BRNO UNIVERSITY OF TECHNOLOGY

Faculty of Electrical Engineering
and Communication

MASTER'S THESIS

Brno, 2022

Bc. Ondřej Zelený

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

TRAFFIC ANALYSIS USING ON MACHINE LEARNING

ANALÝZA DOPRAVNÍHO PROVOZU S VYUŽITÍM STROJOVÉHO UČENÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR AUTOR PRÁCE	Bc. Ondřej Zelený
SUPERVISOR VEDOUCÍ PRÁCE	doc. Ing. Tomáš Frýza, Ph.D.

BRNO 2022

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní program **Elektronika a komunikační technologie**

Ústav radioelektroniky

Student: Bc. Ondřej Zelený ***ID:*** 195654

Ročník: 2 ***Akademický rok:*** 2021/22

NÁZEV TÉMATU:

Analýza dopravního provozu s využitím strojového učení

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je sestavit systém, který bude schopen analyzovat dopravní provoz z obrazových dat a provádět klasifikaci vozidel na základě hlubokého učení. Takový systém je vyžadován pro efektivní systémy řízení provozu v reálném čase, které mohou včas detekovat změny v charakteristikách provozu. Změřená data budou odesílána a zpracovávána na některém z dostupných serverů pro IoT, např. ThingSpeak, Ubidots, Cayenne, aj. Dílčím cílem je proniknout do problematiky strojového učení, dostupných nástrojů a datasetů a ověření možností takového přístupu pro analýzu obrazových dat v reálné dopravě. Prostudujte možnosti detekce a klasifikace vozidel ve video sekvencích, především na dálnicích, silnicích s větším počtem jízdních pruhů a na křižovatkách. Seznamte se s problematikou strojového učení, dostupných datasetů z dopravy a vhodných detekčních metod. Navrhněte a sestavte systém s jednou fixní kamerou, vhodným hardwarem a vyzkoušejte jeho funkčnost pro klasifikaci vozidel na jednoduchém scénáři se slabým provozem.

Zprovozněte sběr dat v reálném systému a celý systém ověřte dlouhodobým testováním. Provádějte automatické shromažďování důležitých statistik, mezi které patří počítání vozidel, klasifikace typů vozidel, ale pokuste se zaměřit také na odhad rychlosti vozidla z videa, sledování využití jízdního pruhu apod. Zveřejněte veškeré informace o systému i měřená data.

DOPORUČENÁ LITERATURA:

[1] Analyzing Traffic Using a Webcam, a Raspberry Pi and ThingSpeak [online]. MathWorks, 2018 [cit. 2021-5-24]. Available on: https://www.mathworks.com/matlabcentral/fileexchange/52456-analyzing-traffic-using-a-webcam-a-raspberry-pi-and-thingspeak

Termín zadání: 11.2.2022 ***Termín odevzdání:*** 11.8.2022

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

prof. Dr. Ing. Zbyněk Raida
předseda rady studijního programu

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRACT

The aim of this thesis is to investigate the problematic of object detection and classification for traffic analysis. The theoretical part of the paper takes insight on numerous methods and techniques of object detection and classification. Further the paper discuses popular frameworks and programming languages for implementation of convolutional neural networks as well as multi-object tracking and communication with IoT server. The practical part shows implementation of chosen model and additional functionalities, object trackers and communication with selected IoT platform as well as data processing in cloud and visualization.

KEYWORDS

Deep Learning, Supervised Learning, Computer Vision, Object Detection, Object Classification, Python, PyTorch, Convolutional Neural Networks, Object Tracking, Internet of Things

Typeset by the thesis package, version 4.07; http://latex.feec.vutbr.cz

Rozšířený abstrakt

Úvod

Každoročně zvyšující se nároky na systémy řízení dopravy mají za cíl nejen snížit dobu vozidel na silnicích, ale také zlepšení bezpečnosti a snížení emisí vytvářené stojícími auty v dopravních zácpách. Takovýto systém musí být schopen detekovat změny v dopravním provozu a analyzovat je v reálném čase. Novodobé systémy pro analýzu dopravy často využívají metody spadající po algoritmy umělé inteligence a to nejčastěji hluboké učení, které se v posledních letech stalo velice populární a to zejména díky neuronovým sítím. Umělý neuron je základní stavebním blokem neuronových sítí, který reprezentuje matematický model s několika vstupy a jedním výstupem. Výpočet výstupu neuronu probíhá tak, že je každý z vstupů vynásoben váhou, která reprezentuje důleitost konkrétního z vstupů. Poté jsou všechny vstupy vynásobené váhami sečteny a je k nim přičtena prahovací hodnota. Výsledná hodnota je poté transformována aktivační funkcí a její výsledek je výstupem neuronu. Neuronové sítě jsou díky tomu principu schopné na základě odchylky od očekávaného výsledku upravit právě své váhy a prahovací honoty a během procesu učení tak zvyšovat svou přesnost.

Metody hlubokého učení se z pohledu učení dají rozdělit do dvou hlavních kategorii a to učení s učitelem a učení bez učitele. Učení s učitelem využívá vstupní data a k nim odpovídající výstupy. Učení bez učitele naopak využívá vstupní data bez jakýchkoliv dalších informací a metody tohoto druhu hledají souvislosti mezi daty a snaží se je kategorizovat.

Konvoluční neuronové sítě jsou jedním z nejpoužívanějším druhů neuronových sítí a to především pro zpracování dat, která mají tenzorovou topologii. Typicky se konvoluční neuronové sítě skládají ze tří vrstev: konvoluční vstva, tzv. poolingová vrstva a aktivační vrstva. Konvoluční vrstva využívá filtru nazývaného kernel, který se posouvá po vstupních datech a vytahuje jednu hodnotu z dané oblasti. To má za následek že každý sektor dat, na které je aplikován kernel je přetransformován do jediné hodnoty a díky tomu se tedy snižuje rozměr dat. Aktivační vstva má za úkol přepočítat data z poolingové vrstvy do vhodnějšího měřítka. Mezi běžné poolingové vrstvy patří například Max pooling nebo Average pooling. Typickým zástupcem aktivačních funkci je ReLU funkce, které je nulová pro záporné hodnoty a pro kladné hodnoty je výstup roven vstupu.

Řešení

Tato práce využívá pro detekci dopravních prostředků plně konvoluční neuronové sítě s architekturou YOLOv5, která je nejen schopná detekovat objekty ve snímcích, ale také k nim přiřadit rámeček ohraničení. Tato architektura využívá metod jako je tzv. Cross Stage Partial a Spatial Pyramid Pooling, které nejen zlepšují extrakci detailů z data, ale také zlepšují korekci parametrů neuronových sítí při trénování. Implementace je provedena v jazyce Python s využitím knihovny PyTorch, která obsahuje veškeré stavební bloky k implementaci algoritmů strojového učení. Implementovaný algoritmus je trénovaný na COCO datasetu, což je obrovský dataset více než 150 tisíci obrázky, z nich jsou vytaženy pouze ty obsahující 5 tříd vozidel a to konkrétně: osobní vozidlo, kolo, motocykl, autobus a nákladní vozidlo. Celý model je trénován s využitím cloud platformy Google Colab, která umožňuje využití vysoce výkonných grafických karet a uložiště typu RAM o velké kapacitě.

Jelikož je dílčím cílem práce také provádět analýzu dopravy, byly k implementaci přidány dva algoritmy na trasování objektu, které umožnují nejen přesné počítání vozidel konkrétních tříd, ale také výpočet přibližné rychlosti vozidel. První trasovací algoritmu funguje na principu přiřazování ID čísla detekovaného objetu, které je nejblíže k poslední známé poloze trasovaného objektu. Druhý algoritmus je založen na ploše objektu a přiřazuje ID objektu k detekci jejíž ohraničující rámeček se překrývá s rámečkem původního trasovaného objektu o více než je určitá limitní hodnota.

K reálným testům modelu je vybrána platforma Jetson Nano s 2 GB RAM pamětí od NVIDIA a pro zachycení dopravy je zvolena kamera IMX219-120 s 8Mpx senzorem a CSI rozhraním, které umožňuje rychlejší přístup k snímkům než klasické USB kamery.

Experimenty

Práce předkládá několik experimentů zabývající se vyhodnocení celého systému. Model je po natrénování schopen detekovat vozidla ve snímcích s přesností přibližně 78 %. Mimo to je model také schopen předpovědět 67 % ohraničujících rámečků, jejíchž IoU se skutečným ohraničujícím rámečkem je vetší než 50 % (mAP:0.5). Testy na reálném scénáři ukazují závislost modelu nejen na úhlu pohledu, ale také na reflektivitě vozidla a jeho barvě. Především pak testy s pohledem shora ukazují problematičnost modelu s detekcí vozidel s lesklou černou barvou.

Rychlost zpracování snímků na reálném systému trvá relativně dlouho a jeden snímek je zpracován přibližně za 110 ms, což odpovídá asi 9 snímkům za sekundu. Tato hodnota je bohužel příliš nízká a z toho důvodu byl model konvertován do formátu TensorRT který je více optimalizovaný s NVIDIA grafickými procesory. Tato konverze měla za následek snížení rychlosti zpracování na přibližně 71 ms, které již odpovídají přibližně 14 snímkům za sekundu.

IoU trasovací algoritmus je ve výchozím nastavení používán k trasování objektů a jejich počítání a odhadu rychlosti. V případě reálného systému byl algoritmus schopen zachytit a korektně trasovat některé pomalé vozidla, avšak díky nízké snímkovací rychlosti nebyl systém sledovat všechna vozidla, především ta s rychlostí nad 50 km/h. Testy na stolním počítací se záběry o 30 snímcích pak ukazovaly až na výjimky velice dobré sledovací vlastnosti a byly schopny zachycovat i rychlost některých vozidel. Aby však byl systém schopen zaručeně odhadovat rychlost všech vozidel, bylo by nutné pracovat se snímkovou frekvencí alespoň 60 snímků za sekundu.

Chování systému v závislosti na poloze kamery bylo testováno pro dvě situace. V první situaci byla kamera umístěna na straně cesty, což z pohledu predikcí lehce zvedlo přesnost. Avšak tato pozice také vedla ke zhoršení výkonu trasovacího algoritmu, jelikož nyní kamera zabírala pouze malou část vozovky a vozidla vstupovala a vystupovala ze záběru příliš rychle. To mělo také za následek že nebyly zachyceny téměř žádné rychlosti vozidel. Druhá pozice kamery nad vozovkou byla naopak mnohem výhodnější pro systém i přesto že předpověděná důvěra v jednotlivé třídy byla lehce nižší. Trasovací algoritmus byl schopen nejen lépe trasovat, ale také i odhadovat rychlost vozidel.

Závěr

Navržený algoritmus detekuje a klasifikuje vozidla s relativně vysokou přesností a také je schopen vozidla trasovat, počítat a odhadovat jejich rychlost. Testy na hardwaru s vyšší výpočetní kapacitou ukazují téměř bezchybné chování na snímkové frekvenci 60 snímků za sekundu, kterých však zvolený hardwarový systém není schopen dosáhnout ani po konverzi do vysoce optimalizovaného back-endu (TensorRT). Volba platformy se tedy prokázala být nedostačující avšak vzhledem k dostupnosti vývojových platforem pro umělou inteligenci v době zadání práce nebyl velký výběr. Z hlediska samotného Jetson Nano by bylo výhodnější zvolit verzi s 4 GB RAM a vyhnout se tak občasným problémům s docházející pamětí. To by také mohlo vézt ke zlepšení rychlosti modelu vzhled k většímu využití RAM namísto SWAP paměti, která je uložena společně se systémem na mikro SD kartě a je využívána v případě, že v RAM paměti není dostatek místa. Vzhledem k náročnosti na frekvenci snímků by bylo vhodné zvolit výkonnější platformu, která by s dostatečnou rezervou zvládla zpracovávat záznam například s 60 snímky za sekundu.

Author's Declaration

Author:	Bc. Ondřej Zelený
Author's ID:	195654
Paper type:	Master's Thesis
Academic year:	2021/22
Topic:	Traffic analysis using on machine learning

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 professional guidance, consultation, patience and suggestions for the work.

Contents

List of Figures

List of Tables

Introduction

For the past few decades the traffic in large cities and highways has become source of many problems such as efficiency and safety. Mainly for those reason, cities started deploying means of traffic control to allow smooth and safe flow of traffic. Traffic lights are the main instrument of traffic control. Traffic lights however have the disadvantage of having "fixed" time when the green light is on which may cause traffic jam in case there are too many vehicles waiting for green light. This may have cascade effect if most of those cars intend to go in the same direction.

For those reason, there is need of traffic control system that could count the occupancy of the road and analyze the traffic to improve the means of traffic control. For those reasons, system that could do such analysis is needed and it has to not only be able to count the number of vehicles but also classify them in order to better understand the traffic. This is an important information because there is a difference in the occupancy when there are couple regular cars, or when there are couple of tow trucks. Such a systems needs to learn the difference between types of cars and correctly detect and classify then in image or video feed.

There are many ways to implement such system, but the most commonly used way is to use deep learning object detector and classier. Deep learning algorithms learn features of objects by processing thousands of images and learning from it. After the algorithm is done learning and can detect and classify vehicles with precision sufficient for its task, it can be deployed in field on a suitable hardware. The hardware needed for this system consists out of two main components: camera and computer.

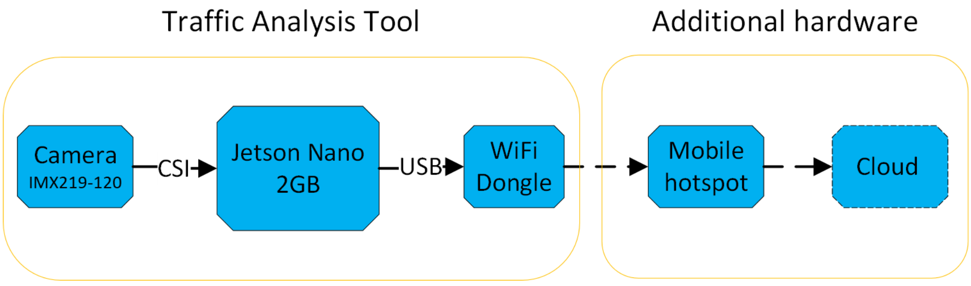


Fig. 1: Proposed system for traffic analysis

Based on research made in theoretical part of this thesis, a custom model based on YOLOv5 architecture will be implemented and evaluated for its accuracy and ability to track objects as well as comunicate with cloud. Real system consisting of Jetson Nano 2 Gb and IMX219-120 8Mpx camera will be build and tested on bridges or overpasses, since the real system would be normally installed on a pole overlooking the traffic. Since these location do not usually have outlets, the system will be powered using portable powersource. Jetson Nano can be connected to the internet via regular Ethernet cable, however, since that will not be available either, WiFi dongle supplied with the Jetson Nano will be used to connect to the internet via mobile hotspot, through which the Jetson will be able to upload data to the cloud.

1 Theory

The theoretical part of this thesis will introduce a reader to the problematic of deep learning and computer vision. The chapter is going to introduce the common concepts in deep learning as well as the tools and language used for implementation of deep learning algorithms.

1.1 Deep learning introduction

Artificial Intelligence (AI) is a field of computer science which focuses on modeling intelligent machines in order to automatize tasks that would otherwise be performed by human. The first AIs were working of purely coded bases where the developer hard coded rules based on which the decision was made. This is today known as symbolic AI. With the increasing need for automatization of more complex tasks, implementation of symbolic AI was no longer possible due to large amount of rules that would have to be hard coded [1].

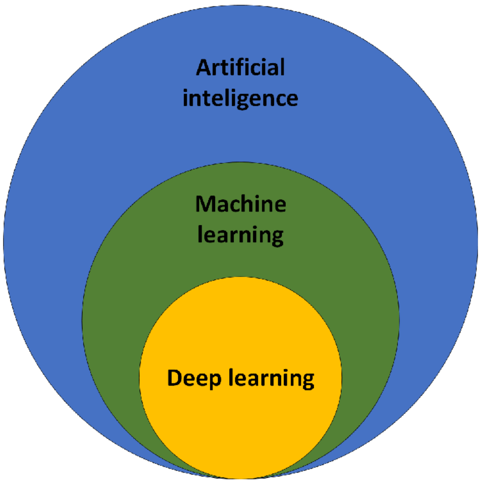


Fig. 1.1: Sub-fields of artificial intelligence.

This problem lead to new approach called *Machine learning* (ML), which instead of producing output based on set of hard coded rules, learns the rules from a set of examples and later uses those rules to process new data. ML has become very popular in recent years and used in many applications like online advertisement, face recognition and autonomous driving. These application often use *Deep learning* (DL) algorithms which use large amount of *Artificial Neuron* (AN) structured in layers to replicate human thinking [1].

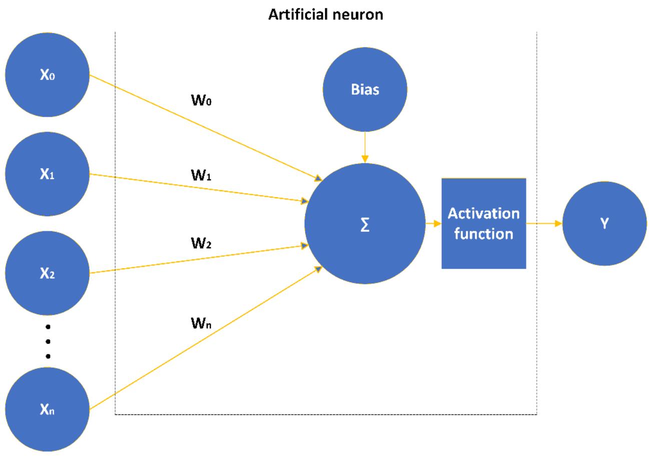


Fig. 1.2: Artificial neuron.

Artificial neuron is the most basic building block of neural network and its structure can be seen in Figure 1.2. Each input has its own weight, which is adjusted according to how much the specific input is relevant. AN does summation across the inputs multiplied by their respective weights. Bias can be added to the summation to adjust the level whenever the neuron should be activated or not. The activation function then translates the output based on the selected type of activation function and the value of the output. The equation 1.1 mathematically describes the AN shown in Figure 1.2 [2].

$$y = AF(b + \Sigma_{i=0}^{n} x_i * w_i) \tag{1.1}$$

Deep learning incorporates *Neural Network* (NN) in successive layers in order to learn from data in a hierarchical manner. These models often involve tens or even hundreds of layers and the term depth is used to annotate, how many layers, excluding input and output layer, contribute to the output.

In general, deep learning models can be divided into two main learning types based on the interaction of the user (teacher) with the data. Supervised learning is the first type and it requires annotated (labeled) data for learning process which often has to be handled by human. The second type of learning is unsupervised learning, which does not require any type of human interaction with the date. The algorithm on its own tries to figure out what the output should be.

1.1.1 Supervised learning

As the name suggests, supervised learning requires human supervision over a set of data that has to be classified. These data are labeled by features that define the meaning of data. For example, these labels could be names of animals on the picture (dog, cat, bird . . .) or predicted values (1, 0.5, 5, . . .). Supervised learning problems can be grouped into two groups based on the algorithm's output variable. When the output variable is a category, we are talking about **classification** and if the output variable is a real value, we are talking about **regression** [2, 3].

The algorithms are trained from these examples and evaluated with test data. This is usually done in multiple epochs, where one epoch represents one pass over the whole training dataset. Occasionally, an issue called overfitting can occur. Overfitting means that the algorithm is precisely tuned to find patterns in training data but may not work in the real application for previously unseen data. For this reason, it is important that the test data are unforeseen by the algorithm. Supervised training models have broad application from weather predictions and market prediction to speech and image classification. Some popular examples of supervised machine learning algorithms are:

- *Convolutional Neural Network* (CNN)
- *Recurrent Neural Network* (RNN)
  - *Long Short-Term Memory* (LSTM)
  - *Gated Recurrent Unit* (GRU)

In the following text, these networks are briefly introduced.

Convolutional Neural Networks

Convolutional Neural Network or shorter ConvNet, is a specific type of neural network that uses a convolution layer for processing data that has grid-like topology. This type of network is highly used for processing images which can be interpreted as a 2D tensor (black and white images) or 3D tensor (RGB images). Traditional CNN consist of three types of layers: Convolution layer, pooling layer, and fully-connected (also known as dense) layer. The structure of such network can be seen on Figure 1.3.

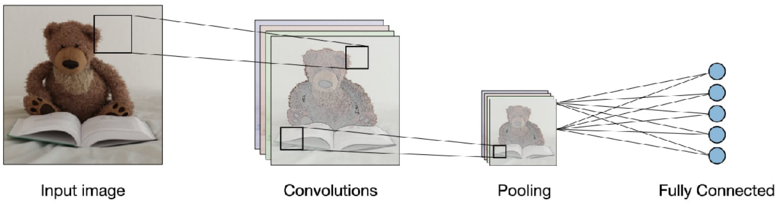


Fig. 1.3: Traditional CNN architecture [4].

The most common building block in CNN is convolutional layer in which a filter called kernel, slides over the input data and performs element-wise multiplication with the data. The input data on convolutional layer are typically tensors and the number channels of kernel match the number of channel of the input data. Padding is a term for added zeros around input data in order to improve the feature extraction on the edges of the data and also to allow kernel better fit the data. Another important parameter is stride, which defines spatial distance between location where the kernel is applies and its default value is usually one. In order to calculate padding, three parameters need to be taken into account: Size of the input W, size of the kernel F and stride S. Padding can be calculated according to the following equation:

$$P = \frac{(S-1)*W-S+F}{2} \tag{1.2}$$

An activation function is used after the convolution is done to help the network learn complex features. The activation function is applied to each individual value in the output tensor. Figure 1.4 shows commonly used activation functions. Sigmoid, ReLU, Leaky ReLU, ELU and tanh are functions of real variable while the Softmax function outputs vector of probabilities [5].

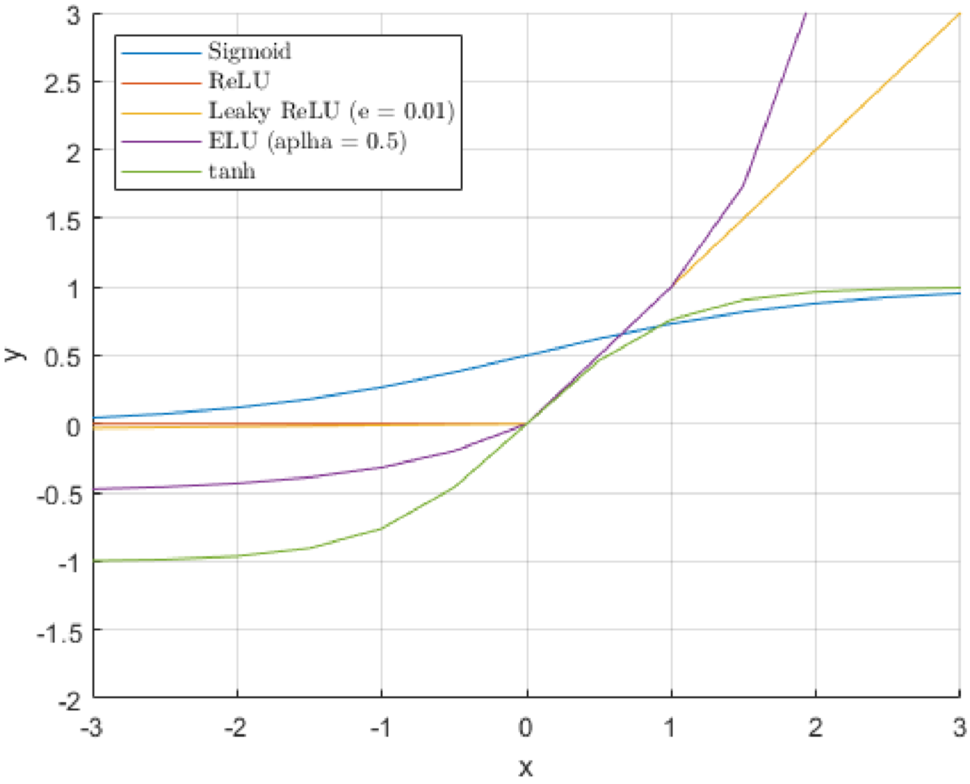


Fig. 1.4: Common activation functions.

Tab. 1.1: Activation functions [4].

Activation function	Equation
Sigmoind	$f(x) = \frac{1}{1+e^{-x}}$
ReLU	$f(x) = max(0, x)$
Leaky ReLU	$f(x) = max(\epsilon x, x)\ where\ \epsilon << 1$
ELU	$f(x) = max(\alpha(e^{x} - 1), x)\ where\ \alpha << 1$
tanh	$f(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
Softmax	$\vec{p}\ where\ p_i = \frac{e^{x_i}}{\sum_{k=1}^{N} k^2}$

Figure 1.5 shows the process of generating an activation map by performing convolution of kernel over the image where the padding is set to zero, stride to one and kernel size to two.

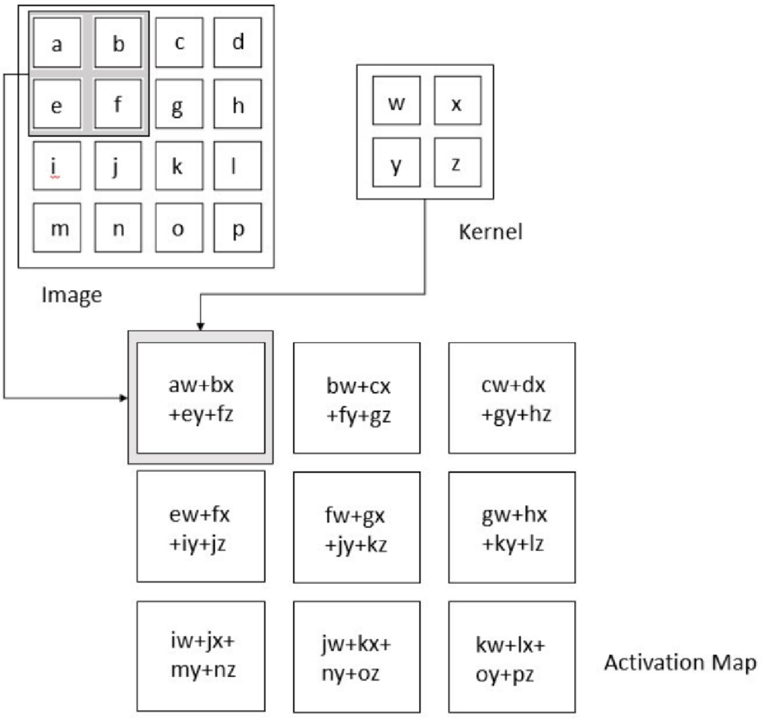


Fig. 1.5: Example of convolution [6].

The pooling layer downsamples the input by processing the spatial invariance of the input. Typically, they use max or average pooling. The down-sampling is done by a sliding window that selects either maximum or average value of the current view. The most commonly used pooling is max pooling because it better preserves features [4].

Fully-Connected layer(s) (FC) layers are usually used at the end of CNN architecture and are used to extract objectives such as class scores. FC layers take in flattened input (vector) and connect each input to all neurons [4].

Recurrent Neural Networks

RNN are networks designed for sequential data or time-series data. RNNs are commonly used for natural language processing and speech recognition by companies like Google and Apple. They are different from other neural networks by having so called "memory". Layers in these networks are made in such a way that they can use previous outputs into account when computing output for new input data. The representation of the previous output is called the hidden state. RNNs are also distinguishable by the fact that the often share parameters across layers of the network [7, 8].

The Figure 1.6 demonstrates the concept of RNN. On the left side, we can see a traditional recurrent neural network with a loop for the hidden state and on the right side we unrolled the network, and we can see the chain-like nature of RNNs. Traditional RNN units usually have a very simple structure, such as a single tanh layer.

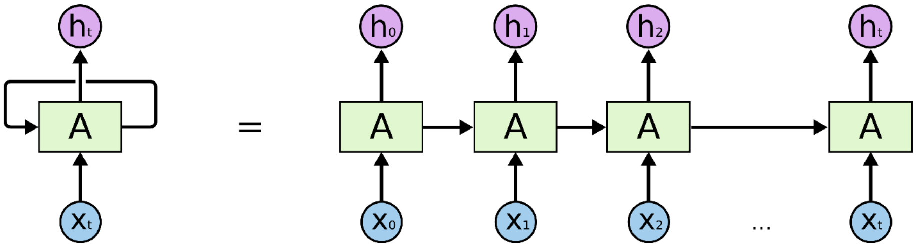


Fig. 1.6: An unrolled Recurrent Neural Network [8].

Traditional RNN however have one problem which is they are not able to learn long-term dependencies due to exploding or vanishing gradients in the learning process. The vanishing gradient means the gradient that as it decreases exponentially (with each layer) as the network learns through back-propagation which causes the early layers in the network to learn by a really small amount or not learn at all. The exploding gradient is the exact opposite, the gradient is too large, which causes the weights of the model to grow, and they will eventually reach a value that cannot be handled by the model (NaN value). For that reason, the solution is to either reduce the number of hidden layers or eliminate some of the complexity in RNN. Based on this thought, two types of RNN units were developed: *Long Short-Term Memory* (LSTM) and *Gated Recurrent Unit* (GRU) [8].

LSTMs are specially designed to capture long-term dependencies by extending the number of layers in a single unit to four, which interact in a certain way. These four layers are called gates, and specifically, they are called input gate, output gate, forget gate, and cell state [9]. Diagram of the LSTM can be seen at Figure 1.7.

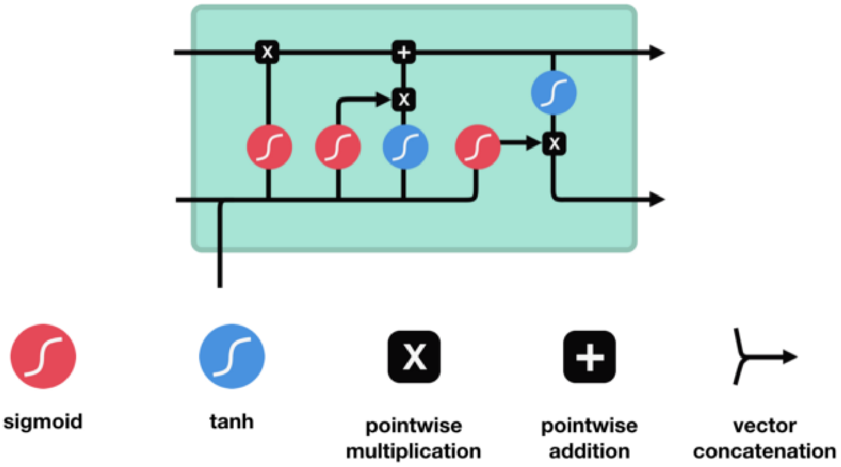


Fig. 1.7: Long Short-Term Memory unit [9].

Forget gate is a layer that decides what information should be kept or forgotten. Its input is a concatenation of the hidden state from the previous unit and input of the current unit. The result of this operation is passed through the sigmoid function which output is in a range from 0 to 1. The output of the sigmoid function plays role in what information from the previous cell state will be kept or forgotten. The lower the number the higher loss of information is produced and vise versa [9].

Input gate passes the hidden state through the sigmoid layer and also through the tanh layer. The sigmoid layer decides what information from the tanh layer is important by point-wise multiplication [9].

The cell state of the current unit is calculated from the cell state of the previous unit by pointwise multiplication with the output of forget gate which is forget vector. After that, we update the cell state by pointwise addition with the output of the input gate [9].

Lastly, the output gate generates the hidden state for the next unit by pointwise multiplication of cell state and output of sigmoid function on which input is again concatenation of current input and previous hidden state [9].

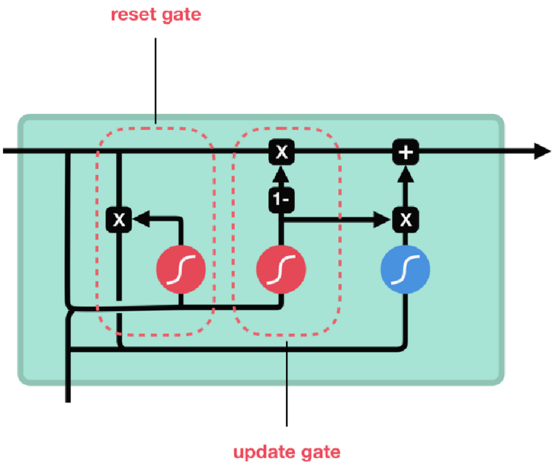


Fig. 1.8: Gated Recurrent Unit [9].

GRU is a newer generation of RNN and it is very similar to LSTM. It consists of two gates, the reset gate, and the update gate. The structure of the GRU can be seen in Figure 1.8 [10]. The GRU units in comparison to LSTM, got rid of the cell state and the input gate. Since there is no cell state, the hidden state for the next unit is generated based on the current input and previous hidden state. These two pieces of information are concatenated and passed through an update gate which consists of a sigmoid function. The output is represented by values between 0 and 1. However, later in the update gate, this value is subtracted from 1 which tells the network what and how much information from the previous hidden state should be passed to the next GRU unit. The reset gate is similar to the forget gate in LSTM, however, the reset gate determines what information from the hidden state should be kept or forgotten before its concatenation with the tanh function.

1.1.2 Unsupervised learning

Unsupervised learning is used when the problem has too many variables and outputs are not known. In that case, unsupervised learning algorithms segment data into groups of examples known as clusters or groups of features. The algorithm is at this point able to add labels to these groups making them labeled. Unsupervised learning is often used in tasks requiring a massive amount of data where labeling is not possible. Practically can be used and often is used, as the first step in supervised learning application for labeling the data for the supervised learning process. Unsupervised learning problems can be divided into **clustering** problems and **association** problems. Clustering problems are tasks where we look for groups hidden in the data. Association problems are tasks, where we want to discover a rule that describes large portions of the data. Some popular examples of unsupervised machine learning algorithms are [2, 3]:

- *Self Organizing Map* (SOM)
- *Autoencoders*

Self Organizing Maps

Self Organizing Map (SOM) is a special type of artificial neural network that does not learn by back-propagation, instead, it uses competitive learning to adjust weights in neurons. This type of artificial NN is used for dimension reduction to reduce data by creating a spatially organized representation, which is useful because it helps us discover the correlation between data. SOM have two layers, the input layer, and the output layer. SOM does not have an activation function in neurons, which means that the weights are directly passed to the output layer (feature map).

As said before, SOM are trained by competitive learning which is done in three steps, Competition, Cooperation, and Adaptation. Each neuron in SOM is assigned a weight vector with the same dimension as input data. We compute the distance between each neuron in the output layer and the input data. The competition step is won by the neuron with the smallest distance. The cooperation step is the second step in the learning process which says that not only the winning neuron will be updated but also its neighbors. Neighbors are chosen by kernel function dependent on time and distance from the winning neuron. In the Adaptation step, we update neighboring neurons depending on the distance from the winning neuron and time [11].

Autoencoders

Autoencoders takes input data and compresses it into a lower-dimensional code and then tries to reconstruct it from this representation. Autoencoders, whose block diagram is visualized at Figure 1.9, are a specific type of neural network where the input and the output are the same. Excluding the input, autoencoders consist of three components: *encode*, *code* and *decoder*. The forward function uses the input data and the selected encoding method to generate a low-dimensional representation of the input called *code*. The decoder afterward uses only this representation and decoding method to reconstruct the input data. The autoencoders have a couple of important properties, the first of which is that they are data-specific. This means that the input data has to be similar to what the neural network was trained on, otherwise it will not work properly. Another property is that the compression will never be exactly the same as the input [12].

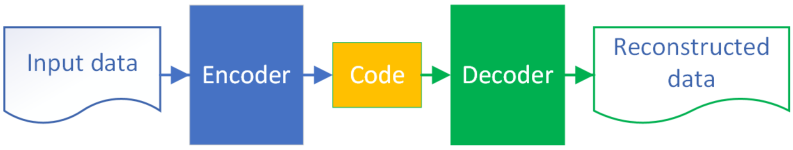


Fig. 1.9: Block diagram of autoencoder.

1.2 Training of neural networks

Training of neural networks is one of the most challenging tasks when developing deep learning algorithms. During the training process, the optimizing algorithm is used to determine optimal values for all the weights and the biases at which the network has a minimal loss. Loss is a number indicating how bad the model's predictions were on a single training sample. Variety of loss functions can be used to determine the loss of the model and its selection is depends on the task being solved. Regression problems use loss functions such as *Mean Absolute Error* (MAE),*Mean Squared Error* (MSE) or *Root Mean Squared Error* (RMSE), classification problems on the other hand, use loss functions such as *Binary Cross-Entropy* (BCE), Hinge loss or *Categorical Cross-Entropy* (CCE). The loss of the model can be minimized by the change of the values of parameters. Loss functions are usually non-linear functions with a large number of parameters and the minimization of the loss of the model is usually done using gradients and back-propagation. A common issue when training neural networks is overfitting, which signalizes that the model is closely trained to fit the training data,but does not generalize the data, which causes bad performance on previously unseen data. The following subsections summarize the main techniques for the training of neural networks [13, **?**].

1.2.1 Loss function for regression tasks

Mean absolute error

Mean absolute error, also known as L1 loss, is a loss function and metric for regression tasks whose result is an average of all absolute errors. Absolute values of the error ensure the individual losses do not cancel out each other and the total loss is always positive and greater or equal than zero. MSE is used in tasks where the training data may contain outliers are as relevant as the rest of the data. Following equation shows how the MAE is calculated [15]:

$$MAE = \frac{1}{M}\sum_{n=1}^{M}|y_n - \hat{y}_n| \tag{1.3}$$

, where $\hat{y}_n$ is the predicted value of the networks, y_n is the expected output and M is the number of training samples.

Mean squared error

Mean squared error, also known as quadratic or L2 loss, is a loss function used for the training of regression algorithms. The loss function represents the average squared distances between predicted and expected values. Squaring the errors helps to ensure the loss is always equal to or greater than zero as well as it better penalizes larger errors, which allows for a better fit when the training data contains important outliers. The following equation shows, how the MSE is calculated [16]:

$$MSE = \frac{1}{M}\sum_{n=1}^{M}(y_n - \hat{y}_n)^2 \tag{1.4}$$

, where $\hat{y}_n$ is the predicted value of the networks, y_n is the expected output and M is the number of training samples.

Root mean squared error

Root mean squared error is one of the most common loss functions and is nearly identical to the MSE, in fact, it is just a square root of MSE. The reason the RMSE is preferred over MSE is that the loss is the same unit as the response unit of the trained model. The following formula shows the calculation of RMSE [17]:

$$RMSE = \sqrt{\frac{1}{M}\sum_{n=1}^{M}(y_n - \hat{y}_n)^2} \tag{1.5}$$

, where $\hat{y}_n$ is the predicted value of the networks, y_n is the expected output and M is the number of training samples.

1.2.2 Loss function for classification tasks

Categorical cross-entropy

Categorical cross-entropy is a loss function used for a single label classification, which is a classification where each data point can belong to one class only. The categorical cross-entropy is based on a comparison of the predicted probability distribution with the target probability distribution. Networks using this type of loss function usually require the use of a soft-max activation function to ensure correct probability distributions for the predictions. The following equation shows how the categorical cross-entropy is calculated [18, 19]:

$$CCE = -\sum_{n=1}^{M} y_n \cdot log(\hat{y}_n) \tag{1.6}$$

, where $\hat{y}_n$ is the predicted probability for the n class, the y_n is the target probability for the n class and M is the number of class labels.

Binary cross-entropy

Binary cross-entropy, also known as log loss, is a loss function used for multi-label classification, which means the data point can be labeled with multiple classes or the task is to distinguish only two classes. Since the BCE needs to calculate both $log(1 - \hat{y}_n)$ and $log(\hat{y}_n)$, the networks outputs needs to be in range from 0 to 1. This is strictly limited to the usage of the sigmoid function at the end of the neural network which is the only activation function that can convert any output of the neural network to this specific range. The following formula shows how the BCE is calculated [20, 21]:

$$BCE = -\frac{1}{M}\sum_{n=1}^{M} y_n \cdot log(\hat{y}_n) + (1 - y_n) \cdot log(1 - \hat{y}_n) \tag{1.7}$$

, where $\hat{y}_n$ is the predicted probability for the n class, the y_n is the target probability for the n class and M is the number of class labels.

Hinge loss

Hinge loss is a loss function primarily used in *Support Vector Machine* (SVM), which does not use the usual labeling convection 0 and 1, instead, it uses labeling -1 and 1. This specific type of loss function not only penalizes bad predictions, but it penalizes predictions that are correct but have low confidence (less than 1) [22, 23].

$$HL = \sum_{n=1}^{M} max(0, 1 - y_n \cdot \hat{y}_n) \tag{1.8}$$

, where $\hat{y}_n$ is the prediction probability for the nth example, the y_n is the target probability for the n example and M is the number of example used for training.

1.2.3 Optimization algorithms

Gradient descent

Gradient descent is an iterative optimization algorithm used to find the local minimum of differentiable functions. The location of the local minimum is done by calculating gradients of the function to a single parameter and optimizing the new value of the parameter in the opposite direction than the gradient. The parameter values are decremented or incremented by a portion of the gradient defined by a step size. The learning rate value is set before the optimization and the selection of the optimal value is crucial to locate the local minimum with respect to a single parameter. Large values can lead to the quick location of the local minimum but the optimizing algorithm can also miss the local minimum or/and never find it. Small step size values usually lead to the precise location of the local minimum but on the other hand, the algorithm has to perform a large number of iterations to find it. Following equation shows how the next value of parameter W is calculated [24]:

$$x_{n+1} = x_n - f^{'}(x_n) \cdot \alpha = x_n - \nabla f(x_n) \cdot \alpha \tag{1.9}$$

, where the α is the learning rate (also called step size) value and $\nabla f(x_n)$ symbolizes gradient of function with the respect to parameter W_n. In deep learning, gradient descent is used to optimize the network's parameters by computing gradients of loss functions with respect to network weights and biases. Finding the local minimum can be sufficient in some cases, however, in most cases, the local minimum can be a cause of large loss of the networks and for that reason, numerous optimizing algorithms use modified gradient descent in order to find the global minimum. Further, the vanilla gradient descent is very slow when used on large datasets since the number of terms in each gradient of the loss function with respect to a single parameter can be very large.

Stochastic gradient descent

Stochastic Gradient Descent (SGD) is an iterative method that introduces randomness into the training process and significantly improves training speed when compared to the plain gradient descent. The algorithm is based on a random selection of a single example or mini-batch of example in each iteration and it only uses those examples to calculate the gradients of the loss function with respect to the model parameters. The outcome of this method is that the number of terms calculated in each gradient is significantly reduced since the loss is not computer from all the data. For that reason, SGD is very fast and is usually able to better find the local or global minimum [25, 26].

Newton method

Newton's method is a second-order iterative method, which is similar to the gradient descent method, but it additionally uses second-order derivatives to better approximate the optimized function. The approximation achieved is no longer linear but

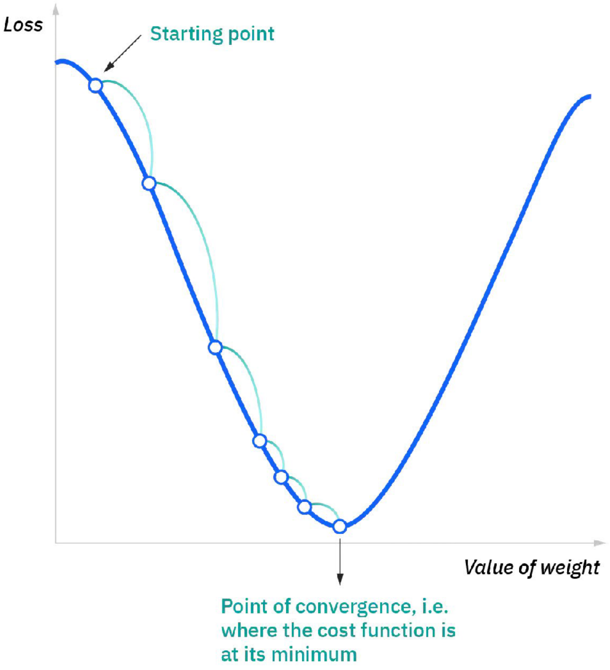


Fig. 1.10: Optimization of loss function [24]

it is quadratic, which shows some good properties in comparison to the gradient descent. One of the properties of using second-order derivatives is that the approximation curve is now quadratic and we no longer have step size training parameters in the equations, which is helpful since. Since the approximation is quadratic, the convergence is much faster than in the gradient descent method. The main issue with the Newton method is scalability. The number of operations to compute the terms become infeasible for functions with a large number of parameters and are in order of n^3, where n is the number of parameters. The following equations show a simple example of how the Newton method calculates a new value for parameter x of function $f(x)$ [26, 27]:

$$x_{n+1} = x_n - \nabla f(x_n) \cdot \nabla^2 f(x_n)^{-1} \tag{1.10}$$

or in different notation,

$$x_{n+1} = x_n - \frac{f^{'}(x_n)}{f^{''}(x_n)} \tag{1.11}$$

, where the α is the learning rate (also called step size) value and $\nabla f(x_n)$ symbolizes gradient of function with the respect to parameter W_n.

Adaptive moment estimation

Adaptive Moment Estimation (Adam) is an extended version of SGD, which increases or decreases the learning rate based on the change in the gradients in order to faster converge. The algorithms define a local gain to each parameter which is then used to compute new values for the parameters according to the following equation [26, 28]:

$$x_{n+1} = x_n - f^{'}(x_n) \cdot g_n \cdot \alpha \tag{1.12}$$

, where the α is the learning rate (also called step size) value and $\nabla f(x_n)$ symbolizes gradient of function with the respect to parameter W_n. The gain parameter is initialized with $g = 1$, and it the

1.2.4 Backpropagation

Backpropagation is a technique behind training supervised learning models which propagates the loss backwards through the networks using the chain and product rule. Conceptually, the method updates parameter in the last first and works its way back through the networks optimizing parameter as it propagates. In the most network where b_3 is a parameter of the last layer, the partial derivatives of error function with respect to parameter b_3 in the last layer is as follow [32, 33]:

$$\frac{\partial E}{\partial b_3} = \frac{\partial E}{\partial prediction}\frac{\partial prediction}{\partial b_3} \tag{1.13}$$

The resulted partial derivatives are then plugged in to the optimizing algorithm which uses it to calculate new values for the parameters. The back propagation is done in layer like manner and tensor or array of partial derivatives corresponds to each layer. Activation function such as sigmoid function make training of large models hard due to the nature of the derivation of the function, which leads to the problem of vanishing gradient mentioned earlier. As the derivative of the sigmoid function make the gradients smaller and smaller as the loss backpropagates, it causes the earlier layers to have little to no affect on the loss of the model which makes them futile. The problem of vanishing gradient however, can be counter-acted by using activation functions such as ReLu [32, 33, 34].

1.2.5 Underfitting and overfitting

Underfitting and overfitting are two main causes of the poor accuracy of the model. These states can occur during training and both result in bad accuracy for the testing data. Underfitting is a state when the model is not capable of capturing the general trend of the date. When underfitting occurs, the model performs badly both on training data and on testing data. This usually happens when the size of the training dataset is small, contains low variance and high bias or when the model is too simple. Overfitting, on the other hand, is a state when the model or algorithm starts learning noise and inaccuracies in the data which may lead to good accuracy for the training data but bad for the testing data. The causes of overfitting are high variance in the training data, high complexity of the model or algorithm, and size of the training data. Regularization techniques are used to balance between underfitting and overfitting in order to achieve high accuracy prediction both during training and testing. Figure 1.11 shows a visualization of underfitting and overfitting on a regression task [29].

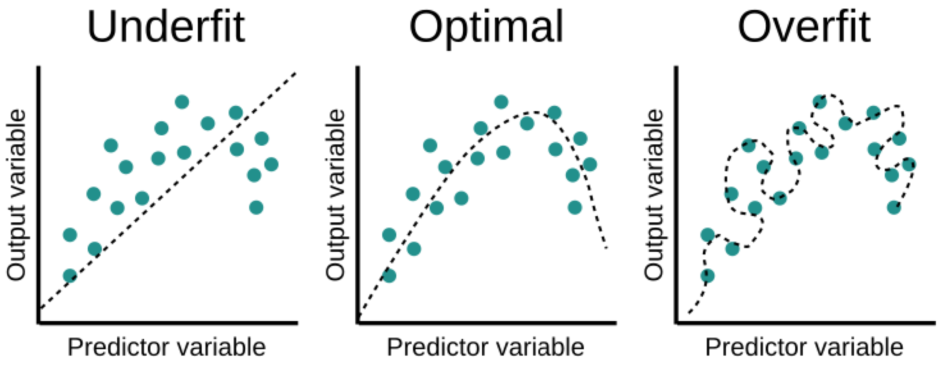


Fig. 1.11: Visualization of underfitting and overfitting [29]

Regularization

Regularization consists of numerous techniques and methods used to address the issue of underfitting and overfitting by minimizing the generalization error with little to no effect on the training error. The regularization can be done in three ways: modifying the loss function, modifying the sampling method, and modifying the training algorithm [30].

In regularization by modification of the loss function, a regularization or penalization term is added to the loss function which has the goal to penalize the model for having high-valued parameters. The most popular penalization terms used for regularization are L1 and L2 regularization. L1 regularization adds a penalty to the loss equal to the sum of the absolute value of all parameters of the model. L2 regularization, on the other hand, adds the sum of the squared values of all parameters to the loss of the model [30].

Modification of sampling method is a commonly used method consisting of two main techniques: data augmentation and cross-validation. Data augmentation uses the available training data to generate more data by applying task-specific changes to existing data. This technique is mainly used in tasks focused on image processing, where numerous augmentations such as cropping and rotation, can be performed on the images to generate new images with a different perspective. Cross validation is a technique used to compare different machine learning methods (models) in order to determine best algorithm. The dataset in cross validation is split into K blocks and during training the training data and testing data consists of different block than in previous iteration, which results in all the data contributing to both testing and validating. This lead to the minimization of overfitting and better accuracy in comparison to the training with a single data split [30, 31].

The third method is to modify the model itself and that is usually done by the dropout technique. Dropout is a technique specific to neural networks which usually consist of multiple hidden layers which take part in the output results. The dropout method randomly removes a small portion of the connections between neurons during each iteration of training which forces the model to learn a more robust set of features which leads to better generalization. The number of connections that are temporarily removed during each iteration is specified by the dropout ratio which has to be carefully selected and usually lies between 25 to 40 percent [30].

1.3 Deep learning for computer vision

In the task of object detection in real-time, there are two main architecture families to consider: *Region based Convolutional Neural Network* (RCNN) and *You Only Look Once* (YOLO), each of these architectures taking a different approach at object detection. RCNN are region-based CNN employ external region proposal method, such as selective search, which is not CNN based, and they extract regions of interest. In particular, selective search proposes 2000 *Region(s) of Interest* (RoI) which tends to capture objects in the image. Each is then processed with a convolutional network and its output is then classified by SVM, which is a supervised learning method, capable of learning a hyperplane which would separate the data based on their classes. Since the RoI are not always very accurate and might cut off a piece of an object, the bounding box regressor processes the output of CNN in parallel with SVM to correct the bounding box. The visualization of RCNN and its successors is shown in Figure 1.12. Training and inference of RCNN models were very slow because of the number of passes through CNN that had to be made. For that reason training of these models on Nvidia K20 took 13 hours and their inference was around 13 seconds per image [35].

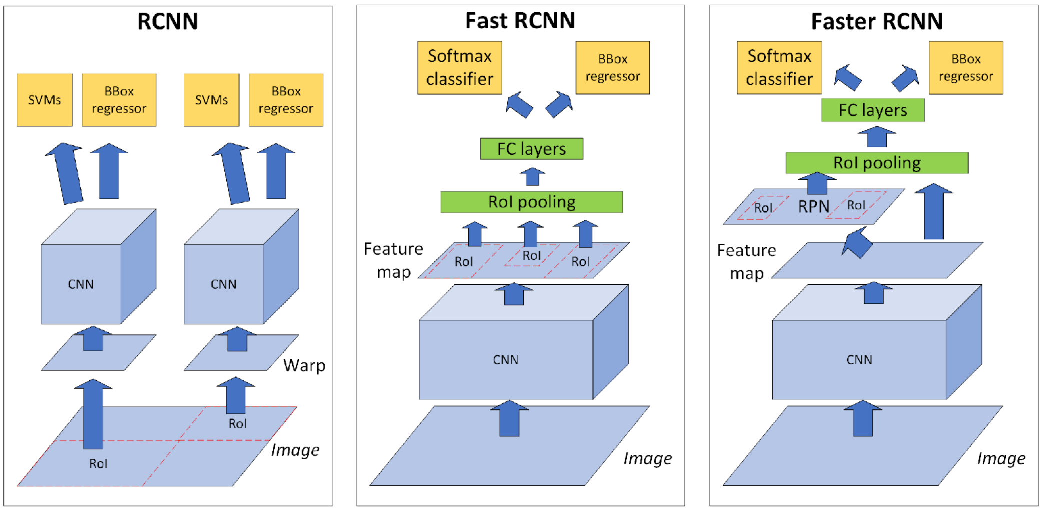


Fig. 1.12: Region based Convolutional Neural Networks architectures.

The successor of RCNN, Fast RCNN, is similar to RCNN but instead of proposing RoI from the image, it forwards the whole image through CNN to extract high-resolution feature map to which is then applied region proposal method. RoI pooling layer then passes these regions to fully connected layer(s). The output of FC layer(s) is then passed to the softmax classifier and bounding box regressor. This improvement in architecture turned out to be very effective since there is only one pass through the CNN. In terms of speed, training of fast RCNN implementations only took a couple of hours and the inference took about 2.3 seconds per image on Nvidia K40. As it turned out, the inference of Fast RCNN was dominated by region proposals, which computation took about 2 seconds. This discovery lead to Faster RCNN, which replaced the non CNN region proposal method with CNN based *Region Proposal Network* (RPN). This sped the inference time to only 0.2 seconds per image on Nvidia K40 which is fast enough for some real-time applications [36, 37].

You Only Look Once (YOLO) analyzes image as a whole, and it is able to do both object detection and classification in a single pass through the network, thus the name You Only Look Once. YOLO architecture divides the image into a grid, where each grid cell is responsible for the prediction of an object which center is in that cell. Each grid cell predicts a number of bounding boxes which are eliminated using *Intersection over Union* (IoU) technique and only the best bounding boxes are kept. The original paper notes a speed of 45 frames per second which is about ten times faster than Faster RCNN.

A couple of improvements were made over the years and there are currently five architectures labeled as YOLO. YOLOv1 (2016) [38], YOLOv2 (2017) [39] and

YOLOv3 (2018) [40] published by Joseph Redmon and his colleagues, each version bringing new improvements over its predecessor.

In 2020 Alexey Bochkovskiy published YOLOv4 which brought improvement in Darknet53 by using Cross Stage Partial technique [41]. Around the same time, YOLOv5 was introduced by Glenn Jocher, and it is surrounded by criticism and controversy for not bringing any major improvements and not having a paper written about it. Last three version has shown a lot of similarities which can better seen in Table 1.2, which demonstrates the evolution of different parts of the YOLO architecture as well the framework in which the model was originally implemented.

Tab. 1.2: YOLO Architecture overview.

	YOLOv3	YOLOv4	YOLOv5
Framework	Darknet	Darknet	PyTorch
Backbone	Darknet53	CSPDarknet53	CSPDarknet53
Neck	FPN	PANet	PANet
Head	Dense Prediction	YOLOv3 head	YOLOv3 head

The YOLO models have shown major advances over the R-CNN family over the years, mainly in terms of speed, scalability, and better performance when detecting smaller objects. For those reasons, YOLOv5 was selected as suitable model for goal of this theses and its architecture will be introduces in following subsections.

1.3.1 YOLOv5 Architecture

The architecture of YOLOv5 can be separated into three parts: Backbone, Neck and Head. The Backbone is a term used for the part of the network where the input image is processed and on its output are feature maps at three different scales. The neck is designator for the part of the network that reprocesses the feature maps and outputs more relevant feature maps. Lastly, the Head is the part of the network that converts the feature maps into predictions.

Backbone

In the YOLOv4 paper [41], the authors considered three options: CSPDarknet53, CSPResNet50 and EfficientNet-B3. Authors made experiments and theoretical research on these networks and came to a conclusion that the CSPDarknet53 was the most optimal feature extractor and its structure can be seen in Figure 1.13, which shows the largest version of YOLOv5. It consists of *Convolution Base Layer* (CBL), *Cross Stage Partial networks* (CSP) and *Spatial Pyramid Pooling* (SPP), all of which will be explained later on. The feature extractor is a series of convolutional layers which decrease the size and increase the depth. This is done by passing the feature map through a series of convolutional layers with kernel sizes 1x1 and 3x3. The 3x3 convolutional layer helps the network to keep the spatial orientation while the 1x1 convolutional layer helps to reduce the depth of the feature map.

Both YOLOv4 and YOLOv5 use this network with slight differences, the YOLOv4 uses the Mish activation function whereas YOLOv5 uses the Sigmoid activation function. Both models then use SPP [44] at the bottom of CSPDarknet53 to further increase the spatial information hidden in the feature map. The cross stage partial network modification of Darknet53 also allows for better learning since CSP allows better gradient flow in backpropagation which nearly eliminates the vanishing gradients problem.

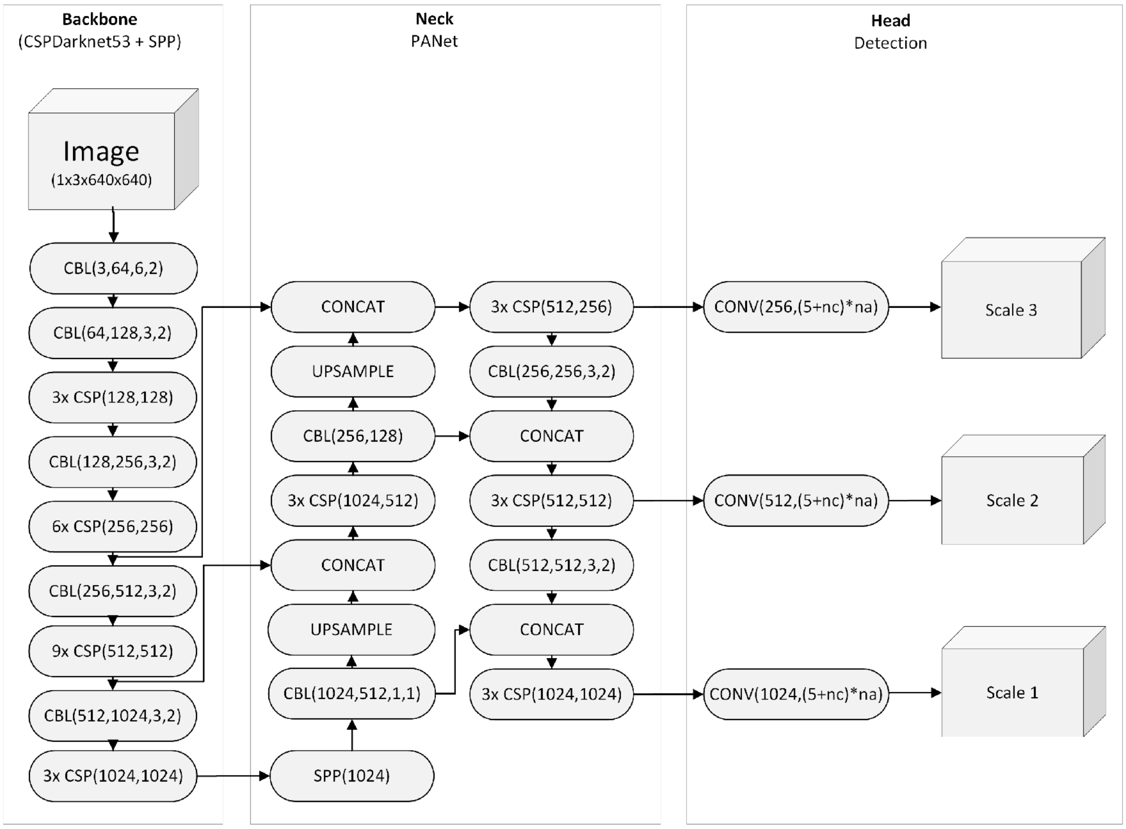


Fig. 1.13: Architecture of YOLOv5.

Neck

As a Neck, YOLOv4 and YOLOv5 use *Path Aggregation Network* (PAN) which allows the models to boost information flow not only in instance segmentation as the original paper [45] claims, but it also boosts the flow of information in object detection. The neck is designed to further process and rationale the feature maps extracted by the backbone network. This is done by a series of up-and-down sampling and in this case concatenation with previous. It is being said that the neck is a key link in the object detection task which is proven by not only experiments but also theoretically.

Head

The backbone in combination with the neck, extract feature maps which are processed through one additional convolutional layer. The output tensor contains all necessary information about the prediction for every cell on that scale. The actual size of the output is derived from the size of the input image as follow:

- Image-size / 32
- Image-size / 16
- Image-size / 8

The example of how the output tensor is structured is shown in Figure 1.14. There are B predictions, where B represents the number of anchor boxes for that scale. Each prediction contains $5 + C$ values, where C is a number of classes (Class scores) and 5 is for bounding box coordinate and objectness score. How the bounding box can be calculated is shown in Figure 1.14. The objectness score represents how sure the model is that there is an object with a center in that particular grid. Based on this information and usage of IoU and *Non-Maximum Suppression* (NMS), the algorithm is able to predict the best bounding box for the object.

1.4 Deep learning Frameworks

Framework is a software package or library that contains basic building block for designing, training and validation of deep learning networks. Following subsection will describe the most popular frameworks for implementation of deep learning models and select suitable framework which will be used for the goal of this thesis.

1.4.1 PyTorch

PyTorch is one of the top machine learning frameworks, and it is based on the Torch library. It is developed by Facebook's AI Research lab, and it is free and open-source software. PyTorch contains deep learning building blocks starting from deep learning primitives, basic NN layer types to activation and loss functions and optimizers. As it was previously mentioned, PyTorch is based on Torch, which is written in CUDA, C++, and Lua, a relatively unpopular programming language. Instead of Lua, PyTorch uses python, which makes it very popular among AI developers. It

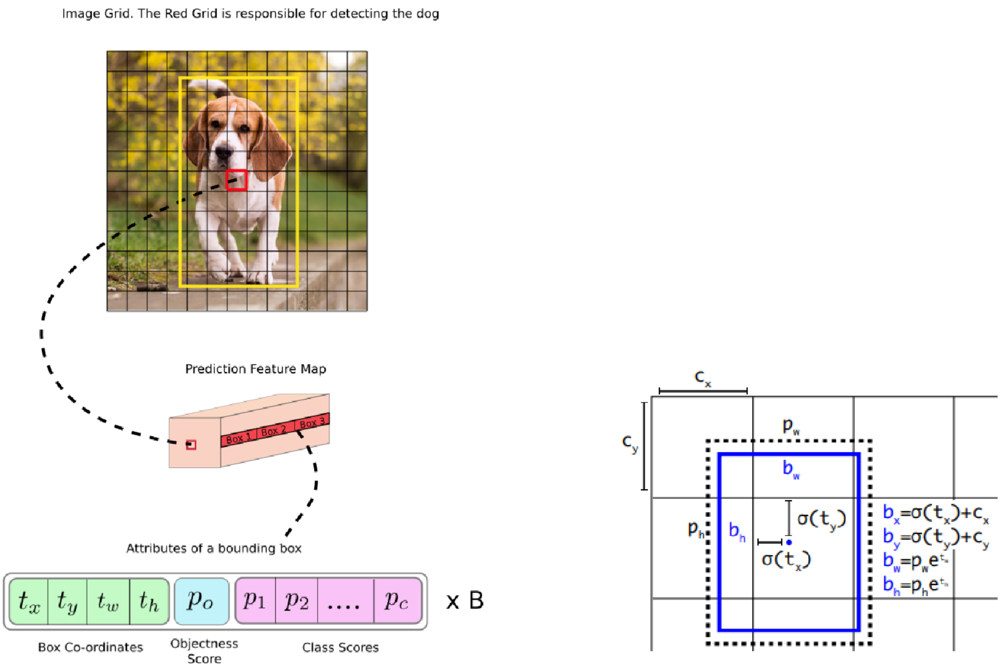


Fig. 1.14: YOLO output representation [46, 47].

contains a set of pre-trained models like Faster RCNN, Mask RCNN, and popular datasets like MS-COCO, MNIST, CIFAR, etc. [48].

1.4.2 TensorFlow

TensorFlow is an open-source framework for AI, particularly for machine learning. It is developed by Google's Google Brain team. TensorFlow can be used in a variety of programming languages, notably Python, Javascript/Java, and C++. It also allows the use of CUDA for GPU acceleration on compatible cards. In comparison to PyTorch, it is a bit more difficult to learn but it has a bigger community behind it, thus finding resources is easier. Similar to PyTorch, TensorFlow also has a couple of pre-trained models and datasets which are easily accessible [49].

1.4.3 Keras

Keras is an open-source and very simplistic framework that provides a Python interface for artificial neural networks. Keras acts as an interface for the TensorFlow library. Its original author is Francois Chollet, an active member of AI community and author of books Deep Learning with Python (edition I and II). Keras is a very intuitive open-source framework designed for developing and evaluating DL algorithms. Keras also has cross-platform capabilities, which allow better scalability and high sophisticated architectures. Keras contains pre-trained less sophisticated networks like ResNet, EfficientNet, DenseNet in numerous versions, and datasets like MNIST, CIFAR, IMBD, and Boston House price regression dataset [50].

1.4.4 MATLAB

MATLAB, which is both a programming environment and a language, has its own deep learning framework in form of a Deep Learning toolbox. It also has Simulink support which allows the more intuitive building of deep learning architectures by using building blocks and allows them to see the diagram of the network's architecture as they build it. MATLAB has also shown an increase in speed over python by roughly three times. Excluding the trial version, MATLAB is not available in any form for free, hence only a small community uses it for programming deep learning algorithms, that being mainly academic workers or students [51].

1.4.5 Nvidia Caffe

Nvidia Caffe or NVCaffe is a deep learning framework developed by the Berkley Vision and Learning Center, and it is maintained by Nvidia. It is purely coded in C++ and CUDA, and it supports interfaces like the command line, Python, and MATLAB. It has a huge database of download-ready models and a well-organized website. On the other hand, even though it supports Python and is developed by Nvidia, not many people use it, so there is not much to learn from besides the official documentation [52].

1.5 Selection of Programming Language

1.5.1 Python

Python is a very versatile object-oriented programming language ranked first in the latest annual ranking of popular programming languages by IEEE Spectrum [53] and it is also the first language at *PopularitY of Programming Language* (PYPL) index. Its power comes from a large library ecosystem, including popular modules for math, scientific computing, and machine learning. Stack Overflow trends also show increasing interest in Python, and it is currently the most questioned programming language on it. Python is liked by many for its scalability, ease of use, flexibility, and open-source nature. It supports development paradigms like object-oriented, functional imperative, and procedural [54].

1.5.2 C/C++

The two languages have been popular for many years and have been the main programming language of many people around the globe. C/C++ are considered low-level languages that are easy to learn and are used in many applications. These languages have been used a lot in machine learning and numerous libraries for machine learning were programmed in C/C++. In particular libraries like Torch and TensorFlow utilize these languages a lot [55].

1.5.3 R

R is also a very popular programming language that is designed for statistical computing and data-mining application including machine learning. It is a programming language mainly used by data scientists and data miners who are not used to coding. It is graphics-based and very easy to learn the language used in machine learning for methodologies like classification, regression, decision trees, etc. [56].

1.5.4 JavaScript/Java

Java and JavaScript were rated as the second and the fifth (respectively) most popular programming languages in the latest annual ranking by IEEE Spectrum [53]. These languages originally developed for Web applications have proven their worth even in machine learning applications due to their support for heavy data processing competencies. Companies like Google, Facebook, Microsoft are utilizing these languages for high-profile projects that process huge amounts of data [57].

1.6 Available datasets

Dataset is a pack of data used for training and validation of models. For our task, the dataset consists of images and annotations. The annotations can be represented in csv file, json file or just a regular text file. The YOLOv5 original implementation uses annotations for a single image are written in text file and each image has its own annotation text file, where the name of annotation file and image are matching except the file extensions. Following subsection will introduce couple of suitable dataset for the goal of this thesis and one of them will be selected for training of the implemented model.

1.6.1 COCO

The *Common Objects in Context* (COCO) dataset is one of the most popular datasets for object recognition there is. The COCO 2017 train/val dataset consists of more than 123K images with around 880K instances divided into 80 classes. The dataset was originally introduced in 2015 by Microsoft and later in 2017 updated by adding around 120K unlabeled images (for unsupervised learning). For our application there is a lot of irrelevant images and instances, so the sub dataset needs to be extracted. Since the goal of this thesis is to detect objects on the road, the interest goes to classes that could occur on the road which are mainly person, car, bicycle, motorcycle, car, and truck. The numbers of instances of these classes that can be found in the COCO dataset are listed in Table 1.3 [58].

Tab. 1.3: Selected instances in COCO dataset.

Class	Number of instances
Car	43 867
Motorcycle	8 725
Bicycle	7 113
Bus	6 069
Truck	9 973

1.6.2 Pascal VOC

Pascal VOC (Visual object classes) is an older and smaller dataset introduced in 2005 and developed till 2012. The current and final version of the dataset consists of 11 530 images with 27 450 RoI annotated objects of 20 classes. Classes in this dataset are structured optimally however there are some redundant classes for the goal of this thesis. The dataset gets much smaller when the selected classes are extracted [59].

1.6.3 Stanford Cars

Stanford Cars[60] is a set of 16 185 images with 196 classes and it is roughly split 50-50 into train and test images. The classes are very detailed and are typically at level *Make*, *Model*, *Year*, etc. This type of class structure is inadequate for the goal of this thesis for a couple of reason. Firstly, the dataset is small and cannot represent adequately each *Make-Model-Year*. Another problem that might come up during testing is that the dataset contains mostly single-object images whereby in our application there will be a lot of background noise and models needs to better learn features of selected classes. Lastly, the geolocation difference would be also an issue since different car models are sold at US and Europe.

1.7 Platforms for Internet of Things

A suitable platform for IoT is necessary for the task of this thesis and for that reason, this section introduces some of the popular cloud-based platforms and selects a suitable candidate for the implementation in the practical part.

1.7.1 ThingSpeak

ThingsSpeak is an IoT analytic platform developed by MathWorks® which is based in the cloud and allows users to aggregate, visualize and analyze live data streams from numerous devices. Further, the platforms allow to execution of MATLAB code directly in the cloud to extract relevant information from the received data. The platform incorporates numerous applications which can react based on analyzed data and it also has its own Python API, which allows for a single command upload and download of the data. The ThingSpeak has numerous licenses including free, student, or academic licenses, which vary in the number of channels, update interval, and annual message limit. The free version offers 3 million messages per year but only 4 channels and the update interval is limited to 15 seconds. Student license offers 33 million messages per year with update intervals down to 1 second as well as 10 channels per purchasable unit and the price starts at 55€ per unit per year. The academic license is similar to a student license with the difference in the number of channels per unit which is 250 channels for the academic license and the price starts at 250€ per unit per year [70].

1.7.2 Ubidots

Ubidots is a cloud-based platform supporting a large number of devices and cloud-based analysis with its analytic engine. The platform directly supports numerous popular devices as well as it contains APIs for multiple languages including Python and C. Ubidots is available for free as well as in a licensed version with the main difference being the number of devices. The free version allows users to utilize up to 3 devices with real-time updates. The IoT Entrepreneur license increases the device limit to 25 devices as well as adds 2-year data retention with the price being 53$ per month. Professional license costs 199$ per month and it increases the number of the device to 200 and the data retention is similar to the previous license 2 years.

The Industrial license is for large-scale projects with up to 1 000 devices for the price of 499$ per month [62].

1.7.3 ThingsBoard

Thingsboard is well known open-source IoT platform for data collection, processing, and visualization. The platform offers a wide variety of functionalities for data processing, device management, and visualization. The platform offers Python API and is available for free, however, the free version does not offer cloud services and has to be installed on other cloud platforms such as Google Cloud Platform, Azure, or DigitalOcean. The subscription-based version offers its own cloud and starts at 10$ per month for up to 30 devices, 30 assets, and up to 10 million data points per month. The Prototype subscription supports up to 100 devices and 100 assets with up to 100 million data points per month and goes for 149$ per month. The Startup subscription further increases the numbers of assets and devices to 500 and data points per month to 500 million for the price of 399$ per month. The highest subscription level offers up to 1 000 devices and 1 000 assets with 1 billion data points per month for 749$ per month [63].

1.7.4 Thinger.io

Thinger.io is an open-source platform for device management, data storage, and visualization. The platform offers support for a large variety of devices and contains numerous coding examples and guidelines for C programming language, but unfortunately does not offer Python API yet. The platform also offers a very small amount of data processing functionalities and is mainly focused on the direct visualization of data and its storage. The platform offers a free subscription for 2 devices and is limited to a single developer, community shared cloud, and basic features. The Small subscription is 25$ per month and offers unlimited devices, a private cloud, and extended features. The Medium subscription is focused on larger projects supporting unlimited devices managed by up to 5 developers and runs in a private cloud and offers business-level features as well as a custom domain for the price of 129$ per month. The Large subscription allows access to up to 15 developers, a better private cloud, 5 private domains, and on top of that it offers daily backups and costs 259$ per month. There is also an unlimited subscription for 519$ per month and it offers a high-end private cloud and unlimited features that are contained in the other subscriptions [64].

2 Traffic analysis tool

Following sections are taking insight into implementation of custom YOLOv5 in PyTorch framework. The Model and all its parts are coded using PyCharm programming environment in Python 3.8. Since the goal of this thesis is to implement real world system, appropriate hardware will be discussed and selected.

2.1 Model implementation

2.1.1 Building blocks

Based on the original implementation, the YOLOv5 architecture consists of 5 main building blocks. Convolution (Conv) module which can be seen in Figure 2.1, consists of three layers: Convolution, batch normalization, and activation. PyTorch already contains all of these layers due to which the implementation is straightforward. The Conv module is based on nn.Module class, and contains two methods for a forward pass. The first method is a simple forward method for passing the input through all three layers mentioned earlier. The second method is forward _ fuse which passes the input only through convolutional and activation layers. This is done as preparation for the implementation of fusion of convolution and batch normalization which will be mentioned later.

The second building block is the standard bottleneck needed for the cross stage partial blocks as well as for the shortcuts for the different scales. The Bottleneck module consists of two Conv Modules through which the input is passed. Depending on the input and output channels of the Bottleneck module and the value of the shortcut argument, the module either returns the result of input being passed through two Conv modules or it performs an addition between the output of the second Conv module and the input of Bottleneck module. The block diagram of the Bottleneck module is shown in the Figure 2.1:

Cross stage partial module is made out of previously implemented modules and since it contains three Conv modules, the class is labeled as C3. In C3 module, whose block diagram can be seen in Figure 2.1, the input goes through two separate branches simultaneously, wherein the first branch, the input is passed through Conv block and Bottleneck block and in the second branch, the input is passed only through Conv module. Both branches are then concatenated along the first dimension (depth) and the concatenated data are passed through the third Conv module and the output is then returned.

The fourth block of YOLOv5 is a modified version of Spatial Pyramid Pooling, which instead of defining three different max-pooling layers utilizes only one, and

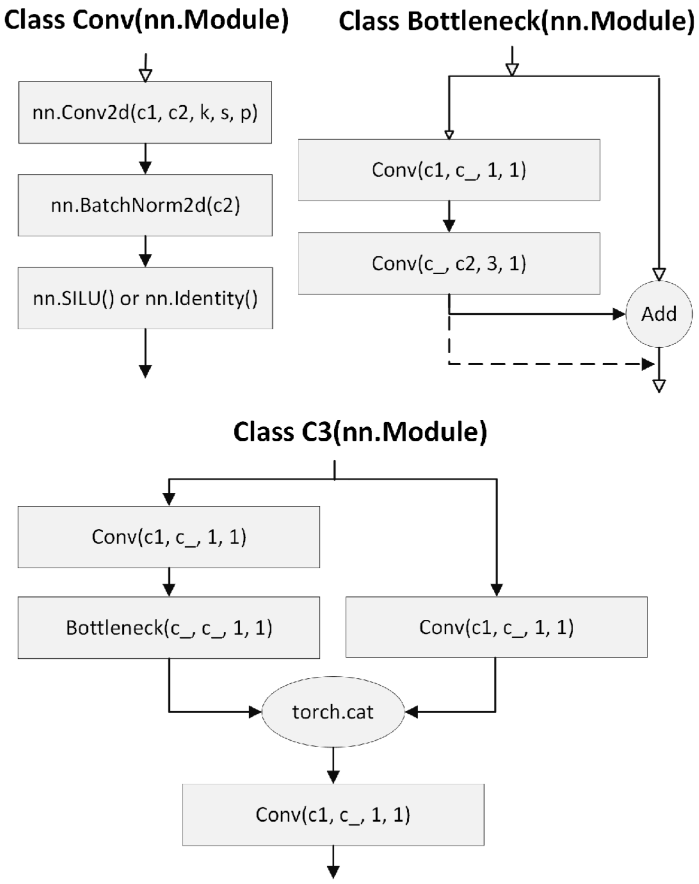


Fig. 2.1: Block diagrams of Conv, Bottleneck and C3 module.

perform multiple passes through it which slightly reduces processing speed. The block diagram of SPPF can be seen in Figure 2.2. The module passes then input through Conv module and performs three passes through the max-pooling layer, after which the input is concatenated with the output of each pass through the max-pooling layer in a specific order noted in Figure 2.2. The result of the concatenation is then passed through the last Conv module whose output is then returned by the SPPF module.

The last module is the Detect module, which transforms the output of each scale into a more convenient format. This specific module is the same used in the previous version of YOLO and as it was described in theory. It performs the last pass through a single convolutional layer for each detection layer and converts the feature maps from format $[bs, na * (nc + 5), gs, gs]$ to $[bs, nl, gs, gs, nc + 5]$, where bs is the batch size, na is the number of anchors, nc is the number of classes and nl is the number of detection layers. In case the training function is running, the class returns the transformed tensor, however in the case of inference, the forward methods further

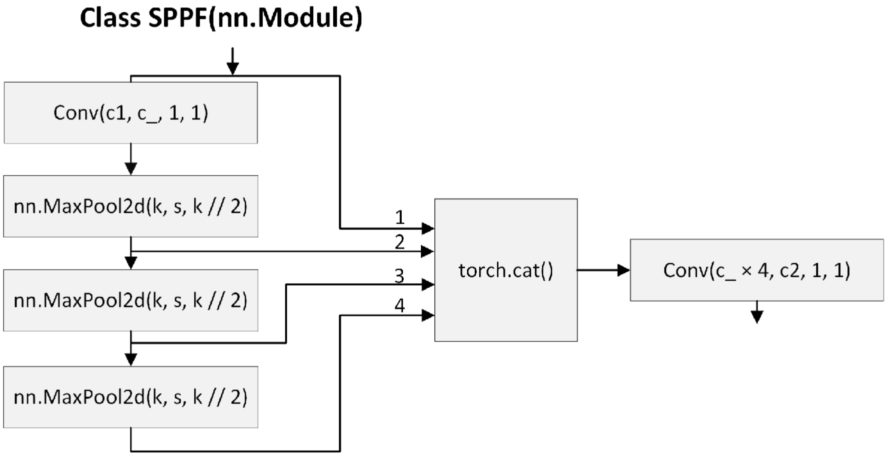

Class SPPF(nn.Module)
Conv(c1, c_, 1, 1)
nn.MaxPool2d(k, s, k // 2)
nn.MaxPool2d(k, s, k // 2)
nn.MaxPool2d(k, s, k // 2)
1
2
3
4
torch.cat()
Conv(c_ × 4, c2, 1, 1)

Fig. 2.2: Block diagrams of faster spatial pyramid pooling.

process the data by scaling the coordinates of bounding boxes and returning the tensor with predictions (Figure 1.14) as well as the transformed tensor.

2.1.2 Model

The original implementation of YOLOv5 as well as some of the previous versions of YOLO, used .yaml files to specify the order of modules in the network, dataset configuration, and hyper-parameters. The build of the model is done in a separate class called Model. This class is initialized with the architecture .yaml file and can be further specified with the number of channels, number of classes, and anchors. For easier use, the default value of channels is 3 and the numbers of channels and anchors are in default read from the config file. The init method checks the config file path and loads the information in form of a dictionary. The init method then calls parse_ model function, which loops through the input dictionary and adds each module into a nn.Sequential according to the module arguments, which are unpacked from the dictionary. This particular nn.Sequential class represents implemented model and is returned to the init method and its biases and weights are initialized. Model class contains two methods, one of which is the forward method, which takes the image as an argument and passes it through all the layers, and the second is the fuse method.

The fuse method processes the list of layers of the model and whenever there is a convolutional layer followed by batch normalization layer, it replaces that couple with single convolutional layer [67] with weights W and biases b, which effectively reduces the number of layers and allows for faster inference. The weights and biases are calculated according to the following equations:

$$\mathbf{W} = \mathbf{W_{BN}} * \mathbf{W_{CONV}} \tag{2.1}$$

$$\mathbf{b} = \mathbf{W_{BN}} * \mathbf{b_{CONV}} + \mathbf{b_{BN}} \tag{2.2}$$

2.2 Dataloader and dataset

Dataloaders are an essential part of training and validation on large datasets and take up a large portion of the code. In the following sub-sections each dataloader and its methods will be briefly explained.

2.2.1 Dataloader

Dataloader is a derived class from PyTorchs DataLoader class with the only difference being it contains is the sampler which runs forever. This dataloader is used for training and validation and uses LoadImageAndLabels class derived from PyTorch Dataset class. The initialization of this class does a couple of things. Firstly, it assigns some of the initial arguments to the class attributes which will be used later on. Next, the class looks through the directory specified in `dataset_config.yaml` and looks for images with the supported data type. The algorithm then automatically processes image paths and figures out the label paths. Caching is used on both image and label paths to improve the training speed and cache files are in default saved in RAM. The LoadImageAndLabes methods are described in the following table:

Tab. 2.1: Summary of LoadImageAndLabels class methods.

Name	Description
___init___	Loads images and labels and caches them
___len___	Lenghts of the dataset
___getitem___	Returns image and labels with specific index
cache_labels	Caches labels in a RAM memory
collate_fn	Merges batch sample

2.2.2 LoadImagesAndClips

LoadImages is a dataloader class used specifically for inference and it is capable of processing both images and video clips. Similar to the previous dataloader, when the class is initialized, it goes through the specified directory and finds all compatible images and video clips, and saves its paths. The files are read during iteration in the `__next__` method. An additional method is used to update the file path is the current file is processed and to measure speed, a setup method was created, which allows the user to input marker and distance for speed estimation. The following table briefly summarizes all implemented methods:

Tab. 2.2: Summary of LoadImagesAndClips class methods.

Name	Description
__init__	Class initialization
__iter__	Iterator method
__next__	Method for getting next image
__len__	Return number of files
next_file	Method for reading new video clip
setup	Method for setting up speed measurement

2.2.3 LoadCamera

LoadCamera class is designed to work both with USB and *Camera Serial Interface* (CSI) cameras which loads frames from the camera in the background of the actual inference, so the algorithm does not wait for the actual frame acquisition. This method works very well and increases performance. The initialization of this class creates a daemon thread targeted on the update method, which based on the camera frame rate, captures the image and passes it back as a class attribute which is later read by the algorithm and processed for inference. The setup method is run only once and its task is to acquire markers and distance information for speed measurements. All class methods are summarized in the following table:

Tab. 2.3: Summary of LoadCamera class methods.

Name	Description
__init__	Class initialization
__iter__	Iterator method
__next__	Method for getting next image
__len__	Return number of files
update	Method called by daemon thread to update image
setup	Method for setting up speed measurement

2.3 Augmentation

Augmentation is an essential part of training that helps to improve generalization which results in better performance. This project uses a couple of augmentation techniques such as mosaic augmentation, flip, hue saturation value augmentation, and perspective augmentation whose probability or value is defined in the `hyp.yaml` file.

The main augmentation performed is the mosaic augmentation introduced in the third version of YOLO and what it does is, it creates a two-by-two grid out of them which effectively puts more instances into a single image which then acts similar to what we would see if we increased batch size. Mosaic augmentation is performed in its function and besides creating the mosaic and updating the label and its bounding boxes, it applies random perspective, which is another type of augmentation that is performed. Perspective augmentation has its function and again based on hyper-parameters, it rotates, scales, shears, translates, or changes perspective using OpenCV functions.

Two more augmentations are implemented. The first is a left-right flip performed using a NumPy function. The last augmentation is *Hue Saturation Value* (HSV) which modifies the magnitude of hue, saturation, or/and value based on gain specified in the hyperparameter file.

2.4 Training

Training is done using the `train.py` file which is designed to be run from command windows with the help of an argument parser. The user can specify parameters such as device, image size, optimizer, or batch size. If all necessary data are available, the Training function starts initializing the model, data loaders, and optimizer. Since the YOLOv5 uses the exponential moving average of everything in the model state_ dict to significantly improve the learning process, `train.py` uses the same function to do the same thing since PyTorch does not have support for it yet. After all preparations are executed, the code enters a loop for epoch and then loop for batch. The code computes loss after each batch and uses backpropagation to optimize the weights and the biases. The cycle repeats until all batches have passed through this loop.

Then the validation process starts by calling the `Validation` function in `val.py`. This function similarly to the batch loop, processes the images, however instead of performing backpropagation, it computes metrics that are returned to the `Training` function. Based on these metrics, we can calculate fitness which represents the weighted combination of each metric. Based on this fitness value and the best previous value, parameters of the model (weights) are saved according to Figure 2.3.

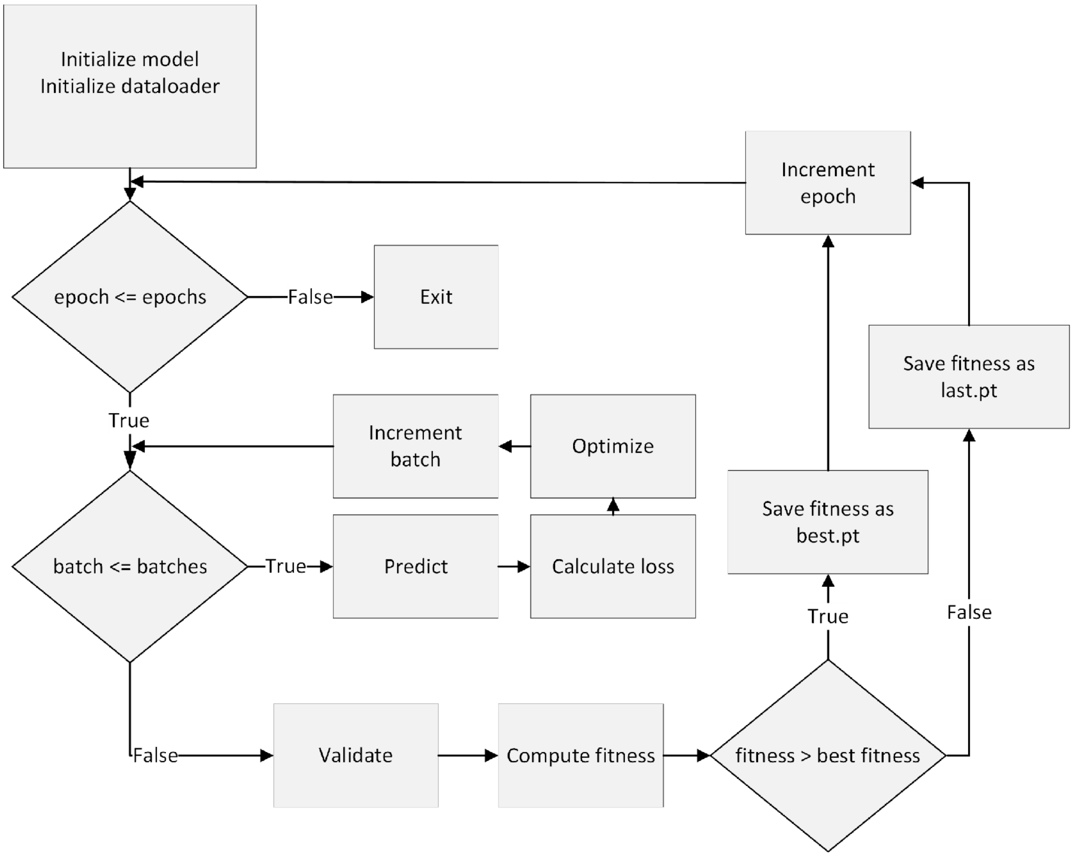


Fig. 2.3: Training loop.

2.5 Inference

For the purpose of inference, a separate python file was made to as clear as possible. In the `detect.py` file, `run` function is executed in similar fashion as it was in `train.py`. A very simple code was written for the inference due to additional custom classes which made the loading of models (weights) easy. The model module is assigned through the `Inference` class which is initialized with the paths to the weights. These weights are then loaded through function *attempt_ load* as an nn.ModuleList and going through this list assigns additional attributes and parameters to each module. The module list is then passed back to the init method of `Inference` class where it is used as a module for the forward method. After that, the model instance of a class Inference is made in the `run` function of `detect.py` file. Based on the input data (image file, video file, camera feed), the model makes predictions to which non-maximum suppresion is applied and the bounding box is re-scaled and added to the original image. If set, the file is then saved to the project directory. The block diagram of the inference loop is shown in the Figure 2.4:

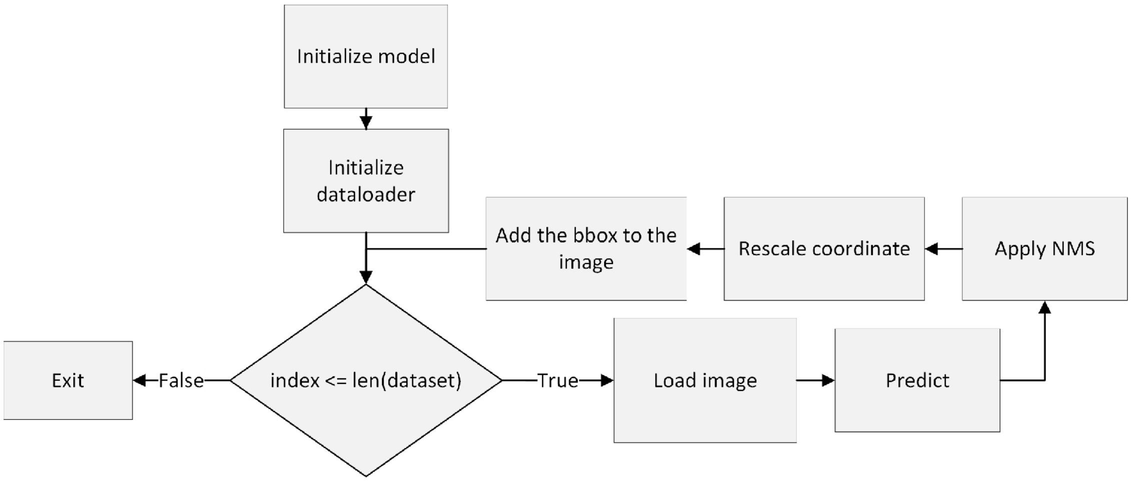


Fig. 2.4: Inference loop.

2.6 Tracking

This section introduces two simple methods for *Multi-Object Tracking* (MOT) for tracking by detection as well as summarizes their advantages and disadvantages. The main class behind both trackers is a *Track* class which is specifically made for storing attributes of tracked objects, and instance of it is created for each tracked object. The class stores information about age as well as about number of frames its been lost for and it keeps track of past class IDs of the objects, from which it take the most common predicted class. Every-time the class attributes are updated using the `update` method, the `estim_speed` method is called which checks for object center passing thresholds set during RoI selection, and assings timestamp to each threshold. Based on time of travel and distance it calculates estimated speed.

2.6.1 Centroid tracker

Centroid tracker works as the name suggests by tracking the centers of detected objects in consequential frames and calculates the euclidean distance between the centroids of the current frame and the centroids of existing objects that are being tracked. The tracked centroids are then updated to the centroids from the current frame that have the smallest euclidean distance. The tracking algorithm usually stores information about the age of the tracked object and the number of frames the object has been lost, based on which the tracked object may get deleted in case its been lost for a large number of frames. Tracking objects using this method is very dependent on the speed of objects and the detection has to be done at a frame rate where the change of object location in two consequential frames is very small.

The disadvantage of this type of object tracking is that in case the objects start overlapping, the tracking algorithm may at that point swap their IDs which results in incorrect tracking [65].

This type of tracker was implemented as a class named `CentroidTracker` which contains numerous attributes including list of tracked objects, counter vehicles for each class since last upload to cloud and values with distance and limits information for speed estimation. The implementation refers to each tracked objects as a track and the class methods mostly self explanatory. The summary of all methods of the CentroidTracker class can be seen in following table:

Tab. 2.4: Summary of CentroidTracker class methods.

Name	Description
__init__	Class initialization
_add_track	Method for adding new track
_remove_track	Method for removing lost track
_update_track	Method for updating existing track
_get_track	Method returning all existing
_preprocess_input	Method for input data conversion
update	Method performing centroid association
speed_est_setup	Method passing information for speed estimation
count	Method for counting vehicle classes
upload	Method for daemon thread which upload data to cloud

2.6.2 IoU tracker

IoU Tracker is an area-based tracker which performs IoU between bounding boxes of the tracked objects and the bounding boxes of objects detected in the current frame. The tracked objects bounding boxes are usually updated if the performed IoU is above a certain threshold which has to be fine-tuned based on the movement of detected objects. This approach partially solves the issue with object overlapping or their occlusion, complete overlapping still may result in incorrect tracking, however, that is not an issue in our case, because, the roads are mainly viewed from poles above the road [66].

The implementation consists out of class IoUTracker, which inherits methods from CentroidTracker class and overrides the update method which now instead of calculating euclidean distance between tracked objects and detected objects, performs intersection over union of bounding boxes of tracked objects and bounding boxes of detected objects. The method summary of the IoUTracker class is shown in following table:

Tab. 2.5: Summary of IoUTracker class methods.

Name	Description
__init__	Class initialization
_add_track	Method for adding new track
_remove_track	Method for removing lost track
_update_track	Method for updating existing track
_get_track	Method returning all existing
_preprocess_input	Method for input data conversion
update	Method performing IoU association
speed_est_setup	Method passing information for speed estimation
count	Method for counting vehicle classes
upload	Method for daemon thread which upload data to cloud

2.7 Selection of hardware

2.7.1 Computer selection

In most cases, computer vision models with deep learning algorithms require a lot of computational power either due to requirements of high fps or/and high resolution. For the goal of this thesis, the smallest version of YOLOv5 was chosen in order to run the model on a single-board computer. There is a couple of option to consider such as the Nvidia Jetson Nano and Google Coral Dev board as well as Raspberry Pi with additional computational power in a form of a USB accelerator such as Intel Neural Compute stick or Coral USB accelerator. The Coral Dev board is the most powerful one, with onboard *Tensor Processing Unit* (TPU) and up to 4 GB of RAM.

The Jetson Nano is less powerful than The Coral Dev board and its power is focused in *Graphical Processing Unit* (GPU) with RAM size up to 4GB. Latest Raspberry Pi is also very powerful with its *Central Processing Unit* (CPU) and up to 8GB of RAM. CPU however, is not very suitable for deep learning applications. This issue can be fixed by adding a USB accelerator such as Coral Edge TPU or Intel Neural Network stick with *Visual Processing Unit* (VPU). Due to chip shortage, most of this hardware is currently not available, for that reason, only available versions will be discussed further.

The Coral Dev board is definitely a very strong candidate for the goal of this thesis, however, it also has some disadvantages like small deep learning framework support and it is limited to Tensorflow lite. Jetson Nano on the other hand supports a wide variety of frameworks as well as has better software support. In terms of price, The Coral Dev board is currently priced at around 3 000 CZK for the 2 GB RAM version, while the Jetson Nano, is also available with 2GB RAM for around 1 500 CZK.

Based on these parameters, Jetson Nano was evaluated as the best platform for the implementation of the goal of this thesis for its flexibility, price, and frameworks support. Table 2.6 shows the specification of the 2GB RAM version of the Jetson Nano Dev board.

Tab. 2.6: Jetson Nano.

Parameter	Jetson Nano	Google Coral	Raspberry PI 4B
CPU	Cortex - A57	Cortex - A53	Cortex - A72
RAM	2/4 GB	1/4 GB	1/2/4/8 GB
GPU	128-core Maxwell™	GC7000 Lite	-
On-Board WIFI	NO	YES	YES
MIPI CSI	YES	YES	YES
Video codec	H.264/H.265	H.263/H.264/H.265	H.264/H.265

2.7.2 Camera selection

Two types of cameras can be used on Jetson nano, a USB or a CSI. The advantage of USB is the high interaction with CPU, thus it is faster when utilizing CPU. CSI on the other hand is directly routed into memory which allows much faster processing of the video feed. Both types of cameras can be used on Jetson Nano, however, the CSI version better for the goal of this thesis since it is directly connected to the GPU, and it is also less expensive than a USB camera. The CSI camera for Jetson Nano uses an IMX219-77 sensor from Sony, which is an 8 Mpx sensor capable of a resolution of 3280x2464 at 30 frames per second.

2.7.3 Additional hardware (optional)

Since the task of this thesis is to analyze the traffic and utilize the IoT server to graph statistics, the connection to the internet is necessary. Jetson Nano can be connected to the internet via Ethernet cable or WiFi USB dongle. Since the selected development kit comes with a USB WiFi dongle, It will be used as a means of connection to the internet via a mobile hotspot. The Jetson Nano requires a quite strong power supply. The manual states at least a 5V/3A power supply connected to the USB-C is needed to run the development kit without issues. Since part of the evaluation will be done in the field, a portable power supply is needed. According to internal measurements, the system draws a current of around 2.4 A when no external peripheries are connected. For the in-field test, a power bank with a capacity of 20 000 mAh is going to be used. Based on these two values, it can be estimated that the system will be able to run for around 8.3 hours.

2.8 Library requirements

Implementation of the project was done purely in PyCharm with Python 3.8 interpreter and it requires wide variety of libraries and packages. Installation on windows can be simply done using *pip*3, however in order to run the project on Jetson Nano, it is quite problematic and requires additional steps mainly in PyTorch installation. The main difference in Windows desktop PC and Ubuntu on Jetson Nano is the support by PyTorch and Python version. PyTorch supports ARM aarch64 processor architecture only with Python 3.6 version of PyTorch but same version is no longer supported on Windows. However, this is not an issues due to minor changes between Python 3.6 and 3.8, and no changes to the project were required. Nvidia offer pre-build wheels [68] for PyTorch and torchvision.

The following requirements are needed for the project:

- numpy ∼= 1.22.3
- opencv-python ∼= 4.5.5.64
- Pillow ∼= 9.0.1
- PyYAML ∼= 6.0
- requests ∼= 2.27.1
- scipy ∼= 1.8.0
- tqdm ∼= 4.63.1
- torch ∼= 1.8.2+cu111
- torchvision ∼= 0.9.2+cu111
- pandas ∼= 1.4.1
- wandb ∼= 0.12.11

3 Experiments

This chapter focuses on the experiments performed during the development of the model as well as summarizes system performance in various tasks. The COCO dataset was selected for the training of the model, however since the model would be heavily biased towards cars, the number of instances in car class was limited to 11 000 instances.

3.1 Object detection and classification

As was described earlier, the algorithm can perform inference on various sources including video clips and cameras. The model was initially tested on video clips made by Apple iPhone 12 with its 12 Mpx, f/1.6 camera. The performance in terms of object detection was good, however, there were a couple of issues with false detection such as detecting and classifying road signs or shadows as a vehicle. An example of the false detection can be seen in Figure 3.1.

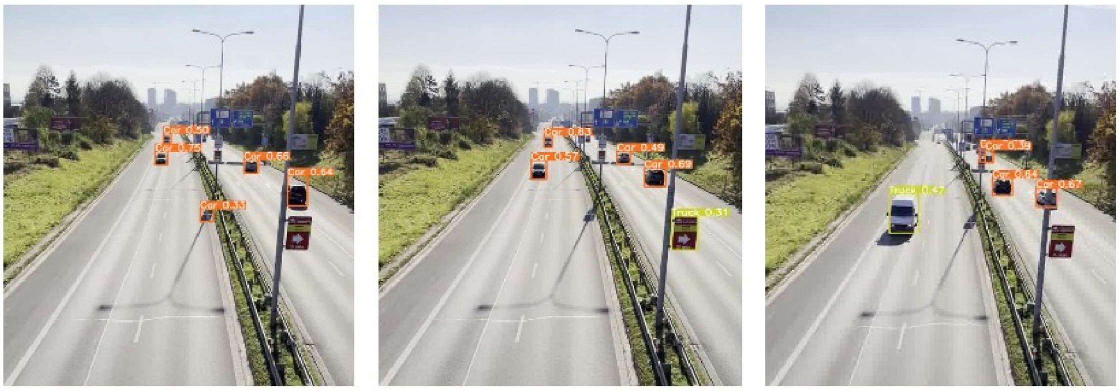


Fig. 3.1: Initial detection example

At that point the model was detecting objects with about 60 % confidence, and about 50 % of the predicted bounding boxes had intersection over union with ground truth bounding boxes higher than 50 %, which is very good and as it can be seen in Figure 3.1, bounding boxes on unseen images (video frames) are very precise. To this point, the model was purely trained on desktop PC with GTX960 4 GB graphic card.

The first trained model was a good start for fine-tuning training settings and working more with the dataset. Moving the training to the Google Colab and upgrading to Pro membership, allowed the utilization of better graphic cards, particularly the Tesla P100 with 16 GB of memory. Training on better hardware highly increases the model performance by allowing to train with batch sizes up to 64 images which resulted in better generalization and the model bounding box predictions.

Training implements a couple of metrics based on which the training algorithm evaluates model fitness. Precision is the ratio of true positives (correct predictions) to the number of positive predictions(true positives plus false positives). This represents how many of the predictions made were correct. Recall is the ratio of true positives to the total number of expected predictions. In other words, says how many of the expected objects were detected by the model. The mAP:0.5 metric is a bounding box related metric, where a true positive prediction is which has IoU of the predicted bounding box and the ground truth bounding box larger than 50 %. The metric calculates precision for each class and from that, it calculates the mean average precision. The mAP:0.5-0.95 performs similar computation except the mean average is now calculated over a ten of IoU threshold starting from 50 % to 95 % with a step of 5 %.

Figure 3.2 shows the evolution of metrics based on which the model is evaluated in the final session of training. The training showed significantly better results than the previous tests and the total detection precision reaches 78 % and 67% of the predicted bounding boxes had more than 50 % IoU with the ground truth bounding box.

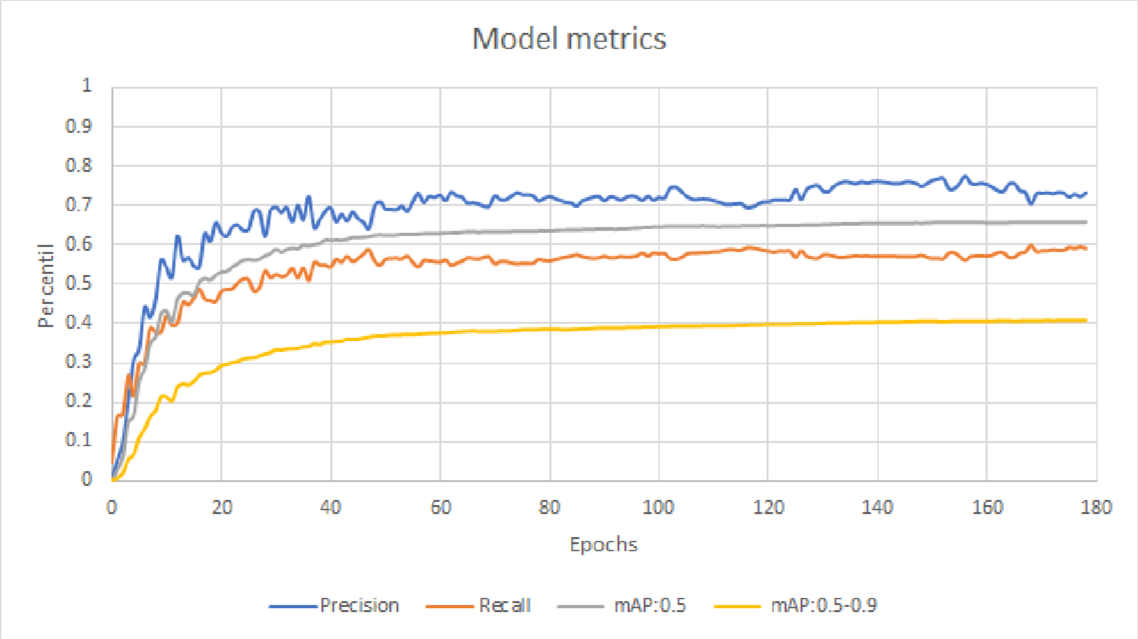


Fig. 3.2: Metric during final training session

Table 3.1 shows the final percentile of each metric for each class. These results show reasonable detection precision and metrics for evaluating bounding boxes. It also shows a sort of expected results for the bicycle and motorcycle classes which are in the dataset mainly represented in up-close pictures and without riders, which makes these two classes very similar, and due to this fact, these classes have the lowest detection precision out of all trained classes.

Tab. 3.1: Precision on specific class.

Class	Detection precision [%]	Recall [%]	mAP:0.5 [%]	mAP:0.5-0.95 [%]
Bicycle	67.3	48.4	56.6	28.1
Car	85.7	62.6	68.6	43.3
Motorcycle	64.3	81.1	71.4	38.4
Bus	83.6	71.7	79.9	61.1
Truck	68.2	50.8	59.6	37.1

The weather and car color both affected detection and its confidence. Visually the model has issues with detecting black metallic cars when the weather was cloudy and white cars on sunny days. This is most likely caused by the cars reflecting light or other objects while moving due to which the car appeared to have unusual features.

3.2 Inference speed

Inference speed is a very important aspect that influences both the system's ability to analyze in real-time and also the performance of tracking algorithms. For that reason, a couple of experiments were performed to achieve the smallest inference time possible. The first tests were performed on video clips on which the inference speed was about 38 ms per frame on PC and 130 ms per frame on Jetson Nano. The Jetson Nano performance was from the hardware perspective accurate and expected, however, the performance on PC, showed to be influenced both by the speed of the hard drive and also by the activity of background services and other applications, which caused the inference speed to be unstable (occasionally the inference speed increased up to 44ms per image). After minimizing the influence of other applications the inference on PC was stable at about 35 ms per frame.

The inference speed using a camera as a source was slightly better with an inference time of about 25 ms on PC and about 110 ms on Jetson Nano. The Jetson Nano performance, however, was not good enough for precise tracking or speed measurements, and for that reason, the model was converted to TensorRT which is a better-optimized back-end for NVIDIA GPUs, which helped to improve the inference speed on both platforms. Table 3.2 shows inference speed for both back-ends on a scenario with 10 objects and live visualization off (visualization using OpenCV was slightly increasing the inference time).

Tab. 3.2: Model speed performance on both platforms with camera as a source.

Platform	GPU	back-end	avg. FPS
Desktop PC	GTX960 OC 4GB	PyTorch	33
		TensorRT	40
Jetson Nano 2GB	128 CUDA® core NVIDIA Maxwell	PyTorch	9
		TensorRT	14

3.3 Tracker performance

Tracker is an essential part the algorithm and its flawless performance is necessary to precisely count vehicles and estimate their speed. Both the centroid and IoU trackers perform well in high frame rate conditions, however, the centroid tracker is very primitive and it often swaps IDs of the tracked objects when they get too close to each other. IoU tracker, on the other hand, was performing very well and had little to no IDs swaps and its performance was reliable. For that reason, IoU is set as the default tracker in the project.

Experiments performed on desktop PC (running at 30 FPS) showed very good results from the algorithm and there were little to no errors in tracking vehicles. The speed estimating is calculated based on the object center passing over two threshold lines for which the distance is know. In case the object is on the line or close to it (± 5 pixels) the object is time-stamped and after it reaches the second threshold line, the speed is computed based on the difference of these time-stamps and the distance entered during the setup. This presents an issue since the Jetson Nano is not capable of processing more than 14 frames per second, due to which the speed of most vehicles is not captured.

Figure 3.3 shows tracking and speed estimation using a bicycle at a speed of 25 km/h. As it can be seen, the algorithm at this speed is capable of keeping up with the objects, but unfortunately, 14 FPS was not good enough for the algorithm to capture the bicycle near the threshold lines, and the speed was not calculated. The bounding box label shows object ID, class CLS, confidence CF, and speed SP.

To test performance on higher frame rates, the road was captured using a mobile phone with 60 frames per second and when running inference on this video, it was clear that the tracking algorithm performed very well and kept track of all objects,

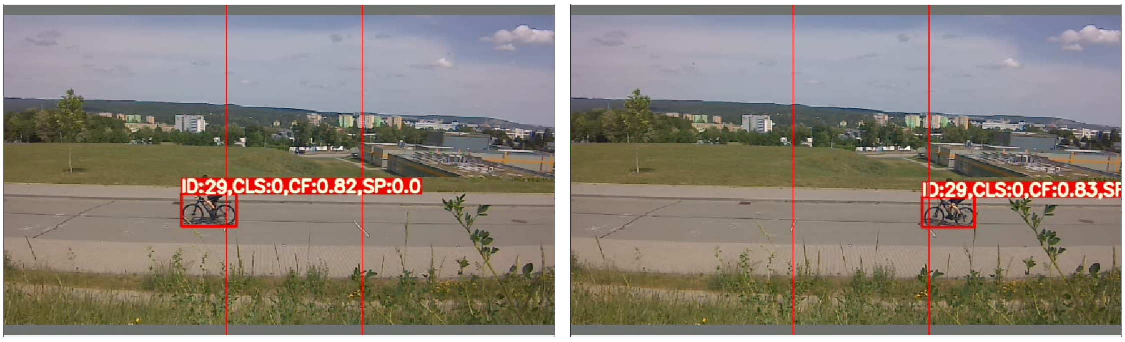


Fig. 3.3: Jetson Nano tracker performance test.

and also performed their speed estimation. Figure 3.4 shows a tracked vehicle with object number 0 and which belongs to class 1 which refers to the car class. The estimated speed is based on the distance between red lines and the time of travel. The distance between red lines was not physically measured but it was taken from the TP133 standard [69] for the horizontal road markings. In this case, the length of the line is 3 meters and the length of space between two lines is 6 meters, which makes the distance 12 meters in total.

3.4 View angles

The camera location is certainly a huge aspect both in detecting and predicting objects, but it also influences tracker performance. Practically, systems for traffic analysis are usually mounted on poles overlooking the traffic or on the side of the road. The system was tested in both mount location and in terms of prediction confidence and bounding box prediction, it performed nearly identical with a slight increase in confidence when the system was located on the side of the road. These results were expected since from the side the model can better visualize the feature and outlines of the car.

In terms of tracking, the algorithm performed better when viewed from above the road where the camera captured a larger part of the road, due to which the change in position of the vehicles between two consequential frames appeared small. That helped the tracking algorithm to perform better both in tracking and also in evaluating the speed since it was able to precisely capture the objects on the threshold lines and timestamp the frames. When viewed from the side, the view of the camera was more narrow due to the change in position of vehicles in two consequential frames being bigger and the tracker required a higher frame rate to perform as well as in the other position. The algorithm did not perform well in terms of speed estimation since when viewed from the side, the threshold line (horizontal

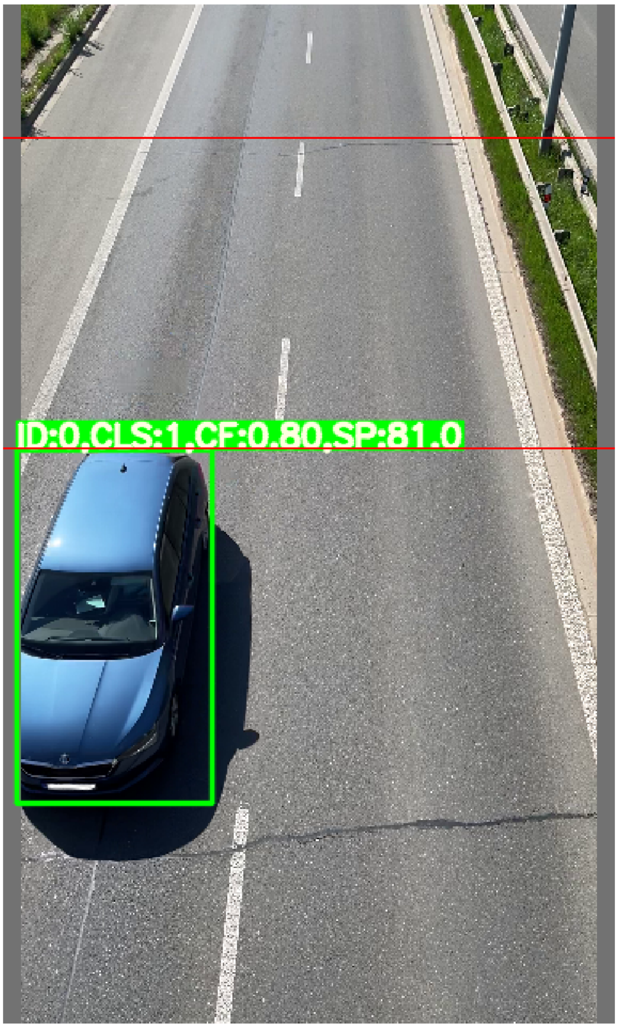


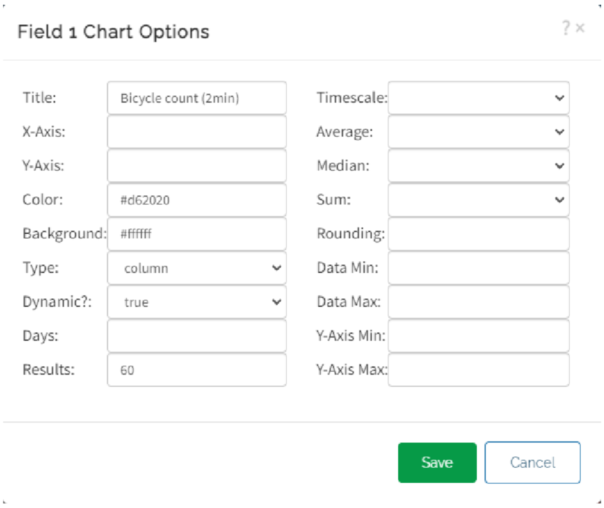
Fig. 3.4: Simulation of detection with 60 FPS

or vertical) captured different distances for each lane.

3.5 Cloud analysis and visualization

An important aspect of visualization and analysis is a cloud-based platform that is reliable, easy to use, and capable of performing additional processing. The initial data collection was performed using the Ubidots platform which was very easy to use due to its Python API. This was very effective and it allowed for a public dashboard with all the data and it allowed visitors to timelines and visualize certain parts of collected data. However the free version turned out to be used by a lot of projects and the access to the data was often unavailable for a number of days, due to which the visualization and data collection was switched to the ThingSpeak.

The data are uploaded to ThingSpeak through six separate channel fields, one for each class and one for speed estimation. Due to readiness of ThingSpeak to visualize the input data in time, its very easy to directly visualize channels with wide variety of setting as shown in Figure 3.5.



Field 1 Chart Options

Title:	Bicycle count (2min)	Timescale:	
X-Axis:		Average:	
Y-Axis:		Median:	
Color:	#d62020	Sum:	
Background:	#ffffff	Rounding:	
Type:	column	Data Min:	
Dynamic?:	true	Data Max:	
Days:		Y-Axis Min:	
Results:	60	Y-Axis Max:	

Fig. 3.5: ThingSpeak direct channel visualization

Further the platform allows to process received data using MATLAB code with its ThinkSpeak-specific functions. A custom MATLAB code was particularly used to plot the numbers of instances for each class in the same figure in order to visualize the dominant vehicles class. A bar graph visualizing the number of instances for each class was done in similar way. An example of visualization can be seen in Figure 3.6 which shows the number of instances for each class for the past 24 hours. The graph shows the car class is the superior user of this road and we can also see increased traffic flow in the morning and the afternoon at around 15 PM.

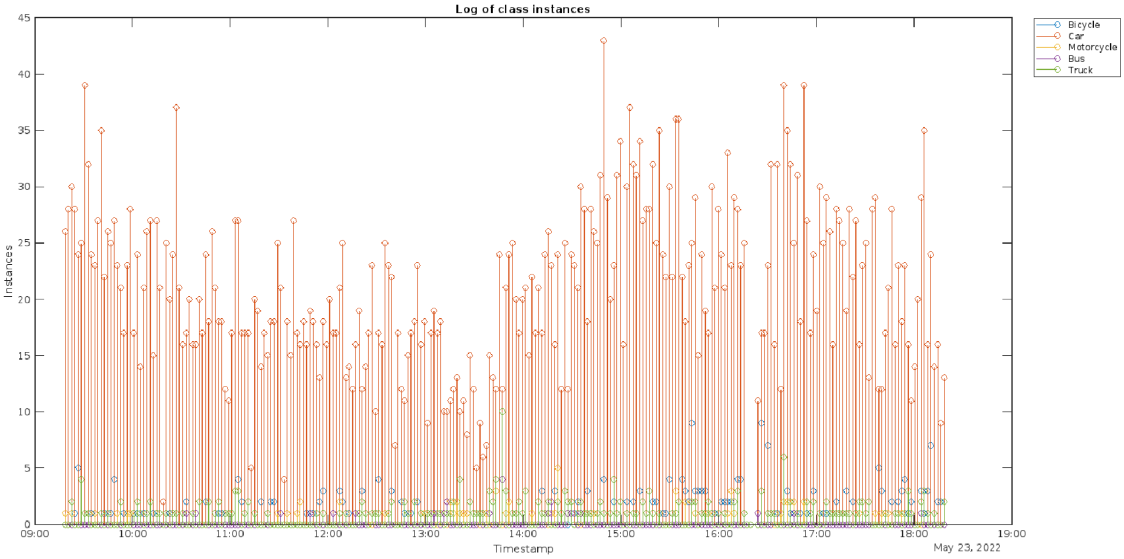


Fig. 3.6: Log of instances visualized in cloud

Conclusion

The implemented algorithm uses fully convolutional YOLOv5 architecture, which has show to be the state of the art algorithm for many application and based on numerous papers [42, 43], it has show very good balancing between speed and accuracy (mAP:0.5-0.95 metric). A IoU tracker was built on top of it to track, count, and estimate vehicle speed. The deep learning model is trained using a subset of the COCO dataset which contains five basic vehicle classes. The evaluation of the model shows that it is capable of predicting vehicles with the precision of about 78 % and predicting bounding boxes where more than 67 % of predicted bounding boxes have IoU with the ground truth boxes higher than 50 % which is very good. The algorithm initially tested and developed on desktop PC with GTX960, 16 GB of RAM, and Intel I5-6660K CPU, shows good performance in bounding box predictions and confidence on various vehicles when tested on video clips captured at the overpass at Hradecká street in Brno. The tracking algorithm works very well in high frame-rate situations (simulated using 60 FPS video clip) and is capable of estimating speed without any issues. The tracker also performed well on 30 FPS video clips, however, the speed estimations are not reliable since the algorithm was capable of capturing only a fraction of the passing cars at set threshold lines. The real system consisting of NVIDIA Jetson Nano and Sony IMX219-77, unfortunately, did not meet the expectations. The system was capable of running the algorithm at about 9 FPS using PyTorch back-end and 14 FPS with TensorRT back-end. At that frame-rate the algorithm was barely capable of tracking and usually lost tracks after they got too close to the overpass where the camera was located. This is caused by larger shifts in position when the vehicles are closer to the camera. Visually the prediction made on Jetson Nano has high confidence and has precise bounding boxes. The system was separately tested for speed estimation using a bicycle and even in this case, the algorithm was not capable of capturing the bicycle speed. The system is capable of sending collected data using the internet into a ThingSpeak cloud where it is ready to visualize road usage for each class and average speed. However, since the platform used for the realization did not perform as well as expected, the speed measurements are not possible since the frame rate is not high sufficiency for timing the travel of vehicles. That makes it impossible to make statistical graphs which would incorporate both vehicle speed information and number of instances to analyze how is each class or vehicles as a whole prone towards speeding. Further, the danger level could be estimated base on the percentage of vehicles speeding, which could be particularly important information on busy roads in the city, where there might be significantly large population of bicycle rides as well as pedestrians.

Bibliography

[1] Oliveira, Rodrigo M. S. de et al, 2017. A System Based on Artificial Neural Networks for Automatic Classification of Hydro-generator Stator Windings Partial Discharges. Available at: <https://doi.org/10.1590/2179-10742017v16i3854/> [Accessed July 31, 2022].

[2] CHOLLET, Francois. Deep Learning with Python. 2nd. New York: Simon and Schuster, 2021. ISBN 9781617296864.

[3] SALIAN, I., 2018. SuperVize Me: What's the Difference Between Supervised, Unsupervised, Semi-Supervised and Reinforcement Learning?. NVIDIA Blog: Supervised Vs. Unsupervised Learning. Available at: <https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/> [Accessed May 23, 2022].

[4] Amidi, A. & Amidi, S., Convolutional Neural Networks cheatsheet. Shervine Amidi. Available at: <https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-convolutional-neural-networks> [Accessed May 23, 2022].

[5] Szandała, T., 2020. Review and Comparison of Commonly Used Activation Functions for Deep Neural Networks. Available at: <https://arxiv.org/abs/2010.09458> [Accessed July 31, 2022].

[6] Goodfellow, I., Bengio, Y. & Courville, A., 2016. Deep learning, Cambridge: MIT Press. Available at: <http://www.deeplearningbook.org> [Accessed May 23, 2022].

[7] IBM Cloud Education, 2020. Recurrent Neural Networks. Get more of Think 2022. Available at: <https://www.ibm.com/cloud/learn/recurrent-neural-networks> [Accessed May 23, 2022].

[8] Olah, C., 2015. Understanding LSTM Networks. Home - colah's blog. Available at: <http://colah.github.io/posts/2015-08-Understanding-LSTMs/> [Accessed May 23, 2022].

[9] Phi, M., 2018. Illustrated Guide to LSTM's and GRU's: A step by step explanation. Towards Data Science. Available at: <https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21> [Accessed May 23, 2022].

[10] Kostadinov, S., 2017. Understanding GRU Networks. Towards Data Science. Available at: <https://towardsdatascience.com/understanding-gru-networks-2ef37df6c9be> [Accessed May 23, 2022].

[11] Khazri, A., 2119. Self Organizing Maps. Towards Data Science. Available at: <https://towardsdatascience.com/self-organizing-maps-1b7d2a84e065> [Accessed May 23, 2022].

[12] Dertat, A., 2017. Applied Deep Learning - Part 3: Autoencoders. Towards Data Science. Available at: <https://towardsdatascience.com/applied-deep-learning-part-3-autoencoders-1c083af4d798> [Accessed May 23, 2022].

[13] Anon., Training strategy. Neural Designer. Available at: <https://www.neuraldesigner.com/learning/tutorials/training-strategy> [Accessed August 7, 2022].

[14] Bushaev, V., 2017. How do we 'train' neural networks ?. Towards Data Science. Available at: <https://towardsdatascience.com/how-do-we-train-neural-networks-edd985562b73> [Accessed August 11, 2022].

[15] Anon., Mean absolute error. Peltarion. Available at: <https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-absolute-error> [Accessed August 11, 2022].

[16] Anon., Mean squared error. Peltarion. Available at: <https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/mean-squared-error> [Accessed August 11, 2022].

[17] Anon., Root Mean Square Error (RMSE). C3 AI - Enterprise AI. Available at: <https://c3.ai/glossary/data-science/root-mean-square-error-rmse/> [Accessed August 11, 2022].

[18] Bruballa, R.G., 2018. Understanding Categorical Cross-Entropy Loss, Binary Cross-Entropy Loss, Softmax Loss, Logistic Loss, Focal Loss and all those confusing names. Raúl Gómez blog. Available at: <https://gombru.github.io/2018/05/23/cross_entropy_loss/> [Accessed August 11, 2022].

[19] Anon., Categorical crossentropy. Peltarion. Available at: <https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/categorical-crossentropy> [Accessed August 11, 2022].

[20] Anon., Binary crossentropy. Peltarion. Available at: <https://peltarion.com/knowledge-center/documentation/modeling-view/build-an-ai-model/loss-functions/binary-crossentropy> [Accessed August 11, 2022].

[21] Godoy, D., 2018. Understanding binary cross-entropy / log loss: a visual explanation. Towards Data Science. Available at: <https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a> [Accessed August 11, 2022].

[22] Aliyev, V., 2020. A definitive explanation to the Hinge Loss for Support Vector Machines. Towards Data Science. Available at: <https://towardsdatascience.com/a-definitive-explanation-to-hinge-loss-for-support-vector-machines-ab6d8d3178f1> [Accessed August 5, 2022].

[23] Chowdhury, K., 2019. Understanding loss functions : Hinge loss. Medium. Available at: <https://medium.com/analytics-vidhya/understanding-loss-functions-hinge-loss-a0ff112b40a1> [Accessed August 11, 2022].

[24] IBM Cloud Education, Gradient Descent. IBM. Available at: <https://www.ibm.com/cloud/learn/gradient-descent> [Accessed August 7, 2022].

[25] Srinivasan, V.A., 2019. Stochastic Gradient Descent — Clearly Explained !!. Towards Data Science. Available at: <https://towardsdatascience.com/stochastic-gradient-descent-clearly-explained-53d239905d31> [Accessed August 11, 2022].

[26] Ruder, S., 2016. An overview of gradient descent optimization algorithms. Sebastian Ruder. Available at: <https://ruder.io/optimizing-gradient-descent/index.html#stochasticgradientdescent> [Accessed August 11, 2022].

[27] De Luca, G., 2020. Gradient Descent vs. Newton's Gradient Descent | Baeldung on Computer Science. Baeldung on CS. Available at: <https://www.baeldung.com/cs/gradient-descent-vs-newtons-gradient-descent> [Accessed August 11, 2022].

[28] Bushaev, V., 2018. Adam — latest trends in deep learning optimization. Towards Data Science. Available at: <https://towardsdatascience.com/adam-latest-trends-in-deep-learning-optimization-6be9a291375c> [Accessed August 11, 2022].

[29] Educative Answers Team, Overfitting and underfitting. Available at: <https://www.educative.io/answers/overfitting-and-underfitting> [Accessed August 8, 2022].

[30] Anwar, A., 2021. Types of Regularization in Machine Learning. Towards Data Science. Available at: <https://towardsdatascience.com/types-of-regu

larization-in-machine-learning-eb5ce5f9bf50> [Accessed August 11, 2022].

[31] Lyashenko, V. & Jha, A., 2022. Cross-Validation in Machine Learning: How to Do It Right. Blog - neptune.ai. Available at: <https://neptune.ai/blog/cross-validation-in-machine-learning-how-to-do-it-right> [Accessed August 11, 2022].

[32] Starmer, J., 2020. Neural Networks Pt. 2: Backpropagation Main Ideas. Youtube. Available at: <https://www.youtube.com/watch?v=IN2XmBhILt4> [Accessed August 11, 2022].

[33] Raschka, S., How to compute gradients with backpropagation for arbitrary loss and activation functions?. Dr. Sebastian Raschka. Available at: <https://sebastianraschka.com/faq/docs/backprop-arbitrary.html> [Accessed August 11, 2022].

[34] Wang, C.-F., 2019. The Vanishing Gradient Problem. Towards Data Science. Available at: <https://towardsdatascience.com/the-vanishing-gradient-problem-69bf08b15484> [Accessed August 11, 2022].

[35] Girshick, R. et al., 2014. Rich feature hierarchies for accurate object detection and semantic segmentation. Available at: <http://arxiv.org/abs/1311.2524v5> [Accessed July 31, 2022].

[36] Girshick, R., 2015. Fast R-CNN. Available at: <http://arxiv.org/abs/1504.08083v2> [Accessed July 31, 2022].

[37] Ren, S. et al., 2016. Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. Available at: <http://arxiv.org/abs/1506.01497v3> [Accessed July 31, 2022].

[38] Redmon, J. et al., 2015. You Only Look Once: Unified, Real-Time Object Detection. Available at: <http://arxiv.org/abs/1506.02640v5> [Accessed July 31, 2022].

[39] Redmon, J.& Farhadi, A., 2016. YOLO9000: Better, Faster, Stronger. Available at: <http://arxiv.org/abs/1612.08242v1> [Accessed July 31, 2022].

[40] Redmon, J. & Farhadi, A., 2018. YOLOv3: An Incremental Improvement. Available at: <http://arxiv.org/abs/1804.02767v1> [Accessed July 31, 2022].

[41] Bochkovskiy, A., Wang, C.-Y. & Liao, H.-Y.M., 2020. YOLOv4: Optimal Speed and Accuracy of Object Detection. Available at: <https://arxiv.org/abs/2004.10934v1> [Accessed July 31, 2022].

[42] Boesch, G., Object Detection in 2022: The Definitive Guide. Available at: <https://viso.ai/deep-learning/object-detection/> [Accessed August 10, 2022].

[43] Baussard, A. et al., 2020. YOLO-Fine: One-Stage Detector of Small Objects Under Various Backgrounds in Remote Sensing Images. Remote Sensing. Available at: <https://doi.org/10.3390/rs12152501> [Accessed August 10, 2022].

[44] He, K. et al., 2014. Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition. Available at: <http://arxiv.org/abs/1406.4729> [Accessed July 31, 2022].

[45] Liu, S. et al., 2018. Path Aggregation Network for Instance Segmentation. Available at: <http://arxiv.org/abs/1803.01534> [Accessed July 31, 2022].

[46] Kathuria, A., 2018. How to implement a YOLO (v3) object detector from scratch in PyTorch: Part 1. Paperspace Blog. Available at: <https://blog.paperspace.com/how-to-implement-a-yolo-object-detector-in-pytorch/> [Accessed July 31, 2022].

[47] Guo, R., Li, S. & Wang, K., 2020. Research on YOLOv3 algorithm based on darknet framework. Available at: <https://doi.org/10.1088/1742-6596/1629/1/012062> [Accessed July 31, 2022].

[48] Facebook AI Research, PyTorch. (2022). Available at: <https://pytorch.org/> [Accessed May 23, 2022].

[49] Google Brain Team, 2022. Tensorflow. Available at: <https://www.tensorflow.org/> [Accessed May 23, 2022].

[50] François Chollet and collegues, 2022. Keras. Available at: <https://keras.io/> [Accessed May 23, 2022].

[51] MathWorks, 2022. Matlab. Available at: <https://ch.mathworks.com/> [Accessed May 23, 2022].

[52] NVIDIA Corporation, 2022 Caffe. Available at: <https://www.nvidia.com/en-au/data-center/gpu-accelerated-applications/caffe/> [Accessed May 23, 2022].

[53] Cass, S. et al., 2021. Top Programming Languages 2021. IEEE Spectrum. Available at: <https://spectrum.ieee.org/top-programming-languages/#toggle-gdpr> [Accessed July 31, 2022].

[54] Python Software Foundation , Python. (2022), Available at: <https://www.python.org/> [Accessed May 23, 2022].

[55] Standard C++ Foundation , C++. (2022), Available at: <https://isocpp.org/> [Accessed May 23, 2022].

[56] The R Foundation , R. (2022), Available at: <https://www.r-project.org/> [Accessed May 23, 2022].

[57] Oracle and/or its affiliates , Java. (2022), Available at: <https://www.java.com/en/> [Accessed May 23, 2022].

[58] Lin, T.-Y. et al., 2014. Microsoft COCO: Common Objects in Context. Available at: <https://arxiv.org/abs/1405.0312> [Accessed July 31, 2022].

[59] Everingham, M. et al., 2009. The PASCAL Visual Object Classes (VOC) Challenge. Available at: <https://doi.org/10.1007/s11263-009-0275-4> [Accessed July 31, 2022].

[60] Krause, J. et al., 2013. 3D Object Representations for Fine-Grained Categorization. Available at: <https://ieeexplore.ieee.org/document/6755945> [Accessed July 31, 2022].

[61] Anon., ThingsSpeak. Available at: <https://thingspeak.com> [Accessed August 10, 2022].

[62] Anon., Ubidots. Available at: <https://ubidots.com/> [Accessed August 10, 2022].

[63] Anon., ThingsBoard. Available at: <https://thingsboard.io/> [Accessed August 10, 2022].

[64] Anon., Thinger.io. Available at: <https://thinger.io/> [Accessed August 10, 2022].

[65] Venkateswarlu, R., Sujata, K.V. & Rao, B.V., 1992. Centroid tracker and aimpoint selection. Available at: <https://doi.org/10.1117/12.138205> [Accessed July 31, 2022].

[66] Bochinski, E., Eiselein, V. & Sikora, T., 2017. High-Speed Tracking-by-Detection Without Using Image Information. Available at: <http://elvera.nue.tu-berlin.de/files/1517Bochinski2017.pdf> [Accessed July 31, 2022].

[67] Nenad, M., Fusing batch normalization and convolution in runtime. Nenad's research notes. Available at: <https://nenadmarkus.com/p/fusing-batchnorm-and-conv/> [Accessed July 31, 2022].

[68] dusty_nv, 2019. PyTorch for Jetson - version 1.11 now available. NVIDIA Developer Forums. Available at: <https://forums.developer.nvidia.com/t/pytorch-for-jetson-version-1-11-now-available/72048> [Accessed May 24, 2022].

[69] Anon., 2012. ZÁSADY PRO VODOROVNÉ DOPRAVNÍ ZNAČENÍ NA POZEMNÍCH KOMUNIKACÍCH, Českomoravská asociace dopravního značení. Available at: <https://www.cmadz.cz/projednavane-predpisy/files/TP_133-1.verze_12-12.pdf>.

[70] Zelený, O., 2022. Traffic analysis tool. ThingSpeak. Available at: <https://thingspeak.com/channels/1627191> [Accessed May 24, 2022].

Symbols and abbreviations

Adam	Adaptive Moment Estimation
AI	Artificial Intelligence
AN	Artificial Neuron
BCE	Binary Cross-Entropy
CBL	Convolution Base Layer
CCE	Categorical Cross-Entropy
CNN	Convolutional Neural Network
COCO	Common Objects in Context
CPU	Central Processing Unit
CSI	Camera Serial Interface
CSP	Cross Stage Partial networks
DL	Deep learning
FC	Fully-Connected layer(s)
HSV	Hue Saturation Value
GPU	Graphical Processing Unit
GRU	Gated Recurrent Unit
IoU	Intersection over Union
LSTM	Long Short-Term Memory
ML	Machine learning
MOT	Multi-Object Tracking
MS-COCO	Microsoft-Common Object in Context
MAE	Mean Absolute Error
MSE	Mean Squared Error
NMS	Non-Maximum Suppression

NN	Neural Network
PAN	Path Aggregation Network
PYPL	PopularitY of Programming Language
RCNN	Region based Convolutional Neural Network
RMSE	Root Mean Squared Error
RNN	Recurrent Neural Network
RoI	Region(s) of Interest
RPN	Region Proposal Network
SGD	Stochastic Gradient Descent
SOM	Self Organizing Map
SPP	Spatial Pyramid Pooling
SVM	Support Vector Machine
TPU	Tensor Processing Unit
VOC	Visual Object Classes
VPU	Visual Processing Unit
YOLO	You Only Look Once

List of appendices

A Content of the electronic attachment

- / ..... Root directory of the archive
  - Python files ..... Source code files
    - dataset.py
    - detect.py
    - iou_tracker.py
    - tracker.py
    - train.py
    - utilities.py
    - val.py
    - yolo.py
  - YAML files ..... Project configuration files
    - dataset_config.yaml
    - hyp.yaml
    - model_config.yaml
  - Text file ..... Project requirements file
    - GNU GPL-V3 license.txt
    - MIT license.txt
    - requirements.txt