BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF CONTROL AND INSTRUMENTATION

ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

DEEP LEARNING ALGORITHMS ON EMBEDDED DEVICES

ALGORITMY HLUBOKÉHO UČENÍ NA EMBEDDED PLATFORMĚ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. Jaroslav Hadzima**
AUTOR PRÁCE

SUPERVISOR — **Ing. Karel Horák, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2019

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Master's Thesis

Master's study field **Cybernetics, Control and Measurements**
Department of Control and Instrumentation

Student: Bc. Jaroslav Hadzima ***ID:*** 173648
Year of study: 2 ***Academic year:*** 2018/19

TITLE OF THESIS:

Deep Learning Algorithms on Embedded Devices

INSTRUCTION:

Computer vision tasks using deep learning algorithms usually require long computing time in training phase, whereas testing phase intended for classification of an unknown objects is very quick. For these reasons, it is complicated to implement deep learning algorithm on a power limited embedded device. Objectives:

1. to study deep learning state-of-the-art
2. to study implementation approaches using various embedded devices
3. to make suitable dataset including training and testing parts
4. to make a fully functional implementation of selected task on given device
5. to evaluate implementation's properties

REFERENCE:

1. Goodfellow I., Bengio Y., Courville A.: Deep Learning. MIT Press, 2016. ISBN 9780262035613. (deeplearningbook.org)
2. Buduma, N., Locascio, N. Fundamentals of Deep Learning. O'Reilly Media, Inc. 2017. ISBN 9781491925614.

Assigment deadline: 4. 2. 2019 ***Submission deadline:*** 13. 5. 2019

Head of thesis: Ing. Karel Horák, Ph.D.

doc. Ing. Václav Jirsík, CSc.
Subject Council chairman

Abstract

This paper describes currently widely used Deep Learning architectures and methods for object detection and classification in video, with intention of using them on embedded systems. We will cover steps and reasoning when choosing the most appropriate embedded hardware for our application. Our test application consists of vehicle detection and free parking space detection using Deep learning methods, all wrapped under name *Smart car park*. This application provides monitoring of vehicle presence in car park and if they occupy parking spot or not. All this is expected to be done using embedded device. Later, there will be covered configuration steps for our embedded device with emphasis on hardware optimization for speed. We will provide comparison of available inference models, which will be rated mostly in categories like speed or F1 score, which have the biggest impact in our application. The best candidate will be selected and used for testing of our application.

Keywords

Machine Learning, Deep Learning, Embedded systems, Raspberry Pi 3 Model B, Object detection, Car detection, Smart car park

Abstrakt

Táto práca popisuje v súčastnosti široko používané architektúry a modely pre Hlboké Učenie, riešiace úlohu detekcie a klasifikácie objektov vo videu. Dôraz tu bude kladený na ich použiteľnosť na vstavaných zariadeniach. Postupne preberieme kroky a odvôvodňovanie pri výbere najlepšieho vstavaného systému pre našu aplikáciu. Ukážková aplikáci pozostáva hlavne z detekcie vozidiel a detekcie voľných parkovacích miest s využitím algoritmov Hlbokého Učenia. Táto aplikácia umožňuje monitorovať počet vozidiel, nachádzajúcich sa na parkovisku a zároveň rozhodnúť, či sa nachádzajú na prakovacom mieste alebo nie. Následne tu budú prebrané kroky nutné ku konfigurácii zariadenia s dôrazom na optimalizáciu hardvéru pre dosiahnutie čo najväčšej rýchlosti. V ďaľšej časti bude poskytnuté porovnanie vybraných modelov, ktoré budú porovnávané hlavne v kategóriách ako rýchlosť alebo F1 skóre. Najlepší kandidát bude použitý na riešenie našej aplikácie a následné testovanie jej vlastností s názvom *Inteligentné parkovisko.*

Kľúčové slová

Strojové učenie, Hlboké učenie, Vstavané systémy, Raspberry Pi 3 Model B, Detekcia objektov, Detekcia vozidiel, Inteligentné parkovisko

Rozšírený abstrakt

Úvod

Systémy strojového učenia nachádzajú čoraz väčšie a väčšie uplatnenie v širokom rozsahu odborov. Jedným z týchto odborov je práve aj spracovanie obrazu, ktoré bude náplňou tejto práce. Algoritmy Hlbokého učenia sú špeciálnym typom Strojového učenia, kde sú algoritmy inšpirované štruktúrou a funkciou mozgu. Sú charakteristické schopnosťou učiť sa podobne ako človek.

V tejto práci využívame práve algoritmy Hlbokého učenia pre riešenie problému detekcie vozidiel na parkovisku, pojatou vytvorením aplikácie Inteligentného parkoviska. Táto aplikácia musí dokázať správne lokalizovať vozidlá na parkovisku a mala by byť schopná správne vyhodnotiť obsadenosti parkovacích miest.

Riešenie pri tom bude aplikované na vstavaný system. To predstavuje výzvu, pretože vstavané systémy nevynikajú výpočetnou silou. Ani zďaleka sa nemôžu porovnávať so systémami využívajúcimi počítač s výkonnou grafickou kartou alebo dokonca komerčnými cloudovými riešeniami, využívajúcimi desiatky prepojených zariadení. Vstavané zariadenia väčšinou nevynikajú parametrami ako je pamäť a hlavne RAM. Modely pre Hlboké učenie môžu byť celkom rozsiahle. Môžu mať veľké množstvo vrstiev a každá vrstva zvyšuje pamäťové nároky. Musíme teda vhodne voliť zariadenie, na ktorom budeme našu aplikáciu testovať, aby bolo schopné nahrať celý model do RAM.

Popis riešenia

Ako bolo spomenuté vyššie, našou vybranou aplikáciou je práve Inteligentné parkovisko. To znamená, že očakávame jednu alebo malé množstvo statických kamier umiestnených vo vyšších častiach budovy, pozorujúcich plochu parkoviska. Prvý problém, ktorý sme museli vyriešiť, bol výber vhodného embedded resp. vstavaného zariadenia. Na trhu sa v súčastnosti nachádza veľké množstvo zariadení, no nie všetky sú vhodné na náš typ úlohy. Niektoré sú zbytočne príliš výkonné a predstavovali by iba finančnú stratu. Iné nemajú dostatočné parametre, ktoré by boli výrazne obmedzujúcim faktorom. V tejto práci sme diskutovali použitie spolu 5 hlavných vstavaných systémov s rôznymi parametrami alebo ich kombinácie. Vyberali sme hlavne riešenia formou CPU/GPU/TPU. Výsledným vybraným kandidátom sa stalo Raspberry Pi 3 Model B, pretože obsahuje najlepší pomer výkonu a ceny a má vhodné parametre pre našu aplikáciu.

V práci sme postupne demonštrovali konfiguráciu zariadenia. Následne sme ukázali proces učenia vybraných modelov Hlbokého učenia. Trénovanie prebiehalo na počítači s GPU, pretože trénovanie na vstavanom zariadení je zatiaľ príliš časovo náročné. Následne sme vyhodnotili vlastnosti jednotlivých predom vybratých modelov. V našej aplikácii sú dôležitými faktormi hlavne rýchlosť a kvalita spracovania. Model s najlepšími vlastnosťami bol práve *ssd_mobilenet_v1_coco,* ktorý dosahuje uspokojivých 1.58 FPS a má F1 skóre 0.72. Nejedná sa však o najrýchlejší alebo najpresnejší model. Niektoré modely boli rýchlejšie, avšak ich presnosť nebola dostatočná, nakoľko mali vysoké množstvo falošných detekcií. Na druhej strane, niektoré modely mali veľmi vysokú presnosť, no ich rýchlosť bola veľmi malá.

Pred tým, ako sme začali riešiť samotnú aplikáciu, pozreli sme sa na možnosti hardvérovej optimalizácie pre dosiahnutie čo najlepšieho výkonu. Úspešne sa nám podarilo skrátiť dobu nahrávania modelu do pamäte RAM a zároveň aj čas inferencie. S využitím vybraného modelu na optimalizovanom zariadení sme začali riešiť samotnú aplikáciu. Táto aplikácia musí byť schopná čo najpresnejšie detekovať vozidlá na parkovisku a vyhodnocovať štatistiky. Keďže sme pozorovali značné množstvo falošných detekcií, zakomponovali sme korekciu maskou, vyhraničujúcou plochu samotného parkoviska. Túto masku sme získali metódami počítačového videnia, konkretnejšie rozdielovými snímkovými metódami v kombinácii so znalosťou o polohe objektu z objektového detektoru. Následne sme museli nájsť konkrétne parkovacie miesta, aby sme mohli vôbec vyhodnocovať ich obsadenosť. Dokázali sme to pomocou štatistických metód, kde sme zaznamenávali najčastejšiu polohu detekovaných objektov v čase vytvorením tzv. Heat mapy. Po dostatočne dlhom zázname sme našli najsilnejšie oblasti a prehlásili ich za parkovacie miesta. Následne sme mohli prejsť k poslednej časti, ktorá pozostávala z vyhodnocovania obsadenosti parkoviska. Riešili sme to porovnávaním prekrytosti oblastí pre parkovacie miesta a aktuálne detekovanými vozidlami.

Výsledky

Táto práca prezentuje porovnanie aktuálne používaných metód v oblasti Hlbokého učenia. Taktiež sa zaoberá problematikou aplikácie týchto metód na vstavané systémy. Porovnávame tu v súčasnosti často využívané hardvérové riešenia a následne prezentujeme celý proces od učenia až po inferenciu na vybranom zariadení, ktoré bolo optimalizované pre vykonávanie danej funkcie. Porovnávame tu viacero vybraných modelov a vyhodnocujeme ich vlastnosti.

Hlavným výsledkom tejto práce je dokázanie, že aj vstavané zariadenia, ktoré neovplývajú výkonom porovnateľným s komerčnými riešeniami sú vhodným kandidátom na problematiku detekcie objektov pomocou metód Hlbokého učenia. Ukázalo sa že existujú modely, ktoré sú schopné dosiahnuť viac ako 8 FPS. Pri zlepšení vlastností samotného modelu pre zníženie počtu falošných detekcií môže byť tento model atraktívnym riešením napríklad v priemysle.

Reference

Printed version:

HADZIMA, Jaroslav. *Algoritmy hlubokého učení na embedded platformě.* Brno, 2019. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/119387. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedoucí práce Karel Horák.

Electronic version:

HADZIMA, Jaroslav. *Algoritmy hlubokého učení na embedded platformě* [online]. Brno, 2019 [cit. 2019-05-09]. Dostupné z: https://www.vutbr.cz/studenti/zav-prace/detail/119387. Diplomová práce. Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav automatizace a měřicí techniky. Vedoucí práce Karel Horák.

Declaration

Hereby I declare that I have written my thesis on Deep Learning Algorithms on Embedded Devices independently, under the supervision of Ing. Karel Horák, Ph.D and using literature and other information sources that are all properly cited and included in the list of references.

Furthermore, as author of this diploma thesis, I declare that in connection with the creation of this thesis I did not infringe the copyrights of third parties, in particular I did not interfere illegally in others copyrights and I am fully aware of the consequences of violation of Section 11 and following of Copyright Act No. 121/2000 Coll., including possible criminal-law consequences arising from the provisions of Part Two, Title VI. Part 4 of the Criminal Law No. 40/2009 Coll.

In Brno on 10th May 2019 .....
Bc. Jaroslav Hadzima

Acknowledgements

I would like to begin by expressing my sincerest gratitude to my thesis supervisor Ing. Karel Horák, Ph.D for igniting my interest in Computer Vision and his assistance in the completion of this thesis. Furthermore, I would like to thank my family, friends and my beautiful girlfriend for their support and perstintant support.

In Brno on 10th May 2019 .....
Bc. Jaroslav Hadzima

Contents

List of tables

List of pictures

INTRODUCTION

We live in time of great technical advances. New methods and systems are being developed every year, which help make peoples life easier and more comfortable. We have learned how to program a machine to do exactly what we wanted them to do. It might be as complex, as programmer's imagination allow. However, these machines are mostly hard-coded. If situation changes only a little bit, these machines would probably be no longer applicable. Need for an intelligent system capable of adapting to changing situations resulted into algorithms, that can learn in similar way students learn from their teacher. No direct programming would be necessary. We would only show what needs to be done, not how and machine would create their own approach.

Deep learning algorithms are subset of Machine learning algorithms. First chapter will briefly cover general information about types and use cases of machine learning algorithms as a whole and acts as basic introduction to Deep learning methods, which are covered in second chapter. There are dozens of different types and applications for Deep learning algorithms so we will be going only over the most well-known.

Third chapter covers general information about embedded devices used for Deep Learning applications. This chapter covers purpose, hardware and software needs for such systems. Different types of Deep Learning algorithms require different hardware and software configurations, but they are all very computationally demanding. Real time computing generally requires more complex and more computationally efficient systems. We will provide short comparison of popular embedded devices commonly used for Deep learning applications.

Fourth chapter discusses reasons and motivation when choosing model application for our tests. We will choose the best embedded device for our model application and describe steps needed for device setup and additional hardware optimization later in fifth chapter.

Sixth chapter describes entire training process of neural network. Second part of this chapter provides benchmark done on our embedded device, evaluating applicable models. Seventh chapter then uses winning model and demonstrates our chosen application. Wi will try to achieve results, sufficient for use in real world applications.

Brief summary is included at the end reviewing achieved results and provides opinions on current and possibly future character of this work.

1 MACHINE LEARNING

Terms like Artificial Intelligence (AI), Machine learning (ML) and Deep learning (DL) are nowadays used interchangeably in most media. For example, when Google DeepMind's AlphaGo program defeated South Korean Master Lee Se-dol in the board game Go in March 2016, all three terms were used to describe how AlphaGo won.[1] They are not the same, although they all overlap in certain areas. The most basic explanation for these terms could be as follows:

Artificial Intelligence - Technique used to create a program that mimics human behaviour. Applied commonly to projects, where designed systems show ability to reason, learn, generalize or find meaning. [2]

Machine Learning - Uses statistical methods which allow machines to improve with experience - to learn. Machine Learning is a subset of Artificial intelligence. A great definition was provided by Tom M. Mitchell in [3]:

> "A computer program is said to learn from experience *E* with respect to some class of tasks *T* and performance measure *P*, if its performance at tasks in *T*, as measured by *P*, improves with experience *E*."

Deep Learning - Is subset of Machine Learning. Deep Learning algorithms are perhaps best exemplified by multi-layer neural networks, which try to make sense from imputed unsorted data based on learnt traits. Uses basic concepts from biology of the brain. Deep Learning is useful when there is quantity of data.

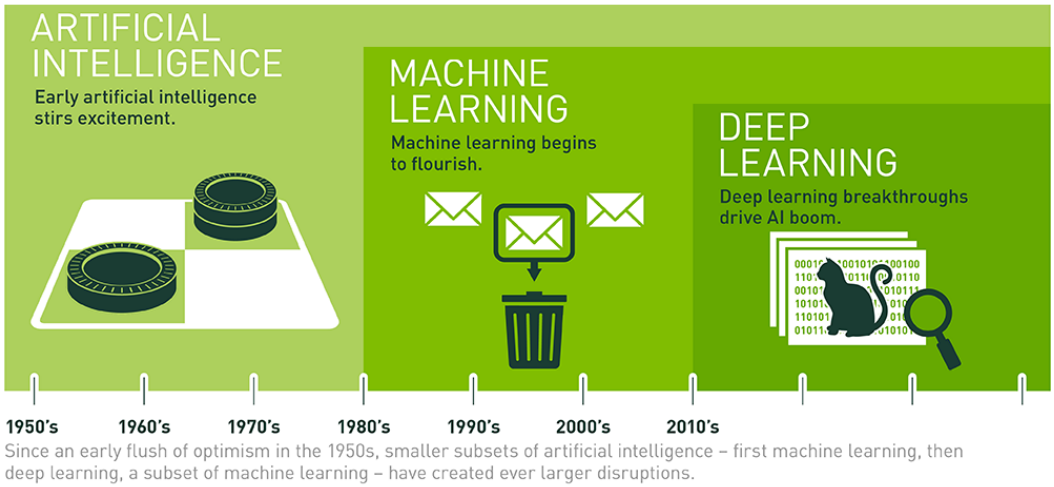

Figure 1: Difference between AI, ML and DL [1]

As we can see, Machine learning is only a small branch of Artificial Intelligence and its purpose is to learn from data using statistical methods. Not everything is programmed. Certain parts could use mechanisms like neural networks, k-Nearest neighbours, SVM or one of many other. Great example, as shown in image above is a spam filter. Easy solution using ML could look similar to this: We would take as many spam and normal messages, as we could find and use them for training, flagging spam messages. Our system may for example find certain words that are common in a spam messages and not in normal conversations. These words will be thus given more weight when determining a spam and non-spam message. Later when a new message comes, our system looks on word composition and based on weights for elemental words computes average score. If average score is over certain threshold, message is automatically deleted or sent to spam folder. Many algorithms and principles could be used for implementation.

1.1 Types of machine learning

When talking about machine learning, we can differentiate between three fundamental types, where each behaves differently, needs different data and could be used in different situations.

1.1.1 Supervised learning

In supervised training an algorithm generates a function that maps inputs to desired output. Dataset is required to contain examples of both inputs and outputs. Its name comes from the fact that the whole process is controlled using provided labels from supervisor. Supervised learning can be divided into two subgroups:

Classification - The most common type of supervised learning. Often, the goal is to get a machine to learn a classification problem that we have created. Machine assigns category to input data. For example, OCR Digit classification is fed with a picture of a number and it classifies or "*label*" it as the right number. [4]

Regression - As name suggest, in this scenario we generalize input data and want to make a statistical prediction estimating future input data. Example could be weather temperature forecasting or stock values prediction. Because this type of model predicts some form of a quantity, its skill has to be reported as error. A simple root mean squared error is typical for this type of error calculation. [4]

1.1.2 Unsupervised learning

This type of learning models has knowledge of inputs but labelled output examples are unavailable. These machines should be able to draw inference from non-labelled data without reference or knowledge of output. By drawing a conclusion, we usually mean discovering underlying structure in data. This is a harder problem, because we expect machines to do something and we don't tell them how.

Two main subcategories are:
Pattern recognition and data clustering - Process of dividing and grouping similar data sample together, thus allowing us to find similarity in data. Can be used later for supervised training.

Reducing data dimensionality - Means decreasing dimensions generated by number of features provided. We may reduce computing constraints caused by lack of computing power by reducing features map dimension count. Reduction of time required for categorization for the next computational processes is an attractive factor.

1.1.3 Reinforcement learning

Presents us with an *Agent*, that learns how to behave in observed world. Every action generates impact on the environment and environment provides feedback, either positive or negative. Based on positiveness of the feedback, *Agent* learns rules of the observed world and how to behave to generate maximally positive feedback. This type of learning is not as common as other two types mentioned before. Typical applications are automatized game playing or robot path navigation.

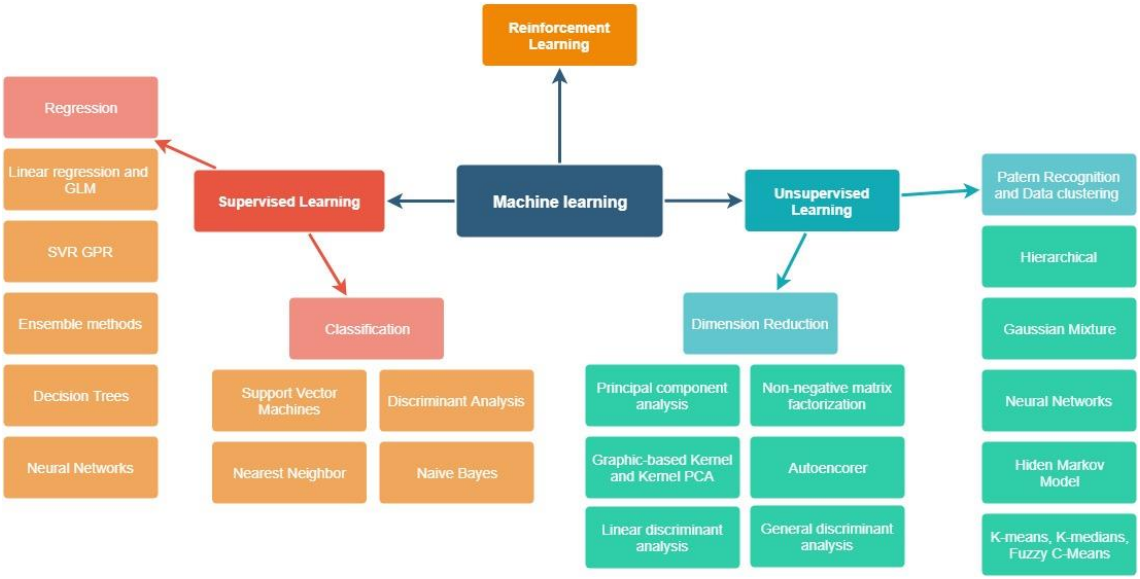

Figure 2: Diagram showing the most well known subgroups of ML algorithms [3,4,5]

2 DEEP LEARNING

Before we jump straight into Deep Learning methods, it is surely useful to define what Deep Learning actually means. There are many high-level descriptions that can be found online. Between them, we can clearly see two key-points that remains in probably every definition [6]:

- Models consist of multiple layers or stages of nonlinear information processing.
- Usage of methods for supervised or unsupervised learning from feature representation at successively higher, more abstract layer.

When writing about Deep Learning, it is common for a literature to actually refer to convolutional neural networks. Definitions don't specify, that it has to be only convolutional neural networks, but they are the most used ones. Similarly, for purposes of this paper we will refer to convolutional neural networks as Deep Learning.

Deep Learnings popularity has been steadily increasing over the last two decades. But why is that? Probably the most significant reasons for this are improvements in the fields of machine learning and signal/images/information processing, increased efficiency and speed of chips (using more capable CPUs and GPUs) but also human curiosity. Process of training Neural network (NN) become undoubtedly many times faster than it was in the past. Training sets could thus become accordingly larger which resulted in increase in NNs precision. These advances allowed DL methods to employ complex compositional nonlinear functions, learn distributed and hierarchical feature representations and make effective use of both labelled and unlabelled data. It has not been that long time ago, that for the first time a real image-based computing using NN could be achieved. It is safe to predict, that with constant increases in technology levels, the usage of DL will be employed in countless applications worldwide and used as an attractive alternative to current common solutions. [6]

2.1 Difference between Machine Learning and Deep Learning

Deep learning is specialized form of ML, subset if you want. With ML you start with data, let's say image of a car. We have to manually chose which features should be extracted using any feature extractor and classifying them with some classifier. Features to be extracted are known beforehand. Vector of given features is then given to the classifier to classify an object. Just to be complete, as a classifier could be used also a NN. The key idea in basic ML algorithms is a separation of steps in the whole procedure. If we don't like output from individual steps, we can tweak or correct parameters of elemental parts.

On the other hand, in DL you feed whole raw data to the NN that does all steps that machine learning system had to separate. Output from DL NN is an classified object. We can't tell, nor influence what features are being extracted and used or how classifier operates when it comes to choosing important features. Networks learn to perform a specific task automatically. One of their great advantage over other ML methods is that they tend to increase their performance with increase of training data even when ML methods already stagnates.

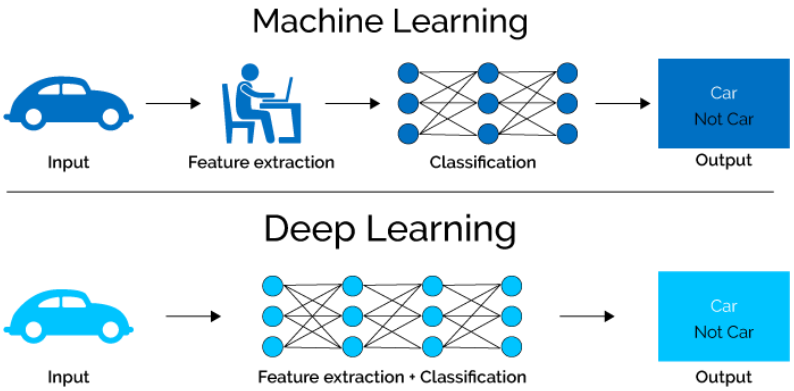

Figure 3: Difference between ML and DL [7]

Another aspect to consider is the speed, their complexity and requirements. If we have small set of data and an easier problem, ML methods will most likely represent a better fit than DL methods. They also require less computational power than DL. Also, if we are able to wisely specify small set of important features, ML algorithms can be relatively fast. If we really want to use DL methods, we have to be aware of burdens they present. We need an extensive amount of data, possibly thousands of pictures, that have to be manually annotated. Also, training a model may take weeks or months for older computers with slow GPU/CPU. Generally having faster GPU/CPU means faster learning so if we don't have access to such devices, we are often left with ML algorithms or using often pricey online cloud learning alternatives.

Examples of DL can be seen everywhere around us. Most of us even came into contact with them in some form or another on everyday basis. Just think about every bigger website like Facebook, YouTube, eBay or Amazon. They all use recommender systems for relevant element propagation. These systems allow retailers to offer personalized recommendations based on your previous purchases or browsing activity. Ads can use them too. This is why if you merely think about something and in the very next page reload it is in the ads, as they were reading your mind. Between other less visible use cases belongs object tracking with applied object detection and classification, prediction of a market trends or risk calculation. Google Maps uses data from network connected smartphones to calculate traffic and suggests the fastest route. Also, Google presents tools for speech recognition.

We will go with Deep Learning algorithms in more detail later in text so just to point out some interesting examples of Deep Learning in image recognition could be seen on sites like Facebook, where it recommends tagging your friend on your photos. Other examples are third party programs with image classification or camera surveillance. Massive corporation giants like Google use DL in their image search recognition. Additionally, Google provides a way to restore or enhance image details using extrapolation and knowledge from thousands similar images we post daily on the internet. Most of us are thankful for those tools but rarely care enough to investigate how these tools work. In practice, DL excels everywhere where identifying patterns in unstructured data is required. Data could represent media such as video, images, sound or other signals, text and sometimes even time series. According to [8], the top 10 use cases for revenue generation incorporating DL are:

1. Static image recognition, classification, and tagging;
2. Machine/vehicular object detection/identification/avoidance;
3. Patient data processing;
4. Algorithmic trading strategy for performance improvement;
5. Converting paperwork into digital data;
6. Medical image analysis;
7. Localization and mapping;
8. Sentiment analysis;
9. Social media publishing and management;
10. Intelligent recruitment and HR systems.
11. As expected, image recognition and automotive use is in the first two places.

2.2 Basics of Deep Learning

When it comes to DL, in general, we are trying to increase probability of correct estimation. If our model is trained on small dataset, so that objects are not recognized at all or even objects of different classes are detected as class members, we are calling that underfitting. On the other hand, if we train our model too much, it becomes overly specific. We call that state overfitting. Somewhere between underfitting and overfitting should be our desired sweet spot.

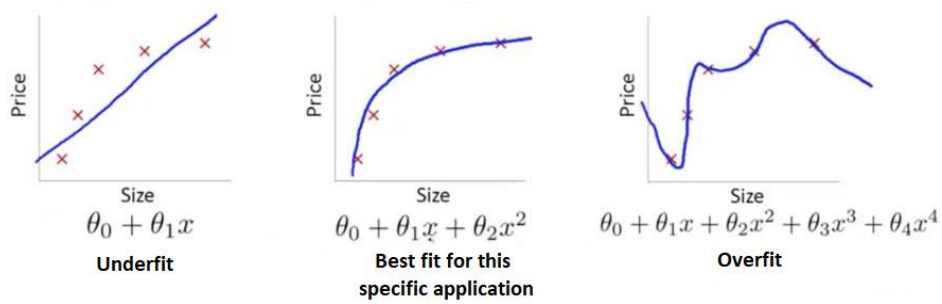

Figure 4: Difference between best-fit, Underfit and Overfit [9]

Testing NNs accuracy is crucial during learning phase. It is a time to stop the learning phase or modify architecture if accuracy start to drop. Speed of the learning process and overall accuracy is determined by many factors such as dataset, type of layers used and activation functions, learning coefficient and many others. Utilizing right layers typically means significant difference in accuracy.

In NN, we can find many different types of layers. It's important to realize that every layer has its own specific purpose. In this paragraph, NN layers and their basic meaning will be discussed. There are three general types of layers in NN based on their location in architecture.

Input layer has passive nodes that only shift input values to the output of its neurons. In essence, acts as a distributor of data for our NN.

Output layer contains active nodes. We can modify, how output from our NN behaves, depending on training outputs from NN. Commonly, they are done as classifiers. If we want only binary classification, describing if an object is present within input data, we have to use only one neuron acting as binary logical element. If we expect classifier to recognize different types of objects, we commonly use amount of output neurons equal to the number of recognizable objects.

Hidden layer is a common name for every layer that is between input and output layer. Hidden layer is hidden from interface of the NN. We know what we put on the input and what outputs mean, but we don't have a deeper understanding why neurons learned to behave certain way and what model it created, therefore these states of neurons are hidden from us in technical sense.

We can see all three types of layers in figure 5, activation functions and their biases are also shown. Next section covers the most common types of hidden layers with their definitions.

2.2.1 Common types of layers

Fully-connected feed forward layer [10] - As name suggests, every neuron in this layer is connected with every neuron from layer before and after. Used most commonly in the last few layers of Convolutional Neural Network (CNN). This layer looks at output from previous layer representing set of features (feature maps), then takes the most relevant features and propagates them to the output. Size of output matrix from this layer is equal to number of neurons in this layer.

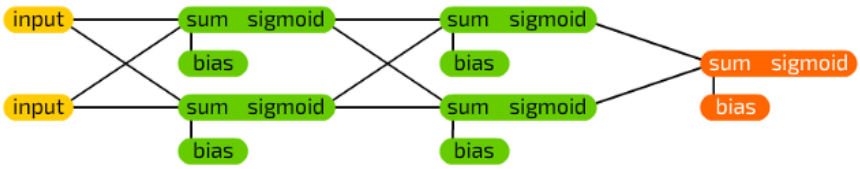

Figure 5: Fully-connected feed forward layer [11]

Convolutional layer [10] - Inspired from vison nerves deep inside the brain which are able to generalize far better than Fully-connected layers. Output from this layer is a feature map. Operation for one-dimensional vector of input data is computed as:

$$x(i) * w(i) = \sum_m x(m) \cdot w(i-m) = \sum_m x(i-m) \cdot w(m) \qquad (1.)$$

Where x is our data and w is a sliding kernel (or filter). We can see continuous version of *Convolution* on the right side in the picture 6 and its equivalent in *Cross-correlation*. Discrete version would look similar. And for two-dimensional data as:

$$S_{cv}(i,j) = I(i,j) * K(i,j) = \sum_m \sum_n I(m,n) \cdot K(i-m, j-n) \qquad (2.)$$

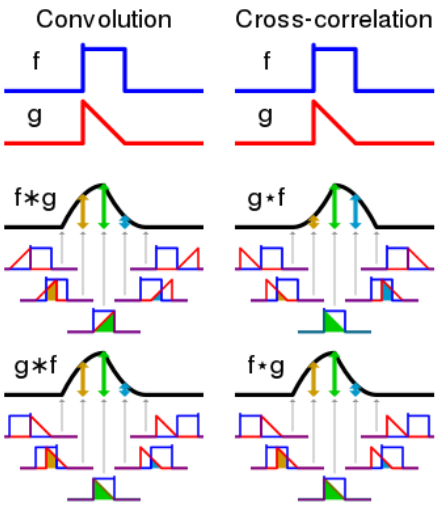

Figure 6: Convolution and Cross-correlation [12]

Where *S(i,j)* is a picture after convolution with pixel in row *i* and column *j*. *I* is our original image and *K* is kernel window with size $M \times N$. In neural networks, cross-correlation replaces convolution with function:

$$S_{cc}(i,j) = I(i,j) * K(i,j) = \sum_m \sum_n I^*(m,n) \cdot K(m+i, n+j) \qquad (3.)$$

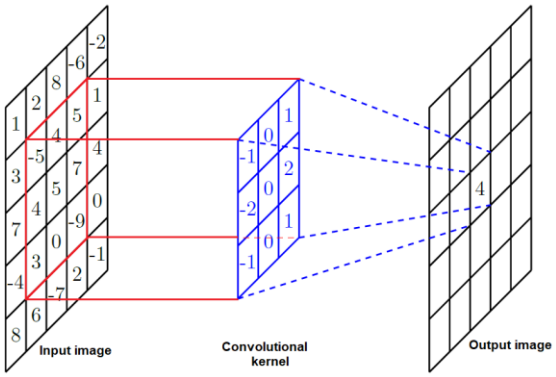

Figure 7: Basic idea behind convolution layer [13]

If we consider scenarios where our kernel peaks outside input matrix, we have two possibilities. Either we zero pad around our original image with half the size of our kernel and use normal convolution or we won't use convolution on side pixels at all. If we choose second option, we end up limited in detecting surface. Choosing a right option depend on input image size and kernel size.

Pooling layer [14] - Pooling layer is used in NN to reduce data input amount. Sometimes referred to as Down-sampling or Subsampling layer. This layer effectively reduces data for following layers. It is favourable, because computational power needed for an application is reduced for architectures with smaller neuron count and their corresponding weights. Overfitting is also greatly reduced, because NN no longer learns on small unimportant features in image. Function of convolutional layer could be summed up in these steps:

1. Select submatrix of whole matrix of data, typically 2x2, 3x3.
2. Compute max value (maxpooling) or mean (meanpooling) or average (averagepooling) or min (minpooling).
3. Computed value put into new grid with smaller size in the place of input submatrix.

This selects only interesting features from input layer and effectively down-samples data size. Principle of maxpooling layer is shown in the picture below and its position in the whole NN architecture.

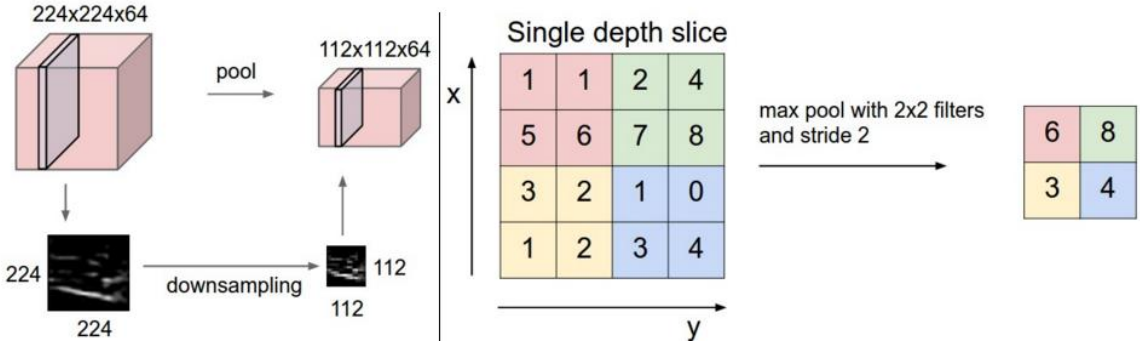

Figure 8: Maxpooling function [15]

Dropout layer [14] - Behaving more like an algorithm that ignores certain percentual portion of input data (sets their activations to zero). Can be used to reduce overfitting, because model is trained for each iteration with different set of data. With correct dropout rate we can simulate bigger training data set.

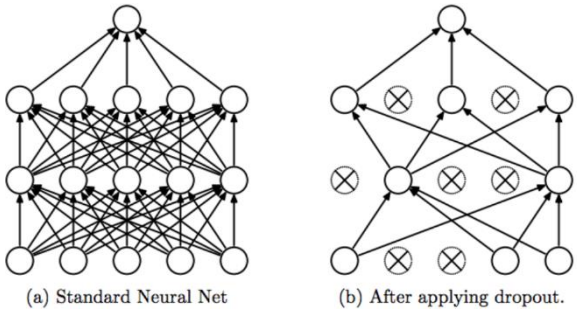

Figure 9: Dropout layer [16]

Flatten layer [14]- Flatten layer is used for reshaping multidimensional data into most commonly one-dimensional vector. As we can see from example, three-dimensional matrix of size $2 \times 2 \times 2$ is reshaped into one dimensional vector with size $8 \times 1 \times 1$.

$$Flatten\left(\left[\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}, \begin{bmatrix}5 & 6\\ 7 & 8\end{bmatrix}\right]\right) = [1,2,3,4,5,6,7,8] \qquad (4.)$$

Batch normalization layer (BN) [13, 17] - This layer normalizes values in mini-batches (all activation functions in all locations) so that we obey convolutional properties. During the inference the BN transform applies the same linear transformation to each activation map. BN Ensures that activations have average 0 and standard deviation 1. This means that most of the values lies somewhere around zero.

We can look on training pass through one layer as transformation consisting of affine transformation followed by element wise nonlinearity, where we add Batch normalization right before the nonlinearity. We can see function below, where *z* is output from our layer, *W* are weights, *u* is input data vector, *b* is bias and *g* is nonlinearity (activation function).

$$z = g\big(BN(W \cdot u + b)\big) \qquad (5.)$$

Since we normalized $W \cdot u + b$, the bias can be ignored because its effect is cancelled by subsequent mean subtraction, thus forming:

$$z = g\big(BN(W \cdot u)\big) \qquad (6.)$$

This layer most commonly sits between convolutional layer and activation layer.

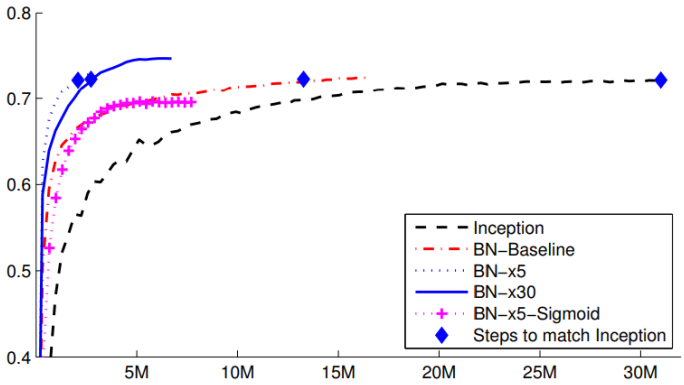

Figure 10: Validation accuracy for Inception and Batch normalized variants vs the number of steps [17]

Thanks to this layer, we can use bigger learning coefficient and care less about correct weights initialization. Technically usable as replacement for a dropout layer as it adds noise into normalization, thus promoting regularization. In practice it is best to use both. Statistics show that this type of layer greatly increases NN success rate and training speed.

2.2.2 Commonly used Neural network architectures

At first, people were using methods like Viola-Jones framework (2001) or Histogram of oriented gradients (2005), which did not use NN. Both were relatively complex with simple reasoning behind them. As computer components got more and more powerful, first techniques employing NN emerged. We went from learning for months for a simple NN to learning in a matter of minutes or hours. With these improvements came stable implementations of popular NN architectures. These architectures let us use model with possibly years of research and apply them in with relatively few steps. This option is incomparably faster than investing months and creating our own from scratch.

CNN - Convolutional Neural Network (2012) [18, 19]

CNN uses sliding window that scans entire picture and for every image window the classifier computes probability that an object is present. There is enormous amount of classifications but most of them has small confidence score. Confidence score represent probability or in other words confidence, that object of that category is present. This method works, but is slow due to high amount of comparisons. Thanks to high amount of computations, CNN can be hardly used as real time classifier.

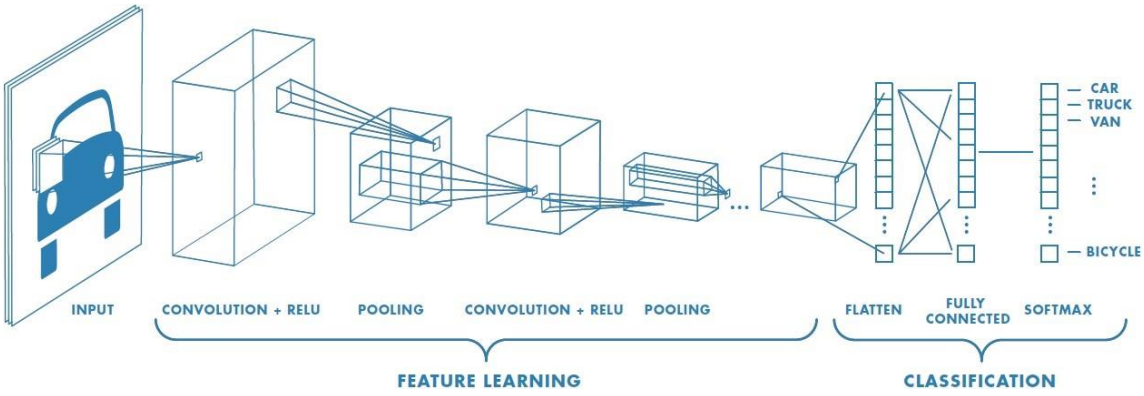

Figure 11: Convolutional Neural Network architecture [18]

Algorithm:

1. Input image is cut into image cuts.
2. CNN classifier is applied on every image cut. This classifier computes confidence score for every category that can be present in the picture.
3. Classified tags are stored only if confidence score is higher than predefined threshold.
4. Rectangles around objects with the highest confidence scores are drawn.

OverFeat (2013) [19, 20]

We don't know size of object in picture. Object could easily cover as much as whole screen or as little as few pixels. OverFeat therefore modifies CNN to use many differently sized sliding windows. That prolongates the whole process but increase accuracy. First published version contained 6 differently sized sliding windows.

Figure 12: Sliding vindow in OverFeat [20]

R-CNN - Regions with CNN (2014) [21, 19]

More sophisticated method than CNN. Sliding windows are no longer used. Instead a process called selective searching happens right after image input. This makes it longer to train because we can't just feed annotated images we want. We have to annotate every extracted bounding box.

Algorithm:

1. Find borders for all areas with similar context and create bounding boxes around them.
2. For every bounding box, extract features with CNN.
3. Use CNN or SVM for classification.

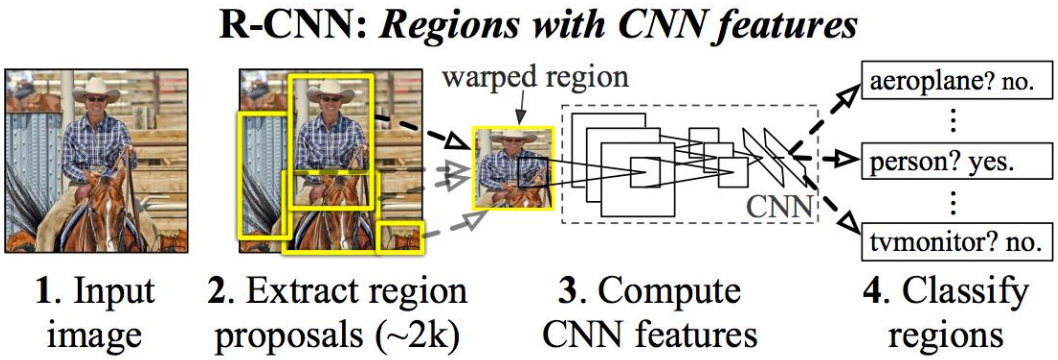

Figure 13: RCNN - Regions with CNN [21]

Results were relatively satisfactory, but this algorithm had one critical flaw. To train you first had to generate proposals for training dataset. That means possibly thousands of smaller proposals in images. On top of this created proposal dataset was later used CNN.

Fast R-CNN (2015) [22, 19]
Consecutive R-CNN with better accuracy, also faster while training and testing. As it turned out Fast R-CNNs biggest disadvantage is selective search that was its strength at the very beginning. Selective search is relatively slow and this method uses selective search for generating bounding boxes with objects but instead of sending them separately for classification, Fast R-CNN uses CNN on a picture as a whole. Later uses both regions of interest and feature map for CNN classification.

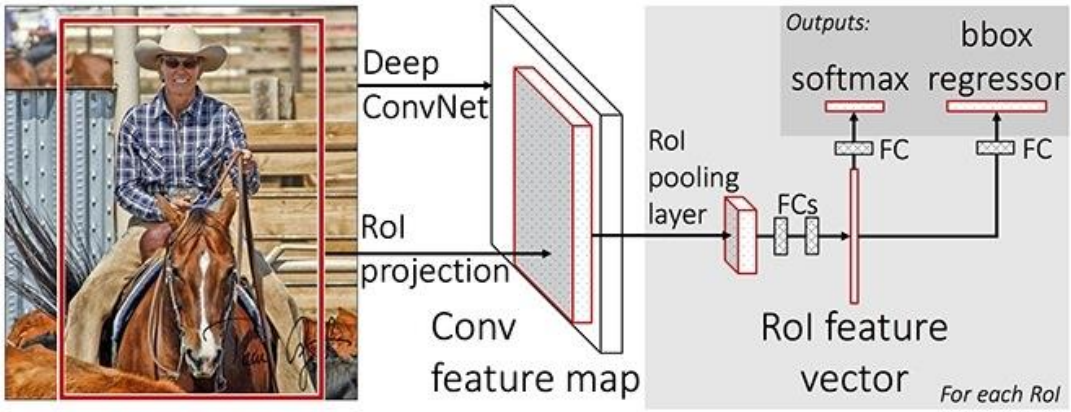

Figure 14: Fast R-CNN architecture [22]

Faster R-CNN (2015) [23, 24, 19]
Improved and much faster form of R-CNN. Removes selective search and instead uses layers for region proposals - Region Proposal Network. Output from Region Proposal Network are objects with their own confidence score. This confidence score characterizes how confident NN is that an object is present in the region. On these objects is then applied approach from Fast R-CNN. This architecture is completely end-to-end trainable.

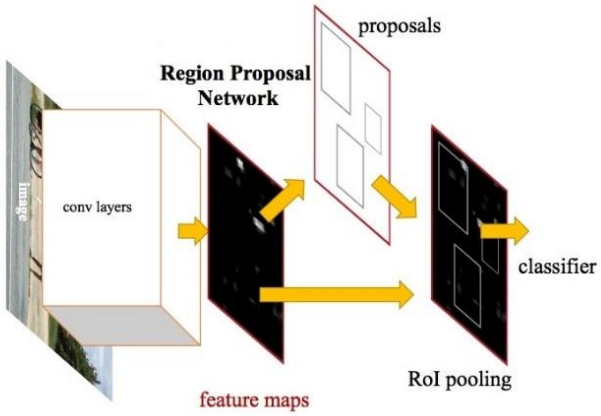

Figure 15: Faster R-CNN [21]

R-FCN - Region-based Fully Convolutional Networks (2016) [25, 19]
Another improvement from Faster R-CN presents R-FCN. This type of NN uses only convolutional network in all parts. Amount of work needed for every region of interest is greatly reduced in this architecture. Feature map is sent to two parallel branches. One branch is computing regions of interest and the other one contains score maps. We can look on score map as matrix evaluating match of one of many classes object with input image. In the next picture a principle of a score maps is shown. Bounding box region is divided into a matrix with size 3×3. Every element from this matrix contain its feature map and compares feature map from comparing classes matrix element feature map on the same position.

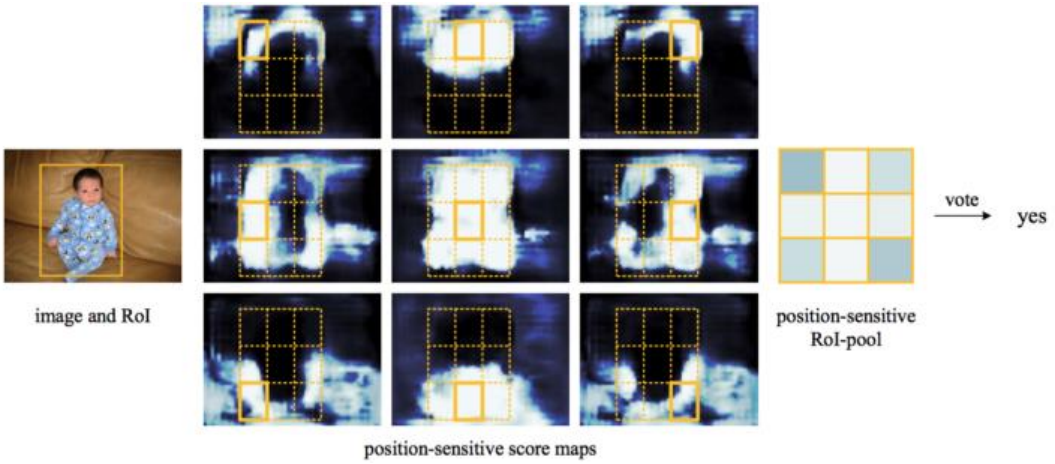

Figure 16: Feature maps of region-based Fully Convolutional Networks [25]

If we use feature map as a whole, we may not get the best match success rate. If we divide feature map into *N* subblocks and correlate them all with matrix feature maps for different objects on the same positions, we get better understanding of image scene. Each subblock has its own vote. Common number of subblocks N is 9.

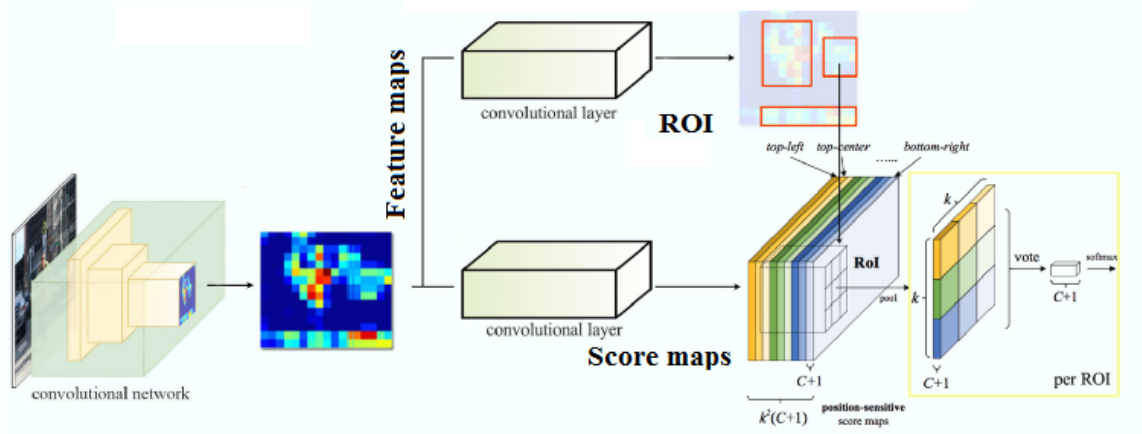

Figure 17: R-FCN architecture [25]

R-FCN architecture is shown in the picture above, which ich very similar with Faster R-CNN but use only convolutional layers.

Mask R-CNN (2017) [26, 19]

Uses similar two-way architecture like Faster R-CNN. Region Proposal Network (RPN) part of the network is identical with Faster R-CNN (prediction of bounding boxes), second parallel part with predicted class Mask R-CNN computes also its binary mask for every Region of Interest. Mask is trained with fully connected NN. It helps in pixel learning and searching pixel-oriented match of class based on similarity. Very satisfying object segmentation was obtained. Its main disadvantage is speed degradation and way harder and more time-consuming image labelling, as image mask has to be used.

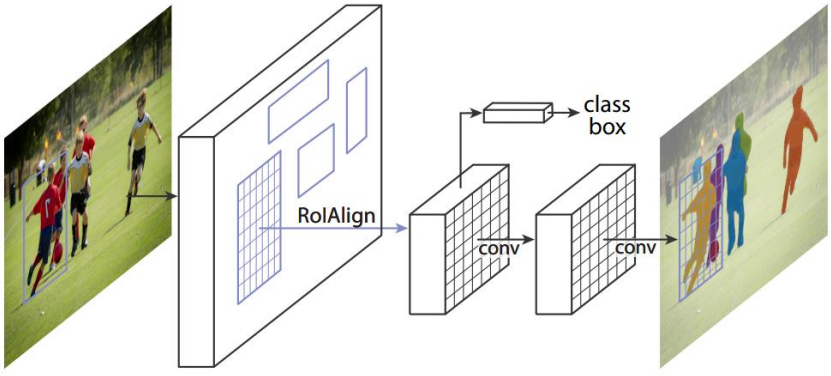

Figure 18: Mask R-CNN architecture [26]

SSD - Single Shot Multi-Box Detector (2015) [27, 19]

All methods mentioned before share one crucial characteristic. They divide whole process into two parts. First part proposes regions for Bounding boxes and second part uses classifiers to classify object class. These methods are relatively simple and precise but pay for this with undesirably slow computation speeds, resulting in small FPS (frames per second). This makes them basically inapplicable on embedded devices. SSD keeps this in mind and tries to overcome that problems. Only one NN is used for both tasks. Instead of NN that creates Bounding boxes suggestions, predefined search-for boxes are created. On these predefined boxes is then used second part of NN which uses feature maps from convolutional layers, where small convolutional kernels are used. These feature maps evaluate probability that an object is present. These predictions are used as bounding boxes for classification. Many different activation layers and differently sized kernels are used for feature map prediction.

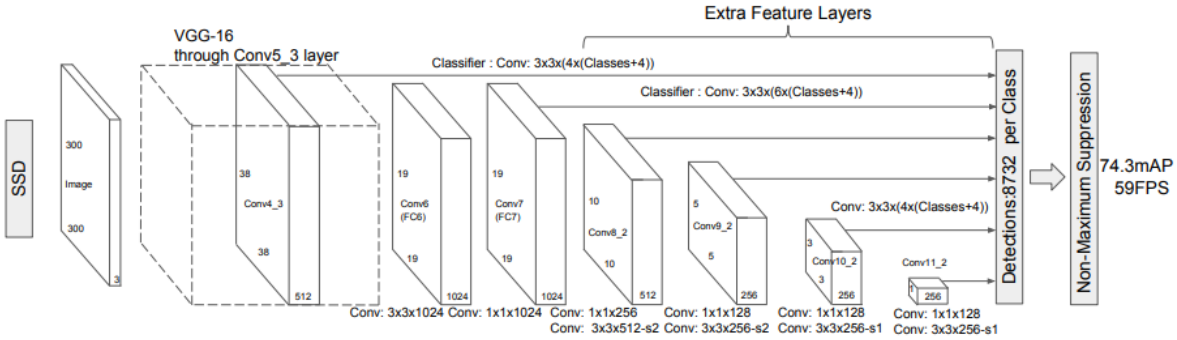

Figure 19: Architecture of SSD [27]

YOLO- You Only Look Once (2016), YOLO v2 (2017), YOLO v3 (2018) [28, 19]

Probably the most publicly well-known type of neural network architecture is YOLO. There are many variations including mobile version Tiny Yolo. Yolo uses only one network for a whole input image. This was a new concept at the time of its introduction. Till this time, almost every DL detector used some form of sliding window. Whole input image is divided into regions and then bounding boxes for objects are predicted with their respective probability for every bounding box. There may be hundreds of predictions but most of them has very little predicted probability, therefore after using threshold of around 0.5 or above only a few dominant remained.

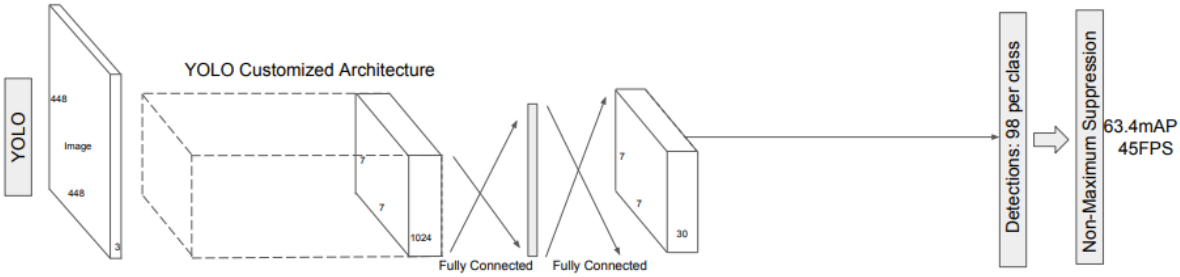

Figure 20: Yolo architecture [27]

Yolo let image pass through CNN only once. This makes it possible for the first time to accomplish real-time application. This version has high success rate on its own and with every later version become even better.

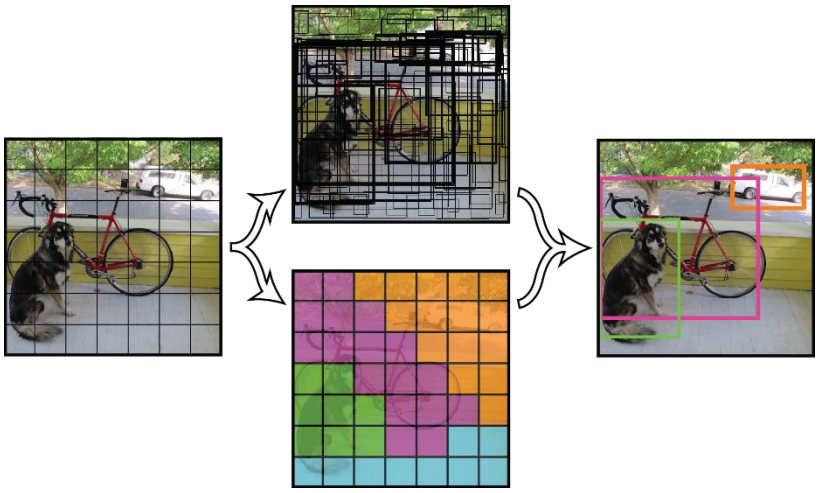

Figure 21: Yolo - predicted bounding boxes [28]

Yolo V2 uses architecture of darknet with 19 layers for bounding box prediction and another 11 layers for classification. Its biggest disadvantage was quality loss of predictions for small objects. Yolo V2 combats this limitation by using feature maps from earlier phases in later layers. [29]

Year after Yolo V2, a new version was revealed. Yolo V3 became the fastest NN used with highest accuracy. But it didn't last long and new versions of SSD and RetinaNet surpassed Yolo V3 in success rate. Yolo V3 uses 53 layers from Darknet for features extraction and another 53 for detection. Great advantage over first versions is increased detection rate for smaller objects. [27]

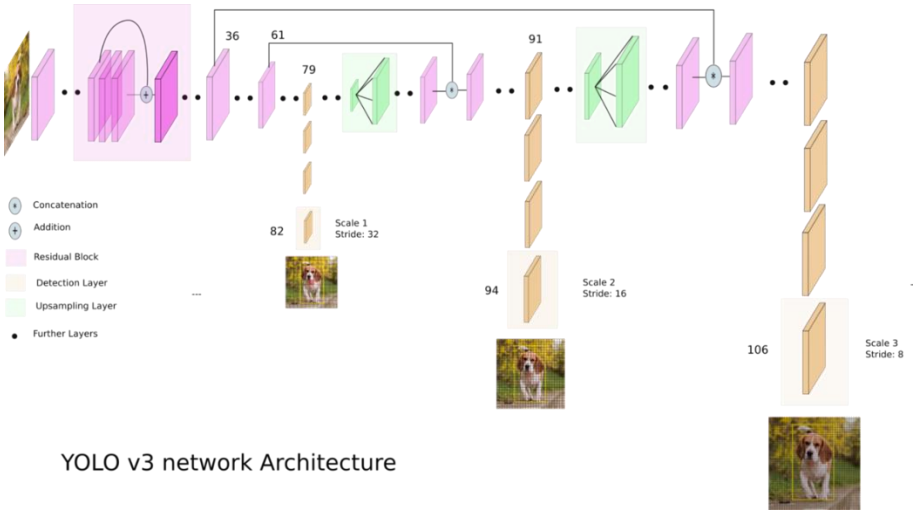

Figure 22: YOLO v3 architecture [27]

2.2.3 Training process and datasets

As we have already mentioned, DL are part of supervised algorithms family. That means we have to have training pictures in which we know if objects are present and their location. Next image demonstrates what we would need, if were training binary classifier that detects goats. We have to provide bounding boxes representing their locations and annotate them with name of their class. In this example it is not really needed, since we are only detecting if object is present or not in image, not its type.

Figure 23: Visual representation of objects in goat detector example

When choosing a dataset, we have to define strictly what we want to detect, so we can pick best suitable option. If we use small dataset, accuracy may not be as good as expected, so we have to have big enough and relatively specific dataset to our detection/classification problem. In reality it may be challenging to create your own dataset large enough to train NN. In that case we have several options, like asking people that worked on a similar project to provide their datasets, taking our own images or using online search engines and annotating every single picture manually or download a whole annotated dataset from one of many online dataset stores. From mentioned, only first and third possibility are really feasible when it comes to large models (thousands of data elements needed). It should be enough to get just few hundred images and annotate them manually for binary classification problem and not as complex architectures.

Right dataset might present noticeable increase in prediction correctness (high positive predictive value and sensitivity, low error). That means using broad variation of objects we want to recognize but with enough specificity. To better describe why, let's imagine that we are modelling classical problem, recognition of cars in Czech Republic. We want to use pictures of different cars from all the different angles and under different light conditions. If we forget to take into consideration factors like mentioned light conditions, we might end up with object classifier that works great during the day, but

not at all during the night or when its foggy. Currently, there are these most commonly used online datasets:

Table 1: Comparison of different Datasets

Dataset name	No of pictures	Main classes	Last updated
ILSVRC'16 [30]	456K	200	2014
COCO [31]	200K	80	2017
Pascal Voc [32]	12K	20	2012
VGGFace2 [33]	3,3M	Only faces -1	2012
KITTI Vision [34]	15K	Only cars - 3	2012
Stanford Car dataset [35]	16K	Only cars -196	2012

There are many more but these shows predominately in online search engines at the moment and offer not only images but also their labels. Car and face/person detection are common problems in computer vision field, that's why they have a whole dataset dedicated to them. Worth noticing in the table is ILSVRC'16 (Large Scale Visual Recognition Challenge 2016). It is only a subset of whole Imagenets dataset. This smaller dataset was used for competition in object detection and classification benchmark that happens every year. Saying "smaller" may induce feeling that this dataset was not big enough but having millions of unlabelled images, thousands of labelled and hundreds of classes is far more than plenty. For example, a few pictures from common class cars from this dataset looks like this:

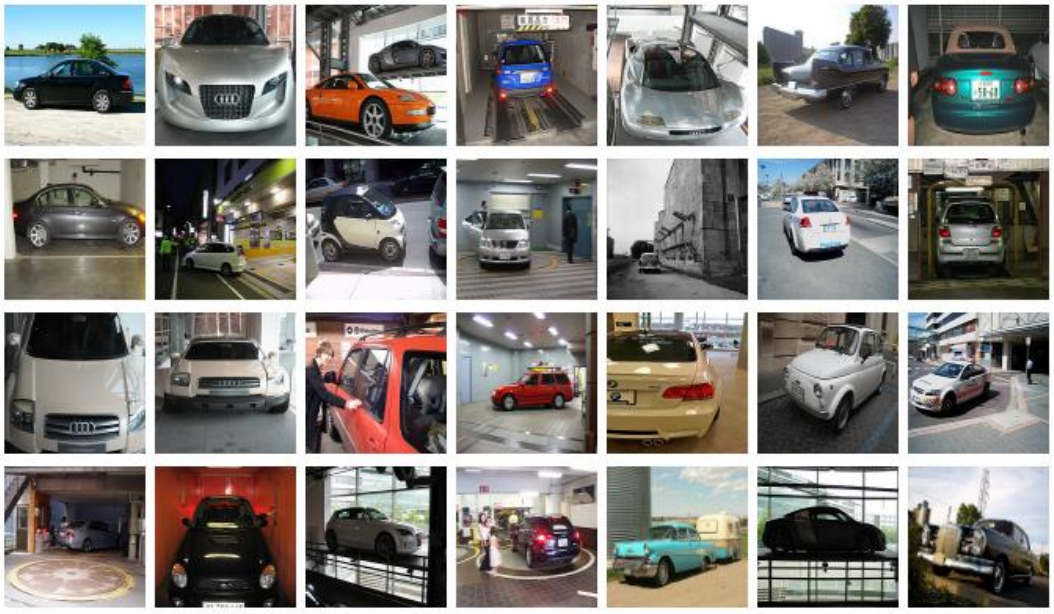

Figure 24: ILSVRC'16[30] cars dataset preview

After obtaining enough data, we usually split them in 9:1 or 8:2 ratio for training and testing dataset. Testing dataset is used for model accuracy testing while training. We should not use the same images for training and testing dataset.

3 EMBEDDED SYSTEMS

In our everyday life, we came into contact with many various electronical devices in one form or other. Some of them are relatively simple devices used for single tasks like calculators, refrigerators or elevators. Some of them are on the other hand as complex as smartphones, computers or tablets that can be programmed with little to no effort to execute many different tasks.

Because this chapter is dedicated to embedded systems, it is relevant to mention that most of currently used electronic devices are embedded devices. But it is equally important to recognize that not all devices are embedded. Embedded devices lie somewhere in between plain "stupid" electronical devices that were built without any complex logic like older washing machines with no display, processor or logical units and on the other side complex machines like computers or smartphones. To understand why some devices are called embedded and other are not, it's useful to distinguish what embedded devices actually are and what are their characteristics. Definitions for embedded systems may differ a bit but the main characteristics remains the same. These characteristics are:

- Contains its own processing unit
- Built to perform one specific task or very small number of tasks
- Has its own memory

This means that smartphones or computers are counted out from list of embedded devices because of their multipurpose usage. Using computer-like systems in the place reserved for embedded devices may mean increase in flexibility for the price of higher cost and physical size. Generally, simpler solution with embedded devices are preferred by manufacturers over usually redundant complex units. There are just too many redundant parts that increase price of final solution. It is although often hard to categorize if a device is still embedded or general-purpose computer.

Figure 25: Examples of embedded devices

Because there are many different HW configurations, we have to be extra cautious. Picking first device that seems promising might not be the best, even with the best intentions. Choosing equally good SW and HW is a must. If we pick a HW with not enough memory and weak CPU, not even superiorly good SW will do much. The same goes other way around too. We can't expect miracles if we take unoptimized code that wastefully uses resources.

Embedded systems may typically use microprocessor or microcontroller. Difference between them is that the microprocessor is contained on board of microcontroller with additional components like RAM, ROM, flash, GPIO pins or other peripheries. Microcontroller is like a miniature computer on its own. It does not require any additional circuits as it has all components needed. Microcontrollers can be called a heart of embedded devices. Microprocessor on its own contains only CPU, cache, dram memory and sometimes GPU. They are dependent on other additional circuits to work, as they lack all other peripheries. Depending whether we want to build our device on our own or use one already pre-built, we can choose between microcontrollers or microprocessors which need additional HW. [36]

It is useful to ask a few questions when deciding on choosing an embedded device. These questions, taking both software and hardware into consideration, should be asked before trying to find feasible solution:

- How much RAM/ROM/HDD/flash we need and should we use CPU, GPU, FPGA, ASIC or TPU?
- Will this device be powered from battery or wall socket?
- Will our device offer direct connection with USB/ethernet/can… or wirelessly?
- What will be the maximum final price?
- Is there a prediction for devices interconnection for increase in performance?
- Are there any possibilities for remote control?
- Should our device have OS installed?
- What problem are we going to solve with ML methods and will this device be sufficient?
- What algorithms/methods/framework/architecture for our project is required and is there support for this device?
- How robust/big the final architecture will be?
- Should be training phase done on the same or different devices?
- What is the required speed of computation/FPS?
- How often are we going to change programmed model architecture?

Before looking for answers it is often helpful to do a web research and find compatible solutions solving problems similar to ours. It is almost always considerably faster to tweak well tested logic and utilizing it to our desired functionality than to create a new one completely from scratch. [37]

3.1 Types of hardware for embedded devices

For common embedded devices, speed is a major problem as they are minimized versions of their larger relatives. Computers can get up to a few dozen FPS in tasks like image recognition and DL. A model designed to run on multi-GPU/TPU system will simply not run effectively on embedded system that uses typically only simple processor. These embedded devices lack hardware support for computing matrices multiplication, which are the most common type of operation in neural networks.

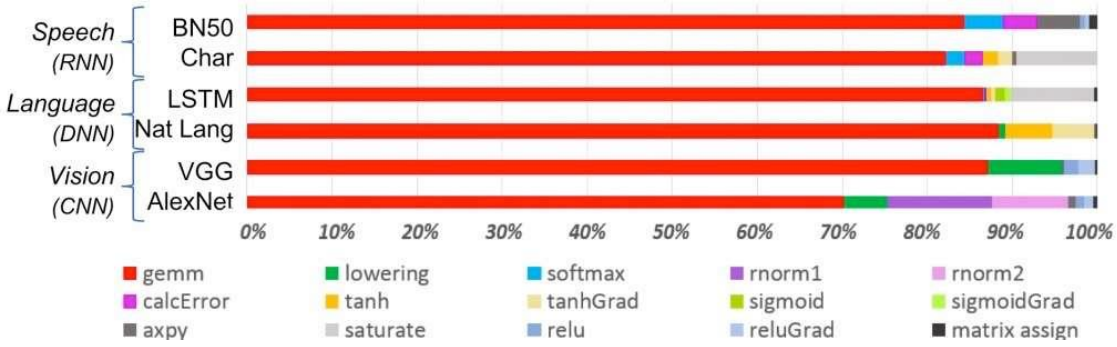

Figure 26: Comparison of matrix multiplication tasks (gemm) and other in neural networks [38]

However, that should not stop developers from trying to intercorporate techniques as NN onto smaller embedded devices. In today's technology advances, it's possible to use frameworks for neural networks on embedded boards like Raspberry Pi.

Using an NN on embedded device require a lot of RAM, because models are loaded in memory. This may present problem for devices with low RAM. Another problem we have in NNs is that we have to create smaller architectures to keep it as fast as possible. If we make NN architecture smaller, it may lead to underfitting problem. Also, training NN architecture on embedded device would take days or weeks. That's why we can see numerous times in literature that training is done on more powerful machines.

Data memory is also a huge problem as images with higher resolution takes more memory. For example, picture with resolution 1280×720 $(720p)$ will have around 3.7Mb in uncompressed TIFF CMYK 4x8 bit/pixel. Now take into consideration video stream with 25-30 FPS. We will definitely need additional memory device like SSD disk if we plan to save recordings. Having 640×480 $(480p)$ could be considered luxury. We may get few FPS when using DL methods on embedded device anyway.

Today's trend in neural networks characterized as "Bigger is better". Embedded devices are not accommodated for this trend. Next picture demonstrates budget of device with its expected recognition rate.

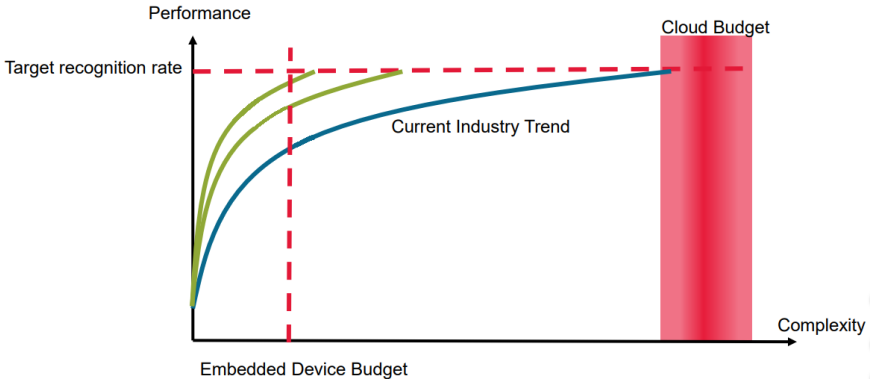

Figure 27: Trend in Deep Learning [39]

As this project possess limited resources, low cost is a preferable option. But before we start with choosing the best fitting device, we have to make a clarification on different HW types, that may be used. In general, we will be talking about a CPU/GPU/FPGA and ASIC. Each type of processing unit has its own advantages and disadvantages, characteristics, considerations and uses. For our designated purpose are needed characteristics like computational capability, latency, cost and energy-efficiency. Quantifying hardware performance is typically done using number of MAC operations performed in given time unit - Millions of Connections Per Second (MCPS), weights updating is quantified with Millions of Connections Updates Per Second (MCUPS). These two measurements correspond to traditional Mega Floating-point Operations per Second (MFLOPS) measured on conventional systems. [40, 41]

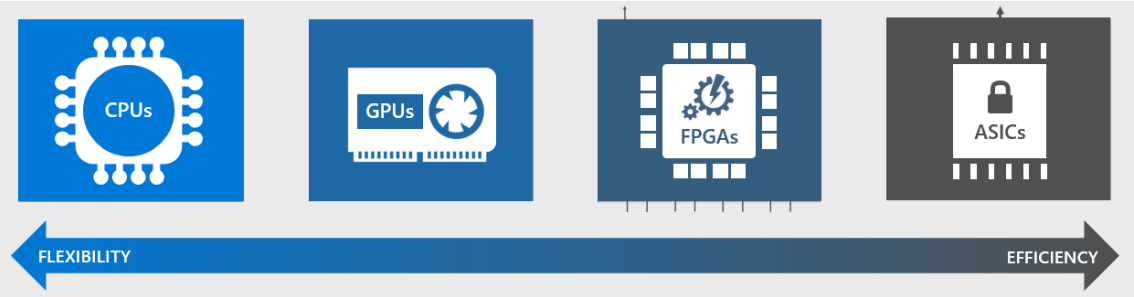

Figure 28: Comparison of different hardware options [42]

3.1.1 CPU - Central Processing unit

CPUs are referred to as brains of computers, smartphones, laptops or tablets. They are vital component. Because of how many different use cases they should be doing, they must be as flexible as possible. Increased flexibility causes decrease in the efficiency and performance for specific task. CPU is supposed to handle big workloads. In their essence, they contain mostly less than 10 cores, typically Dual Core, Quad Core, Hexa Core, Octa Core. These cores have large cache memory and each core is capable of running few threads at the time. CPU is a scalar machine, which means it processes instructions step by step, in other worlds, optimized for serial computations. [43] Object detection may require parallel computation for optimal performance. Computers may contain combinations of CPUs ang GPUS for better performance.

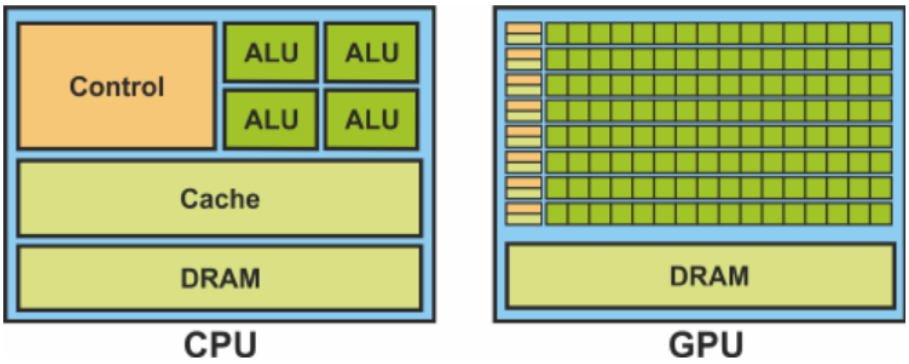

Figure 29: Comparisson of CPU and GPU structure [43]

3.1.2 GPU - Graphics Processing Unit

These processing units are currently the most widely used HW for NN. GPUs are designed for high level of parallelism and high memory bandwidth. With GPU we are trading flexibility for efficiency and performance with processing graphics intense workloads. Strength of GPUs lies in energy-efficient matrix multiplication and convolutions. They offload CPUs from computationally intense work. Because ML and DL requires tremendous amount of iterations of matrix multiplication and convolution while learning, GPUs shows advantage over CPUs. GPUs are also good at fetching large amounts of memory, contains hundreds of cores that can handle thousands of threads simultaneously. In article [44] we can see comparison between GPU with 56 processors that each has 32 cores (total sum of 1792 cores) running at 1.48 GHz versus 16 core CPU running at 3.0 GHz. They both perform multiply-add instructions. GPU has peak performance of 5300 GFLOPS and CPU only 96 GFLOPS. This superior floating-point performance was achieved thanks to large number of cores. GPU generally provide an order of magnitude over CPU in processing power with the same cost. [44] Problem with typical applications using GPUs is a need to accompany them with CPU. CPU can run without GPU, but GPU without CPU can’t, they are typically unable to run operating

system like Windows, Linux or Unix. This means more energy and physical space needed and price will likely double.

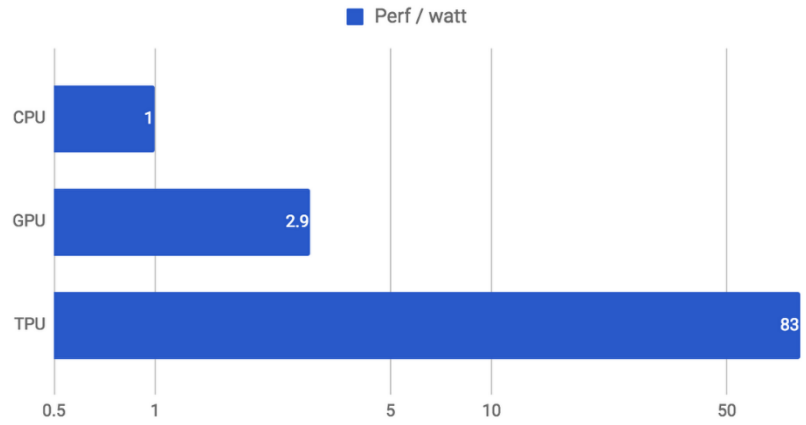

Figure 30: Performance to watts consumed comparison compared to CPU/TPU [45]

3.1.3 FPGA - Field programmable Array

FPGA uses hardware descriptive language HDL and can be programmed or reconfigured basically infinitely. As name suggests they contains matrix of configurable logic blocks connected via programmable interconnections. Loading programs to FPGAs takes a lot of time which could be perceived as time wasting but their strength is in lower power consumption for the same performance as GPU, which can be important in certain applications like self-driving cars or IoT. They are not as good at floating-point based operations as GPUs but still can provide quick results for uploaded pre-trained models stored in FPGA memory. As data could be directly received and processed inline, a lot of resources are saved from what would normally consume running host application. FPGAs are suitable for real time applications. FPGAs flexibility aids in delivering deterministic low latency and high bandwidth. As they are reconfigurable and can be reconfigured basically infinite times, they become viable option for algorithms using topology adaptation mechanisms. FPGAs were designed for customizability when running irregular parallelism and custom data types. If trend continues, FPGAs will become applicable for running more NN applications. As of today, their price is many times higher than common embedded devices using CPU/GPU for the same purpose. [46]

3.1.4 ASIC - Application Specific Integrated Circuits

ASIC are not as cheap as other candidates. Because of their task specificity and custom design, their prices can climb really high. But we really pay for their performance of magnitude higher than GPUS or FPGAs. They might be more energy efficient as they lack redundant logic for their designated super-specific purpose. Some ASICs offers low-latency, high-memory bandwidth chip built specifically for deep learning. [47]

3.1.4.1 TPU - Tensor Processing unit

Belong to ASIC device category and are currently developed by Google. These devices are by far the fastest among other types mentioned before. TPUs are specialized in multi-dimensional matrix computations. They are generally used for very large models. [43] Problem with these devices is that they are currently not publicly available. Only way to obtain their computational power is to rent them on Google cloud. Cloud TPU v2 currently costs $4.95 USD per TPU per hour. [48] Another option is to use USB TPU accelerator, which requires host. However, performance is not as great.

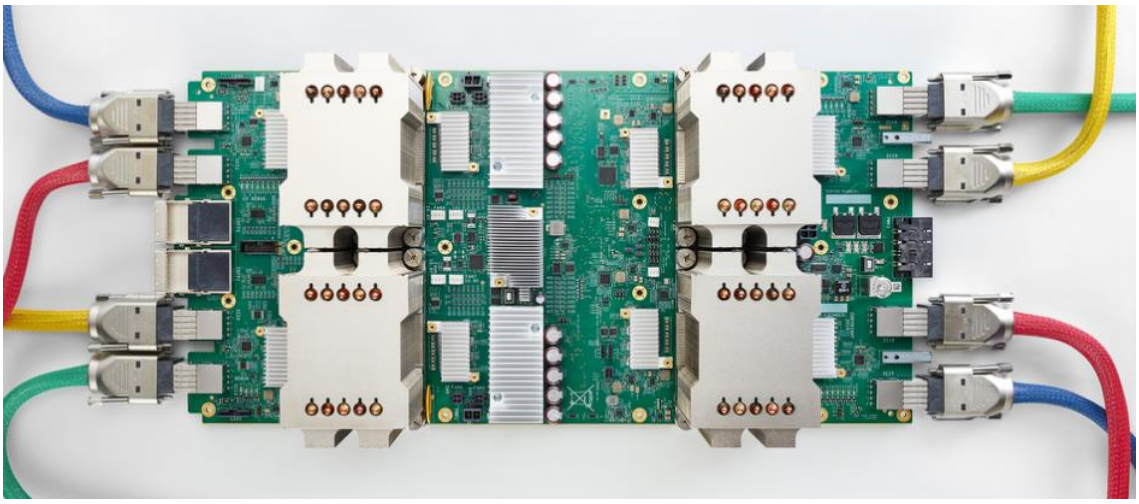

Figure 31: Google TPU2 [48]

From all mentioned before, we can conclude that the best fit for specific application would be ASIC or FPGA. However, because our implementation will be changing very often, as we will be implementing everything beside object detection for our application from scratch, using FPGA will be extremely time-consuming. Also, FPGA and ASIC are significantly more expensive. In our case thus will be the best fit some microcontroller with CPU and GPU integrated onto the same board or microcontroller with CPU and USB accelerator.

3.2 Commonly used hardware for Deep Learning solutions

Most of high-performance solutions use high-end configurations with extremely powerful computer utilizing GPU or TPU clusters. Prevalent example of TPU cluster is Google cloud. These solutions are for commercial or serious research purposes. Embedded enthusiast, student and most of DL users amongst general public use CPU+GPU combination. This chapter provides a brief discussion on some of the most convenient embedded devices for DL.

3.2.1 STM32F407VG MCU [49]

Arm based microcontrollers belong to a group of most well-known microcontroller units (MCU) on the market. They are popular mainly thanks to their applicability in applications, where a lot of data transfer from input/output connection points is required. With reasonable price and relatively small size offer reliability. As progress advances in machine learning, so does MCU. STM32F407VG and many other ARM Cortex-M core microcontrollers offer collection of optimized neural network functions like convolution, depth separable convolution, fully-connected, pooling and activation layers. With its utility functions, it is also possible to construct more complex NN modules. While Cortex-M series processors are capable of running OS like WinCE / Linux or Android, they are aimed for different purpose. STM32 is a development board created for developers as a tool for prototyping their concept and later on creating their own board. Using their peripheries directly allows them to gain control over execution and resources.

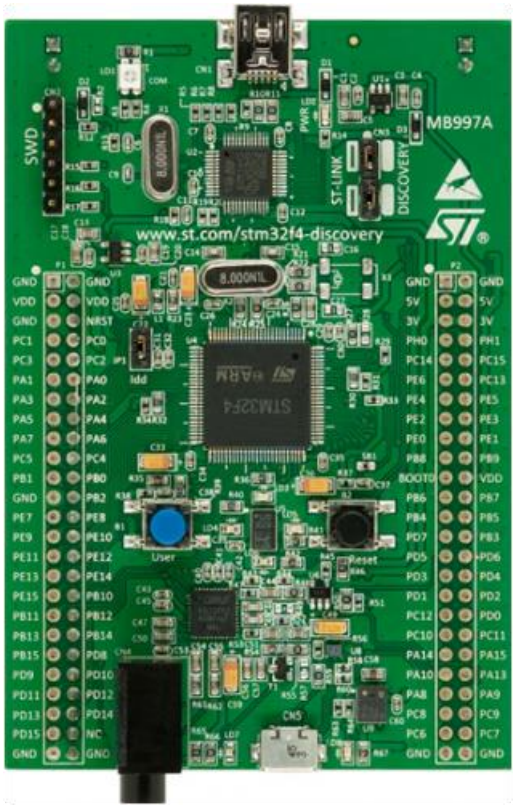

Figure 32: STM32F407VG MCU [49]

All training takes place on PC. Its weights and biases are quantized to 8 bit or 16 bit integers, then the model used for interference is moved to STM32F407VG. STM supports popular frameworks like Tensorflow or Caffe.

Table 2: Specifications for STM32F407VG MCU [49]

Specifications for STM32F407VG MCU	
CPU	ARM®32-bit Cortex®-M4
Frequency	Up to 168 MHz
Fastest Calculations	<0.2 GFLOPS
Flash	1 Mbyte
SRAM/RAM	196 Kbytes/64Kbyte CCM
Power supply	5V
Operating system	Raspbian/ Windows / Linux
Connection points	USB 2.0/Ethernet / SDIO/ CAN/I2C/ UART,SPI
Storage space	Support for Compact Flash/ SRAM/PSRAM/NOR/NAND
Display	External (ILI9325, ILI9341, SSD2119, SSD1963)
Sensor	8- to 14-bit parallel camera interface up to 54 Mbytes/s
Dimensions	74.8mm x 57.5mm

3.2.2 Raspberry Pi 3 Model B [51] [50]

Raspberry Pi 3 is a small and relatively cheap single-board minicomputer. At this time its price is around 35€. Despite its low price and small size, RPI possesses greater computing power and better specification than single-boards in the same price range. Its main advantage is low price and large number of contributors on online forums in vast number of applications.

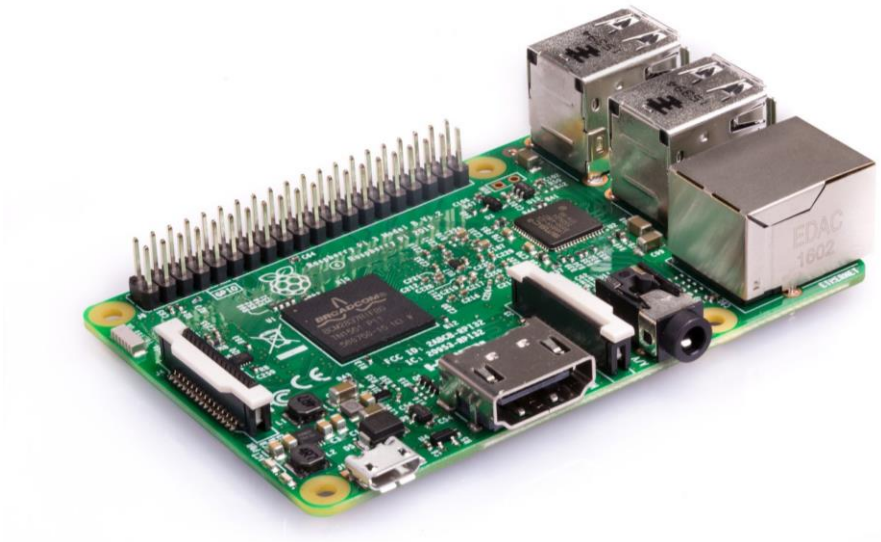

Figure 33: Raspberry Pi 3 Model B [50]

Another advantage that increase attractiveness of RPIs is direct camera connection. Camera modules are cheap, whole HW price could get as low as 60€.

Table 3: Specifications for Raspberry Pi 3 Model B and camera module v2 [50]

Specifications for Raspberry Pi 3 Model B	
CPU	1.2GHz 64-bit quad-core ARMv8
Fastest calculations	192 MFLOPS (double precission) [66]
RAM	1GB (@900 MHz)
Power supply	5V 0,7A (up to 2.5A with peripheries)
Operating system	Raspbian/Windows10/Linux/FreeBSD
Connection points	802.11n WLAN/ Ethernet/ Bluetooth/ 40-pin GPIO/ 4x USB 2.0
Storage space	Uses Micro SD (64GB tested)
Display	HDMI/ composite video /DSI display
Dimensions	85mm x 49mm
Specifications for CAMERA MODULE V2	
Image sensor	Sony IMX219 (CMOS)
Resolution	8MP (3280x2464)
FPS	Max 90
Image format	PEG, JPEG + RAW, GIF, BMP, PNG, YUV420, RGB888
Video format	raw h.264 (accelerated)

Camera Module V2 is typically used for image capture, but other options with Infra-Red sensor are available. Parameters for standard *Camera Module V2* are shown above in table. Easy code implemented libraries allows users to capture photos with just a few lines of code.

3.2.3 NVIDIA Jetson TX2 [51]

One of the most popular and extremely powerful boards to develop and test deep learning. Winner of Image Classification Efficiency Challenge in 2016 with cuDNN 4.0. Useful for computer vision and deep learning. From all listed devices, Jetson TX2 and TX1 provide the best hardware systems specifications. That's why they are widely used, but its price is high in comparison with other devices listed in this chapter. They stand well above maximum budget with price of around 450€. This solution also requires additional HW.

Table 4: NVIDIA Jetson TX2 Specifications [51]

Specifications for NVIDIA Jetson TX2	
CPU	HMP Dual Denver 2/2 MB L2 + Quad ARM® A57/2 MB L2 (2GHz)
GPU	NVIDIA Pascal 256 CUDA cores
Fastest computation	46.8 GFLOPS (double precission)[53]
Flash	1 Mbyte
LPDDR4	8 GB
Power supply	5.5-19.6V
Operating system	Linux/Ubuntu
Connection points	USB 2.0 + 3.0/ Ethernet/ Bluetooth/ I2C/ 802.11ac WLAN/ CAN /UART/ SPI/ GPIO
Storage space	32 GB eMMC, SDIO,SATA
Display	2x DSI, 2x DP 1.2/HDMI 2.0 / eDP 1.4
Sensor	Up to 6 cameras (CSI2 D-PHY 1.2)
Dimensions	50mm x 87mm

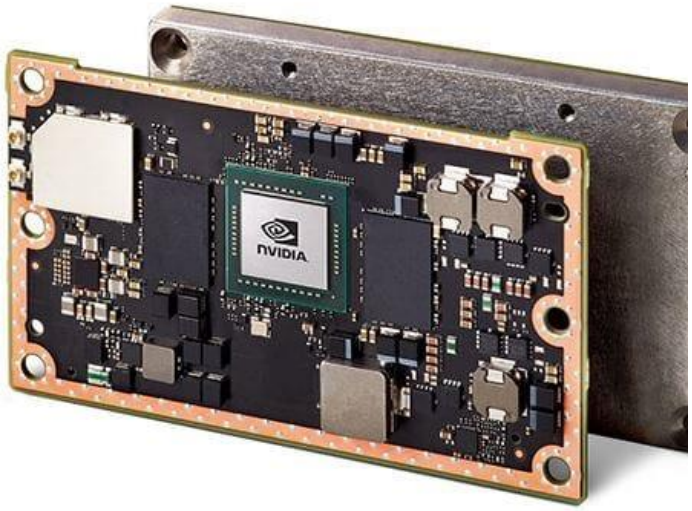

Figure 34: NVIDIA Jetson TX2 [51]

3.2.4 Intel NCSM2450.DK1 Movidius USB Accelerator [52]

Intel Movidius is small USB connected fan-less deep learning accelerator designed for AI programming. Uses Movidius Visual processing unit, that is built in many smart security cameras, robotics or industrial machine vision equipment.

Table 5: Intel Movidius: requirements on host computer [52]

Requirements on host computer:	
Operating system	Ubuntu 16.04 x86_64
Fastest computations	100 GFLOPS (half-precision) [54]
Connection point	USB 3.0 Type A plug
RAM	1GB
Free storage space	4GB

This device require computer or another embedded device as a host. Its huge benefit is ease of portability from device to device. Requirement for host OS is only restriction. Another benefit is that it is possible to connect to embedded devices with less processing power. Inference model is trained on PC, then transferred to MOVIDIUS stick that is later inserted to embedded device. All image computation happens on MOVIDIUS stick. Because all image processing is done on MOVIDIUS, embedded devices have free resources for other processes. We can run multiple devices on the same platform to scale performance.

Figure 35: Intel NCSM2450.DK1 Movidius [52]

3.2.5 Coral Google Edge TPU USB Accelerator and Dev Board

Coral TPU USB Accelerator (neural network co-processor) is ready to use device, that has to be connected using USB cable. This device can connect to any Linux-based system and perform accelerated ML inference. This USB accelerator is compatible with Raspberry Pi boards at USB 2.0 speeds only. It can be even used on minimalistic devices as Raspberry Pi zero. [55]

Table 6: Specifications for Google Edge TPU USB Accelerator [56]

Specifications for Coral Google Edge TPU USB Accelerator	
CPU	ARM Cortex M0 +
Connectivity	USB type-C cable
Power supply	5 V (from host device using USB)
Supported Frameworks	TensorFlow lite
Supported OS	Debian, Linux
Data bus width	32 bit
Dimensions	65mm x 30mm x 8mm

Figure 36: Google Edge TPU USB Accelerator [56]

Edge TPU Dev Board is capable of high-speed machine learning inference for low-power devices. TPU USB accelerator on itself could be connected to many various devices, but Edge TPU Dev Board was designed solely for this purpose. This all-in-one prototyping tool allows user to create systems that demands fast ML inference. Edge TPU Dev Board is ASIC device with high performance, considering its low-power nature. According to [56], it can execute vision models like MobileNet V2 at 100+ FPS. Edge TPU Module is removable, can be integrated without base board to other embedded device.

Table 7: Specifications for Edge TPU Dev Board (Base board) [56]

Specifications for Edge TPU Dev Board (Base board)	
Flash	MicroSD slot
Connectivity	Type-C OTG, Type-C power, Type-A 3.0 host, Micro-B serial console, Gigabit Ethernet port, 40-pin expansion header,
Supported video output	HDMI 2.0a (full size), 39-pin FFC connector for MIPI-DSI display (4-lane)
Supported camera connection	24-pin FFC connector for MIPI-CSI2 camera (4-lane)
Power supply	5V DC (USB Type-C)
Dimensions	85mm x 56mm

Table 8: Specifications for Edge TPU Dev Board (EDGE TPU MODULE - SOM) [56]

Specifications for Edge TPU Dev Board (EDGE TPU MODULE - SOM)	
CPU	NXP i.MX 8M SOC (quad Cortex-A53, Cortex-M4F)
GPU	Integrated GC7000 Lite Graphics
RAM	1 GB LPDDR4
Flash	8 GB eMMC
Connectivity	Wi-Fi 2x2 MIMO (802.11b/g/n/ac 2.4/5GHz), Bluetooth 4
Dimensions	40mm x 48mm

Problem with these devices is their availability. Their availability is limited and there are currently no local vendors. Although this device could be bought from online sources, their usage is still experimental.

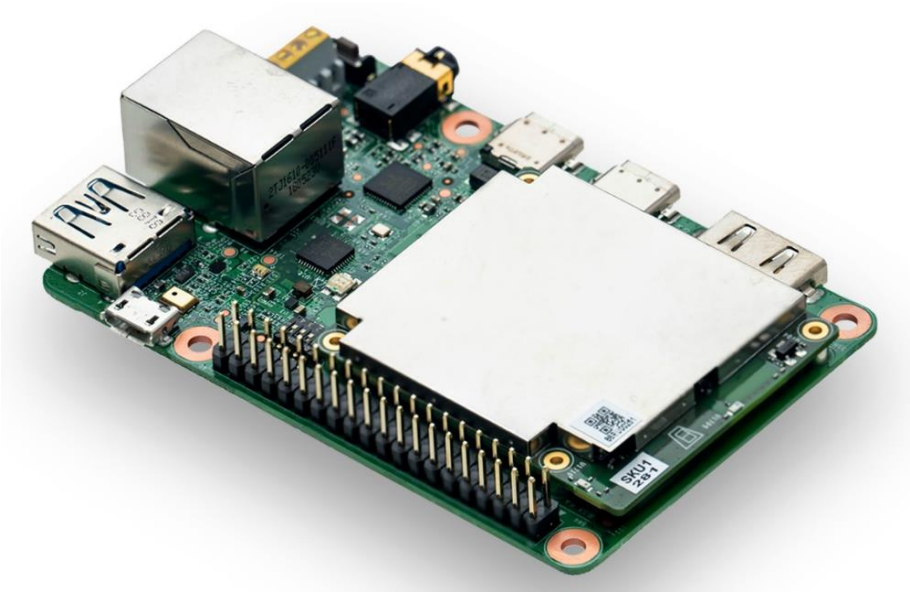

Figure 37: Edge TPU Dev Board (Base board + SOC) [56]

3.3 Choosing the best option for our application

This paper covers development and application of DL principles on embedded device. We are not going to create our own implementation of framework for DL inference, as this could take possibly years for a team of dozen people. We will attempt to utilize commonly used embedded device and apply DL into it with and later optimize it for the best performance. We have chosen vehicle detection as our main task. Therefore, we will be using static camera and detect vehicles moving around car park. Considering this, we can have a closer look at mentioned HW.

STM32F407VG MCU might provide a limited amount of memory. This is a huge disadvantage, as we plan on using slightly larger models (still very small in comparison with their counterparts).

Probably the best performance should be expected from **Jetson TX2board**. This is practically a small PC, which could be employed in very performance demanding application. However, from all listed device is the most expensive with price of around 450€.

From listed devices, only **Raspberry Pi 3 Model B** could be considered as useful at this stage of development. We will use Camera Module V2 with native support. Later in the future, it is possible to incorporate **MOVIDIUS USB stick** or **Coral TPU USB** to increase performance, possible with RPi like board supporting USB 3.0.

Because we have chosen RPI, we have to use one of many OS that are supported. Raspbian Stretch is the official OS for RPi. Desktop version is preferred over lite for visual debugging directly on device. When it comes to frameworks, Tensorflow v1.5will be used. All other programming will be done using Python 3.6.2. Our device in simple acrylic case is shown in image below.

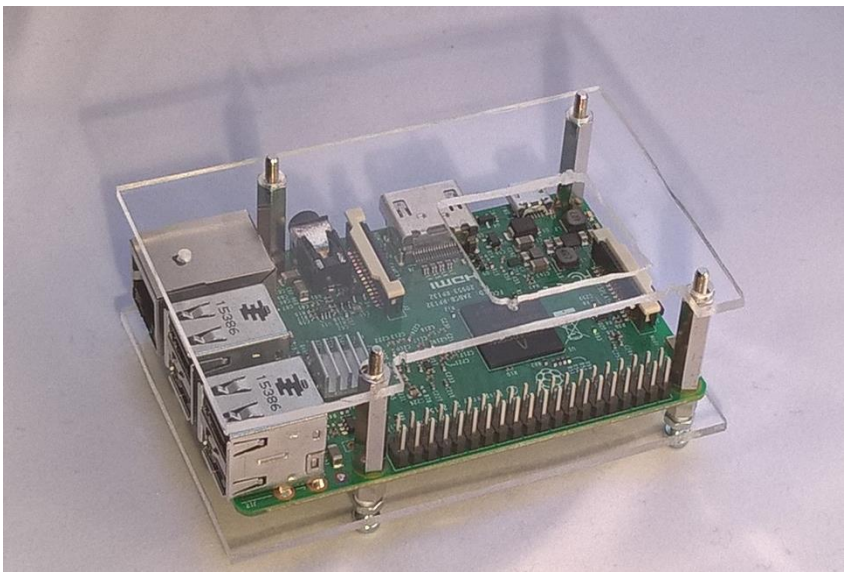

Figure 38: Final outlook for our Raspberry Pi in case with no cooling

4 MODEL APPLICATION - SMART PARK

Two most prominent areas for object detection are person detection and vehicle detection, thus have the most available information about them. Naturally, we will choose one of these two for our test application. Vehicle detection is a little bit more difficult for vast amount of variations in size, shape, colour and many more characteristics. This is ideal as it presents more problems that embedded systems have to overcome.

4.1 Motivation

If car parks want to detect and monitor number of free parking spaces, they have to employ one of many different systems solving this type of a problems. Simpl systems typically use gates which count the number of vehicles entering and leaving from car park, and show calculated number of free spaces. These systems are easily fooled and over time could cumulate error. They have to be reset from time to time to correct actual count of vehicles present. Other types of smart parking solutions use vehicle presence sensor on every parking space, detecting presence using in most cases radio-wave, magnetic induction or laser proximity sensors. In these cases, sensors alone are unreliable and break often. Error will cumulate and calling service for single sensor is just not worth it, so in most cases these are repaired or replaced in batches, meaning longer time for vehicle count correction. However, these systems are probably the only reasonable solution in under-ground car parks or parking lots that are obscured. As object detection systems get more and more robust, another possibility present itself. Using only single or small number of static cameras covering whole car park. They are much cheaper and faster to assemble. Cameras will not be accessible for people in car park, as they will be mounted in higher positions. The device count in most cases correlates with frequency of reparations. This should lower error in prediction of correct free park spaces. Our application will attempt to replace other systems with only a single camera monitoring place. Testing will be done on our own car park and images from internet.

4.2 Specification of the problem

Our testing application will be used in a small local garden centre with car park problems. Most of the time, there is only a few vehicles present. However, during the peak hours or weekends, there are way too many cars and cooperating them is difficult. Images below shows car park in the morning and during peak hours. People are parking chaotically and are often blocking others from leaving during the peak hours. There even has to be a supervising employee cooperating them (wearing green vest in image 38).

Figure 39: Car park in the morning (left) and later during peak hours (right)

Using gate barriers would be counterproductive, as there are 4 roads, from which cars could get to this car park. Not all cars will stop there, some of them are only passing. Because space is not specifically divided for parking spaces, vehicles are not parked always on the same places. Using spot presence detectors would be thus pointless. We will have to use camera or other similar system.

This system should be able to correctly detect presence of vehicles in image and show statistics about approximate park spaces. When car park is full, redirect incoming people to second smaller car park from the other side of the building. Beside this, system should be able to show basic statistics about daily traffic.

5 DEVICE CONFIGURATION

In this chapter, basic configuration of RPI will be briefly described in steps. We won't go into much detail, because it's not as necessary. All steps were already described in other materials. RPI setup was discussed in detail in our older paper [51] where RPI was used for image processing. Details for installation of SW later in text will have references to full step by step detailed guides.

5.1 Brief device setup

RPi is small pocket-size device with enough power for this application. This small embedded has almost all contained on a board, we need only a few things for configuration and a fraction of them for use in model application. For our model application in everyday use, RPi requires:

- Power supply adapter 5V/2.5A
- Ethernet cable with internet connection
- SD card - at least 16Bg (32Gb was used)
- Camera Module v2
- Preferably board case or something similar
- While configuration, we will need additionally:
- Keyboard and mouse
- Monitor with HDMI connection
- at least 1x USB with min 2 GB space

For this paper, we will use SanDisk 32GB microSDHC Extreme 90MB/s, UHS-I card, as this card excel in download/upload speed on official test site for RPI SD cards [57] with specifications:

Reading: 22.8MB/s
Writing: 25.2MB/s

Before we start with configuration, we have to copy OS image from official Raspberry Pi site onto microSD card. We can use one of many free software like Win32 Disk Imager. Object detection expects visualization of results, so we have to choose OS with desktop support. We have chosen OS Raspbian Stretch with Desktop as it has better support for our platform. Once we have copied OS image to SD card, we can insert it into RPI. We will also connect Camera Module v2, keyboard, mouse, HDMI connector from monitor and ethernet cable. Internet connection is needed as we will be downloading a lot of data. After we have all peripheries and connectors connected, we can connect power supply. RPI will start instantly booting up and after a while, we should see initialization and later the desktop. Then we have to configure connection options. We can do that by modifying raspi-config file, which is used while booting up.

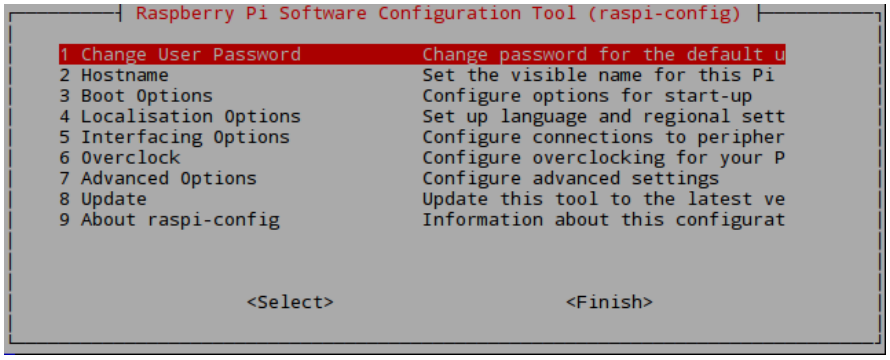

Figure 40: RPI config main-menu [51]

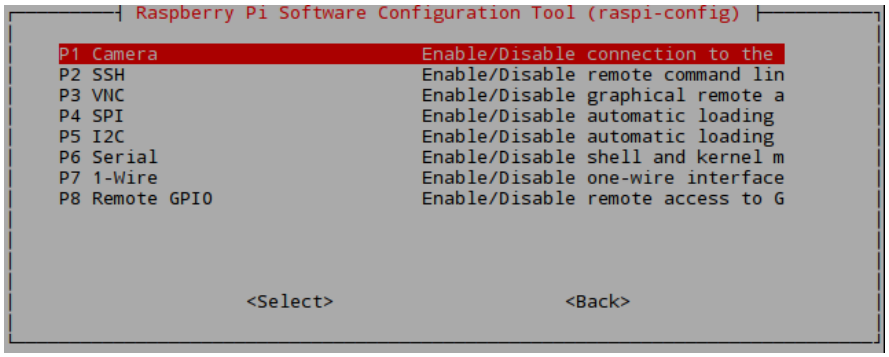

Figure 41: Configuring interfaces [51]

In Interface Options we will allow camera, SSH and VNC. Then we have to use Expand Filesystem in Advanced options, because RPI won't use whole SD card by default. After all is done, we will reboot the RPI and wait for desktop again.When system boots up again, we have to connect to WiFi or cable and update and upgrade RPI. This will take some time. Then we will try to connect from remote PC using VNC. Because we will be using a lot of GitHub source code, we have to download and install Github.

When it comes to software like Tensorflow, we have to install quite a few prerequisites [58, 59]. Configuration and installation of Tensorflow might be problematic mainly because we can't simply install it on our device. We have to build it first using Bazel, then build Tensorflow and after that, we can finally install it. Step by step tutorial from [59] could be summed as:

1. **Installing necessary packages that are needed for Tensorflow installation:**
   `wheel, gcc, c++, swig`

2. **Setting up USB disk as additional swap space.**
   Because RPI contains only 1GB RAM and 100MB swap space, it's not enough for Tensorflow installation. We have to manually set up swap space on USB. We have to connect at least one 2GB+ USB.

3. **Compilation and installation of Bazel**

   Bazel is free build system that is necessary for Tensorflow building from source code. We have to keep in mind that not all versions are currently supported and work correctly together. RPi should officially support Tensorflow v1.9, but we were not able to successfully install it, we have to settle with older version. We will be using latest Tensorflow v1.5 which is 2 version lower and around 6 months older than v1.9. At this time, latest known compatible version of Bazel and Tensorflow v1.5 is Bazel v 0.8.0. If we use any other combination, we might run into errors later in process.

4. **Compilation and installation of Tensorflow**

   We have to use Bazel to compile and build Tensorflow. After we are done with installation, we have to remove extra swap space from our system or we might not be able to boot up after reboot, if USB is removed. After we are done, we have to install few more dependency packages/tools/applications:

   ```
   libatlas-base-dev ,   cython,          pillow,     lxml,
   jupyter,              matplotlib,      python-tk
   ```

5. **OpenCV installation:**

   Tensorflow typically uses matplotlib, but OpenCV seems to be less error prone. It has greater support but configuring it with all other installations to work correctly might be harder than expected. In this work we will use OpenCV. First, we need to install dependencies required and then we can install current latest version:

   ```
   libjpeg-dev,    libtiff5-dev,    libjasper-dev, libpng12-dev,
   libavcodec-dev, libavformat-dev, libswscale-dev, libv4l-dev,
   gfortran,       libxvidcore-dev, libx264-dev,  qt4-dev-tools
   ```

   One important note is to install OpenCV-contrib with the same version as our OpenCV installation. OpenCV-contrib contains many algorithms, that were a part of OpenCV in the past, but then they were split up.

6. **Compilation and installation of Protobuf**

   Protobuf is a package that implements Google's Protocol Buffer data format. There is currently no direct installation available so we have to compile and then install it from source. This process may take up to two hours. Then we will have to wait for another two hours after using command make. Additionally, some python path modifications are required. For detailed info, please refer to [59].

7. **Making working directory for Tensorflow and downloading models**
   Create a new file in your home directory called tensorflow1. Then we have to download all models from Git and update PYTHONPATH variable [59].

8. **Testing Tensorflow installation**
   We have quite extensive list of all available models used for inference from Tensorflow detection model zoo [60]. Model zoo is Google's collection of pre-trained object-detection models with various levels of speed and accuracy. Current categories are:
   - COCO-trained models
   - Kitti-trained models
   - Open Images-trained models
   - iNaturalist Species-trained models
   - AVA v2.1-trained models

   There are currently 34 trained models. But as we are using older version of TensorFlow, we have to use one of older Github commits for version 1.5. For testing purposes, we will use *ssdlite_mobilenet_v2_coco* from Tensorflow setup guide.

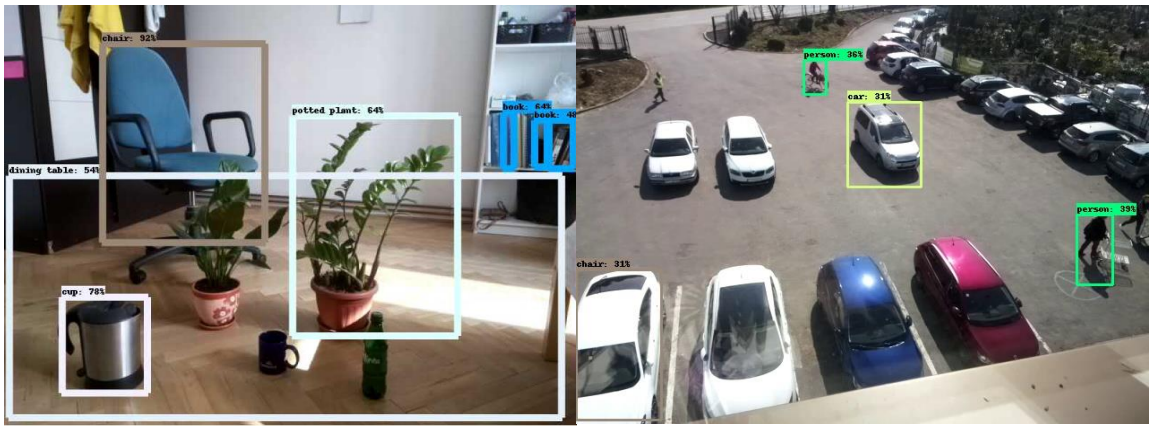

Figure 42: Testing ssdlite_mobilenet_v2_coco model using Tensorflow

As we can see, object detector trained on 90 classes works. Detections are not the best, but that's trade of for a NN with such a small size. Second test image shows, that using general detector in our specific application is insufficient as only a small fraction of cars is detected.

5.2 CPU overclocking

In order to gain as much from this tiny device, we have to improve every small detail. For our application, mainly CPU frequency is important. RPI has CPU frequency of only 1.2GHz, we will aim to get a little bit more. GPU is not as important in our case, as RPi does not support controlled direct usage. Results before and after overclocking were tested using *sysbech:*

```
sysbench --test=cpu --cpu-max-prime=1000 --num-threads=4 run
```

Which basically computes all prime numbers up to number specified, in our case 1000. Stability testing was done using all NN mentioned in later sections. Maximum stable overclock configuration was found to be for our RPi:

```
arm_freq=1260
gpu_freq=500
sdram_freq=500
over_voltage=4
total_mem=1024
sdram_schmoo=0x02000020
sdram_over_voltage=2
force_turbo=1
boot_delay=1
```

Although lowest stable CPU frequency was reported to be 1300MHz and some users were able to get as much as 1500MHz, 1260 was our maximum when using object detection with consistent 100% CPU usage for around 2 hours acting as stability testing (incorporating active cooling from next section). Everything more would freeze after some time and require manual hard reset. Every device is different, same RPi models from the same batch might easily have 200MHz difference in maximal overclock CPU frequency.

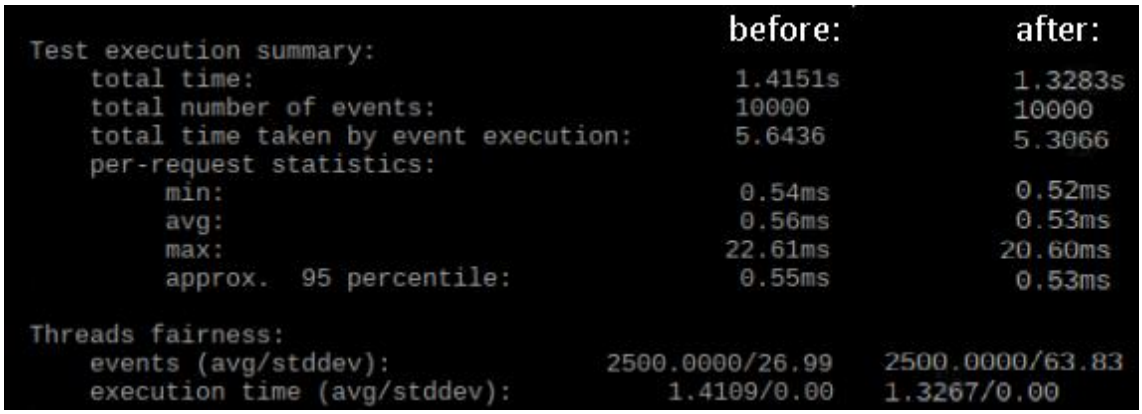

Figure 43: Results from command *sysbench* before and after overclocking

From comparing statistics from before and after, we can conclude that the total time was 6.1% shorter. We will slightly leap into next chapter and show the difference before and after overclocking on total pass-through duration for different models. We have gained few percent decrease in pass-through duration. We have no data for models *ssdlite_mobilenet_v2_coco* and *embedded_ssd_mobilenet_v1_coco* before overclocking, as they were added later.

Table 9: Model results before and after overclocking

	No overclock		Overclock		Time saved [%]
	sec/ image [s]	Total time [s]	sec/ image [s]	Total time [s]	
ssdlite_mobilenet_v2_coco (PM)	0.75	74.57	0.71	71.16	4.6
ssdlite_mobilenet_v2_coco	N/A	N/A	1.11	111.36	N/A
ssd_inception_v2_coco	1.74	173.53	1.58	158.54	8.6
ssd_mobilenet_v1_coco	0.77	76.96	0.65	65.87	14.4
embedded_ssd_mobilenet_v1_coco	N/A	N/A	0.13	13.24	N/A
faster_rcnn_inception_v2_coco	17.57	1757.35	16.79	1678.67	4.5
faster_rcnn_resnet50_coco	68.84	6884.05	67.43	6742.57	2.0

Notes: **PM** = Pretrained model. General object detector without additional training
Total time consists of using object detector on series of 100 images

5.3 Increasing swap space

Loading NN models into RAM might take few hundred MB of memory. When testing different models, we were able to run only a fraction of them. RPi has only 1 GB RAM and 100 MB swap space by default, which is just not enough. We have to increase swap space. There are 3 most common ways how to do that:

- using swap space on SD card
- using swap space on SSD/HDD or USB
- using ZRAM/ZSWAP

As we are going to use a lot of read/write operations, using swap space will certainly decrease lifespan of media that holds it. This is why we should not use our SD card with OS. Only a single data corruption could cause system crashes in the future.

On the other hand, using SSD or HDD is not optimal as we don't need that much space, because we are not going to store that much data. Also, using them connected to RPI would mean either separate power supply or powering them from RPi, and that would be counterproductive as it would dramatically increase total power consumption and size of the system. On the other hand, using USB stick could be beneficial as it would shift wear and tear from main SD card and increase lifespan of our device. USB sticks are cheaper in comparison with SD cards and their replacement is easy. They are a lot slower though. RPi has only USB2.0 ports which would make data transfer unnecessary slow.

ZRAM is a special compressing module for linux kernels, that uses compressed block device in RAM, in which paging takes place until its necessary to use the swap. ZSWAP is lightweight cache with applied compression for swap pages. Pages that are about to get swapped are compressed and stored into dynamically allocated RAM-based memory pool. [61]

While testing which option would be the best, only microSD card swap and USB swap place were successful. ZRAM/ZSWAP kept freezing RPi when a lot of data had to be stored there. Tensorflow models are on itself very large in size. Test confirmed our expectations. MicroSD card is slightly faster and thus may allow us to obtain a small FPS boost. For testing purposes, we will be using microSD Swap to get the most from this application and later, when device will be in everyday use USB as we don't want to corrupt data on our microSD card. If we choose smaller model, we might not need to use this swap space often anyway, just when initializing model or using other RAM expensive application in the background.

Table 10: Model results for different swap locations

	USB Swap		microSD swap		Time saved with microSD Swap [s]	RAM used [MB]
	sec/ image [s]	Total time [s]	sec/ image [s]	Total time [s]		
ssdlite_mobilenet_v2_coco (PM)	0.71	71.16	0.67	66.90	5.98	31
ssdlite_mobilenet_v2_coco	1.13	112.69	1.11	111.36	1.18	314
ssd_inception_v2_coco	1.62	162.19	1.58	158.54	2.25	610
ssd_mobilenet_v1_coco	0.65	65.87	0.63	63.16	4.11	540
embedded_ssd_mobilenet_v1_coco	0.13	13.24	0.13	12.50	5.59	138
faster_rcnn_inception_v2_coco	17.08	1707.85	16.79	1678.67	1.70	594
faster_rcnn_resnet50_coco	67.43	6742.57	66.93	6692.79	0.73	743

Notes: **PM** = Pretrained model. General object detector without additional training
Total time consists of using object detector on series of 100 images

RAM used represents average RAM used by a model (without inclusion of background processes), not maximum RAM used. We have to take into consideration that average RAM without running any application (IDLE mode) is around 120-150MB, with disabled desktop around 50MB. Also, there is a significant peak while initializing session. If there are other processes active, used RAM could easily get over 1 GB. Keeping USB or microSD swap space activated at all times might save us from lagging and freezing and allow us to even run certain model.

5.4 Increasing SD Card read/write speed

RPi uses 50 MHz as a SD card clock by default. This is to ensure compatibility with all types of SD cards. However, we are using high-end type of SD card that is capable of 100 MHz. We can try to increase SD card clock speed. This should decrease time for reading and writing images to memory but mainly decrease time needed to load model into memory. Note that this process may reduce lifespan of SD card. Speed tests were performed before and after overclocking of the SD card clock using copy commands (where bs is block size and count represent number of blocks to be copied). Firstly, we are writing on SD card data from /dev/zero folder, which exist only after booting and is deleted at shutdown. Files in this file are loaded into RPi memory, not SD card. Second part reads data stored in home directory that are stored in SD card and copy them to RPi memory. Write speed could be tested issuing following command to terminal:

```
dd if=/dev/zero of=~/test.tmp bs=500K count=1024
```

Read speed likewise, with first clearing cache, as it might skew results:

```
dd if=~/test.tmp of=/dev/null bs=500K count=1024
```

We were able to gain around 55 % speed increase for write operations and 99 % for read operations. This should come handy in cases where we will be recording to memory.

Table 11: SD card speed test before and after overclocking

	Default 50 MHz speed	Overclocked 100 MHz	Improvement
Write speed	23.8 MB/s	37.0 MB/s	55.46 %
Read speed	23.6 MB/s	46.1 MB/s	95.33 %

Testing duration time for a model loading into a memory showed up to around 12% duration decrease. Model load time represent all time needed for a specified model to be loaded from SD card into RAM. Image pass represents image being loaded from SD card, passed through object detector and resulted image saved back to SD card.

Table 12: Effect of overclocking SD card speed on model loading time

	Default 50 MHz speed		Overclocked 100 MHz speed		Model loading time Improvement [%]
	Model load time [s]	Single image pass [s]	Model load time [s]	Single image pass [s]	
ssdlite_mobilenet_v2_coco (PM)	26	1.02	25	1.02	3.85
ssdlite_mobilenet_v2_coco	13	1.44	12	1.41	7.69
ssd_inception_v2_coco	16	2.09	14	2.07	12.50
ssd_mobilenet_v1_coco	13	0.92	12	0.90	7.69
embedded_ssd_mobilenet_v1_coco	10	0.41	9	0.41	10
faster_rcnn_inception_v2_coco	14	17.83	14	17.66	0
faster_rcnn_resnet50_coco	23	68.96	21	68.29	8.69

5.5 Overheating problems

Object detection is computationally demanding task. Efficiently implemented algorithms will most likely use all available CPU. This creates problem for devices with poor cooling, as temperature could potentially limit speed or even worse, permanently damage device. We have to either increase cooling enough to keep CPU cool or lower amount of computations, as device will have more time for cooling. Only the first solution makes sense in terms of efficiency. According to official Raspberry Pi hardware specifications [62], maximum stable temperature is around 60 °C (soft limit). If CPU reaches temperatures between 80 °C to 85 °C a warning temperature thermometer icon should show up. If the temperature reaches around 85 °C, device should start throttling down heavily. According to documentation, devices chip frequency and current input will drop as way of restricting CPU usage and therefore let RPi cool slightly, after cooling enough, current and frequency will increase again. However, our device got to unresponsive state every time the temperature limits were tested at temperature slightly above 80 °C, with no other way to reset it other than powering it off and on (reset button is not soldered onto the board by default).

When tested, TensorFlow shows CPU usage at approximately 100% all the time. This means the RPi might get hot really quickly. For purposes of testing temperature stability, our python script was created, which can be used from file containing this Python script in command line:

```
$ python3    DP_CPU_core_test.py    { No-of-cores }
```

where *No-of-cores* represent integer value of cores to use. RPi is Quad core which means that 4 cores are available. Script creates requested number of processes which count up counter infinitely, until aborted.

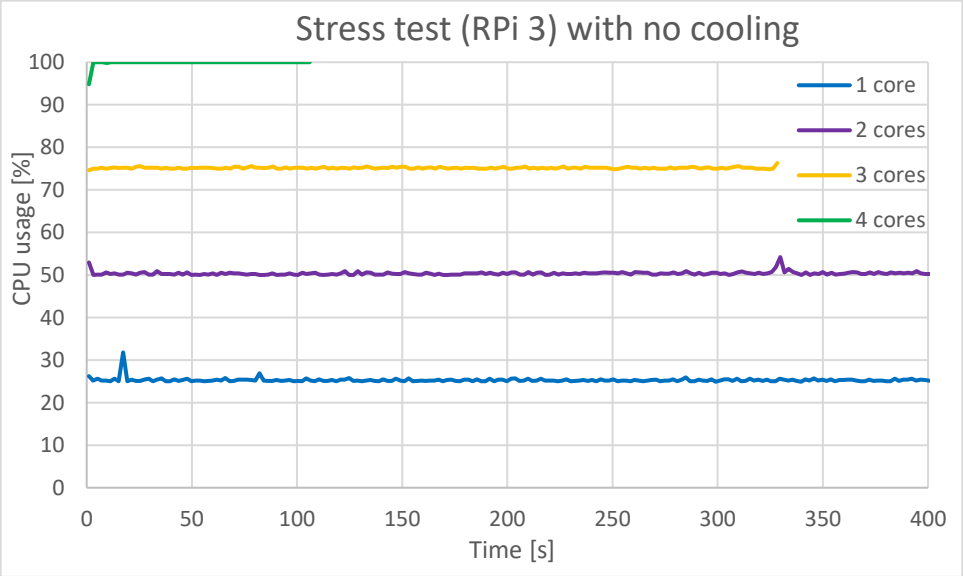

Figure 44: Stress test - CPU usage

With no workload on CPU, average usage is somewhere in range from 1-3 %, this represents IDLE mode with no user applications running and no peripheries connections, only graphical display mode for display output. IDLE mode was not tested as temperature stayed stable at 40-ish range. Picture above demonstrates CPU usage for all 4 tests with different number of active cores. Occasional spikes represent other active processes.

Tests were done sequentially for 0, 1, 2 ,3 and finally all 4 cores. Stable temperatures are shown in table 13. As could be seen from graph in picture 44, starting temperature for all tests was around 41 °C. This represents average temperature of RPi with no cooling attached and in IDLE mode. All tests were done at room temperature. Note that tests were interrupted if temperature reached 77.5°C.

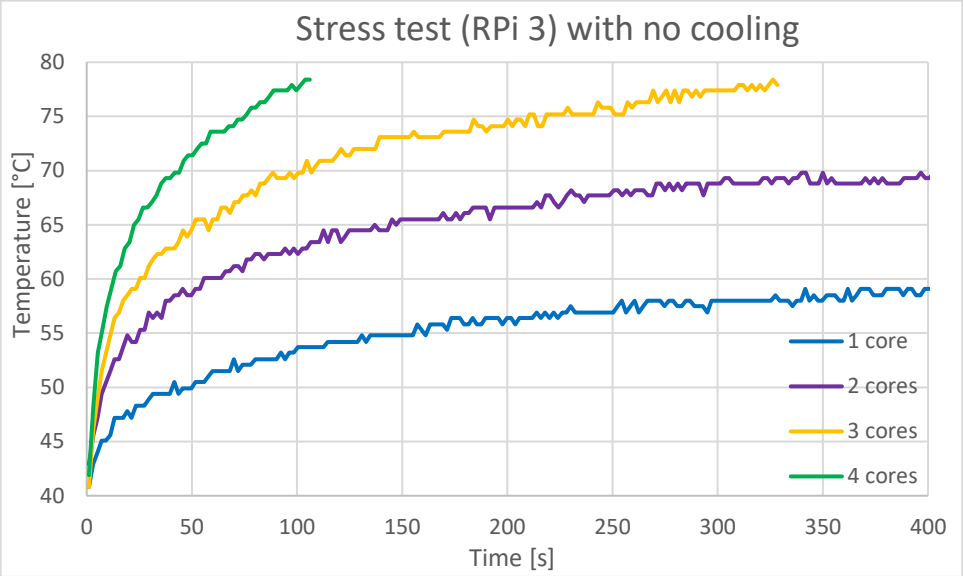

Figure 45: Stress testing without cooling

Topmost temperature for optimal performance is under 60 °C. TensorFlow uses all four cores with CPU usage of 100% which will provide around 1.5min object detection window, after which RPi freezes. RPi definitely needs additional cooling. Image showing RPi without any cooler is in image 38.

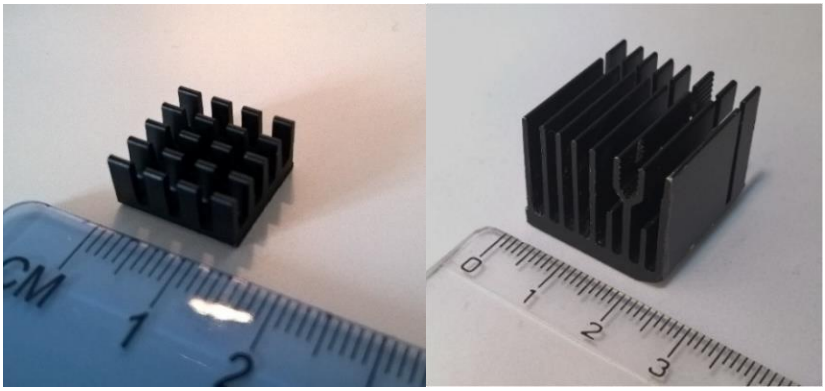

Figure 46: Official Raspberry Pi 3 Model B heatsink and our custom made

Official small copper heatsink for RPi (Figure 46 left) with dimensions 14x14x6 mm shows only a small difference in temperature. We still get over 80°C in about 2 minutes. Used thermo tape which came with copper heatsink did not work very well. Temperature for 2 cores with 50% CPU usage stopped at around 66°C. That means we would gain only around 5°C decrease in temperature.

After adding bigger aluminium passive heatsink (Figure 46 right) with dimensions 20x20x16 mm and thermal paste, another stress test series was performed. Next graph shows how much of a difference different heatsinks could make. More than 10°C difference for 50% CPU usage (measured at 250s). Maximal temperature for full 100% CPU usage stabilises after approximately 10 minutes at around 70°C which is still a lot.

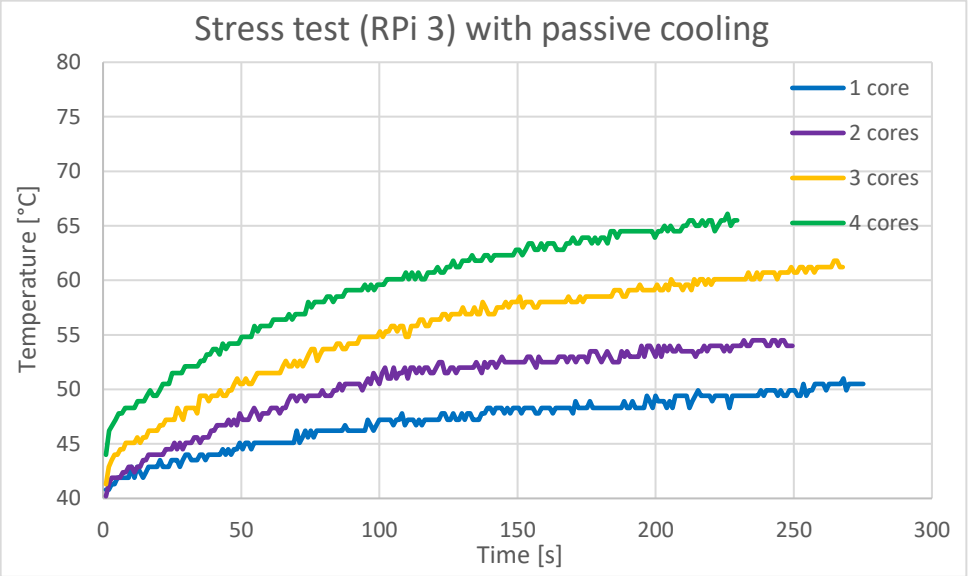

Figure 47: Stress testing with large passive aluminium heatsink

Significant improvement could be seen from graph in picture 47, which represents results of stress test with added active 5V cooling. Temperature drop of 20°C in compare with passive cooling and more than 30° in comparison with no cooling. Temperatures in 50-ish range are ideal. We should not be able to get over 60 °C (soft limit), even in very hot days. Settled temperatures for all combinations are in table 13.

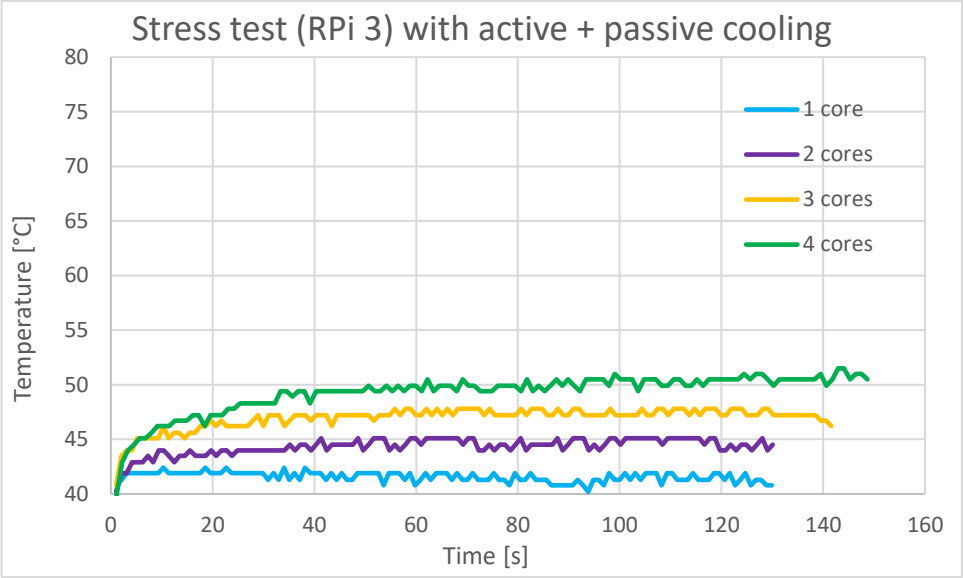

Figure 48: Stress testing with large passive aluminium heatsink and fan

Table 13: Stress test for RPi3 Stable temperatures

Stress test for RPi 3 Model B - Stable temperatures						
Number of cores used [-]		IDLE	1 core	2 cores	3 cores	4 cores
CPU usage [%]		1 - 3	25	50	75	100
Stable temperature [°C]	No cooling	41	61	73	>77.9	>77.9
	Passive cooling	41	52	57	64	70
	Passive + active cooling	32	42	45	48	51

Final appearance of case with attached passive and active cooling is in picture 49. There were later added plastic plates at sides of active fan to help with air flow regulation.

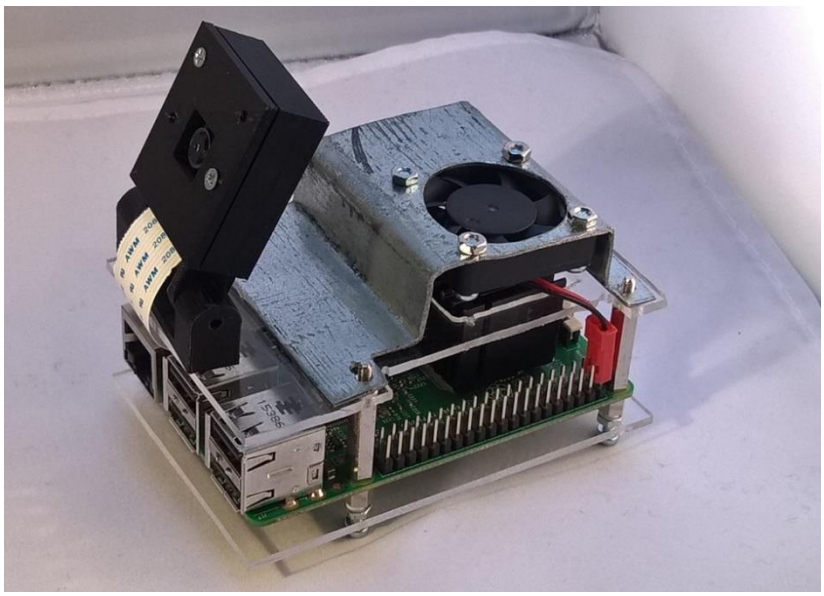

Figure 49: Final appearance for our Raspberry Pi with heatsink and active cooler

6 MODEL TRAINING

Since we are using Tensorflow v1.5.0 on RPi, we have to use the same version while training. If we don't, we may run into compatibility problems. According to CUDAs website and documentation [63], there has to be installed pack of predefined SW with specific versions for Tensorflow v1.5.0 using Windows 10. Compatible SW versions are as follow:

- Visual Studio 2015 14.0 (RTW and updates 1, 2, and 3)
- CUDA 9.0.176
- cuDNN 7
- Python 3.5 or 3.6

Few different combinations were tried before with no luck, since it is remarkably difficult to get right drivers installed after newer versions have been installed before. It is best to start with clean install of Windows 10. In case of clean install of Windows 10, installation for CUDA and other support drivers is straightforward. They all are available online and downloadable in *.exe* format, installed with default settings. Detailed installation guide is written in [63] by searching in directories for installation guide. At first, only CPU version was successfully installed (Intel i5-323M CPU @2.60GHz/4Gb). However, it turned out to be absolutely inapplicable, as single step took around 9 seconds and according to setup installation, there is needed at least 50000 steps (batch size of 1 = 1 image per step) for reasonable output, which would take around 5-6 days in total, with no other activity on this computer. This process could take easily more than two weeks for satisfactory output.

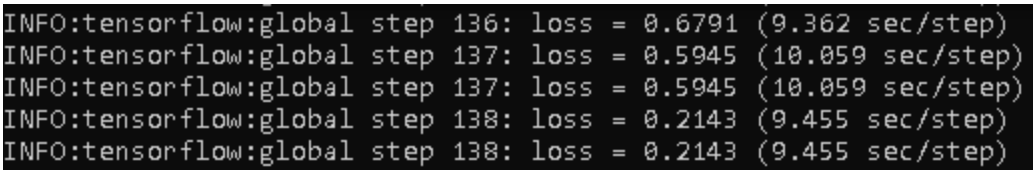

Figure 50: Training process on CPU

After successfully installing all prerequisites and all dependencies on different PC, including Tensorflow v1.5.0, in accordance with [63], we could start with second part of the training process, which is image data gathering. Specifications for this PC are as follow:

CPU:

Intel i5-4570 CPU 3.2GHz / RAM - 8Gb

GPU:

NVIDIA GeForce GTX 1050 Ti (Compute capability 6.1) / RAM - 8Gb

Using GPU is definitely preferable option. We were able to decrease time for a single step approximately to 1/30.

```
INFO:tensorflow:global step 144: loss = 0.5676 (0.313 sec/step)
INFO:tensorflow:global step 145: loss = 0.9293 (0.313 sec/step)
INFO:tensorflow:global step 146: loss = 0.9372 (0.328 sec/step)
INFO:tensorflow:global step 147: loss = 0.7940 (0.313 sec/step)
```

Figure 51:Training process on GPU

6.1 Data gathering

Second part consist of gathering enough data for our training and testing dataset. In our case, it will be slightly easier as we will be mostly using pretrained models. If we want to train our pretrained object detector with reasonable precision, we have to gather at least few hundred images, ideally thousands with objects in them. For this purpose, we have mounted our RPI with Camera module v2 in place, where we expected it to be in future and took pictures from RPi camera while monitoring car park. Pictures were taken every few minutes on average for few days when a car park was opened. We were able to gather 720 pictures in total with different number of vehicles in them.

Half of gathered images pictures was horizontally mirrored, because we want to preserve angles, distance and size of vehicles in images, but we don’t want our object detector to be trained only on specific positions on our car park. We also used 200 car pictures from google and ILSVRC'16 dataset, with view on vehicles from slightly above. Ideally, we would want more images from other sources, but it is quite complicated to find images of multiple vehicles in a place with similar conditions. There could be found thousands of car images taken from sides but almost none from above. We were able to gather 920 images in total.

6.2 Labelling process

Third part consist of hand labelling all obtained images using free program called labelImg [64]. Labelling is significantly tedious process, which took around 22 hours in total. We had to draw bounding boxes for every image in our dataset. There were between 1 to 30 cars per image, totalling in 9204 cars captured in them. Next image shows labelling process. After all images are labelled, we divide them randomly in 8:2 ratio for training and testing image set.

Figure 52: Labelling process in program labelImg

6.3 Choosing models for training

As we have already discussed in chapter 5.1, there are multiple available neural network models with different inference mechanisms behind them and architectures. In order to use the best neural network, we have to try multiple of them that seem reasonable for given application. Picture below shows pretrained compatible neural network models for Tensorflow 1.5 and version-compatible models from *Tensorflow Model Zoo* [60].

We will be using mostly pretrained models (Transfer learning), as these are well tested and are able to generalize and learn a lot of features in images. Worth noting is that we will use NN pretrained on the same dataset - COCO. Different datasets have distinct way to evaluate performance for models. This enables us to pick relatively the best model for our application. All available pretrained models are listed below ([60]) :

COCO-trained models {#coco-models}

Model name	Speed (ms)	COCO mAP[^1]	Outputs
ssd_mobilenet_v1_coco	30	21	Boxes
ssd_inception_v2_coco	42	24	Boxes
faster_rcnn_inception_v2_coco	58	28	Boxes
faster_rcnn_resnet50_coco	89	30	Boxes
faster_rcnn_resnet50_lowproposals_coco	64		Boxes
rfcn_resnet101_coco	92	30	Boxes
faster_rcnn_resnet101_coco	106	32	Boxes
faster_rcnn_resnet101_lowproposals_coco	82		Boxes
faster_rcnn_inception_resnet_v2_atrous_coco	620	37	Boxes
faster_rcnn_inception_resnet_v2_atrous_lowproposals_coco	241		Boxes
faster_rcnn_nas	1833	43	Boxes
faster_rcnn_nas_lowproposals_coco	540		Boxes

Figure 53: COCO - pretrained models [60]

As we will be running our model on computationally weak device, we have to go for smaller and faster architectures in exchange for precision. We have chosen from officially supported NN models these:

- *ssd_mobilenet_v1_coco*
- *ssd_inception_v2_coco*
- *faster_rcnn_inception_v2_coco*
- *faster_rcnn_resnet50_coco*

After looking around in code samples for a while, we have noticed that there are also sample configuration xml files for models not listed in table above, which will allow us to train our models from scratch. From these we included:

- *ssdlite_mobilenet_v2_coco*
- *embedded_ssd_mobilenet_v1_coco*

These two architectures are either not included in Model ZOO at all or are included in future commits, thus pretrained model is not fully compatible with our version of Tensorflow. *Embedded_ssd_mobilenet_v1_coco* is much smaller than others, so ideal for embedded devices, just like could be deduced from its name. *Ssdlite_mobilenet_v2_coco* has available frozen inference graph in examples, and first tests were done using this model. Average FPS was around 1.2, which is not bad for device like this. We will try to obtain similar results.

After downloading and configuring pretrained models, the training process began. Every model has slightly different configuration while training. We tried to use as big batch size as possible. The batch size represents number of samples propagated through the NN in single step, from which is calculated error. Lets' say we have batch size of 5, thus we are training our NN to 5 images at the time in a single step. This means we have to allocate less memory than when using a whole dataset. When we detected any of *'Out of memory'* error, we decreased batch size. In most cases, batch size of 1 or 5 was used, depending on memory requirements for used model. We can visualize training process using *TensorBoard*. Different types of NN have slightly different graphs, but all of them have classification, localization and total loss.

classification loss - represents goodness of classification of object
localization loss - represents goodness of localization in image
total loss - total loss computed from other losses

Next image shows mentioned losses for training of *ssdlite_mobilenet_v2_coco*.

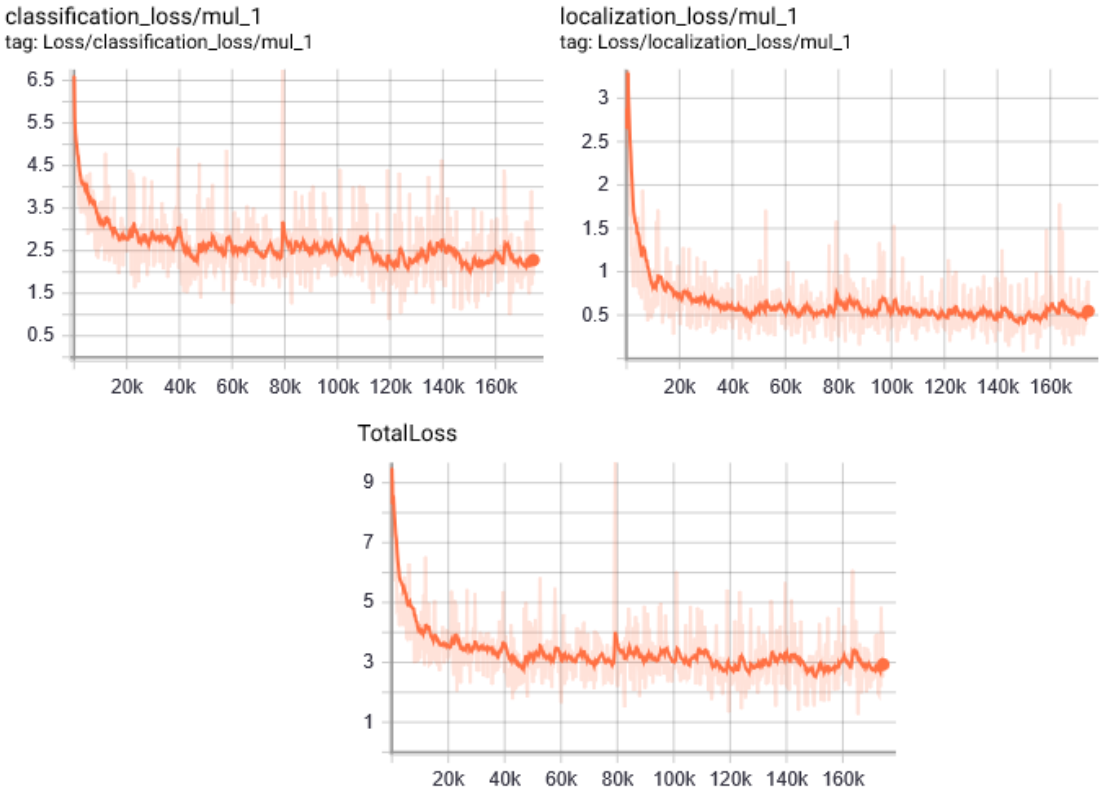

Figure 54: Example of training statistics provided by Tensorboard for *embedded_ssd_mobilenet_v1_coco*

Ideally, we would want to have shown accuracy score too, but this version of Tensorflow and *TensorBoard* does not offer support for that. We have a slightly harder way of telling when to stop training our NN with this version. In our case, all NN were stopped training after they started to plateau or decrease in losses were insignificant. For *embedded_ssd_mobilenet_v1_coco* we had to stop sooner, because the number of false negatives started to dramatically increase (not shown in graphs). **Training graphs for every trained model are included in attachment 1.**

Table 14: Training process statistics

Model name	Batch size	steps	Time/ step	Time/ image	Total time	PM (*1)
ssdlite_mobilenet_v2_coco	4	188k	0.265s	0.066s	23h15m55s	No
embedded_ssd_mobilenet_v1_coco	5	173k	0.453s	0.091s	20h58m17s	No
ssd_inception_v2_coco	5	43k	1.047s	0.209s	4h59m56s	Yes
ssd_mobilenet_v1_coco	5	60k	2.970s	0.594s	10h23m53s	Yes
faster_rcnn_inception_v2_coco	1	104k	0.498s	0.498s	4h15m27s	Yes
faster_rcnn_resnet50_coco	1	45k	0.853s	0.853s	6h55m55s	Yes

(*1) PM - Model was pretrained on COCO dataset, else trained from scratch

Note the discrepancy between number of total steps, time for single step and total time. Apparently, Tensorflow *Time/step* (and *Time/image*) does count only time necessary for pass through, not time needed for supporting functionality so it does not reflect entirely *Time total*.

6.4 Model performance comparison

When evaluating performance of trained model, it is important to choose appropriate tests to match application requirements. Table in figure 53 evaluates performance (tested on Nvidia GeForce GTX TITAN X) using COCO mAP - MSCOCO evaluation protocol [60]. This type of evaluation is not the best for our application, as it uses 12 types of metrics and there are only a few interesting for us. Also, this process is computationally expensive. For the most basic comparison we need only confusion matrix and time duration/ frequency for single pass through NN. Confusion matrix also known as error matrix belongs mostly to the field of machine learning and statistical classification. Contains two dimensions for "real" and "predicted" outputs.

		Actual class	
		Condition positive	**Condition negative**
Predicted class	**Positive prediction**	True Positive	False Positive
	Negative prediction	False Negative	True Negative

Figure 55: Confussion matrix [65]

Information in next section are from [65].
Individual 4 values in confusion matrix are as follow:
True positive (TP) - Object was correctly identified. Correct state.
False positive (FP) - Object was found but there is no object in image.
False negative (FN) - Object was not found, but there is object in picture.
True negative (TN) - Object was not found and there is no object. Correct state.

Besides speed of NN, there are two more parameters we are interested in:
Positive predictive value PPV (Precision)- Describes how many of labelled objects in images are really objects of given class.

$$PPV = \frac{TP}{TP + FP} \tag{7.}$$

True positive rate TPR (Recall or sensitivity)- Describes how many objects are correctly labelled from total number of objects in image.

$$TPR = \frac{TP}{TP + FN} \tag{8.}$$

From calculating Precision and Recall, we have to somehow pick the best model for our application. Beside FPS, it's still two variables that we have to evaluate. We will use **F1 score** for this purpose, as it will effectively reduce our evaluation to only two variables. F-score is used in statistical analysis or binary classification as a measure of a test's accuracy. F-score represents harmonic mean of precision and recall.

$$F1 = 2 \cdot \frac{precision \cdot recall}{precision + recall} \quad (9.)$$

We have created NN benchmark script, that takes for every model 50 images from our car park and 50 random car images from google and does object detection on them. Script creates time statistics and few other parameters from every detection. Later we had to manually check every picture and determine correct image labelling for every model. **Samples from object detection process are included in second part of Attachment 1, not here as they would take too much space.** For reference purposes, there is included general *ssdlite_mobilenet_v2_coco* trained on 90 classes.

Table 15: DL models benchmark - our car park

Name:	TP		FP(*1)		FN		PPV	TPR	F1	fps
	[-]	[%]	[-]	[%]	[-]	[%]	[-]	[-]	[-]	[img/s]
ssdlite_mobilenet_v2_coco (PM)	28	4.21	0	0.00	637	95.79	1.00	0.04	0.08	1.51
ssdlite_mobilenet_v2_coco	506	76.09	64	11.23	158	23.76	0.88	0.76	0.82	0.95
ssd_inception_v2_coco	575	86.47	2	0.35	90	13.53	0.99	0.86	0.92	0.64
ssd_mobilenet_v1_coco	580	87.22	1	0.17	85	12.78	0.99	0.87	0.93	1.62
embedded_ssd_v1_coco	428	64.36	152	26.21	237	35.64	0.73	0.64	0.68	8.35
faster_rcnn_inception_v2_coco	659	99.10	6	0.90	6	0.90	0.99	0.99	0.99	0.06
faster_rcnn_resnet50_coco	654	98.35	17	2.53	11	1.65	0.97	0.98	0.97	0.02

(*1) Percentual false positives were taken as a proportion from all predicted positives

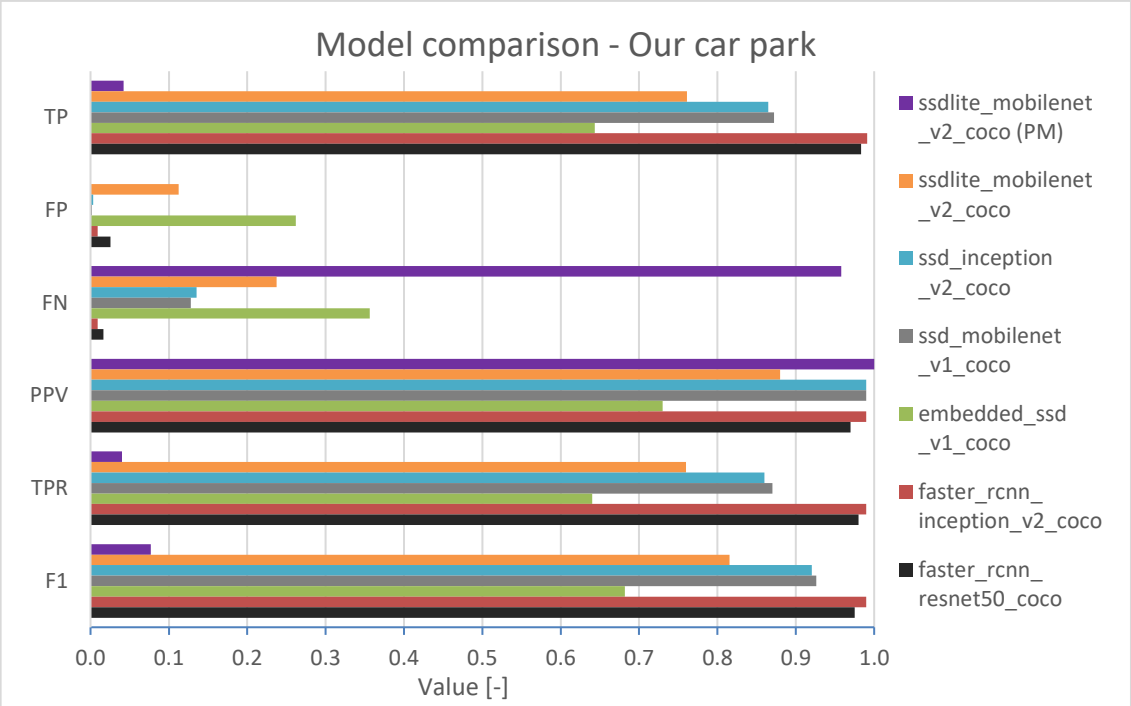

Figure 56: Model comparison - Our car park

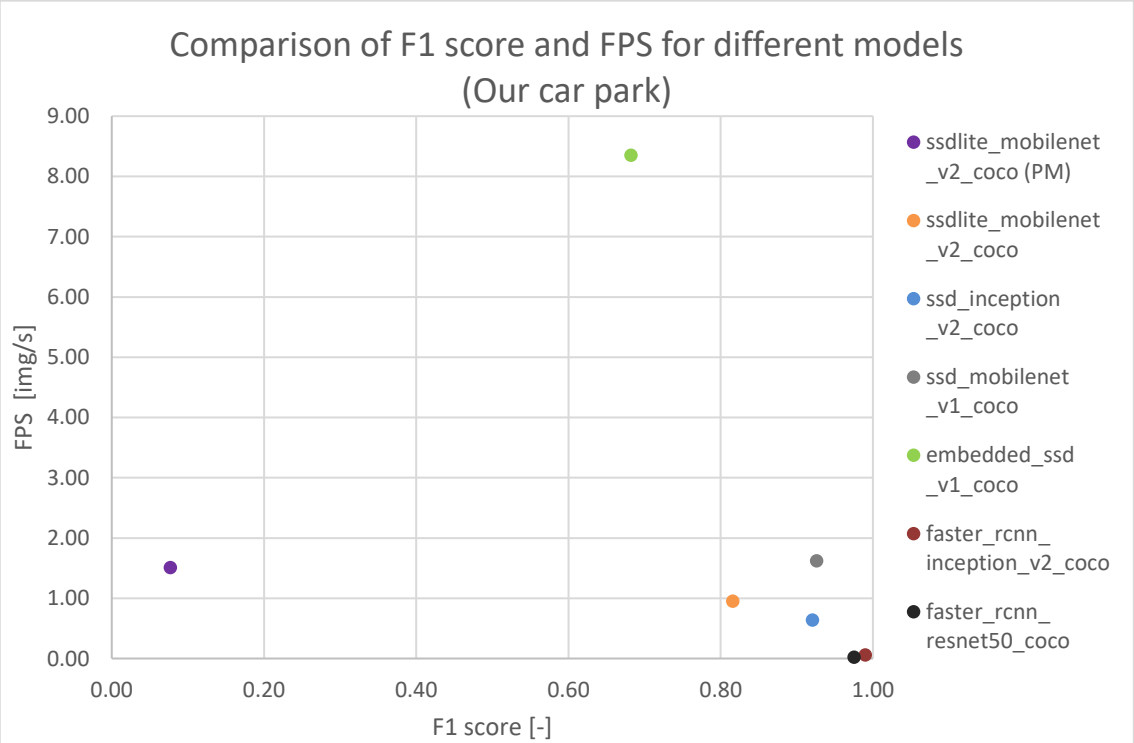

Figure 57: Comparison of F1 score and FPS for different DL models - Our car park

From graphs, we can conclude that the best Precision and Recall have for our specific application *faster_rcnn_inception_v2_coco* and *faster_rcnn_resnet50_coco*. Too bad that they are super slow. On the other hand, if we would look for the fastest model *embedded_ssd_v1_coco* would seem like a fine pick, however this model does not have the best Precision, nor Recall. This model dominates with 152 false detections for car park and 348 for random pictures, which is not a good thing. Next image shows detections using this model.

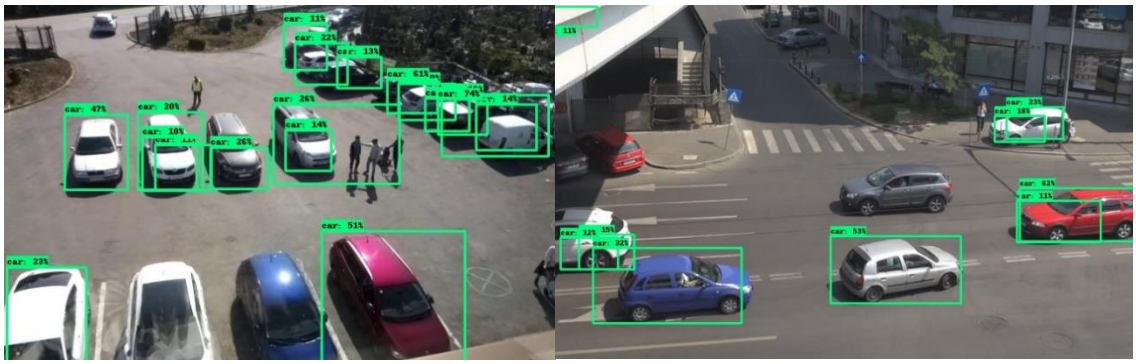

Figure 58: Sample for embedded_ssd_v1_coco detection showing false detections

Model *ssdlite_mobilenet_v2_coco* has a lot of false detections too. Besides these four, other models seem applicable. Looking at F1 score and FPS, we will pick *ssd_mobilenet_v1 _coco*, as it provides the best F1 score and FPS for our application.

Table 16: DL models benchmark - general object detector

Name:	TP		FP(*1)		FN		PPV	TPR	F1	FPS
	[-]	[%]	[-]	[%]	[-]	[%]	[-]	[-]	[-]	[img/s]
ssdlite_mobilenet _v2_coco(PM)	129	43.73	12	8.51	165	56.27	0.91	0.43	0.58	1.48
ssdlite_mobilenet _v2_coco	156	52.88	160	50.63	139	47.12	0.49	0.52	0.50	0.93
ssd_inception_v2 _coco	192	65.08	28	12.73	103	34.92	0.87	0.65	0.74	0.64
ssd_mobilenet_v1 _coco	184	62.37	31	14.42	111	37.63	0.85	0.62	0.72	1.58
embedded_ssd_v1 _coco	151	51.19	348	69.74	144	48.81	0.30	0.51	0.38	7.52
faster_rcnn_incep tion _v2_coco	265	89.83	70	20.90	30	10.17	0.79	0.89	0.84	0.06
faster_rcnn_resne t50 _coco	271	91.86	90	24.93	24	8.14	0.75	0.91	0.82	0.02

(*1) Percentual false positives were taken as proportion from all predicted positives

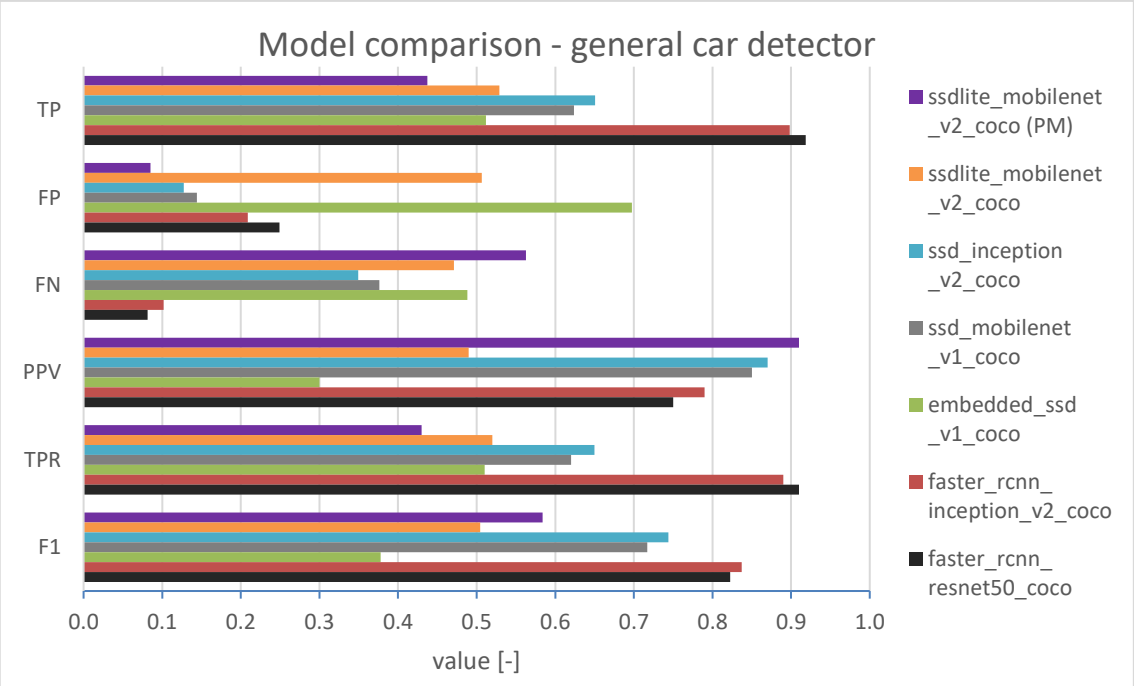

Figure 59: Model comparison - general car detector

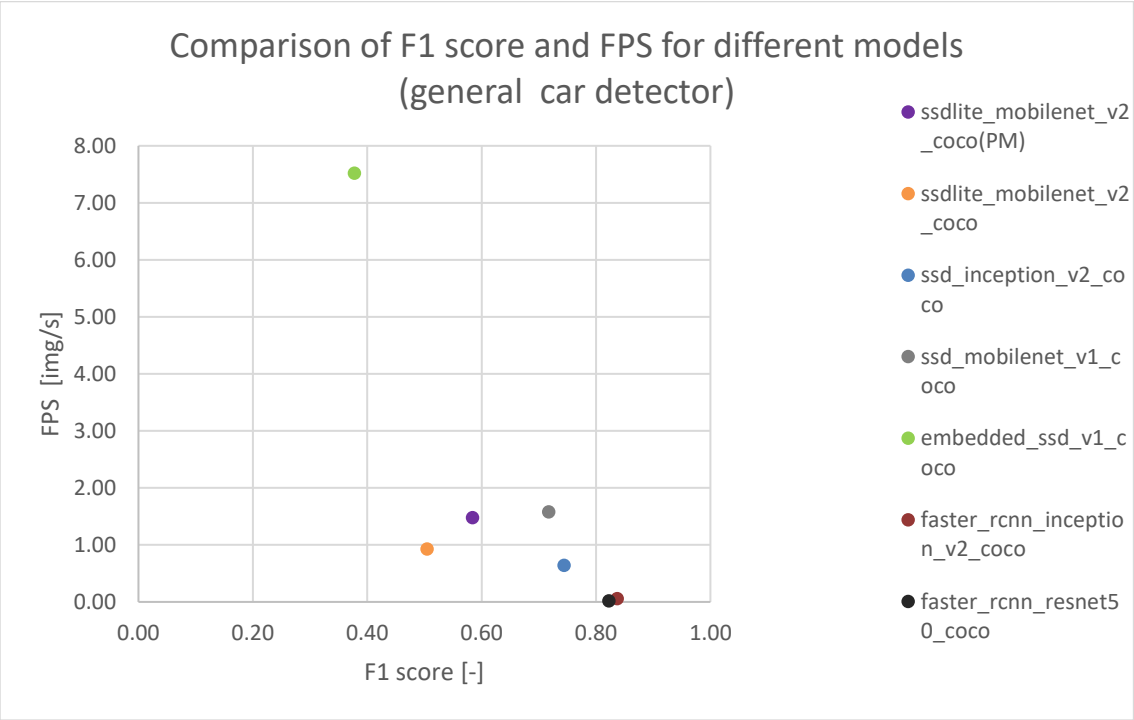

Figure 60: Comparison of F1 score and FPS for different DL models - general car detections

For different applications than ours, *ssd_mobilenet_v1 _coco* and *ssd_inception_v2_coco* seem like a reasonable choice. *embedded_ssd_v1 _coco* may be used in application, which solves easier problem than vehicle detection. Using Faster RCNN on RPi is currently unreasonable, as it takes forever for a single image to get processed.

7 APPLICATION

Our application should be able to be as autonomous, as possible. This is why we added one step before actual monitoring, that allow us to gather enough information about car park. No manual intervention should be required. All programs are written in Python, as Tensorflow currently has primary support in this programming language. C++ is also supported when using C API. However, it is still not fully rewritten. Other programming languages are partially rewritten but still in experimental phase.

7.1 Auto-configuration

If we want to fully understand car park, we have to know just a few specific characteristics. Firstly, we have to know the area, in which vehicles could be present. Secondly, we have to obtain possibly very precise parking place location. Knowing these two, we can always determine number of vehicles in car park and if they are parked in parking spots. To obtain both information, we have divided process of autoconfiguration into two parts - finding accessible space and parking locations. More about these two in upcoming subchapters.

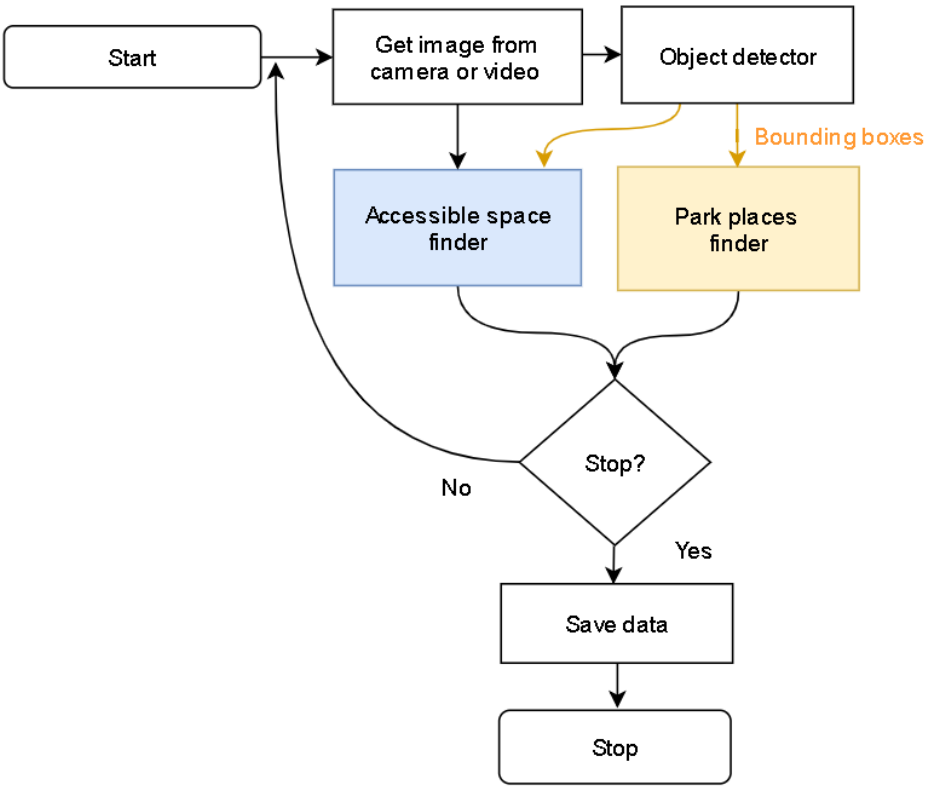

Figure 61: Application: auto-configuration dataflow

7.1.1 Obtaining accessible vehicle space as a mask

Object detection is a demanding task with considerable error rate. Minimizing error with as little additional computations as possible is desired. In our case, for example, we can tell with certainty that vehicles will not stand or ride over kerbstones or into plant fields.

Figure 62: False detections on billboards by car detector

We want to restrict area where cars could be present and area where they can't. There are several ways how to do this but every single one has certain limitations. If we only recorded positions from object detector, we would not get good results as object detector has many false positives on billboards or places near car park. On the other hand, if we recorded only moving objects and map movement of bigger objects, we should obtain map of all places, where vehicles could get. This kind of works too, but we will record movement of other unwanted objects. Employing both techniques should be beneficial. In theory, if we monitor moving objects in video and combine them with output of vehicle detector and record their position over time, we can get all available space, where vehicles can get. It might be helpful when finding parking lines in video too, as we won't look for lines outside car park. Our approach is described in picture 63.

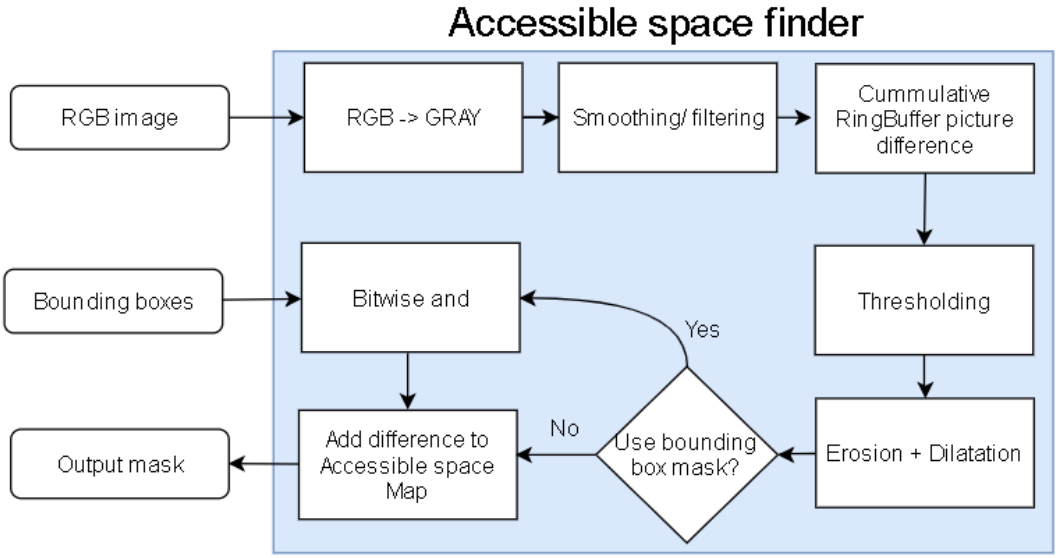

Figure 63: Application: Accessible space finder dataflow

We have used our own implementation of ring buffer for faster and less memory demanding container for our image data. This container does not shift images by one index whenever a new image is added but rather only replaces image at specific incrementing index. This container holds last 5 to 25 images of video capture (depending on FPS) and computes absolute image difference over them.

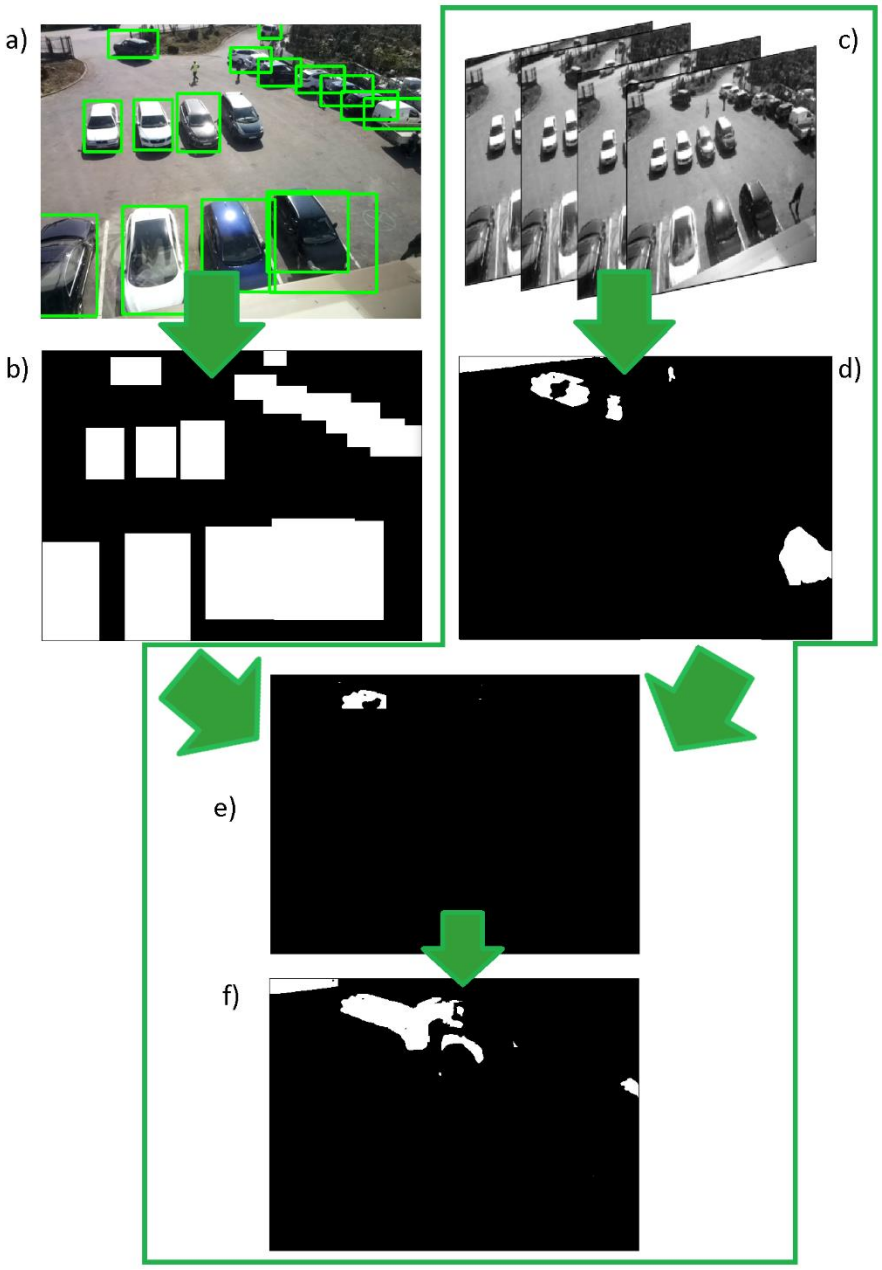

Figure 64: Visualisation for Accessible space mask finding

Image 64 shows these steps in visually more comprehensive format. Images a) and b) represent output from object detector and corresponding image mask. These two are not included in area restricted with green colour, because they are not needed in applications where there is no background movement causing noise. Including them will reduce false

mask writes from movement of different object than vehicles. Image c) represents images holder by our implementation of ring buffer. Absolute pixel difference is computed over all of them and resulting mask is in image d). This should show all places in image, where any object moved in the last number of frames. We can see more white blobs but only one of them represents a real car. Combination of detected movement mask and mask from car detector produces image e) which shows only area of moving car. Adding this information into existing movement map will produce Accessible space mask. Picture below shows average peak hours. There is a lot of movement of both vehicles and people.

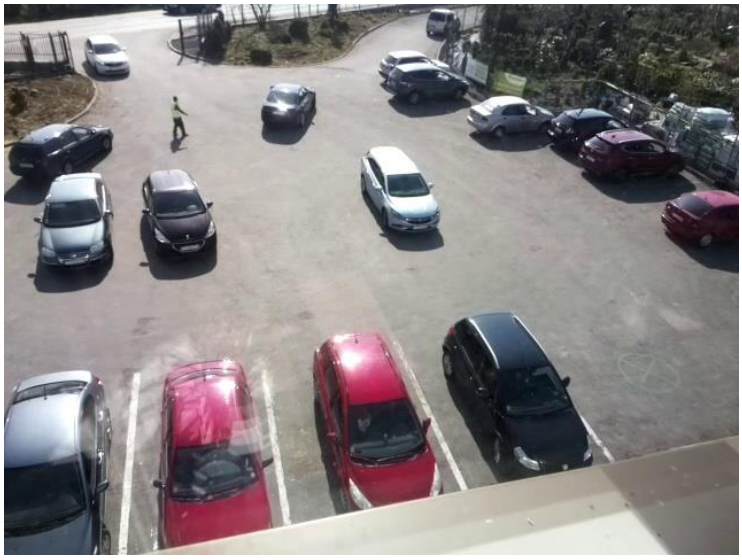

Figure 65: Car park with movement of both cars and people

In picture 66 we can see process of obtaining accessible space mask from moving objects. Images represent obtained mask after 10 seconds, 1 minute, 5 minutes and 30 minutes, respectively. Small particle noise filtering was used to eliminate small spots as could be visible in upper right part of obtained accessible mask. Obtained park space mask shows relatively correct area accessible with vehicle and thus, could be used as mask for line finding and filtering of false detected image localizations.

In applications with no background noise movement, implementation could be a little bit more straightforward. All we have to do is to record movement of all moving objects with more than minimum threshold size and record all their occurrences in video over time. In cases where there is a lot of movement around the car park, this simplification should not be used.

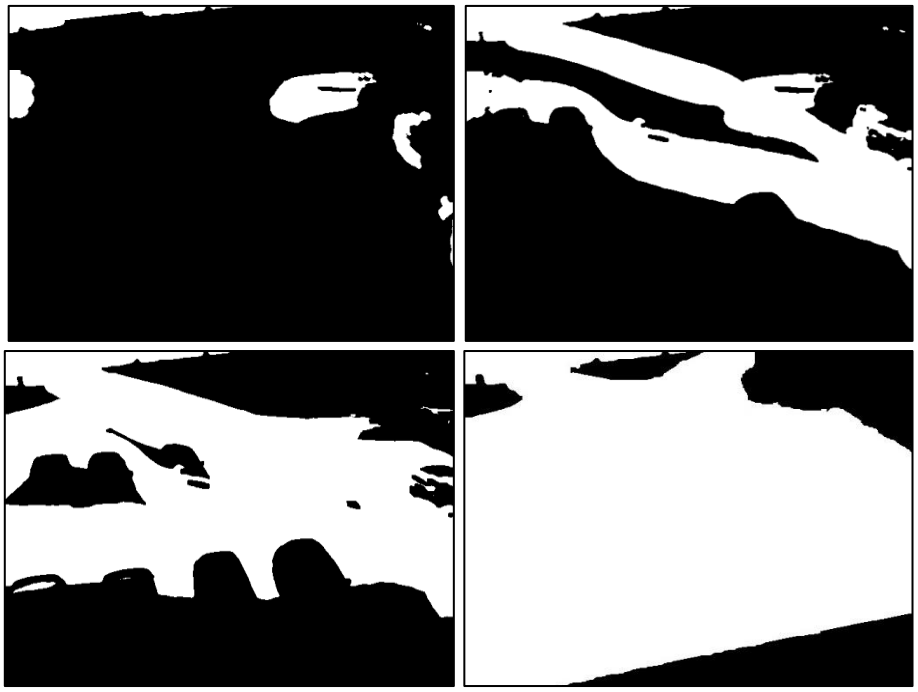

Figure 66: Process of obtaining accessible space mask after 10s, 1m, 5m and 30m

7.1.2 Obtaining park spots locations

If we want to detect and monitor fullness of car park, we have to somehow tell, if park spaces are full or not. This means we should identify all available park spaces and then monitor if they are occupied. First tactics was to detect all line segments and from them estimate parking spaces. Process could be characterized as:

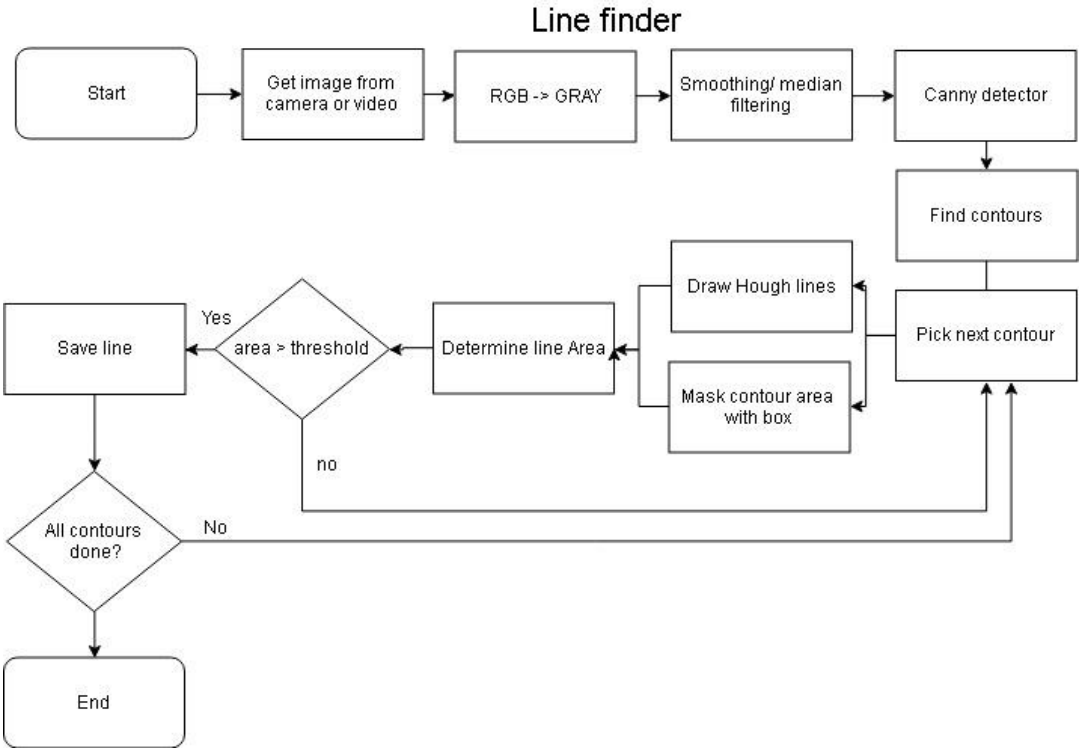

Figure 67: Application: Line finder data flow

Next two images show original image converted to Grayscale and output from Canny detector with all found contours on the right side. Note that in the first picture, there are only 5 lines. Later were more added. Also note that for visualization no Accessible space mask was used, to highlight possible error, when skipping this part.

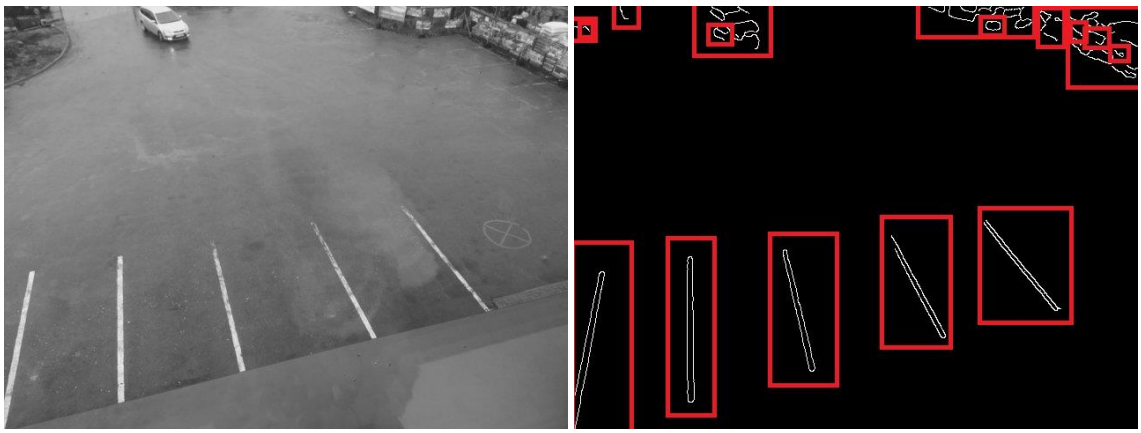

Figure 68: Grayscale image and found contours from canny detector

Bounding box restricting pixels corresponding to every found contour are used as a mask (picture 69 a) first image). Later for every pixel from contour *an hough line* is created. We will get many *hough lines* overlapping, with slightly different trend. Dominant line is picked from them and used as mask 2 (picture 69 a) second image). Combining them together creates possible prediction of parking line (picture 69 a) third image). Not all predictions are correct, though. Almost all false predictions are characterized as very short lines so we can filter them out using Accessible space mask and later simple size comparison to threshold value that was determined empirically. Picture 69 b) shows line that was incorrectly determined and will be deleted.

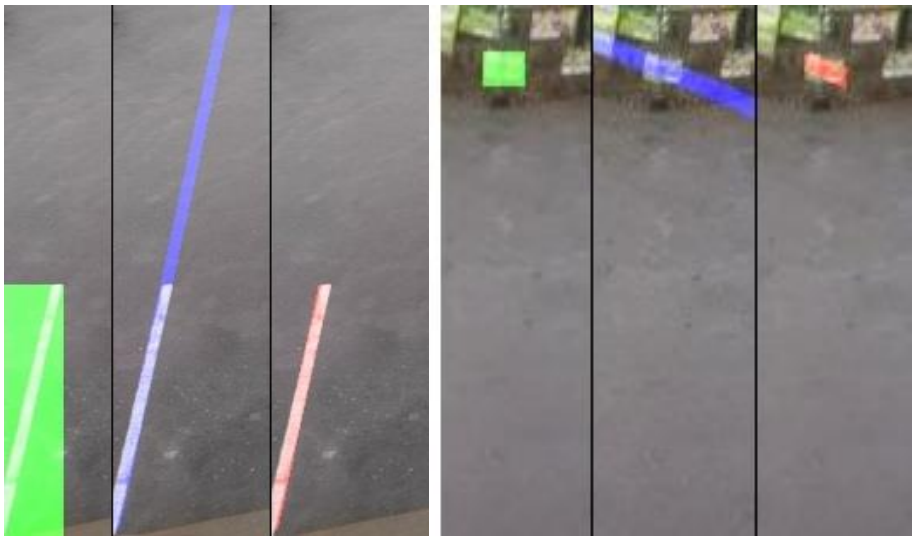

Figure 69: Proces of marking a)correct lines and b)noise points

After filtering out all incorrect lines, we are left with lines from which we can estimate park spaces. Obtained park lines are shown in picture 70.

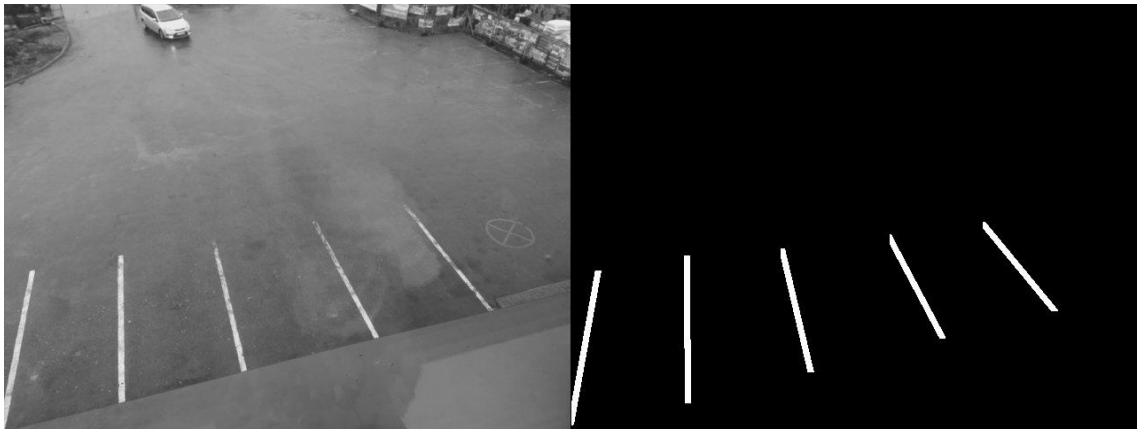

Figure 70: Grayscale image and found parking lines

This approach is applicable for parking spaces with camera positioned higher or looking directly down, with parking spaces strictly separated with small room for faulty human parking. When testing this solution on car park shown in pictures above, we have noticed a specific trend amongst people. If we gave them a lot of space, they almost always chosen wrong and parked their vehicle incorrectly. Next picture 71 shows some instances that will impair this car park solution quality. First two images show that people might park however they like, even when there are unoccupied spaces. Last image shows that people park in the middle of the car park when there is no space left. There is smaller car park in the back that is rarely used. As there are no park lines, this makes it difficult to track them as being occupied.

Figure 71: Wrong parking examples

Second possible way is to detect all vehicles in image over time and save their bounding box location. After doing this for quite a long time, we should be able to create cumulative heat map, showing the most occupied car park locations central points. This should be done during peak hours as we want to have as much parking places occupied as possible. Next picture shows average vehicle detection in image and corresponding heat map for bounding boxes after 30 minutes in use.

Figure 72: Found vehicles in image by object detector and generated heat map

After obtaining heat map, we can easily threshold image and find central point of every blob in image. In this step, we should have correct park spot central point locations. Now we will make average bounding box position from detected bounding boxes with central point close to our desired in image. Picture below shows found central points and their corresponding parking space locations.

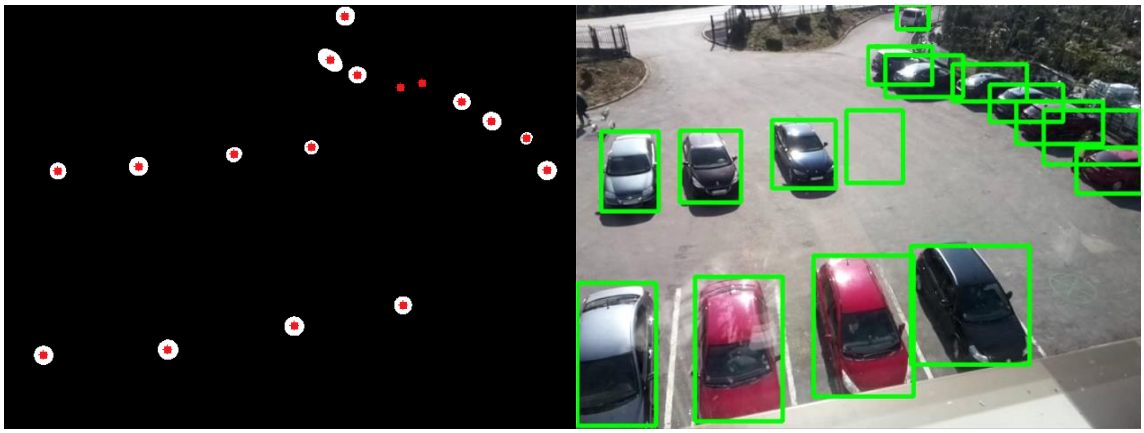

Figure 73: Thresholder heat map with found centre points

This whole process is explained using data flow diagram in picture 74. We can see that this process is composed from 2 main steps and one sub-step. First step is heat map generation, then comes sub-step which localizes parking spot centres. Last step tries to estimate correct bounding box positions.

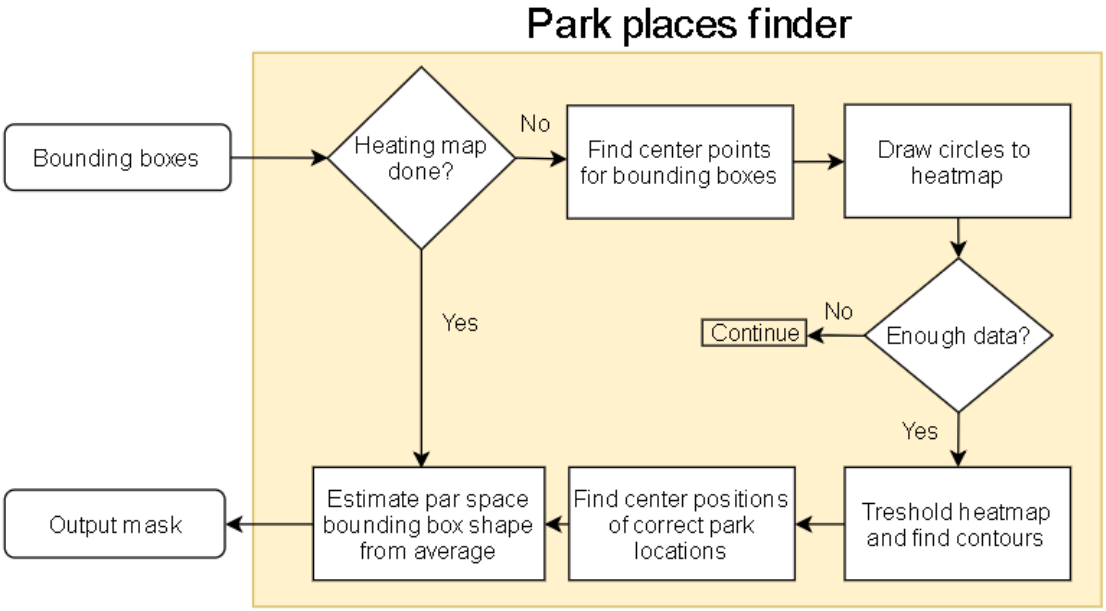

Figure 74: Application: Accessible space finder dataflow

7.2 Parking places monitoring

After the competition of auto-configuration, we can start monitor capacity of our car park. This should be relatively simple. All we need to do is to mask captured image with accessible space mask to decrease possible false detections. Then feed masked image to object detector and get bounding box prediction. Compare predicted bounding boxes and found car park places from auto-configuration by overlaying them and evaluating overlay ratio. Evaluation if they belong to correct park spot will be done using simplified pseudocode:

$$if\ \left(\frac{overlapping\ area}{whole\ area\ of\ both} > threshold\right):$$

True

else:

False

We have chosen threshold value of 0.6. Image showing real time parking spot monitoring is in image 75. Legend describing bounding boxes meaning was added.

Figure 75: Free park spaces monitoring with added legend

There are few problems with this solution. Object detection does not provide reliable enough data. Some cars might not get detected, like the car in the middle, then the parking place they are occupying is flagged as free even though it is not. This means that evaluated statistics will be always slightly off. Next image shows statistic describing number of vehicles present during colder Monday.

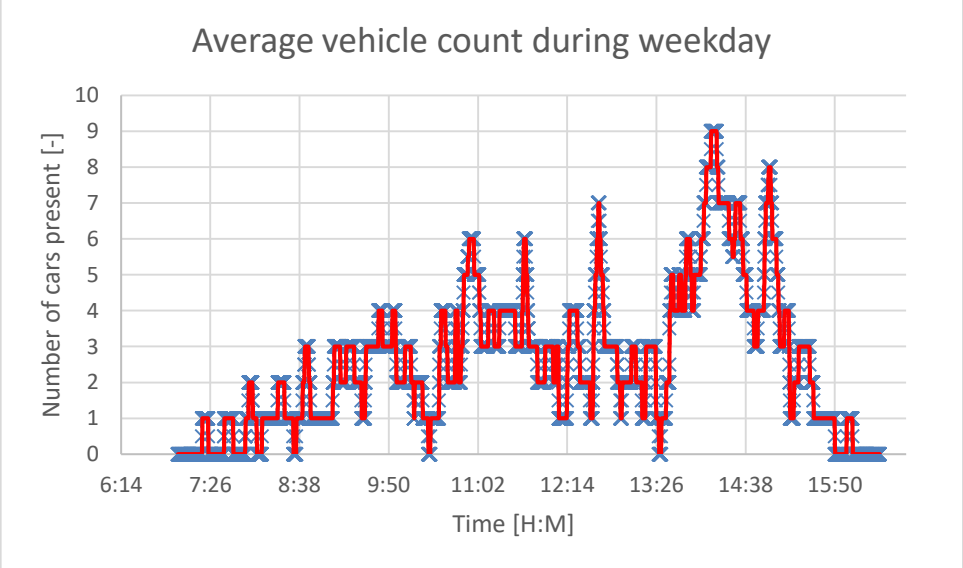

Figure 76: Average vehicle count during weekday on our car park

7.3 Posible additional improvements

Additionally, we have tested ways, how to improve value of our application. Specifically, we have added means for vehicle ID labelling and vehicle tracking. As these are not directly in focus of this paper, we will not go into details.

Different methods for vehicle tracking were tried and every single method has its own limitation. The best results were achieved using grid of Lucas-Kanade point trackers with correction after every frame to ensure grid parameters. Points tend to shift towards locally strongest edge without corrections, which were most commonly passing by people. This allows us to track movement of vehicles and create a better picture about duration of stay.

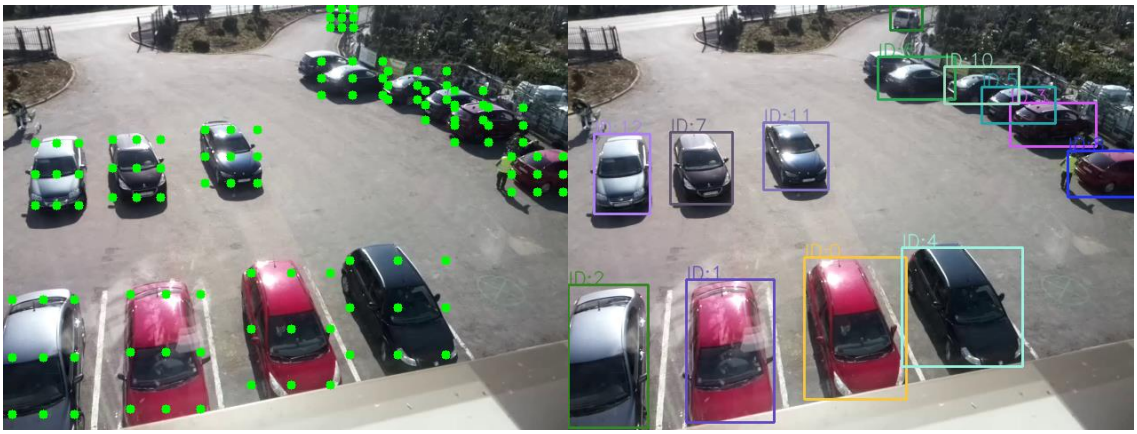

Figure 77: a) Grid of Lucas-kanade point trackers and b) monitored vehicles with assigned ID

However, this solution requires slight tweaking. All points tend to move towards centre, thanks to statistical correction. This will not be a problem, when update rate for object detection is frequent. We were able to obtain sufficient results for update every 20 frames.

8 CONCLUSION AND FUTURE WORK

As cities grow in size, so does number of cars. Automatized monitoring of free parking places will surely be needed. This work demonstrates use of machine learning algorithms, specifically Deep Learning algorithms for this purpose.

Fundamental goal of this paper is to present possibilities of incorporating different embedded systems in demanding tasks like object detection and classification in places, where high-end computers are typically used. We have used Raspberry Pi 3 Model B for this purpose. Later we took six promising models, from which 4 were pretrained and trained them on our own dataset. Then we tested the results. Considering its small size and relatively low computational power in comparison with common commercial solutions, this device is fast enough to provide real time inference. Surprisingly, after series of hardware optimizations, we were able to achieve up to 8.35 frames per second for inference with slightly worse precision for one of the tested Deep Learning models. This might be ideal choice for simpler problems. However, our application required more precise information, so we utilized slower model with better overall characteristics besides speed.

Our model application consisted of creating a modern automatic system monitoring free parking places. This was successful as demonstrated at the end of chapter 7. Our solution is sufficient for statistical use. However, there are the two main areas, where our solution could see some improvement there. There is still room for an improvement. Currently, we have to either sacrifice speed or precision. Firstly, we would welcome improvement in detection precision, which could be done by tweaking learning parameters and increasing learning dataset. Secondly, developing a way to incorporate Raspberry Pi 3 Model B GPU, as currently there is no direct support for GPU computations. Another possibility is using USB accelerator for CPU offloading. Because Raspberry Pi 3 Model B has only USB 2.0., this may be problematic.

Future work will consist of code optimization for faster support functionality. Also, we will do another model training process with slightly different parameters for *embedded_ssd_mobilenet_v1_coco* and *ssd_mobilenet_v1_coco*. We are expecting to see increase in precision with increased size of our dataset.

Object detection has still a long road ahead and we are curious, what will bring next decade. Demanding problems are still solved using preferably high-end machines, but gap between them and embedded devices is constantly shrinking. With improvement in embedded hardware, we should see better performance soon.

References

[1] COPELAND, MICHAEL. What's the Difference Between Artificial Intelligence, Machine Learning, and Deep Learning?. Blogs.nvidia.com [online]. 29.07.2016 [cit. 2018-10-14]. Available from: https://blogs.nvidia.com/blog/2016/07/29/ whats-difference-artificial-intelligence-machine-learning-deep-learning-ai/

[2] MALOO, Jinesh. Artificial Intelligence ,Machine Learning & Deep learning. *Medium* [online]. 16.08.2018 [cit. 2019-05-08]. Available from: https://becominghuman.ai/artificial-intilligence-machine-learning-deep-learning-df6dd0af500e

[3] TOM M., Mitchell. Machine Learning. 01.03.1997. New York, NY, USA: McGraw-Hill, 1997. ISBN 0070428077 9780070428072.

[4] AYODELE, Taiwo. Types of Machine Learning Algorithms. New Advances in Machine Learning. University of Portsmouth, 2010. DOI: 10.5772/9385.

[5] VISHAKHA, Jha. Machine Learning Algorithm - Backbone of emerging technologies. In: *TechLeer* [online]. 17.07.2017 [cit. 2019-05-08]. Available from: https://www.techleer.com/articles/203-machine-learning-algorithm-backbone-of-emerging-technologies/

[6] DENG, Li a Dong YU. Deep Learning: Methods and Applications. Foundations and Trends in Signal Processing. 2014, 2013(7), 197–387. DOI: 10.1561/2000000039.

[7] MAHAPATRA, SAMBIT. Why Deep Learning over Traditional Machine Learning?. Towards Data Science [online]. 21.03.2018 [cit. 2018-10-14]. Available from: https://towardsdatascience.com/why-deep-learning-is-needed-over-traditional-machine-learning-1b6a99177063

[8] Deep Learning Has Been Commercialized into More than 100 Use Cases, According to Tractica: oftware Revenue from Deep Learning Will Reach $35 Billion Worldwide by 2025. *BusinessWire* [online]. 30.05.2017 [cit. 2019-05-08]. Available from: https://www.businesswire.com/news/home/20170530005338/ en/Deep-Learning-Commercialized-100-Cases-Tractica

[9] Deep Learning 12: Energy-Based Learning (2)–Regularization & Loss Functions [online]. [cit. 2018-12-20]. Available from: https://ireneli.eu/2016/07/07/deep-learning-12-energy-based-learning-2-regularization-loss-functions/

[10] BAŽÍK, Martin. Optimalizace hlubokých neuronových sítí. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Prof. Ing. Lukáš Sekanina, Ph.D.

[11] KOJOUHAROV, Stefan. Cheat Sheets for AI, Neural Networks, Machine Learning, Deep Learning & Big Data: The Most Complete List of Best AI Cheat Sheets. Becoming Human: Exploring Artificial Intelligence & What it Means to be Human [online]. 09.07.2017 [cit. 2018-10-14]. Available from: https://becominghuman.ai/cheat-sheets-for-ai-neural-networks-machine-learning-deep-learning-big-data-678c51b4b463

[12] Cross-correlation. In: Wikipedia: the free encyclopedia [online]. San Francisco (CA): Wikimedia Foundation, 2001- [cit. 2018-10-19]. Available from: https://en.wikipedia.org/wiki/Cross-correlation
[13] STRATIL, Jan. *Hluboké neuronové sítě pro rozpoznání tváří ve videu*. Brno, 2017. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Michal Hradiš, Ph.D.
[14] KOLARÍK, M.Hluboké uèení pro klasikaci textù. Brno: Vysoké učení technické v Brne, Fakulta elektrotechniky a komunikačních technologií, 2017. 50 s. Vedoucí Ing. Lukáš Povoda.
[15] SANTOS, Leonardo Araujo dos. Pooling Layer [online]. [cit. 2018-11-02]. Available from: https://leonardoaraujosantos.gitbooks.io/artificial-inteligence/content/pooling_layer.html
[16] BUDHIRAJA, Amar. Dropout in (Deep) Machine learning. Medium [online]. 15.12.2016 [cit. 2018-11-10]. Available from: https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5
[17] IOFFE, Sergey a Christian SZEGEDY. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [online]. 1600 Amphitheatre Pkwy, Mountain View, CA 94043, February 2015 [cit. 2018-11-22].
[18] ROHAN, Thomas. Convolutional Networks for everyone. Medium [online]. Jan 15, 2018 [cit. 2018-11-22]. Available from: https://medium.com/@rohanthomas.me/convolutional-networks-for-everyone-1d0699de1a9d
[19] HADZIMA, Jaroslav, Sabó MAREK a Kratochvíla LUKÁŠ. *Detekcia áut v obraze pomocou Neurónových sietí*. Brno, 2018. Semestral projekt. Vysoké Učení Technické v Brne. Vedoucí práce Ligocki Adam, Ing.
[20] POIRSON, Ric, Justin JOHNSON, Fei-Fei LI a Andrej KARPATHY. Spatial localization and detection.
[21] REY, Javier. Object Detection with Deep Learning: The Definitive Guide [online]. In: . Wed, Aug 30, 2017 [cit. 2018-11-28]. Available from: https://tryolabs.com/blog/2017/08/30/object-detection-an-overview-in-the-age-of-deep-learning/
[22] GIRSHICK, Ross. Fast R-CNN [online]. 27 Sep 2015 [cit. 2018-11-28]. Available from: https://arxiv.org/pdf/1504.08083.pdf
[23] ERDOĞAN, Göksu. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks [online]. In: . 1 Feb 2016 [cit. 2018-01-12]. Available from: https://web.cs.hacettepe.edu.tr/~aykut/classes/spring2016/bil722/slides/w05-FasterR-CNN.pdf
[24] REN, Shaoqing, Kaiming HE, Ross GIRSHICK a Jian SUN. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks [online]. 6 Jan 2016 [cit. 2018-12-08]. Available from: https://arxiv.org/abs/1506.01497

[25] HUI, Jonathan. Understanding Region-based Fully Convolutional Networks (R-FCN) for object detection. Medium [online]. [cit. 2018-08-12]. Dostupné z: https://medium.com/@jonathan_hui/understanding-region-based-fully-convolutional-networks-r-fcn-for-object-detection-828316f07c99
[26] HE, Kaiming, Georgia GKIOXARI, Piotr DOLLAR a Ross GIRSHICK. Mask R-CNN: Facebook AI Research (FAIR) [online]. 24 Jan 2018 [cit. 2018-12-08]. Available from: https://arxiv.org/pdf/1703.06870.pdf
[27] LIU, Wei, Dragomir ANGUELOV, Dumitru ERHAN, Christian SZEGEDY, Scott REED, Cheng-Yang FU a Alexander C. BERG. SSD: Single Shot MultiBox Detector [online]. 29 Dec 2016 [cit. 2018-12-08]. Available from: https://arxiv.org/pdf/1512.02325.pdf
[28] REDMON, Joseph, Santosh DIVVALA, Ross GIRSHICK a Ali FARHADI. You Only Look Once: Unified, Real-Time Object Detection [online]. 9 May 2016 [cit. 2018-12-08]. Available from: https://arxiv.org/pdf/1506.02640.pdf
[29] REDMON, Joseph a Ali FARHADI. YOLO9000: Better, Faster, Stronger [online]. 2017 [cit. 2018-12-10].
[30] ImageNet [online]. [cit. 2018-12-11]. Available from: http://image-net.org/index
[31] COCO dataset. COCO dataset [online]. [cit. 2018-12-11]. Available from: http://cocodataset.org/#download
[32] Pascal VOC data sets [online]. [cit. 2018-12-11]. Available from: http://host.robots.ox.ac.uk/pascal/VOC/
[33] VGG data sets (faces) [online]. [cit. 2018-12-11]. Available from: http://www.robots.ox.ac.uk/~vgg/data/vgg_face2/
[34] The KITTI Vision Benchmark Suite [online]. [cit. 2018-12-11]. Available from: http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=2d
[35] Cars Dataset [online]. [cit. 2018-12-11]. Available from: http://ai.stanford.edu/~jkrause/cars/car_dataset.html
[36] Difference Between Microprocessor and Microcontroller. ELECTRONICS HUB [online]. MAY 29, 2015 [cit. 2018-12-13]. Available from: https://www.electronicshub.org/difference-between-microprocessor-and-microcontroller/
[37] NEKUŽA, Karel *Odlehčená kryptografie pro embedded zařízení*: bakalářská práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav telekomunikací, 2016. 42 s. Vedoucí práce byl Ing. Zdeněk Martinásek, Phd.
[38] EVANCZUK, Stephen. Applying machine learning in embedded systems. Embedded [online]. JULY 11, 2018 [cit. 2018-12-15]. Available from: https://www.embedded.com/design/prototyping-and-development/4460862/3/Applying-machine-learning-in-embedded-systems

[39] HIJAZI, Samer. Neural Network Technology for Embedded Systems. Embedded [online]. 2016 [cit. 2018-12-15]. Available from: http://nfic-2016.ieeesiliconvalley.org/wp-content/uploads/sites/16/2016/05/NFIC-2016-Cadence-Samer-Hijazi.pdf
[40] LIAO, Yihua. *Neural Networks in Hardware: A Survey* [online]. Department of Computer Science, University of California, Davis One Shields Avenue, Davis, CA 95616 [cit. 2018-12-15]. DOI: 10.5121/ijaia.2018.9105. Available from: https://bit.csc.lsu.edu/~jianhua/shiv2.pdf
[41] ROJAS, Raul. Neural Networks: A Systematic Introduction. Springer-Verlag, Berlin, 1996.
[42] ROBINSON, Jamal. FPGAs, Deep Learning, Software Defined Networks and the Cloud: A Love Story Part 1: Digging into FPGAs and how they are being utilized in the cloud. Medium [online]. Nov 11, 2017 [cit. 2018-12-15]. Available from: https://medium.com/@jamal.robinson/fpgas-deep-learning-software-defined-networks-and-the-cloud-a-love-story-part-1-c685dc6b657b
[43] RAVAL, Siraj. *TPU MachineLearning* [online]. Sep 28, 2018 [cit. 2019-04-22]. Available from: https://github.com/llSourcell/TPU_Machine_Learning/blob/master/TPU_MachineLearning%20(2).ipynb
[44] WHY GPUS?: NVIDIA Tesla Pascal P100 GPU [online]. [cit. 2018-12-15]. Available from: http://www.fmslib.com/mkt/gpus.html
[45] SATO, Kaz, Cliff YOUNG a David PATTERSON. *An in-depth look at Google's first Tensor Processing Unit (TPU)*[online]. May 12, 2017 [cit. 2019-04-22]. Available from: https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu
[46] NAKHARE, Gaurav. Hardware options for Machine/Deep Learning. Stanford University [online]. July 31, 2017 [cit. 2018-12-15]. Available from: https://mse238blog.stanford.edu/2017/07/gnakhare/hardware-options-for-machinedeep-learning/
[47] JAWANDHIYA, Pooja. *HARDWARE DESIGN FOR MACHINE LEARNING* [online]. School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore, 1, January 2018 [cit. 2018-12-14]. DOI: 10.5121/ijaia.2018.9105. Available from: http://aircconline.com/ijaia/V9N1/9118ijaia05.pdf
[48] DEAN, Jeff a Urs HÖLZLE. *Build and train machine learning models on our new Google Cloud TPUs* [online]. May 17, 2017 [cit. 2019-04-22]. Available from: https://www.blog.google/products/google-cloud/google-cloud-offer-tpus-machine-learning/

[49] Discovery kit with STM32F407VG Microcontroller Unit. Element 14 [online]. [cit. 2018-12-18]. Available from: https://www.element14.com/community/docs/DOC-80768/l/discovery-kit-with-stm32f407vg-microcontroller-unit#overview
[50] Raspberry Pi 3 Model B. Raspberry Pi [online]. [cit. 2018-12-18]. Available from: https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
[51] HADZIMA, J. 3D skener se strukturovaným osvětlením. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2017. 81 s. Vedoucí bakalářské práce Ing. Aleš Jelínek.
[51] NVIDIA 900-83310-0001-000 | Jetson TX2. Arrow [online]. [cit. 2018-12-18]. Available from: https://www.arrow.com/en/products/900-83310-0001-000/nvidia
[52] Intel Movidius™ Neural Compute Stick. Mouser electronics [online]. [cit. 2018-12-18]. Available from: https://eu.mouser.com/new/intel/intel-movidius-stick/
[53] *Jetson TX2 Performance* [online]. 10/07/2017 [cit. 2019-04-23]. Available from: https://devtalk.nvidia.com/default/topic/1024825/cuda-programming-and-performance/jetson-tx2-performance/
[54] BORN, Eric. *Intel crams 100 GFLOPS of neural-net inferencing onto a USB stick* [online]. July 20, 2017 [cit. 2019-04-23]. Available from: https://techreport.com/news/32272/intel-crams-100-gflops-of-neural-net-inferencing-onto-a-usb-stick
[55] WILLIAMS, Alun. *DevBoard Watch: Google's AIY Edge TPU Boards are Raspberry Pi friendly* [online]. 13.09.2018 [cit. 2019-04-28]. Available from: https://www.electronicsweekly.com/blogs/gadget-master/raspberry-pi-gadget-master/devboard-watch-googles-aiy-edge-tpu-boards-raspberry-pi-friendly-2018-09/
[56] *Edge TPU Devices: Now available from Coral!* [online]. [cit. 2019-04-28]. Available from: https://aiyprojects.withgoogle.com/edge-tpu
[57] RPi SD cards. Embedded Linux Wiki [online]. [cit. 2018-12-20]. Available from: https://elinux.org/RPi_SD_cards
[58] GODARD, Tuatini. Building TensorFlow 1.3.0-rc1 for Raspberry Pi/Ubuntu 16.04: a Step-By-Step Guide [online]. [cit. 2018-12-20]. Available from: https://gist.github.com/EKami/9869ae6347f68c592c5b5cd181a3b205
[59] GODARD, Tuatini. Tutorial to set up TensorFlow Object Detection API on the Raspberry Pi [online]. 12-10-2018: [cit. 2018-12-20]. Available from: https://github.com/EdjeElectronics/TensorFlow-Object-Detection-on-the-Raspberry-Pi
[60] *Tensorflow detection model zoo* [online]. 18.11.2017 [cit. 2019-04-11]. Available from: https://github.com/tensorflow/models/blob/079d67d9a0b3407e8d074a200780f3835413ef99/research/object_detection/g3doc/detection_model_zoo.md
[61] *Difference between ZRAM and ZSWAP* [online]. 26.08.2013 [cit. 2019-04-29]. Available from: https://stackoverflow.com/questions/18437205/difference-between-zram-and-zswap

[62] *Raspberry Pi hardware* [online]. 18.01.2019 [cit. 2019-04-28]. Available from: https://www.raspberrypi.org/documentation/hardware/raspberrypi/

[63] JURAS, Evan. TensorFlow-Object-Detection-API-Tutorial-Train-Multiple-Objects-Windows-10: How to train a TensorFlow Object Detection Classifier for multiple object detection on Windows [online]. 10.04.2019 [cit. 2019-04-29]. Available from: https://github.com/EdjeElectronics/TensorFlow-Object-Detection-API-Tutorial-Train-Multiple-Objects-Windows-10/projects

[64] LIN, Tzu Ta. *LabelImg* [online]. 22.04.2019 [cit. 2019-04-29]. Available from: https://github.com/tzutalin/labelImg

[65] F1 score. In: *Wikipedia: the free encyclopedia* [online]. San Francisco (CA): Wikimedia Foundation, 16.12.2018 [cit. 2019-05-08]. Available from: https://en.wikipedia.org/wiki/F1_score

[66] BENCHOFF, Brian. *PI 3 BENCHMARKS: THE MARKETING HYPE IS TRUE* [online]. March 1, 2016 [cit. 2019-04-22]. Available from: https://hackaday.com/2016/03/01/pi-3-benchmarks-the-marketing-hype-is-true/

9 LIST OF ABBREVIATIONS

AI	-	Artificial Intelligence
BN	-	Batch normalization
CCTV	-	Closed Circuit Television
CNN	-	Convolutional neural network
CPU	-	Central processing unit
DL	-	Deep Learning
FPGA	-	Field programmable gate array
FPS	-	Frames per second
GPU	-	Graphical processing unit
HW	-	Hardware
MCU	-	Micro-controller unit
ML	-	Machine learning
NN	-	Neural network
SW	-	Software
OCR	-	Optical Character Recognition
OS	-	Operating system
PRI	-	Raspberry Pi 3 model B
SVM	-	Support vector Machines
TPU	-	Tensor processing unit

List of attachments

Attachment 1. Models_Training_and_testing .pdf

Attachment 2. Program.zip