BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

DETECTION AND CLASSIFICATION OF VEHICLES FOR EMBEDDED PLATFORMS

DETEKCE A KLASIFIKACE VOZIDEL PRO VESTAVĚNÉ PLATFORMY

BACHELOR’S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR **PATRIK SKALOŠ**
AUTOR PRÁCE

SUPERVISOR **Ing. JAKUB ŠPAŇHEL**
VEDOUCÍ PRÁCE

BRNO 2023

Abstract

An abstract of the work in English will be written in this paragraph.

Abstrakt

Keywords

Here, individual keywords separated by commas will be written in English.

Klíčová slova

Reference

SKALOŠ, Patrik. *Detection and Classification of Vehicles for Embedded Platforms.* Brno, 2023. Bachelor's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Jakub Špaňhel

Detection and Classification of Vehicles for Embedded Platforms

Declaration

I hereby declare that this Bachelor's thesis was prepared as an original work by the author under the supervision of Mr. X The supplementary information was provided by Mr. Y I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Patrik Skaloš
May 1, 2023

Acknowledgements

Here it is possible to express thanks to the supervisor and to the people which provided professional help (external submitter, consultant, etc.).

Contents

List of Figures

Chapter 1

Introduction

Chapter 2

Background and Related Work

2.1 Common Approaches to Vehicle Detection

Camera-based object detectors based on convolutional neural networks are evaluated in this paper. However, to provide further context to the problem, other commonly used solutions for the problem of vehicle detection are briefly explained in this section – non-camera-based object detectors and camera-based image processing techniques.

2.1.1 Vehicle Detection Without a Camera

Although camera-based vehicle detection systems have gained significant popularity due to advancements in computer vision and machine learning, there are alternative techniques that do not rely on cameras for vehicle detection. These methods offer different advantages, mainly reduced computational requirements or improved performance in certain environmental conditions. Since each detector has its own limitations, a vehicle detection system typically consists of different types of detectors to overcome these limitations. Following is a summary of non-camera-based techniques widely used for vehicle detection tasks.

Magnetic Sensors

One of the simplest solutions to detect vehicles is to use an induction loop.[4] The sensor, composed of a single wire, can be buried under concrete and detect the presence of metal objects passing by. With a controller, induction loops typically provide data about vehicle presence, but using more advanced algorithms, speed, approximate classification and much more can be determined from this simple sensor. Although the design is very simple, installation is not and induction loops can even get damaged over time, while repairs call for temporary shutdowns of roads. It is worth noting that there are many other types of magnetic sensors, which can provide more detailed and accurate data with simpler sensor installation, but the idea stays the same.

Ultrasonic Sensors

Another type of inexpensive and simple detector is an ultrasonic sensor.[25] These sensors can operate in a variety of conditions and are typically mounted on existing infrastructure. These distance measurement sensors are usually used to detect the presence and distance of nearby vehicles and their speed, but they can be utilized in many different ways to even provide a simple shape of a vehicle to classify it.

Radars and Lidars

Radars (Radio Detection and Ranging), which can also be mounted on existing infrastructure above ground, work similarly to ultrasonic sensors, but a single radar can oversee a much wider area of a road.[11] They are usually used to detect the presence, speed, heading and shape of a vehicle. The shape can then be used to predict a class of the vehicle. Lidars (Light Detection and Ranging) can be used in a similar way, but are far more accurate, which makes them better at detecting shapes and locations of objects.

2.1.2 Vehicle Detection Based on Image Processing

Image processing techniques take a camera feed as input and in addition to detecting the presence, speed, heading and shape of a vehicle, they can also provide its color, license plate and countless other characteristics, limited mostly by the software, not by the detector. These camera-based systems, which are relatively inexpensive, can be mounted on existing infrastructure, do not emit any energy and are highly versatile, while providing a large amount of data.

An algorithm applies a series of pre-defined filters and transformations to an image to extract patterns and features that resemble vehicles. Although modern convolutional neural networks (explained in the following section 2.2) are generally more effective at recognizing more complex patterns, image processing can still be extremely helpful for simpler tasks or as a component of a larger vehicle detection system.

A huge amount of research has been conducted on using image processing to solve the problem of vehicle detection [29]. Many of the techniques proposed can operate in real-time and are very reliable in typical environmental conditions. However, they can be sensitive to changes in illumination or have their performance affected by rain, snow, shadows, occlusions, or noise. While there are methods for addressing some of these challenges, such as shadow removal techniques [21], they add to the computational requirements of the system. Overall, our research suggests that the image processing approach is well-suited for simpler tasks or systems with lower accuracy requirements, but it is often difficult to implement and may lack versatility in more complex or demanding scenarios.

2.2 Convolutional Neural Networks

[[Lepšie ich popísať? Sú základom projektu, no]]

Convolutional neural networks (CNNs) are a class of deep learning models that have gained widespread popularity in recent years, mainly thanks to their ability to learn hierarchical features from raw image data. CNNs are particularly useful in the field of computer vision for various tasks, including image classification, object detection and segmentation. In the context of vehicle detection, these models have demonstrated a superior combination of accuracy, efficiency and flexibility compared to traditional detection methods explained earlier. For example, compared to object detection based on image processing, the main advantage of CNNs is that they are able to learn automatically from raw image data instead of relying on pre-defined filters and transformations. [19]

2.2.1 Architectures of Convolutional Neural Networks

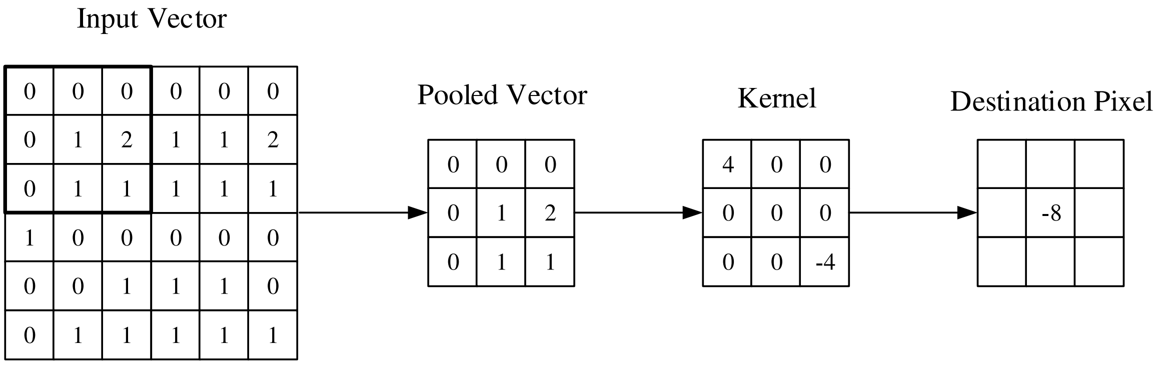
A typical CNN consists of several key components, including convolutional layers, pooling layers, activation functions and fully connected layers. In this subsection, details of these key components and their roles in the context of object detection are explained. Please note that a modern convolutional neural network consists of more building blocks which will not be discussed here to keep this introduction short.

Convolutional Layers

Convolutional layers form the backbone of a CNN and perform 2D convolution operations on the input data using trainable kernels to detect specific patterns or features. A kernel, also known as a filter, is responsible for detecting a particular feature, such as an edge or a texture. It is practically a matrix, usually small in spatial dimensionality and its weights are trainable parameters adjusted during the training phase. [22]

A two-dimensional convolution is a mathematical operation that involves computing element-wise multiplications of the convolution kernel and the corresponding sub-region of the input image. This process is typically repeated throughout the entire image in a sliding manner, resulting in a new image (matrix) as an output called a feature map. It can be better explained by Figure 2.1.

Many filters (often of different types) are used in a typical CNN and together produce a set of feature maps that capture countless different aspects of the input image.



0	0	0	0	0	0
0	1	2	1	1	2
0	1	1	1	1	1
1	0	0	0	0	0
0	0	1	1	1	0
0	1	1	1	1	1

0	0	0
0	1	2
0	1	1

4	0	0
0	0	0
0	0	-4

	-8	

Figure 2.1: Example of computing a 2D convolution on an input vector with an example kernel, from [22].

Pooling Layers

The pooling layers serve to gradually reduce the spatial dimensions of feature maps output from the convolutional layers while retaining as much information as possible. These layers improve the computational efficiency of the CNN by reducing the number of its parameters. Generally, the pooling operation takes a window or a filter and moves it over the input feature map in strides, most commonly taking the maximum value of the inputs within the window. This operation is called max-pooling. [22]

Fully-connected Layers

A fully-connected layer, also known as a dense layer is a type in which each neuron in the next layer is connected to each neuron in the previous one. In a CNN, it is typically used at the end of the network to classify the input data (features extracted by the previous convolutional layer). [22]

2.2.2 Popular Architectures of Convolutional Neural Networks

[[Treba vôbec toto? Nejak to neviem naviazať na nasledujúce subsections]]

Over the years, various CNN architectures have been developed to address different challenges and requirements. In this subsection, we review several influential architectures which achieved state-of-the-art[1] performance in a wide range of tasks, including image classification, object detection or semantic segmentation.

[[skontrolovať či som niečo nesplietol a či je všetko správne]]

LeNet-5

LeNet-5 [18], introduced by Yann LeCun and his team in 1998 is considered one of the first successful applications of convolutional neural networks. Designed for handwritten digit recognition, LeNet-5 was the source of inspiration for modern CNNs with its combination of convolutional, pooling and fully connected layers.

AlexNet

Developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton in 2012, AlexNet [16] marked a significant breakthrough in the field of deep learning and won the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) by a considerable margin. It popularized the use of deep CNNs for image classification and featured the use of Rectified Linear Units (ReLU) as activation functions, dropout technique for regularization and data augmentations for training.

VGGNet

VGGNet [24], proposed by Karen Simonyan and Andrew Zisserman in 2014, is best known for its uniform architecture, consisting of a series of stacked convolutional layers with 3×3 filters followed by max-pooling. VGGNet demonstrated that deeper networks generally achieve better performance, achieving top results in the ILSVRC with its 16-layer and 19-layer variants (VGG-16 and VGG-19). However, its success lies in being very computationally expensive.

ResNet

In 2015, ResNet (Residual Network) [13] was introduced by Kaiming He and his team, addressing the degradation problem that occurs in very deep CNNs. ResNet incorporates skip connections, enabling the network to effectively "skip" layers during the training process. This innovation allowed ResNet to scale up to hundreds of layers while improving performance, which was thought impossible. ResNet achieved state-of-the-art results in various computer vision tasks and has inspired many subsequent architectures.

MobileNet

MobileNet [14], developed by Andrew G. Howard and his team at Google in 2017, is a popular light-weight convolutional neural network architecture that has been widely used on mobile and embedded devices with limited computational resources. One of its key

[1] The term "state-of-the-art" refers to methods that have achieved superior results compared to previous best methods in a specific task or application.

features is its use of depthwise separable convolutions. Traditional convolutional layers perform a full convolution on the input, but depthwise separable ones perform a depthwise convolution followed by a pointwise convolution, which reduces the number of computations required while maintaining accuracy. Another advantage is the model's small size and low computational requirements compared to previously discussed networks allowing for real-time use, of course by sacrificing some accuracy.

[[Dať niekam MobileNetV2 keď ho už používam?]]

2.3 Object Detection

While the CNNs discussed in the previous section section 2.2 can be used for image classification[2], the object detection task also involves localization – marking all objects in the input image by a bounding box. Modern object detectors can be divided into two categories – one-stage detectors and two-stage detectors. **[[Že sa tie architektúry vrátane mobilenetu dajú použiť ako backbone?]]**

2.3.1 Two-Stage Object Detectors

In the two-stage object detection task, the first stage selects region proposals (selecting regions that are likely to contain an object) and passes them to the second stage for classification. In 2013, the R-CNN framework [12] (Regions with CNN features) was designed by Girshick *et al.*, replacing the old and inefficient sliding window detection technique and making a breakthrough in object detection. [19]

Although accurate, object detectors based on the R-CNN architecture (including fast R-CNN and faster R-CNN) are generally computationally expensive compared to one-stage object detectors and won't be discussed here in detail.

2.3.2 One-Stage Object Detectors

[[čerpať z Li2022]] [[SSD a celá história YOLO?]]

TODO SSD

YOLO

[[Čo všetko tu popísať? Všetky verzie YOLO? Alebo iba YOLOv8 + background o YOLO celkovo?]]

2.3.3 Evaluation Metrics

Evaluation metrics are crucial for assessing the performance of different object detection models, enabling comparison between different architectures and tracking improvements during training. In this subsection, we provide a brief explanation and background of the Mean Average Precision (mAP) metric, which will be used to evaluate the detectors trained in this project.

[2] Image classification is the task of classifying an image into a class category – for example recognizing whether an image contains a cat or a dog.

Average Precision

The Average Precision (AP) is a widely used metric that calculates the performance of an object detector by measuring the area under the precision-recall curve.

Precision is a measure of the proportion of true positive detections out of all detections (true positives and false positives), while recall measures the proportion of true positive detections out of all ground truth[3] objects (true positives and false negatives) in the dataset.

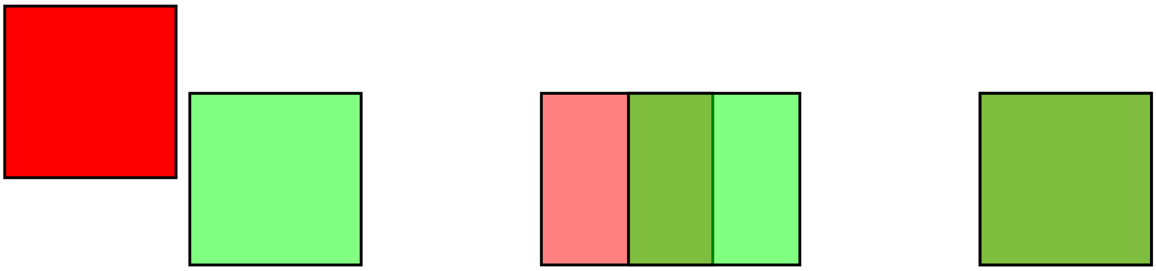
To compute the AP, the precision and recall values are calculated at different confidence score[4] thresholds. By plotting precision against recall for these various thresholds, we obtain the precision-recall curve. The AP value is then equal to the area under the precision-recall curve and ranges from 0 to 1, where AP of 1 indicates perfect precision and recall at all thresholds.

[[cite mAP v7labs]]

Mean Average Precision

To evaluate multi-class object detectors, Mean Average Precision (mAP) is used instead of the previously explained Average Precision. Computing the mAP involves finding the AP for each class and calculating the average over the number of classes.

However, for object detection tasks, precision is typically calculated with different thresholds of the Intersection over Union (IoU) metric. The IoU metric measures the overlap between two bounding boxes – between the model's prediction and the ground truth bounding box, and represents the quality of the alignment between the two boxes. Calculating the IoU simply means dividing the intersection area of the two bounding boxes over their union. When calculating precision, the IoU metric is used to determine whether a predicted bounding box should be considered a true positive (IoU is higher than the defined IoU threshold) or a false positive (IoU is lower than the threshold). For visualization, see Figure 2.2.



(a) $IoU = 0$

(b) $IoU \approx 0.333$

(c) $IoU = 1$

Figure 2.2: Visualization of the Intersection over Union calculation.

In this paper, we consider the COCO mAP specification and calculate the mAP as an average of AP calculated for all classes and over 10 IoU thresholds ranging from 0.50 to (and including) 0.95 with step 0.05. Similarly, values $\text{mAP}^{\text{IoU:0.50}}$ and $\text{mAP}^{\text{IoU:0.75}}$ denote mAP calculated with IoU thresholds equal to 0.50 and 0.75 respectively. Additionally, $\text{mAP}^{\text{small}}$ is only calculated for objects of area smaller than 32^2 pixels, $\text{mAP}^{\text{medium}}$ for

[3] Ground truth refers to the actual, true data used as a reference for comparison with a model's predictions.

[4] Confidence score is a number output from a detector for each detection stating the model's confidence that the detection is correct.

objects larger than 32^2 px but smaller than 96^2 px and finally, $\text{mAP}^{\text{large}}$ for objects larger than 96^2 px.

[[cite mAP v7labs]]

2.3.4 Transfer Learning

[[Nedať to niekde inde? Kam?]] [[Alebo treba to vôbec?]]

2.4 Network Optimization and Compression

[[pruning, quantization, knowledge distillation,...]]

As CNNs have grown deeper and more complex to improve accuracy, the computational and memory requirements have increased significantly. This makes it difficult to deploy CNNs on embedded devices with limited resources. To address this challenge, researchers have developed various model optimization and compression techniques that aim the reduce the computational cost and memory footprint while trying to maintain the model's accuracy. A brief overview of the most significant model optimization and compression techniques is given in this section. However, in this paper, only one of these techniques – weight quantization will be used.

2.4.1 Network Pruning

Typically, there are many parameters in a CNN which were not utilized during the training phase and do not contribute to the network's performance. The pruning model optimization technique aims to simply remove these redundant parameters from the model while maintaining accuracy. Of course, more aggressive pruning can be performed, removing even more parameters, including useful but still less important ones, although sacrificing some accuracy.

Various pruning techniques exist, including weight pruning (removal of a single weight), neuron pruning (removal of an entire neuron and its connections), filter pruning and layer pruning.

Although the pruning technique was developed to reduce the model's storage requirements, it is also used to reduce its computational requirements.

[[cite https://link.springer.com/article/10.1007/s10462-020-09816-7]]

2.4.2 Knowledge Distillation

Knowledge distillation is a model compression technique in which a smaller, more compact student network is trained to mimic the behavior of a larger, high-performing teacher network (or an ensemble of them) to learn the teacher model's generalization capability. The student network learns from outputs from the teacher network instead of the ground truth labels. For more effective knowledge distillation methods, these outputs often include intermediate feature maps of the teacher network. Generally, the student network cannot achieve accuracy as high as the teacher network, but when performed correctly, it typically achieves higher accuracy than if trained the conventional way.

[[cite 2022 structural KD for object detection]]

[[vymazať alebo rephrase:]] Because this method practically involves training two models, with one being much larger than the other, it requires more training time than training just the student model would. It is however very useful when training and comparing several student networks.

Although this technique was initially considered highly advantageous for addressing the problem proposed in this paper, upon careful analysis, we assumed that the associated complexity would pose significant obstacles, primarily in terms of increased development time. Therefore, this technique was not employed in the current study; however, its exploration is recommended for future work.

2.4.3 Weight Quantization

[[todo [6] nasledujúce tri odstavce]]

In convolutional neural networks, weights and biases are stored as 32-bit floating-point numbers (`fp32`), which provide precision often unnecessary for the CNNs to be accurate. Quantization is the process of reducing the number of bits used to represent these parameters and generally decreases the storage and computational requirements of the network.

Although reducing the precision of the model's parameters decreases its accuracy, the drop in accuracy is typically insignificant when compared to the substantial benefits in storage and computational efficiency gained, which are essential factors when developing a high-performance object detector.

Most commonly, models parameters are quantized to either a 16-bit floating-point (`fp16`) representation or an 8-bit integer (`int8`) representation. Quantizing to `fp16` usually doesn't require any post-quantization steps. However, quantizing to an integer representation, such as `int8`, is a different process. Because of the limitations of the integer representation and the distinction from floating-point representations, weight calibration[5] during the quantization process or even model fine-tuning[6] after it is recommended to maintain the highest possible prediction accuracy. This, of course, only applies to post-training quantization (PTQ) while several other quantization techniques are available – mainly training the model with weights in the desired representation from the beginning (quantization-aware training - QAT). Furthermore, a more advanced technique called the mixed-precision quantization (MPQ) can be used to quantize parameters in a more nuanced manner by assigning different precisions to individual parameters or parameter groups depending on their sensitivity to numerical errors [26].

While the weight quantization technique can significantly decrease a model's computational requirements, it is crucial to note that not all devices are compatible with every parameter representation. Therefore, it's important to verify whether a specific device supports computations with the desired number representations.

[[Toto tu písať? Alebo radšej vymazať? Ak áno, rephrase:]] Most deep learning inference libraries support weight quantization and offer tutorials on how it's done. The MMDeploy model deployment library along with the TensorRT inference (both will be discussed later in this chapter **[[ref?]]**) library makes the quantization process very simple...

2.5 Embedded Platforms for Machine Learning

[[RPI?]] [[K jetpacku ešte verziu pythonu a tak?]]

[5] Weight calibration during weight quantization refers to the process of adjusting the quantized weights to minimize the loss of accuracy that occurs due to the reduction in numerical precision.

[6] Model fine-tuning refers to further training of a model with lower learning rate to refine the model's parameters.

Embedded platforms are a combination of hardware and software components that are designed to perform specific tasks. For object detection, or machine learning in general, these systems are optimized to be power efficient while providing significant processing capabilities for the development, deployment and execution of machine learning algorithms. In this section, we will discuss some of the most popular embedded devices and platforms used for machine learning, including object detection.

Embedded devices typically incorporate custom processors, microcontrollers or specialized accelerators specifically engineered for efficient execution of machine learning tasks, such as graphics processing units (GPUs), which can perform many computations in parallel, tensor processing units (TPUs) and many more.

2.5.1 Google Coral

Designed for TensorFlow Lite [[**footnote**]] models, the Google Coral platform offers an Edge TPU – a low-power, high-performance ASIC (application-specific integrated circuit) enabling on-device machine learning inference [[**footnote inference ak som doteraz nevysvetlil**]]. Coral provides development boards, USB accelerators along with a variety of modules and peripherals for edge AI applications.

2.5.2 Movidius Neural Compute Stick

Intel's Movidius Neural Compute Stick, commonly paired with a popular single-board Raspberry Pi computer, is a small, low-power USB-based hardware accelerator featuring a vision processing unit (VPU) designed to accelerate neural network computations.

2.5.3 NVIDIA Jetson

NVIDIA Jetson is a series of widely-used embedded computing platforms that feature powerful GPU accelerators and ARM-based CPUs while being energy-efficient.

Devices of the NVIDIA Jetson family support the Linux for Tegra (L4T) operating system, a customized Linux distribution designed specifically for the platform's unique capabilities. NVIDIA also offers the JetPack SDK, which includes the CUDA toolkit and the cuDNN library accelerating deep learning tasks on NVIDIA GPUs, the TensorRT library used for optimizing deep learning models to achieve higher inference speeds on NVIDIA GPUs, along with various multimedia and computer vision libraries.

NVIDIA Jetson AGX Xavier

Jetson AGX Xavier is the flagship model in the NVIDIA Jetson family. It is a high-performance, energy-efficient platform designed for more demanding AI workloads. With an integrated NVIDIA Volta GPU with 512 CUDA cores and 64 Tensor cores, an 8-core NVIDIA Carmel ARM CPU and 16 GB of memory, it offers substantial computational capabilities for deploying state-of-the-art real-time object detectors. Its typical power consumption ranges from 20 W to 30 W.

NVIDIA Jetson Xavier NX

The NVIDIA Jetson Xavier NX features an NVIDIA Volta GPU with 384 CUDA cores and 48 Tensor cores, a 6-core NVIDIA Carmel ARM CPU and 8 GB of memory. Compared to

Jetson AGX Xavier, it is more compact and power-efficient, with the typical consumption of 15 W, while of course offering lower performance and memory capacity.

NVIDIA Jetson Nano

The smallest and most popular module from the NVIDIA Jetson family is the Jetson Nano. Equipped with a 128-core NVIDIA Maxwell GPU, a quad-core ARM Cortex-A57 CPU and just 4 GB of memory, it serves as an entry-level, low-cost AI embedded platform, offering lower performance but lower power consumption (ranging from 5 to 10 W) compared to the previously mentioned Jetson boards.

2.6 Vehicle Detection

[[**referencie na iné papiere? Postup? Že treba dataset no nie je ťažké ho získať a také?**]]

2.7 Tools and Libraries

In this section, tools and libraries used in this project are briefly discussed. [[**However, not all**]]

2.7.1 LabelBox annotation app

LabelBox [17] is a web-based application and data labeling platform used for training machine models. It provides an easy-to-use interface and a suite of tools to manage and annotate datasets efficiently. With the option of importing data, we have used the app to reannotate one of the datasets.

2.7.2 PyTorch

PyTorch [23] is a popular open-source, Python-based machine learning library designed to provide flexibility, ease of use, and high performance for deep learning applications. It offers a rich ecosystem of tools and libraries for various tasks, including computer vision. With support for GPU acceleration using NVIDIA's CUDA platform, PyTorch enables fast and efficient computation of large models, making it an ideal choice for high-performance deep learning tasks.

2.7.3 MMDetection Library

MMDetection [8] is a popular open-source deep learning toolbox for computer vision tasks, including object detection. Developed by the Multimedia Laboratory at the Chinese University of Hong Kong [[**cite?**]] (OpenMMLab), MMDetection provides a flexible, extensible and modular framework that aims to simplify the process of training and deploying state-of-the-art models for various computer vision tasks, and provides a rich set of tools. Some of the key features of the library include:

- It is based on the PyTorch machine learning library.

- It builds upon and uses other open-source libraries from the OpenMMLab project, such as MMCV[7] and MMEngine[8].

- Library's modular design allows for easy customization and extension of the codebase.

- It includes pre-trained models and their configurations which makes training and comparing different models fast and simple, while making it possible to use the transfer learning technique[[**footnote ak som nevysvetlil transfer learning v teórii**]] which significantly speeds up the process.

2.7.4 MMYOLO Library

Although the MMDetection library doesn't include the latest detectors of the YOLO family, such as the YOLOv8 detector developed by Ultralytics, which will be used in this project, the OpenMMLab project features a different library called MMYOLO [9], an extension of the MMDetection library, which addresses this issue and focuses solely on detectors of the YOLO family. It contains implementations of YOLO-specific components, such as the CSPDarknet and PANet backbone networks, and YOLO-specific training techniques, like data augmentations or loss functions.

2.7.5 MMDeploy Library

The MMDeploy library [7], which is also a part of the OpenMMLab project, offers useful tools for deploying OpenMMLab models to a wide range of platforms and devices. It enables the conversion of PyTorch models trained with MMDetection or MMYOLO into backend models for execution on target devices. MMDeploy supports various backends, including ONNX, TensorRT, OpenVino, TorchScript and numerous others. In addition to streamlining the deployment process, the library also optimizes the converted models for their target platforms.

2.7.6 ONNX and ONNXRuntime

ONNX (Open Neural Network Exchange) [3] is an open-source project aimed at creating a consistent format for deep learning models. It was initially developed by Facebook and Microsoft, but several other companies and organizations joined later on. The primary goal of ONNX is to enable developers and researchers to easily switch between different machine learning frameworks without having to worry about model compatibility. ONNX defines a standard representation for neural network models, making it possible to train a model in one framework and use it for inference in another.

Additionally, ONNX provides a set of tools and libraries, such as ONNX Runtime [10], which is a high-performance inference engine for ONNX models.

2.7.7 TensorRT

TensorRT [[**cite ale nenašiel som citáciu**]] is a high-performance deep learning inference optimizer and runtime library developed by NVIDIA. It is designed to accelerate the

[7] MMCV is a library for computer vision research including building blocks for convolutional neural networks, tools for image processing, transformations and much more

[8] MMEngine library serves as the training engine for all OpenMMLab codebases, supporting hundreds of algorithms frequently used in deep learning

deployment and inference of models on NVIDIA GPUs for various applications including computer vision.

In addition to optimizing the target models for improved inference performance and reduced memory footprint, TensorRT also supports multiple precision modes, including `fp32`, `fp16` and `int8`, allowing developers to choose the best balance between accuracy and performance.

Chapter 3

Datasets

Big and high-quality datasets are very important when training a CNN-based detector. In this section, used datasets are listed and analyzed. First, the criteria for recognizing an appropriate dataset for our task are explained. Each chosen dataset is then analyzed individually. Finally, a summary of all used datasets is provided.

3.1 Dataset Criteria

Here, we briefly explain the most important criteria for selecting datasets to be used in a project like this.

Camera Angle

Since we're building a detector for a camera mounted on infrastructure, it is recommended to use datasets containing surveillance-type images. Datasets containing only images taken from, for example, a car dashboard camera, were therefore disregarded.

Classes

If we don't want to re-label the dataset manually, its classes must be mappable to *our* classes. In this work, 8 object classes are considered:

- Bicycle
- Motorcycle (any two-wheeled motorized vehicle)
- Passenger Car
- Transporter (or a van, pick-up truck, etc.)
- Bus (including a minibus)
- Truck
- Trailer
- Unknown

Many available datasets didn't annotate some of these classes or aggregated some of them into one and were therefore ignored.

Diversity of Images

For the trained detector to generalize well, it's important for a dataset to contain images with different camera angles, lighting conditions, weather, etc. 60 FPS continuous video does not bring much of an advantage.

Dataset Quality

We observed that many datasets contained incorrect annotations or classifications. It is important to check the dataset and either fix faulty annotations, ignore incorrectly annotated images or even disregard the whole dataset.

3.2 Individual Datasets

This section individually analyzes all datasets used in this work. Table 3.1 compares these datasets on a higher level for an overview and shows how the datasets were split for training, validation and testing.

3.2.1 UA-DETRAC

The DETRAC dataset [28] provided by the University at Albany is the biggest and the most important dataset for this work, originally containing 1 274 055 annotations of 8250 vehicles in 138 252 images. The dataset is provided as frames from 100 video sequences of 25 fps with the resolution being 960 × 540 pixels. The length of these sequences varies, but is usually between 30 seconds and 90 seconds. The video is of a surveillance type and almost all sequences have a unique point of view, usually from a bridge. Many sequences were recorded in rain or at night and there is no lens flare from cars' headlights. One of the images from this dataset is shown in Figure 3.1.

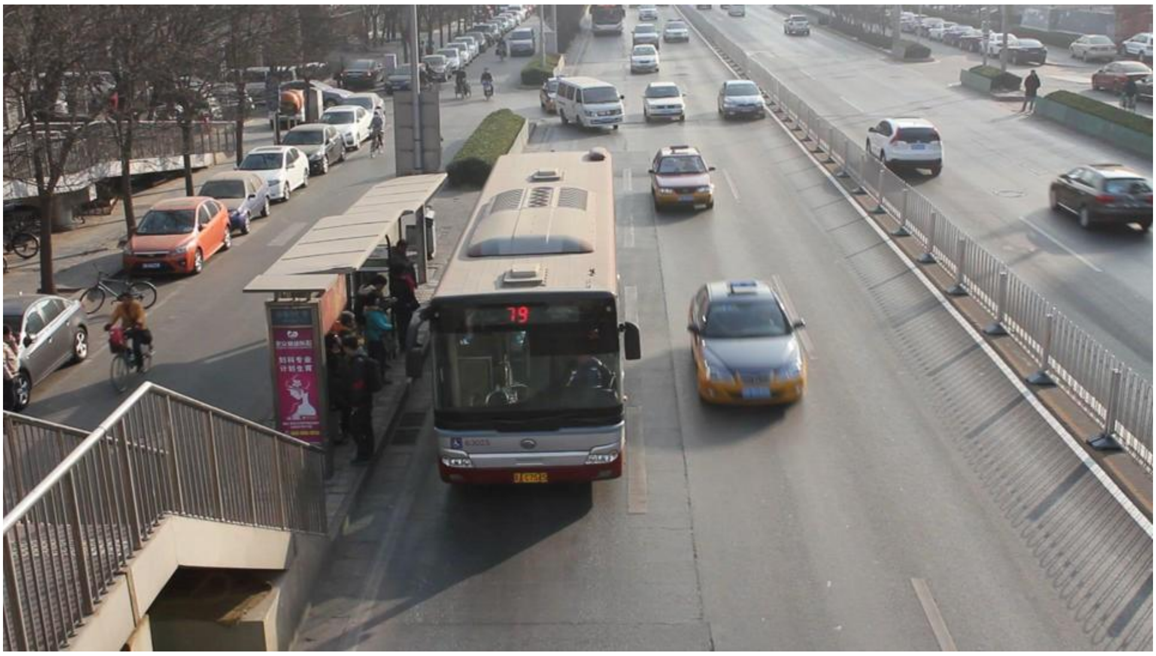


Figure 3.1: Example image from the UA-DETRAC dataset.

However, there are several issues with this dataset:

- Bicycles, motorcycles and a few vehicles that cannot be classified are not annotated at all
- Bounding boxes are often loose and do not fit tightly to the objects
- Vehicles near the edge of the frame, although fully visible, sometimes lack labels
- In several sequences, vehicles are tracked and annotated even on frames on which they are fully occluded (by other vehicles or infrastructure)
- Vehicle annotations are inconsistent in relation to masks, as some are labeled even when masked, while others are left unlabeled even when already fully visible outside of a mask. This is likely due to the camera being hand-held while the mask is static throughout the sequence

These problems were fixed by importing the dataset to the LabelBox labeling application, adjusting the masks (while also making them dynamic), annotating or masking bicycles, motorcycles and „unknown“ vehicles and finally, carefully repairing individual annotations if needed. Many sequences were hectic or were annotated so poorly that reannotating would be too time-consuming, so only 71 of 100 sequences were reannotated.

The reannotated dataset contains 733 833 annotations in 99 771 images and these classes (the dataset contains no trailers):

- Bicycle
- Motorcycle
- Car
- Bus
- Van - mapped to *transporter*
- Others - mapped to *truck*
- Unknown

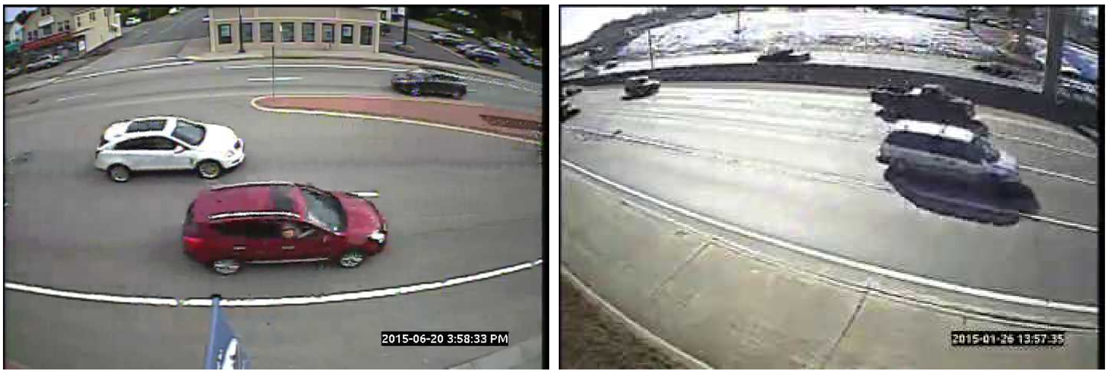
3.2.2 Miovision Traffic Camera Dataset

The MIO-TCD (Miovision traffic camera dataset) [1] is another huge and very important dataset. Images are taken at different times of day by thousands of traffic cameras in Canada and the United States. Roughly 79% of all images are of resolution 720 × 480 pixels, but the image quality seems to be lower. The rest is of resolution 342 × 228 pixels. Example images of both resolutions are shown in Figure 3.2.

The dataset consists of two parts: *Classification dataset* and *Localization dataset*. Only the *Train* subset of the *Classification* part is used here because the *Test* subset is not annotated.

The part used contains 351 549 objects of these classes:

- Pedestrian - ignored
- Bicycle



(a) Resolution 720 × 480

(b) Resolution 342 × 228

Figure 3.2: Example images from the Miovision dataset.

- Motorcycle
- Car
- Pickup truck - mapped to *transporter*
- Work van - mapped to *transporter*
- Bus
- Articulated truck - mapped to *truck*bridge. Many sequences were recorded in rain or at night and there is no lens flare from cars' headlights.
- Single unit truck - mapped to *truck*
- Non-motorized vehicle - mapped to *trailer*
- Motorized vehicle - mapped to *unknown*

The processed dataset (without pedestrian annotations) contains 344 416 objects in 110 000 images.

The images are of low quality, there aren't many bicycles and motorcycles, and more pictures with different weather and lighting conditions are needed.

3.2.3 AAU RainSnow Traffic Surveillance Dataset

Another important dataset is the *AAU RainSnow* dataset [2]. The authors mounted two synchronized (one RGB and one thermal) cameras on street lamps at seven different Danish intersections to take 5-minute long videos at different lighting and weather conditions - night and day, rain and snow. They then extracted 2200 frames from the videos and annotated them on a pixel-level. Several different types of masks were also created and included in the dataset.

In our work, we only use the annotated frames from the RGB camera, containing 13 297 annotations in 2200 frames (before processing) of resolution 640×480 pixels. See an example in Figure 3.3.

The dataset uses these 6 classes:

- Pedestrian

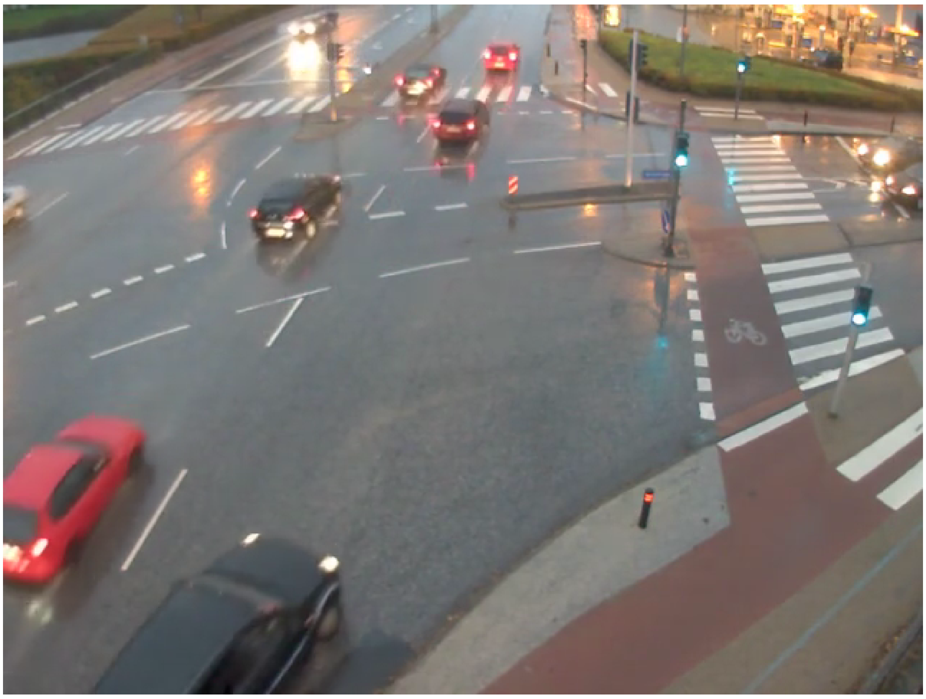


Figure 3.3: Example image from the AAU RainSnow dataset.

- Bicycle
- Motorbike
- Car
- Bus
- Truck

This introduces a problem - vehicles of our internal class *transporter* (van) don't have their own class in the dataset, but are classified as trucks. However, the dataset is small and it is impossible to perfectly divide transporters and trucks into two classes as there are many different models between which a line cannot be drawn. Ignoring this issue should therefore not cause any problems.

Several other minor problems were found when processing this dataset:

- Frames in groups `Egensevej-1`, `Egensevej-3` and `Egensevej-5` are hardly usable because of the low-quality camera and challenging weather and lighting conditions, so they were dismissed
- Some frames had bounding boxes over the whole frame - this is most certainly an annotation error. These labels were ignored as well
- The mask for `Hadsundvej` intersection didn't fully cover the area that should be ignored. This was fixed by simply editing the mask

After processing, the dataset contains 10 545 objects in 1899 images.

3.2.4 Multi-View Traffic Intersection Dataset

For the MTID dataset, the authors [15] recorded one intersection from two points of view at 30 fps - one camera was mounted on existing infrastructure and one was attached to a hovering drone. The dataset contains 65 299 annotated objects in 5776 frames (equal share of frames for both cameras). An example from this dataset can be seen in Figure 3.4.

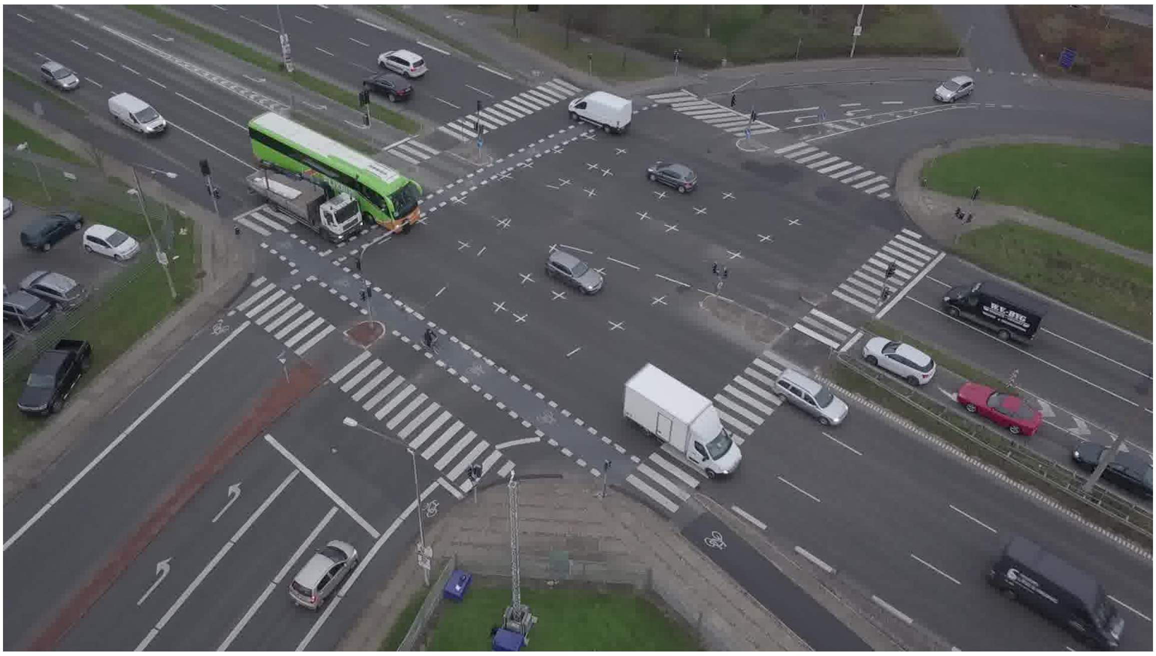


Figure 3.4: Example image from the Multi-View Traffic Intersection dataset.

All annotated objects fall into one of four classes:

- Bicycle
- Car
- Bus
- Lorry - mapped to *truck*

This, at first, might not seem like enough, but a closer inspection of the annotated frames reveals that there are no pedestrians or motorcycles. However, similarly to the AAU dataset in subsection 3.2.3, there are transporters in the frames that are classified as trucks. Again, this issue is simply ignored.

When processing this dataset, two other problems were encountered:

- Vehicles that are not on the road are not annotated, so they have to be masked out. This is not as easy for the drone video because the camera is moving, but it is still simple enough
- Many frames of the drone footage lack some or all labels and have to be ignored. Ranges of images numbers which are ignored: $[1, 31]$, $[659, 659]$, $[1001, 1318]$ and $[3301, 3327]$

The processed dataset contains 64 979 objects in 5399 frames.

3.2.5 Night and Day Instance Segmented Park Dataset

Another useful dataset is the *NDISPark* [20], which contains images of parked vehicles taken by a camera mounted on infrastructure. Although the annotated part of the dataset only contains 142 frames after processing, there are 3302 objects annotated in total. This still makes it a tiny dataset, but it provides images of vehicles from many different points of view and also contains many occluded vehicles. See Figure 3.5 for an example image from this dataset. Additionally, all frames are 2400 px in width and within 908 px and 1808 px in height.

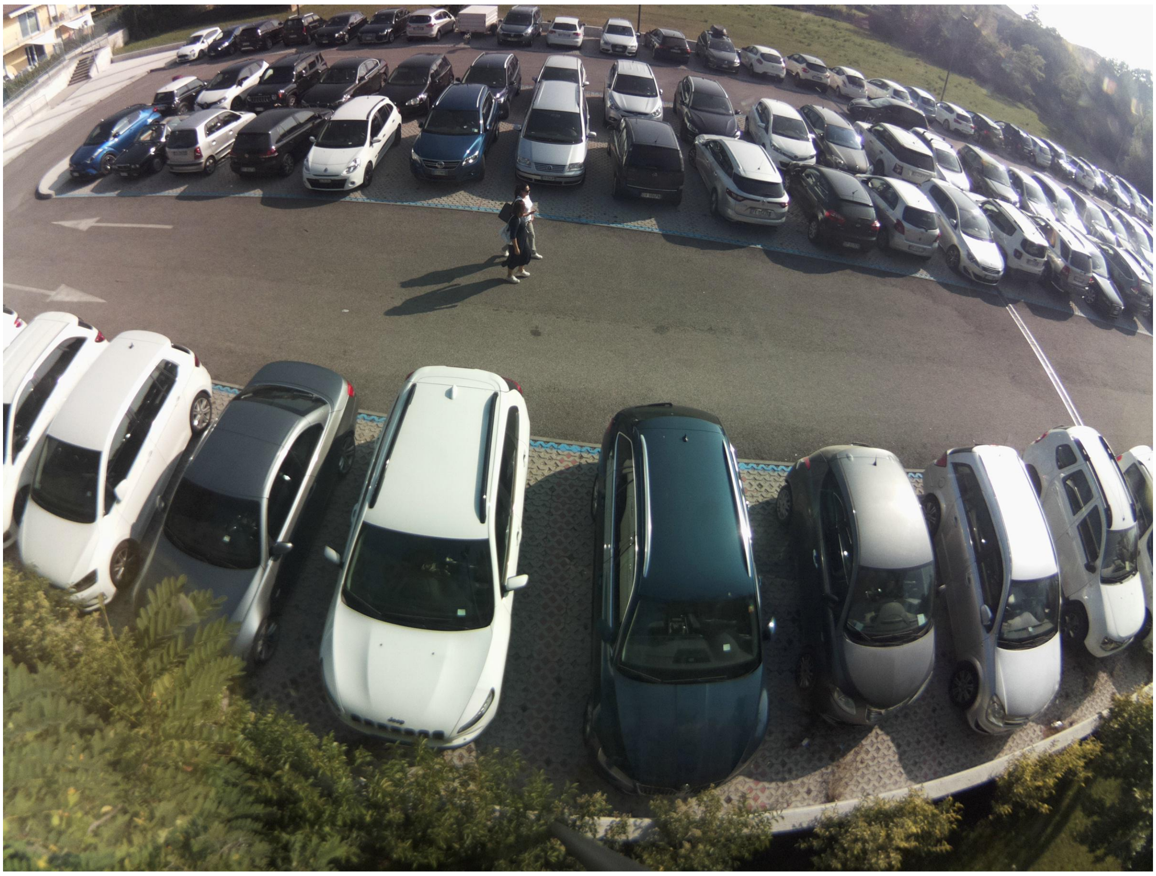


Figure 3.5: Example image from the NDISPark dataset.

This dataset does not contain any classifications, but luckily, it only contains cars, *transporters* and a few unattached car trailers. All *transporters* and *trailers* were manually classified.

3.2.6 VisDrone Dataset

The VisDrone dataset [30] is very different from all the previous datasets since it doesn't just contain traffic surveillance images. Images are taken by a camera mounted on a drone, from many different points of view. After processing, there are 47 720 annotated objects in 1610 frames. An example is displayed in Figure 3.6.

This dataset might be a helpful addition to our datasets as it contains useful negative images (many annotated people and new points of view) and it often captures vehicles from a bid's-eye view.

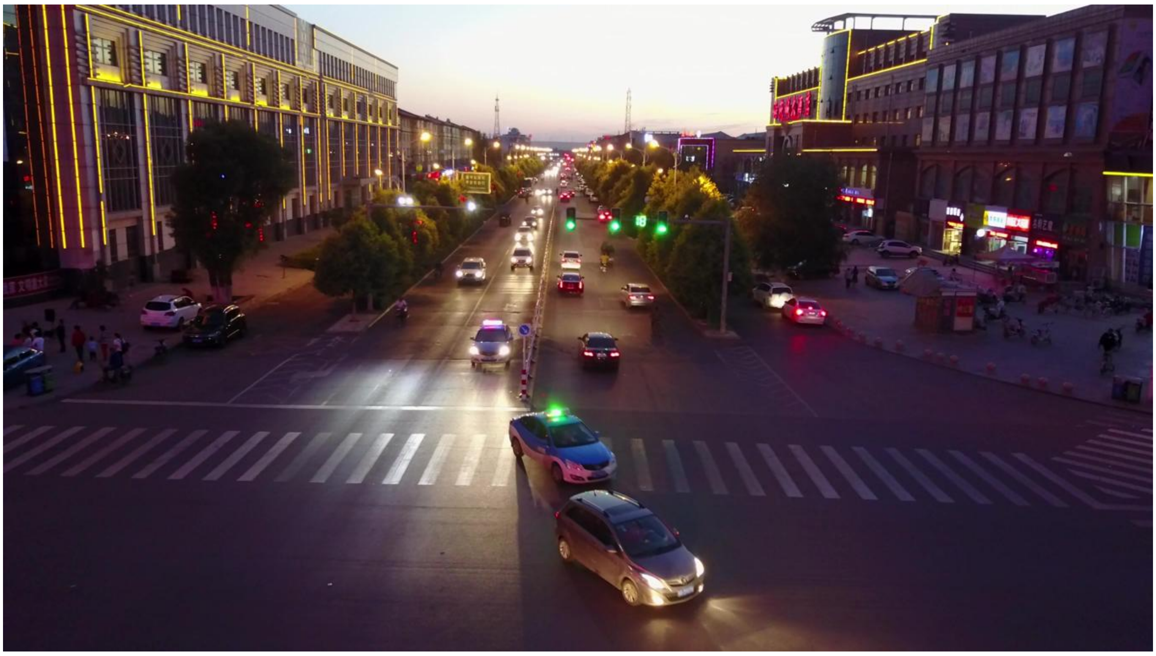


Figure 3.6: Example image from the VisDrone dataset.

Additionally, objects in this dataset are classified into 12 classes, which can be easily mapped to our 8 *internal* classes.

3.3 Dataset processing

[[v akom formáte sú datasety, do akého ich konvertujem, aké súbory vytváram a tak. Asi pokojne vcelku podrobne]]

Before training, all datasets used in this project must be converted to a unified format, object classes need to be mapped to be the same in each dataset, some datasets include images with regions shat should be masked out and certain subsets or images of some datasets need to be ignored.

For this, a python script was developed for each dataset. It first loads the annotations from the original format - COCO **[[ref?]]** for AAU RainSnow, MTID and NDISPark, XML for DETRAC and different CSV formats for MIO-TCD and VisDrone. The script then maps the classes to ones shown in **[[ref]]** and if needed, removes the ignored subsets or images before saving the labels in a COCO format. The processing script for NDISPark dataset also corrects the object classes before saving, as it was not done on the original ground-truth **[[footnote explain?]]** file (class corrections are defined in the script itself).

MMDetection's middle format was considered as a more efficient alternative to the COCO format, but the COCO format is more popular and is supported by most relevant application, while also being human-readable (MMDetection's middle format is saved as a pickle **[[footnote?]]** file), and most importantly, MMYOLO and the newest version of MMDetection at the time of writing this paper (v3.0.0) does not seem to fully support datasets in the middle format.

Several other python processing scripts were developed, to apply masks, combine all ground-truth files into one and split the combined ground-truth file into train, validation and test subsets. Additionally, scripts to review datasets manually were created - one to visualize a dataset by simply adding bounding boxes (with class labels) to the images and one converts the visualized images to video (or videos).

A few more scripts were written, of which two are worth mentioning - one for uploading the DETRAC dataset to LabelBox for reannotation and one for downloading the reannotated labels.

3.4 Summary of Datasets

In Table 3.1, we compare used datasets on a higher level and show the number of images and instances contained with additional comments. It is clear that the DETRAC subsection 3.2.1 dataset amounts for most of our data and might have been enough by itself, but to make this project as successful as possible, every available useful dataset should be used. The images from the selected datasets feature a great variety of lighting and weather conditions, points of view, object scales and other relevant factors. Additionally, in Table 3.2 we show the number of annotated object instances per class in all used datasets combined. **[[Povedať niečo k class inbalance?]]**

Dataset tag	# images	# instances	Comments
DETRAC	99 771	733 833	Large Continuous video High-quality camera Different lighting conditions
MIO-TCD	110 000	344 416	Large Low-quality images
AAU	1899	10 545	Small Different weather conditions
MTID	5399	64 979	Small Continuous video
NDISPark	142	3302	Small Occlusions
VisDrone	1610	47 720	Small Negative images New points of view
Total	218 821	1 204 795	-

Table 3.1: High-level comparison of used datasets.

3.5 Training, Validation and Testing dataset split

We chose to only include the Miovision and DETRAC-UA datasets for validation and testing, because they represent data from a typical traffic surveillance camera the best.

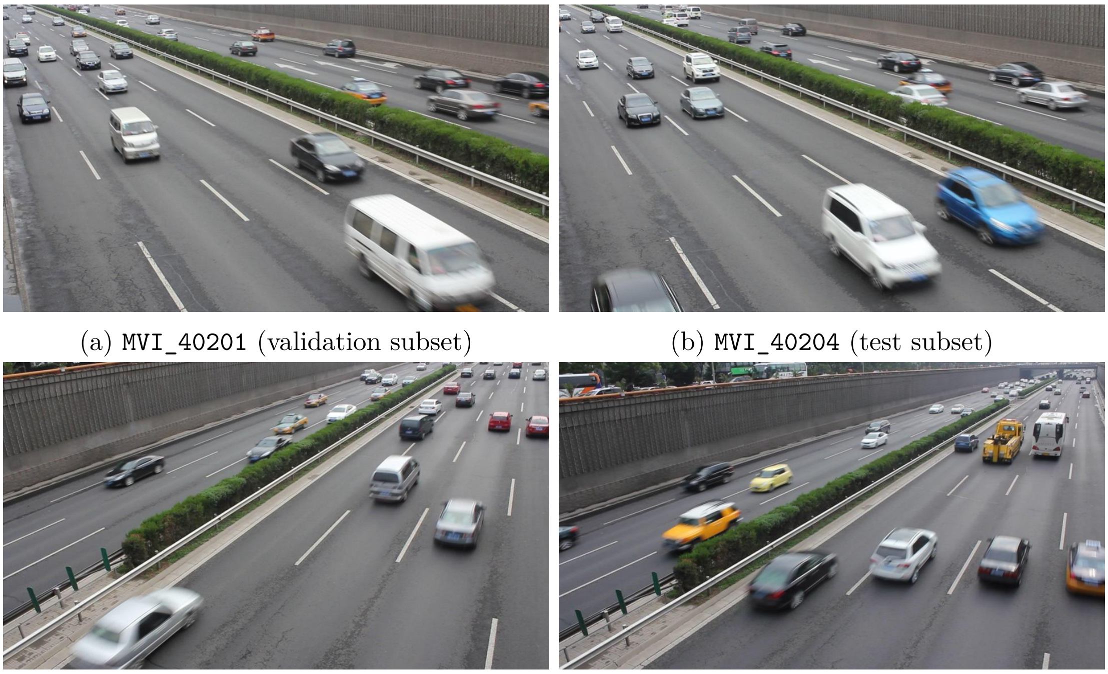
Because only the train subset of the Miovision dataset was used (only this subset contained annotations), the images used for validation and testing are chosen randomly. This, however, should not be a problem since the dataset contains many different camera angles and each image is very unique.

Class	# instances
Bicycle	14 036
Motorcycle	17 187
Passenger car	916 317
Transporter	123 585
Bus	64 529
Truck	38 069
Trailer	2360
Unknown	28 712
Total	1 204 795

Table 3.2: Numbers of class instances in all datasets combined.

From the DETRAC-UA dataset, two sequences were selected for the validation subset (`MVI_40201` and `MVI_40244`) and two for the test subset (`MVI_40204` and `MVI_40243`). The chosen sequences are very different from the ones in the training subset, which is important for the evaluations to be accurate. However, the `MVI_40201` sequence is recorded from the same angle as `MVI_40204` and the same applies to sequences `MVI_40244` and `MVI_40243`, so the validation and test subsets are alike, but of course, contain different data. Example images are shown in Figure 3.7.

The validation subset contains a total of 36 969 objects on 7770 images, of which 5500 images are from the Miovision dataset and 2270 from DETRAC-UA. Similarly, the test subset contains a total of 50 357 objects on 7990 images – 5500 images from the Miovision dataset and 2490 from DETRAC-UA.



(a) `MVI_40201` (validation subset)

(b) `MVI_40204` (test subset)

(c) `MVI_40241` (validation subset)

(d) `MVI_40243` (test subset)

Figure 3.7: Example images from sequences from the DETRAC-UA dataset used for the validation and testing subsets.

Chapter 4

Object Detection Models

[[Nedať do k experimentom?]]

In this chapter, the reader will find an overview of used object detection models, their training configurations and information about the training process.

4.1 Model Architectures

This section introduces all object detection architectures evaluated in this paper. The main focus is on the YOLOv8 object detector, which is currently the state-of-the-art real-time object detector and offers different model sizes for different applications.

Along with standard model sizes, YOLOv8-medium, YOLOv8-small and YOLOv8-nano, several others were trained and evaluated in this work: YOLOv8-pico and YOLOv8-femto model versions, which are simply smaller versions of the same YOLOv8 model. Finally, a YOLOv8-large model with the CSP Darknet backbone replaced by a popular MobileNetV2 backbone is introduced (hereafter referred to as YOLOv8 MobileNetV2). **[[indices used or more about it]]**

Normally, a square shape is used as a model input, but standard traffic cameras output a video of a rectangular shape, usually with the aspect ratio being 16:9. To optimize the inference speed, a rectangular input shape is used for all trained detectors. The aim was to use aspect ratios as close to 16:9 as possible, but the widths and heights of the input shape have to be multiples of 32, so some input shapes are further from it than others.

For a summary of all used model architectures, their sizes, input shapes and the amounts of their floating point operations (FLOPS) and parameters, see Table 4.1. Numbers of floating point operations and parameters were calculated using MMDetection's analysis script `get_flops.py`. Although the script calculates the output using input shape 720×480, simply multiplying the output by the difference between the input shapes provides an accurate result. This can be explained by the following equation:

$$N_{new} = N_{720\times480} \times \frac{new\_width \times new\_height}{720 \times 480} \tag{4.1}$$

[[napísať tu toho viac?]]

4.2 Model Configurations

This section provides an overview of configurations used to train the vehicle detectors. A configuration of an object detection model contains a set of parameters that define the

Architecture	Deepen factor	Widen factor	Input shape	Number of floating point operations	Number of parameters
YOLOv8-medium	0.67	0.75	640 × 384	7.85 G	18.39 M
YOLOv8-small	0.33	0.5	640 × 384	2.63 G	3.73 M
YOLOv8 MobileNetV2	1	1	512 × 288	0.560 G	2.133 M
YOLOv8-nano	0.33	0.25	640 × 384	0.758 G	1.688 M
			512 × 288	0.455 G	1.013 M
			448 × 256	0.354 G	0.788 M
YOLOv8-pico	0.166	0.125	512 × 288	0.1510 G	0.3067 M
			448 × 256	0.1175 G	0.2386 M
			384 × 224	0.0881 G	0.1790 M
YOLOv8-femto	0.166	0.0625	512 × 288	0.0780 G	0.1288 M
			448 × 256	0.0607 G	0.1002 M
			384 × 224	0.0455 G	0.0751 M
			352 × 192	0.0358 G	0.0591 M

Table 4.1: Summary of different YOLOv8 model architectures used and comparison of amounts of their floating point operations and parameter with various model input shapes.

model's behavior and the training, validation and testing pipelines. These parameters can have a significant impact on the model's speed and accuracy and tuning them is essential to achieve good results.

Most of the YOLOv8 parameters are left unchanged from the default configuration of YOLOv8-m[1], like:

Optimizer: Stochastic Gradient Descent with momentum 0.937 and weight decay 0.0005

Parameter scheduler: Linear `YOLOv5ParamScheduler` with learning rate factor of 0.01

However, many parameters related to datasets and augmentations were adjusted and will be explained in the next subsections. Apart from those, the only relevant parameter that was changed is the batch size, which was set to the highest possible for every trained model. For the smallest one, YOLOv8-femto with 352 × 192 input shape, the largest batch size of 760 was used. Because training batch sizes above 128 usually result in lower model precision [27], models with large batch sizes were trained for 500 epochs instead of the default 300 epochs. Along with other model-specific training parameters, these settings can be found in Table 4.2

4.2.1 Dataset Wrappers

In the MMYOLO (and the MMDetection) model configurations, datasets to use for training, validation and testing are specified using a dataset wrapper, from which a *dataloader* (an object internally representing a dataset) is created. Because the datasets used in this project were in the COCO format, the *YOLOv5CocoDataset* wrapper was used for each of the 6 used datasets.

To compensate for some datasets being smaller than others while being important and of high quality, a dataset wrapper *RepeatDataset* is used, which makes the underlying dataset n-times more frequent when training. All datasets are finally concatenated into one by the *ConcatDataset* wrapper. The repetition factors of individual datasets are shown in Table 4.3.

[1] The default YOLOv8-m configuration used, `yolov8_m_syncbn_fast_8xb16-500e_coco.py`, can be found at https://github.com/open-mmlab/mmyolo/tree/v0.4.0/configs/yolov8

Architecture	Input shape	Learning rate	Batch size	Epochs	Warmup epochs
YOLOv8-medium	640 × 384	0.00125	46	300	5
YOLOv8-small	640 × 384	0.00125	76	300	5
YOLOv8 MobileNetV2	512 × 288	0.01	96	300	5
YOLOv8-nano	640 × 384	0.00125	112	300	5
	512 × 288	0.00125	192	300	5
	448 × 256	0.00125	256	300	5
YOLOv8-pico	512 × 288	0.01	224	500	10
	448 × 256	0.01	384	500	10
	384 × 224	0.01	512	500	10
YOLOv8-femto	512 × 288	0.01	380	500	10
	448 × 256	0.01	420	500	10
	384 × 224	0.01	640	500	10
	352 × 192	0.01	760	500	10

Table 4.2: Training configurations for individual models, including learning rate, batch size, number of training epochs and number of warmup epochs.

Dataset name	# images	Repetition factor	# images after over-sampling
DETRAC	99 771	1	99 771
MIO-TCD	110 000	1	110 000
AAU RainSnow	1899	3	5697
MTID	5399	5	26 995
NDISPark	142	25	3550
VisDrone	1610	4	6440

Table 4.3: Repetition factors for each dataset used when training.

4.2.2 Training Augmentation Pipeline

In this subsection, the augmentation pipeline used when training the chosen models is explained. Data augmentations are very important when training an object detector, especially for detectors from the YOLO family. An augmentation in this context refers to the process of applying various transformations to an image to artificially increase the size and diversity of the training dataset, which helps prevent overfitting and improves the generalization ability of the trained model. To a convolutional neural network, even a tiny rotation, translation, image flip, noise or color distortion makes an input image appear to be something completely different, so applying these transformations randomly to the input images is crucial to train a robust model. Following are the transformations in the main training augmentation pipeline:

Resize

First, the image is resized to fit in the model's input size while, of course, keeping the aspect ratio unchanged.

Pad

If the aspect ratio of the input image is not the same as the model's input, extra pixels around the input image must be added to adapt to the model input's aspect ratio. The **color value** of the padded pixels is set to RGB(114, 114, 114).

Random Affine

The *YOLOv5RandomAffine* applies affine transformations to the image, while randomly selecting the values from configured ranges. Parameters:
Maximum translation ratio: 0.05
Maximum rotation degree: 5
Maximum shear degree: 3
Scaling ratio is set individually for each dataset as shown in Table 4.4.

Dataset name	Minimum scaling ratio	Maximum scaling ratio
DETRAC	0.8	1.0
MIO-TCD	1.0	1.1
AAU RainSnow	0.9	1.1
MTID	0.9	2.0
NDISPark	0.9	1.5
VisDrone	1.5	2.5

Table 4.4: Image scaling ratios in *RandomAffine* transformation for each dataset.

The **color value for padding** around the tranformed image (if needed) is set to RGB(114, 114, 114) to be the same as in the *Pad* 4.2.2 transformation.

Cut-Out

The *CutOut* transformations randomly selects regions of the image and fills them by a single color. Again, parameters of this transformation are set individually for each dataset and are shown in Table 4.5.

Dataset name	Number of regions	Size of a single region
DETRAC	6	22 × 22 px
MIO-TCD	4	26 × 26 px
AAU RainSnow	8	10 × 10 px
NDISPark	12	20 × 20 px
MTID	12	10 × 10 px
VisDrone	20	8 × 8 px

Table 4.5: *CutOut* transformation parameters for each dataset.

The **fill color value** is again set to RGB(114, 114, 114), same as for padding in the previous transformations.

Custom Cut-Out

A custom cut-out transformation was developed, similar to *CutOut* used in the previous step. Here, the regions are selected within each bounding box with a certain probability, rather than being chosen randomly within the image. Also, the region size is specified as a range of areas in relation to the bounding box area - if the upper value is 10% and a bounding box area is 100 pixels, the maximum region area can be 10 pixels.

With the boolean option *random_pixels* toggled, the color of each pixel of a cut-out region is generated randomly instead of filling it with a pre-defined color. However, it was found to have no effect after testing.

The **probability** of a region being dropped from each bounding box is set to 0.5 and the **region area** is set to be randomly selected from interval $[5\%, 35\%]$.

Albumentations

Albumentations[5] is a popular open-source library for data augmentation. The MMYOLO **[[cite or something?]]** library provides the option to use these transformations in the data augmentation pipeline. Settings are left unchanged from the original **[[ref or something?]]** *YOLOv8-m* configuration:
Blur probability: 0.01
Median blur probability: 0.01
Grayscale probability: 0.01
CLAHE probability: 0.01

HSV Random Augmentations

The *YOLOv5HSVRandomAug* simply adjusts the hue, saturation and value of the image randomly.

Random Flip

With **probability** of 0.5, the image is horizontally flipped using the *RandomFlip* augmentation.

Photometric Distortion

The *PhotoMetricDistortion* augmentation distorts an image sequentially, while each transformation is applied with a probability of 0.5. It modifies the brightness, contrast, converts color from BGR to HSV, modifies the saturation, hue, converts from BGR to HSV, modifies the contrast and finally, randomly swaps the color channels.

Filter Annotations

As the last step in the pipeline, *FilterAnnotations* is called to remove bounding boxes with **width or height** lower than 8×8 pixels.

Fine-Tuning Augmentation Pipeline

Originally, MMYOLO's YOLOv8 models are configured to switch to a simplified augmentation pipeline for the last 10 training epochs. This model fine-tuning strategy is kept and in the augmentation pipeline for the last 10 epochs, cut-out augmentations are omitted and affine transformations are changed so that no rotation, translation or shear is applied to a sample.

However, this setting seemed to cause a consistent decrease in the validation Mean Average Precision (mAP) metric during the fine-tuning phase (last 10 epochs).

[[graf validačného mAP?]] [[že ktorú epochu som vybral pre každý model??]]

Chapter 5

Experiments

5.1 Devices Used in Experiments

[[písať technické parametre o MX150, i7 a ARM niekde v teórii?]]

While the main focus is on NVIDIA Jetson embedded devices, which were designed specifically for tasks like object detection, tests were run on several other devices for comparison of both ends of the performance gauge. In this section, details about each device that the models were evaluated on are provided, including details about the software and device configurations.

5.1.1 NVIDIA Jetson Platforms

Technical details of NVIDIA Jetson embedded platforms were discussed in subsection 2.5.3 and software versions will be shown later in this section. However, one more important detail to note before reading about the experiments is the used power plan. All used Jetson devices feature several power plans to choose from to adjust the performance and power consumption for a specific task. For all experiments, these power plans were chosen:
NVIDIA Jetson AGX Xavier: 30 W power plan with 4 out of 8 cores running
NVIDIA Jetson Xavier NX: 20 W power plan with 4 out of 8 cores running
NVIDIA Jetson Nano: 10 W `MAXN` power plan with all 4 cores running

5.1.2 NVIDIA GeForce MX150

To be able to compare inference speeds on these embedded devices to ones on a regular GPU, tests were also done on an NVIDIA GeForce MX150 GPU with 2 GB of memory on a DELL Latitude 5401 laptop. Because of the GPU memory constraint, not all models can be evaluated with all inference batch sizes, as will be discussed later in this chapter. Additionally, we were unable to perform tests on this device using the TensorRT library because some of the packages required could not be installed.

5.1.3 Intel Core i7-9850H

Models were also evaluated on a higher-end laptop CPU, Intel Core i7-9850H with the base frequency 2.6 GHz to demonstrate how the inference speeds of YOLOv8 object detection models differ between a GPU and a CPU.

Typically, the frequency at which a CPU operates is adjusted to accommodate the load and it often spikes up when a computation-hungry process starts. After a while, when the CPU temperature rises above a certain threshold, the CPU frequency has to drop to avoid overheating. This is called dynamic frequency scaling, also known as CPU throttling. To make the performance measurements as accurate as possible, the number of warmup samples when testing is increased from 10 to 100. This means that the inference speed of these samples will not count into the final result.

However, the test results still not might be accurate enough and might depend on the underlying operating system, running applications and the environment.

[[performance governor (throttling)?]]

5.1.4 ARM Cortex-A72

Finally, the popular Raspberry PI 4B single-board computer with ARM Cortex-A72 CPU with base frequency 1.8 GHz was used to test the trained models. Although it was not developed to run object detection models, it is perfect to test the smallest models and see how far go the possibilities of the real-time YOLOv8 object detector.

5.1.5 Software versions

Information about software installed on all previously mentioned devices can be found here. See Table 5.1 for a compact table displaying versions of relevant software. Following are the reasons behind some odd choices or compatibility issues.

JetPack SDK

Despite both Jetson Xavier NX and Jetson AGX Xavier being supported by NVIDIA JetPack SDK version `5.1.1`, deploying YOLOv8 models using this version often led to an untraceable fatal error. The error originated from one of NVIDIA's proprietary libraries and provided limited information regarding its cause. Fortunately, downgrading the JetPack version to `4.6.3` resolved this issue.

Although the root cause of the error and the specific package (or library) that required downgrading were not fully determined, it is suspected that the problem was related to TensorRT version `8.5.2`, as the error originated from the TensorRT development library `libnvinfer`.

For the sake of providing further context to the reader, the error message received was `operation.cpp:203: DCHECK(!i->is_use_only()) failed`. No other relevant warnings or error messages preceded this one, making it challenging to pinpoint the exact cause.

Old Python Version on Jetson Devices

All NVIDIA Jetson devices used to evaluate the trained models utilize the same version of the JetPack SDK, `4.6.3`, which includes Python version `3.6.9`. However, several essential Python packages – specifically, `protobuf` version `3.20.2`, `MMCV` version `2.0.0` and `MMEngine` version `0.7.2` – require a Python version of `3.7.0` or higher. As these packages (and the specified versions) were crucial for the model deployment and testing to be possible, we had to manually modify their requirements to allow for installation with Python version `3.6.9`.

Of course, this approach is not an ideal solution to the problem and its success was not guaranteed. Fortunately, no indications of compatibility issues were discovered during model deployment or testing.

One might suggest that upgrading to a newer Python version would be the most appropriate solution. However, due to compatibility and dependency constraints on Jetson devices, this is not feasible.

JetPack version `5.1.1` includes a more recent Python version but does not support Jetson Nano, and when installed on Jetson Xavier NX or Jetson AGX Xavier, the deployment of the trained models fails, as explained earlier in this section. Although installing a different Python version than the one provided with the JetPack SDK is possible, installing other necessary packages for the newer Python is not. This is because many such packages were developed specifically for Jetson devices and only support a certain Python version – the one pre-installed with the JetPack SDK.

Name	All NVIDIA Jetson devices	NVIDIA MX150	Intel Core i7-9850H	ARM Cortex-A72
Operating system	Linux for Tegra (L4T OS `32.7.3`)	Linux Debian 12		Linux Raspbian 11
JetPack SDK	`4.6.3`	-		
Python	`3.6.9`	`3.10.0`		`3.7.0`
CUDA	`10.2`	`11.8`	-	
TensorRT	`8.2.1`	-		
ONNX	1.13.1			`1.12.0`
ONNX Runtime	`1.11.0` (ver. GPU)	`1.12.0` (ver. GPU)	`1.12.0`	`1.11.0`
PyTorch	`1.10.0`	`2.0.0`		`1.8.0`
MMCV	`2.0.0`			
MMDeploy	`1.0.0`			`1.0.0rc3`
MMDetection	`3.0.0`			
MMEngine	`0.7.2`			
MMYOLO	`0.4.0`			

Table 5.1: Versions of relevant software installed on devices used to deploy and test the trained models.

5.2 Model Deployment and Optimizations

For model deployment, we have created an automated script utilizing the MMDeploy library's deployment script. The script uses all available deploy configurations[1] to deploy all trained PyTorch models to a specified backend (or backends).

There are only two deploy configurations for the ONNX Runtime backend – one with a static model shape[[**footnote čo je model shape**]] and one with a dynamic shape in the batch dimension[[**footnote**]]. For the TensorRT backend, two additional deploy configurations were created, both for a model with a dynamic shape in the batch dimension – one for weight quantization to the `fp16` representation, and one for weight quantization to the `int8` representation including weight calibration using the validation dataset.

To optimize inference on the Raspberry PI, we aimed to use the NCNN inference framework designed for mobile and embedded devices with limited computing resources. However, deploying a YOLOv8 model to the NCNN format is not yet possible, as the YOLOv8 model contains operations that are not yet supported by the framework and the MMYOLO library

[1] When deploying models using MMDeploy, deploy configurations are used to specify the parameters of the deployment process, including target backend or whether to apply post-training quantization

does not yet support deploying its models to NCNN either. **[[meaning it doesn't yet convert unsupported operations to supported ones when deploying to NCNN?]]**

Devices Used for Deployment

The models in the ONNX format were all deployed on a single device (with ONNX package version `1.13.1`) and distributed to all devices. The TensorRT backend is only used on NVIDIA Jetson devices and all models were deployed to the TensorRT engine[2] individually on each NVIDIA Jetson device, because they are platform-specific and transferring them across different devices is not recommended.

Weight Quantization

The MMDeploy library supports post-training quantization to `fp16` and `int8` representations during the process of model deployment to TensorRT. To preserve the model's accuracy after quantization to `int8` precision, weight calibration was done using the validation dataset.

Although the ONNX Runtime framwork also supports quantization (to both `fp16` and `int8`), the process is not as straightforward and doesn't seem to be supported by the MMDeploy library. Although evaluating models quantized to `int8` representation would be beneficial, the quantization to the `fp16` precision would probably have a little effect on performance on CPUs as they do not usually support operations with this precision and calculate them using the same operators as numbers in the `fp32` representation.

Additionally, TensorRT on the NVIDIA Jetson Nano with Maxwell GPU doesn't support fast inference of models quantized to `int8`, so quantization to `int8` won't be performed on this device.

5.3 Experiments and Evaluation

In this section, we will discuss additional details of all performed experiments – mainly the combinations of models, devices, inference backends, deployment configurations, optimization techniques and batch sizes of individual tests. Insights gained from these experiments and additional experiment-specific details will be discussed in chapter 6.

mmdeploy má test skript, nepoužil som svoj lebo predpokladám že vedia čo robia

vytvoril som test skript ktorý ho spúšťa robí automaticky pre všetky kombinácie

ďalší skript ktorý z logov testov zozbiera priemerné FPS a mAP metriky

Možno tabuľka všetkých testov alebo tak

metriky - FPS a coco mAP metriky

pre onnxruntime som nastavil inter a intra thread na 1

[2] A TensorRT engine is a deployed model in the TensorRT format.

Chapter 6

Results

Chapter 7

Future Work

knowledge distillation, pruning,... kvantizácia onnx

Porovnať onnxruntime a tensorRT na bežnej GPU

NCNN alebo ARM NN pre RPI

Chapter 8

Conclusion

Bibliography

[1] MIO-TCD: A New Benchmark Dataset for Vehicle Classification and Localization. *IEEE Transactions on Image Processing*. Institute of Electrical and Electronics Engineers (IEEE). oct 2018, vol. 27, no. 10, p. 5129–5141. DOI: 10.1109/tip.2018.2848705.

[2] Bahnsen, C. H. and Moeslund, T. B. Rain Removal in Traffic Surveillance: Does it Matter? *IEEE Transactions on Intelligent Transportation Systems*. Institute of Electrical and Electronics Engineers (IEEE). aug 2019, vol. 20, no. 8, p. 2802–2819. DOI: 10.1109/tits.2018.2872502.

[3] Bai, J., Lu, F., Zhang, K. et al. *ONNX: Open Neural Network Exchange* [https://github.com/onnx/onnx]. 2019.

[4] Bugdol, M., Miodonska, Z., Krecichwost, M. and KASPEREK, P. Vehicle detection system using magnetic sensors. *Transport Problems*. march 2014, vol. 9.

[5] Buslaev, A., Parinov, A., Khvedchenya, E., Iglovikov, V. I. and Kalinin, A. A. Albumentations: fast and flexible image augmentations. *ArXiv e-prints*. 2018.

[6] Choudhary, T., Mishra, V., Goswami, A. and Sarangapani, J. A comprehensive survey on model compression and acceleration. *Artificial Intelligence Review*. Springer Science and Business Media LLC. feb 2020, vol. 53, no. 7, p. 5113–5155. DOI: 10.1007/s10462-020-09816-7.

[7] Contributors, M. *OpenMMLab's Model deployment toolbox*. December 2021. Available at: https://github.com/open-mmlab/mmdeploy.

[8] Contributors, M. *OpenMMLab Detection Toolbox and Benchmark*. August 2018. Available at: https://github.com/open-mmlab/mmdetection.

[9] Contributors, M. *MMYOLO: OpenMMLab YOLO series toolbox and benchmark* [https://github.com/open-mmlab/mmyolo]. 2022.

[10] developers, O. R. *ONNX Runtime* [https://onnxruntime.ai/]. 2021.

[11] Fang, J., Meng, H., Zhang, H. and Wang, X. A Low-cost Vehicle Detection and Classification System based on Unmodulated Continuous-wave Radar. In: *2007 IEEE Intelligent Transportation Systems Conference*. 2007, p. 715–720. DOI: 10.1109/ITSC.2007.4357739.

[12] Girshick, R., Donahue, J., Darrell, T. and Malik, J. Rich feature hierarchies for accurate object detection and semantic segmentation. arXiv. november 2013. DOI: 10.48550/ARXIV.1311.2524.

[13] He, K., Zhang, X., Ren, S. and Sun, J. Deep Residual Learning for Image Recognition. arXiv. december 2015. DOI: 10.48550/ARXIV.1512.03385.

[14] Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W. et al. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv. april 2017. DOI: 10.48550/ARXIV.1704.04861.

[15] Jensen, M. B., Mogelmose, A. and Moeslund, T. B. Presenting the Multi-view Traffic Intersection Dataset (MTID): A Detailed Traffic-Surveillance Dataset. In: *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*. IEEE, Sep 2020. DOI: 10.1109/itsc45102.2020.9294694.

[16] Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In: Pereira, F., Burges, C., Bottou, L. and Weinberger, K., ed. *Advances in Neural Information Processing Systems*. Curran Associates, Inc., 2012, vol. 25. Available at: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[17] Labelbox. *Labelbox* [online]. 2022. Available at: https://labelbox.com.

[18] Lecun, Y., Bottou, L., Bengio, Y. and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*. Institute of Electrical and Electronics Engineers (IEEE). 1998, vol. 86, no. 11, p. 2278–2324. DOI: 10.1109/5.726791.

[19] Li, Z., Liu, F., Yang, W., Peng, S. and Zhou, J. A Survey of Convolutional Neural Networks: Analysis, Applications, and Prospects. *IEEE Transactions on Neural Networks and Learning Systems*. Institute of Electrical and Electronics Engineers (IEEE). dec 2022, vol. 33, no. 12, p. 6999–7019. DOI: 10.1109/tnnls.2021.3084827.

[20] Luca, C., Carlos, S., Paulo, C. J., Claudio, G. and Giuseppe, A. *Night and Day Instance Segmented Park (NDISPark) Dataset: a Collection of Images taken by Day and by Night for Vehicle Detection, Segmentation and Counting in Parking Areas*. Zenodo, 2022. DOI: 10.5281/ZENODO.6560822.

[21] Murali, S. and V K, G. Shadow Detection and Removal from a Single Image Using LAB Color Space. *Cybernetics and Information Technologies*. march 2013, vol. 13. DOI: 10.2478/cait-2013-0009.

[22] O'Shea, K. and Nash, R. An Introduction to Convolutional Neural Networks. arXiv. november 2015. DOI: 10.48550/ARXIV.1511.08458.

[23] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E. et al. Automatic differentiation in PyTorch. 2017.

[24] Simonyan, K. and Zisserman, A. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv. september 2014. DOI: 10.48550/ARXIV.1409.1556.

[25] Stiawan, R., Kusumadjati, A., Aminah, N. S., Djamal, M. and Viridi, S. An Ultrasonic Sensor System for Vehicle Detection Application. *Journal of Physics: Conference Series*. IOP Publishing. apr 2019, vol. 1204, no. 1, p. 012017. DOI:

10.1088/1742-6596/1204/1/012017. Available at: https://dx.doi.org/10.1088/1742-6596/1204/1/012017.

[26] Tang, C., Ouyang, K., Wang, Z., Zhu, Y., Wang, Y. et al. Mixed-Precision Neural Network Quantization via Learned Layer-wise Importance. arXiv. march 2022. DOI: 10.48550/ARXIV.2203.08368.

[27] Wang, T., Zhu, Y., Zhao, C., Zeng, W., Wang, Y. et al. Large Batch Optimization for Object Detection: Training COCO in 12 minutes. In: *Computer Vision – ECCV 2020*. Springer International Publishing, 2020, p. 481–496. DOI: 10.1007/978-3-030-58589-1_29.

[28] Wen, L., Du, D., Cai, Z., Lei, Z., Chang, M.-C. et al. UA-DETRAC: A New Benchmark and Protocol for Multi-Object Detection and Tracking. arXiv. november 2015. DOI: 10.48550/ARXIV.1511.04136.

[29] Wu, K., Xu, T., Zhang, H. and Song, J. Overview of video-based vehicle detection technologies. *ICCSE 2011 - 6th International Conference on Computer Science and Education, Final Program and Proceedings*. august 2011. DOI: 10.1109/ICCSE.2011.6028764.

[30] Zhu, P., Wen, L., Du, D., Bian, X., Fan, H. et al. Detection and Tracking Meet Drones Challenge. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. Institute of Electrical and Electronics Engineers (IEEE). nov 2022, vol. 44, no. 11, p. 7380–7399. DOI: 10.1109/tpami.2021.3119563.