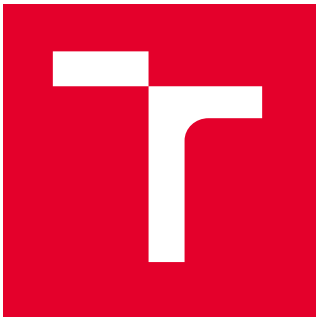


BRNO UNIVERSITY OF TECHNOLOGY

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

FAKULTA ELEKTROTECHNIKY
A KOMUNIKAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF RADIO ELECTRONICS

ÚSTAV RADIOELEKTRONIKY

DETECTION OF PARKING SPACE AVAILABILITY BASED ON VIDEO

DETEKCE DOSTUPNOSTI PARKOVACÍCH MÍST NA ZÁKLADĚ VIDEA

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR — **Miloslav Kužela**
AUTOR PRÁCE

SUPERVISOR — **doc. Ing. Tomáš Frýza, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Bakalářská práce

bakalářský studijní program **Elektronika a komunikační technologie**

Ústav radioelektroniky

Student: Miloslav Kužela ***ID:*** 240648

Ročník: 3 ***Akademický rok:*** 2023/24

NÁZEV TÉMATU:

Detekce dostupnosti parkovacích míst na základě videa

POKYNY PRO VYPRACOVÁNÍ:

Projekt je zaměřen do oblasti zpracování video signálů a využití strojového učení pro možnosti smart parkingu. Výsledkem práce bude systém, který umožní najít volné místo na parkovišti, bude schopen vyhodnocovat průběžnou obsazenost a detekovat špatně zaparkovaná vozidla. Prostudujte současné systémy umožňující detekci vozidel a jejich využití pro smart parking. Prostudujte možnosti klasifikátorů strojového učení zaměřené na vozidla, případně chodce. Vyberte vhodné komponenty a navrhněte koncepci systému, který bude schopen zaznamenávat provoz na parkovišti a odesílat data na vhodné online úložiště.

Systém aplikujte na vhodné parkoviště, naprogramujte celou aplikaci a proveďte detailní testování. Systém musí být schopen vyčíslit celkovou obsazenost parkovacích ploch, určit nejbližší volné místo a detekovat anomálie v parkování. Svou práci publikujte na vhodné platformě, např. GitHub, studentská soutěž apod.

DOPORUČENÁ LITERATURA:

[1] CHEN, Lun-Chi, Ruey-Kai SHEU, Wen-Yi PENG, Jyh-Horng WU a Chien-Hao TSENG. Video-Based Parking Occupancy Detection for Smart Control System. Applied Sciences [online]. 2020, 10(3) [cit. 2023-05-29]. ISSN 2076-3417. Dostupné z: doi:10.3390/app10031079

[2] LEE, Cheng Pin, Fabian Tee Jee LENG, Riyaz Ahamed Ariyaluran HABEEB, Mohamed Ahzam AMANULLAH a Muhammad Habib ur REHMAN. Edge computing-enabled secure and energy-efficient smart parking: A review. Microprocessors and Microsystems [online]. 2022, 93 [cit. 2023-05-29]. ISSN 01419331. Dostupné z: doi:10.1016/j.micpro.2022.104612

Termín zadání: 16.2.2024 ***Termín odevzdání:*** 27.5.2024

Vedoucí práce: doc. Ing. Tomáš Frýza, Ph.D.

doc. Ing. Lucie Hudcová, Ph.D.
předseda rady studijního programu

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

Bachelor's Thesis

Bachelor's study program **Electronics and Communication Technologies**

Department of Radio Electronics

Student: Miloslav Kužela ***ID:*** 240648

Year of study: 3 ***Academic year:*** 2023/24

TITLE OF THESIS:

Detection of parking space availability based on video

INSTRUCTION:

The project is focused on the area of video signal processing and the use of machine learning for smart parking. The result of the work will be a system that will allow you to find a free parking space, will be able to evaluate the current occupancy and detect poorly parked vehicles. Study current systems that allow the detection of vehicles and their use for smart parking. Study the possibilities of machine learning classifiers focused on vehicles, or pedestrians. Select suitable components and propose a concept for a system that will be able to record parking lot traffic and send data to a suitable online storage. Apply the system to a suitable parking lot, program the entire application and perform detailed testing. The system must be able to calculate the total occupancy of parking spaces, determine the nearest free space and detect anomalies in parking. Publish your work on a suitable platform, such as GitHub, a student competition, etc.

RECOMMENDED LITERATURE:

[1] CHEN, Lun-Chi, Ruey-Kai SHEU, Wen-Yi PENG, Jyh-Horng WU a Chien-Hao TSENG. Video-Based Parking Occupancy Detection for Smart Control System. Applied Sciences [online]. 2020, 10(3) [cit. 2023-05-29]. ISSN 2076-3417. Dostupné z: doi:10.3390/app10031079

[2] LEE, Cheng Pin, Fabian Tee Jee LENG, Riyaz Ahamed Ariyaluran HABEEB, Mohamed Ahzam AMANULLAH a Muhammad Habib ur REHMAN. Edge computing-enabled secure and energy-efficient smart parking: A review. Microprocessors and Microsystems [online]. 2022, 93 [cit. 2023-05-29]. ISSN 01419331. Dostupné z: doi:10.1016/j.micpro.2022.104612

Date of project specification: 16.2.2024 ***Deadline for submission:*** 27.5.2024

Supervisor: doc. Ing. Tomáš Frýza, Ph.D.

doc. Ing. Lucie Hudcová, Ph.D.
Chair of study program board

ABSTRACT

Determining a parking space occupation is often solved by using physical sensors located near a parking space, but with the rise of machine learning, it is possible to apply such technology with the use of cameras and detection algorithms to solve such a problem. This thesis focuses on the use of this machine learning model. Presents currently available models and detectors, discusses the creation of a custom data set with a custom file structure, trains such a model, and consults its results based on its accuracy when applied in a parking lot. This model is then used among side a created web server that allows the users of the parking lot to view the current occupancy and history. All by using the Python programming language with Torchvision libraries.

KEYWORDS

Car, Parking, Detection, Machine learning, Machine vision, AI, Camera, Video

ABSTRAKT

Detekování obsazenosti parkovacích míst je často řešeno použitím senzorů umístěných v blízké lokaci parkovacího místa. Se vzrůstem strojového učení je možnost využití této technologie za použití kamer a detekčních algoritmů. Práce se zabývá právě vytvořením a použitím takového modelu k detekci obsazenosti parkoviště. Probírá existující modely a detektory, vytvoření vlastního datasetu s konkrétní strukturou, vytvoření a naučení různých typů modelů a probrání vysledků při testování daných modelů na vlastních záznamech z parkovací plochy. Poté následné vytvoření webové aplikace na které můžou návštěvnící parkoviště pozorovat obsazenost parkoviště. Vše za použití programovacího jazyka Python s knihovnami Torchvision.

KLÍČOVÁ SLOVA

Auto, Parkování, Detekce, Strojové Učení, Strojové vidění, AI, Kamera, Video

ROZŠÍŘENÝ ABSTRAKT

Automobily se v současné době staly nedílnou součástí našich životů. Automobilů stále přibývá a to přináší určité problémy. Automobily potřebují někde parkovat, což vede k nárůstu počtu velkých parkovacích ploch, které se stává obtížné a zdlouhavé projíždět, aby se našlo volné parkovací místo. Přišli jsem s řešeními pro tento problém v podobě různých fyzických senzorů, které monitorují jednotlivá parkovací místa. Avšak ceny těchto systémů rostou s počtem parkovacích míst, což je bohužel současný trend.

S rozvojem umělé inteligence, známé také jako strojové učení a strojové vidění, se objevují alternativy pro řešení těchto problémů. Umělá inteligence je obor informatiky, který se studuje již dlouhou dobu, ale v poslední době dosáhl mnoha důležitých průlomů. Vzhledem k tomu, že se kamery často používají k monitorování veřejných nebo soukromých parkovišť, proč je nevyužít také k monitorování obsazenosti parkovacích míst? Tato práce se zaměřuje na aplikaci strojového vidění na videozáznam parkoviště za účelem detekce obsazených a volných parkovacích míst s minimálním uživatelským vstupem. Všechny testy a finální aplikace budou prováděny na parkovišti vedle budovy T10 Vysokého učení technické v Brně.

Pro strojové vidění existuje několik typů detektorů a modelů. Po analýze dostupných konvolučních neuronových sítí byly vybrány tři modely. Mobilenet V3 large, Mobilenet V3 small a Resnet50. Tyto modely byly testování na dvou detektorech: Faster RCNN a RetinaNET.

Pro řešení zmíněného problému se využil programovací jazyk Python 3.11 nejvýznamněji s knihovnami Torchvision a CV2 které umožňují vytváření, upravování a trénování různých modelů strojového učení a práci s obrázkovými daty. S použitím těchto knihoven byl napsaný program, který umožňuje uživateli zvolit z listu modelů a natrénovat jej na množině obrázků parkoviště s označenými parkovacími místy. Aplikace ještě zaznamenává průběh trénování na stránky Comet, kde může uživatel pozorovat postup trénování na přehledných grafech. Pro označení parkovacích míst byla taktéž vytvořena jednoduchá aplikace v prostředí Jupyter.

Pro natrénování konvoluční neuronové sítě jsou třeba obrázková data. K tomu byl zprvu využit telefonní mobil iPhone X s aplikací která pořídila fotku co 10 minut. Následně bylo pořizovací zařízení změněno na Raspberry Pi 4 které je vybaveno kamerou Camera Module V2. Pomocí aplikace v prostředí Jupyter byly fotky zpracovány, a se zakreslenými parkovacími místy a jejich stavem obsazenosti vytváří dataset T10LOT který je součástí výsledku práce.

Následně byla naprogramována testovací funkce, která ověřila všechny natrénované modely a vypočítala jejich F1 skóre. Program disponuje funkcí uložení obrázků, na kterých lze jednotlivé detekce pozorovat. Tyto obrázky společně s natrénovanými

modely a datasetem T10LOT jsou přiloženy na Google Drive odkazované v práci.

Po porovnání testovacích výsledků se došlo k závěru, že nejefektivnější síť je Mobilenet V3 large s přetrénovanými váhy na YOLO datasetu. F1 skoré této sítě vychází jako jedno z nejlepších. Velkou výhodou je i přetrénovaný stav tohoto modelu. To znamená že není třeba model trénovat na velké hodnotě dat, ale stačí malá skupina fotek parkoviště, na kterém bude model použit.

Pro finální aplikaci byla napsána knihovna ArgonPark, která má metody pro vytvoření a načtení modelu který se poté používá pro analýzu parkovacích míst na vstupních fotkách. Knihovna je jednoduchá a zdokumentována pomocí Sphinx. Zdrojový kód je přístupný jak v elektronické příloze, tak na GitHub stránce této práce.

Jako poslední byla využita databáze MongoDB která slouží pro ukládání stavu celého parkoviště. K těmto datům se poté dá následně libovolně přistupovat z jakéhokoliv zařízení. Data obsahují časové razítko, kdy byl záznam vytvořen, jméno parkovacího místa, jeho lokaci na obrázku pomocí čtyř bodů, obsazenost a poměr překrytí s detekcí.

Pro uživatelskou přívětivost byly tyto nástroje zakomponovány do webové aplikace, která je provozována na Raspberry Pi, na kterém se nachází i kamera kterou je snímáno parkoviště. Na této webové stránce lze vidět aktuální fotku parkoviště se nakreslenými zónami a jejich obsazenost danou barvou. Tato fotka se obnovuje automaticky co dvě sekundy. Následně lze na stránce vyžádat informace o parkovaní, kdy se uživateli poskytne počet obsazených míst a nejbližší dostupné parkovací místo. Dále se na stránce nachází možnost pro upravení mapy parkoviště. To se provádí stáhnutím obrázku bez zakreslených parkovacích míst a pomocí využití přiložené grafické aplikace se označí parkovací místa. Aplikace následně vygeneruje soubor, který se pomocí stránky nahraje a mapu zaktualizuje. Jako poslední funkce se nabízí zobrazení historie obsazenosti parkoviště, a to za pomocí barového grafu.

Použitím počítače Raspberry Pi v4, který nemá dostatečně výkonný procesor se způsobuje delší čas na inferenci jednoho snímku, orientačně se pohybuje v rozmezí 4 až 5 sekund. Tento časový úsek není pro konkrétní použití zcela kritický, ale správnou optimalizací modelů, jako například kvantizační metodou, se dá čas zkrátit. To ale obnáší složitější úpravu trénovací funkce. Při použití výkonnějšího Hardwaru by se inference mohla velice zrychlit, a to na několik snímků za sekundu.

Webová aplikace je nyní v provozu a sbírá data, která se později dají graficky vizualizovat a využít k dalšímu výzkumu.

KUŽELA, Miloslav. *Detection of parking space availability based on video*. Bachelor's Thesis. Brno: Brno University of Technology, Faculty of Electrical Engineering and Communication, Department of Radio Electronics, 2024. Advised by doc. Ing. Tomáš Frýza, Ph.D.

Author's Declaration

Author:	Miloslav Kužela
Author's ID:	240648
Paper type:	Bachelor's Thesis
Academic year:	2023/24
Topic:	Detection of parking space availability based on video

I declare that I have written this paper independently, under the guidance of the advisor and using exclusively the technical references and other sources of information cited in the paper and listed in the comprehensive bibliography at the end of the paper.

As the author, I furthermore declare that, with respect to the creation of this paper, I have not infringed any copyright or violated anyone's personal and/or ownership rights. In this context, I am fully aware of the consequences of breaking Regulation § 11 of the Copyright Act No. 121/2000 Coll. of the Czech Republic, as amended, and of any breach of rights related to intellectual property or introduced within amendments to relevant Acts such as the Intellectual Property Act or the Criminal Code, Act No. 40/2009 Coll. of the Czech Republic, Section 2, Head VI, Part 4.

Brno ....

author's signature*

*The author signs only in the printed version.

ACKNOWLEDGEMENT

I would like to thank the advisor of my thesis, doc. Ing. Tomáš Frýza, Ph.D. for his valuable comments, time and help in writing this thesis.

Contents

List of Figures

List of Tables

Listings

Introduction

Cars in current time have became such an important part of our lives. In a recent study in 2021 in the Czech Republic, there are approximately 579 cars per 1000 inhabitants [4]. This poses certain problems. These cars need to go somewhere, so there are more larger parking spaces that are very difficult to navigate to find free parking places. We have already come up with solutions to solve these problems with all kinds of physical sensors that monitor a single parking place. But the prices of such systems increase with the number of parking places, which is unfortunately the current trend.

With the rise of AI (Artificial Intelligence), or better known machine learning and machine vision, there are alternatives to solve such problems. Artificial intelligence is a field of computer science that has been studied for ages but has recently made many important breakthroughs. Since CCTV (Closed-circuit television) cameras are often used to monitor a public or private parking lot, why not also use them to monitor parking occupancy? This thesis focuses on applying machine vision to a video recording of a car park to detect occupied and free parking spaces with minimal user input. The final working version will make use of cloud edge computing. All tests will be conducted in a parking lot next to a BUT (Brno University of Technology) building.

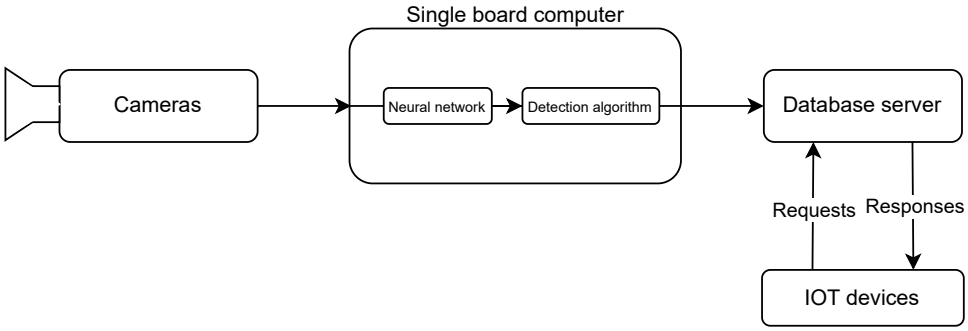


Fig. 1: Function diagram

1 Theory and approaches

1.1 Parking space detection

Current solutions that solve the problem of parking space detection can be divided into two groups. The first uses a localized system to detect only incoming and outgoing cars, and the other monitors individual parking spaces. Both have their own advantages and disadvantages. The first system utilizes gates that allow cars to enter and leave the parking lot and often prints tickets that would later be used to pay for parking; with the addition of cameras, it is possible to log the license plates and link them with parking tickets. This system has the ability to only know about the current number of cars in the parking lot. This information is then compared with the maximum capacity to determine how many parking places should be left unoccupied, but not which ones. This also presents the problem that if there were an electrical or technical problem of the gate system, the count of cars currently on the parking lot could change without the system knowing.

The second system, on the other hand, uses a detector of some kind to monitor individual parking spaces. This then allows parking lot managers and potential visitors to know what exact parking spaces are free and which are not, as well as to reserve certain spots for frequent visitors. Such systems often only signal the occupancy of a parking space with a light located near it, see Figure 1.1.

Fig. 1.1: Signalization of occupancy

1.1.1 Sensors

As mentioned above, sensors are used for systems that monitor individual parking places. These sensors need to be connected to a main station, which then provides

Fig. 1.2: Ultrasonic sensor in a parking garage mounted on a ceiling

all the necessary logging and computation. Here are some of the most common types of sensors used:

- **Ultrasonic sensors** - Such sensors are mostly located above the parking slot, preferably on the ceiling, which is why they are prevalent in parking garages. They work by transmitting and then receiving ultrasonic waves and measuring the time between, thus obtaining the distance [5]. An example of a system used in a parking garage is shown in Figure 1.2. The disadvantage of such a sensor is that it detects any obstacle, including those that are not made of metal. This can cause false positive detection, which affects the accuracy of the system. To add to this disadvantage, the speed of sound changes with air temperature and moisture, so the sensor needs to compensate by having additional sensors.
- **Magnetic sensors** - Magnetometers detect the change of electromagnetic fields when there is a vehicle in close proximity. They are often installed under each parking space, which is reflected in installation costs, but provide an accurate detection of occupancy [6]. They are suitable for outdoor and indoor parking lots.
- **Milimiter wave radars** - MMW (Milimeter wave) radars have recently become more popular in the automotive industry. They in a way work similar to ultrasonic sensors, but use much higher frequencies (30 GHz and higher). Because they use high frequencies, these sensors use small antennas, which is why the receiver and transmitter can fit onto a single chip. A good example is a PCR (pulsed coherent radar) developed by Acconeer [7]. They are often installed on the ground and can detect a standing car with high accuracy. With more precise algorithms, it is possible to differentiate between different objects. They can be considered an IoT (Internet of Thing) device, as they

are often deployed in an open parking lot and communicate using Wi-Fi or Bluetooth [8].

There is also the possible use of induction loops under the parking space. Such induction loops are used at crossroads with traffic lights to correctly and efficiently time the light change according to the number of waiting cars. Using a coil connected to a power source that detects the change in inductance when a car drives over it is simple however, its simplicity is overshadowed by the cost of installation and the potential power loss. With the addition of having to replace such coils every 3 to 7 years, makes this solution unfeasible for parking lots [9].

1.1.2 Use of cameras

When using video data, the biggest obstacle to detecting a free space is to locate the parking space itself; this can be done in a number of ways, be it by using machine vision or ML models. Numerous studies on the topic of parking occupancy detection test different types of ML object classifiers. The most prominent detectors used are R-CNN (Regional-Based Convolution Neural Network), Resnet (Residual neural network), and YOLO (you only look once), all with different results and use cases[10][11]. The biggest problem with using cameras is the inaccuracy with which the image can be captured. That means different environmental factors such as weather and lightning conditions. The photos on Figure 1.3 were captured in a short time frame of each other with the same camera, but the second photo had such a lightning condition that the camera incorrectly adjusted its saturation.

When using machine vision algorithms to detect cars, a change in the color of the lightning can possibly change the detection result. Having the ability to train the models in these situations can prevent inaccuracies. However, the use of cameras for parking detection is not yet as popular as that of physical sensors due to their accuracy and recentness.

Fig. 1.3: Comparison with visible color distortion

1.2 Artificial intelligence and machine learning

1.2.1 Artificial intelligence

Intelligence itself would be defined as the ability to make decisions on your own when presented with a certain amount of data. This decision would be influenced not only by the input data presented, but also by your previous knowledge. Computer programs do not have this ability. They work by using an algorithm, which is a set of predetermined steps that the program should take to solve a problem [12]. This problem is usually predetermined, like solving a math equation, for example. Making an artificial intelligence is to make a program that would solve a problem by using what it has learned in the past on a similar but not the same problem, hence why its called ML.

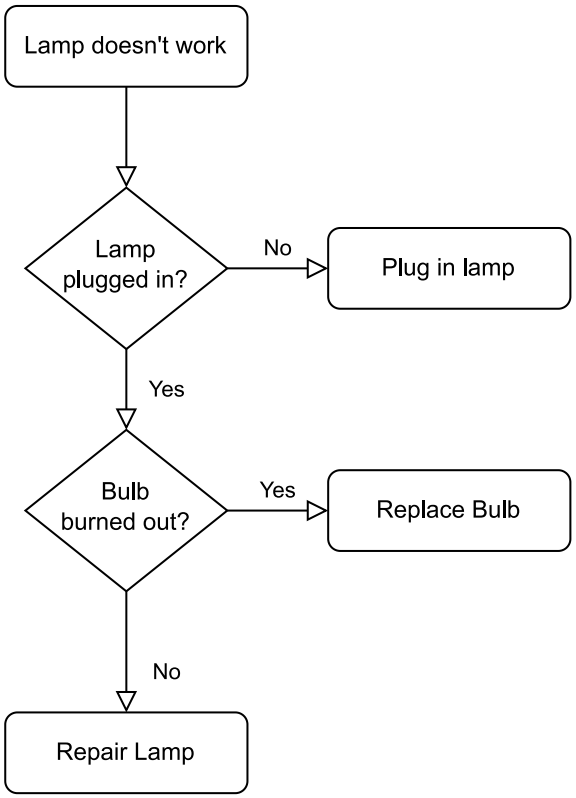


Fig. 1.4: Example of an algorithm

1.2.2 Neural networks

The initial research on neural networks was based on the workings of the human brain. That is, "the neural network is a distributed parallel system of executive elements that are organized so that the network can perform the desired function F" [13]. To better understand this sentence, "executive elements" are neurons that are interconnected in layers, where individual neurons are connected to other neurons in other layers by a connection that has a certain weight. The arrangement of these connections differs between different neural nets, but the purpose remains the same.

If we ignore the inner workings, the neural network can be described by a simple diagram on Figure 1.5. Here, the function $F(x)$ is performed by the neural network. The result of the function is affected by the weight of the connection. This weight can be taken as a form of signal strength that affects the final result. These weights are the way for the neural network to "remember" and are bound to change during the learning process.

If a single neuron in a neural network is described by a mathematical description, it would result in a simple diagram, see Figure 1.6. Note that each input connection has its own weight, which affects the input value. The next thing to notice is that the neuron has its own weight, called bias. This bias also affects the calculation.

Connecting these neurons to a network consisting of hidden layers of n neurons will result in a neural network. There are multiples of different layouts and connections, which are called typologies. This thesis will only describe a single topology, which is important to understand the following materials, as the use case for each topology is different. Since machine vision mostly uses CNN (Convolution Neural Network), the theoretical part will describe the basics of an FNN (Feedforward Neural Network), specifically Perceptron. FNNs are named in such a way because the information flows forward through their layers and never loops back.

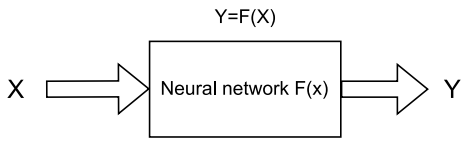


Fig. 1.5: Simple neural network function

1.2.3 Perceptron

Single layer perceptron

A single layer perceptron is just a single neuron that has n inputs and an output, see Figure 1.6. Due to its simplicity, it will be used to explain the basics of neural

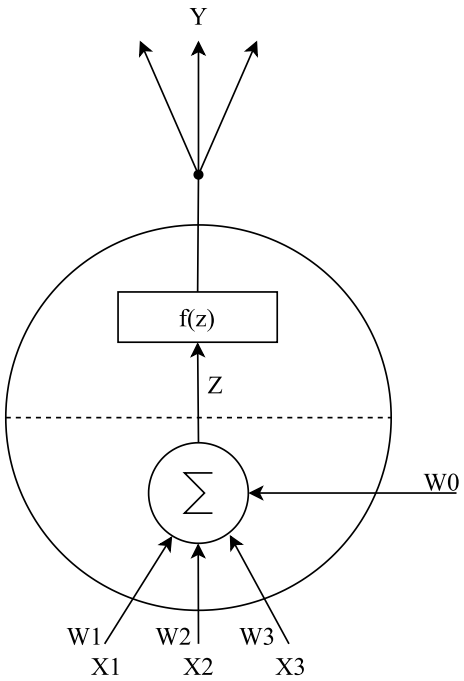


Fig. 1.6: Single neuron

networks. Understanding these fundamentals is really useful when dealing with other topics of machine learning later. The neuron has its bias, and each of its inputs has its weight. The neuron uses these inputs and weights to solve a single predetermined function. Its workings can be described by an equation (1.1).

$$y = f(z) \quad \text{where} \quad z = w_0 + \sum_{i=1}^{n} w_i x_i \tag{1.1}$$

Here, f is a transfer function and z is the inner potential of neurons that is affected by the input and its weight. There is a multiple of transfer functions. Those without a threshold on Figure 1.7 and those with it on Figure 1.8. The output of a neuron without a threshold function can be 0/1 or -1/0; on the other hand, when using a threshold, the neuron can have up to three values as output, -1/0/1. It is important to note the rules for where the output jumps to a different value. Equation (1.3) represents bipolar functions and (1.2) for unipolar functions with a threshold.

$$\begin{aligned} if\ z < \theta,\ then\ f(z) = 0 \\ if\ z >= \theta,\ then\ f(z) = 1 \end{aligned} \tag{1.2}$$

$$\begin{aligned} if\ z <= -\theta, then\ f(z) = -1 \\ if\ -\theta < t < \theta, then\ f(z) = 0 \\ if\ z >= \theta, thenf(z) = 1 \end{aligned} \tag{1.3}$$

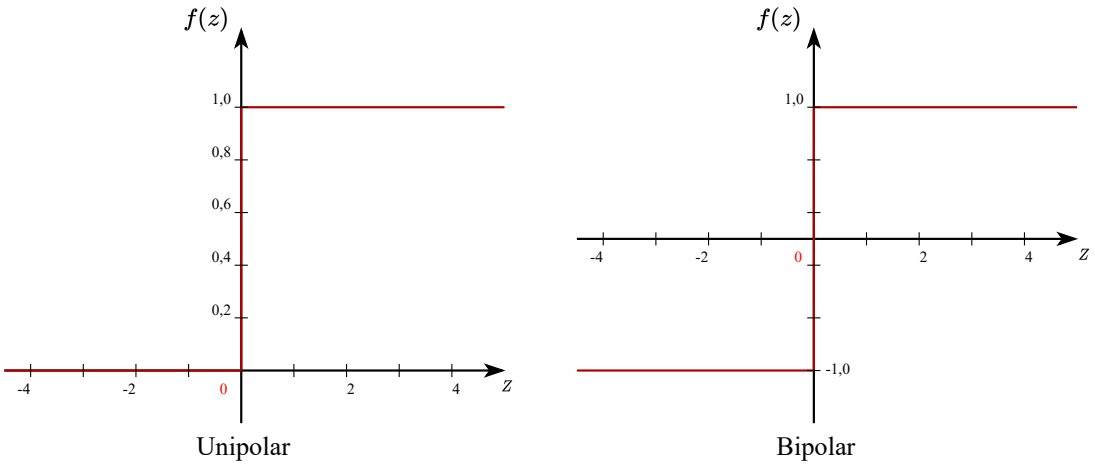


Fig. 1.7: Transfer functions without a threshold

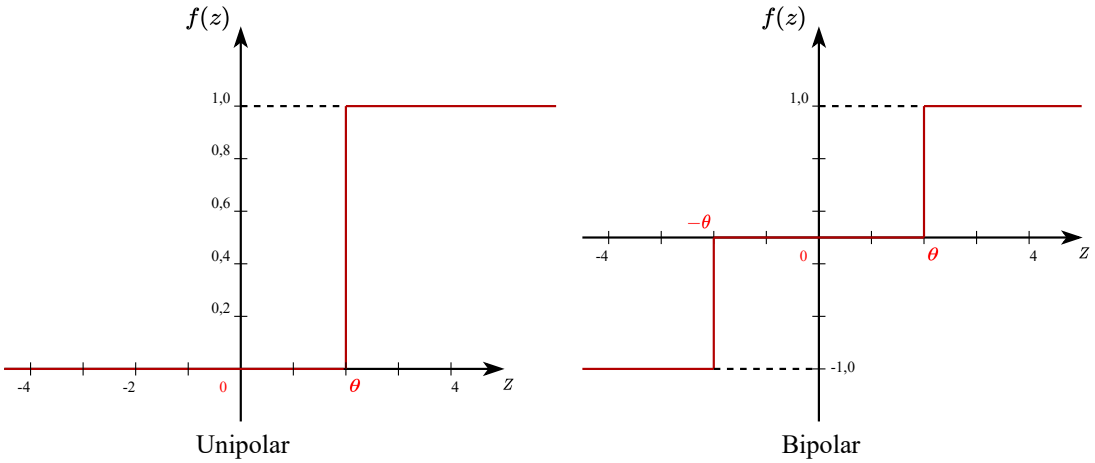


Fig. 1.8: Transfer functions with a threshold

Learning

Linearly separated classes of objects are needed that consist of inputs that the perceptron should sort into classes. This network uses learning with a teacher, which means that to learn the network, one needs to provide the correct classification along with the data set. Then it is possible to find the correct weight vector that separates the values into their correct class in a finite number of loops, see Figure 1.10. Where t is the Number of iterations, s current sample, and p correct result. The starting value of the weights does not play such an important role in the learning process. Learning or training a model is, in simplicity, a desire to reach a global minimum of a loss function. A loss function is the difference between the desired output and the calculated one. Finding the global minimum is achieved by changing the so-called

hyperparameters of a model, in this case the weights. There are numerous functions that exist to solve this problem; such functions are called optimizers.

Overfitting

Training a model on a small variance of input data for too long can cause overfitting. This means that the resulted model will be too generalized on the training data sample. It is preferred to validate the model on a different sample of data to verify that the model is not being overfitted. In Figure 1.9 an overfitted network is plotted in red and a fitted one in green.

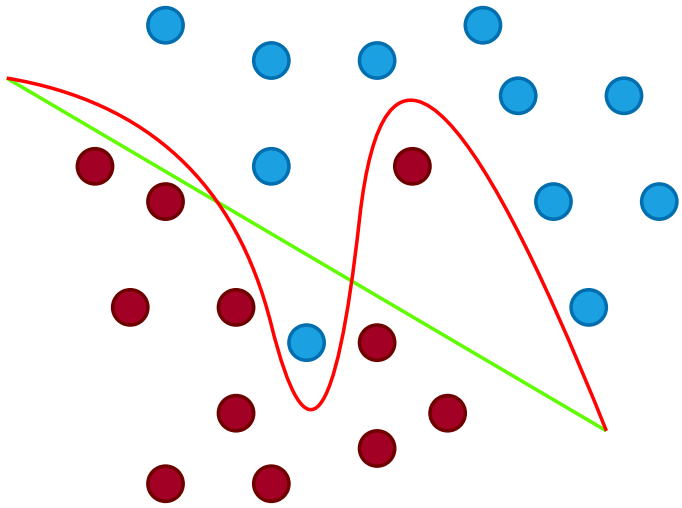


Fig. 1.9: Overfitting

Inference

Inference is a process that tests the learned network on new sets of data. If it has been trained correctly, it should classify these data correctly. Inferring a trained perceptron is done by calculating the output once and observing the output.

Multi layer perceptron

Perceptron has the ability to be combined and layered into a neural network. By doing so, it increases the number of inputs and outputs that it can process, see Figure 1.11. All the principles stay the same, learning is done in the same way as previously, only that the result of the previous layer continues forward to the next one. This topology is called this MLP (Multilayer Perceptron).

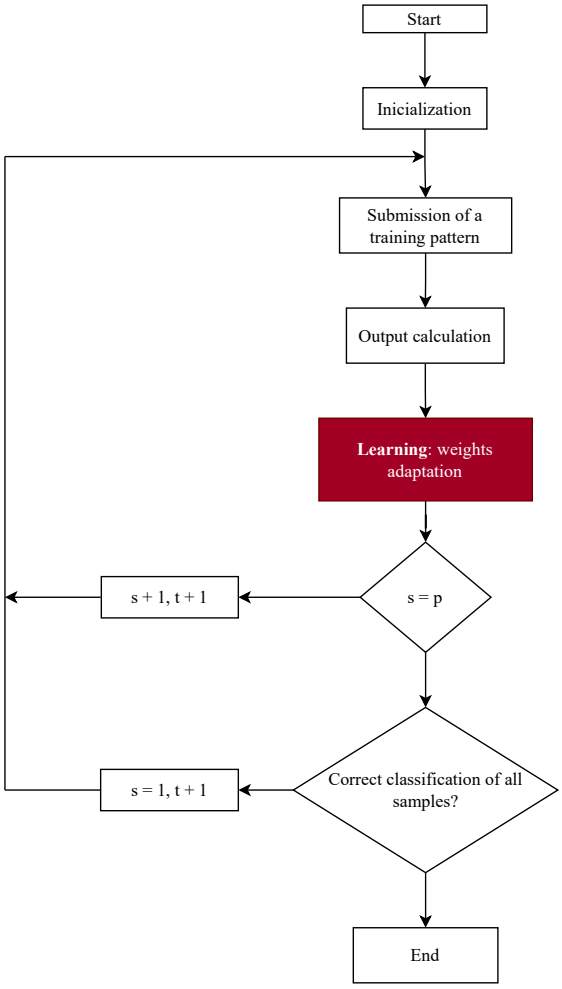


Fig. 1.10: Learning diagram

Perceptron critique

The first Perceptron network was created by Frank Rosenblatt. Considering the age, there has been a long time gap in which machine learning has not really progressed. This can be explained by a simple critique. Perceptron is unable to classify non-linearly separable sets of samples. As a simple example, you cannot train a perceptron to perform the XOR logical function. This has slowed research and affected the motivation for the further development of AI. This problem was later solved by changing the inner function of the neuron to a different shape, such as a sigmoid, and implementing a new learning algorithm called backpropagation [14].

1.3 Machine vision

Machine vision is a term used to describe the process of a machine visualizing, processing and recognizing certain objects or stimuli in an image. Compared to a

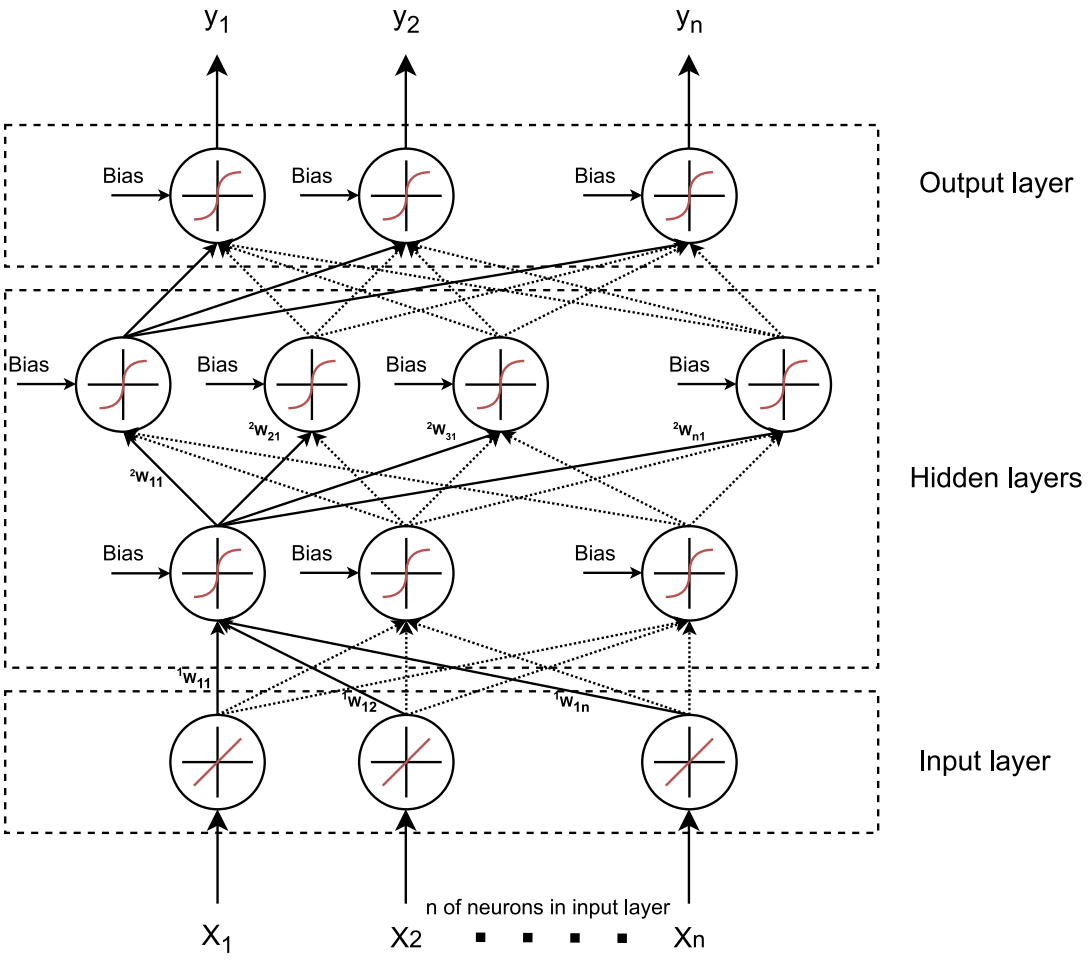


Fig. 1.11: Oriented figure of a neural network

human simply seeing and classifying an object in a real world, is not as easy of a task for a computer. Machine vision does not necessarily need to include machine learning. Rather, it is a concept of using mathematical approaches to recognize or detect an identical or familiar image in an object. The machine vision process can be split into individual stages as can be seen on Figure 1.12:

1. Image capture and digitization - Firstly, the real world needs to be captured and logged into a digital form. This can be achieved by using a camera, for example. This process produces unwanted noise and loss of information that are hard to avoid [15].
2. Image preprocessing - This step is used to make desired features of an image more visible, and vice versa. Processes that fall into this category are: pixel brightness transformations, geometric transformations, and local preprocessing. For example, image noise reduction falls into this category [16].
3. Segmentation - The goal of this section is to separate objects from backgrounds and the preparation for describing the image. It is also used to reduce un-

wanted data [17].

4. Description - The description of an object in an image. It should not be affected by orientation, noise, scale, brightness, or shape[18].
5. Classification - Classification of an image or object into a class.

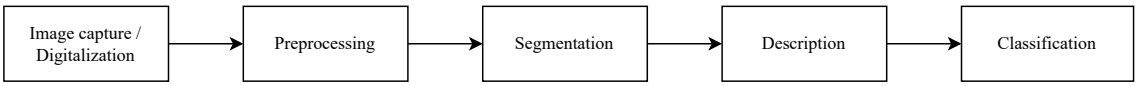


Fig. 1.12: Stages of recognition

1.3.1 CNN

Convolution neural networks are FNNs with a convolution operation instead of matrix multiplication in at least one of their layers [19]. They make use of filters or kernels for convolution Figure 1.13. Convolution is needed because for a network to be able to process a 100x100 pixel image, each neuron would have 10,000 weights on their inputs. By employing cascaded convolution filters, only 25 neurons are required to process 5x5 tiles [20]. One of the methods that can also be used is down-sampling, which effectively reduces the number of pixels in the image. By changing the sequence of convolution, down-sampling, and FNNs can create multiple of models.

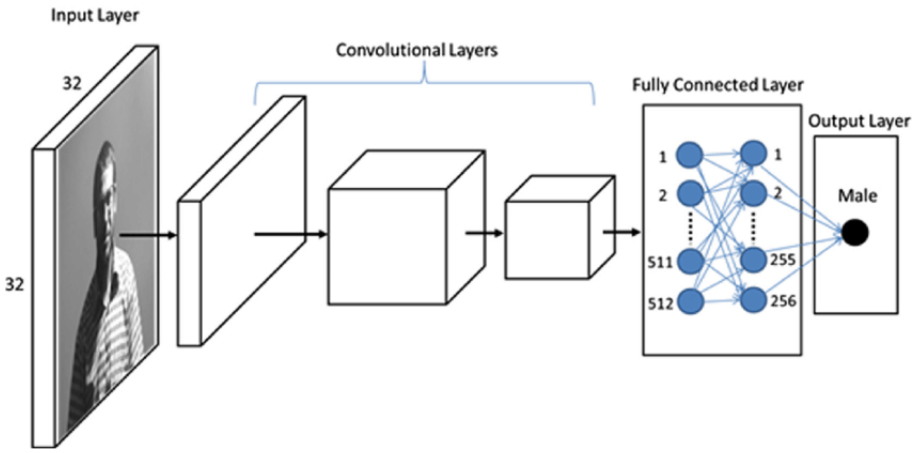


Fig. 1.13: A CNN model with an input image and a class output [21]

1.3.2 Image and object recognition

Image recognition is when an image has a single object or feature that can be categorized into a class. For example, a picture of a dog will be classified as a dog

class, and a picture of a cat will be classified as a cat class. These models are often termed image classifiers. On the other hand, object detection is when a model can recognize a class or a feature in a picture and correctly determine its location in an image. Training these models is more difficult since the object in the image needs to be manually labeled and classified; hence, the creation of data sets is time consuming. These models are often referred to as object classifiers.

1.3.3 CNN models for image classification

There are multiple models that are being used for image classification. The most well-known are MobileNet, ResNet50, and VGG. Mobilenet is the most preferred one in this work because it is designed to be inferenced on less powerful/mobile devices, like those using ARM processors. The newest MobileNet to mention is MobileNetV3 [22] which is tested in this thesis with its predecessor MobileNetV2 [23]. MobileNetV3 is noticeably faster and improves upon the V2 bottlenecks. Other types like ResNet (Residual Network)[24] and VGG19 are also worth mentioning. All of these models can be used in an object classifier such as R-CNN(regional-based convolution neural network)[25] or Retinanet[26].

1.3.4 R-CNN

A Regional Based Convolution Neural Network is a CNN at base, but uses regions proposal to be able to detect objects in an image. This regional proposal takes place before the CNN input [27]. That is why its called a two-stage detection detector or network. The initial regional proposal is handled by one neural network and the class classification is done by a CNN. As time progressed, new versions of R-CNN were created. Firstly, Fast R-CNN and then Faster R-CNN which is the most current version of an R-CNN, although considering its naming, it is still slower than other networks used for object detection, but is considered the most accurate of them all [28] [25]. The way in which the convolution and detection layers are organized significantly changes the way the detector works. For better reference, R-CNN will be referred to as an object classifier or detector and its inner structure as an image classifier or model. This is because at the core R-CNN is still just an image-classifying convolution network with a region proposal.

1.3.5 Retinanet

Compared to R-CNN, Retinanet uses a one-stage approach to object recognition [26]. Its main advantage is in the detection of small objects in a dense environment, so it has found use in aerial photos and medical fields [29]. It uses FPN (Feature

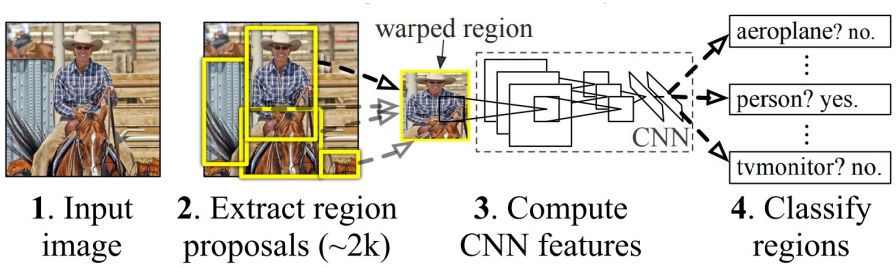


Fig. 1.14: R-CNN object detection [27]

Pyramid Network) and Focal Loss. FPN is an augmentation of a standard convolution network that creates a top-down pathway of the image. This was inspired by resnet architecture that used a bottom-up architecture. This pyramid creates a feature rich multiscale maps at different resolutions from a single resolution input imag, see Figure 1.15 a) to b). This makes the network really good at detecting the same objects at different scales. It is possible to use the detector with different models than Resnet, but the creators recommend using Resnet50 FPN for best results.

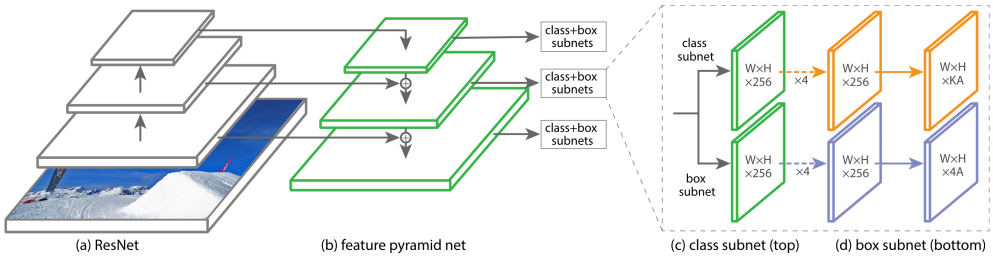


Fig. 1.15: Feature pyramid network example [26]

1.3.6 YOLO

YOLO is an object detection model that utilizes a single-detector approach. This means that a single neural network predicts bounding boxes and objects class directly from a single resolution image in one evaluation. Hence, the name you only look once. The detection pipeline is direct and takes the whole image, divides it into an NxN grid, creates bounding boxes with confidence rating that reflects the probability that that box contains an object, and a class probability map that gives every cell the possibility to contain a class of a certain type. The model then makes a final prediction, see Figure 1.16. This simple pipeline makes the network fast, but not as accurate as other object classifiers. That is because of space limitations, because grids have only two bounding boxes and one class probability, when an

image contains a small enough object in a bigger group, the network will most likely miss it. Using a higher resolution image can help but increases the complexity of the network and its speed [30]. The newest version of YOLO is YOLOV8 developed by Ultralytics released in the current year 2023 [31].

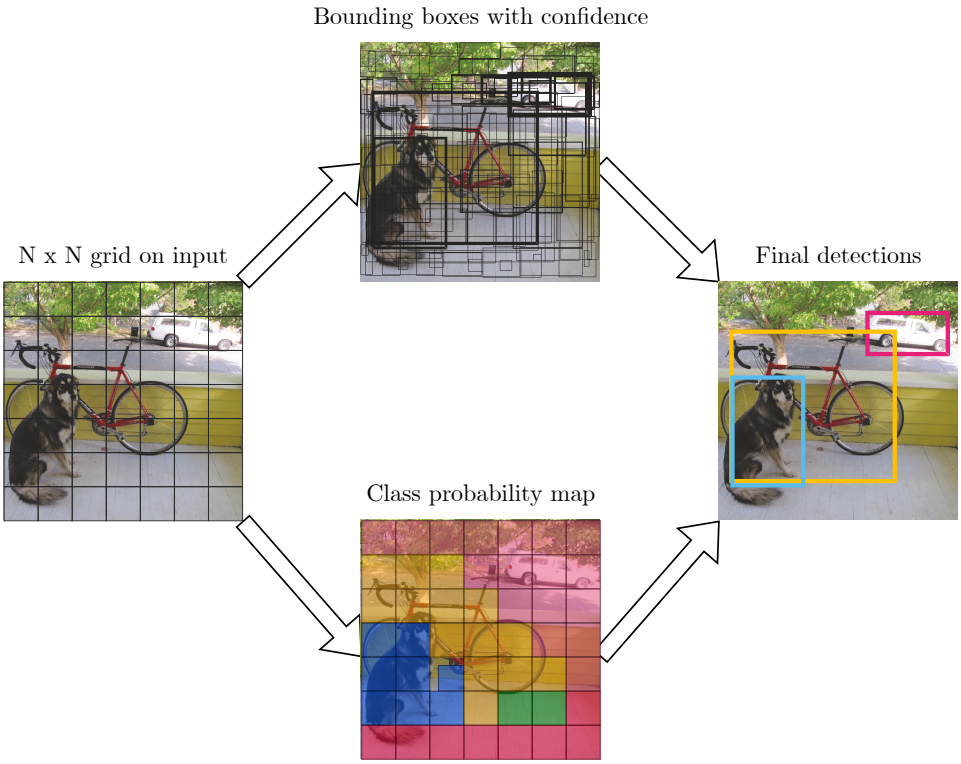


Fig. 1.16: YOLO model [30]

1.4 Cloud edge computing

Cloud edge computing is a way of bringing computing abilities closer to the devices that capture the data [32]. This means utilizing a cloud storage as a database for storing already processed information from the device. For this works example, the determination of what parking place is occupied would be performed on the device that is also controlling the parking lot, then the information would be sent to a remote database server that would contain the information about the parking place occupancy, which then could be retrieved by any device with an access.

2 Code and relevant software tools

2.1 Python

Python is a popular high-level object-oriented programming language that is ideal for large-scale data analysis and machine learning [33]. It is an interpreted programming language, which means that it does not require any compiler. This makes it easy to quickly deploy working scripts. The disadvantage of an interpreted programming language is the need for an interpreter. This can cause version conflicts, missing modules/libraries, and other specific problems that you would not otherwise have on a compiled language.

2.1.1 Jupyter notebook

Due to the way Python works, having the option of having small scripts that could share data with each other would be beneficial. That is where Jupyter notebooks [1] come in handy. It is a Python cell scripting helper. It allows to run scripts that are divided into cells, but runs them all in the same Python interpreter environment with all the same variables. Most of the code used in this thesis is written in modules separated in individual folders. These modules are then imported into Jupyter notebooks and subsequently used in individual cells.

2.1.2 Anaconda

One of the advantages of using Python is its ability to import packages or libraries. These packages are freely available on the Internet. There are tools such as PIP which is a Python package manager that helps to install and manage such packages. The problem arises when, after coding one project that is dependent on one set of libraries and then wanting to switch to another, will create conflicts and result in installing and removing different packages and, or even switching Python versions. A tool like Anaconda, or Conda [2] for short, solves this problem by creating manageable Python environments, and it is able to package the environment for use on different systems with cross-compatibility. It features its own package manager and PIP.

2.1.3 Torch and torchvision library

The torchvision library is a machine vision library from the Torch machine learning framework. Pytorch is its Python equivalent. This framework makes it easy to

[1] https://jupyter.org
[2] https://docs.conda.io

create, train, and evaluate machine learning models in a Python environment. Its machine vision library includes a multiple of pre-trained models from popular papers. Refer to the torch vision documentation for more in-depth information [34].

2.1.4 OpenCV library

OpenCV[3] is a popular open source library used mainly as a tool for real-time machine vision models. It is written in multiple languages, including Python. It includes tools to load images in multiple color formats and provides the ability to draw rectangles, text, and circles into an image.

2.1.5 Flask

Flask[4] is a Python web framework library that allows the user to create simple web servers in Python. This means responding to an HTML request and sending HTML templates. It uses jinja2 template engine to create HTML templates. Compared to a framework like Django, Flask is more Python-like and is easier to get started with. If a user needs, there are many community-made extensions that extend the set of features. For example, adding the ability to access and manage a remote or local database.

2.2 Comet

When creating and training an ML model, it is often beneficial to record the progress of training and validation in graphs so that they can be compared later with different versions and iterations. An online cloud service called Comet helps to track these metrics. A Python library is used to communicate and send metrics data to the cloud during training; these data can then be analyzed live in a Web GUI (Graphical User Interface), as can be seen in Figure 2.1 [35].

2.3 Datasets

Datasets are needed for training custom ML models. When talking about object recognition, data sets are considered to be sets of images with annotations that label an object in an image and specify its class. For parking occupancy detection, a data set containing parked cars is preferred. There are many data sets available freely on the Internet, but it is always recommended to create one that fits the desired use

[3]https://opencv.org
[4]https://flask.palletsprojects.com/en/3.0.x/

Fig. 2.1: Example of Comet web interface

case. All images shown have ROIs (regions of interest) plotted in them. ROIs are generated from annotations and need to be rectangles; that is why in most cases the boxes overlap.

There were 4 existing datasets considered for use in this thesis:

- **PKLOT** - This dataset contains 12,417 images of a parking lots of universities in Brazil. The images were acquired every 5 minutes for a period of more than 30 days. The photos were captured by a high-definition camera Microsoft LifeCam HD-5000 in a lossless quality with a resolution of 1280x720 pixels. It contains pictures during different weather conditions, such as rainy, sunny, and overcast. All images were annotated by a human [36]. See Figure 2.2 for an example of annotations. This dataset was chosen due to the height and angle at which the photos were taken, presumably from a window.
- **ACPDS** - Contains 293 images captured at a 10 meter height with a GoPro Hero 6 camera with a resolution of 4000x3000 pixels [11]. It contains a variety of parking lots and has been considered in this work due to the specific angles it offers. An example of an image from the dataset with ROIs Figure 2.3
- **CNRPark-EXT** - This dataset contains a total of 12,000 annotated images; it features 4 different camera angles on a car park that includes trees and street lightning lamps that behave as obstacles [37]. Refer to 2.4 for an example.
- **COCO** - COCO (Common Object in Context) is a data set created by Microsoft to train object recognition models in the context of a scene [38]. It features 328,000 images with 2.5 million labels of 91 different object classes with a short text description that gives context to the scene on the image. Aside from featuring object localization labels, it contains semantic segmen-

tation maps that precisely copy the shape of the object; this can be used in segmentation models. The data set is featured in this work because most of the pre-trained models featured in the TorchVision library are trained on a COCO data set.

Only three were selected from this list. That is PKLOT for it's huge collection of images, CNRPark-EXT for the inclusion of different weather conditions and COCO as torch-pretrained models are trained on it.

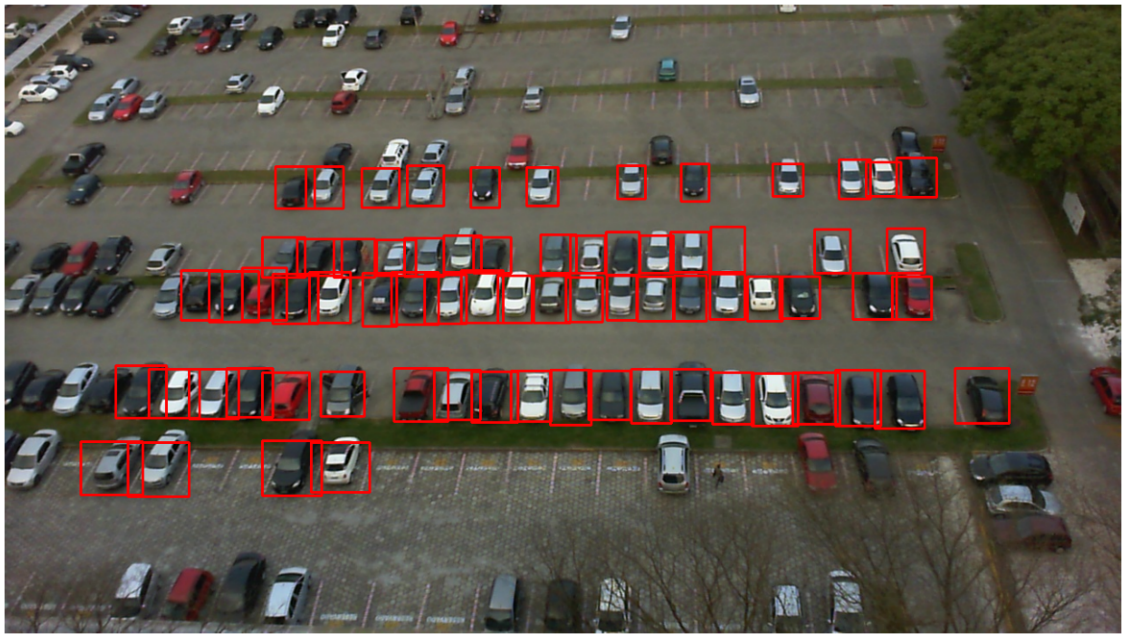


Fig. 2.2: Example of an annotated image from PKLOT dataset

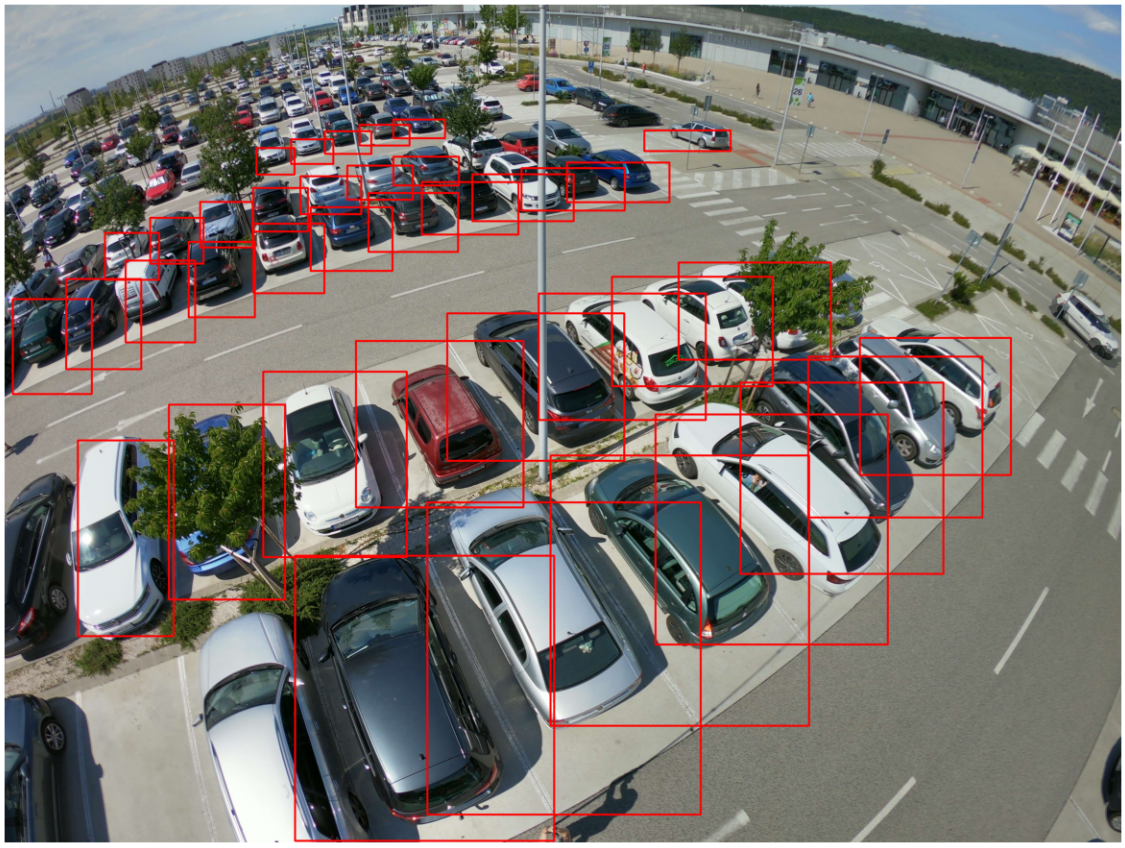


Fig. 2.3: Example of an annotated image from ACPDS dataset

Fig. 2.4: Example of an annotated image from CNRpark dataset

Fig. 2.5: Example from the COCO dataset with semantic segmentation [38]

3 Solution proposals

3.1 Creating a dataset

When creating an ML model that has a specific use case, in case of this thesis; parking occupancy detection, it is beneficial to create a dataset not only for training but also for validating and testing of the model. Since a video is just pictures captured in fast succession, instead of recording a parking place and then obtaining thousands of images, the dataset should consist of images that are different in some way from each other. Therefore, it is better to set up a device that captures a photo after a certain period, for example, every 30 minutes.

3.1.1 Data acquisition

Due to the fact that the parking lot on which this ML model will be applied is next to a university building and there are no usable CCTV cameras that would capture the parking lot. The images would be taken from the top floor of that university building. Firstly, an iPhone X was chosen as the capturing device with a time-lapse application that captured an image every 30 minutes throughout the day. This was done during the late autumn and winter months, from November to early January. The device was mounted on a 3D printed stand that was rigidly fixed to a window frame, see Figure 3.1. Since the camera needed to be placed behind a window, a black paper was placed under the stand to reduce reflections caused by the stand. This fact also brought about an advantage because during a rainy day, water droplets would appear on the outside of the glass, which simulates how moisture could accumulate on a camera lens.

The dataset also includes a couple of images taken at night; these photos were cherry picked to include only those in which a human can differentiate a parked car. Such images were included in both the training and the testing set of images, see Figure 4.5. A total of 83 images were captured this way.

The second part of this data set was captured using a single-board computer Raspberry-pi V4 with a Pi Camera Module v2 and both were mounted on the same arm using a 3D printed adapter. The camera module captures photos in higher resolution, precisely 3200×1800. Note that all photos are kept in 16:9 aspect ratio. All images are then scaled down to match the resolution of 1920×1080 px. More than 200 images were produced from which about 117 images were used in the dataset, which brings the total number of images to 190. These images were captured through the months of April up to the beginning of May.

All images were labeled and annotated using the included tools, and the resulting dataset is called T10LOT and can be downloaded from a shared Google drive that can be found in the GitHub repository[1]. of this work.

Fig. 3.1: iPhone capturing setup

Fig. 3.2: Raspberry Pi combo mount

3.1.2 Dataset structure creation

A dataset structure from [10] was used with small modifications to its file structure. A dataset root directory contains two folders:

1. **The main dataset folder** has the same name as the dataset, and it should contain all images of the dataset in a "images" folder. The rest of the data set structure is generated later by the annotation widgets.
2. **Annotation classes**, as the name suggests, houses images split into visual condition classes, such as "sunny", "fog" and "winter", for example. The images must be manually sorted and placed here. Currently, this directory is not used, other than for user information or testing purposes.

3.1.3 Labeling

Annotation and labeling of captured images were done using a Jupyter widget application inspired by [10], their work focused on testing multiple different models with a custom dataset structure. The annotation script was rewritten to work with the latest Python version with a different file structure. Since the camera remained stationary, a single map was applied to all images using the apply to all button; see Figure 3.3. After annotating individual parking places, the labeling widget Figure 3.4 was used to label parking places occupied (marked in blue) or busy (marked in red). The widget then creates the desired dataset folder structure with all the

[1] https://github.com/slavajda02/parking-research-argon

generated outputs. A more in-depth guide for the use of both annotation widgets is available in a GitHub repository[2].

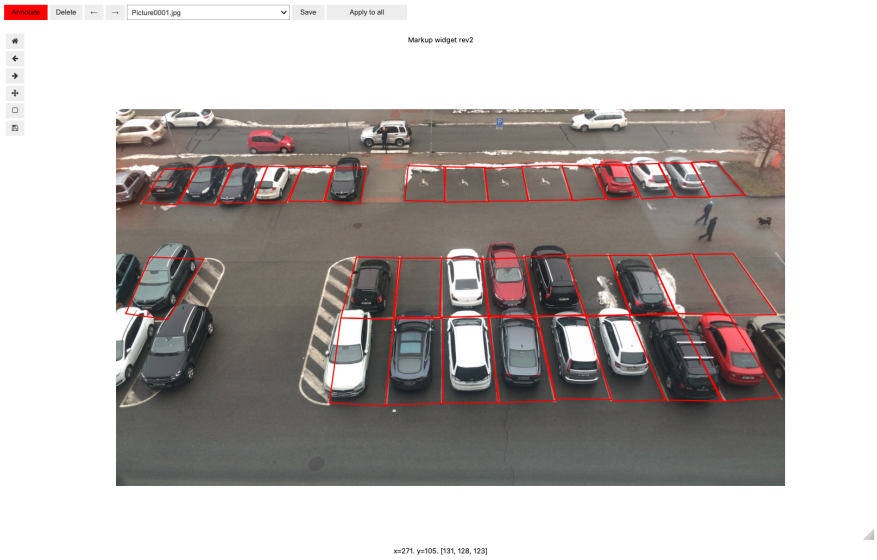


Fig. 3.3: Annotating widget A

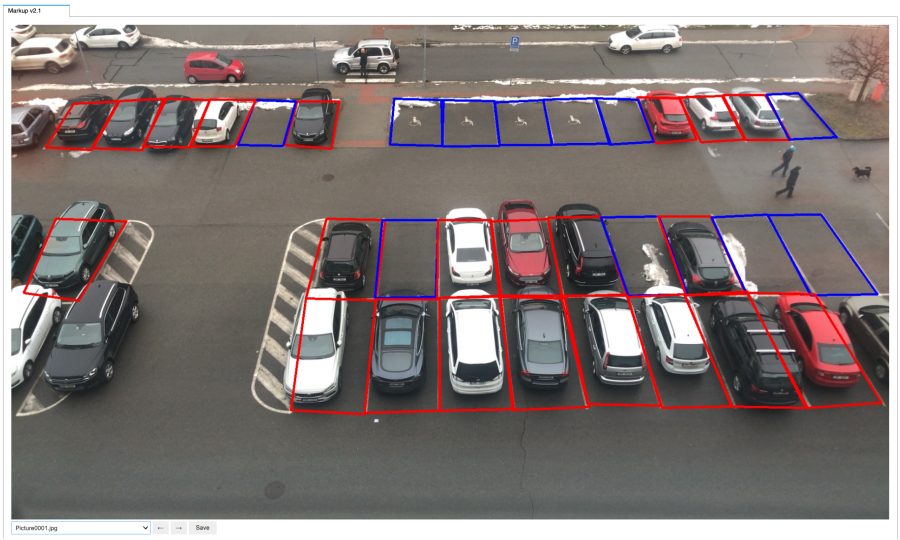


Fig. 3.4: Annotating widget B

3.2 Creating an ML model

Due to the large number of different models available to test and use. A training script was written that helps with the creation and subsequent training of the model

[2]https://github.com/slavajda02/parking-research-argon

on a wanted dataset. The training script uses a training method inspired by [10] since they used object detection and image recognition with models: Resnet50, MobileNetV2 and VGG in combination with object detectors Faster R-CNN and ResNET. Most of the train loop code has been rewritten to suit this thesis, but the dataloader class has been kept the same, as the work uses the same dataset structure. The training script along with the libraries used is available in a GitHub repository[3] and is a fork of the original repository of the mentioned authors.

3.2.1 Using a pre-trained model

When using the torchvision library, there is an option to use models with pre-trained weights. Most of these models are constructed according to their original papers, although in some cases the models are improved to be faster or more accurate. Using a pre-trained model is quite simple, but differs from model to model. Most often, the library provides a method that constructs the given model with a custom number of output classes. This means that there is no need to edit the model's structure to accommodate the pre-trained weights. Listing 3.1 describes the way to obtain a pre-trained model class of a faster R-CNN detector with a mobilenet V3 large backbone with 2 output classes, one for a car class, the other one for a background.

```
from torchvision.models.detection import fasterrcnn_mobilenet_v3_large_fpn

model = fasterrcnn_mobilenet_v3_large_fpn(weights = "DEFAULT", num_classes = 2)
```

Listing 3.1: Obtaining a pretrained model

```
from torchvision.ops import MultiScaleroIAlign
from torchvision.models.detection.rpn import AnchorGenerator

backbone = torchvision.models.mobilenet_v3_small().features
backbone.out_channels = 2
new_anchor_generator = AnchorGenerator(sizes=((32, 64, 128, 256, 512),),
                                       aspect_ratios=((0.5, 1.0, 2.0),))
new_roi_pooler = MultiScaleRoIAlign(featmap_names=['0'],
                                    output_size=4, sampling_ratio=1)
model = FasterRCNN(backbone=backbone,
                   num_classes=2,
                   min_size=min_size,
                   max_size=max_size,
                   image_mean=mean,
                   image_std=std,
                   rpn_anchor_generator = new_anchor_generator,
                   box_roi_pool=new_roi_pooler)
```

Listing 3.2: Example of model creation

[3] https://github.com/slavajda02/parking-research-argon

3.2.2 Constructing a custom model

Construction of a desired model is handled in the "inter_utils.py" file. Listing 3.2 is describing the creation of a custom mobilenetV3 small model with a Faster R-CNN detector without any weights. First, using the torchvision library, the backbone of the model is obtained; it contains the convolution layers and features of the network. Then an anchor generator is needed along with a ROI (Region Of Interest) pooler. The anchor generator defines anchors for ROIs in an image, ROI pooler is a function that aligns ROIs for these anchors. Then lastly calling the Faster R-CNN method from the torchvision library creates a class of the model that has methods for inferencing the model. This class is later used in both the training scripts and testing scripts. The torch library also features a function that has the ability to save this model class to a state dictionary file, so that a model can be later loaded in a different instance.

3.2.3 Training

Training an object recognition model with the use of the torchvision library goes as follows:

1. Defining a dataset class that has a "___getitem___" method which returns dictionaries of images and targets, for both the training and validation batch of images.
2. Create a data set loader class from the torchvision library and pass it the dataset classes.
3. Defining an optimizer that changes the model parameters through the learning phase.
4. Running the training loop, which consists of feeding a batch of images to the model using the dataset loader class, returns its calculated loss. The loss is then used by the optimizer to optimize the model parameters. A batch of images is used during a single iteration. The batch size depends on the hardware on which the network is being trained. A bigger batch will require more VRAM (Video Random Acess Memory).
5. Validating the model on the validation dataset. The loss function from this loop suits as a metric for the user to determine whether the model is learning and not overfitting.
6. Running both the training and validation loops for x amount of epoch. The number of epochs depends on the size of the dataset and the model used. A smaller dataset will require more epochs, and vice versa.

All of these steps have been incorporated into an automatic training function that allows the user to easily train a model from a list of available models. In the

beginning, the training script asks the user a couple of questions to configure the training Figure 3.5. After that, the script loads the selected dataset and trains the network on the selected datasets. The command-line application then saves the final model state into a file which can then be used for the deployment. Figure 3.6 shows a simple flow chart that describes the training script. The script also features a logging feature. The train progress is sent and logged on the Comet cloud service as mentioned in Section 2. Comet automatically plots metrics like training loss and validation loss.

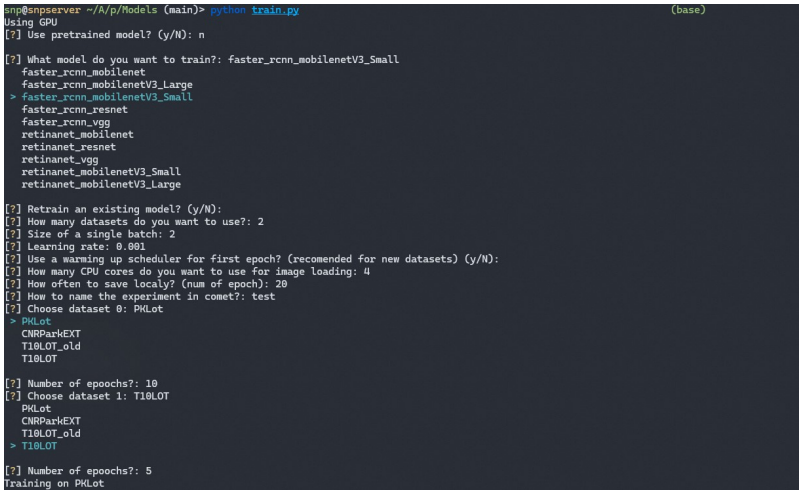


Fig. 3.5: Training script prompts

3.2.4 Retraining

Retraining a model with pre-trained weights is really similar to training a model from random weight values. This means that the only difference is in creating the model class, then all that is needed is to run the standard training and validation loop. The training script also has this incorporated.

3.3 Hardware setup proposal

When deciding what hardware components to use, it is necessary to remember that inferencing an AI network is computationally difficult. Due to this reason, it is proposed to use a Raspberry Pi 4, which is a powerful single board computer that utilizes an ARM (Advance RISC Machine) considering its size. When combined with a camera, such as the Raspberry Pi Camera v2 which uses a Sony IMX708 sensor with 12Mpx, it could be used to continuously monitor a parking lot. This setup could be enclosed in a 3D printed box. When considering cloud edge computing, the

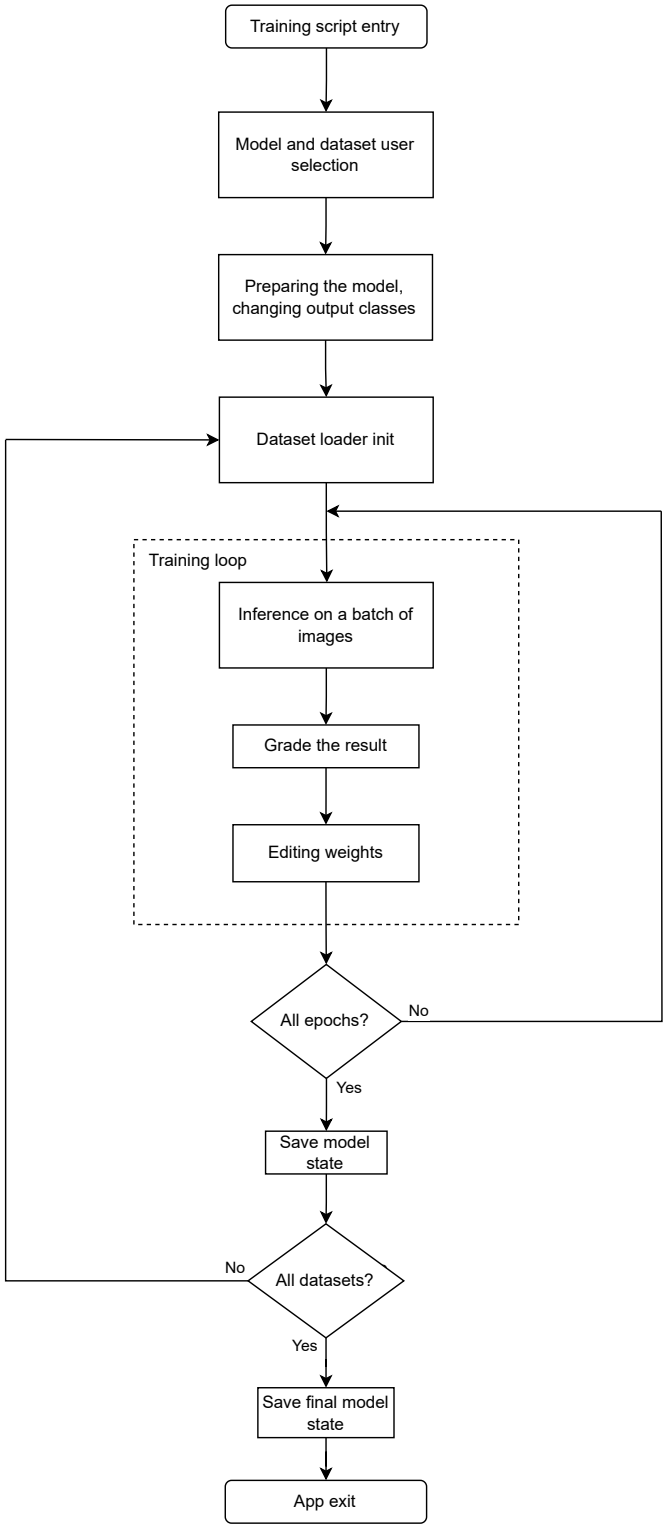


Fig. 3.6: Training script flowchart diagram

Raspberry Pi is equipped with a WiFi interface for connectivity; this means that there would be no need to provide an Ethernet connection and just an adequate power supply would suffice.

4 Testing results

A mulitple of testing was done to figure out what model combination would be the most effective and accurate. From the models mentioned in the theory Section 1.3.3 of this work, three models were chosen. The MobilenetV3 small, Mobilenet V3 large and Resnet50. All of these models were tested on Faster R-CNN and Retinanet. Two pre-trained models using the COCO dataset were also used: FasterRCNN MobilenetV3 Large FNP and FasterRCNN resnet50 FPN V2, which were both provided by the Torchvision library.

Models that were trained from scratch were trained with datasets that go in this order:

1. **PKLot** - 35 training epochs
2. **CarParkEXT** - 50 training epochs
3. **T10LOT** - 100 training epochs

Such epochs numbers were chosen after seeing that further training epochs have almost no improvement on the loss progression.

4.1 Training results

4.1.1 Pretrained model

When looking at Figure 4.1, it is possible to see the training progress of a pretrained Faster R-CNN MobileNet V3 large model on a T10LOT dataset. First, the loss is really high and eventually drops while going through the first 10 epochs. In Figure 4.2 you can see the loss progress relative to the image batches when one batch is 4 images. If the dataset were to be larger, the loss function would approach its minimum faster. Training was stopped after 50 epochs, as there was no more significant improvement; because of this, the time to train is short relative to training on a larger dataset.

4.1.2 Custom models

Following the training order of the datasets, training took about an hour on an RTX 2070 super GPU. Information about the minimum loss achieved on a T10LOT dataset is in Table 4.1.

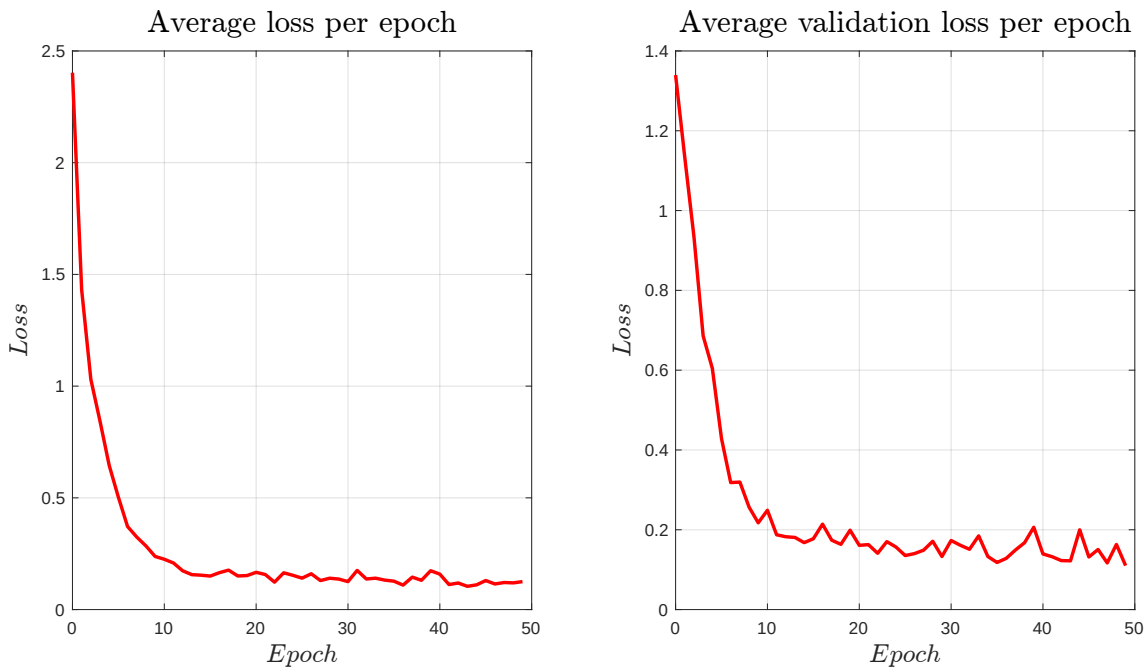


Fig. 4.1: Average losses per epoch

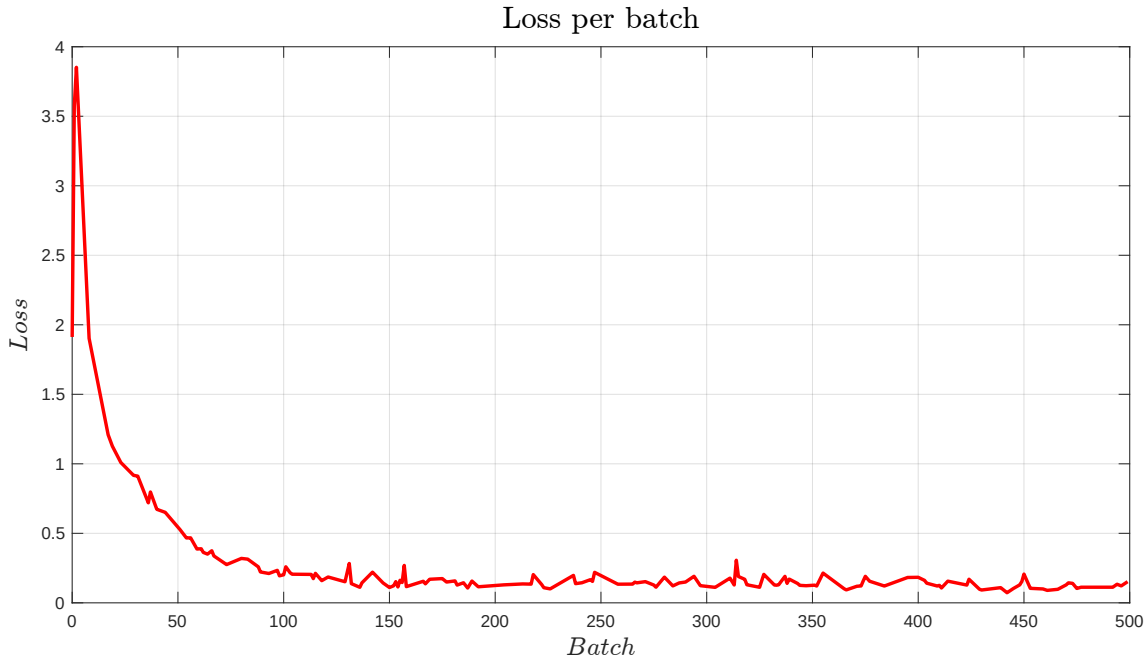


Fig. 4.2: Training loss per batch

4.2 Testing

4.2.1 Testing methods

The verifying of an object detector can be quite complicated. It is possible to look just at the result of a loss function when evaluating the model on a testing data set. However, this does not fit the use case. Because the goal of this model is to detect a car in a parking spot, the following method was proposed. Since the result of an inference is a box surrounding a detected object, it is possible to use the middle point of this rectangle to check whether it lies inside a labeled parking lot or not. The correct parking lot bounding boxes come from the T10LOT dataset. Knowing this information, it is possible to sort the different types of detection into categories:

- TP (True Positive) detection - A car detected where a car is
- FP (False Positive) detection - A car is detected where a car is not
- FN (False Negative) detection - A car is not detected where a car is

Using this information, one can calculate the F1 score of the model with the Equation (4.3). This is an error metric that measures the model performance by calculating the harmonic mean of precision and recall for the minority positive class. It takes into account both the recall Equation (4.2) and precision Equation (4.1) ability of the model. The resulting number is the measure of the overall performance of the model ranging from 0 to 1, where 1 is the best [39]. For this use case, a script was written that uses a trained model, tests it on the testing batch of images from a dataset, and calculates the model's F1 score. The script has the option to save the test results in images as can be seen in Figure 4.3. This figure intentionally features a badly trained model to demonstrate visualization. The red dots are the centers of the model detection, while the squares are the parking slot locations with cars present. If a detection is present in this space, it gets classified as a TP detection. All of the testing was performed on an RTX 2070 super GPU, this should be taken into consideration when judging the inference time. It is excepted to be longer on weaker hardware, especially ARM CPUs.

$$precision = \frac{TP}{TP + FP} \tag{4.1}$$

$$recall = \frac{TP}{TP + FN} \tag{4.2}$$

$$F1 = \frac{2 \cdot precission \cdot recall}{precission + recall} \tag{4.3}$$

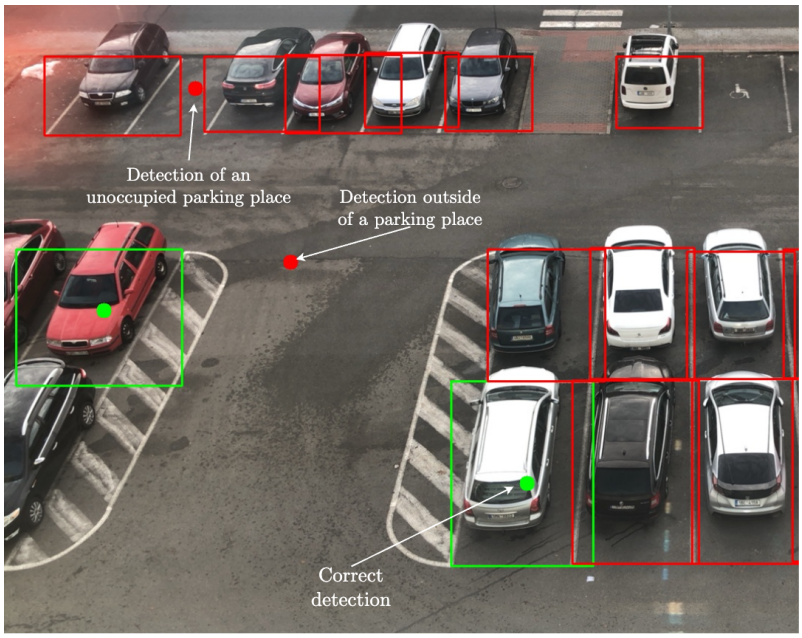


Fig. 4.3: Example of testing result

4.2.2 Testing results

When directly comparing pre-trained models with custom-created ones in Table 4.1, the MobileNet V3 large with Faster R-CNN is more accurate than a custom-created mobilenet V3 large and small, the pretrained RetinaNet is not as accurate as the custom-created one. The inference time is fastest on the MobileNet V3 small model. There is a high chance that it could become more accurate if re-trained on a larger dataset. Another thing to note is that the model with pre-trained weights was more likely to detect a car that was not in a parking place, compared to models that were trained only on the parking lot datasets. The custom made ResNet50 Faster R-CNN network performed the worst, this could be due to the training not lasting long enough or the use of a too small of a dataset. The bar graph Figure 4.4 compares the accuracy and inference time of the models with different detectors.

Individual test images included difficult scenarios, such as water droplets in front of the camera and different lightning conditions as a result of a change of time. Even through such complications, the models performed mostly without any problem. Some models performed shockingly well on dark images such as mobilenetV3 Figure 4.5. Images that include a difficult scenario are sorted in folders according to the condition. This directory is named "annotationy_classes" and is in the root folder of the data set provided. It should be noted that the faster R-CNN models were more likely to focus on detecting only cars that were on a parking spot, not approaching or parked cars; this could have been due to the way two-stage detectors work, and their region proposal stage was optimized for the current parking lot.

When tested with a slightly different camera angle but the same location, the models ability to detect cars was not significantly impacted, which shows that the models are not over fitted and a small camera shift will not impact the models accuracy. When the models were tested on high-resolution images captured at completely different angles, the models were unable to correctly detect the cars. This shows that when retraining a model on a small dataset, it will mostly work only on that use case. All images that are the result of the testing are included in the electronic attachment located on Google Drive.

Model	Pretrained	Detector	Training loss	Validation loss	F1 score	Inference time
MobileNetV3 small	NO	FasterRCNN	0.0236	0.0256	0.972	11.7 ms
MobileNetV3 large	NO	FasterRCNN	0.0259	0.0284	0.970	13.8 ms
ResNet50	NO	FasterRCNN	0.481	0.431	0.218	16.4 ms
MobileNetV3 small	NO	RetinaNet	0.0136	0.0237	0.881	10.2 ms
MobileNetV3 large	NO	RetinaNet	0.0134	0.0232	0.873	15.1 ms
ResNet50	NO	RetinaNet	0.0949	0.0984	0.206	35.9 ms
MobileNetV3 large	YES	FasterRCNN	0.078	0.120	0.990	22.5 ms
ResNet50V2	YES	FasterRCNN	0.056	0.0637	0.962	97.9 ms
ResNet50V2	YES	RetinaNet	0.047	0.0758	0.961	73 ms

Table 4.1: Table of results for individual models

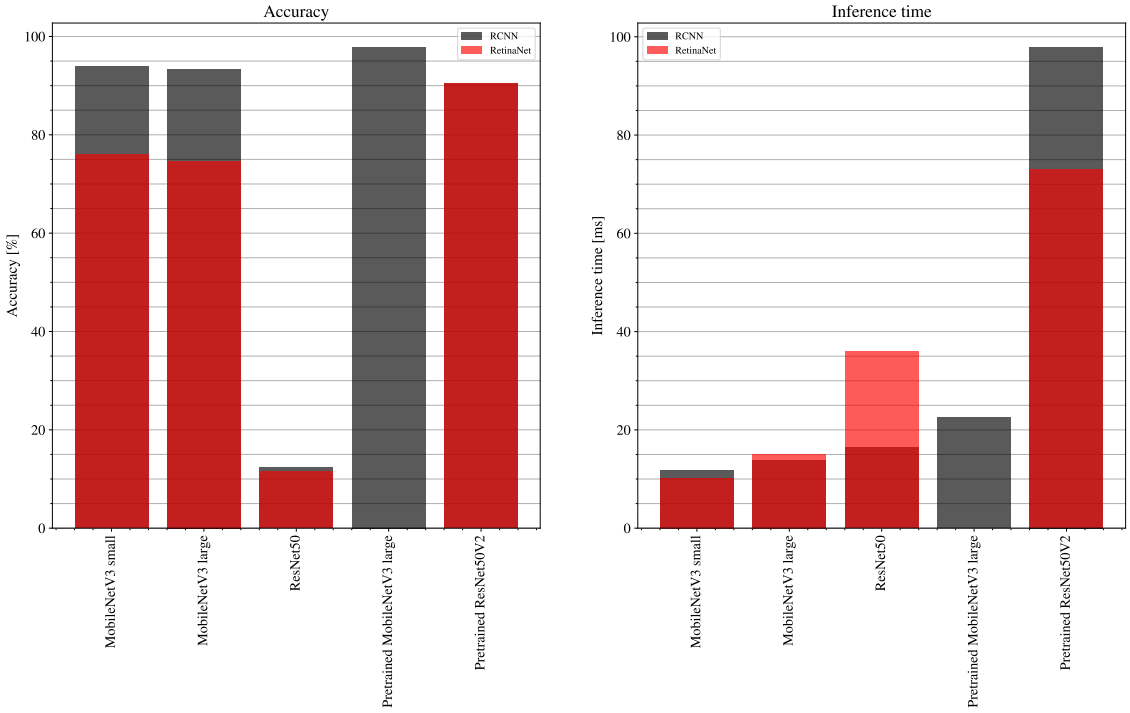
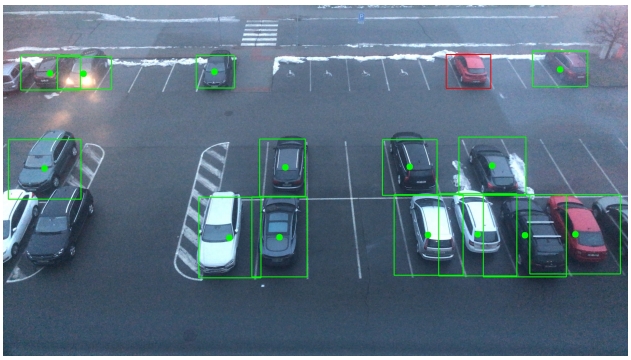
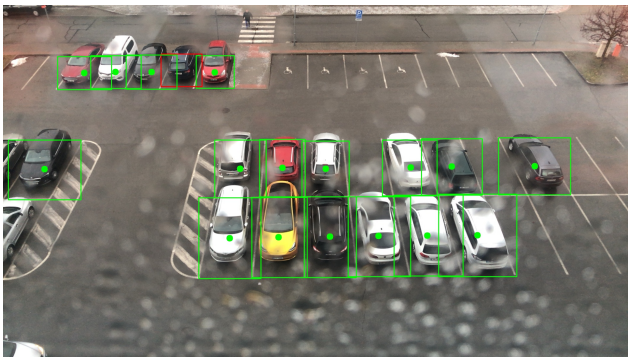


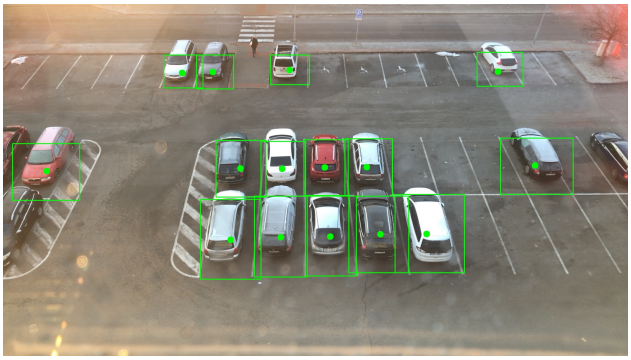
Fig. 4.4: Bar graph of accuracy and time



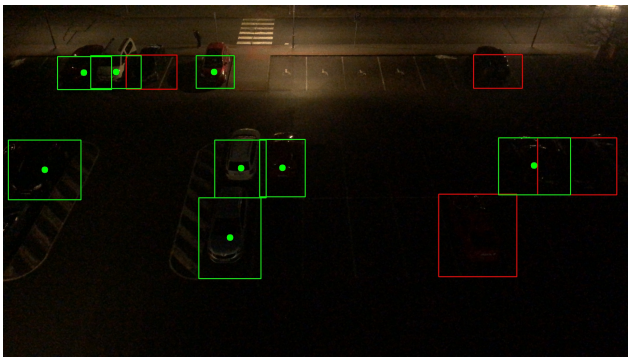
Fog



Water droplets



Sun glare



Night

Fig. 4.5: Example of a testing output on a difficult scenarios

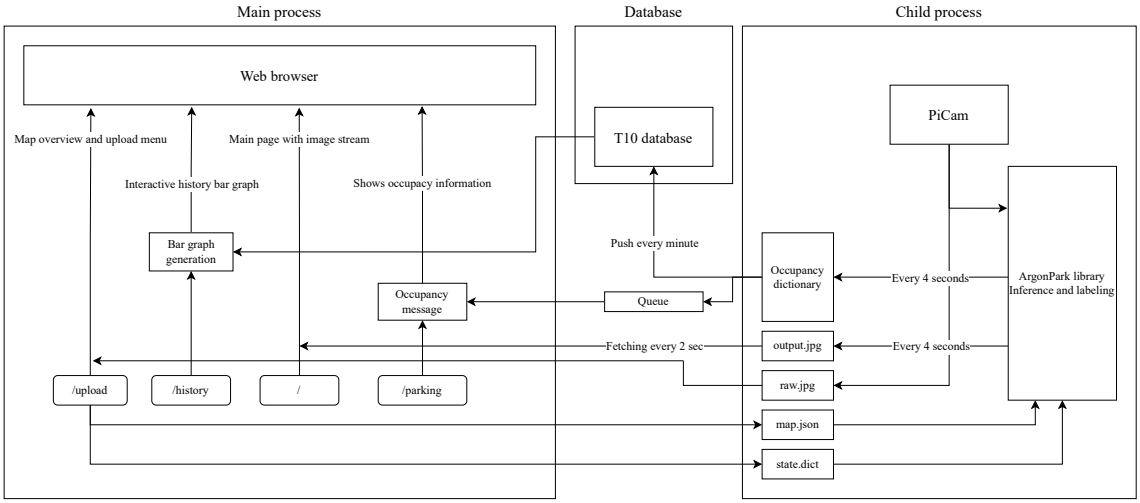


Fig. 5.1: Web application diagram

5 Solution

5.1 Hardware choices

After considering some factors, the final setup is made out of a Raspberry-pi V4 with a Pi camera module v2. This hardware combination is mounted on the same arm as before with the same 3D printed mount as mentioned in Section 3.1.1. After testing the camera module v2 it was decided that such resolution will be enough for this use case, the field of view is very similar to the iPhone X which was used before as a capturing device.

5.1.1 Remote access

Since the Raspberry Pi is in a remote location and is not connected to any HID (Human Interface Devices), a remote access method was devised. Specifically, the use of a VPN (Virtual Private Network). A VPN client connects the Raspberry Pi to a virtual subnetwork hosted by a remote server. Other users can then connect to this VPN and access the Raspberry Pi over the network securely. The VPN in question is called WireGuard [1], which is a simple, secure, and fast VPN that is relatively easy to set up on almost any platform. After connecting the Raspberry Pi to a VPN server, an SSH (Secure Shell) connection can be established from any client on the same virtual network. With such a connection, an application like VSCode [2] can be used for remote development and application deployment.

[1] https://www.wireguard.com
[2] https://code.visualstudio.com

5.1.2 Position and mounting solution

For the ease of testing a same location and mount was used as during image capture for the T10LOT dataset. The 3D printed articulated arm and mount remained the same as shown in Figure 3.2 and are available to download on a Printables website[3]

5.2 Software

In addition to the libraries mentioned in Chapter 2, some new libraries were added. Most notably, the picamera2 library, which is used to interface and control the camera connected to the Raspberry Pi. The main advantage of this version when compared to the first one is the ability to run on a 64bit architecture, which the Raspberry Pi 4 has. This then allows for the use of the torchvision library, which does require such architecture. The last thing to mention is the Plotly library [4], which is used to generate interactive figures that can be run in the Web browser of the users thanks to Javascript.

5.2.1 ArgonPark library

ArgonPark is a custom written library that uses the trained model to classify parking spaces for the occupants. Methods from this library takes an input image and a parking lot map that can be created using an included Jupyter notebook script and inferences the trained machine learning network. When first calling the class of this library, the user needs to provide a map.json file that contains the locations of parking spots and a state_dict_final.pth file that contains the final-state dictionary of a trained model. The class can then calculate the overlap and decide if a specific parking lot is occupied or not from a provided image. All of this information is returned as a dictionary. There is also an option to generate an image with free and busy parking spaces that can be used for visualization purposes. The library was written with docstring comments that explain each method and it's input and return variables.

Area calculation

After evaluating an image, the network outputs two objects. One is a list of boundary boxes that contain a specific class and a precision float. The boundary boxes are squares that are not perfectly aligned with the cars and may protrude or crop a part

[3] https://www.printables.com/cs/model/892765-raspberry-pi-4-and-camera-module-v2-holder-with-a

[4] https://plotly.com/python/

of the car. Parking spaces, on the other hand, are marked as polygons. To try and solve the problem where the detection boxes partially overlap neighboring parking spaces or do not completely overlap the parking space Figure 5.2, an intersection area calculation was introduced. After which it is possible to calculate the IoU (Intersection over Union) Equation (5.1), this value can then be used to make the final decision about the slot being occupied or free. It is also possible to compare the neighboring parking lot IoU to see if there is no incorrectly parked car. An example would be a car parking on the dividing lines, making its boundary box occupy about 50% of the area of two parking slot polygons.

$$IoU(A, B) = \frac{(A \cap B)}{(A \cup B)} \tag{5.1}$$

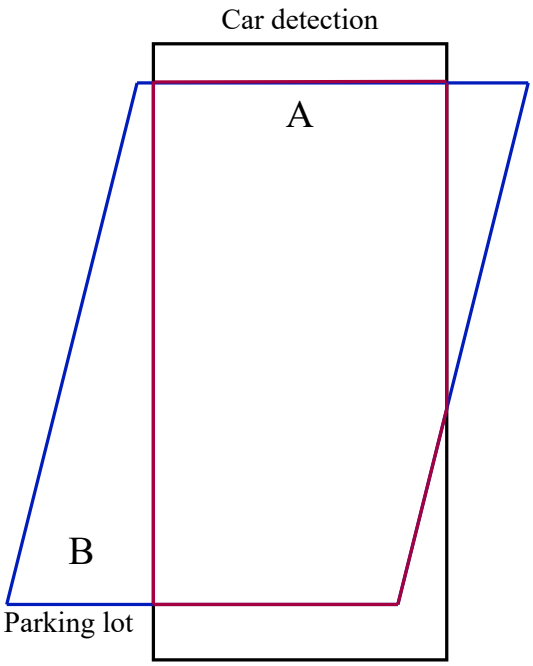


Fig. 5.2: Intersection over union

5.3 Web server

ArgonPark library facilitates a way of easily evaluating parking spaces in a known parking lot. But a Python script is still needed to perform the actions a user wants. That means that another application needs to be created that allows the user to easily interface with the ArgonPark library. One of the ways to do this is to use a website that has a simple GUI interface. For this a web server application was created. A user can then load a website that shows them the image of the parking

space, history, and the ability to change both the map of the parking place and the model weights.

5.3.1 Multiprocessing

Running the inference will hold a Python process for longer than 4 seconds. This is unacceptable when there is also a web server running in the same process. To solve this problem, a new process is created on another core of the Raspberry Pi. This is done through a Multiprocessing Python library, which provides a way to interact with python processes running on different logical cores. After using this library, the entire ArgonPark library is initialized on a new core while the webserver is running in the main process. Every time the child process finishes a new inference, it pushes data to a database and a queue, which is a FIFO (First-in, First-out) type of memory, and the parent process reads this information. The Web server can also request a photo of the parking plot by setting a dedicated flag. This invokes the child to save an image with parking information to a specified folder, which is then read repeatedly by the client's browser.

5.3.2 Database

A database is used to store the occupancy status of all numbered parking spaces in the parking lot. This data is saved every time a new camera image is pulled and saved. This occurs approximately every 4 seconds, due to the speed of the computer. Data are saved according to the time they have been acquired. The project uses a MongoDB database. This is a no SQL type database widely used in the data analysis and IoT sector. Currently, the database is hosted off-site in the cloud on an Atlas service [5]. The database can be easily transferred to a local one, as hosting the database on the Raspberry Pi. The Web application sends data to the database every minute. The following data are sent:

- Timestamp of the inference
- Then for each parking slot:
  - A parking slot number.
  - 4 points coordinates representing the parking slot polygon.
  - Occupancy status as a Boolean value.
  - IoU float value.

This information can then be fetched by any other application/device. For example, a car approaching the parking lot can request the data from the database and know which parking slot is occupied or not and show this information to the driver.

[5] https://www.mongodb.com/en/atlas

5.3.3 User interface

Fig. 5.3: Main page of the web application

The site features a simple user interface using a popular CSS front-end toolkit called Bootstrap [6]. This ensures that the site adapts to different screens, such as mobile. The main page consists of the navigation panel and live view of the parking lot with plotted parking lot status Figure 5.3. This image updates every 2 seconds thanks to a small JavaScript code. The next option on the navigation panel is history. Opening this link loads all data from the database and creates an interactive bar graph that shows the number of cars in the parking lot depending on the hour of the day, Figure 5.4.

The last interface to note is the map upload. Here, a user can download a raw image in full resolution and use it to plot a new map of the parking lot using the included Jupyter mapy_creator widget, which is a part of the argonPark library. After doing so, they can upload the map to the server, which will automatically refresh it and show a new updated image of the parking lot with the new map seen in Fig 5.5.

The site also features a request parking button that shows the number of occupied parking spaces and the closest parking space relative to the entrance. This is valid only for the T10 parking place. If a user wants to keep this feature in their own parking lot, they need to correctly label parking spaces according to their position. The first one to label is the closest one, and so on.

[6] https://getbootstrap.com

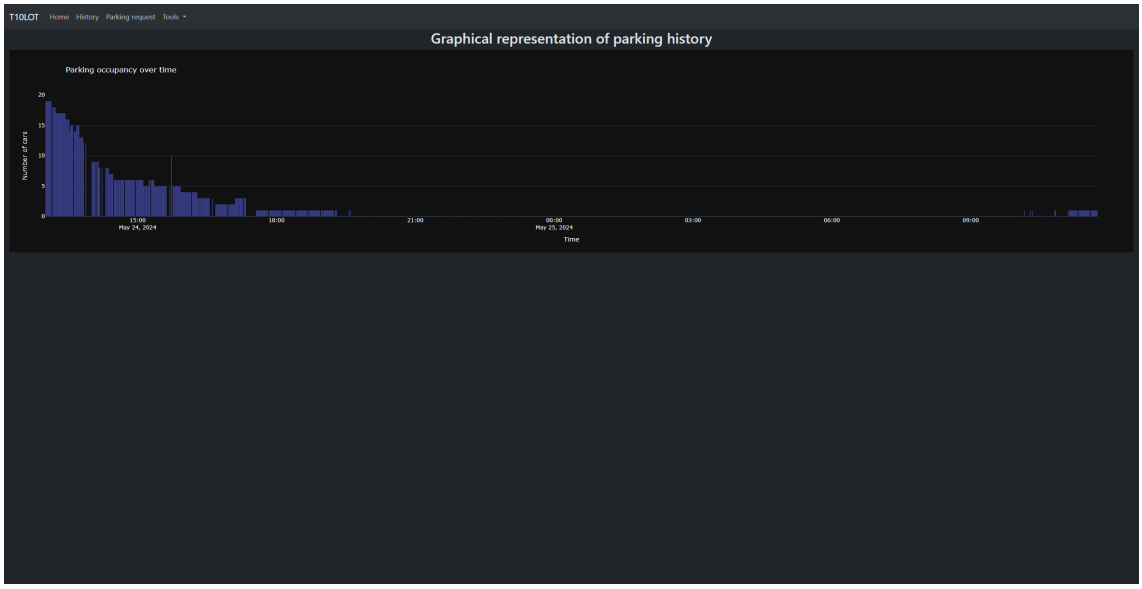


Fig. 5.4: History page view

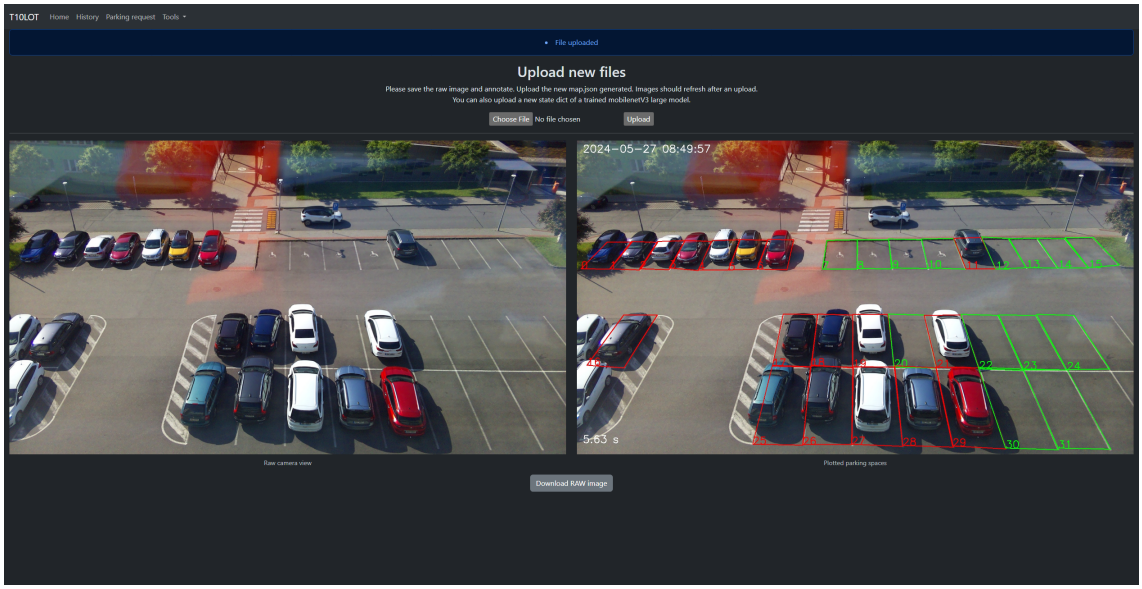


Fig. 5.5: Map edit page view

Conclusion

A certain list of models were picked after doing research about the available convolution networks with object recognition. These models were MobileNet V3 large, Mobilenet V3 small and ResNet50. All of these models were to be tested and evaluated for accuracy.

Having established this fact, a couple of interactive command-line scripts were written in a Python language. These scripts incorporate the use of multiple Python libraries for machine learning such as Torch, Torchvision, and CV2.

The next decision was to choose the datasets to which to use for training. After considering the use-case, a couple of datasets fitting the criteria were selected: PKLot and CarParkEXT. Both datasets feature photos taken from a building above a parking lot with a similar angle. The same is the final use case of this thesis.

Lastly, models need to be trained for the particular parking lot in mind. For this a collection of photos was taken on an iPhone X and a Raspberry Pi v4 with a camera module. These photos were then sorted, labeled, and processed in the same format as previous datasets. The resulting dataset is called T10LOT and is available at a Google Drive referenced in this works GitHub repository.

After training and testing the proposed models and evaluating their result on the data that were available, it seems that using a custom made MobileNet V3 small model on an faster R-CNN would be preferred, due to it's inference time and accuracy, even though the pretrained MobileNet V3 large is more accurate, providing more training data of the T10LOT dataset could improve the models accuracy. However, due to the simplicity of using a model with pretrained weights and quickly retraining it on a small dataset, the final decision to use the MobileNet V3 large with the pretrained weights was made.

After this decision a library that simplifies the inference and calculation of occupancy is made. The library initializes the MobileNet V3 large model and loads the provided trained model state dictionary. The library also requires a map.json file, which has the locations of individual parkng slots as polygons. This map can be created using an included Jupyter notebook scripts located inside of the library folder. Once all of the files are satisfied, the user can call methods of the library that evaluate, calculate, and create images that have the status of individual parking lots plotted.

This approach is good for a low-level program in which the user can use it for their own project, but for the demonstration of this solution a simple WebServer application that runs on the Flask Python library was written. This application runs on a Raspberry Pi v4 computer with a v2 camera module aimed at the T10 parking lot. The WebServer application then utilizes the ArgonPark library to analyze and

create data on parking lot occupancy. It communicates with a MongoDB database to store a large history. The users can then see the current parking occupancy situation in the parking place with a simple image. The site also features an option to upload a new map.json file and trained model state dictionary.

The Web server uses a multiprocessing approach due to the calculation complexity of the neural network. This means that the ArgonPark library is running as a child sub-process of the main WebServer process. Communication across these two processes is then managed through flags, events, and queues.

Further expansion of the application is possible; as the data is available in the database, any device with Internet access could request the actual state of the parking lot. Such as a car infotainment system to display the parking lot map and status. The data feature an IoU parameter that can be used to determine if a car is overlapping multiple parking places.

The selected hardware is relatively sufficient regarding it's performance. But the inference time is still longer than 4 seconds in which the system cannot judge the occupancy of the parking lot. The machine learning model can be further optimized by dynamic quantization. That would require modification of the training script to quantify the model during training. There could be a rather large performance increase.

During the writing of the web server, generative machine learning was used to guide with general code ideas. This tool is called copilot and is integrated in the IDE that was used during the writing of this thesis. No code has been copied, but snippets of the generated code were used as an inspiration for the current version of the web application.

Overall, the use of a camera for occupancy detection seems to be plausible compared to a sensor approach. It is not as accurate as a sensor would be, but, by using a single well-placed camera, one could, in theory, replace all sensors that would be covered by such a camera. With the ability to connect more cameras to enough powerful hardware that would run a larger machine learning model, one could effectively replace all sensors in a parking lot. The web application is continuously running and collecting more data which can later be used for further research.

Bibliography

1. KUŽELA, Miloslav; FRÝZA, Tomáš. Detection of parking space availability based on video. In: Brno, 2024.
2. FRÝZA, Tomáš; ONDŘEJ, Zelený; KUŽELA, Miloslav. Using Computer Vision and Machine Learning for Efficient Parking Management: A Case Study. In: 2024.
3. *Dive into Deep Learning — Dive into Deep Learning 1.0.3 documentation* [online]. [N.d.]. [visited on 2024-05-26]. Available from: `https://d2l.ai/index.html`.
4. EUROSTAT. *Stock of vehicles by category and NUTS 2 regions* [online]. [visited on 2023-11-06]. Available from: `https://ec.europa.eu/eurostat/databrowser/view/TRAN_R_VEHST__custom_6385961/bookmark/table?lang=en&bookmarkId=8dee8b9a-0c4f-4f1d-b24c-999f39c62a35`.
5. KIANPISHEH; MUSTAFFA, Norlia; LIMTRAIRUT, Pakapan; KEIKHOSROKIANI, Pantea. Smart Parking System (SPS) Architecture Using Ultrasonic Detector. *International Journal of Software Engineering and Its Application*. 2012, vol. 6.
6. PAIDI, Vijay; FLEYEH, Hasan; HÅKANSSON, Johan; NYBERG, Roger G. Smart parking sensors, technologies and applications for open parking lots: a review. *IET Intelligent Transport Systems* [online]. 2018, vol. 12, no. 8, pp. 735–741 [visited on 2023-12-26]. ISSN 1751-9578. Available from DOI: `10.1049/iet-its.2017.0406`. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1049/iet-its.2017.0406.
7. *Acconeer Products* [online]. [N.d.]. [visited on 2023-12-26]. Available from: `https://www.acconeer.com/products/`.
8. SANDBLOM, Daniel. *This is how you will park your car in the future* [online]. 2023. [visited on 2023-12-26]. Available from: `https://www.acconeer.com/news/this-is-how-you-will-park-your-car-in-the-future/`.
9. LAMAR, Jack. *Ending the era of inductive loops* [online]. [N.d.]. [visited on 2024-04-26]. Available from: `https://ouster.com/insights/blog/ending-the-era-of-inductive-loops`.
10. MARTYNOVA, Anastasia; KUZNETSOV, Mikhail; PORVATOV, Vadim; TISHIN, Vladislav; KUZNETSOV, Andrey; SEMENOVA, Natalia; KUZNETSOVA, Ksenia. *Revising deep learning methods in parking lot occupancy detection*. 2023. Available from arXiv: `2306.04288 [cs.LG]`.

11. MAREK, Martin. *Image-Based Parking Space Occupancy Classification: Dataset and Baseline*. 2021. Available from DOI: 10.48550/arXiv.2107.12207.

12. PRESS, Cambridge University. *Cambridge Advanced Learner's Dictionary & Thesaurus*. 2023. Available also from: https://dictionary.cambridge.org/dictionary/english/algorithm.

13. JIRSÍK, Václav. *Umělé neuronové sítě terminologie* [Kurz předmětu BPC-UIN]. 2023.

14. JIRSÍK, Václav. *Umělé neuronové sítě Perceptron* [Kurz předmětu BPC-UIN]. 2023.

15. JIRSÍK, Václav. *Strojové vidění – Snímání* [Kurz předmětu BPC-UIN]. 2022.

16. JIRSÍK, Václav. *Strojové vidění – Předzpracování* [Kurz předmětu BPC-UIN]. 2022.

17. JIRSÍK, Václav. *Strojové vidění – Segmentace* [Kurz předmětu BPC-UIN]. 2022.

18. JIRSÍK, Václav. *Strojové vidění – Popis* [Kurz předmětu BPC-UIN]. 2022.

19. HORÁK, Karel. *Introduction to Convolutional Neural Networks*. 2022.

20. HABIBI AGHDAM, Hamed; JAHANI HERAVI, Elnaz. Traffic Sign Detection and Recognition. In: HABIBI AGHDAM, Hamed; JAHANI HERAVI, Elnaz (eds.). *Guide to Convolutional Neural Networks: A Practical Application to Traffic-Sign Detection and Classification* [online]. Cham: Springer International Publishing, 2017, pp. 1–14 [visited on 2023-12-19]. ISBN 978-3-319-57550-6. Available from DOI: 10.1007/978-3-319-57550-6_1.

21. ASLAM, Aasma; HAYAT, Khizar; UMAR, Arif; ZOHURI, Bahman; ZARKESHHA, Payman; MODISSETTE, David; KHAN, Sahib; HUSSAIN, Babar. Wavelet-based convolutional neural networks for gender classification. *Journal of Electronic Imaging*. 2019, vol. 28, p. 1. Available from DOI: 10.1117/1.JEI.28.1.013012.

22. HOWARD, Andrew; SANDLER, Mark; CHU, Grace; CHEN, Liang-Chieh; CHEN, Bo; TAN, Mingxing; WANG, Weijun; ZHU, Yukun; PANG, Ruoming; VASUDEVAN, Vijay; LE, Quoc V.; ADAM, Hartwig. *Searching for MobileNetV3* [online]. arXiv, 2019 [visited on 2023-12-23]. Available from DOI: 10.48550/arXiv.1905.02244. arXiv:1905.02244 [cs].

23. SANDLER, Mark; HOWARD, Andrew; ZHU, Menglong; ZHMOGINOV, Andrey; CHEN, Liang-Chieh. *MobileNetV2: Inverted Residuals and Linear Bottlenecks* [online]. arXiv, 2019 [visited on 2023-12-23]. Available from DOI: 10.48550/arXiv.1801.04381. arXiv:1801.04381 [cs].

24. HE, Kaiming; ZHANG, Xiangyu; REN, Shaoqing; SUN, Jian. *Deep Residual Learning for Image Recognition* [online]. arXiv, 2015 [visited on 2023-12-25]. Available from DOI: 10.48550/arXiv.1512.03385. arXiv:1512.03385 [cs].

25. REN, Shaoqing; HE, Kaiming; GIRSHICK, Ross; SUN, Jian. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks* [online]. arXiv, 2016 [visited on 2023-12-23]. Available from DOI: 10.48550/arXiv.1506.01497. arXiv:1506.01497 [cs].

26. LIN, Tsung-Yi; GOYAL, Priya; GIRSHICK, Ross; HE, Kaiming; DOLLÁR, Piotr. *Focal Loss for Dense Object Detection* [online]. arXiv, 2018 [visited on 2023-12-25]. Available from DOI: 10.48550/arXiv.1708.02002. arXiv:1708.02002 [cs] version: 2.

27. GIRSHICK, Ross; DONAHUE, Jeff; DARRELL, Trevor; MALIK, Jitendra. *Rich feature hierarchies for accurate object detection and semantic segmentation* [online]. arXiv, 2014 [visited on 2023-12-19]. No. arXiv:1311.2524. Available from arXiv: 1311.2524[cs].

28. RODRIGUEZ, Maria L. *Different Models for Object Detection* [online]. 2021. [visited on 2023-12-23]. Available from: https://medium.com/geekculture/different-models-for-object-detection-9c5cda7863c1.

29. *How RetinaNet works?* [online]. [N.d.]. [visited on 2023-12-25]. Available from: https://developers.arcgis.com/python/guide/how-retinanet-works/.

30. REDMON, Joseph; DIVVALA, Santosh; GIRSHICK, Ross; FARHADI, Ali. *You Only Look Once: Unified, Real-Time Object Detection* [online]. arXiv, 2016 [visited on 2023-12-26]. Available from: http://arxiv.org/abs/1506.02640. arXiv:1506.02640 [cs].

31. TERVEN, Juan; CORDOVA-ESPARZA, Diana. *A Comprehensive Review of YOLO: From YOLOv1 and Beyond* [online]. arXiv, 2023 [visited on 2023-12-27]. Available from: http://arxiv.org/abs/2304.00501. arXiv:2304.00501 [cs] version: 1.

32. *What is Edge Computing? - Edge Computing Explained - AWS* [online]. [N.d.]. [visited on 2023-12-28]. Available from: https://aws.amazon.com/what-is/edge-computing/.

33. FOUNDATION, Python software. *Python* [online]. [visited on 2023-12-10]. Available from: https://www.python.org.

34. *Torchvision — Torchvision 0.16 documentation* [online]. [visited on 2023-12-10]. Available from: https://pytorch.org/vision/stable/index.html.

35. *Docs Home - Comet Docs* [online]. [N.d.]. [visited on 2023-12-26]. Available from: https://www.comet.com/docs/v2/.

36. ALMEIDA, Paulo R. L. de; OLIVEIRA, Luiz S.; BRITTO, Alceu S.; SILVA, Eunelson J.; KOERICH, Alessandro L. PKLot – A robust dataset for parking lot classification. *Expert Systems with Applications* [online]. 2015, vol. 42, no. 11, pp. 4937–4949 [visited on 2023-12-27]. ISSN 0957-4174. Available from DOI: 10.1016/j.eswa.2015.02.009.

37. G., Amato; F., Carrara; F., Falchi; C., Gennaro; C., Vairo. *CNRPark*. 2016.

38. LIN, Tsung-Yi; MAIRE, Michael; BELONGIE, Serge; BOURDEV, Lubomir; GIRSHICK, Ross; HAYS, James; PERONA, Pietro; RAMANAN, Deva; ZITNICK, C. Lawrence; DOLLÁR, Piotr. *Microsoft COCO: Common Objects in Context* [online]. arXiv, 2015 [visited on 2023-12-27]. Available from DOI: 10.48550/arXiv.1405.0312. arXiv:1405.0312 [cs].

39. ALLWRIGHT, Stephen. *How to interpret F1 score (simply explained)*. 2022. Available also from: https://stephenallwright.com/interpret-f1-score/.

Symbols and abbreviations

AI	Artificial intelligence
ARM	Advance RISCM Machine
BUT	Brno University of Technology
CCTV	Closed-circuit television
CNN	Convolution neural network
CPU	Central processing unit
DSP	Digital Signal Processing
FIFO	First-in, first-out
FNN	Feedforward neural network
FPN	Feature pyramid network
GPU	Graphical processing unit
HID	Human Interface Device
IoT	Internet of Things
IoU	Intersection over Union
t	Number of iterations
s	Sample
p	Wanted result
R-CNN	Regional Based Convolutional Neural Networks
ML	Machine learning
MMW	Milimeter wave
PCR	Pulsed Coherent Radar
Resnet	Residual neural network
ROI	Region of interest
SSH	Secure Shell

VPN	Virtual Private Network
VRAM	Video Random Acess Memory
YOLO	You only look once
VPN	Virtual Private Network

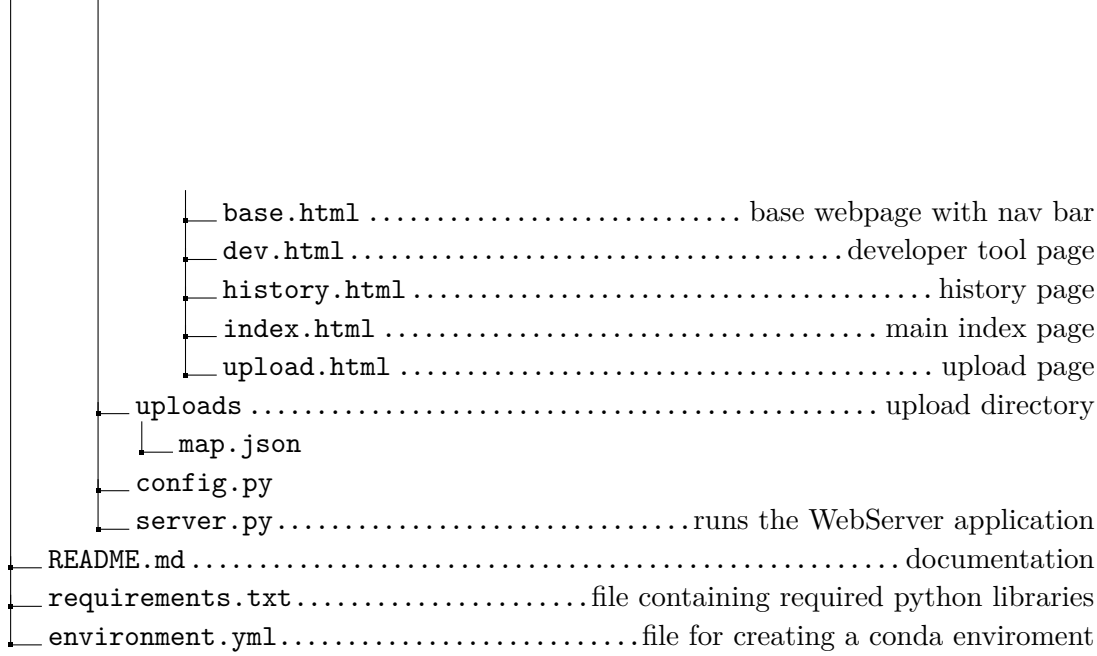
List of appendices

A Content of the electronic attachment

The electronic attachment contains all of the current code from the GitHub repository related to this work. The code was tested on Python3.11 running on Windows and Linux machines. The directory tree ommits python library files such as "___init___.py" which are needed for the language interpreter, but can be ignored by the user. Due to file size constrains, the electronic attachment does not include the trained models, datasets and tested images. Please refer to the GitHub[1] repository for these files.

```
/ ..... root of the attached archive
├── Annotating
│   ├── markup_widget_a.ipynb ..... jupyter widget for annotating parking places
│   ├── markup_widget_b.ipynb ..... jupyter widget for labeling occupancy
│   ├── README.md ..... documentation
│   └── scripts ..... directory containing the libraries for annotating widgets
│       ├── annotation_widget.py
│       ├── labeling_widget.py
│       └── process_data.py
├── Models
│   ├── train.py ..... script for training models
│   ├── test.py ..... script for testing models
│   ├── inference.ipynb ..... jupyter notebook for testing on a single image
│   └── baselines ..... directory containing libraries for training and testing scripts
│       ├── intersection_based
│       │   └── inter_models.py
│       └── utils
│           ├── common_utils.py
│           ├── inter_utils.py
│           ├── process_data.py
│           └── queries.py
└── rPi_host ..... Raspberri Pi programs
    ├── camera.py ... takes photos with raspberry pi camera used for dataset creation
    └── webServer ..... WebServer application
        └── app ..... WebServer library
            ├── ArgonPark ..... ArgonPark library
            │   ├── argonPark.py
            │   ├── argonPark_example.py ..... shows exaple usage of the library
            │   └── map_Creator
            │       ├── Creator.ipynb ..... for creating map.json files
            │       └── scripts
            │           └── annotation_widget.py
            ├── Static ..... WebServer static files
            └── templates ..... HTML jinja2 templates
```

[1] https://github.com/slavajda02/parking-research-argon


base.html ..... base webpage with nav bar
dev.html ..... developer tool page
history.html ..... history page
index.html ..... main index page
upload.html ..... upload page
uploads ..... upload directory
map.json
config.py
server.py ..... runs the WebServer application
README.md ..... documentation
requirements.txt ..... file containing required python libraries
environment.yml ..... file for creating a conda enviroment