computer. This gives a lot of room to customize, but since this solution is less specialized, it follows that it is also less efficient. An industrial computer would probably be more expensive, consume more power, and be less portable that the other options, but would also perhaps be much easier to set up. We now evaluate an industrial computer, an example of which is shown in Figure 6.6a, based on the requirements set for it in Section 6.1.

- ◇ **Inferences per Second**
  Our experiments for the industrial computer were all run on a 2012 MacBook Pro, with its built-in GPU – the Intel HD Graphics 4000. This resulted in an inference time of 21.847 seconds. The exact timing of the inference program was found using the Python-package cProfile [56]. This rate is approximately 66 times less than our requirement, and is thus much too slow for our needs. However, this GPU is also very old. Testing on a more modern GPU is far more relevant for this project. The aforementioned MacBook Pro was the only machine available to the author at the time of testing[3], so we must therefore look to the related literature, and use numbers from model similar to ours.

  In the original Mask R-CNN paper [16], R. Girshick et al. used a NVIDIA Tesla M40, a GPU from 2015, to run a model almost identical to the R101 Model presented in this thesis. This GPU should theoretically be able to run at a far better rate than the previously mentioned Intel HD Graphics 4000. The experiment of Girshick et al. resulted in an inference time of 0.195s, or about 5 inferences per second, which meets our requirements in this category. This GPU is

[3]This was due to restrictions related to the COVID-19 pandemic of 2020

also quite old, being released to the public about 5 years ago. Even more modern GPUs could potentially achieve better results.

◇ **Power Consumption**
The highest amount of power the Tesla M40 can sustain over time – its TDP – is 250W and the suggested power-supply is 600W [57]. This is incredibly high in comparison to what can be achieved using a more specialized system. And this is only for the GPU. An industrial computer would have to consist of many other components – motherboard, CPU, et cetera – that also require a steady supply of power. To get an estimate of the actual power consumption of an industrial computer, we can add up the Thermal Design Powers of all the components needed. This mainly includes the GPU and the CPU, but also the extra components needed, such as the disk, memory, motherboard, et cetera.

The TDP of the GPU is as mentioned 250W. An arbitrary high-end CPU from Intel, the i9-10980XE, has a TDP of 125W [58]. In their paper, A. Mahesri et al. show that the system power consumption can vary greatly depending on the workload, and ended up being between 8W and 30W [59]. If we take the greater of these numbers, we end up with a total estimated power consumption of 405W.

◇ **Size and Weight**
The GPU itself is not very large, but an industrial computer is built up of more components than just a GPU, and it needs a casing to hold all these components. A general such casing – on the smaller side – found on the internet had the following dimensions: $48\text{cm} \times 30\text{cm} \times 4.4\text{cm}$ [60]. The largest of these is almost the exact width of our trolley. This thus meets our minimum requirements.

◇ **Hardware Cost**
It is hard to determine the exact cost of an industrial computer, as many different parts are needed to have a functioning system. Since the original Mask R-CNN paper, NVIDIA has retired the Tesla brand, and finding a price for it is difficult. Without looking at each specific part we can still bring an estimate as to how expensive an industrial computer is. In an article presenting the best computers available on the commercial market, TechRadar gives a list of computers with a

wide array of prices, ranging from 10.000 NOK to 100.000 NOK [61]. We would probably not require the higher end of this spectrum for our project, so we estimate a price range from 15.000 NOK to 30.000 NOK.

◇ **Complexity of Development**
The development complexity for an industrial computer is very low, and has already been done by the author when training and evaluating models for this thesis. It consists of setting up a simple operating system, and then installing Python and Mask R-CNN, and the required additional packages.

6.2.2 NVIDIA Tegra

The NVIDIA Tegra series is a line of SoC Integrated Circuits (ICs), manly aimed at mobile devices. An example IC from the Tegra line is shown in Figure 6.6b. The most recent of the Tegras is the Xavier [62]. Another embedded board under the Tegra name is the Jetson TX2. We now evaluate these two Tegras, again based on the requirements set in Section 6.1.

◇ **Inferences per Second**
For the TX2, there has already been a successful deployment of Mask R-CNN, done by gustavz on GitHub [63]. In this deployment, for an input image of size 512×512 pixels, they present 3000 proposals a second. A proposal in this sense is one of the many bounding boxes the model suggests, with different certainties. The ones with the highest certainty become the predictions. The Mask R-CNN configurations for the model puts the maximum amount of proposals per image at 200. If we linearly scale this number to the image resolution used in the project, we could be looking at $1 - 2$ inferences per second.

Another similar deployment based on the one *gustavz* did was done by GitHub user *naisy*, who meticulously tested on both the TX2 and the Xavier [64]. This was done with a live video feed at 1280×720 pixels. For the TX2, they managed, at best, to get 38 detections per second, which linearly adjusted to match our case is 2.30 inferences. This is not quite enough to meet our requirement of 3 per second. With the Xavier being a newer model, it naturally was able to perform better. At the same resolution as before (1280×720), it reached 58 detections per second. Linearly adjusted for our model, this ends up being an

estimated 25.7 detections per second, or around 3.5 inferences per second. In our dataset, the average number of detections needed are 7.335, as shown before in Section 6.1.1. In the worst case, this number was 11 clusters per frame. Thus, for our purpose, with the minimal requirement for inferences per second being 3, we would need around 22 detections a second, or as many as 33 in the worst case.

◇ **Power Consumption**
In the experiments of *naisy*, during detection, the TX2 was consuming 17.7W. The Xavier ran with a higher power consumption of 31.6W. These experiments are a good indicator that an SoC from the NVIDIA Tegra family would be a very viable platform for our model to run on. If we instead look at the TDPs of the two, we see that the TX2 has its at 15W, whereas the Xavier has its TDP at 30W.

◇ **Size and Weight**
The TX2 and the Xavier are both around the same size, with the TX2 being 8.7cm $\times$ 5.0cm, and the Xavier being 10.0cm $\times$ 8.7cm [65]. The weights of the two SoCs are negligible.

◇ **Hardware Cost**
Based on the information given by NVIDIA on their websites, we find that the TX2 costs around $399, or around 4.000 NOK [66], and that the Xavier costs around $699, or around 7.000 NOK [67].

◇ **Complexity of Development**
Development on an SoC would not necessarily be any more complicated than that of an industrial computer. Still, our model would have to be more optimized if we were to run it on one of the Tegras. With the industrial computer, we can get a very high inference rate with little to no optimization. This does however come at the cost of much higher power consumption. So, getting the same performance on an SoC as on an industrial computer would require additional development regarding optimization. Even with extra work, this might still be very worthwhile, since this would also reduce the power consumption by at least one order of magnitude.

6.2.3 Xilinx Zynq

Developing a deep neural network on an FPGA can be more difficult and time-consuming than the other options presented [25]. It can however also save quite a lot of power, and thus shave off large amounts of latency. Xilinx, one of the leading manufacturers of FPGAs, has a line of Multi-Processor System-on-Chip (MPSoCs) they call the *Zynqs*. The Zynqs *combine the software programmability of a processor with the hardware programmability of an FPGA*, and they promise to *provide unrivaled levels of system performance, flexibility, and scalability* [68]. A picture of one of the Zynqs, the ZCU104, can be seen in Figure 6.6c. We now evaluate the Zynqs, specifically the ZCU104, based on our requirements set in Section 6.1.

- ◇ **Inferences per Second**
  In their 2019 paper, Sharma et al. experimented with implementing different convolutional neural networks on a Zynq-based FPGA called the ZCU104 [26]. One of these Convolutional Neural Network (CNN) implementations was Faster R-CNN, which, as previously mentioned in Chapter 4, is the predecessor of Mask R-CNN. They showed that Faster R-CNN was the most accurate and the fastest among all the architectures they tested. With right above 10 million parameters, they managed to get an inference time of 58ms. Naïvely adjusted for our project, being based on the same meta-architecture and having six times more parameters, we could imagine that our specific model would run with an inference time of 348ms. This would mean just under 3 images per second, which is not within our requirements. However, increased efficiency when mapping the model to the FPGA could decrease this number.

- ◇ **Power Consumption**
  The paper from Sharma et al. [26] does not mention power consumption. Inspecting the data-sheets given by Xilinx presents no word about the TDP of the ZCU104. Therefore, we must look for other examples from relevant literature.

  In the previously mentioned FINN-paper, Y. Umuroglu et al. present a Zynq (ZC706) using less than 25W total system power during intensive image classification [29]. This power consumption was measured when classifying 12.3 million images per second. For our purpose we would

probably require lesser performance than this, as we are only analyzing a couple of images a second. However, these images are much larger than those of the FINN-paper, and we are not classifying them, we are detecting and localizing objects in them. Still, we can estimate that our power consumption will be around or below 25W.

Using another Zynq-based platform, in the presentation of the TULIPP STHEM tool-chain, T. Kalb et al. get around a $3\mathtt{W} - 4\mathtt{W}$ consumption [69]. In general, there is sufficient support to the notion that FPGAs out-perform GPUs in terms of power consumption, and that the FPGAs perform increasingly better as the complexity of the task grows [70]. We can conclude from this that the Zynqs have a power consumption in the same order of magnitude as the Tegras, but with potential to out-perform them in complex tasks.

◇ **Size and Weight**
The Zynqs are quite small, at an approximate size of $15\mathtt{cm} \times 18\mathtt{cm}$ [71]. This is a bit larger than the Tegras, but will likely not be an issue at all. The exact weight of a Zynq ZCU104 is not given by Xilinx, but based on the size of the FPGA, we can assume that its weight is negligible.

◇ **Hardware Cost**
From Xilinx own websites, one can find that the Zynq MPSoC the evaluations of this thesis are based on, the ZCU104, costs around $1.295, or around 13.000 NOK [72].

◇ **Complexity of Development**
The results from related literature seem promising. However, mapping a neural network model such as ours to an FPGA is not an easy task [25]. We must also note that the power consumption of an FPGA during object detection is directly proportional to how well the model has been mapped: M. Koraei et al. show that power consumption increases with the number of elements that are actively used [55]. So the efficiency of area usage becomes very important. Out-performing the SoCs with any significance will require an extensive process of development.

TABLE 6.2: Comparison of Three Possible Hardware Solutions

	Zynq	Jetson TX2 / Xavier	PC
Inference Time	348ms	434ms / 285ms	195ms
Power	5W – 25W	15W / 30W	∼ 405W
Cost	∼ 13K NOK	∼ 5K NOK / ∼ 7K NOK	∼ 15K NOK
Weight	Light	Light	Very Heavy
Approx. Size	15cm × 18cm	5cm / 10cm × 8.7cm	48cm × 30cm
Complexity	Very High	Low	Very Low

6.3 Hardware Comparison

A concise comparison of the four hardware systems can be found in Table 6.2. An industrial computer is physically large, and requires a lot of energy. An SoC from the Tegra-line presents a much smaller power consumption, and not much more complexity of development. Using an FPGA, like one of the Zynqs from Xilinx, would most likely result in sufficient performance at a somewhat lower power consumption than the Tegra. But the FPGAs are very hard to set up, and it might not be worth the effort. However, as discussed in Section 2.3, in more recent times, we see an increase of tool facilitating the process of mapping a machine learning model to an FPGA. So, in the future, an FPGA might be the obvious choice when considering the trade-off between inference time and power consumption. Per today, however, an SoC such as the TX2 or the Xavier is more suitable for our needs. In our case, the superior performance of the Xavier compared to the TX2 is needed to meet our requirements, and thus, we choose the Xavier as the most suitable platform to deploy our model on.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

7.1.1 Machine Learning Models

In this project, five different models were trained for 15 epochs each. Four of these were based on an implementation of Mask R-CNN by Girshick et al., and the fifth one was a sequential model with 15 layers.

Our results strongly support that transfer learning can be very lucrative in reducing training time needed. Even without the objects from the pre-trained weights resembling the actual objects we want to detect very well, transferring weights improved greatly upon the models that were without transferred weights. The model trained without any transfer learning did not go above 2.12% accuracy in any of the 15 epochs.

In an attempt to reduce the storage size needed for our final product, a model with a different backbone was trialed. The backbone was changed from ResNet101 to ResNet50. This reduced the storage space needed by 76.7MB, but also reduced the accuracy down to 16.5%. More training time could perhaps have given a more viable result.

Attempts to refine the first model trained were futile, possibly due to being too drastic. A sizeable adjustment of the non-maximum suppression threshold of the model, from 0.3 to 0.7, showed that a high threshold increases training time significantly. Attempts to reduce the amount of Type I Errors, by increasing the confidence needed for a detection to be done, were also made. The threshold was set from 0.7 to 0.85. Again, these efforts resulted in a severely reduced accuracy, perhaps for being too extreme.

Training a Sequential Model was also attempted, but due to the nature of this type of architecture, the results were poor. A sequential model such as

the one in question lacks the ability to localize objects. So unless there is, in addition to the sequential image classification module, a system that tries all the different sections of a frame, such as an R-CNN architecture, the model will not work for object detection.

Out of all the models trained and evaluated, the R101 Model, based on the standard Mask R-CNN meta-architecture, was by far the most competent. It had a maximal object detection accuracy of 90%. It requires 255.9MB to store its 63.733.406 parameters. This is in tune with Requirement M_1 and Requirement M_2.

7.1.2 Model Deployment on Hardware

Our finalized model would most likely be able to run comfortably on all the suitable embedded systems presented. Previous work has shown that similar models run satisfactory on all of them, and also that optimization schemes for mapping such a system to an FPGA or similar exist, and are in continual development.

Our aim was to maximize our systems ability to detect ripe tomato clusters on the vine – without consuming too much power, being too immobile, or taking up too much space. Specific requirements were set up based on this, and four potentially suitable hardware systems were examined: An industrial computer, two SoCs, and an FPGA. The hardware system best suited for our purpose ended up being the NVIDIA Xavier. It can provide an acceptable inference time of 285ms, while keeping a relatively low consumption, with a TDP of 30W. This meets Requirement M_3.

7.2 Future Work

Future work should begin with actually testing the finalized model on the various embedded systems presented in the thesis. It is very difficult to gauge whether or not such a system works as theorized, and the optimal or most advantageous hardware platform in theory might not be the best one in practice.

Also, further research should be done on utilizing quantization to reduce the decimal-point accuracy of the weights of the model, even going as far as using binary weights. Pruning of the network and compression could also

be studied. This could potentially increase inference times for all hardware solutions by decreasing the complexity needed.

This might also warrant a study of other hardware options than the ones presented in this thesis. Given a more extensive understanding of the workings and requirements of our model could open up the possibility for other embedded systems.

A more in-depth analysis of the actual minimal requirement of inferences per second could also give more insight in how strong the hardware really needs to be. This could be done by developing a simulator of some kind, accurately representing the greenhouse environment, and analyzing the behavior of the robot as it recognizes and harvests the relevant fruits.

For the model itself, an even more extensive empirical fine-tuning of the various hyperparameters could result in a better accuracy. It is highly unlikely that the initial parameters of Mask R-CNN are the exact ones that are optimal for our specific dataset. More training data could also potentially ameliorate the many erroneous detections of clusters of tomatoes that are not yet ripe, and clusters residing in the wrong aisle.

It would also be interesting to see a Mask R-CNN model, similar to the R101 Model presented in this thesis, be trained on only the part of the image shown selected by the heat-map in Figure 6.1. This could increase accuracy and would definitely reduce inference times. The size of the image that needs to be analyzed plays a large part in determining the total inference time.

Appendix A

On Harvesting Piccolo Tomatoes

The ripeness of a cluster of Piccolo tomatoes can simply be determined by its color. Mature tomatoes present a deep red color, whereas the tomatoes that are not yet ripened are more yellow or even green.

In most cases, however, the individual tomatoes are never all ripe at the same time. The ripen from top to bottom, and a couple of the bottom tomatoes are usually not completely ripe when the cluster is harvested. If all the tomatoes in a cluster are completely red, the top ones are most likely over-ripened. Therefore, the best candidates for picking are the clusters where the lowermost couple of tomatoes are red, or where they have a faint yellow tint.

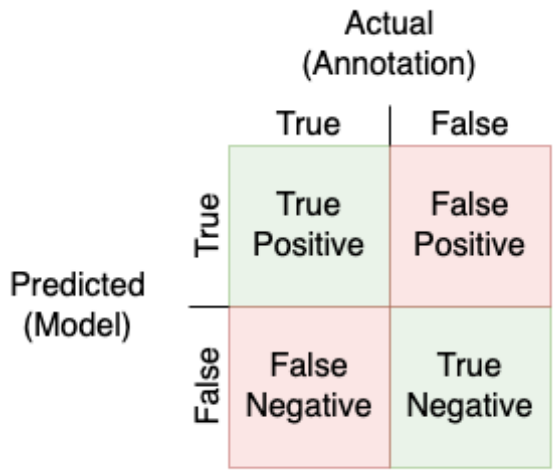
FIGURE A.1: Tomato Cluster Soon Ready for Harvest

Appendix B

The Confusion Matrix

The confusion matrix is a classic way of representing the different combinations of what the model predicts versus what is actually true.

TABLE B.1: The Confusion Matrix



TP or True Negative (TN) occur when the model and annotation are in agreement, i.e. the model has predicted correctly.

FP, which this thesis refers to as Type I Errors, occur when the model predicts an object without there actually being one there.

False Negative (FN) occur when the model fails to detect an object. These are often referred to as Type II errors [36].

Appendix C

Mask R-CNN Hyperparameters

```
BACKBONE
BACKBONE_STRIDES
BATCH_SIZE
BBOX_STD_DEV
COMPUTE_BACKBONE_SHAPE
DETECTION_MAX_INSTANCES
DETECTION_MIN_CONFIDENCE
DETECTION_NMS_THRESHOLD
GPU_COUNT
GRADIENT_CLIP_NORM
IMAGES_PER_GPU
IMAGE_CHANNEL_COUNT
IMAGE_MAX_DIM
IMAGE_META_SIZE
IMAGE_MIN_DIM
IMAGE_MIN_SCALE
IMAGE_RESIZE_MODE
IMAGE_SHAPE
LEARNING_MOMENTUM
LEARNING_RATE
MASK_POOL_SIZE
MASK_SHAPE
MAX_GT_INSTANCES
MEAN_PIXEL
MINI_MASK_SHAPE
NAME
NUM_CLASSES
POOL_SIZE
POST_NMS_ROIS_INFERENCE
POST_NMS_ROIS_TRAINING
PRE_NMS_LIMIT
ROI_POSITIVE_RATIO
RPN_ANCHOR_RATIOS
RPN_ANCHOR_SCALES
RPN_ANCHOR_STRIDE
RPN_BBOX_STD_DEV
RPN_NMS_THRESHOLD
RPN_TRAIN_ANCHORS_PER_IMAGE
STEPS_PER_EPOCH
TOP_DOWN_PYRAMID_SIZE
TRAIN_BN
TRAIN_ROIS_PER_IMAGE
USE_MINI_MASK
USE_RPN_ROIS
VALIDATION_STEPS
WEIGHT_DECAY
```

List generated from the Matterport Mask R-CNN configuration [52].

Bibliography

[1] Magnus Själander et al. "EPIC: An Energy-Efficient, High-Performance GPGPU Computing Research Infrastructure". In: (Dec. 2019). URL: https://arxiv.org/abs/1912.05848v1.

[2] Jakasania G. and Dr Yadav. "Automation: New Horizon in Agricultural Machinery". In: *Journal of Ergonomics* 08 (Jan. 2017). DOI: 10.4172/2165-7556.1000.e178.

[3] Haydn Valle. *Australian vegetable growing farms: an economic survey, 2012-13 and 2013-14*. ABARES, 2014.

[4] B. Astrand et al. *An economic feasibility assessment of autonomous field machinery in grain crop production*. Jan. 1970. URL: https://link.springer.com/article/10.1007/s11119-019-09638-w.

[5] Kadeghe Fue et al. "An Extensive Review of Mobile Agricultural Robotics for Field Operations: Focus on Cotton Harvesting". In: *AgriEngineering* 2 (Mar. 2020), pp. 150–174. DOI: 10.3390/agriengineering2010010.

[6] Kaiming He et al. *Deep Residual Learning for Image Recognition*. 2015. arXiv: 1512.03385 [cs.CV].

[7] *Common Objects in Context Dataset*. URL: http://cocodataset.org/.

[8] Lars Grimstad and Pål From. "The Thorvald II Agricultural Robotic System". In: *Robotics* 6.4 (2017), p. 24. DOI: 10.3390/robotics6040024.

[9] Inkyu Sa et al. "DeepFruits: A Fruit Detection System Using Deep Neural Networks". In: *Sensors* 16.8 (Mar. 2016), p. 1222. DOI: 10.3390/s16081222.

[10] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks".
In: *CoRR* abs/1506.01497 (2015). arXiv: 1506.01497.
URL: http://arxiv.org/abs/1506.01497.

[11] *ImageNet - Online image database*. URL: http://www.image-net.org/.

[12] Kyosuke Yamamoto et al. "On Plant Detection of Intact Tomato Fruits Using Image Analysis and Machine Learning Methods".
In: *Sensors* 14.7 (Sept. 2014), pp. 12191–12206.
DOI: 10.3390/s140712191.

[13] Libin Zhang et al.
"Recognition of greenhouse cucumber fruit using computer vision".
In: *New Zealand Journal of Agricultural Research* 50.5 (2007), pp. 1293–1298. DOI: 10.1080/00288230709510415.

[14] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2013. arXiv: 1311.2524 [cs.CV].

[15] Ross Girshick. "Fast R-CNN".
In: *2015 IEEE International Conference on Computer Vision (ICCV)* (2015).
DOI: 10.1109/iccv.2015.169.

[16] Kaiming He et al. "Mask R-CNN".
In: *2017 IEEE International Conference on Computer Vision (ICCV)* (2017).
DOI: 10.1109/iccv.2017.322.

[17] Stylianos I. Venieris, Alexandros Kouris, and Christos-Savvas Bouganis. *Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions*. 2018.
arXiv: 1803.05900 [cs.CV].

[18] Ahmad Shawahna, Sadiq M. Sait, and Aiman El-Maleh.
"FPGA-Based Accelerators of Deep Learning Networks for Learning and Classification: A Review". In: *IEEE Access* 7 (2019), pp. 7823–7859.
ISSN: 2169-3536. DOI: 10.1109/access.2018.2890150.
URL: http://dx.doi.org/10.1109/ACCESS.2018.2890150.

[19] Tobias Kalb et al. "TULIPP: Towards Ubiquitous Low-Power Image Processing Platforms".
In: *2016 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation (SAMOS)* (2016).
DOI: 10.1109/samos.2016.7818363.

[20] Ahmad Sadek et al. “Supporting Utilities for Heterogeneous Embedded Image Processing Platforms (STHEM): An Overview”. In: *Applied Reconfigurable Computing. Architectures, Tools, and Applications Lecture Notes in Computer Science* (2018), pp. 737–749. DOI: 10.1007/978-3-319-78890-6_59.

[21] A. Canziani, E. Culurciello, and A. Paszke. “Evaluation of neural network architectures for embedded systems”. In: *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*. 2017, pp. 1–4.

[22] H. Mao et al. “Towards Real-Time Object Detection on Embedded Systems”. In: *IEEE Transactions on Emerging Topics in Computing* 6.3 (2018), pp. 417–431.

[23] Jiantao Qiu et al. “Going Deeper with Embedded FPGA Platform for Convolutional Neural Network”. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA 16* (2016). DOI: 10.1145/2847263.2847265.

[24] Song Han et al. “EIE: Efficient Inference Engine on Compressed Deep Neural Network”. In: *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)* (2016). DOI: 10.1109/isca.2016.30.

[25] David Bacon, Rodric Rabbah, and Sunil Shukla. “FPGA Programming for the Masses”. In: *Queue* 11.2 (Feb. 2013), pp. 40–52. ISSN: 1542-7730. DOI: 10.1145/2436696.2443836. URL: https://doi.org/10.1145/2436696.2443836.

[26] A. Sharma, V. Singh, and A. Rani. “Implementation of CNN on Zynq based FPGA for Real-time Object Detection”. In: *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*. 2019, pp. 1–7.

[27] *PYNQ: Python productivity for Zyng*. URL: http://www.pynq.io/.

[28] Maciej Wielgosz and Michał Karwatowski. "Mapping Neural Networks to FPGA-Based IoT Devices for Ultra-Low Latency Processing". In: *Sensors* 19.13 (July 2019), p. 2981. ISSN: 1424-8220. DOI: 10.3390/s19132981. URL: http://dx.doi.org/10.3390/s19132981.

[29] Yaman Umuroglu et al. "FINN". In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17* (2017). DOI: 10.1145/3020078.3021744. URL: http://dx.doi.org/10.1145/3020078.3021744.

[30] Michaela Blott et al. *FINN-R: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks*. 2018. arXiv: 1809.04570 [cs.AR].

[31] Jørgen Boganes. "Towards an Efficient Workflow for Object Detection in Agricultural Robotics". In: (Dec. 2019).

[32] *iPhone XS – Technical Specifications*. URL: https://support.apple.com/kb/SP779.

[33] OpenCV. *Computer Vision Annotation Tool*. Nov. 2019. URL: https://github.com/opencv/cvat.

[34] *Nvidia Tesla V100*. URL: https://www.nvidia.com/en-us/data-center/v100/.

[35] *NTNU Department of Computer Science*. URL: https://www.ntnu.edu/idi.

[36] Derek Hoiem, Yodsawalai Chodpathumwan, and Qieyun Dai. "Diagnosing Error in Object Detectors". In: *Computer Vision – ECCV 2012*. Ed. by Andrew Fitzgibbon et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 340–353. ISBN: 978-3-642-33712-3.

[37] *FAIR Facebook AI Research*. URL: https://ai.facebook.com/.

[38] Jonathan Huang et al. "Speed/Accuracy Trade-Offs for Modern Convolutional Object Detectors". In: *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)* (2017). DOI: 10.1109/cvpr.2017.351.

[39] Waleed Abdulla. *Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow*. https://github.com/matterport/Mask_RCNN. 2017.

[40] *Matterport*. URL: https://matterport.com/.

[41] Jason Yosinski et al. "How Transferable are Features in Deep Neural Networks?" In: *arXiv.org* (Nov. 2014). URL: https://arxiv.org/abs/1411.1792.

[42] Nobuaki Kimura et al. "Convolutional Neural Network Coupled with a Transfer-Learning Approach for Time-Series Flood Predictions". In: *Water* 12 (Dec. 2019), p. 96. DOI: 10.3390/w12010096.

[43] Nitish Shirish Keskar et al. *On Large-Batch Training for Deep Learning: Generalization Gap and Sharp Minima*. 2016. arXiv: 1609.04836 [cs.LG].

[44] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *Lecture Notes in Computer Science* (2016), pp. 21–37. ISSN: 1611-3349. DOI: 10.1007/978-3-319-46448-0_2. URL: http://dx.doi.org/10.1007/978-3-319-46448-0_2.

[45] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2015. arXiv: 1510.00149 [cs.CV].

[46] Forrest N. Iandola et al. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size*. 2016. arXiv: 1602.07360 [cs.CV].

[47] *Building powerful image classification models using very little data*. URL: https://blog.keras.io/building-powerful-image-classification-models-using-very-little-data.html.

[48] *ResNet50*. URL: https://resources.wolframcloud.com/NeuralNetRepository/resources/ResNet-50-Trained-on-ImageNet-Competition-Data.

[49] *ResNet101*. URL: https://resources.wolframcloud.com/NeuralNetRepository/resources/ResNet-101-Trained-on-ImageNet-Competition-Data.

[50] *ResNet151*.
URL: https://resources.wolframcloud.com/NeuralNetRepository/resources/ResNet-152-Trained-on-ImageNet-Competition-Data.

[51] Ata Jodeiri et al. *Region-based Convolution Neural Network Approach for Accurate Segmentation of Pelvic Radiograph*. 2019.
arXiv: 1910.13231 [cs.CV].

[52] *Mask R-CNN Base Configurations*. URL: https://github.com/matterport/Mask_RCNN/blob/master/mrcnn/config.py.

[53] David A. Patterson and John L. Hennessy.
Computer Architecture: A Quantitative Approach.
San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1990.

[54] *What Is the Real Cost of an Industrial Robot Arm?*
URL: https://insights.globalspec.com/article/4788/what-is-the-real-cost-of-an-industrial-robot-arm.

[55] Mostafa Koraei, Omid Fatemi, and Magnus Jahre. "DCMI: A Scalable Strategy for Accelerating Iterative Stencil Loops on FPGAs".
In: *ACM Trans. Archit. Code Optim.* 16.4 (Oct. 2019). ISSN: 1544-3566.
DOI: 10.1145/3352813. URL: https://doi.org/10.1145/3352813.

[56] *The Python Profilers*.
URL: https://docs.python.org/3/library/profile.html.

[57] *Nvidia Tesla M40*.
URL: https://www.techpowerup.com/gpu-specs/tesla-m40.c2771.

[58] *Intel Core i9-10980XE*. URL: https://www.intel.com/content/www/us/en/products/processors/core/x-series/i9-10980xe.html.

[59] Aqeel Mahesri and Vibhore Vardhan.
"Power Consumption Breakdown on a Modern Laptop".
In: *Power-Aware Computer Systems*.
Ed. by Babak Falsafi and T. N. VijayKumar.
Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 165–180.
ISBN: 978-3-540-31485-1.

[60] *Advantech IPC-120*. URL: https://www.advantech.eu/products/1-2jkcty/ipc-120/mod_efc33129-4c04-403d-8eb3-aeb485664586.

[61] *Best Computers of 2020*.
URL: https://www.techradar.com/news/computing/pc/10-of-the-best-desktop-pcs-of-2015-1304391.

[62] *NVIDIA Tegra Xavier*. URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.

[63] *Mask R-CNN on NVIDIA Jetson TX2*.
URL: https://github.com/gustavz/Mobile_Mask_RCNN.

[64] *Real-time Object Detection on Jetson Xavier/TX2/TX1, PC*.
URL: https://github.com/naisy/realtime_object_detection.

[65] *NVIDIA Jetson Hardware Page*.
URL: https://developer.nvidia.com/embedded/develop/hardware.

[66] URL: https://www.nvidia.com/nb-no/autonomous-machines/jetson-store/.

[67] *Jetson AGX Xavier*. URL: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-agx-xavier/.

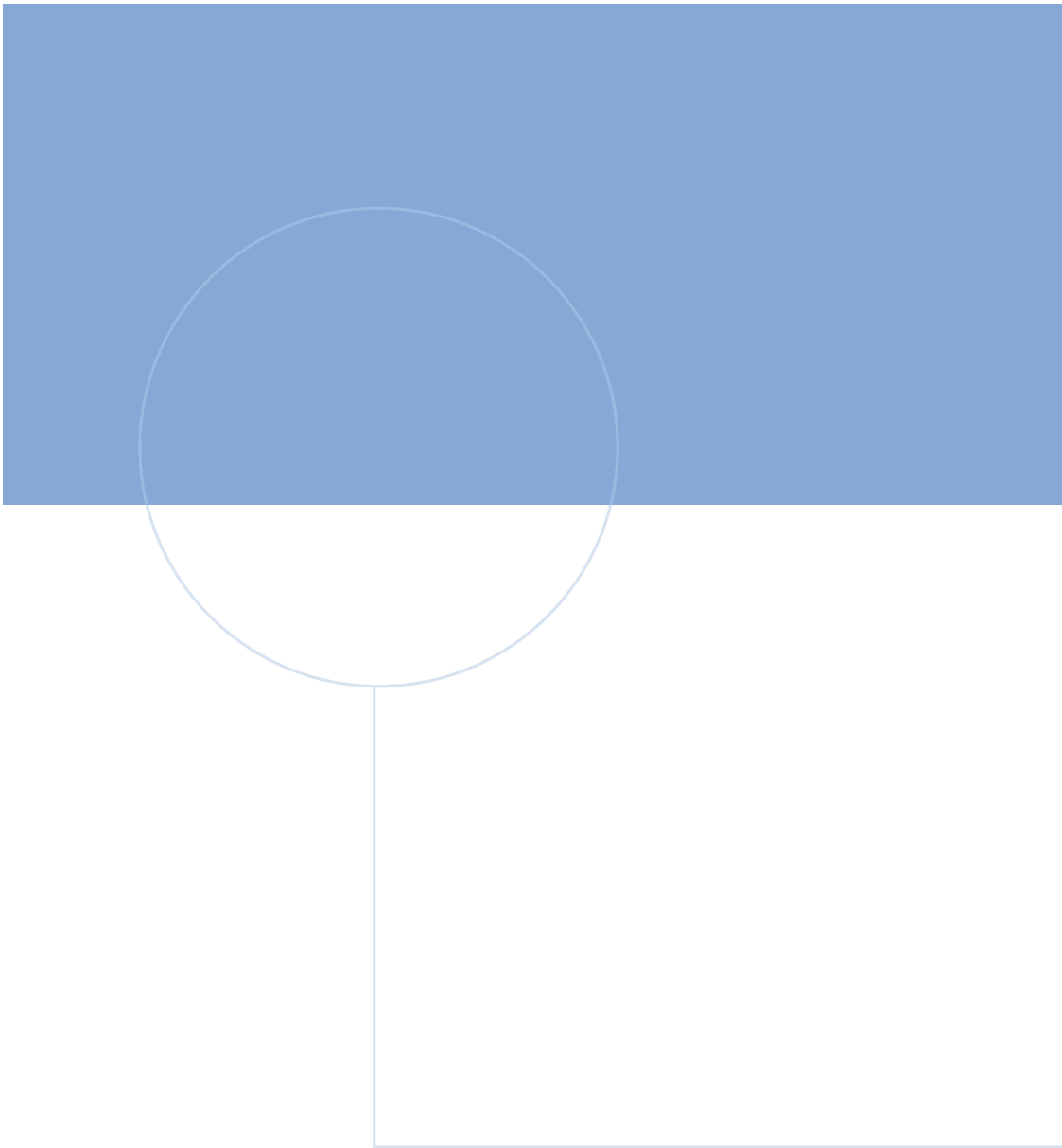
[68] *Xilinx SoC Portfolio*. URL:
https://www.xilinx.com/products/silicon-devices/soc.html.

[69] *TULIPP Grant Agreement*. URL:
http://tulipp.eu/wp-content/uploads/2019/01/d44-final.pdf.

[70] Murad Qasaimeh et al. *Comparing Energy Efficiency of CPU, GPU and FPGA Implementations for Vision Kernels*. 2019.
arXiv: 1906.11879 [cs.CV].

[71] *ZCU104 Evaluation BoardUser Guide*.
URL: https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf.

[72] *Zynq UltraScale MPSoC ZCU104 Evaluation Kit*. URL:
https://www.xilinx.com/products/boards-and-kits/zcu104.html.



NTNU

Norwegian University of
Science and Technology