Självständigt arbete på avancerad nivå

Independent degree project – second cycle

Elektroteknik
Electronics Engineering

Machine visual feedback through CNN detectors
Mobile object detection for industrial application

Kastriot Rexhaj

Mittuniversitetet
MID SWEDEN UNIVERSITY

Campus Härnösand Universitetsbacken 1, SE-871 88. **Campus Sundsvall** Holmgatan 10, SE-851 70 Sundsvall.
Campus Östersund Kunskapens väg 8, SE-831 25 Östersund.
Phone: +46 (0)771 97 50 00, Fax: +46 (0)771 97 50 01.

MID SWEDEN UNIVERSITY
Electronics design division

Examiner: Mattias O'Nils, Mattis.ONils@miun.se
Supervisor: Benny Thörnberg, Benny.Thornberg@miun.se
Author: Kastriot Rexhaj, kare1301@student.miun.se
Degree programme: Master of Science in Engineering, Electronics Engineering, 300 credits
Main field of study: Electronics Engineering
Semester, year: Spring (VT), 2019

Sammanfattning

Den här rapporten behandlar objekt detektering som en möjlig lösning på Valmets efterfrågan av ett visuellt återkopplingssystem som kan hjälpa operatörer och annan personal att lättare interagera med maskiner och utrustning. Nya framsteg inom djupinlärning har dem senaste åren möjliggjort framtagande av neurala nätverksarkitekturer med detekteringsförmågor. Då industrisektorn svårare tar till sig högst specialiserade algoritmer och komplexa bildbehandlingsmetoder (som tidigare varit fallet med objekt detektering) så ger djupinlärningsmetoder istället upphov till att skapa självlärande system som är återanpassningsbara och närmast intuitiva i dem fall där sådan teknologi åberopas. Den här studien har därför valt att studera ett par sådana teknologier för att hitta möjliga implementeringar som kan realiseras på något så enkelt som en mobiltelefon. Urvalet har därför bestått i att hitta detekteringsmodeller som är hårdvarumässigt resurssnåla och implementera ett sådant system för att agera prototyp och underlag till Valmets vidare diskussioner kring objekt-detekteringsslösningar. Studien valde att implementera en SSD-MobileNetsV2 modellarkitektur då den uppvisade lovande egenskaper kring hårdvarukraven. Modellen implementerades och utvärderades på Valmets mest förekommande telefon *Samsung Galaxy S8* och resultatet visade på en god förmåga för modellen att detektera objekt. Den valda modellen gav 60 % precision på utvärderingsbilderna och lyckades nå 4.7 FPS på den implementerade telefonen. TensorFlow användes för programmering och som stödjande mjukvaruverktyg för träning, utvärdering samt vidare implementering. Studien påpekar optimistiska förväntningar av denna typ av teknologi; kombinerat med bättre smarttelefoner i framtiden kan det leda till revolutionerande lösningar för både industri och konsumenter.

Nyckelord: Object detection, CNN, SSD, Faster R-CNN, YOLO, TensorFlow, ARM, Android, Deep learning, Machine vision.

Abstract

This paper concerns itself with object detection as a possible solution to Valmet's quest for a visual-feedback system that can help operators and other personnel to more easily interact with their machines and equipment. New advancements in deep learning, specifically CNN models, have been exploring neural networks with detection-capabilities. Object detection has historically been mostly inaccessible to the industry due the complex solutions involving various tricky image processing algorithms. In that regard, deep learning offers a more easily accessible way to create scalable object detection solutions. This study has therefore chosen to review recent literature detailing detection models with a selective focus on factors making them realizable on ARM hardware and in turn mobile devices like phones. An attempt was made to single out the most lightweight and hardware efficient model and implement it as a prototype in order to help Valmet in their decision process around future object detection products. The survey led to the choice of a SSD-MobileNetsV2 detection architecture due to promising characteristics making it suitable for performance-constrained smartphones. This CNN model was implemented on Valmet's phone of choice, *Samsung Galaxy S8*, and it successfully achieved object detection functionality. Evaluation shows a mean average precision of 60 % in detecting objects and a 4.7 FPS performance on the chosen phone model. TensorFlow was used for developing, training and evaluating the model. The report concludes with recommending Valmet to pursue solutions built on-top of these kinds of models and further wishes to express an optimistic outlook on this type of technology for the future. Realizing performance of this magnitude on a mid-tier phone using deep learning (which historically is very computationally intensive) sets us up for great strides with this type of technology in the future; and along with better smartphones, great benefits are expected to both industry and consumers.

Keywords: Object detection, CNN, SSD, Faster R-CNN, YOLO, TensorFlow, ARM, Android, Deep learning, Machine vision.

Acknowledgments

(Sv. **Förord**.)

Tack till min familj som stöttat mig under arbetets gång.

Tack till min handledare Benny Thörnberg för råd och tips.

Tack till Valmet för att ha låtit mig utföra examensarbete hos dom.

Table of contents

Sammanfattning **1**
Abstract **2**
Acknowledgments **3**
Table of contents **4**
Terminology **6**
1 Introduction **7**
1.1 Background and problem motivation 8
1.2 High-level problem statement 10
1.3 Scope 12
1.4 Concrete and verifiable goals 13
1.5 Contributions 14
2 Theory **15**
2.1 Neural networks 15
2.2 Convolutional neural networks 22
2.3 Object detection 26
3 Model review **27**
3.1 R-CNN 27
3.2 Fast R-CNN 30
3.3 Faster R-CNN 31
3.4 SSD 32
3.5 YOLO 34
3.6 Base nets 35
3.7 Comparison 36
4 Model realization **38**
4.1 Software frameworks 38
4.2 Creating training data 39
4.3 Android-powered device 39
5 Method **41**
5.1 Hardware and preparation 41
5.2 Model specification 41
5.3 Evaluation criteria's 43
6 Implementation **45**
6.1 Prerequisites 45
6.2 Software installation 45

6.3 Tensor 46
6.4 Functions 47
6.5 Model configuration 48
6.6 Labelling regions 50
6.7 Fine-tuning checkpoint 53
6.8 Model optimization for ARM devices 53
6.9 Implementation on Samsung Galaxy S8 54
7 Results 57
7.1 Model performance 57
7.2 Android deployment 63
8 Discussion 66
8.1 Social aspects 69
8.2 Ethical aspects 70
9 Conclusions 71
9.1 Future work 73
References 74
Appendix A: Pipeline configuration 79
Appendix B: Image evaluation 83

Terminology

Abbreviations

NN	Neural Network
CNN	Convolutional Neural Network
SSD	Single Shot Detection
mAP	Mean Average Precision
IoU	Intersection-over-Union
YOLO	You Only Look Once
VoTT	Visual Object Tagging Tool
GPU	Graphical Processing Unit
CPU	Central Processing Unit
MAdd	Multiply-accumulate (operation)
RAM	Random Access Memory
ARM	Advanced RISC Machine
SVM	Support Vector Machines
FC	Fully-connected (layer)
ReLU	Rectified Linear Unit
CONV	Convolutional (operation)
POOL	Pooling (operation)

1 Introduction

Today what's known as deep learning (a subset of the more encompassing term machine learning) is set to revolutionize major areas directly affecting human life and the technology for which society has come to rely on. This becomes apparent by using any search engine and weekly finding hundreds of new articles detailing deep learning and its advancements in various fields.

An analysis from Google Trends tools on the term *deep learning* shows mild and barely noticeable search activity until about the end of the year 2014 where it saw an exponential increase in popularity and is currently enjoying an all-time high in 2019.[1]

Deep learning consists of algorithms operating under a base of mathematics, computer science and neuroscience. A deep learning algorithm trains a network by being fed data and having the network shape itself accordingly in order to recognize said data and its different patterns. It is hard to set limits to what can be achieved by using deep learning tools. One of the more highlighted achievements of said technology is that of self-driving cars. A problem so complex that pure logic programming would not suffice in reliably guaranteeing efficacy. It is precisely these types of problems where deep learning shines. Any problem where human vision and its inherent object detection capabilities is needed could theoretically be solved by training a neural network with deep learning algorithms. It is therefore precisely image classification and object detection that is most responsible for advancing the deep learning research as of today.[2]

This report steers an effort towards bringing deep learning methodologies into the domain of an industry related problem and to assess its usefulness in solving a company-specific problem of object detecting qualities. Valmet is a Finnish company specializing in providing technologies related to pulp, paper and energy. They have expressed an interest in surveying deep learning tools in an effort to obtain the groundwork needed to develop future Augmented Reality-enabled solutions to their clients

for machine inspection and identification. They hope to finalize technologies that can help technicians and other staff on-site to easily identify and troubleshoot a machine without prior experience. Deep learning is a valuable tool in realizing such an effort and its usefulness to the presented problem will be surveyed and detailed through further chapters in this report.

1.1 Background and problem motivation

The work contained herein is a result of a 20 week long independent study carried out as part of a master's thesis in Electronics Engineering at Mid Sweden University. The report aims to answer the open-ended problem laid out by Valmet in regards to creating a system to help them build a future machine inspection tool. Valmet has an office in Sundsvall where this thesis idea first originated. Since early 2018 they have been exploring various technologies to expand their current business model and offer customized tools to help operators and customers more easily interact with their equipment and machines. Technologies like virtual reality, augmented reality, industrial ethernet and more recently machine learning has been cornerstones in the discussions taking place. The university has touched upon both machine vision and machine learning in various forms - both as research and coursework. Therefore, it naturally seemed fit to form a thesis project which could further a valid research contribution in this field while also helping the local industry in solving a relevant problem.

The task set forth was to thoroughly explore the possibilities of creating a system capable of recognizing a machine's different parts and utilities in real time and broadcasting information about any detected objects to the observing user. In addition to simply exploring viable strategies to implement this, the company was also interested in having a proof-of-concept prototype be developed that showcases the described functionality. The goal is to gather enough information as to assess the viability of such a solution given current available technology. Furthermore, to avoid necessitating cumbersome setups and additional equipment, interest has been expressed in documenting if said solution could be mobile and preferably accessed through a standard android-powered smartphone (for which

the company issues their employees). This would have the added benefit of being immediately available to any of their employees who stand to benefit from this kind of solution.

The third point which came to influence the direction of this work; is the requirement that any such solution surveyed and prototyped should be able to be re-designed with little effort to suit different types of machines and equipment. Thus, making sure that the final system is built in such a fashion that the development itself readily lends itself to be streamlined into concrete steps not requiring advanced knowledge, effort and competences for any smaller tweaks and changes that might arise from future needs. It is this point that prompted an overwhelmingly favorable focus towards evaluating the use of deep learning methodologies in solving the stated problem as opposed to simpler machine learning algorithms. For one, deep learning often handles complex multi-class problems more efficiently as is object detection. Secondly, deep learning methodologies cuts out the level of expertise typically required to break down problems into "features" that can be understood by the computer. Instead, deep learning, given enough data, incrementally learns itself using convolutional neural networks. This kind of mechanism is more viable in creating an end-to-end type of workflow that facilitates creating a streamlined process for which expertise knowledge is not critical for maintenance and development purposes.

Object detection is nothing new and has been attempted in various forms even before the advent of machine learning. In those cases it often seemed to be crude object tracking that failed to discriminate objects outside its specific intended function. An example would be setting up a camera at the roadside, pointing it at the freeway in order to track the number of passing cars. Given the right image processing operations it is possible to segment "moving objects" out of the more static background and apply further operations to ensure the captured object is "big enough" to be deemed a "car" for which the program can subsequently keep track of.[3]

The preceding example uses human ingenuity and available image processing tools to single out a desired object given known variables (size,

speed, color and shape). This works exceptionally well in a limited problem and might even out-perform machine learning in some cases; hardware-constraints being one of them. This last reason is why often a mix of these two methodologies are employed to strike a balance between performance and hardware cost. It does however become increasingly difficult to rely solely on image processing operations for object detection given more complex problems and there is a point where manually engineering algorithms for image detection start to become tedious and impractical. This is why machine learning have renewed interest for object detection and made it viable in circumstances which might have been more difficult in the past. Machine learning applications using object detection are found in a diverse range of fields including areas such as video surveillance, medical analysis finding tumors, detecting forest fires, unmanned aerial vehicle detection, pedestrian detection and also the widely used optical character recognition that the reader might have encountered in any of the scan-text-to-digital applications found commercially.[4-8]

Interest for object detection can also be seen seeping into traditional industry, most notably manufacturing[9][10]. The main area of interest seems to be quality management, sorting and finding visual cues to aid an assembly line in making the correct decisions. There seems to be a relative lack of work done exploring object detection at the operator level, that would aid human-to-machine interaction. The type of machine detail detection proposed in this report - that could hypothetically improve troubleshooting and installation of a machine by providing the concerned user with viable information about the different parts and their respective function in real-time. Therefore, it seems both valid and beneficial to be carrying out research in this direction and hopefully contribute to finding new use cases for both object detection and the various methodologies it relies on.

1.2 High-level problem statement

The overarching objective with this study is to thoroughly identify and assess the available machine learning methodologies and otherwise; along with their respective available technologies and tools in order to

form an opinion as to the viability of a technical solution as follows: A live-feed video-capturing system that is capable of running on ARM architecture and autonomously detect various parts and details of a machine in order to provide relevant information to the user. It is proposed that such a system might greatly benefit the recipient of said technology in providing field technicians or otherwise with an inspection tool meant to replace extensive and time-consuming instructions and manual reading in order to identify relevant machine details. The system should be easy and accessible to end users but also allow for an intuitive development scheme that facilitates easy changes to the software to meet future needs.

To fully answer and satisfy the formulated problem, it would have to be attempted in two ways:

1. A documentation over relevant technologies that need to be used in order to create a system as described. In the case of multiple competing technologies, a survey should be done leading up to key differences between them and their benefits to the proposed application.
2. A prototype using some or any of the technologies which shows the system performing basic functionality.

The final write-up of this project, including both the theory work and implementation, should if successful, help the interested parties in knowing how and in which direction to proceed with a project of this type and scope. The result should at the very least provide insights and deeper knowledge into the feasibility of the described solution and the challenges which need to be overcome to successfully implement it. All of these efforts and their intended function is reflected on throughout the disposition of this report and further subchapters will delve into the specifics of what could be expected from this kind of study in terms of real world applications of the formulated problem statement.

1.3 Scope

The very fact that this report covers an independent study which is limited to a 20 week period necessitates clear and well-defined boundaries through which the work has to maintain itself in order to meet relevant deadlines and the finalization of its results. To be able to meet these goals within the relevant time frame, the following limitations has been worked out for this project:

1. For surveying technologies including but not limited to machine learning and deep learning algorithms and their accompanying neural networks; the comparing of these should be done exclusively by analyzing scientific papers detailing their proven effectiveness and benefits. No own tests will be done with different setups and algorithms in order to compare them due to the time-consuming process of training neural networks.
2. Only one model and algorithm will be implemented in the prototype stage. This model and its corresponding algorithms will be chosen based on the comparison study depending on the factors shown which make it the most viable for object detection purposes and implementation in ARM architecture.
3. The mobile application prototype should only include basic functionality. It should encircle detected machine details and provide visual cues as to their location in the image along with a descriptive name. The application should not accept inputs from the user and have other complex functionality because this is deemed to be simple but time-consuming programming which fails to contribute to the focused problem statement as much as it would risk delaying potential finalization of the report.
4. The mobile application should only focus on one machine and detect only a few (four to eight) machine details deemed most interesting by an engineer employed at Valmet. Proving the system works for one machine will sufficiently provide evidence of the application being feasible for other machines and equipment and is therefore considered time-efficient.
5. An emphasis will be set to exploring models or combination of models in which the feature extraction is fully covered by both the

learning algorithm and the deployment model (more in theory chapter). This is done in order to help meet the company's requirement of creating an end-to-end system. Techniques which require precise image hacking and lacks a generalized way of working with data will therefore be ignored.

The study also recognizes there might be other ways to approach object detection other than solely utilizing deep learning models but has chosen to narrow its focus down to this area due to its popularity in being used for this exact reason in the scientific community. Commercial software and programs will also be avoided and only open-source frameworks will be utilized. This is done in order to gain insight into the inner workings of any model chosen and to effectively evaluate the parameters that may or may not lead to a successful implementation. This insight is not possible with commercial solutions and effectively halts the scientific contribution of any such findings.

1.4 Concrete and verifiable goals

Now that the general idea of the study has been presented it is important to concretize this idea into verifiable questions for which this report is set out to answer. Being able to properly answer the stated problem statement requires a thorough answering of the following questions:

P1: What are (functionally) the prerequisites for using a neural network to achieve object detection for this particular application area; in terms of algorithms or supporting operations that are needed?

P2: What common neural network models exist with characteristics suitable for object detection purposes?

P3: Which of these architectures or combination of models are least demanding computational wise and might realistically allow for real-time deployment on ARM hardware?

P4: What open-source tools and frameworks are available that facilitate the development, training and evaluation of neural network models?

P5: What hardware, setup and image acquisition method (along with relevant image operations) are necessary for training a model with relevant data for the chosen Valmet machine?

P6: What smartphone is most commonly used in Valmet and what are the concrete steps for encapsulating a deployed model within the constraints of the application technology available (android SDK or otherwise)?

These above question directly relate to the first part of the thesis goalpost: providing the interested party with a thorough documentation of what is available today with current technology in solving the chosen problem (machine detail detection). These questions are therefore vital in completing the project's second objective in creating a fully functional prototype. Formulated as following:

P7: Based on the theoretical body acquired through P1-P6 demonstrate if it is possible to:

A) Train a model to successfully detect the chosen machine details.
B) Employ this model on the chosen smartphone and its underlying software framework for real-time purposes.

This should be demonstrated with a fully-functioning prototype and the prototype should be assessed for real-time performance and relevant performance metrics (accuracy, precision, speed). The evaluation metrics used and the working procedure employed to answer problems P1-P7 will be more thoroughly explored in subsequent chapters.

1.5 Contributions

This study is carried out solely and independently by the report's author. The author is responsible for the contents in every chapter.

2 Theory

Since most of the remaining report after this chapter will deal with setting up the methods for answering problems P1 through P7 and successively presenting any acquired results; an attempt will be made to set up some basic theory required to understand any further conclusions.

Neural networks will be explored in 3.1 because of its importance in understanding more advanced architectures which in turn are built by various form of neural networks. Convolutional neural networks are the cornerstone of deep learning models and is given its due write-up in 3.2.

2.1 Neural networks

Neural networks try to model the human brain which consists of neurons. Connected to these neurons are synapses. These neurons aren't floating freely but are connected to one another by synapses. A neuron will activate or "fire" when enough synapses connected to the neuron activates enough to reach a threshold limit set by various factors. It is this firing of neurons that in the biological system corresponds to the brain's innate decision-making process.

We loosely try to model this behavior by programming artificial neurons to behave like its biological counterpart and its connections to behave like synapses. Not every synapse has the same effect on a neuron. The synapses are "weighted" which means some synapses affect a neuron more than others. It is this "weighting" that is crucial to how a neural network is trained and what decisions it takes.

The simplest neural network has one layer of input neurons x_0, x_1, x_2, …, x_k which is connected to one output y by weights (illustrated as connecting lines), see figure 1. This neuronal network can be used for any problem where there exist two classes, e.g. class A, class B. The output function y takes input values and determines which class they belong to. This is also known as a two-category classification problem.[11, pp. 77–78]

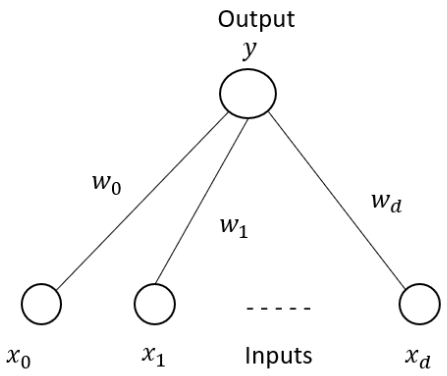

Figure 1. A general single-layer network with one output

This two-class problem is realized mathematically by using a discriminant function. The simplest discriminant function is linear and could in this example be formulated as

$$y(x) = w^T x + w_0 \quad (1)$$

where w is the weight vector and w_0 is any applied bias. If supposedly y is the artificial neuron of focus, then the firing of said neuron is achieved once a threshold -w_0 is surpassed. In this case the firing of the neuron could be considered whatever class belongs to y > 0. The linear decision process for this example could be illustrated by the line y(x) = 0 in x-space, see figure 2.[11, p. 79]

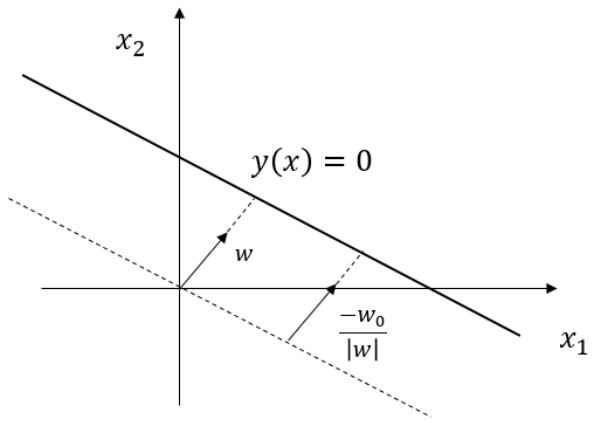

Figure 2. A linear classifier

More advanced applications are often multi-class problems. The generalized multi-class categorization problem looks similar to the previous two-class example but instead of one output y it has several outputs y_1, y_2, …, y_c interconnected in a similar fashion to the input vector x by weights, showcased in figure 3 below.

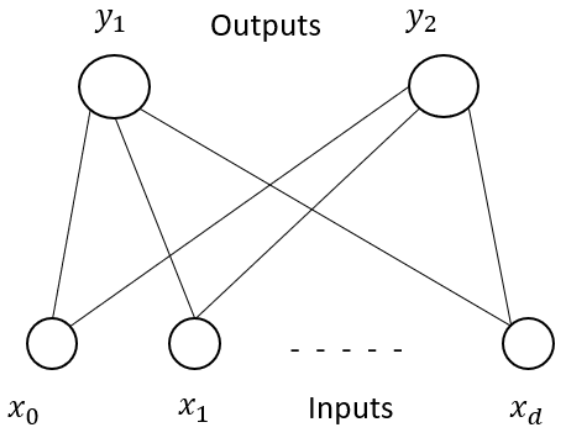

Figure 3. A neural network with two outputs

Even though there are more classes to categorize it is still possible and fully appropriate to use linear discriminants. The difference is that now it is required to have one such linear function for each class. This means that the linear discriminator for class A would be able to distinguish between

A and not-A's. Similarly, the linear classifier for class D can classify between D and not-D's. This is the reason for why it is necessary to have one classifier per class in these types of problems. The generalized linear classifier would take the shape of equation 2 below

$$y_k(x) = {w_k}^T x + w_{k0} \tag{2}$$

where k denotes any of the different classes, C_1, C_2, …, C_k. In order to find the network outputs, the discriminant functions are evaluated for each class on any interconnected input vectors and for each weight vector (and bias) and this gives us the following equation 3:[11, pp. 80–81]

$$y_k(x) = \sum_{i=1}^{d} w_{ki} x_i + w_{k0} \tag{3}$$

The sum of the above equation determines if the neuron y_k in above equation recognizes the input parameters as belonging to class C_k ($y > 0$) or determines it to be outside of that class ($y < 0$). It becomes readily apparent that the classifying function and its weighted components make up for the entirety of how the system behaves once an input vector is applied to the network. Care will be taken not to delve into the various types of classifier but it is important to note that classifiers are not always linear but can take different shapes, especially in cases where a linear function fails to properly separate the data set.[11, pp. 82–87]

While the discussion up until now has treated the behavior of single-layer neural networks, it is of equal interest to know about the training process of said networks. The training of a neural network refers to the changing of the weights to more accurately categorize input data. The common way of training a network is to start off by assuming some values for w and calculate the output y. Now those initial values are used and compared to the expected output value (target value) to find the "error" between them. The last step involves changing the weights while trying to minimize this error until some confidence has been satis-

fied, or the error is practically gone. Once that point is reached the neural network could be deemed to work well in solving the chosen problem. New input vectors applied to the neural network should be classified or grant output values in accordance with predefined expectations; optimally with as small an error as possible.

For a regression network (continuous-valued output nodes) it is straightforward to estimate an error by using the sum-of-squares error function, see equation 4.

$$E(w) = \frac{1}{2}\sum_{n=1}^{N}\sum_{k=1}^{c}\{y_k(x^n; w) - t_k^n\}^2 \tag{4}$$

where y_k is any of the possible output values denominated by k, x^n is the input vector and the weight vector is marked w. t_k is the expected value for the given input vector (where y_k is the actual value). The square error as a difference between actual value and expected value is summed over all training patterns N and number of outputs c.[11, pp. 89–91]

In order to minimize this error, gradient-based optimization algorithms can be utilized. One such algorithm is called gradient descent and is a iterative process of the following form (equation 5):

$$w_{kj}^{(\tau+1)} = w_{kj}^{\tau} - \eta \left.\frac{\partial \mathrm{E}}{\partial \mathrm{w}_{kj}}\right|_{w^{\tau}} \tag{5}$$

where w^{τ} denotes the sequence of weight vectors. This equation illustrates an updating of the future weight vector w(τ+1) by a movement in w-space in the direction of $-\nabla_w E$. The direction moved is chosen to be whatever direction the error E decreases. The learning rate parameter η specifies how fast the weight is changed and is generally a positive number. A too small of a value of η might cause slow learning while a value too large might cause non-convergent oscillations. Optimally a point will be reached where the weight vectors converge to a minimized E and it is at this point we can consider the network to be finished and usable.[11, pp. 95–97]

The prior example works fine for regression problems but most types of neural networks are of classification type, where a problem requires output nodes to converge to a decision or a probability. This is also how the human brain works and is crucial for most advanced machine learning problem areas. The principle uses various threshold activation functions, which transforms the nodes into bounded values (often mapping them non-linearly), otherwise the neural network is just outputting values which have no tangible meaning for more complex problems. It is akin to using a steel hose to water your garden, you can't bend it or go around bushes meaning some spots might be impossible to reach, which often is the case with linear regression neural networks unless the linearity is broken with a non-linear activation function and allow for more complex decision making; in the above example this would correspond to using a bendable rubber (or soft plastic) water hose that can turn and twist.

This nonlinearity necessitates taking greater care in whatever error function is applied and whatever method is used for minimizing the error. The mean-squared-error might still be used for classification oriented neural networks but since this type of error function is based on the assumption of a gaussian distributed target data, the back-propagation will effectively be training our network while missing key information. This works fine for activation functions like the ReLU (Rectified Linear Unit) because even if it is not technically linear in half of its domain, the gradient descent works fine on the side which is linear; this being the side that actually affects the neural network. If instead softmax activation function is used, then problems might arise. The Softmax activation function normalizes and outputs probabilistic values between 0 and 1 for each class. Applying MSE (Mean Squared Error) which was designed for linear regression problems, on a neural network containing probabilistic functions can delay or cause erroneous training. For probabilistic systems it is instead preferred to use cross entropy, see equation 6.[11, ch. 6][12]

$$E = -\sum_{n}\sum_{k=1}^{c} t_k^n \ln y_k^n \tag{6}$$

The sum over the batch of training data with size n where t equals target data ($t = 1$ when input vector with unit k does belong to class c and 0 otherwise) and y is the actual output probability as performed by the neural network. This can be interpreted as the distance between what the neural network "thinks" the answer is and what the answer ought to be. When predicted value and target value is the same ($t = y$) the error function results in 0, which effectively means that there exists no error. If this cross entropy is used instead of MSE and with a proper backpropagation algorithm, it is expected to result in a much quicker convergence to a minimization of the error, resulting in training the network in a shorter amount of time and with less computational effort.[11, ch. 6]

By discussing different activation functions it must be mentioned the idea of neural networks with more than just two layers. For a neural network to even work with data that is not linearly separable, the system needs at least one more layer (so called hidden layer). For now, we are concerned with the type of networks that lack feedback loops; meaning the outputs can be calculated as a function of the inputs and weights, see figure 4.

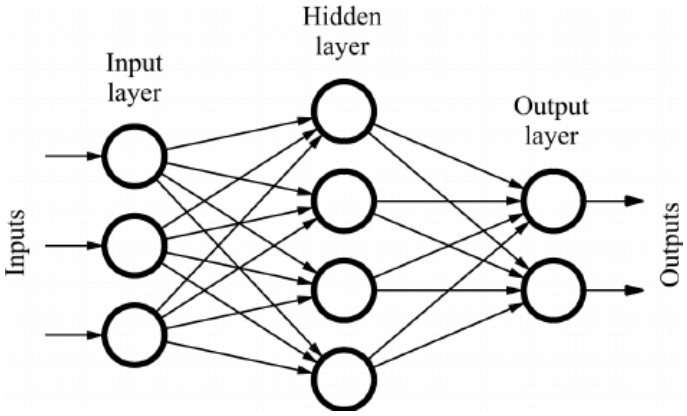

Figure 4. A neural network with one hidden layer

Having a hidden layer adds complexity to the neural network and allows for solving even more difficult problems. The hidden layer is the intermediary layer which connects to the input layer and output layer. Without going through the mathematical properties, it is of importance to know that it has been proven beyond reasonable doubt that a neural network

with one hidden layer is capable of approximating any smooth continuous function. Jumping to two hidden layers grants the network the ability (yet again, while using threshold activation functions discussed above) to represent any decision boundaries and also approximate any mappings with arbitrary accuracy[11, pp. 116–135]. Most machine learning problems can be solved with one or two hidden layers with the exception of highly complex problems, which might be the case in image classification and object detection. In those cases neural networks are realized with highly specialized hidden layers contributing to breaking down an image into patterns. The most notable example is that of convolutional neural networks (CNN's) and their function will be explored further in the ensuing chapters.

2.2 Convolutional neural networks

It is readily apparent that neural networks accept inputs and tries to make sense of the data to either output a value back or classify the input data into pre-chosen classes. What has not been explored is what this input data is and how to prepare it. This is especially important when working with image data. For some readers it might intuitively understood that just inputting pixel data into a neural network would not grant much usefulness; even skipping the problems arising due to the fact that a small image with 300x300 pixels would require 90 000 input nodes. Also, having fully connected (FC) layers of that size would be a computational nightmare, and grant little ability for the neural network to understand the data in any generalized way.

In order to solve this dilemma, the image is turned into features by inputting it through various filters. One filter might detect edges, another filter might pick up "roundness". The filters usually comes in various window sizes like for example 3x3 pixels big.[13] These filters produce so called feature maps that are smaller than the input image and provides the system with valuable information about patterns in the image, see figure 5.

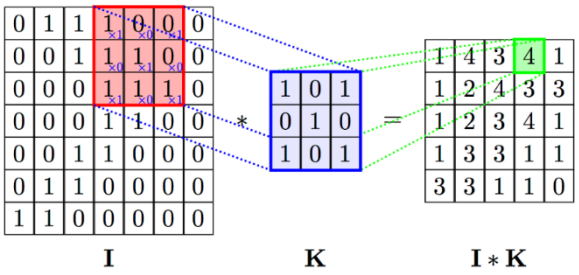

Figure 5. A convolutional filter applied on a matrix and its resulting output[14]

If the image has three color channels (RGB) then a filter is often applied on each channel, summing their final input before producing the feature map. Applying thresholding or activation functions on a feature map results in an activation map which thresholds in order to effectively highlights "strong features". An especially important function in this regard is the previously mentioned ReLU, see figure 6.

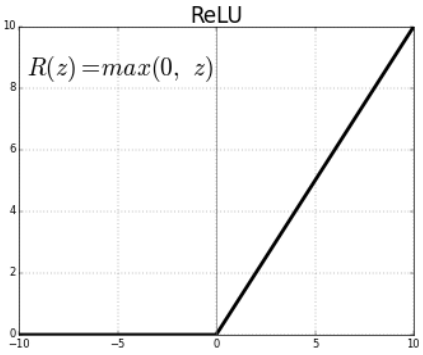

Figure 6. A rectified linear unit[15]

To reduce the computational effort even more, a CNN typically employs pooling layers. A pooling layer can be thought as a "summarizer" that takes the most important features from the activation map and outputs them with reduced spatial size. The most common technology employed is called "max pooling" and is simply a filter that outputs the highest value of a neighborhood of neurons NxN. The pooling layer also makes the classification translation-invariant. This means that features and patterns are still relevant regardless of where in the image they are located. The example below shows a pooling layer applied to a 4x4 activation map; the pooling function itself having a window size of 2x2 and stride 2

(moves 2 neurons in the chosen direction). The stride is often chosen so as to not have overlapping windows. Figure 7 visualizes the max-pooling behavior.[16][17, pp. 339–347]

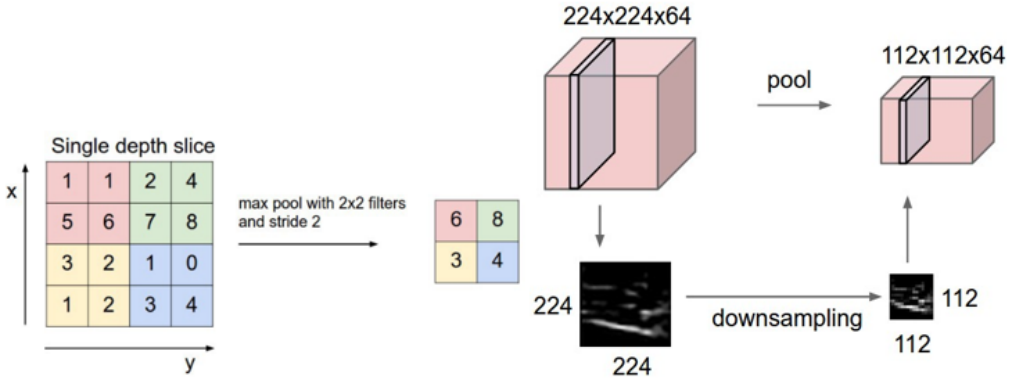

Figure 7. The process of max pooling operation on a 4x4 matrix[18]

The convolution layer, ReLU layer and pooling layer is often done repeatedly in succession. The first convolution layer might be excellent at catching low-level details but applying several convolution layers in succession leads to the system catching more high-level details. Every object has a geometric shape that is made up of several low-level features. Being able to convolute these low-level features in the right amount of magnitude gives our system a chance to learn what combination of features represent a given object (high-level detail).[16][17]

The final hidden layers in a CNN is the more traditional fully connected layers that were explored in previous chapter. All the layers (CONV, ReLU and POOL) preceding the fully connected layers are therefore operations meant to structure and prepare the data into useful features that enables the FC layers to capture relationships between the data and classify it. The final network often takes the shape as seen in figure 8.

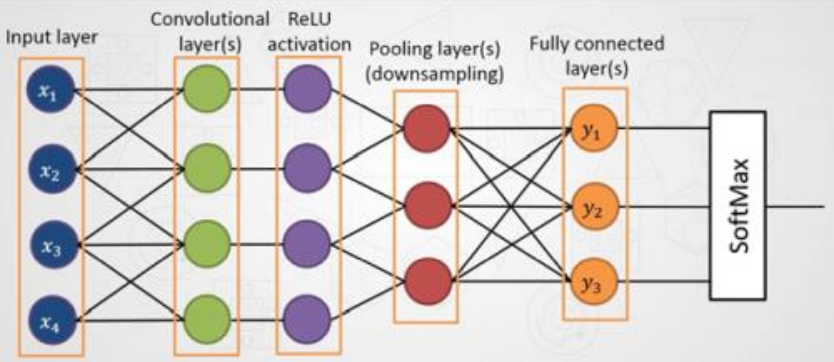

Figure 8. Basic classification CNN architecture

Finally, the SoftMax activation function (or other type of activation) enables the network to output probabilistic output values that can be interpreted as likelihoods of the outputted class being in the right category.

Training the CNN is done by applying back propagation like illustrated in figure 9.

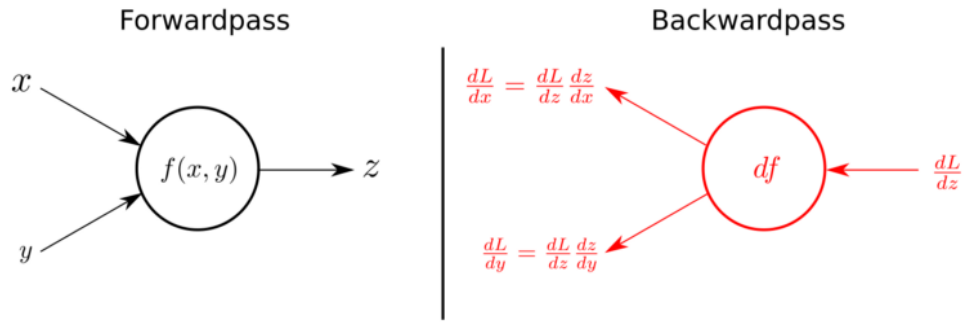

Figure 9. Back propagating technique employed in training a CNN

A backward pass receives the gradient of the loss function with respect to the output. The gradient is calculated by any of the relevant loss functions described above, for example cross entropy. There are different algorithms for this but most of them aim at achieving the same thing - minimizing loss by using the already known loss to predict better weights on the network.

2.3 Object detection

Finally in order to achieve object detection it is easy to see the problem divided in two steps:

- Localizing the object within an image
- Classifying the object

Of these two steps, previous chapters might have given the reader a grasp and a rough idea of how classifying can be achieved. Using a CNN the object is manipulated through various filters to find patterns that can help the CNN's final layers get an idea of what kind of object has been provided to the input layer. In most cases, at least the theory behind this, might become straight-forward and clear as how to proceed. The first step is trickier and often quite computationally intensive. There exist various forms to tackle a localization problem, which can intuitively can be understood as a regression problem. This effectively turns object detection to both a regression and classification problem. Applied algorithms and models to tackle this will be explored in the next chapter. The reader might now have the necessary background information to partake in the literature review aimed at documenting possible strategies for creating such a system as outlined in earlier chapters.

Some models include both steps while other models might require the programmer to provide localization prior to being applied. The diverse range of models and their usage are outlined in the next chapter *Model review.*

3 Model review

This chapter aims to answer problems P1 through P3 by surveying recent and relevant papers detailing neural network models that can be used for object detection purposes. This study is restricted to models employing convolutional neural networks in order to fulfill the limitations outlined under the report's scope (end-to-end systems in terms of feature extraction). The papers describing each and every architecture is detailed and so an attempt will be made to summarize a model's key points only and in each succeeding subchapter highlight any differences from one model to the next.

3.1 R-CNN

The first attempt at incorporating CNN for an object detection problem was proposed by Girschick et al. in 2014[20]. The method proposed was called R-CNN and can be thought of as region proposal with CNN. Looking at the proposed system in a top-down manner reveals three important structures. The first system consists of an algorithm responsible for generating arbitrarily defined regions which might possibly contain objects (there are several systems aimed at solving this type of problem). The second module consist of a traditional CNN that interprets every proposed region into appropriately sized feature vectors. The third modus operandi of R-CNN is a layer of Support Vector Machines (SVM). The SVM is a linear classifier that is trained independently for each class and attempts to classify the feature vectors to their respective class membership. Figure 10 illustrates this process.

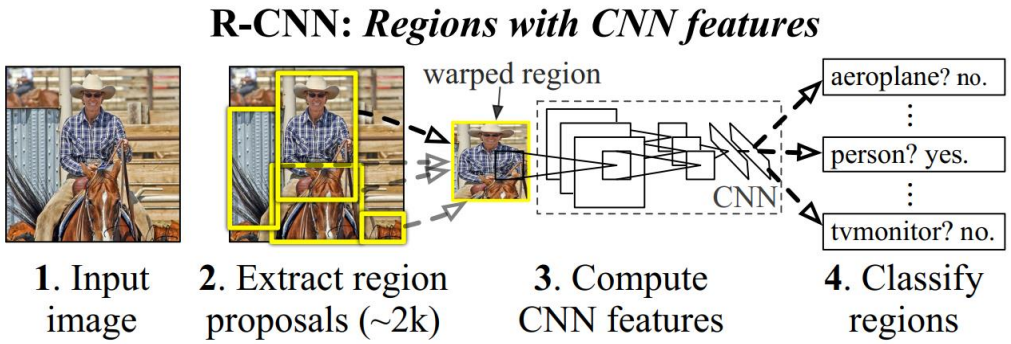

Figure 10. The region-proposal, CNN operations and classification in R-CNN[20]

There exist a few different approaches to proposing regions that are compliant with the proposed R-CNN model. The author himself did not specify which technique should be used. The region proposal algorithm is tasked with finding regions that might possibly contain objects. As long as this is fulfilled, false-positives are not too problematic since it is the CNN along with the classifier that is responsible for actually deciding if the proposed regions contain any object of interest. However it is typically understood that proposing too many regions can slow down the system considerably since every region needs to analyzed by the object detector[20][21]. For region proposal, two methods are worth noting and commonly used. Selective Search (which the R-CNN paper itself implements) and Edge Boxes[22].

Selective search tries to capture interest regions (without over-producing too much) these by a few different approaches. The most important is the use of segmentation methods based on work from Felzenszwalb and Huttenlocher(2004)[23]. Not going into specifics of the segmentation process the main ideas are visualized in figure 11. In the figure a segmentation is seen occurring at the first step and then a grouping of adjacent regions by texture, color and region size. This makes possible individual object stand out easier and on this over-segmentation template, algorithms are applied to generate regions based on different parameters. The end result is much fewer regions outputted than merely producing regions at random.

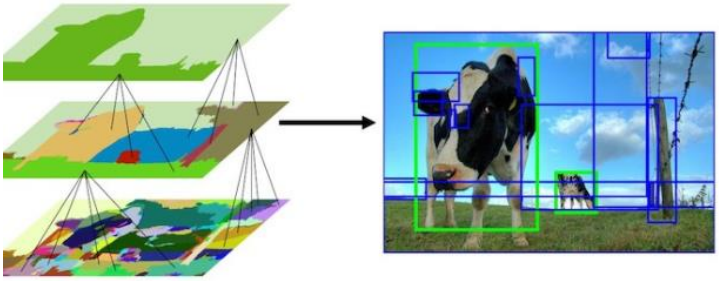

Figure 11. Selective search and its region-generating procedure by segmentation[24]

Edge Boxes is another such region-proposing algorithm and might lend itself to more easily conceptualizing. The authors motivate the method by recognizing (from observation) that the number of contours enclosed by a bounding box increases the indication of an object being present in the region. An enclosed contour within a box can be seen as edge pixels which belong to the contour and are located inside the box. Edges in this manner are likely to pinpoint possible object existences. Algorithmically, a scoring function is employed that analyzes edge strengths within a box and computes the weighted sum of all of these. This is then used in combination with smart search strategies to produce considerably "relevant" object proposals. This method is comparable to selective search in terms of accuracy.[21]

Given a well-suitable region proposal system, it is expected for the R-CNN to first warp all incoming region proposals to a fixed size as required by the convolutional neural network (originally 227x227 RGB). The intended R-CNN system described in the paper then passes these on to five convolutional layers and ends with 2 fully connected layers. This typically produces a 2000 x 4096 feature matrix which is fed into the SVM matrix where it is classified. This forward-pass is done separately for each region proposal. Due to no inherent parallelism, this model's performance heavily varies on a well-chosen region proposal method.[20]

For the sake of completeness, it's important to note the bounding-box regressor employed. This was implemented to improve localization performance. After SVM classification and scoring it is possible to compare the proposed region to the ground-truth box and apply regression methods

to learn and predict a more correct detection window. Keeping track of normalized (center points) dx, dy, (width)dw and (height)dh coordinates enables the regressor to employ algorithms to push the proposed image region into ground-truth box coordinates. Further reading is possible on Girschick's paper (2014).[20]

3.2 Fast R-CNN

Fast R-CNN is an improved technique building on previous R-CNN principles. To make the model even faster, Girshick (2015) introduced parallelism into the system to break the previously slow and expensive pipelines. Instead the paper proposes a single-stage training algorithm by feeding the CNN layers with the "whole" image at once. This produces a feature map of the image (as explained in theory chapter). This feature map is of the same size as the original image. This allows for applying a pooling layer on the feature map for every proposed region of interest. See figure 12.[25]

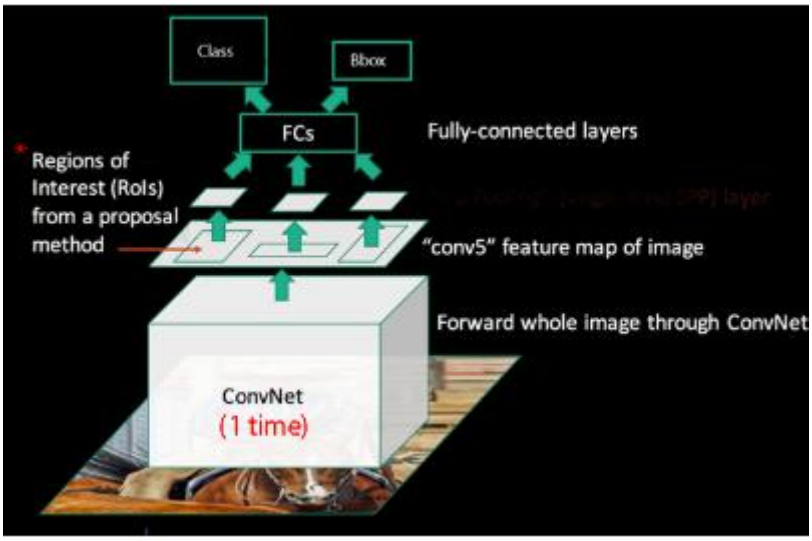

Figure 12. The Fast R-CNN architecture creates RoI's directly on the feature map[26]

This leaves the system with several slices of convoluted feature maps for each object proposal. The first problem that has to be addressed is their inevitably varying sizes. A fully-connected layer only accepts fixed inputs. This is solved by applying smart pooling algorithms that divide all

region of interests into a fixed number of grid cells; apply max pooling on the cells; conjoining them to feature matrices of desired size. These can be seen in figure 12 where they are fed into the FC-layer.

The FC layer further connects to two final layers:

- Softmax layer which estimates class probabilities
- Box regressor which employs regression to refine candidate boxes.

Due to its end-to-end nature, training a Fast R-CNN is more intuitive than the earlier model. The connectedness of the system enables the use of previously mentioned back-propagation algorithms and gradient descent operation to shape the hidden layer weights. Avoiding the use of multiple forward passes for each proposed region allows the model to cut down on both training and testing time considerably. The remaining factor that is constant through both R-CNN and Fast R-CNN is the use of an external region proposal model. This is also the reason why the chosen proposal model often takes the role of a bottleneck in the Fast R-CNN and disallows even faster execution times.[25]

3.3 Faster R-CNN

The knowledge of the region proposal bottleneck prompted researchers to find a solution that could achieve the same functionality but at reduced computational cost. Ren. et. al. (2016) experimented with the idea of discarding state-of-the-art region proposal methods such as Selective Search and Edge Boxes in favor of computing proposals with an innovative and repurposed deep CNN[27]. The idea being that a such a system could be designed to share the already existing convolutional layers in the object detector and thus reducing computational cost considerably. The referenced paper devised a method called Region Proposal Networks (RPNs) that use the existing convolutional features of Fast R-CNN but augments a few extra layers to add object proposing characteristics.[27]

RPNs works by passing a sliding window over the CNN feature map and at each window outputting K potential bounding boxes and scores for how accurate each anchor box might be. For each anchor box one bounding box is created and a score per position in the image. These bounding

boxes are then forwarded into the CNN for classification and to tighten the bounding boxes using previously mentioned regression methods. Figure 13 illustrates this process.

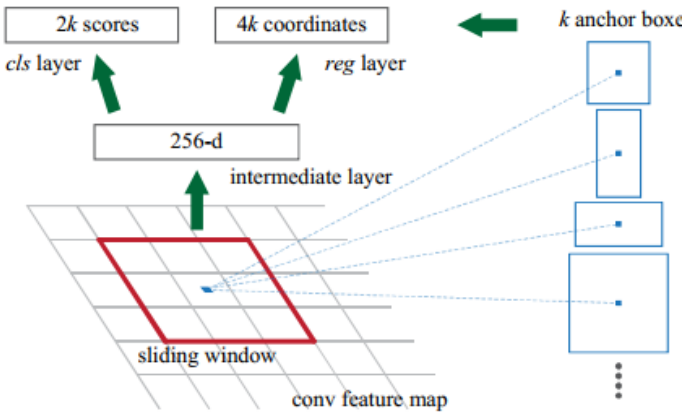

Figure 13. Anchor boxes used in the proposal algorithm[27]

The method proposed provides translation-invariant anchors. This means that regardless of the object location in the image, the method should predict a proposal for that object. Discarding spatial information means a reduction in model size and provide even more cost efficiency. Creating a region proposal algorithm within the already existing CNN architecture also grants it the ability of utilizing back-propagation and stochastic gradient descent for training. This unique solution allows Faster R-CNN to provide frame rates in the realm of real-time applications.

3.4 SSD

Single shot detection appeared as a result of seeking better real-time performance than Faster R-CNN and similar methods. It concerned itself not with accuracy but model performance. SSD share some similarities with Faster R-CNN. It utilizes bounding boxes (albeit fixed) and implements various convolutional filters to feature maps in order to predict category scores and box offsets. The SSD, like its name suggest, work by providing object localization and object classification through one single pass

through the network. In the paper, C. Szegedy et al.(2016) also provided the multibox technique used for bounding box regression.[28]

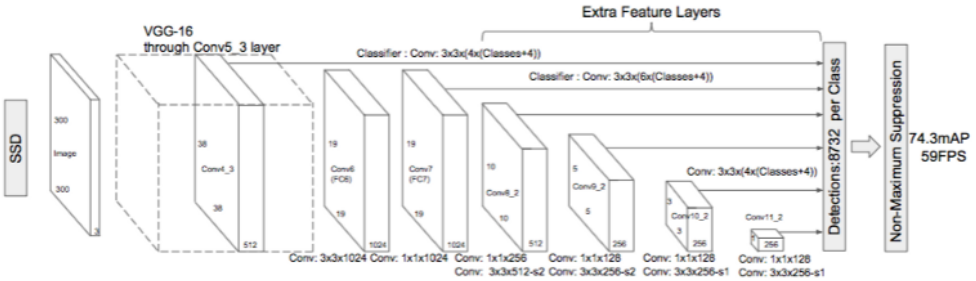

Figure 14. An overview of the SSD architecture and its various layers[28]

Overview of the SSD approach is seen in Figure 14. The readily apparent observation is the lack of a fully connected layer. Instead from the sixth convolutional layer it adds on extra convolutional layers which enables it to perform feature extraction for different scalings of an object while simultaneously decreasing the input size for each pass through subsequent layers. This is important because the SSD skips the costly techniques utilized in Faster R-CNN to propose regions that are scale invariant, and these extra convolutional layers helps the architecture return scale invariance back to the system.[28]

Training an SSD model consists of the predefined boxes of various scales and sizes being evaluated in order to relate them to the box offsets and prediction confidence compared to the ground truth. This information is used to find a confidence loss and localization loss and incorporate it into a total loss (weighted sum of the two). Similarly to the R-CNN family, this is done to regress the (x, y) center, width (w) and height (h) to optimal locations and improve future spawned boxes.[28]

Various implementations, tweaks, fixes and loss changes (for training) seem to exist for SSD. The main focus for this report is the use of additional CNN layers and the removal of the FC layer which helps reducing computational cost. A high level focus enables a realizing that SSD is a good approach to enabling realizable real-time implementation.

3.5 YOLO

YOLO[29][30] (You Only Look Once) is an interesting approach for an object detection architecture and was first outlined in Redmond et al.(2015)[29] but went through several revisions and is up to iteration called YOLOv3[31]. An overview of the general YOLO architecture can be seen in Figure 15.

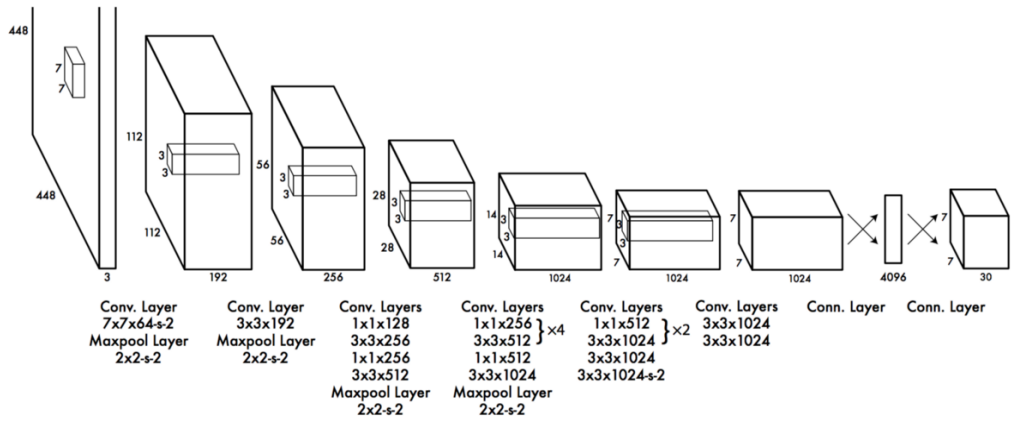

Figure 15. The original YOLO architecture and its layers responsible for prediction[29]

It starts by a classical bounding box prediction. Dimension clusters are used as anchor boxes and these are used to predict coordinates for each bounding box. The coordinates are used during training to compare against ground truth boxes and regress towards smaller errors. Class prediction is performed by letting each box predict any of the available classes that it might contain. Cross-entropy loss is used for class predictions. To solve prediction with different scales, YOLO extracts features using feature pyramid networks, similar to how SSD solves this problem. For feature extraction it uses up to 53 convolutional layers, much more than any other model. One key difference to SSD is the usage of fully connected layers in the original version, as can be seen in figure 15. Later versions remove this fully connected layer and start to liken more to SSD.[30]

3.6 Base nets

For every object detection architecture there are tons of variations and augmentations possible. Due to the impracticality of treating every such iteration the study will concern itself only with more recognized high-level changes such as changing the object detectors "backbone" network, here from referred to as base net.

One common way to modify object detectors such as Faster R-CNN and SSD it to change the base net. The base net refers to the CNN used for extracting high-level feature data from the image and creating a feature map. The rest of the Faster R-CNN and SSD architecture works from the feature map and forward to localize and detect objects. This means that it is fully possible to change anything before the feature map without breaking the architecture. Figure 16 shows a variant of the VGG base net being used in the SSD detector architecture.

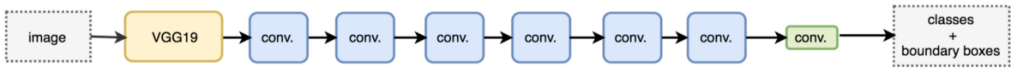

Figure 16. The positioning of a base network is shown for the SSD architecture[32]

YOLO is an exception due to using the Darknet as a base which is fine-tuned to be in a specific way as to allow the rest of the YOLO architecture to predict classes and bounding boxes. It does not lend itself to changes in a practical way for a change in the feature extraction CNN to be practical and as such considerations regarding base nets for YOLO will not be explored. This object detection architecture will be analyzed as-is.

VGG[33]**.** This base net is originally used in SSD and proves to be quite effective. It has six different configurations and uses classical 3x3 convolution layers as described in theory. The least demanding configuration has 8 convolutional layers and 3 FC layers and and it works its way up to 16 convolutional layers in the best performing version (but costlier). The FC layers need to be removed before VGG can be used in Faster R-CNN or SSD for object detection purposes.

MobileNets[34]. Howard et al.(2017) laid out interesting ideas for a deep CNN. Instead of standard convolutional layers it proposes the usage of depthwise separable convolutions.

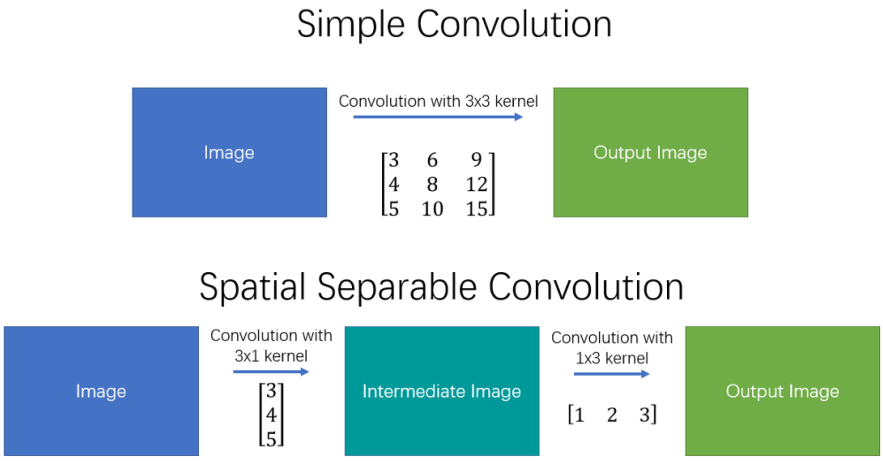

Figure 17. The difference between simple versus separable convolution[35]

Discarding 3x3 kernel shapes in favor of 3x1 and 1x3 kernels. The first is called depthwise convolution with a filter. The second is a pointwise convolution of 1x1 size that iterates through every single point. The figure says 1x3 because an image with RGB contains 3 channels. Separating the convolutional operation into two separate more easily computed convolutions like this helps reducing computational cost. This has the disadvantage of reducing the amount of parameters that can be retrieved out of convolution.

Inception[36]. This model, like MobileNets, attempts to bring down computational cost. Inception is therefore a collection of methods to scale up networks in favor of computation efficiently, by employing factorized convolutions and aggressive regularization. It doesn't push down kernel sizes as aggressively as MobileNets so it stills retains useful feature rich channels.

3.7 Comparison

As implied, out of the R-CNN family, Faster R-CNN is much faster due to the RPN network that utilizes the already existing CNN network for

object proposals. R-CNN and Fast R-CNN are therefore immediately discarded as choices for implementing on ARM.

YOLO is very fast and uses predefined grids to find objects in a mostly parallel fashion. This gives great FPS (frames per second) performance[37]. SSD might not be as fast due to lacking the parallel modelling shown in YOLO. SSD and later revisions of YOLO do not however use any fully connected layers. As discussed in the theory chapter treating the CNN architecture, a FC layer is the most computationally intensive operation out of all layers. Faster R-CNN uses a fully connected layer. The reason why YOLO is par with SSD is due to the fact that the architectures are deployed on GPU with inherent parallelization techniques. Lacking these, a deduction can be made that ARM architecture won't be able to showcase this type of speed. Instead SSD might be a more favorable solution.

MobileNetV2 has shown to be most computationally light compared to other deep neural network architectures.[38] Drawbacks include being the least accurate. Since this paper is not concerned with achieving optimal accuracy, MobileNets prove to be a high-value candidate of choice.

YOLO version 2 is comparable to SSD but its feature extractor is unchangeable and much heavier than a combination of MobileNetV2 and SSD. YOLOv2 contains 50 million parameters compared to a MobileNetV2 + SSD combination which only contain 4.3 million.[39] **Note.** This comparison was done with YOLOv2 which uses DARKNET-19 containing 19 convolutional layers for feature extraction[30] as opposed to YOLOv3 which instead uses 53 layers[31] making YOLOv3 possibly even more computationally intensive than SSD+MobileNetV2 and even YOLOv2. 50.7 million parameters correspond to 17.5 billion multiply-add (MAdd) operations while MobileNetV2 + SSD require only 0.8 billion MAdd. These comparisons were done on the same data set.

4 Model realization

This chapter aims to further understand problem P4 through P6. The first chapter discusses software frameworks while the latter two sub-chapters pinpoint important ideas for creating training data and deploying a model to android.

4.1 Software frameworks

To discard smaller, less useful and non-mature frameworks; and to facilitate easier development; a search is done for frameworks based on following criteria's:

- At least 3 million page search hits on Google.
- Actively developed.
- Uses python for quick prototyping.
- Open-source.

Pytorch is an open-source deep learning platform that supports research prototyping. It's quite a new addition to deep learning frameworks. It's based on a LUA framework but runs on python. It's a python package that provides multidimensional arrays (called tensors). It uses dynamic computation graphs and is able to compute gradients automatically. It's commonly used for images (detection and classification).[40]

Tensorflow is another platform for developing deep learning applications. It's supported and initiated by the Google team and it is completely open source. It also includes TensorBoard which is an innovative visualization framework to help evaluate a model visually. Instead of dynamic graphs it uses fixed graphs.[41]

Keras is meant to be a very high level API for working with deep learning. Aimed at beginners and people who first start out learning about neural networks. Using Keras exclusively might prove less flexible but since it uses Tensorflow as backend; many features are inherited.[42]

The main selling points of every framework can be summarized as following:

Tensorflow has TensorBoard which provides easy visualization. It can be programmed using both C++ and Python. Has a large supporting community and supported by Google. Includes a lot of learning resources. Drawbacks include: Static computation graphs and requires more extensive programming experience than Keras.[41]

Keras is high level and provides opportunities for fast experimentation. Uses TensorFlow as backend. Disadvantages: Less flexible, not as much learning resources despite being easier.[42]

Pytorch is gaining traction and is developed by Facebook's AI research group. Instead of fixed graphs they use dynamic computation graphs. Might be easier to use than TensorFlow. Very popular amongst researchers but not always suitable for production deployment.[40]

4.2 Creating training data

The most commonly used labelling format is called PASCAL VOC[43]. This one will be used due to being exclusively used across frameworks. There exist several tools available for labelling images. VoTT is an open-source labelling program from Microsoft that can export to several different formats. A camera is needed for extracting picture and they need to be stored digitally on a computer for further annotation. The resolution of the pictures might be important but it is relatively easy to resize images. Labelling needs to be done so that the model used can see where our ground-truths are located for each object. It is common to divide the images so that training data is 70-80 percent and evaluation data (the images we use to evaluate themodel) is around 20-30 percent.[44]

4.3 Android-powered device

Valmet currently issues a Samsung Galaxy S8 smartphone to its employers. This type of phone ranks 114th in general android performance as of 4th June 2019, according to PassMark evaluation software[45].

To build android applications Android Studio is often used along with the necessary Android Software Development Kit (SDK).[46]

TensorFlow is the only deep learning collection to offer a complete skeleton for deploying NN models on mobile and IoT devices. This skeleton is provided as is with the android SDK as prerequisite and structured for easy integration in Android Studio. Named TFLite for Android.[47]

5 Method

5.1 Hardware and preparation

Training will be performed on a Intel(R) Xeon(R) W-2123 (3.60 GHz) processor. The system is a 64-bit Windows 10 Operating system and has 32.0 GB of RAM memory. GPU acceleration has not been used due to limitations in acquiring proper hardware. Since the main idea is not to achieve best-possible accuracy, training is not required to be run for more than necessary (defined as 50-60% accuracy for this use case). Instead a productive desktop processor of recent generation will suffice in fulfilling the goals.

Tensorflow framework will be used for programming and training the model. TensorBoard will be used for evaluation purposes. The Tensorflow framework is used due to easily accessible visualization tools and the existence of android abstraction API's for deploying trained models through android studio.[47][48]

The model will therefore be deployed on an android-powered Samsung Galaxy S8 device using the TensorFlow framework for android deployment called TFLite.

Image acquisition will be done through the Samsung Galaxy S8 camera. 159 pictures will be used for training and 49 pictures for evaluation.

Labelling is done with VOTT (open-source) and exported to Tensorflow records. The annotation follows the PASCAL VOC structure but instead of XML files, TFRecords conjoin both the bounding box information and the image into one single file[49]. See implementation.

5.2 Model specification

For the model, a traditional SSD implementation was implemented with MobileNetsV2 as base net. The image tensor is resized to 300x300 pixels to help with computational efficiency. In chapter *Model review* the conclusion was made in favor of SSD due to cost efficiency by skipping the FC-layers and relying on several classification steps between convolution layers to solve scale-invariance. The MobileNets showed promising benefits

due to separating the standard convolution operation into depthwise and point-wise convolutions which are easier translated matrix operations and require less computational effort. The implementation of SSD with MobileNetV2 is shown in the implementation chapter.

Localization loss will be a combination of L1 loss and L2 loss. See equation 7 and equation 8.

$$L_1^{LossFunction} = \sum_{i=1}^{n} |y_{true} - y_{predicted}| \tag{7}$$

$$L_2^{LossFunction} = \sum_{i=1}^{n} \left(y_{true} - y_{predicted}\right)^2 \tag{8}$$

Smooth L1 loss will be implemented. If A is an arbitrarily chosen value and if the absolute value in above equation is greater than A then L1 loss is used. If the absolute value of |true - predicted| is less than A the L2-loss is used. This way the loss function provides steady gradients for higher values and less oscillations when the absolute difference value is closer to zero. See implementation.

Classification loss will be calculated by cross entropy (after a weighted sigmoid activation in the last output layer). Refer to theory chapter.

Note, losses on training set will be disregarded. In the case of a small training data it does not provide useful information. An improvement in loss for the training data might imply improvement but it can also be a mere overfitting and thus increasing actual real-world loss.

Instead these losses are only calculated for the evaluation data. The evaluation data will be configured in the TensorFlow framework with the necessary runtime functions and the results will be visualized with the accompanying TensorBoard script.

Note. Most hyperparameters related to the training of a deep learning model like for example the learning rate (used in the gradient descent function) are not mathematically derived but empirically chosen to whatever constant best performs. These hyperparameters are extensive and will therefore not be explored but set in whatever default value is provided by the framework. This includes helper-algorithms that aim to help a system's loss functions to not get stuck in a local minima during training.

5.3 Evaluation criteria's

Scientifically explored and sound criterias for assessing performance of object detection can be found in various papers throughout the model review[27][28][30][33]. The most commonly used one is mAP and was defined by the Pascal VOC organization[43]. mAP stands for mean average precision. The average precision is defined as the integration of the precision/recall curve. These two are in turn defined in the following way:

Precision = true positives / (true positives + false positives)

(How many selected items are relevant?)

Recall = true positives / (true positives + false negatives)

(How many relevant items are selected?)

Calculating mean average precision for full 100% localization matches would not prove very useful since full matches are rare. Instead it is useful to define mean average precision for varying matches in the form of *Intersection over Union* defined as area of overlap divided by area of union, see figure 18.

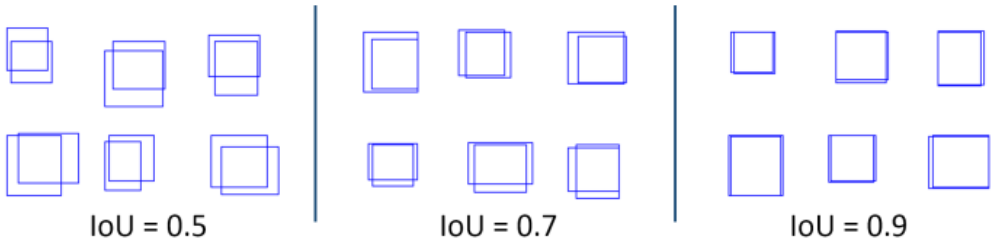

Figure 18. Different Intersection-over-Union (IoU's) are illustrated

The model deployment on the Android device will be evaluated for *inference time*, defined as the time it takes for the model fully realize a class prediction attempt on the image. It will be measured in milliseconds and related to the FPS in order to evaluate the real-time capabilities.

Note. Training will be stopped once classification loss and localization loss start to converge or start making negligible improvements.

6 Implementation

This chapter describes the implementation process for training, evaluating and deploying the model.

6.1 Prerequisites

The following programs or libraries are necessary for the implementation phase:

- Python version 3.4 or later
- TensorFlow version 1.13.1 (CPU-optimized version for AMD64 architectures)
- Protobuf 3.0.0
- Python-tk
- Pillow 1.0
- lxml
- Jupyter notebook
- Matplotlib
- Cython
- contetlib2
- cocoapi

Most of these can be installed utilizing Python pip installation tool by running pip install "package" from any python enabled command line.

6.2 Software installation

TensorFlow has a GitHub with over 70 repositories including documentation, example code, tools, projects, agents and various models.[50] It is of primary interest to explore the *models* repository which contain different implemented models. This repository contains three main folders, "official models" - collection of well-maintained tested and optimized models using TensorFlow's high level API's. "research models" contain models implemented by researchers and are not officially supported by TensorFlow. The sample folder contain code snippets and examples. Lastly the tutorials folder includes tutorials.

This implementation will be utilizing tools in the folder “research” which includes the Tensorflow object detection APIs.

This repository can be cloned with GIT to be used locally.

6.3 Tensor

Tensors can be described as follows:

> “TensorFlow, as the name indicates, is a framework to define and run computations involving tensors. A tensor is a generalization of vectors and matrices to potentially higher dimensions. Internally, TensorFlow represents tensors as n-dimensional arrays of base datatypes.”[51]

The tensor is the object that is manipulated by the model and the variable vector passed around to functions. All TensorFlow programs are built by graphs containing tensor objects. These graphs detail how each tensor is computed and any program created is written to run specific parts of these graphs based on what is required and what results are sought after. More information can be found at the TensorFlow documentation.

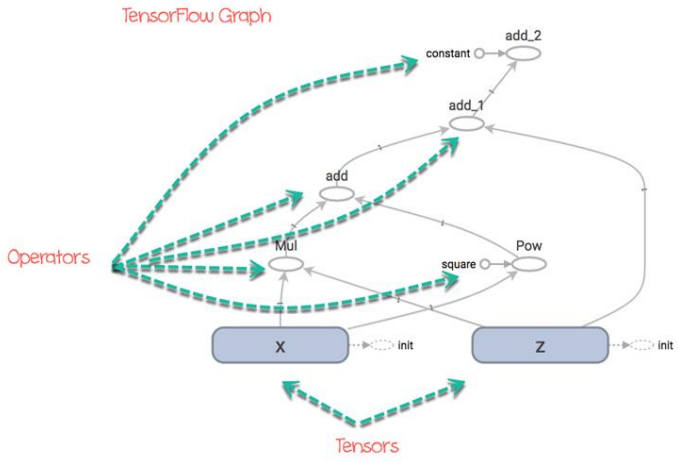

Figure 19. A tensor is a vector of any allowed datatype

6.4 Functions

Training the model is done by utilizing the routines in model_lib.py found under \TensorFlow\models\research\object_detection\

This python script contains the hooks to TensorFlow API necessary to construct the model, its inputs and setting up the training environment.

The workflow can be explained as following functions (in the order found in the library not in order of execution):

1. Creating a map of names that help building the model.
2. **_prepare_groundtruth_for_eval().** Extract the groundtruth information from the chosen detection model and prepare it into appropriate TensorFlow-realized tensors. It can be modified by three arguments, allowing the choice of detection model, setting whether detections are class-agnostic (true/false) and a flag limiting the number of ground truth boxes. This routine returns FLOAT32 tensors of groundtruth boxes, masks, annotations and other model specific information.
3. **unstack_batch().** Retrieves a dictionary of all available batched groundtruth tensors. Tensors in the dictionary contain batch_size, height, width, channels, num_boxes and other box spatial information. This third routine is responsible for unstacking all necessary tensors from the dictionary and returning unstacked and unpadded tensors to be passed wherever necessary.
4. **create_model_fn().** Training and evaluating a model in TensorFlow is done by using the Estimator class. It is necessary to instantiate an object from this class and wrapping it around a model specified by *model_fn*.[52] This fourth routine helps creating the model_fn function. It accepts any hyperparameters, a *detection_model_fn* object, configs and postprocess flags. Based on these arguments it constructs the model function that will later parse *model_fn* to be used by the Estimator class.
5. **model_fn().** This function creates the object detection model with the same name *model_fn* (it is in reality an object that encapsulates all the parameters contained in the model[53]; it is this object that is passed to the Estimator). This function takes feature tensors as

arguments, labels (ground-truth tensors), parameters. The parameters specify if the function is called for training, evaluation or prediction purposes.

6. **create_estimator_and_inputs().** Here the Estimator object is created. The function takes a specified pipeline config file (see chapter below) as argument, a HParams object containing hyperparameters, number of training steps and an integer value specifying how often evaluation examples should be processed. This function returns an 'estimator' object along with training input function, evaluation functions and their names inclueint prediction input functions.
7. **continous_eval().** This function takes the generated 'estimator', along with arguments specifying model directory, input vector, and number of training steps and performs evaluations on the checkpoints found in the model directory.

In order to interface with this library in a structured way it is possible to use the model_main.py binary which sets all the necessary flags and calls the functions from the library with extracted arguments from the chosen hyperparameters, model and other functions. This is done by giving the binary a text file with necessary information called *pipeline.config*. It is here where most of the model parameters are specified, including base net and object proposal method.

6.5 Model configuration

The following parameters are specified fed to the model_main.py binary (all these specialized function python scripts can be found under the *object_detection* folder):

- Architectural model: SSD
- Feature extractor: ssd_mobilenet_v2

Note. These two arguments above will be matched and the program will point to ssd_mobilenet_v2_feature_extractor.py during runtime.

- Depth multiplier: 1.0
- Minimum depth: 16

- Regularizer function: l2_regularizer
- Initializer: truncated_normal_initializer
- Activation: ReLU_6
- Box_coder: faster_rcnn_box_coder.py with parameters:
    - y_scale: 10.0
    - x_scale: 10.0
    - height_scale: 5.0
    - width_scale: 5.0

Note that this does not mean the usage of a Faster R-CNN box regression. It is merely a specification on the way boxes are coded. They were originally used for Faster R-CNN so this name was kept.

- Matcher: argmax_matcher.py with following parameters
    - matched_threshold: 0.5
        - Threshold for positive matches
    - unmatched_threshold: 0.5
        - Threshold for negative matches
    - ignore_thresholds: false
    - negatives_lower_than_unmatched: true
    - force_match_for_each_row: true
- similarity_calculator: iou_similarity
- box_predictor: convolutional_box_predictor.py
- initializer: truncated_normal_initializer
- anchor_generator: ssd_anchor_generator
    - num_layer: 6
    - default aspect ratios
- post_processing: batch_non_max_supression
    - score_threshhold: 0.3
    - iou_threshold: 0.6
    - max_detections_per_class 100
    - max_total_detections
- localization_loss
    - weighted_smooth_l1
        - Combination of L1 and L2 loss (see method)
- classification_loss
    - weighted_sigmoid (cross-entropy as described in theory)

Learning rate is set to 0.004 (constant). The images are resized to 300x300 pixels in order to reduce training cost. To achieve this it is possible to utilize the fixed_shape_resizer by specifying this flag in the config file. See the Appendix A for the full pipeline.config file.

6.6 Labelling regions

VoTT software is used to create regions for every image. Six labels are created: Feeding Wheel, Servo Drive Unit, Twisting Drive unit, HMI (Human Machine Interface), Wire Feeding Door and Wire Guide System. The labelling process can be seen in figure 20.

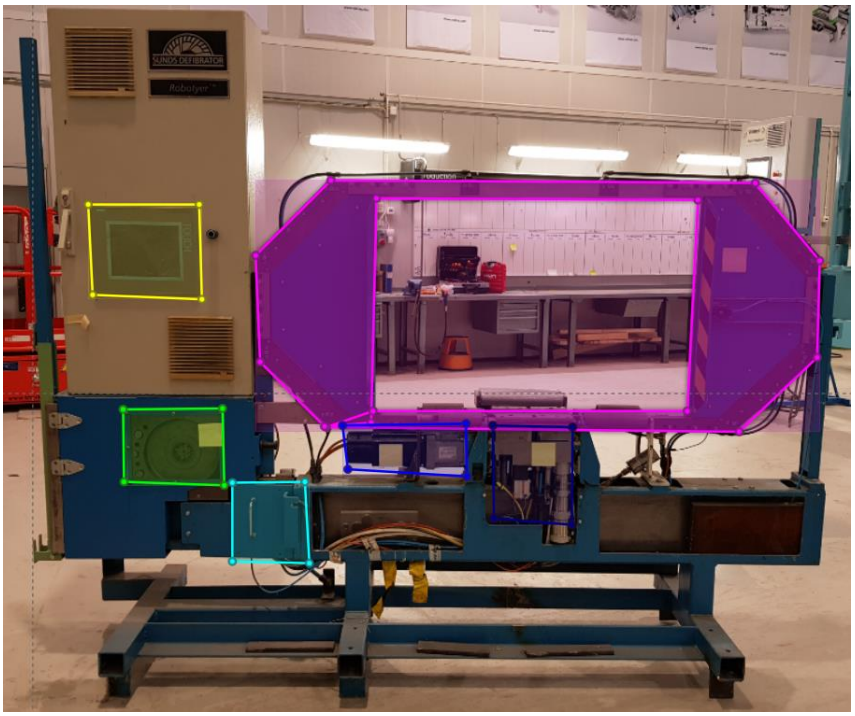

Figure 20. An illustration of using the VoTT tool to label regions

Figure 21 illustrates tagged regions and number of regions per class. There are total 168 tagged images and their distribution is shown in the bar graph in figure 21. The figure shows that 159 pictures were used (tagged assets) and the distribution of how many of each object were tagged. There were 168 pictures originally but 9 pictures were discarded due to being too blurry or taken from extreme perspectives.

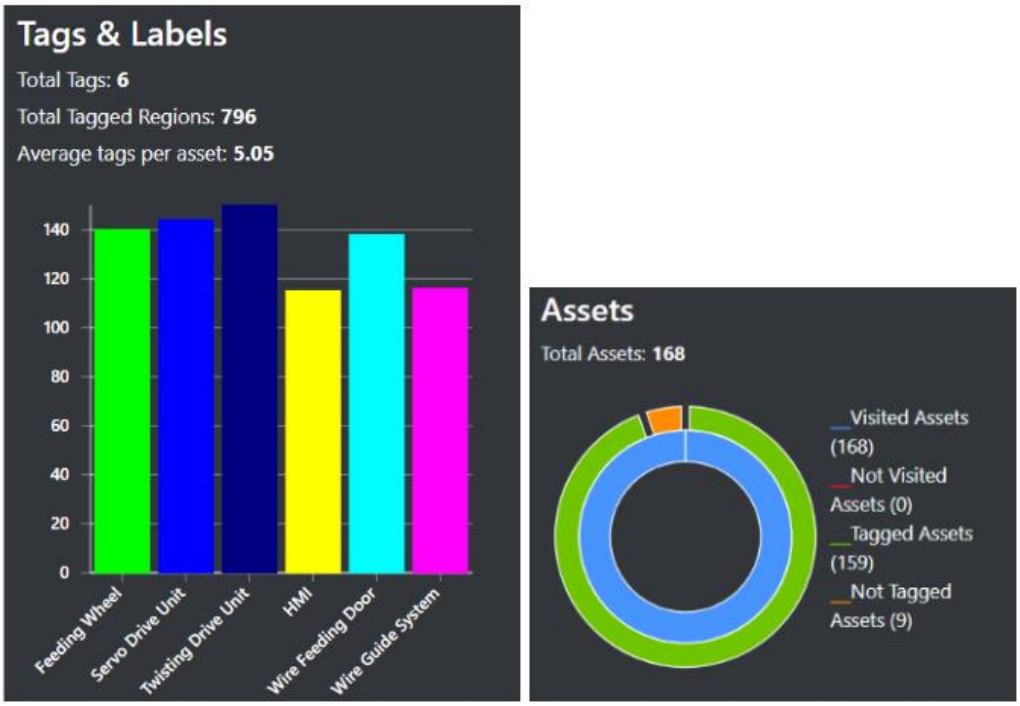

Figure 21. VoTT provides information about the number of tagged assets

VoTT creates JSON files containing region information in the following format:

```
"regions": [
    {
        "id": "HtfgX_VeP",
        "type": "RECTANGLE",
        "tags": [
            "Feeding Wheel"
        ],
        "boundingBox": {
            "height": 772.3636363636364,
            "width": 536.7272727272727,
            "left": 3179.4545454545455,
            "top": 1158.5454545454545
        },
        "points": [
            {
                "x": 3179.4545454545455,
                "y": 1158.5454545454545
            },
            {
                "x": 3716.1818181818185,
                "y": 1158.5454545454545
            },
            {
                "x": 3716.1818181818185,
                "y": 1930.909090909091
            },
            {
```

```
                "x": 3179.4545454545455,
                "y": 1930.909090909091
            }
```


where all four points (x,y) of the bounding region rectangle is specified. These JSON files can be exported to TensorFlow records. The TensorFlow records will then have this region information contained and can be specified as train data to the test system as specified in the pipeline.config in Appendix A. It is also necessary to include a text file (.pbtxt) containing the chosen class labels. This is a simple text file containing following:

```
item {
 id: 1
 name: 'HMI'
}
item {
 id: 2
 name: 'Feeding Wheel'
}
item {
 id: 3
 name: 'Wire Feeding Door'
}
item {
 id: 4
 name: 'Servo Drive Unit'
}
item {
 id: 5
 name: 'Twisting Drive Unit'
}
item {
 id: 6
 name: 'Wire Guide System'
}
```

Care must be taken not to use id: 0 as this is reserved by the TensorFlow library for private class functions.

Another set of pictures are labelled (49 pictures) and exported in the same manner but saved to another folder and used only for evaluation purposes. TensorBoard can be run and pointed to the created model directory and will evaluate in real-time the model performance.

6.7 Fine-tuning checkpoint

Using a pre-trained network is better for time efficiency. This means leaving the lower layers fixed. The lower layers are general features that the neural network has learned and these are often problem independent. It is the higher layers that are problem-specific. That is why it is often these layers that are trained while the more general lower layers can be frozen.

The pre-trained network is downloaded from TensorFlow github page under models[54]. Model name: ssd_mobilenet_v2_coco

Under the pipeline it is possible to point to this model by setting *fine_tune_checkpoint* to where this model is downloaded.

This saves time by having low-level features trained on the COCO data set and making the network skip train weights to recognize these patterns. Instead the later layers are trained and this should be enough for tuning the network to the desired objects.

6.8 Model optimization for ARM devices

The model_graph as produced by the model_main.py binary file will have operations that might not work on ARM devices. In order to prepare the model for deployment on the android device. This is done in two steps:

Freezing the graph. The model needs to be frozen so that all files are saved to one singular file and information about the graph, variables and hyperparameters become constants and unchangeable. All information that is necessary for training, such as gradients are discarded so the model becomes lighter. Tensorflow saves frozen models to .pb file extension. The normal function to freeze a graph can't be used but instead *export_tflite_ssd_graph.py* is employed. This script appends extra post-processing operations that are needed to run in an ARM environment. Practical execution from bash environment:

```
python object_detection/export_tflite_ssd_graph.py \
--pipeline_config_path=/TensorFlow/models/research/object_detec-
tion/models/model/pipeline.config \
--trained_checkpoint_prefix=TensorFlow/models/research/ob-
ject_detection/models/my_model_6000/model.ckpt-6000 \
```

```
--output_directory=TensorFlow/models/research/object_detec-
tion/frozen_graph \
--add_postprocessing_op=true
```

Conversion to TensorFlow Lite FlatBuffer. This is needed in the Android deployment interpreter TFLite code. The model has to be re-shaped with correct operations and made smaller through various use of algorithms. The model becomes compressed but suffers some accuracy loss. This requires the toco converter[55]. Usage as follows:

```
tensorflow/bazel-bin/tensorflow/lite/toco/toco \
--input_file=tflite_graph.pb \
--output_file=frozen_inference_graph.tflite \
--input_shapes=1,300,300,3 \
--input_arrays=normalized_input_image_tensor \
--inference_type=FLOAT \
--mean_values=128 \
--std_values=128 \
--output_arrays='TFLite_Detection_PostProcess','TFLite_Detec-
tion_PostProcess:1','TFLite_Detection_PostProcess:2','TFLite_De-
tection_PostProcess:3' \
--change_concat_input_ranges=false \
--allow_custom_ops
```

Important. Toco does not seem to support Windows as of June 2019. Workarounds exist but comes with varying results. The author did not manage to run above script in Windows, instead this was executed successfully in a Linux environment.

6.9 Implementation on Samsung Galaxy S8

Given the correct model optimization above. It is possible to use the TensorFlow Lite framework for Android to interpret the trained model in real-time. The workflow is illustrated in figure 22.

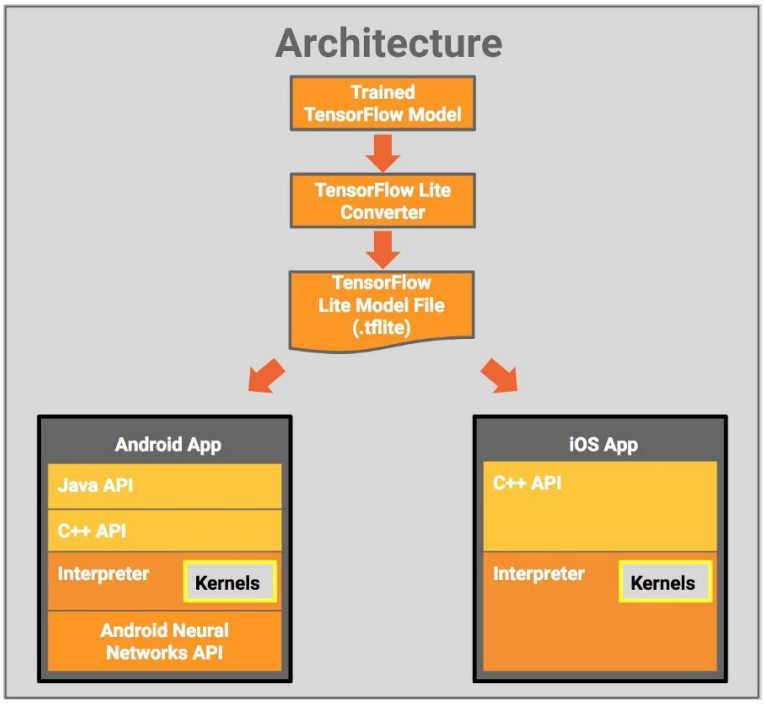

Figure 22. The workflow used in realizing a model deployment on Android[56]

The Android framework uses a custom built interpreter but takes advantage of the Android Neural Networks API[57]. This API is developed by the Android team and offers abstracted layers of functionality that can help create proper normalization, convolutions, pooling and activation functions which make up a neural network. This API can also take advantage of special dedicated neural network hardware. Unfortunately the Samsung Galaxy S8 does not have such capabilities, but the API still provides optimized code to execute CPU operations for these type of devices that lack specialized drivers.

The model deployment on TFLite interpreter for Android is fairly straightforward. It can be found under the official TensorFlow repository: https://github.com/tensorflow by browsing to folders *Examples* → *lite* → *examples* → *object_detection* → *android.*

The example program already contains its own model trained on the COCO dataset. Under *app* → *java* → *tflite* → *DetectorActivity.java* it is necessary to point to the .tflite model. A label text file will also have to be

provided with the chosen class labels (can use the same one exported earlier). The private static final boolean *TF_OD_API_IS_QUANTIZED* need to be set to false. This tells the interpreters that the model uses FLOAT data types. In the same file, the *DESIRED_PREVIEW_SIZE* can be set to whatever is preferred. Here it is set to 640x480 pixels. Remember that the model will still resize the input image tensor to 300x300 pixels regardless of preview size.

7 Results

This chapter shows the results from the trained network in recognizing said objects. The evaluation criteria's chosen in the method section are applied on the trained model and shown below.

7.1 Model performance

Four pictures were extracted with TensorBoard to see how well the model performs for a few evaluation sets. Front-facing pictures (orthogonal to the machine) seem to do well but pictures with more than 30-40 degrees deviation seem to produce mixed results or several blind spots, see Appendix B.

The model training was stopped at 6000 training steps when it was determined that both localization loss and classification loss started to converge and further training did not produce much better performance. Localization loss curve can be seen in figure 23 and approached roughly 0.35 (error unitless). Classification loss was just under 2.0 (error unitless). The training ran for a little bit over 39 hours on our Windows machine in order to reach about 60% precision.

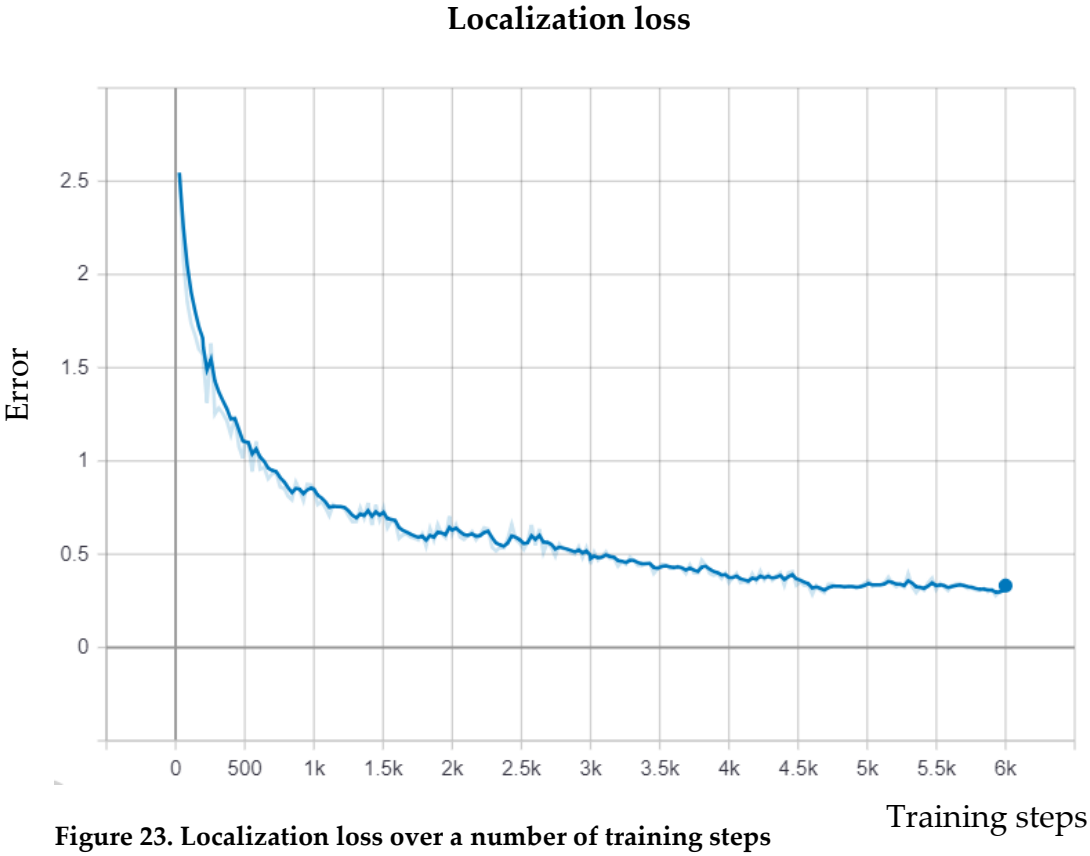

Figure 23. Localization loss over a number of training steps

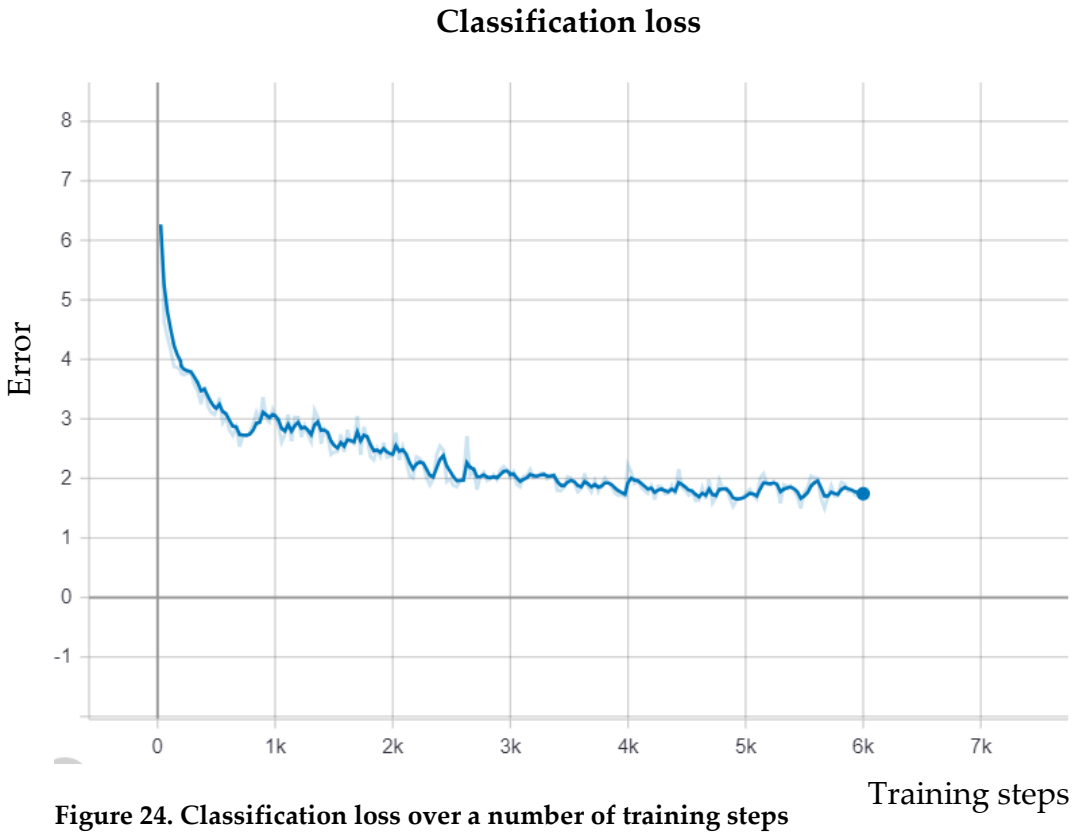

Figure 24. Classification loss over a number of training steps

The mAP for 0.5 IoU reached to be around 85% precision. For 0.75 IoU the system managed to predict roughly 70% correct after training for 6000 steps.

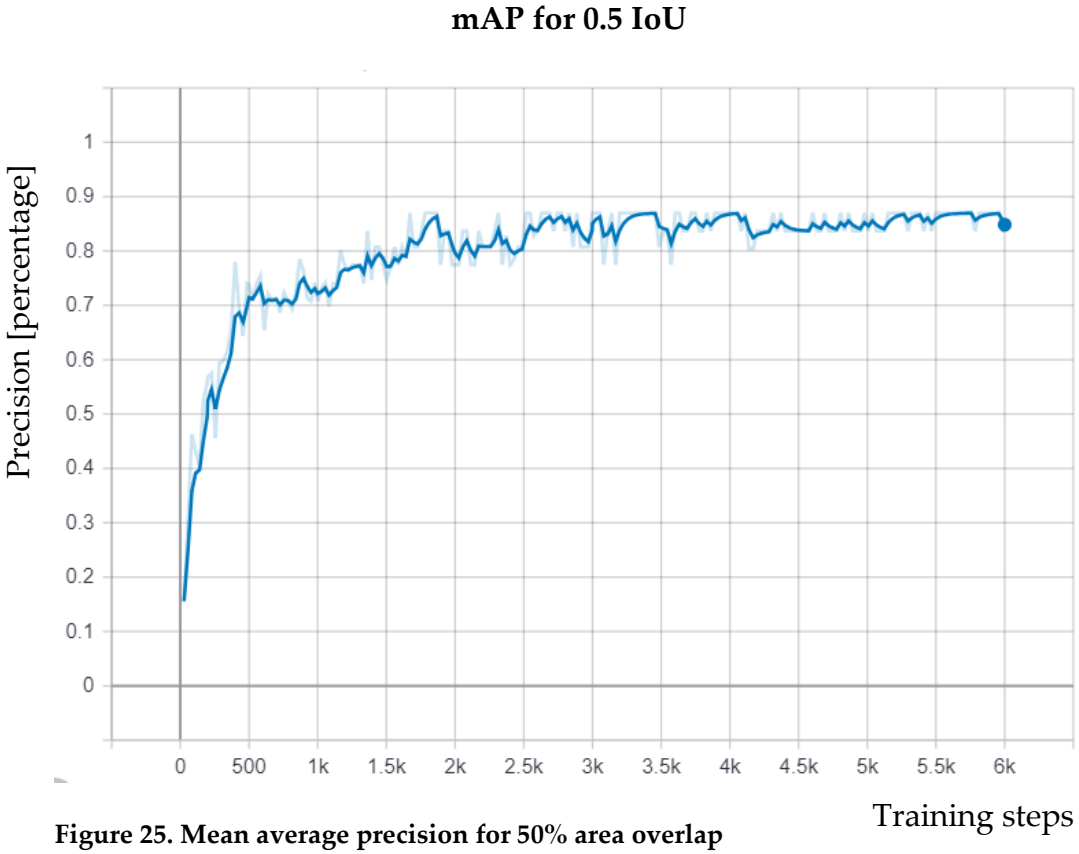

Figure 25. Mean average precision for 50% area overlap

It is expected for the average precision for 0.75 IoU to be considerably smaller. It is harder to predict within 0.75 overlap than for only 0.5 overlap.

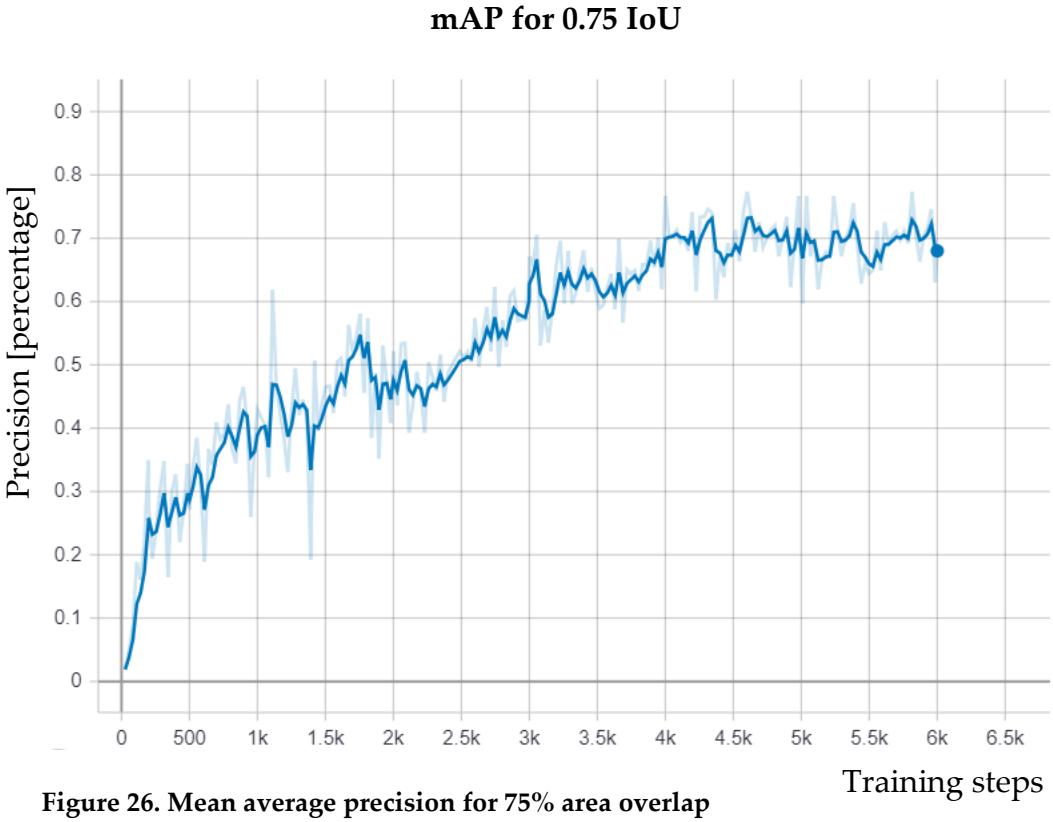

Figure 26. Mean average precision for 75% area overlap

Taking the average over multiple IoU (the minimum being 0.5) it is possible to find a total mean average precision. An IoU of 1.0 is ignored since this is hardly achievable. The total mAP is then given by taking the average mAP from 0.5 IoU to 0.95 IoU with a step size of 0.05 IoU. The result can be seen in figure 27.

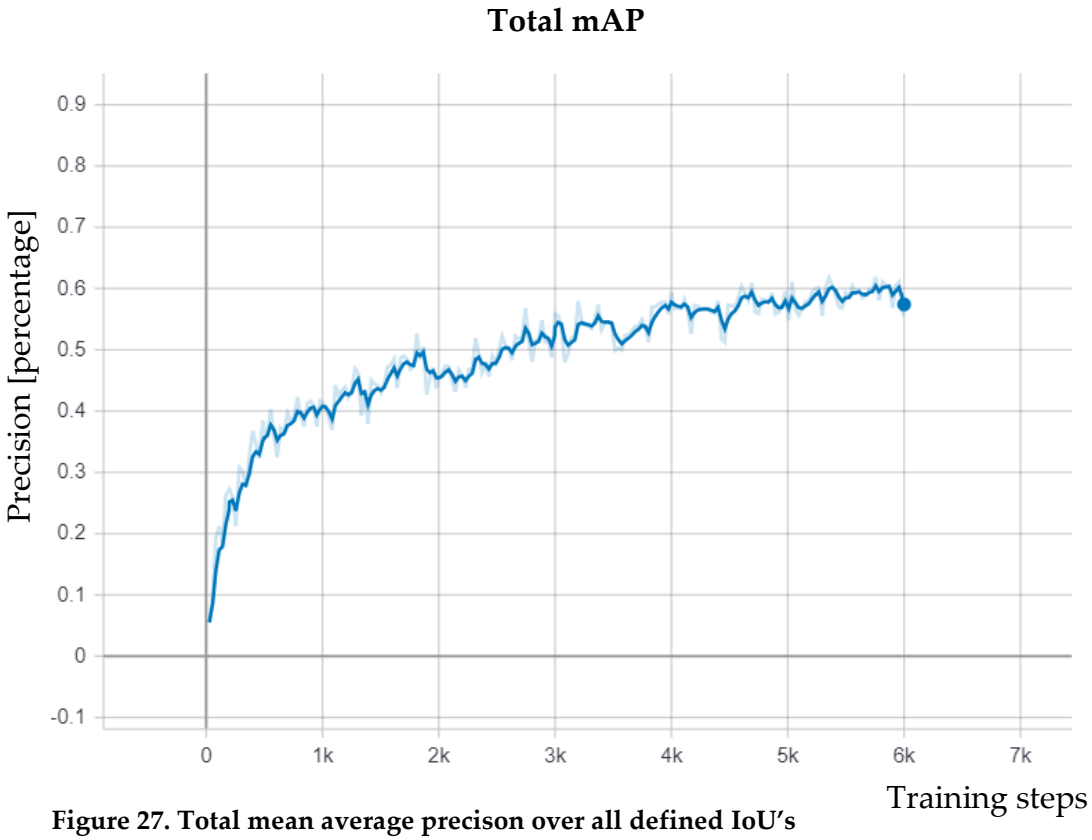

Figure 27. Total mean average precison over all defined IoU's

The total mean mAP as defined above reaches up to barely 60%. This is good considering this is calculated only on the evaluation data and the limited amount of training data that was provided to the system.

The average recall as can be seen in figure 28 is around the 60% mark at the end of the training.

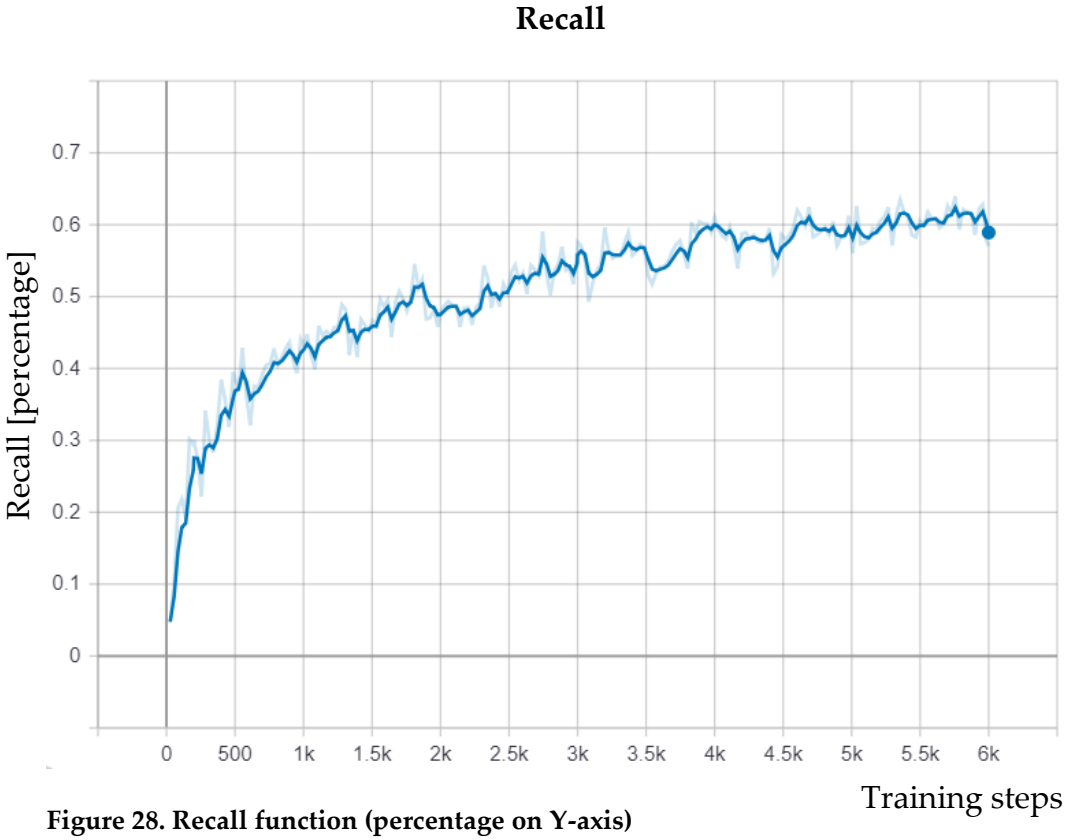

Figure 28. Recall function (percentage on Y-axis)

The results seem to show a system that have learned to generalize and can detect objects on images that it has not previously seen (the evaluation data set).

7.2 Android deployment

The model employed on Android works and is capable of locating objects. The application is started from either the Android Studio or by locating the app once installed on the device. It does not lag and seem to update object boxes quite reliably once an object is detected. The application can be seen in figure 29.

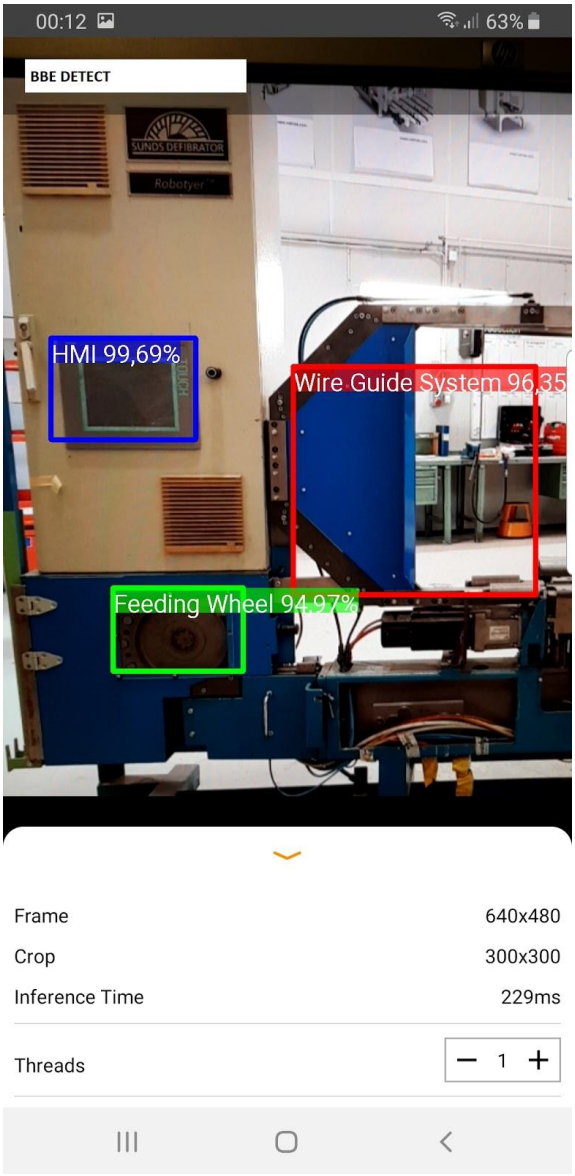

Figure 29. Deployed model shown working on Samsung Galaxy S8

Running the android application for one minute and taking the mean value of inference gives a mean inference time of 213 ms which can be translated to roughly 4.7 FPS performance.

Observation. The android application seem to have a hard time distinguishing objects from afar. In the provided example above it effectively

recognized 3 out of 5 classes in the image. Zooming in on the area of interest, seem to make the model perform better. Going even further back, the model loses the ability to recognize any object except the wire guide system (presumably because of its size).

8 Discussion

Attempts at implementing SSD (with VGG16) on the PASCAL VOC tests seem to generally achieve a 73-80% total mAP[28]. The general methodology seem to be using a dataset of over 5000 images. Having the proposed model in this study achieve a total of 60% mAP on 159 pictures is therefore not entirely bad. The SSD paper runs the training for upwards of 60k steps compared to 6k steps. Notice that their batch size is 8 while 24 is used in this case so the steps might not be directly comparable (along with different feature extractor). Still it gives an idea that to achieve a mAP in the 70% and 80% figures requires long training time, more images and expensive feature extraction networks. This study was not interested in making a breakthrough in precision so naturally these kind of comparisons are of limited value. There is however a requirement of a system that "works" because having a quick model that doesn't detect anything is hardly beneficial to anyone.

The scope of this study was interested in sampling the current technologies as of today and see what is realizable "real-time" wise in order to help Valmet in their decision process of pursuing an object detection solution for their future products. Having the android application perform at roughly 4.7 FPS is actually considered a smooth experience when polling various employers at Valmet. A minor nuisance is the fact that the model requires zooming in the camera phone close to the objects in order to detect them. Ren et al. showcases a 49-59 FPS performance (for input resolutions of 300x300) using a Titan X graphics card optimized with NVIDIA's deep neural network library cuDNN. Comparing to a mid-tier android-powered smartphone device achieving 4.7 FPS is very promising and begs the question how newer models might perform. Even more so, devices equipped with neural network processing units such as the Kirin 970 chip which help perform operations required for deep learning models. Naturally providing the company relevant information about their currently used technology seem useful for the chosen project direction but future improvements in mobile technology seem promising to investigate.

It is quite easy to see why the model performs less than optimal for images that are far away from the camera. The image input from the phone camera (300x300) becomes convoluted to 38x38 in the first layer and this feature map is immediately sent to the detector. Next layer is convoluted to 19x19. Each feature map is sent to the detector so naturally, SSD should pick up on both low-level features and high-level features. What's problematic is that small objects never survive through later layers. If they object in question is already a small part of the 300x300 image, then the convolution to 38x38 will make it miniscule. Theoretically, it should still be detectable because even this feature map is sent to be classified. The issue is that the first feature map (as discussed in the theory chapter) only contains low-level features like edges and contours. These features don't give too much information about the object, so the classifier will struggle to identify it. It is only later layers that provide information about texture and shapes (high-level features). Objects that take up a big portion of the image will therefore survive through all later layers and provide the system with more information and thus become more easily detectable. Naturally a potential "improvement" to this problem would be to scale the picture to higher resolutions, at the cost of computational efficiency.

It can be hypothesized that the problem of non-detection of small objects do not pose a problem to this specific study since the "detection" part is only meant as a tool for visual inspection (in order to receive information). Visual inspections differ from more traditional object detections problems such as counting items and surveillance. In the latter problem classes there is high-value in capturing every object and a big disadvantage in missing objects. For visual inspections as is the case of the operator trying to find the name for a specific machine detail; the person in question will actively point at the detail in question. Zooming in on the object of interest and directing his camera to it in order to gain information. There is therefore less demand on capturing every object in the image and more lax requirements. One could argue that in this use case providing a "smoother experience" in terms of FPS might favor the intended use case more than a system that accurately captures objects far away from the camera and the operator.

The training data provided is good in the sense that it was taken from different heights, perspectives and distances. The data might seem a little scarce and having acquired more training data might have improved the system generalize even more. This became apparent (in Appendix B) when the system tried to classify machine details from perspectives that were further away than just a few degrees from being orthogonal to the machine. The training data could also strip the RGB channels to reduce computational cost. It was left as a help to the system since SSD systems lack fully connected layers and any extra features can prove quite useful. The machine having this particular color as standard makes this not an inherently bad choice. Still it could be attempted. Many different configurations were unfortunately left out due to not having access to proper hardware and due to any small changes requiring more than one day of training. This forced the study to work on a high-level. Taking shortcuts, generalize and apply deductive reasoning in choosing models and configurations as much as possible.

The requirement of an end-to-end system as defined in the problem statement required an extensive use of frameworks, open-source tools and high-level API's in order to prove that constructing object detection from CNN approaches can be streamlined into a workflow that can be utilized by anyone once constructed; given little training. This study shows that there exist frameworks that make this a possibility. TensorFlow was used in this particular implementation but many others exist and are being actively developed. TensorFlow proved to be quite useful and it is a big benefit having visualization tools readily available to analyze the network during training. The large community behind TensorFlow proved useful due to the various pre-trained models and tools that could be employed. It can be hypothesized that it would take little effort to document a strategy and generalize the process of creating an object detection on the mobile phone to any employer on Valmet. The process would just consist of taking a set of said pictures, label them, export them to proper format, use the implemented pipeline that interfaces automatically to the chosen TensorFlow functions to train the model. This means that it would be relatively easy to for anyone to re-purpose the model to work on any machine and machine details.

Further improvements to the model in the realm of easily accessible augmentations might be a quantization of the model tensors. TensorFlow provides tools to train a model using only 8-bit tensors instead of the 32 bits required to represent float-typed weights in the network. It is possible to quantize post-training but that might introduce quite a lot of accuracy loss which can't be afforded for this model.

Returning to the FPS discussion. It is hard to define a smooth mobile experience because the available articles and papers discuss only video frames. The video frames used in the android application is 30 FPS at the minimum. This is very smooth, so the user would not experience any hiccups or "lag" as might be assumed when 4.7 FPS is mentioned. This number merely refers to the how often the image updated with new object detections. The mobile application implemented effectively updates the video stream with new object detections almost 5 times per second. This can be seen as very good in terms of the requirements in the earlier proposed visual inspection tool. In the android example code provided there are also "object tracking" algorithms implemented that add to this smoothness. These algorithms attempt to follow the object between detections as to not disrupt an already existing box on the user's screen.

Returning to the high-level problem statement. This implementation was meant to serve as backend for a mobile application that could be interactive. For example, it would not take much effort to program functions in the code to enable the user to "tap" on any detected box and make a new window appear describing the object in question. This window could provide the user with manuals or instructions about interfacing with the said machine detail. It is easy to see why this technology could provide companies with added value in terms of creating new innovative products.

8.1 Social aspects

The ideas presented in this study regarding the use of CNN networks to detect objects promises great benefits for all societies regarding helpful technology to aid people's lives. The results presented in this report can

easily be extended to helping visually impaired people in detecting objects in their environment. This could be extended even further and might allow for these kind of people to be employed in workplaces previously inaccessible with their kind of disability. Making object detection even less computationally intensive, along with improvements in embedded computing might make object detection realizable as part of augmented glasses. Glasses that can be worn by people all day and help them recognize various materials and objects. Integrating information about objects that could help mentally or visually impaired people to easier navigate their environment.

8.2 Ethical aspects

Object detection and image recognition have the disadvantage of being able to extend into unethical uses. One such unethical use would be for an employer to implement facial recognition systems and installing these in various places throughout the workplace. These could be used for tracking employees (even uniquely so because CNN are capable of learning to recognize individuals) and using this information for various reasons. Tracking the time an individual spends leisuring around, taking coffee breaks, visiting the toilet maybe even being distracted and not working. This information can be used by companies to punish individuals who take longer breaks and rank personnel in terms of efficiency; in a dystopian play of thought. CNN's are powerful and can be used for anything. It is also quite easily to circumvent camera laws while doing so. It is possible to set up a system that doesn't capture video in a "traditional sense". A CNN convolutes the image and makes it unrecognizable but can still identify the person and even his activity. All of this without even having an image physically stored anywhere. So, it can be argued by virtue of loopholes that this technology doesn't fall under camera laws, effectively abusing the technology for ill behavior.

This was only one example, but unethical use cases for this technology might extend further than what is known today because of how fast this technology improves. It is important to regulate the use of this technology through discussions and laws set in place to prevent unethical use cases.

9 Conclusions

In the scope of this report an attempt has been made to investigate the possibilities of creating a real-time visual-feedback system on ARM architecture meant to serve as an aid to operators serving and operating machines. To achieve this, various CNN architectures have been explored for their suitability in creating a light object detection model that could possibly be implemented on performance-constrained hardware. Recent developments in object detection models based on convolutional neural networks show promising attributes for realizing this type of solution at reduced computational cost; making them viable candidates to be considered for such a system. Two of the most notable candidates, YOLO and SSD ditch computationally intensive region proposals in favor of performing predictions (both classification and localization) directly from the feature maps; effectively achieving object detection with one pass through the network. Reviewing recent literature showcases SSD as involving convolutions and operations that involve less processing time. Changing the original base network in SSD to use the lightweight MobileNets feature extractor further amplifies this cost reduction. It was therefore of interest to apply an SSD MobileNets architecture to evaluate the idea of creating a real-time object detecting system customized for recognizing chosen machine details at Valmet. The emphasis being on achievable real-time performance and the possibility of creating end-to-end systems that scale well with Valmet's diverse product portfolio.

The chosen model was trained with 159 training images and 49 evaluation images and produced a total mean average precision of 60% for the finalized model. This model was post-processed by TensorFlow tools in order to export it to the ARM-friendly TFLite format. The model was then encapsulated in an Android framework provided by TensorFlow which takes advantage of Google's neural networks API. The model performed well and gives an average of 4.7 FPS performance on the chosen Android device model Samsung Galaxy S8. The device was first released in June 2016 (almost precisely three years ago) and is ranked 114th on performance by the PassMark Android evaluation toolkit. To put this into perspective, the paper detailing the chosen feature extractor MobileNets was not even published at the S8 date of release. Improvements in smartphone

processing power has generally always been on an uptrend so it can be concluded that even better performance gain can be realized with newer smartphone devices. This sentiment becomes readily apparent when taking into account recent breakthroughs by chipsets like Kirin 970 which include a Neural Network Processing Unit to further enhance the processing of neural networks which this object detection model depends on.

Even though the model achieved a mean total precision of 60% when tested on the evaluation data set it was immediately obvious once deployed that it struggled with detecting machine details from a long distance (when they appear small on the pixel map). This is caused due to the SSD architecture's extensive use of pooling operations to reduce computational effort. However, for this particular use case, this isn't much of an issue. An operator trying to recognize a machine detail using this software would naturally be focusing the camera on the specific object. There exist no clear need to detect objects from a long distance. It's much more important for the model to be fast and easy to use. The performance to accuracy trade-off employed by the chosen architecture is therefore

Based on the results, the report concludes that object detection on a mobile device has shown all the necessary requirements to be deemed feasible to be used as a visual aid for operating and maintenance purposes. The use of appropriate lightweight single-shot detectors along with high-level APIs shows promising results for creating a generalized and scalable workflow that can be extended into recognizing any machine or equipment. The promising performance along with the end-to-end process required to realize this solution, fulfills the requirements set out by Valmet in creating future commercial solutions based on this technology. The increasingly high-performance mobile devices being developed along with the rapid development and refinements of object detection models almost guarantees the long-term potential of said system. The final sentiment being that; the study findings support the decision for Valmet in pursing object detection technologies to be incorporated in their already existing product offerings.

9.1 Future work

Given more time it would be interesting to explore changing base networks on the SSD detector to use Inception and VGG16. These could be compared to each other and MobileNets in order to find an optimal accuracy/speed trade-off for a given application. It would be useful to know what changes cause the most trade-off in any given direction and plot the results against various hyperparameters. This could help interested parties to strike a good balance between speed and accuracy. The comparison could be extended to testing across the various meta-architectures, YOLO and Faster R-CNN.

An interesting approach to object detection on Android would be to research what operations are most computationally intensive on the ARM architecture in order to shape neural networks to a more realizable real-time performance on smartphones by using only operations that are handled efficiently. Another perspective to consider: is the use specialized circuitry (mentioned in earlier chapters) that some phone companies have started developing. The performance gains of such systems can prove to be massive and are worth exploring.

Further, using a ToF sensor which exist in for example newer phone models like Samsung S10 could help remove objects which are further away than the intended use case and reduce computational cost by providing the system with less redundant information. This could help make an implementation on the smartphone run quicker and achieve higher FPS values.

Smartphone and various mobile devices are easily accessible and adopted massively across generations. This extensive adoption means that mobile object detection, given the right realizable application value to people's everyday lives; might be the next thing that revolutionizes humanity's way of living. Further research should be done exploring the various use cases for mobile object detection and how it can enrich society and the people living in it.

References

[1] 'Google Trends', *Google Trends*. [Online]. Available: https://trends.google.com/trends/explore?date=all&q=deep%20learning. [Accessed: 10-May-2019].

[2] T. J. Sejnowski, '1 The Rise of Machine Learning', in *The Deep Learning Revolution*, MITP, 2018, pp. 3–26.

[3] P. K. Bhaskar and S. Yong, 'Image processing based vehicle detection and tracking method', in *2014 International Conference on Computer and Information Sciences (ICCOINS)*, 2014, pp. 1–5.

[4] A. Raghunandan, Mohana, P. Raghav, and H. V. R. Aradhya, 'Object Detection Algorithms for Video Surveillance Applications', in *2018 International Conference on Communication and Signal Processing (ICCSP)*, 2018, pp. 0563–0568.

[5] B. J. Erickson, P. Korfiatis, Z. Akkus, and T. L. Kline, 'Machine Learning for Medical Imaging', *RadioGraphics*, vol. 37, no. 2, pp. 505–515, Feb. 2017.

[6] S. Wu and L. Zhang, 'Using Popular Object Detection Methods for Real Time Forest Fire Detection', in *2018 11th International Symposium on Computational Intelligence and Design (ISCID)*, 2018, vol. 01, pp. 280–284.

[7] J. Zhang, J. Xiao, C. Zhou, and C. Peng, 'A multi-class pedestrian detection network for distorted pedestrians', in *2018 13th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, 2018, pp. 1079–1083.

[8] A. M. Sabu and A. S. Das, 'A Survey on various Optical Character Recognition Techniques', in *2018 Conference on Emerging Devices and Smart Systems (ICEDSS)*, 2018, pp. 152–155.

[9] K. Kim, S. Kang, J. Kim, J. Lee, J. Kim, and J. Kim, 'Multiple objects recognition for industrial robot applications', in *2013 10th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, 2013, pp. 257–259.

[10] S. Gupta, 'How Visual Object Detection can Transform Manufacturing Industries', *Hacker Noon*, 28-May-2018. [Online]. Available: https://hackernoon.com/how-visual-object-detection-can-transform-manufacturing-industries-8b6698cc0a47. [Accessed: 19-May-2019].

[11] C. M. Bishop and P. of N. C. C. M. Bishop, *Neural Networks for Pattern Recognition*. Clarendon Press, 1995.

[12] X. Glorot, A. Bordes, and Y. Bengio, 'Deep Sparse Rectifier Neural Networks', p. 9.

[13] T. J. Sejnowski, '9 Convolutional Learning', in *The Deep Learning Revolution*, MITP, 2018, pp. 127–142.

[14] 'Convolutional neural networks - Learn Unity ML-Agents - Fundamentals of Unity Machine Learning [Book]'. [Online]. Available: https://www.oreilly.com/library/view/learn-unity-ml-agents/9781789138139/16671cc8-0aff-433a-878c-7430be8b9aa1.xhtml. [Accessed: 12-Jun-2019].

[15] K. Sarkar, 'ReLU : Not a Differentiable Function: Why used in Gradient Based Optimization?', *Medium*, 31-May-2018. .

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, 'ImageNet Classification with Deep Convolutional Neural Networks', in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.

[17] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, Massachusetts: The MIT Press, 2016.

[18] 'CS231n Convolutional Neural Networks for Visual Recognition'. [Online]. Available: http://cs231n.github.io/convolutional-networks/. [Accessed: 01-Jun-2019].

[19] K. Fukushima, 'Neocognitron: A hierarchical neural network capable of visual pattern recognition', *Neural Netw.*, vol. 1, pp. 119–130, Jan. 1988.

[20] R. Girshick, J. Donahue, T. Darrell, and J. Malik, 'Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation', in *2014 IEEE Conference on Computer Vision and Pattern Recognition*, Columbus, OH, USA, 2014, pp. 580–587.

[21] C. L. Zitnick and P. Dollár, 'Edge Boxes: Locating Object Proposals from Edges', in *Computer Vision – ECCV 2014*, 2014, pp. 391–405.
[22] J. Hosang, R. Benenson, and B. Schiele, 'How good are detection proposals, really?', *ArXiv14066962 Cs*, Jun. 2014.
[23] P. F. Felzenszwalb and D. P. Huttenlocher, 'Efficient Graph-Based Image Segmentation', *Int. J. Comput. Vis.*, vol. 59, no. 2, pp. 167–181, Sep. 2004.
[24] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders, 'Selective Search for Object Recognition', *Int. J. Comput. Vis.*, vol. 104, no. 2, pp. 154–171, Sep. 2013.
[25] R. Girshick, 'Fast R-CNN', *ArXiv150408083 Cs*, Apr. 2015.
[26] P. Sharma, 'A Practical Implementation of the Faster R-CNN Algorithm for Object Detection (Part 2 — with…', *Medium*, 04-Nov-2018. .
[27] S. Ren, K. He, R. Girshick, and J. Sun, 'Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks', *ArXiv150601497 Cs*, Jun. 2015.
[28] W. Liu *et al.*, 'SSD: Single Shot MultiBox Detector', *ArXiv151202325 Cs*, vol. 9905, pp. 21–37, 2016.
[29] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, 'You Only Look Once: Unified, Real-Time Object Detection', *ArXiv150602640 Cs*, Jun. 2015.
[30] J. Redmon and A. Farhadi, 'YOLO9000: Better, Faster, Stronger', *ArXiv161208242 Cs*, Dec. 2016.
[31] J. Redmon and A. Farhadi, 'YOLOv3: An Incremental Improvement', *ArXiv180402767 Cs*, Apr. 2018.
[32] J. Hui, 'What do we learn from single shot object detectors (SSD, YOLOv3), FPN & Focal loss (RetinaNet)?', *Medium*, 28-Mar-2018. .
[33] K. Simonyan and A. Zisserman, 'Very Deep Convolutional Networks for Large-Scale Image Recognition', *ArXiv14091556 Cs*, Sep. 2014.
[34] A. G. Howard *et al.*, 'MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications', *ArXiv170404861 Cs*, Apr. 2017.

[35] C.-F. Wang, 'A Basic Introduction to Separable Convolutions', *Towards Data Science*, 14-Aug-2018. [Online]. Available: https://towardsdatascience.com/a-basic-introduction-to-separable-convolutions-b99ec3102728. [Accessed: 12-Jun-2019].

[36] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, 'Rethinking the Inception Architecture for Computer Vision', *ArXiv151200567 Cs*, Dec. 2015.

[37] Taegyun Jeon, '[PR12] You Only Look Once (YOLO): Unified Real-Time Object Detection', 08:57:51 UTC.

[38] S. Bianco, R. Cadene, L. Celona, and P. Napoletano, 'Benchmark Analysis of Representative Deep Neural Network Architectures', *IEEE Access*, vol. 6, pp. 64270–64277, 2018.

[39] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, 'MobileNetV2: Inverted Residuals and Linear Bottlenecks', *ArXiv180104381 Cs*, Jan. 2018.

[40] 'PyTorch'. [Online]. Available: https://www.pytorch.org. [Accessed: 04-Jun-2019].

[41] 'TensorFlow'. [Online]. Available: https://www.tensorflow.org/. [Accessed: 13-Jun-2019].

[42] 'Home - Keras Documentation'. [Online]. Available: https://keras.io/. [Accessed: 13-Jun-2019].

[43] 'The PASCAL Visual Object Classes Homepage'. [Online]. Available: http://host.robots.ox.ac.uk/pascal/VOC/. [Accessed: 05-Jun-2019].

[44] 'Splitting the Data into Training and Evaluation Data - Amazon Machine Learning'. [Online]. Available: https://docs.aws.amazon.com/machine-learning/latest/dg/splitting-the-data-into-training-and-evaluation-data.html. [Accessed: 13-Jun-2019].

[45] 'PassMark Android Benchmark Charts'. [Online]. Available: https://www.androidbenchmark.net/passmark_chart.html. [Accessed: 04-Jun-2019].

[46] 'Download Android Studio and SDK tools', *Android Developers*. [Online]. Available: https://developer.android.com/studio. [Accessed: 04-Jun-2019].

[47] 'TensorFlow Lite', *TensorFlow*. [Online]. Available: https://www.tensorflow.org/lite. [Accessed: 04-Jun-2019].
[48] 'TensorBoard: Visualizing Learning | TensorFlow Core', *TensorFlow*. [Online]. Available: https://www.tensorflow.org/guide/summaries_and_tensorboard. [Accessed: 13-Jun-2019].
[49] 'Using TFRecords and tf.Example | TensorFlow Core', *TensorFlow*. [Online]. Available: https://www.tensorflow.org/tutorials/load_data/tf_records. [Accessed: 13-Jun-2019].
[50] 'tensorflow', *GitHub*. [Online]. Available: https://github.com/tensorflow. [Accessed: 04-Jun-2019].
[51] 'Tensors | TensorFlow Core', *TensorFlow*. [Online]. Available: https://www.tensorflow.org/guide/tensors. [Accessed: 04-Jun-2019].
[52] 'tf.estimator.Estimator | TensorFlow Core 1.13', *TensorFlow*. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/estimator/Estimator. [Accessed: 04-Jun-2019].
[53] 'tf.estimator.EstimatorSpec | TensorFlow Core 1.13', *TensorFlow*. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/estimator/EstimatorSpec. [Accessed: 04-Jun-2019].
[54] *Models and examples built with TensorFlow. Contribute to tensorflow/models development by creating an account on GitHub*. tensorflow, 2019.
[55] 'tf.lite.TocoConverter | TensorFlow Core 1.13', *TensorFlow*. [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/lite/TocoConverter. [Accessed: 04-Jun-2019].
[56] 'TensorFlow Lite guide | TensorFlow Lite | TensorFlow'. [Online]. Available: https://www.tensorflow.org/lite/guide. [Accessed: 13-Jun-2019].
[57] 'Neural Networks API | Android NDK', *Android Developers*. [Online]. Available: https://developer.android.com/ndk/guides/neuralnetworks. [Accessed: 05-Jun-2019].

Appendix A: Pipeline configuration

```
model {
  ssd {
    num_classes: 6
    image_resizer {
      fixed_shape_resizer {
        height: 300
        width: 300
      }
    }
    feature_extractor {
      type: "ssd_mobilenet_v2"
      depth_multiplier: 1.0
      min_depth: 16
      conv_hyperparams {
        regularizer {
          l2_regularizer {
            weight: 3.99999989895e-05
          }
        }
        initializer {
          truncated_normal_initializer {
            mean: 0.0
            stddev: 0.0299999993294
          }
        }
        activation: RELU_6
        batch_norm {
          decay: 0.999700009823
          center: true
          scale: true
          epsilon: 0.0010000000475
          train: true
        }
      }

      use_depthwise: true
    }
    box_coder {
      faster_rcnn_box_coder {
        y_scale: 10.0
        x_scale: 10.0
        height_scale: 5.0
        width_scale: 5.0
```

```
    }
  }
  matcher {
    argmax_matcher {
      matched_threshold: 0.5
      unmatched_threshold: 0.5
      ignore_thresholds: false
      negatives_lower_than_unmatched: true
      force_match_for_each_row: true
    }
  }
  similarity_calculator {
    iou_similarity {
    }
  }
  box_predictor {
    convolutional_box_predictor {
      conv_hyperparams {
        regularizer {
          l2_regularizer {
            weight: 3.99999989895e-05
          }
        }
        initializer {
          truncated_normal_initializer {
            mean: 0.0
            stddev: 0.0299999993294
          }
        }
        activation: RELU_6
        batch_norm {
          decay: 0.999700009823
          center: true
          scale: true
          epsilon: 0.0010000000475
          train: true
        }
      }
      min_depth: 0
      max_depth: 0
      num_layers_before_predictor: 0
      use_dropout: false
      dropout_keep_probability: 0.800000011921
      kernel_size: 3
      box_code_size: 4
      apply_sigmoid_to_scores: false
    }
  }
  anchor_generator {
```

```
      ssd_anchor_generator {
        num_layers: 6
        min_scale: 0.20000000298
        max_scale: 0.949999988079
        aspect_ratios: 1.0
        aspect_ratios: 2.0
        aspect_ratios: 0.5
        aspect_ratios: 3.0
        aspect_ratios: 0.333299994469
      }
    }
    post_processing {
      batch_non_max_suppression {
        score_threshold: 0.300000011921
        iou_threshold: 0.600000023842
        max_detections_per_class: 100
        max_total_detections: 100
      }
      score_converter: SIGMOID
    }
    normalize_loss_by_num_matches: true
    loss {
      localization_loss {
        weighted_smooth_l1 {
        }
      }
      classification_loss {
        weighted_sigmoid {
        }
      }
      hard_example_miner {
        num_hard_examples: 3000
        iou_threshold: 0.990000009537
        loss_type: CLASSIFICATION
        max_negatives_per_positive: 3
        min_negatives_per_image: 3
      }
      classification_weight: 1.0
      localization_weight: 1.0
    }
  }
}
train_config {
  batch_size: 24
  data_augmentation_options {
    random_horizontal_flip {
    }
  }
  data_augmentation_options {
```

```
    ssd_random_crop {
    }
  }
  optimizer {
    rms_prop_optimizer {
      learning_rate {
        exponential_decay_learning_rate {
          initial_learning_rate: 0.00400000018999
          decay_steps: 800720
          decay_factor: 0.949999988079
        }
      }
      momentum_optimizer_value: 0.899999976158
      decay: 0.899999976158
      epsilon: 1.0
    }
  }
  fine_tune_checkpoint: "models/model/model.ckpt"
  num_steps: 6000
  fine_tune_checkpoint_type: "detection"
}
train_input_reader {
  label_map_path: "data/tf_label_map.pbtxt"
  tf_record_input_reader {
    input_path: "data/a_train.record*"
  }
}
eval_config {
  num_examples: 49
  max_evals: 10
  use_moving_averages: false
}
eval_input_reader {
  label_map_path: "data/tf_label_map.pbtxt"
  shuffle: false
  num_readers: 1
  tf_record_input_reader {
    input_path: "data/a_eval.record*"
  }
}
```

Appendix B: Image evaluation

Left picture is attempted object recognition by trained model with 6000 steps on target image.

Right picture is target image with overlaid ground-truth boxes.

#1 Detects all details

#2 Detects all details

#3 Struggles to detect all details

#4 Detects all details