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

BRNO UNIVERSITY OF TECHNOLOGY
FACULTY OF INFORMATION TECHNOLOGY

Bachelor's Thesis Assignment

144778

Institut:	Department of Computer Graphics and Multimedia (UPGM)
Student:	**Skaloš Patrik**
Programme:	Information Technology
Specialization:	Information Technology
Title:	**Detection and Classification of Vehicles for Embedded Platforms**
Category:	Image Processing
Academic year:	2022/23

Assignment:

1. Learn about methods for detecting and classifying objects in an image.
2. Obtain a suitable dataset for detecting different types of vehicles in the image.
3. Find deep learning methods that focus on the object detection problem, primarily methods that support multi-class detection or object detection coupled with subsequent classification.
4. Select appropriate methods for use on embedded platforms and experiment with them.
5. Evaluate the selected methods in an appropriate manner and discuss the results obtained.
6. Create a poster and video presenting your work, its objectives and results.

Literature:

- GE, Zheng, et al. Yolox: Exceeding yolo series in 2021. *arXiv preprint arXiv:2107.08430*, 2021.
- HUANG, Jonathan, et al. Speed/accuracy trade-offs for modern convolutional object detectors. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017. p. 7310-7311.
- Further as instructed by supervisor.

Requirements for the semestral defence:

- Completion of the first three points of the assignment
- Considerable work on the fourth point of the assignment

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Špaňhel Jakub, Ing.**
Head of Department:	Černocký Jan, prof. Dr. Ing.
Beginning of work:	1.11.2022
Submission deadline:	10.5.2023
Approval date:	31.10.2022

Abstract

This paper evaluates the performance trade-offs of state-of-the-art YOLOv8 object detectors for vehicle detection in surveillance-type images on embedded and low-performance devices. YOLOv8 models of varying sizes, including one with the lightweight MobileNetV2 backbone and YOLOv8-femto with fewer than 60 000 parameters, were benchmarked across six devices, including three NVIDIA Jetson embedded platforms and the low-performance Raspberry Pi 4B. Various factors influencing performance were considered, such as weight quantization, input resolutions, inference backends, and batch sizes during inference. This study provides valuable insights into the development and deployment of vehicle detectors on a diverse range of devices, including low-performance CPUs and specialized embedded platforms.

Abstrakt

Táto práca hodnotí kompromisy rýchlosti a presnosti najmodernejších detektorov objektov YOLOv8 pre detekciu vozidiel v snímkoch z monitorovacích kamier na vstatných a nízkovýkonných zariadeniach. Modely YOLOv8 rôznych veľkostí, vrátane jedného s efektívnou sieťou MobileNetV2 na extrakciu príznakov a modelu YOLOv8-femto s menej ako 60 000 parametrami, boli testované na šiestich zariadeniach, vrátane troch vstavaných platforiem z rodiny NVIDIA Jetson a počítačom Raspberry Pi 4B s nízkou výpočtovou silou. V práci boli zohľadnené rôzne faktory ovplyvňujúce výkonnosť modelov, ako napríklad ich kvantizácia, rozlíšenia vstupu, inferenčné knižnice a veľkosti dávok počas inferencie. Táto štúdia poskytuje užitočné informácie k vývoju a nasadeniu detektorov vozidiel na širokú škálu zariadení, od nízkovýkonných procesorov po špecializované vstavané platformy.

Keywords

vehicle detection, traffic surveillance, object detection, convolutional neural networks, real-time, trade-offs, embedded devices, low-performance devices, YOLO, YOLOv8, NVIDIA Jetson, Raspberry Pi, OpenMMLab, MMYOLO, quantization, ONNX Runtime, TensorRT

Klíčová slova

detekcia vozidiel, monitorovanie dopravy, detekcia objektov, konvolučné neurónové siete, detekcia v reálnom čase, kompromisy, vstavané zariadenia, nízkovýkonné zariadenia, YOLO, YOLOv8, NVIDIA Jetson, Raspberry Pi, OpenMMLab, MMYOLO, kvantizácia, ONNX Runtime, TensorRT

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 Ing. Jakub Špaňhel. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Patrik Skaloš
May 8, 2023

Acknowledgements

I would like to express my gratitude to my supervisor, Ing. Jakub Špaňhel, for his valuable advice throughout the course of this research. His guidance and expertise helped me overcome difficult challenges and enabled me to focus on the important aspects of this work to successfully complete this thesis.

Contents

Chapter 1

Introduction

Vehicle detection systems are an important tool for enhancing road safety, and with the rise of convolutional neural networks and the recent advancements in object detection, these systems have become more affordable and accurate. However, high-performance object detectors still require significant computational resources and sacrifices must be made to perform this task in real-time on embedded devices or those with limited computational resources. This paper aims to evaluate the performance trade-offs of state-of-the-art object detectors on low-performance devices and to explore various strategies for improving their efficiency.

A popular family of real-time object detectors is the YOLO (You Only Look Once) series, with the latest, state-of-the-art YOLOv8 model architecture being the primary focus of this paper. To thoroughly evaluate its performance, several YOLOv8 models of different sizes are benchmarked, including one trained with the lightweight MobileNetV2 backbone. These models are tested across six different devices, including three NVIDIA Jetson embedded platforms. To accelerate the process of training, deploying and testing different models, this paper utilized PyTorch-based libraries from the OpenMMLab project, including MMYOLO and MMDeploy.

The models were trained on several different datasets that consist of surveillance-type images, most commonly captured by traffic monitoring cameras mounted on existing infrastructure. The datasets combined contain a total of 1 204 795 object instances in 218 821 images.

In addition to benchmarking the trained models on all devices in terms of their inference speeds (FPS) and mean Average Precision (mAP) metrics, this paper also aims to explore additional factors that can influence their performance, including weight quantization and the usage of different inference backends, inference batch sizes and input resolutions of the models.

In Chapter 2, we begin by explaining essential concepts, tools and libraries relevant to this study. We then proceed with a detailed review of the datasets employed in our research in Chapter 3, followed by a discussion on the evaluated object detector models and their training process in Chapter 4.

Subsequently, in Chapter 5, we delve into the specifics of our experiments, outlining the methodologies employed, providing an overview of the devices used for testing and discussing the important aspects to consider when interpreting the results. These results are then presented in detail in Chapter 6, where we share our findings and insights gained from the experiments. Finally, in Chapter 8, we suggest potential areas for future work that were beyond the scope of this paper.

Chapter 2

Background and Related Work

In this chapter, an overview of the key concepts and techniques in the field of vehicle detection is provided, with a focus on those relevant to the study at hand. The discussion begins with traditional approaches to vehicle detection before the introduction of Convolutional Neural Networks (CNNs). This is followed by an exploration of the problem of object detection using CNNs, network optimization, and network compression techniques. The chapter also covers evaluation metrics for assessing a model's performance, a review of existing literature on this subject, an overview of embedded platforms for designed for machine learning applications and finally, a review of various tools and libraries utilized in this paper.

2.1 Traditional 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 in concrete to detect the presence of metal objects passing by. With a controller, induction loops typically provide data about vehicle presence, but using more advanced algorithms, vehicle speed, length 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 the ultrasonic sensor [33]. 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 the 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 [38]. 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 [25], 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

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 [21]. CNNs are particularly useful in the field of computer

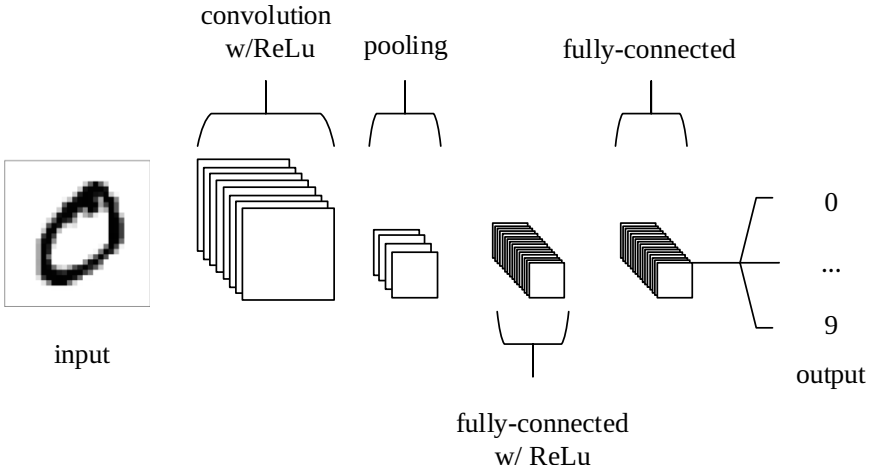


Figure 2.1: An example architecture of a CNN with five layers [27] for hand-written digit recognition. ReLu (Rectified Linear Unit) is a commonly used activation function in neural networks, but won't be discussed here in detail.

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.

2.2.1 Architecture of Convolutional Neural Networks

A typical CNN consists of several key components, including convolutional layers, pooling layers 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. A CNN with a simple architecture is illustrated in Figure 2.1.

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 [27]. A kernel, also called 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 parameters (also called weights) are adjusted during the training phase.

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 explained visually by Figure 2.2.

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.

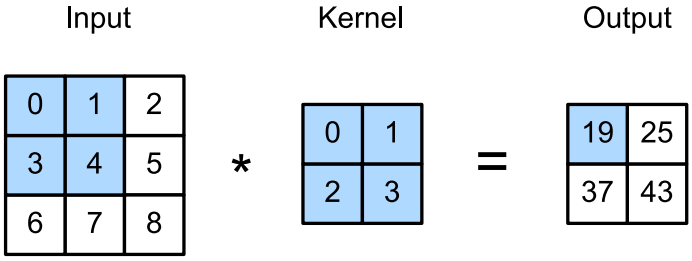


Figure 2.2: Computation of a 2D convolution on a small input image with an example 2×2 kernel [40]. The output is also called a feature map.

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 [27]. 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 specific operation is called max-pooling.

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 [27]. In a CNN, it is typically used at the end of the network to classify the input data (features extracted by the previous convolutional layers).

2.2.2 Popular Architectures of Convolutional Neural Networks

Over the years, various CNN architectures have been developed to address different challenges and requirements. This subsection contains a review of 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.

LeNet-5

LeNet-5 [20], introduced by Yann LeCun and his team in 1998 is considered one of the first successful applications of convolutional neural networks. Designed for hand-written digit recognition, LeNet-5 was the source of inspiration for modern CNNs with its combination of convolutional, pooling and fully connected layers. Its simple architecture is illustrated in Figure 2.3.

AlexNet

Developed by Alex Krizhevsky, Ilya Sutskever, and Geoffrey Hinton in 2012, AlexNet [18] marked a significant breakthrough in the field of deep learning and won the ImageNet

[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.

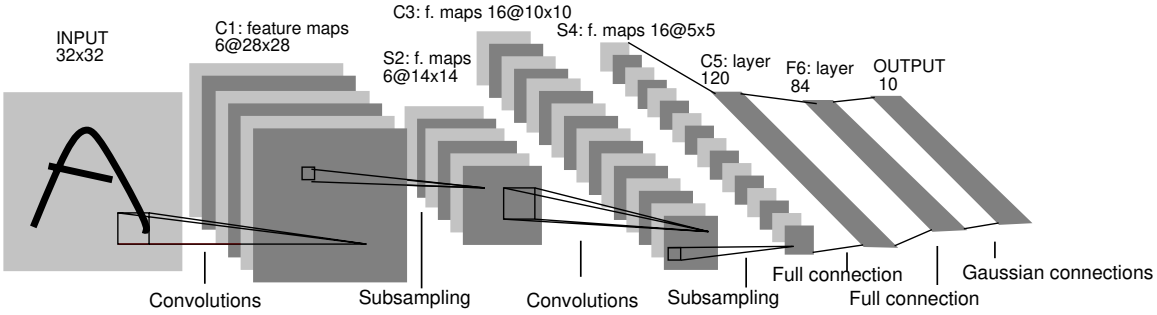


Figure 2.3: Architecture of the LeNet-5 convolutional neural network designed for the recognition of hand-written digits. In this figure the planes represent feature maps. [20]

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 [32], 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 small (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 mostly lies in it being very computationally expensive.

ResNet

In 2015, ResNet [14] (Residual Network) was introduced by Kaiming He and his team, addressing the degradation problem that occurs in very deep CNNs. ResNet incorporates residual connections, which 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 [15], developed by Andrew G. Howard and his team at Google in 2017, is a popular lightweight convolutional neural network architecture that has been widely used on mobile and embedded devices with limited computational resources. One of its key 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.

MobileNetV2

MobileNetV2 [30], an evolution of the original MobileNet was proposed by Mark Sandler and his team at Google in 2018. The architecture introduces the concept of *Inverted Residuals* and *Linear Bottlenecks*, aiming to enhance the network's representational capacity while maintaining low computational complexity. *Inverted Residual* blocks leverage the idea of residual connections to mitigate the vanishing gradient problem and improve the training phase. *Linear Bottlenecks*, on the other hand, reduce the computational cost of convolutional layers by decreasing the number of channels before applying a convolutional operation and restoring the original number of channels afterward. These advancements enable MobileNetV2 to achieve superior accuracy and efficiency compared to its predecessor.

2.3 Object Detection Based on Convolutional Neural Networks

While the CNNs discussed in 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. However, these CNNs can, and often are used as backbones[3] in object detectors to extract features from the input images. Modern object detectors can be divided into two categories: two-stage detectors and one-stage detectors.

2.3.1 Two-Stage Object Detectors

In 2013, Girshick *et al.* proposed the R-CNN framework [12] (Regions with CNN features) which revolutionized object detection by using a two-stage approach. In the first stage, the detection system takes the input image and selects region proposals (regions that are likely to contain an object). These proposals are then passed to the second stage where features are extracted from each of the proposed regions, which can then be classified. This algorithm marked a breakthrough in object detection, outperforming the current state-of-the-art detectors.

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 further detail.

2.3.2 One-Stage Object Detectors

When building a real-time object detector, most commonly a one-stage detector is selected. Although their accuracy is generally lower than that of two-stage detectors, their speed allows for real-time object detection even on resource-constrained devices.

Single Shot MultiBox Detector (SSD)

The Single Shot MultiBox Detector (SSD) is an object detection algorithm that aims to provide a balance between accuracy and computational efficiency. Proposed by Wei Liu *et*

[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.

[3]The backbone of an object detector is responsible for extracting features from the input image.

al. in their 2015 paper [22], SSD addresses the challenge of detecting objects in images with varying scales and aspect ratios in real-time.

Unlike the two-stage object detectors which consist of a region proposal stage followed by a classification stage, SSD directly predicts both the object locations and their corresponding class labels in a single forward pass of the network.

You Only Look Once (YOLO)

YOLO (You Only Look Once) is a family of one-stage object detectors that prioritize real-time performance. Introduced by Joseph Redmon *et al.* in 2015 [29], YOLO divides the input image into a grid, where each grid cell is responsible for predicting bounding boxes and class probabilities for objects located within that cell. The original YOLO algorithm has undergone many iterations each improving upon the previous version's performance and efficiency: from YOLOv2 to YOLOv8 and including YOLOX, YOLOR and several more.

This work is focused on the latest, state-of-the-art object detector YOLOv8 [17] developed by Ultralytics, which, at this time, doesn't have a paper released. However, information about the detector can be drawn from a post by OpenMMLab [26].

Built upon the YOLOv5, the YOLOv8 detector updated the head[4] module to be a decoupled one, separating the classification and detection heads, while the new backbone and neck[5] modules are based on the YOLOv7 ELAN concept. In terms of training strategy, YOLOv8 extends the training epochs from 300 to 500 and the data augmentation process is modified during the final 10 epochs, with a reduction in the intensity of augmentations. Additionally, the loss function undergoes a revision, incorporating the TaskAlignedAssigner from TOOD and introducing Distribution Focal Loss for regression loss, which further enhances the detector's accuracy and efficiency.

2.4 Network Optimization and Compression

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 provided in this section. However, in this paper, only one of these techniques — weight quantization — will be used.

2.4.1 Network Pruning

As explained by Tejalal Choudhary *et al.* in their paper [6], there are typically many parameters in a CNN which are 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,

[4]The head of an object detector predicts the object classes and bounding box coordinates using features input by the neck.

[5]The neck module of an object detector refines features input from the backbone.

more aggressive pruning can be performed, removing even more parameters, including not just redundant, but also less important ones, although sacrificing some accuracy.

Various pruning techniques have been developed, including weight pruning (removal of individual weights), neuron pruning (removal of an entire neuron and its connections), filter pruning and layer pruning. Although this technique was developed to reduce the model's storage requirements, it can also be used to reduce its computational requirements.

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 [13]. The student network learns from outputs from the teacher network instead of the ground truth labels. Of course, practically, more advanced and effective knowledge distillation methods are used, but will not be discussed in this paper. 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.

2.4.3 Weight Quantization

In convolutional neural networks, weights and biases are stored as 32-bit floating-point numbers (also called `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 [6].

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 real-time object detector.

Most commonly, the parameters of models 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 the integer representation (also called precision in this context), such as `INT8`, is a different process. Because of the limitations of the integer representation and the distinction from floating-point representations, weight calibration[6] during the quantization process or even model fine-tuning[7] 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 [34].

Most deep learning libraries which can be used for model deployment or inference on a target device — including TensorRT and ONNX Runtime — support weight quantization and offer tutorials on how it's done.

[6] 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.

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

2.5 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. This section provides a brief explanation and background of the mean Average Precision (mAP) metric, which will be used to evaluate the detectors trained in this project. The information in this section is derived from a blog post on the V7Labs website [31].

Precision and Recall

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[8] objects (true positives and false negatives) in the dataset. In other words, precision represents how many predictions of the model are correct and recall measures how many of the ground truth objects were predicted by the model.

If the precision and recall values are computed at different confidence score[9] thresholds, they can be plotted one against the other to obtain a precision-recall curve, which can be used to visually evaluate the overall accuracy of the model.

Average Precision

The Average Precision (AP) is a metric widely used to represent the performance of an object detector. It is simply calculated as 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.

Mean Average Precision

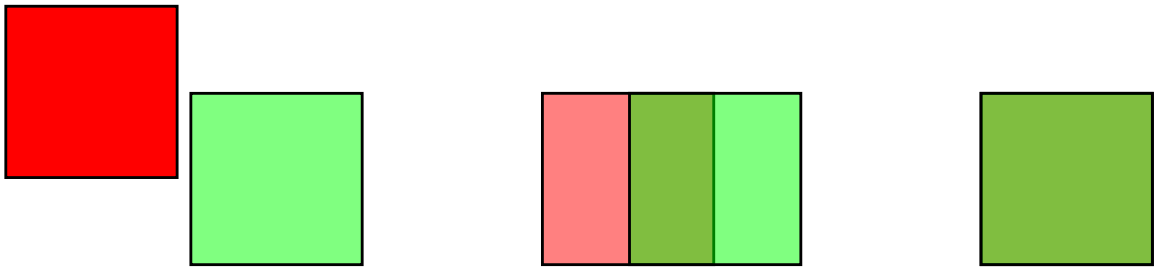
To evaluate multi-class object detectors, mean Average Precision (mAP) is used instead of the previously explained Average Precision. Computing the mAP involves finding the AP for each class and calculating the average over the number of classes.

However, for object detection tasks, precision is typically calculated with different thresholds of the Intersection over Union (IoU). The IoU measures the overlap between two bounding boxes — 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 2.4.

In this paper, we consider the COCO mAP specification and calculate the mAP as the average of AP calculated for all classes and over ten 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

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

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



(a) $IoU = 0$ (b) $IoU \approx 0.333$ (c) $IoU = 1$

Figure 2.4: Visualization of the Intersection over Union calculation.

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. The mAP values in this paper will be presented as percentages.

2.6 Vehicle Detection Based on Convolutional Neural Networks

This section delves into the existing literature on vehicle detection using deep learning and convolutional neural networks. We advise the reader to be cautious when comparing their results because they vary based on the selected evaluation metrics, used datasets and devices used to evaluate the models.

Deep learning has revolutionized vehicle detection by significantly improving accuracy and robustness. In a study from 2019 [35], the authors compare five mainstream deep learning-based object detection algorithms for vehicle detection in autonomous driving: Faster R-CNN, R-FCN, SSD, RetinaNet and YOLOv3. The results indicate that two-stage detectors generally have better detection accuracy than one-stage models, while SSD and YOLOv3 algorithms were found to have excellent real-time performance and generalization abilities.

In a more recent paper by Maity *et al.* [24], the authors present a comprehensive review of existing Faster R-CNN and YOLO-based vehicle detection and tracking methods, comparing them and highlighting their advantages over traditional vehicle detection methods. The authors also discuss several vehicle detection methods featuring modified YOLO models to improve detection performance.

In [39], the authors propose YOLOv5-Ghost — an improved neural network structure based on YOLOv5-small — to detect vehicles in the CARLA[10] virtual environment. By replacing BottleneckCSP in the YOLOv5-small with Ghost Bottleneck, the detection speed is increased from 29 FPS (YOLOv5-small) to 47 FPS (YOLOv5-Ghost) while only reducing the mAP from 83.3 % to 80.7 % on their test dataset.

Based on the literature reviewed, the main focus of this paper is placed on object detectors of the YOLO family, with the understanding that these models can be further optimized to achieve faster inference speeds.

[10]CARLA (Car Learning to Act) is an open-source, realistic urban environment simulator designed for the development, testing and validation of autonomous driving systems in various traffic scenarios and conditions.

2.7 Embedded Platforms for Machine Learning

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. The focus of this section is on discussing some of the most popular embedded devices and platforms used for machine learning, including object detection.

These 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) or tensor processing units (TPUs).

2.7.1 Google Coral

Designed for TensorFlow Lite[11] 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[12]. Coral provides development boards and USB accelerators along with a variety of modules and peripherals for edge AI applications.

2.7.2 Movidius Neural Compute Stick

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

2.7.3 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. In this paper, three NVIDIA Jetson devices will be evaluated. Following is a brief overview of the hardware and software for these platforms.

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

[11] TensorFlow Lite is a lightweight machine learning framework designed for running TensorFlow models efficiently on devices with limited computational resources.

[12] Inference in machine learning is the process of using a trained model to make predictions on new data, previously unseen by the model.

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.

Software for NVIDIA Jetson devices

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.

On all used NVIDIA Jetson platforms, JetPack SDK version `4.6.3` was used. It's important to note that using this version of the JetPack SDK means using older software versions, including Python version `3.6`, TensorRT version `8.2.1` and CUDA version `10.2`. The reason for not using a more recent JetPack SDK (even though it supports the Jetson AGX Xavier and Jetson Xavier NX), along with the complications it caused, will be discussed in Chapter 5.

2.8 Tools and Libraries

In this section, we provide a brief overview of some crucial tools and libraries utilized throughout the project.

2.8.1 LabelBox

LabelBox [19] is a web-based application and data labeling platform widely used when 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, the application was used to reannotate one of the datasets.

2.8.2 PyTorch

PyTorch [28] 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.

2.8.3 MMDetection

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 (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[13] and MMEngine[14].
- Its modular design allows for easy customization and extension of the codebase.
- It includes pre-trained models and their configurations, making it possible to use the transfer learning technique[15] for simpler and faster training and comparison of different models.

2.8.4 MMYOLO

Although the MMDetection library doesn't include the latest detectors of the YOLO family, such as the YOLOv8 detector 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, including data augmentations or loss functions.

2.8.5 MMDeploy

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.

2.8.6 ONNX and ONNXRuntime

ONNX [3] (Open Neural Network Exchange) 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

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

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

[15] Transfer learning is a machine learning technique that leverages knowledge of a pre-trained model to more efficiently train a new model, reducing training time and data requirements.

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.

2.8.7 TensorRT

TensorRT[16] is a high-performance deep learning inference optimization and runtime library developed by NVIDIA. It is designed to accelerate the 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.

[16]The TensorRT library is available at https://developer.nvidia.com/tensorrt

Chapter 3

Datasets

In this chapter, each of the used datasets is first thoroughly analyzed and at the end, a summary of all used datasets is provided along with details on how the dataset was split into training, validation and testing subsets[1]. The main focus when selecting appropriate datasets was on the camera's point of view (this paper aims to develop a vehicle detector for inputs of a surveillance type) and the categories to which the vehicles are classified. In this paper, eight vehicle classes are considered:

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

3.1 Analysis of Individual Datasets

This section individually analyzes all datasets used in this work, discussing their sizes, origins, advantages and drawbacks and also provide an example image from each of the datasets.

3.1.1 UA-DETRAC

The UA-DETRAC dataset [37] (hereafter referred to as DETRAC) provided by the University at Albany is the biggest and the most important dataset for this work, originally containing 1 274 055 annotations of 8 250 vehicles in 138 252 images. The dataset is provided as frames of 100 video sequences at 25 FPS with resolutions of 960×540 pixels. The

[1] Developing an object detector includes training it on the training dataset while regularly evaluating its progress by testing it on the validation dataset. Finally, the trained model is evaluated on the testing dataset to provide an independent metric.

Figure 3.1: Example image from the DETRAC dataset.

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.

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 and therefore moving while the masks are static throughout the individual sequences.

These problems were fixed by importing the dataset to the LabelBox labeling application, adjusting the masks (while also making them dynamic to compensate for camera movements), 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 the following classes (the dataset contains no trailers):

(a) Resolution 720 × 480 (b) Resolution 342 × 228

Figure 3.2: Example images from the Miovision dataset.

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

3.1.2 Miovision Traffic Camera Dataset

The Miovision Traffic Camera Dataset [1] (hereafter referred to as Miovision dataset) is another large and very important dataset. The images were 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, while 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 objects of the following classes:

- Pedestrian (ignored)
- Bicycle
- Motorcycle
- Car
- Pickup truck (mapped to *transporter*)
- Work van (mapped to *transporter*)
- Bus

Figure 3.3: Example image from the AAU dataset.

- Articulated truck (mapped to *truck*)
- 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.

3.1.3 AAU RainSnow Traffic Surveillance Dataset

Another important dataset is the AAU RainSnow Traffic Surveillance dataset [2] (hereafter referred to simply as AAU). 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 2 200 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 this paper, only the annotated frames from the RGB camera are used, containing 13 297 annotations in 2 200 frames (before processing) of resolution 640×480 pixels. See an example in Figure 3.3.

The dataset uses these six classes:

- Pedestrian
- Bicycle
- Motorbike
- Car

Figure 3.4: Example image from the drone subset of the MTID dataset.

- Bus
- Truck

This introduces a problem — vehicles of the 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 was 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 1 899 images.

3.1.4 Multi-View Traffic Intersection Dataset (MTID)

For the Multi-View Traffic Intersection Dataset [16] (hereafter referred to as MTID), the authors recorded a single 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 5 776 frames (equal share of frames for both cameras). An example from this dataset can be seen in Figure 3.4.

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, 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 5 399 frames.

3.1.5 Night and Day Instance Segmented Park Dataset (NDISPark)

Another useful dataset is the Night and Day Instance Segmented Park dataset [23] (hereafter referred to as NDISPark), 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 3 302 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 2 400 px in width and within 908 px and 1 808 px in height.

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.

3.1.6 VisDrone Dataset

The VisDrone dataset [41] 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. In the test subset of the object detection part of this dataset after processing, there are 47 720 annotated objects in 1 610 frames, which will be used in this paper. An example is displayed in Figure 3.6.

We consider it to be a helpful addition to our datasets as it contains useful negative images, many pedestrians and new points of view, and it often captures vehicles from a bid's-eye view. Additionally, objects in this dataset are classified into 12 classes, which can be easily mapped to the 8 internal classes.

3.2 Dataset Processing

Before training, all datasets used in this project must be converted to a unified format and vehicle classes need to be mapped to be the same in each dataset. Additionally, some

Figure 3.5: Example image from the NDISPark dataset.

datasets include images with regions that should be masked out, and certain subsets or images of some datasets need to be ignored.

Therefore, a Python script was developed for each dataset. It first loads the annotations from the original format — COCO[2] for AAU, MTID and NDISPark, XML for DETRAC and different CSV formats for Miovision and VisDrone datasets. The script then maps the classes to ones shown earlier in this chapter and if needed, removes the ignored subsets or images before saving the labels in the COCO format. The processing script for NDISPark dataset also corrects the object classes (since the original dataset annotates all objects with a single class) before saving, and therefore does not modify the original dataset file (class corrections are defined in the script itself).

MMDetection's middle format was considered as it's 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 (the MMDetection's middle format is saved as a pickle[3] file), and most importantly, MMYOLO and the most recent version of MMDetection at the time of preparing the datasets (`v0.4.0` and `v3.0.0rc2`) do not seem to fully support datasets in the middle format to be used in all of their dataset wrappers.

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

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

[2]The widely used COCO annotation format defines how a dataset (its categories, list of images, annotations and other metadata) should be stored in the JSON (JavaScript Object Notation) format.

[3]The Python's pickle format refers to a binary serialization and deserialization method allowing for the conversion of Python objects into a byte stream.

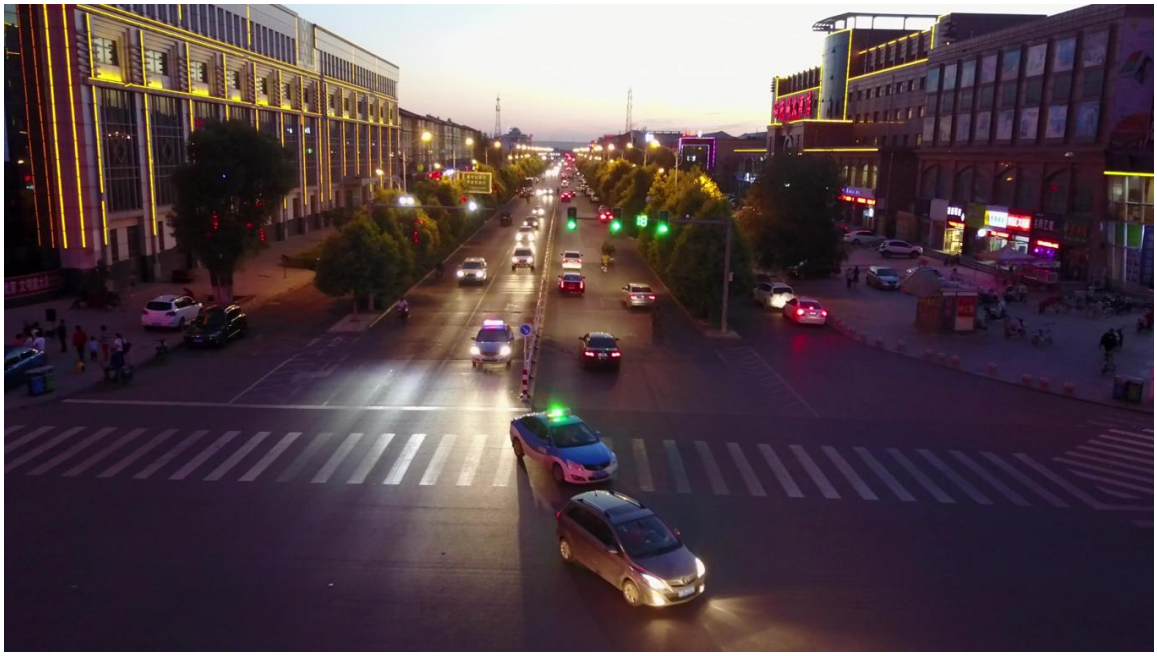


Figure 3.6: Example image from the VisDrone dataset.

3.3 Summary of Datasets

Table 3.1 contains a comparison of used datasets on a higher level and shows the number of images and instances contained with additional comments. It is clear that the reannotated DETRAC dataset amounts for most of the data and might have been enough by itself. However, to train a robust vehicle detector, it is essential to utilize every available and relevant dataset. The images from the selected datasets feature a great variety of lighting and weather conditions, points of view, object scales, occlusions and other relevant factors. Additionally, Table 3.2 shows the number of annotated object instances per class in all used datasets combined.

3.4 Training, Validation and Testing Dataset Split

We chose to only include the Miovision and DETRAC 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.

From the reannotated DETRAC 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.

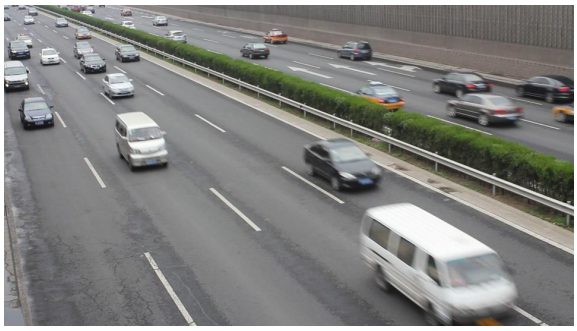
Dataset	Images	Instances	Comments
DETRAC	99 771	733 833	Large Continuous video High-quality camera Different lighting conditions
Miovision	110 000	344 416	Large Low-quality images
AAU	1 899	10 545	Small Different weather conditions
MTID	5 399	64 979	Small Continuous video
NDISPark	142	3 302	Small Occlusions
VisDrone	1 610	47 720	Small Negative images New points of view
Total	218 821	1 204 795	–

Table 3.1: High-level comparison of used datasets after processing: the number of images and object instances, and additional comments for each of the datasets.

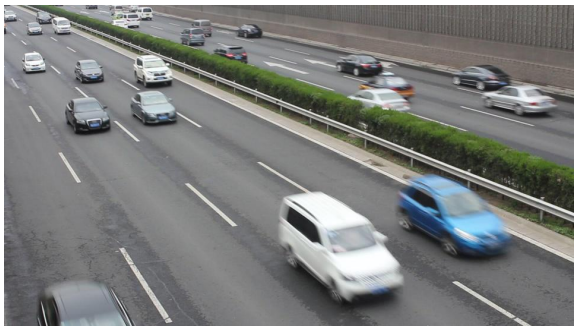
The validation subset contains a total of 36 969 objects on 7 770 images, of which 5 500 images are from the Miovision dataset and 2 270 from DETRAC. Similarly, the test subset contains a total of 50 357 objects on 7 990 images — 5 500 images from the Miovision dataset and 2 490 from the DETRAC dataset.

Class	Instances
Bicycle	14 036
Motorcycle	17 187
Passenger car	916 317
Transporter	123 585
Bus	64 529
Truck	38 069
Trailer	2 360
Unknown	28 712
Total	1 204 795

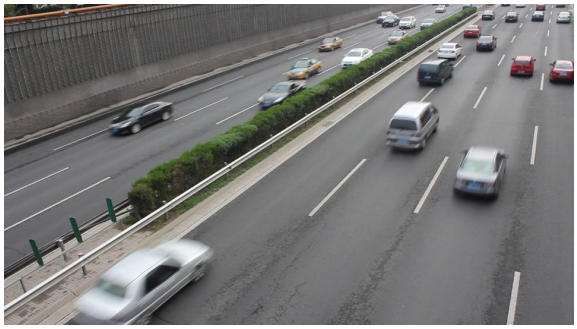
Table 3.2: Numbers of class instances in all datasets combined after processing.



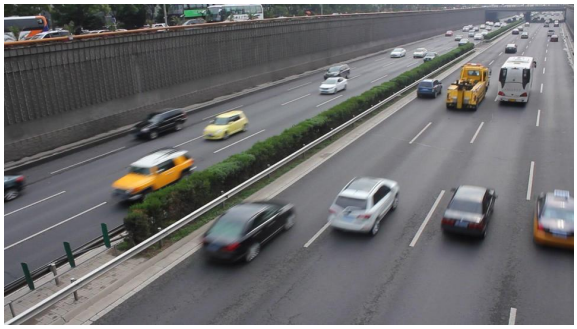
(a) `MVI_40201` (validation subset)



(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 dataset used for the validation and testing subsets.

Chapter 4

Object Detection Models

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 the current 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).

When training the standard-sized models, pre-trained models were downloaded from the repository of the MMYOLO library and used as the starting point to utilize the transfer learning technique. Similarly, a pre-trained MobileNetV2 model from the MMClasification (recently renamed to MMPretrain) repository was used when training the YOLOv8 MobileNetV2. The transfer learning technique was not employed when training the rest of the models — YOLOv8-pico and YOLOv8-femto.

In the YOLOv8 MobileNetV2 model, the output indices[1] selected were $[2, 4, 6]$. Several models with different `out_indices` settings were trained for just 75 epochs to compare their performance, however, none of the alternative settings led to improved performance (in terms of both inference speed and accuracy).

Normally, a square resolution is used for the model's 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 resolution 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 resolution have to be multiples of 32, so some input resolutions are further from it than others.

For smaller models, several input resolutions were tested to provide insights into how a vehicle detection model can be optimized by decreasing the resolution of the input image.

[1] Output indices enable the selection of specific layers in the architecture, from which the output feature maps should be used. In this case, only these feature maps are then used as inputs to the object detector's neck.

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

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

Although the inference speed should be (approximately) directly proportional to the number of pixels in the input image, we decided to test it and also find the limit — how small can the model's input resolution be for it to start performing poorly on our datasets.

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

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

4.2 Model Configurations

This section provides an overview of configurations used to train the vehicle detectors. A configuration of an object detection model when using the MMDetection or the MMYOLO library contains a set of parameters that define the 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-medium[2], like:

Optimizer: Stochastic Gradient Descent with momentum 0.937 and weight decay 0.000 5

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, only two relevant parameters were changed: the learning rate (adjusted individually for different models) and the batch

[2]The default YOLOv8-medium 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 resolution	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.

size, which was set to the highest possible for every trained model. For the smallest one, YOLOv8-femto with 352 × 192 input resolution, the largest batch size of 760 was used. Because training batch sizes above 128 usually result in lower model precision [36], models with large batch sizes were trained for 500 epochs instead of the default 300 epochs. Along with other model-specific training parameters, like the number of warmup epochs[3], 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 dataset wrappers, from which `dataloaders` (objects internally representing a dataset) are created. Because the datasets used in this project were in the COCO format, the `YOLOv5CocoDataset` wrapper was used for each of the six 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.

4.2.2 Training Augmentation Pipeline

In this subsection, the training augmentation pipeline is explained. Data augmentations are important when training an object detector, especially detectors of the YOLO family. An augmentation in this context refers to the process of applying a transformation 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

[3] During warmup epochs, the learning rate is gradually increased from a lower value to the target learning rate.

Dataset name	Images	Repetition factor	Images after over-sampling
DETRAC	99 771	1	99 771
Miovision	110 000	1	110 000
AAU	1 899	3	5 697
MTID	5 399	5	26 995
NDISPark	142	25	3 550
VisDrone	1 610	4	6 440

Table 4.3: Repetition factors for each dataset used for training, including the numbers of images before and after over-sampling.

distortion makes an input image appear to be something completely different, so applying these transformations randomly to the input images is crucial when training a robust model. Examples of images augmented by the training augmentation pipeline can be seen in Figure 4.1. Following are the transformations in the main training augmentation pipeline:

Resize

First, the image is resized to fit the model's input resolution 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).

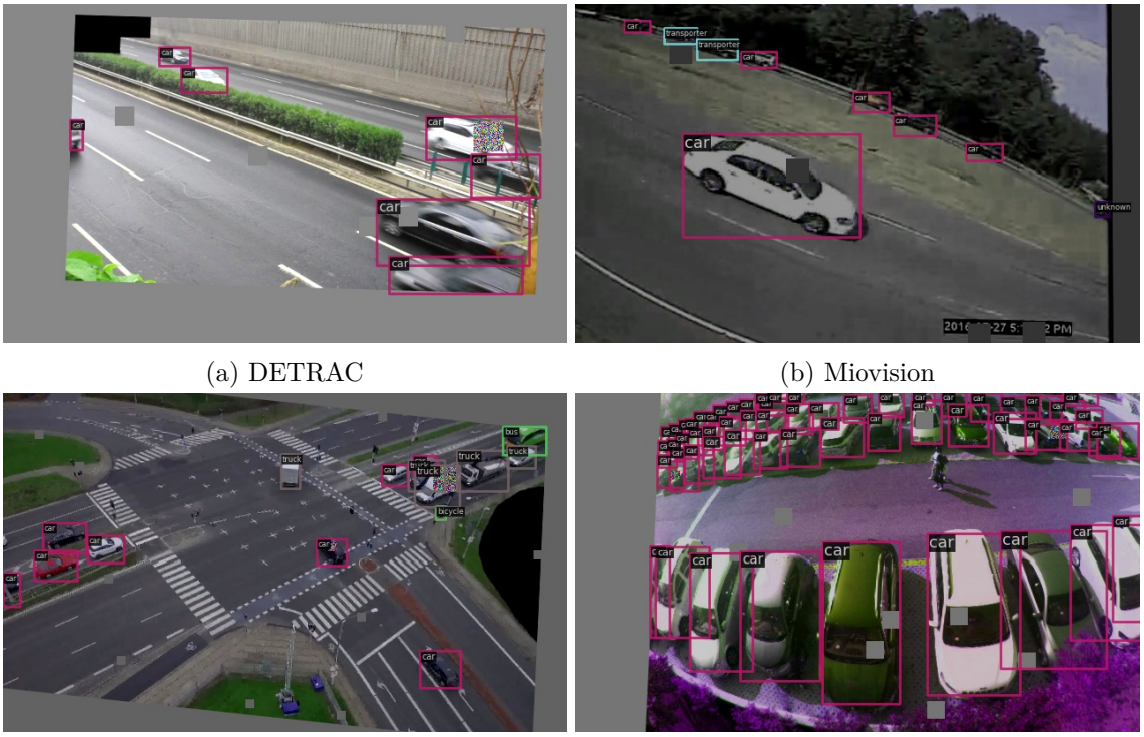
Random Affine

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
Miovision	1.0	1.1
AAU	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 the `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` transformation.



(a) DETRAC (b) Miovision

(c) MTID (d) NDISPark

Figure 4.1: Examples of images augmented by the training augmentation pipeline, including the ground truth labels.

Cut-Out

The `Cut-Out` transformation randomly selects regions of the image and fills them with 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
Miovision	4	26×26 px
AAU	8	10×10 px
NDISPark	12	20×20 px
MTID	12	10×10 px
VisDrone	20	8×8 px

Table 4.5: `Cut-Out` 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 `Cut-Out` 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 entire image. Additionally, 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 area of a cut-out region is 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.

The **probability** of a region being dropped from each bounding box is set to 0.05 (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 library provides the option to use its transformations in the data augmentation pipeline. Settings are left unchanged from the original YOLOv8-medium 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[4] to HSV[5], modifies the saturation, hue, converts from HSV to BGR, 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 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

[4]BGR (Blue, Green, Red) only differs from the well known RGB format by the order of its channels. It is widely used in image processing for legacy and compatibility reasons.

[5]HSV (Hue, Saturation, Value) is a cylindrical-coordinate color model, where hue represents the color type, saturation refers to the color's intensity and value corresponds to brightness.

in the augmentation pipeline and for the last 10 training 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). Because the epoch with the highest validation mAP is selected for deployment, as a result, none of the evaluated models used the last 10 training epochs and this configuration was therefore not utilized.

Chapter 5

Experiments

This chapter provides details about future experiments, including the benchmarked devices, information about the deployment and model optimization processes, and the methodology employed for testing. Additionally, it offers important considerations to keep in mind before drawing conclusions from the results.

5.1 Devices Used in the Experiments

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 Section 2.7 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 selected:
Jetson AGX Xavier: 30 W power plan with 4 out of 8 cores active
Jetson Xavier NX: 20 W power plan with 4 out of 8 cores active
Jetson Nano: 10 W `MAXN` power plan with all 4 cores active

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 on this device, as will be pointed out in individual relevant experiments. Additionally, we were unable to perform any tests using the TensorRT library on this device because some of the packages required could not be installed on the system.

Name	All NVIDIA Jetson devices	NVIDIA MX150	Intel Core i7-9850H	Raspberry Pi 4B
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.

5.1.3 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 (FPS) of these first 100 samples will be ignored when calculating the average FPS.

However, please note that the test results still not might be accurate enough and might depend on the underlying operating system and the environment.

5.1.4 Raspberry Pi 4B

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 (or would be too complex). 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.

5.2 Model Deployment and Optimizations

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

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

There are only two deploy configurations for the ONNX Runtime backend: one with a static model shape[2] and one with a dynamic shape in the batch size dimension[3]. The dynamic models were deployed to accept a maximum batch size of 32, but were optimized for a batch size of 1. 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 quantization to `INT8` including weight calibration using the validation dataset.

To optimize inference on the Raspberry Pi 4B, we aimed to use the NCNN inference framework designed for mobile and embedded devices with limited computing resources. However, the MMYOLO library does not yet support converting the YOLOv8 operations unsupported by NCNN to supported ones, so the NCNN framework was not used.

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[4] individually on each NVIDIA Jetson device, because they are platform-specific and transferring them across different devices is not recommended.

Training Checkpoints Used for Deployment

Checkpoint files, which can be selected for deployment, are saved periodically during training. When deploying, usually the checkpoint with the highest validation mAP is selected, which is particularly useful if the mAP starts decreasing during training.

As mentioned in Section 4.2, the mAP usually decreased during the last 10 epochs, but at times — most noticeably for YOLOv8-medium and YOLOv8-small — it started decreasing even sooner, probably due to overfitting. Because it sometimes stopped increasing (or started slowly decreasing) while other mAP metrics (mainly the $\text{mAP}^{\text{small}}$), were still rising, some checkpoints to use for deployment were selected manually for a trade-off between negligibly lower mAP (at worst) and higher, for example, $\text{mAP}^{\text{small}}$. The checkpoints used for deployment, including their mAP values are shown in Table 5.2.

We imagined the overfitting of YOLOv8-medium and YOLOv8-small models might be due to a low learning rate (of 0.00125), so we also tried training the YOLOv8-medium with the default learning rate (from the original MMYOLO configuration) of 0.01. However, this proved unsuccessful as the validation mAP of the model with the higher learning rate didn't come close to the one of the original model, although it was rising more consistently.

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`, weight calibration was done using the validation dataset.

[2] A model shape can be static or dynamic. A static model can only receive inputs of a certain shape specified when deploying (batch size and input resolution — width and height), whereas a dynamic model accepts various input shapes.

[3] A dynamic model shape in the batch size dimension means the model accepts different input shapes but still requires a fixed input image resolution.

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

Model (YOLOv8)	Input resolution	Epoch with best mAP	mAP of best epoch (%)	Manually selected epoch	mAP of selected (%)
medium	640 × 384	75	62.6	–	–
small	640 × 384	55	59.4	–	–
MobileNetV2	512 × 288	289	57.2	–	–
nano	640 × 384	235	55.1	–	–
	512 × 288	280	54.6	290	54.6
	448 × 256	289	55.3	–	–
pico	512 × 288	470	48.5	–	–
	448 × 256	440	46.4	450	46.4
	384 × 224	470	45.7	490	45.7
femto	512 × 288	460	36.0	–	–
	448 × 256	492	35.8	–	–
	384 × 224	460	33.7	490	33.7
	352 × 192	410	30.2	–	–

Table 5.2: Epoch numbers and validation mAP for best epochs and manually selected epochs (if applicable).

Although the ONNX Runtime framework 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` would be beneficial, the quantization to `FP16` would probably have a little effect on performance on CPUs as they do not usually support operations between numbers in this representation 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` was not performed on this device.

5.3 Experiments and Evaluation

In this section, we discuss additional details about all performed experiments, including how the underlying measurements were taken and important things to note before drawing conclusions from the results, which will be presented in Chapter 6.

All experiments will analyze the data collected by testing. Testing was done using a Python script included as a tool in the MMDeploy library, `test.py`, which uses the library to create a wrapper for the deployed model. It then uses the test dataset to evaluate the model's performance, calculating the inference speed in FPS and six mAP metrics as explained in Section 2.5. The inference speed is calculated including the input preprocessing and non-maximum suppression[5] (NMS).

To test all possible combinations of models, backends, deployment configurations and batch sizes on individual devices automatically, a new Python script `test_all.py` was developed, which uses the MMDeploy's `test.py` script and executes it for each possible test combination (unless specified differently by the user).

The MMDeploy's `test.py` saves the test logs, from which the mAP metrics and inference speeds can be read. This is also done automatically by our `collect_test_results.py`

[5]Non-maximum suppression in object detection is a method that selects a single prediction bounding box out of several overlapping bounding boxes, which are likely generated for the same object.

Python script, which reads all available test logs to output all metrics structured in JSON format to a file. Although the `test.py` script also outputs the final test metrics in JSON format to a folder, the folder is named after the test start time (eg. `20230429_043829/`) and it cannot be overridden, so reading the data from the log file is simpler and less error-prone.

Number of Threads When Performing Inference on CPUs

When testing models in the ONNX format on CPUs, the MMDeploy library doesn't control how many processes the ONNX Runtime backend uses to run inference. To provide accurate and reproducible results, the source code of the library was modified to create the ONNX Runtime inference session with the option of only running in a single thread. This means that the inference speeds on CPUs are not the highest possible. Please note that simply multiplying the inference speed by the number of CPU cores to get the maximum FPS possible on the device is generally incorrect, because running a multi-threaded inference is not as efficient as running it on a single thread. On the other hand, when n separate inference processes on a CPU with n cores are used, it is theoretically possible to achieve inference speeds n times higher than with a single-threaded inference, but the duration of the inference for a single input would remain the same (at best).

Chapter 6

Results

This chapter presents the results of the conducted experiments and provides a detailed analysis to explain the insights that can be gained from these findings. Thanks to the extensive data collected from the tested devices, we can identify trends, strengths, and limitations of each model and hardware combination, as well as highlight potential areas for future research. Of course, complete data can be rarely displayed in a single figure, so in these experiments, we typically only focus on the most important or representative sets of data, but include additional data in the Appendix, which will also be referenced from individual experiments.

The chapter begins by evaluating the trained models in terms of precision and recall through precision-recall curves. We then examine several factors that could potentially impact the performance of the tested models, namely input resolution, inference backends, model shape (static or dynamic), and batch sizes during inference. Subsequently, we benchmark the models in terms of their inference speeds and various mean Average Precision metrics. Lastly, we present the inference speeds of some of the trained models for comparison across all six tested devices.

It is important to note that the mAP values measured on NVIDIA Jetson platforms (where the models were deployed individually on each device) exhibit minor differences. Out of 930 metrics collected from each device, 276 varied across the platforms. To analyze whether these deviations were significant, we calculated the median, mean and maximum values of the deviations. The median was found to be 0.1 %, the mean was calculated to be 0.1413 % and the maximum, with a total of 12 occurrences, was 0.4 %. We therefore consider these deviations insignificant and don't consider them in the following experiments.

6.1 Precision-Recall Curves of Major Models

In this experiment, we measured the precision and recall values of four of the trained models on the test dataset to plot the precision-recall (PR) curves in Figure 6.1. We could not show the curves of all the models because the figure would simply not fit on a single page, so instead, only the four most representative models were selected. A more complete figure with PR curves for the rest of the models can be seen in Appendix A. The precision and recall values were calculated by MMDeploy's test script, to which we have inserted a few lines of code to save the values to a file.

In the plots, the performance for the bus class appears abnormally high. We hypothesize that the reason behind it is the presence of a few easily-detectable, large buses in numerous

images in the reannotated DETRAC dataset within the testing set. Additionally, the low performance of models in detecting vehicles belonging to the *unknown* category should not be a major concern, as the diversity of the vehicles in this category makes it difficult to accurately detect and classify them.

As could be expected, the YOLOv8-medium model demonstrated superior performance, achieving the highest precision and recall values. Furthermore, the performance of the YOLOv8 MobileNetV2 model is comparable to that of the YOLOv8-nano, although slightly superior. In contrast, the YOLOv8-femto with an input resolution of 352×192 could be an example of a model with considerably low performance.

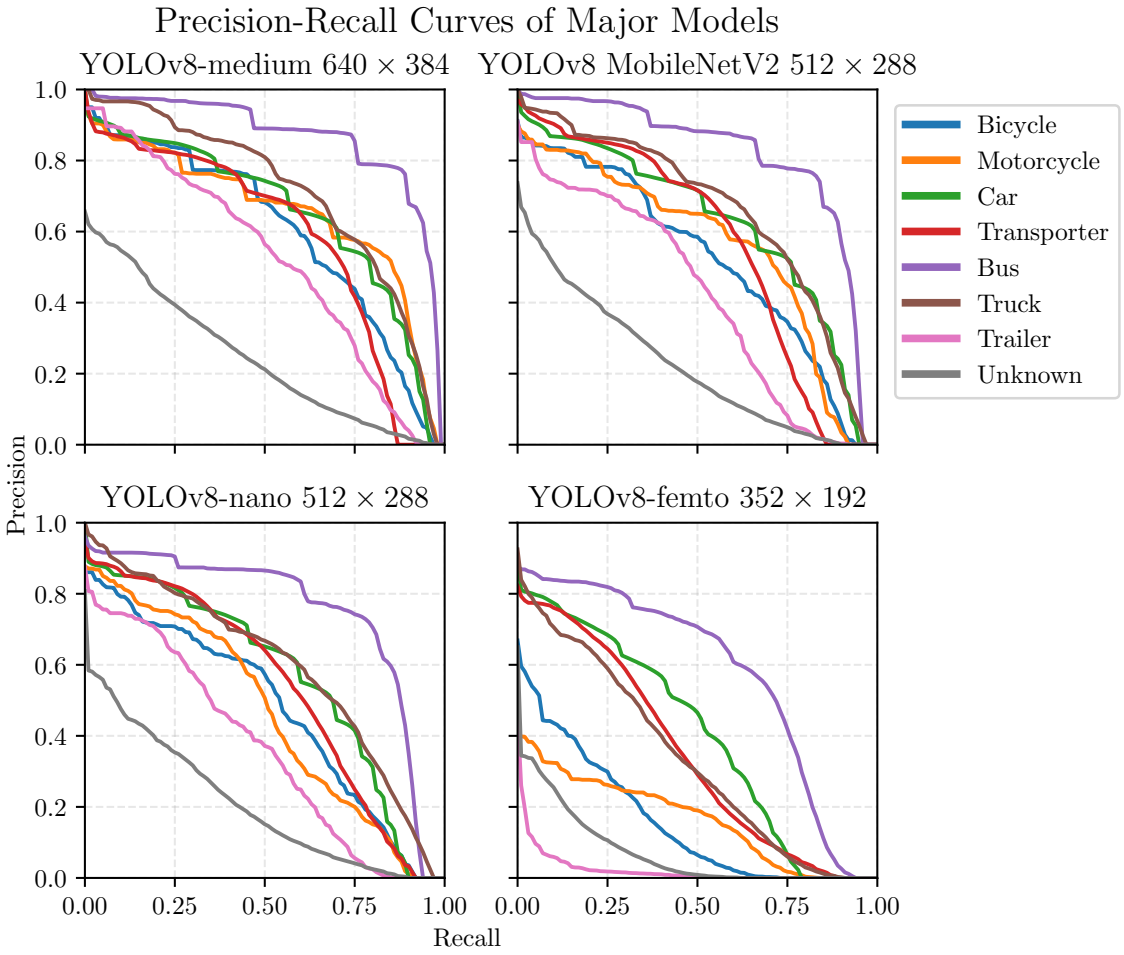


Figure 6.1: Precision-Recall curves are displayed for four major models, including YOLOv8-femto with the smallest input resolution of 352×192. It is important to note that, in this figure, the YOLOv8-nano model chosen for comparison has an input resolution of 512×288, although most experiments used the model with an input resolution of 640×384. This decision was made to simplify the comparison between YOLOv8-nano and YOLOv8 MobileNetV2. The measurements were taken utilizing the ONNX Runtime backend.

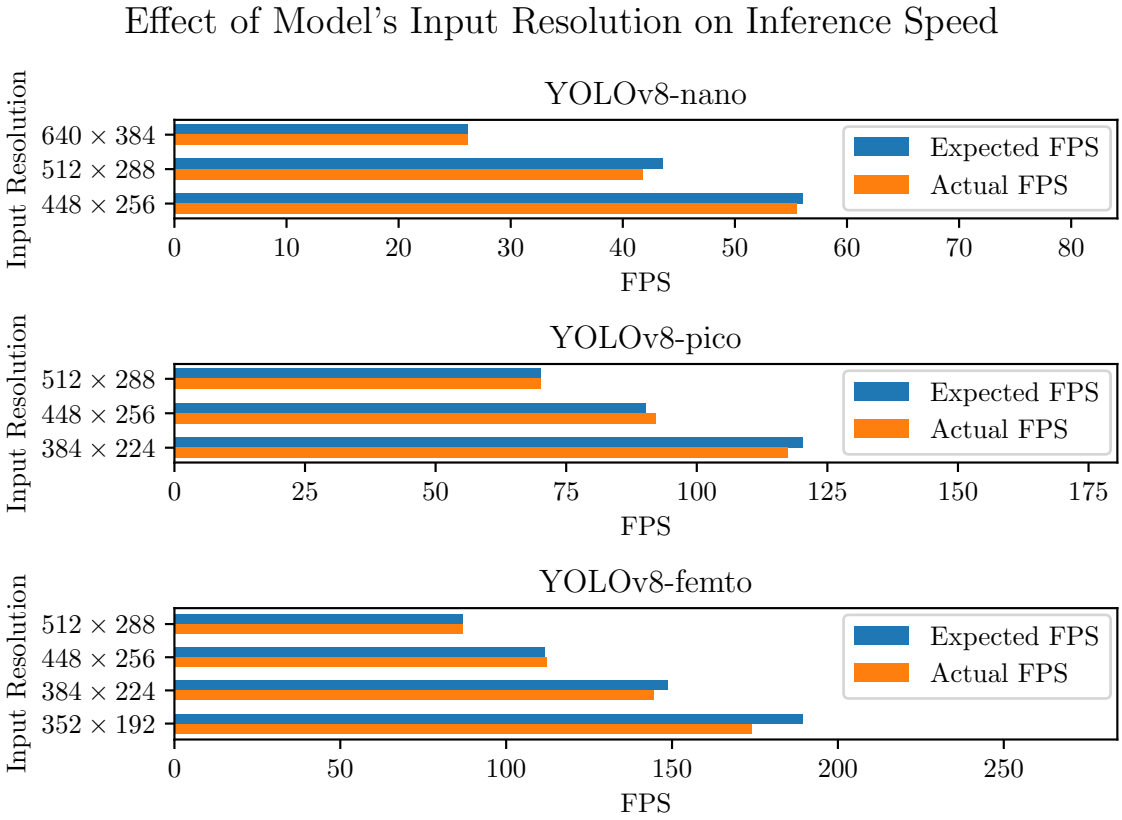


Figure 6.2: Inference speed (FPS) comparison of YOLOv8-nano, YOLOv8-pico, and YOLOv8-femto models with different input resolutions. The results generally support our hypothesis that the reduction in model's input resolution is directly proportional to the increase in inference speed, although with some deviations. These results were obtained by running tests on NVIDIA Jetson Nano using the TensorRT backend. The tested models were deployed with dynamic shape in the batch size dimension to run the tests with a batch size of 16 to most accurately measure the speeds of the smallest models.

6.2 Effect of Model's Input Resolution on Inference Speed

Smaller models — YOLOv8-nano, YOLOv8-pico and YOLOv8-femto — were trained with different input resolutions to understand how it affects the model's performance on our test dataset.

We expect that the reduction in the number of pixels in a model's input resolution should be directly proportional to the increase in its inference speed (FPS). For example, we anticipate that a detector with an input resolution of 640×320 would be twice as fast compared to a detector with an input resolution 640×640. Figure 6.2 compares the inference speeds of models with different input resolutions. The biggest input resolution is selected as the base to which other input resolutions will be compared, and the expected FPS is calculated using this simple equation:

$$\text{Expected FPS} = \text{Base FPS} \times \frac{\text{Base input width} \times \text{Base input height}}{\text{Compared input width} \times \text{Compared input height}} \tag{6.1}$$

While the results show that the relationship between input resolution and FPS cannot be exactly captured by the above equation, they generally supported our hypothesis. Following this experiment, we will suppose that the input resolution reduction is indeed directly proportional to the inference speed increase, and in some of the following experiments, we will only compare one input resolution for each model to make the results more informative and concise.

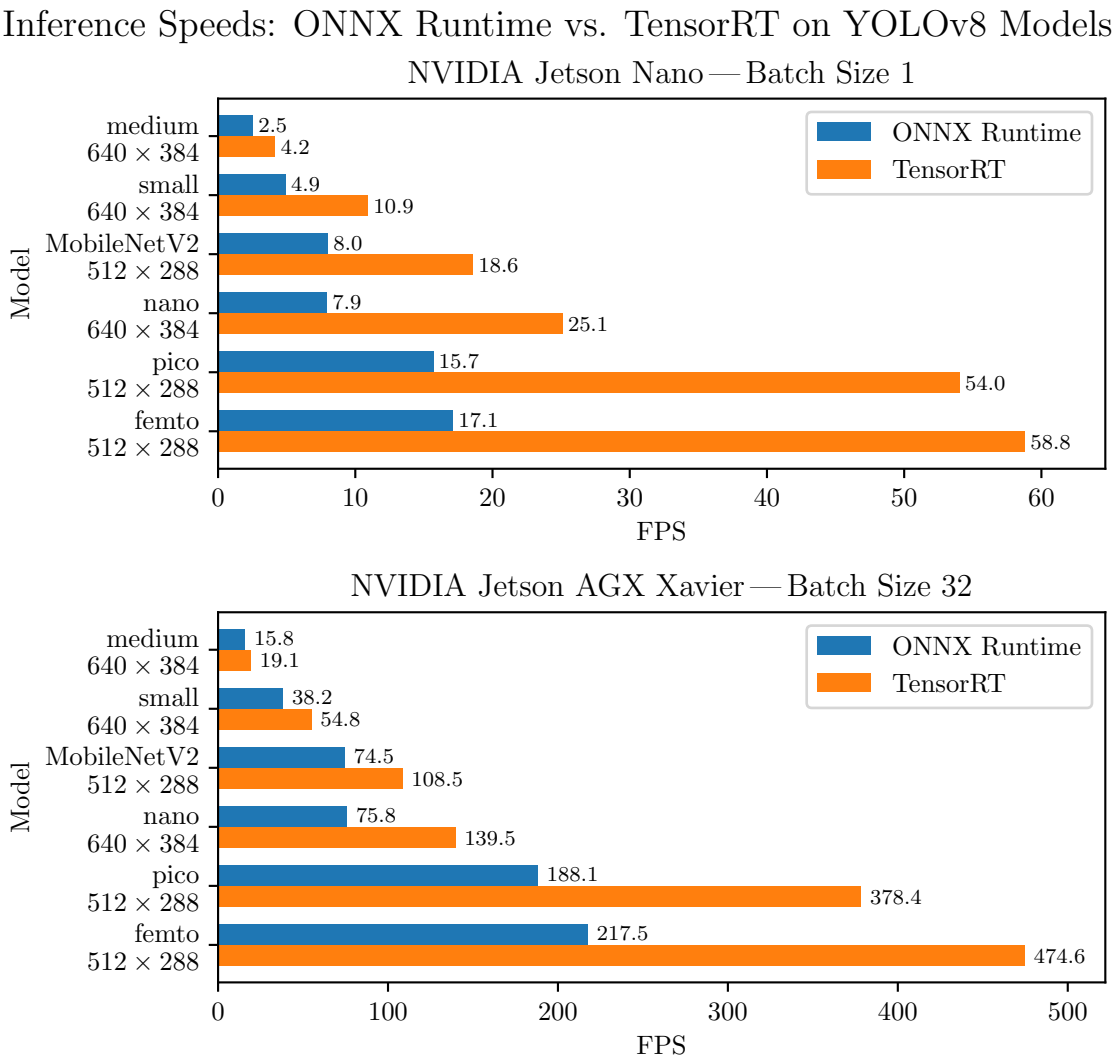


Figure 6.3: Comparison of inference speeds of all trained YOLOv8 model architectures between the TensorRT and the ONNX Runtime inference backends (of course, both utilizing the devices' GPU). Tests were conducted on NVIDIA Jetson Nano with a batch size of 1 and on NVIDIA Jetson AGX Xavier with a batch size of 32. Please note that for each model architecture, only the model with the largest input resolution was tested and all tested models were deployed as dynamic in the batch size dimension.

6.3 Inference Speeds: ONNX Runtime vs. TensorRT

This experiment studies how the inference speed is increased by using the TensorRT backend for inference on NVIDIA Jetson devices instead of the ONNX Runtime backend. The inference speeds (FPS) measured by the test script are compared on both the lowest and the highest-performing NVIDIA Jetson devices with different batch sizes to be able to draw general conclusions from the results. For each model architecture, only the highest input resolution is selected for testing, following the insights gained by the experiment in Section 6.2. Additionally, all tested models were dynamic in shape. The results of this experiment are shown in Figure 6.3.

The results clearly show the superiority of the TensorRT inference backend on NVIDIA Jetson devices for all tested models and different batch sizes. Naturally, the choice of the inference backend has no effect on the mean Average Precision of the tested models. Therefore, none of the following experiments will feature the ONNX Runtime backend on NVIDIA Jetson devices.

6.4 Effect of Model's Shape on Inference Speed

While a model with a static shape has a fixed input resolution and a fixed batch size, it can be set to accept input images of different resolutions and different batch sizes. This paper utilizes the MMDeploy library to deploy the trained models into both static and dynamic shapes. However, the deployment process was configured so that dynamic models would only be flexible in the batch size dimension while the model's input resolution stays static. Anyways, we will call these models dynamic.

In this experiment, we compare inference speeds achieved by static models and dynamic models to learn whether deploying a model to a dynamic shape (in the batch size dimension) decreases the inference speed. Only the most representative models were selected for comparison to keep the plots concise. Naturally, all tests were conducted with a batch size of 1. Tests were conducted on the NVIDIA Jetson Nano with the TensorRT inference backend and on the Raspberry Pi 4B with ONNX Runtime to make the measurements as accurate as possible by using the least-performing devices for both inference backends. The comparison of inference speeds on all model architectures is presented in Figure 6.4.

Because the inference speed is not consistently faster for static models, we conclude that it does not significantly decrease if a dynamic shape is used for the model's input in the batch size dimension. Therefore, we will only evaluate dynamic models in future experiments.

6.5 Effect of Batch Size on Inference Speed

An object detector can achieve higher inference speeds (FPS) if a larger batch size is used — if the detector is given more than one frame in a single input. Although rarely suitable for real-time object detection, a larger batch size enables more efficient utilization of GPU resources at the cost of higher memory usage. This experiment investigates how the inference speed increases with larger batch sizes used when performing inference with different models and on different devices.

In Figure 6.5 (please note the logarithmic horizontal axis), results of tests on Raspberry Pi 4B, NVIDIA Jetson Nano and NVIDIA Jetson AGX Xavier are shown. Although tests

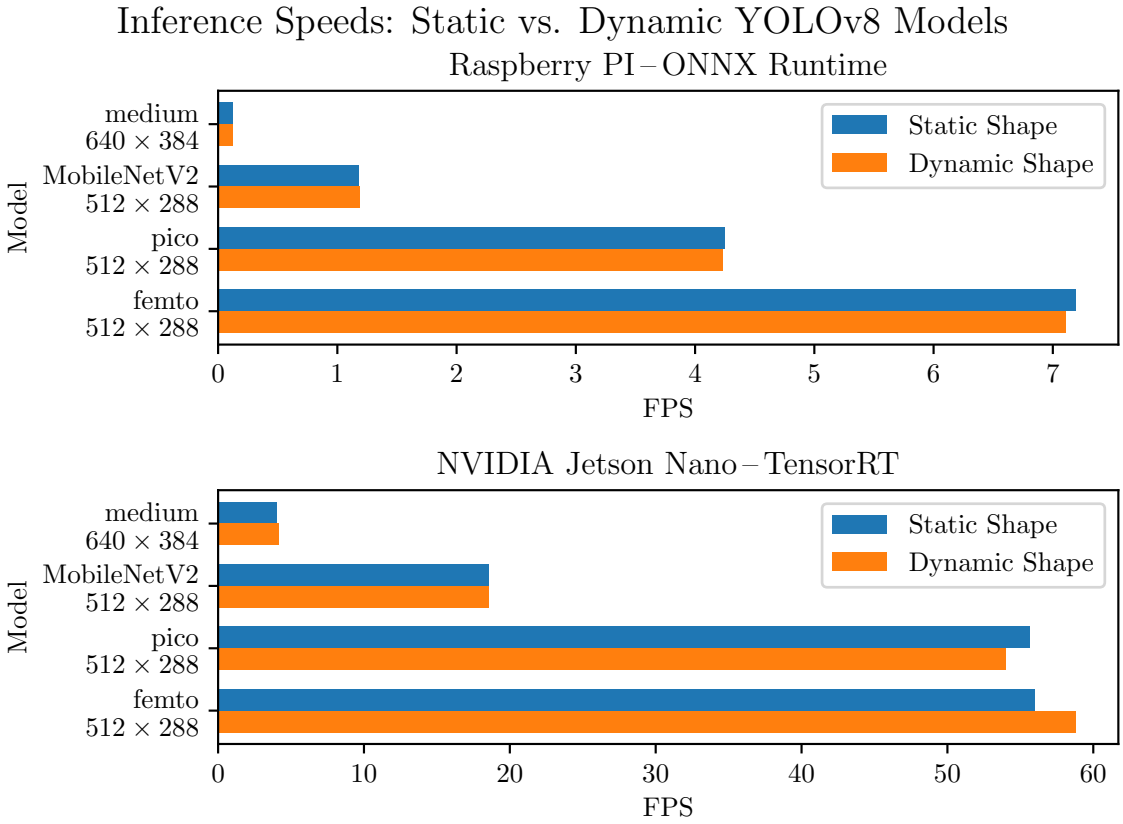


Figure 6.4: Comparison of inference speeds between models with static and dynamic shapes in the batch size dimension. Only the most representative models were selected for comparison, as the rest of the data do not provide additional insights.

were run with six different batch sizes, to make the figure more compact, only batch sizes 1, 2, 8 and 32 were selected for comparison.

The results show that using a larger batch size for inference does indeed often increase the inference speed, but the increase is generally only significant when performing inference on smaller models, where the inference speed is higher. This is well illustrated by the NVIDIA Jetson AGX Xavier, where the increase in inference speed with larger batch sizes is relatively subtle for the largest detector, YOLOv8-medium 640 × 384. In contrast, for the smallest model, YOLOv8-femto 352 × 192, raising the batch size from 1 to 32 results in an almost ten-fold increase in inference speed. Therefore, using larger batch sizes is generally only suitable in cases where high inference speeds (higher than real-time) are desired, for example in a multi-camera real-time vehicle detection system.

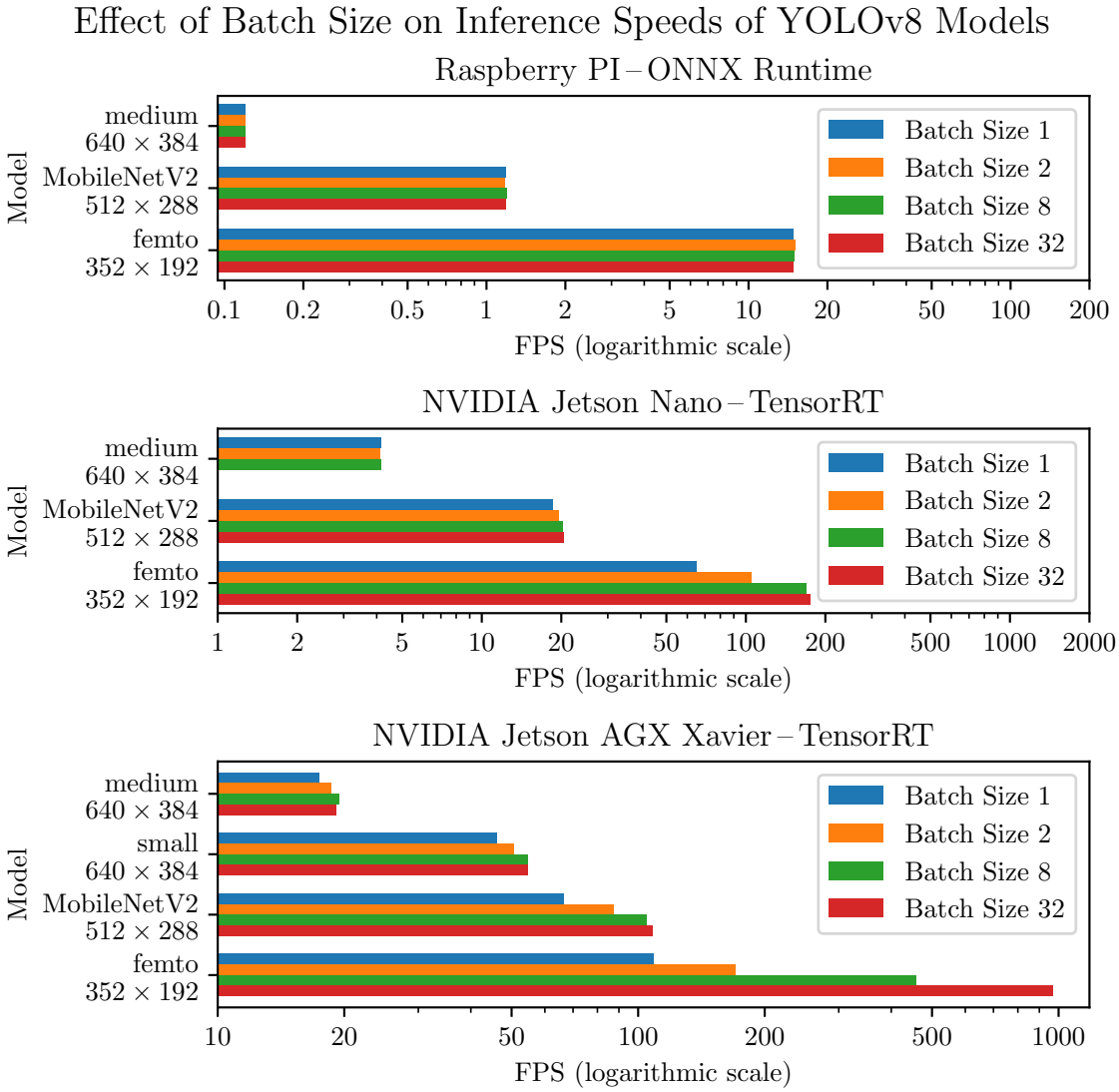


Figure 6.5: Inference speeds achieved by using different batch sizes. Tests were conducted on Raspberry Pi 4B, NVIDIA Jetson Nano and NVIDIA Jetson AGX Xavier and only the most representative models were selected for comparison to keep the figure compact. Please note that the horizontal axes are logarithmic to better present the differences in inference speeds. Additionally, inference speed of YOLOv8-medium 640×384 with the batch size of 32 on NVIDIA Jetson Nano is missing because the model did not fit into the memory.

6.6 Mean Average Precision and Inference Speed Benchmark

In this experiment, we compare the trained and quantized models in terms of their mAP metric (mean Average Precision) and their inference speed in FPS.

For the tests, we wanted to choose two least-performing devices, with the Raspberry Pi 4B being an obvious choice. Additionally, the NVIDIA Jetson Nano would be a great choice, but since the device is not optimized for operations with numbers in the `INT8` representation, NVIDIA Jetson Xavier NX was selected instead. For the tests to be as representative as possible even for the smallest models, tests on Jetson Xavier NX were

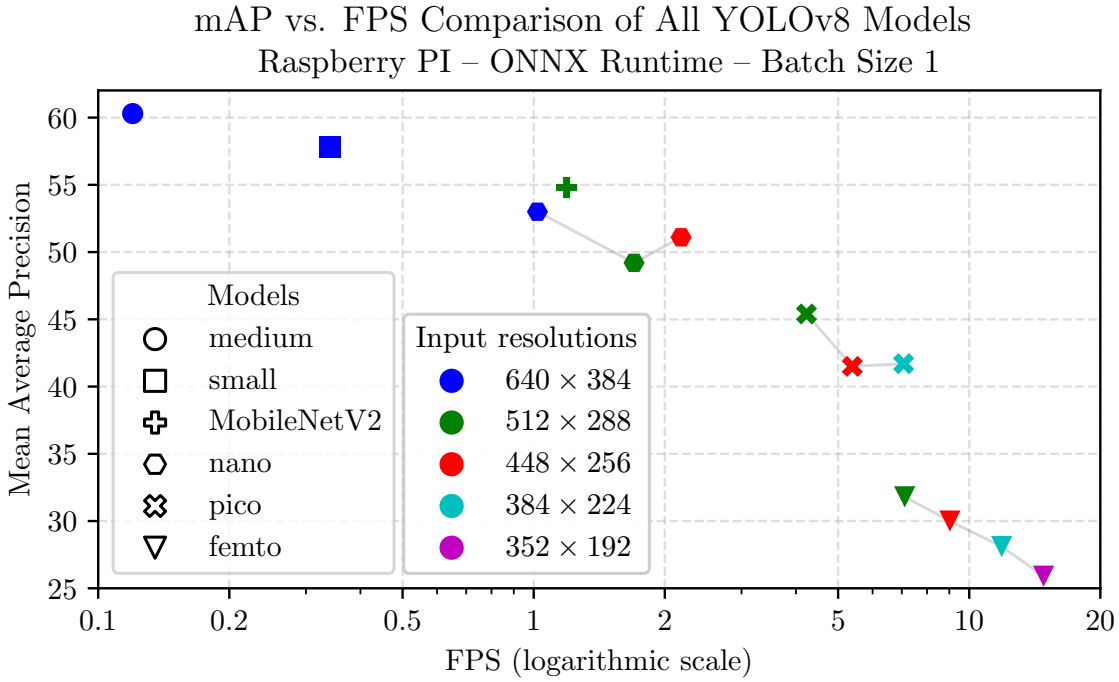


Figure 6.6: Performance comparison of all YOLOv8 models in terms of their mAP (mean Average Precision) and inference speed (FPS) on a Raspberry Pi 4B with ONNX Runtime inference backend and the batch size of 1. Please note that the x-axis is logarithmic to be able to display all models in a single plot for easier analysis. We also remind the reader that only a single CPU thread was used when testing models on this device.

run with a large batch size of 32, while on the Raspberry Pi, where the batch size doesn't matter, a batch size of 1 was used instead.

Therefore two plots are presented, both comparing the mAP and FPS of all models. On the Raspberry Pi 4B, ONNX Runtime backend was used to test the models on the CPU. For the NVIDIA Jetson Xavier NX with TensorRT backend, the plot also features results of models quantized to both `FP16` and `INT8` representations and these models are connected in the plot. The mAP and FPS comparison for the Raspberry Pi can be found in Figure 6.6, while the corresponding results for the Jetson Xavier NX including the quantized models are shown in Figure 6.7. Please note that in both plots, a logarithmic scale was chosen for the x-axis (FPS) to include all models (large and tiny) in the figure. For a complete benchmark of all trained and quantized models on all devices, see Appendix B.

An important thing to notice is the effect of quantization on the mAP and FPS. Although post-training quantization to `INT8` rarely seems to result in better performance than training a smaller model, it is clear that quantizing to `FP16` results in a significantly higher inference speed with no decrease in accuracy.

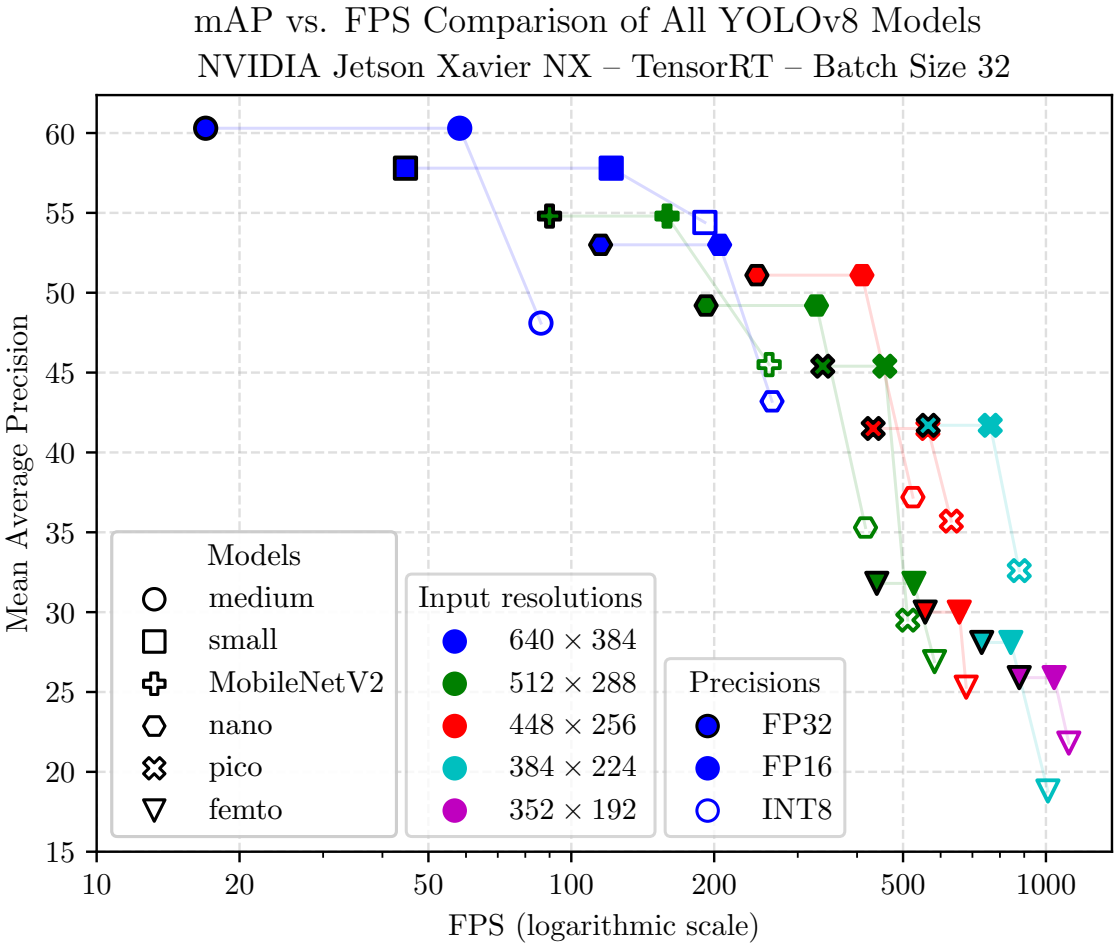


Figure 6.7: Performance comparison of all YOLOv8 models, including quantized ones, in terms of their mAP (mean Average Precision) and inference speed (FPS) on NVIDIA Jetson Xavier NX with TensorRT inference backend and the batch size of 32. Please note that the x-axis is logarithmic to be able to display all models in a single plot for easier analysis.

The performance of the YOLOv8 MobileNetV2 model can also be discussed here, as tests on the Raspberry Pi show the superiority of the YOLOv8 model with the MobileNetV2 backbone compared to the YOLOv8-nano models. However, tests conducted on the NVIDIA Jetson Xavier NX with the TensorRT backend show that the opposite is true and the MobileNetV2 brings little to no advantage when compared to YOLOv8-small or YOLOv8-nano models on this device. To draw further conclusions about this matter, the reader is encouraged to look at the tables in Appendix B, which present the complete mAP and FPS values. Additionally, all 6 COCO mAP metrics of different models are compared in the next experiment.

In both plots resulting from this experiment, a strange phenomenon can be seen in which the mean Average Precision of a model with a higher input resolution (YOLOv8-nano 512×288) is lower than the one of a model with a lower input resolution (YOLOv8-nano 448×256). The same seems to apply for YOLOv8-pico models, where the mAP with input resolution 448×256 is lower (although not as significantly) than with 384×224, instead of being higher. However, this problem doesn't seem to affect the YOLOv8-femto models, of

Model (YOLOv8)	Input resolution	mAP (%)					
		mAP	IoU:50	IoU:75	small	medium	large
medium	640 × 384	60.3	80.5	69.5	31.6	60.6	77.0
small	640 × 384	57.8	78.6	66.0	27.3	58.5	74.4
MobileNetV2	512 × 288	54.8	75.9	61.8	23.5	55.0	71.5
nano	640 × 384	53.0	74.9	60.8	22.9	52.9	68.5
	512 × 288	49.2	69.9	56.9	21.0	48.3	67.3
	448 × 256	51.1	72.3	59.2	20.6	50.8	68.1
pico	512 × 288	45.4	66.4	50.2	15.1	44.8	61.4
	448 × 256	41.5	62.1	46.8	13.6	40.8	55.2
	384 × 224	41.7	61.5	46.5	12.8	40.5	58.9
femto	512 × 288	31.8	48.8	35.1	7.2	29.8	41.6
	448 × 256	30.0	47.7	31.8	7.0	25.9	42.9
	384 × 224	28.1	45.2	30.2	7.0	26.3	38.6
	352 × 192	25.9	43.0	27.5	5.4	22.7	37.0

Table 6.1: Comparison of multiple mean Average Precision metrics for each trained model and input resolution. The mAP metrics displayed in this table are explained in Section 2.5.

which the mAP and FPS values are as expected. We tried examining the problem, mainly by double-checking the results and training configurations of these models, but could not find the root cause and did not have enough resources to conduct further experiments to provide more insights into this phenomenon.

6.7 Evaluating Models Across Multiple Mean Average Precision Metrics

In other experiments, we compare the performance of models just by using a single metric — the mean Average Precision (mAP) — for objects of all sizes and over ten IoU thresholds ranging from 0.50 to 0.95. However, models can also be compared using five additional, more specific mAP metrics as explained in Section 2.5: mAP with the IoU threshold equal to 0.50, with IoU threshold 0.75, and mAP for small, medium-sized or large objects.

In Table 6.1, we display these metrics for all trained models. However, we omit the quantized models from the table because the mAP values tend to stay the same after reducing precision to `FP16` and quantizing to `INT8` is rarely beneficial, as illustrated by the experiment in Section 6.6. For the complete table featuring all precisions of all models, see Appendix C.

It can be observed in the results that smaller models detect large and even medium-sized objects reasonably well. However, when it comes to detecting small objects (of area lower than 32×32 px), their performance drops significantly, even at higher input resolutions. However, the smaller models might still be highly suitable for cases in which detecting small objects is not a priority, as they generally perform well for larger objects while being dramatically faster.

6.8 Inference Speeds on Different Devices

In this experiment, we benchmark all available devices by measuring the inference speeds of all model architectures. For each architecture, only one input resolution was selected, however, differently than in the other experiments: YOLOv8-nano 512×288 was selected to better compare with YOLOv8 MobileNetV2, and YOLOv8-femto 352×192 was chosen to include the smallest model. A second set of measurements was created to only benchmark models quantized to the `FP16` representation on the NVIDIA Jetson devices. Both plots can be seen in Figure 6.8. Please note that the x-axis is in a logarithmic scale in both plots because of the performance differences between individual devices.

We decided to measure the inference speed with higher batch sizes to get the highest possible inference speeds possible, but since larger models do not fit into memory on lower-performing devices when a large batch size was used[1], we could not benchmark these models with the largest batch size of 32. However, thanks to insights provided by Section 6.5, we consider it safe to use a smaller batch size of 8 for YOLOv8-medium, YOLOv8-small and YOLOv8 MobileNetV2 on all devices. For the rest of the models, a batch size of 32 was used. This applies to both plots resulting from this experiment.

What's interesting to see from the first plot is that the NVIDIA Jetson Nano is comparable to the NVIDIA MX150, while consuming just a fraction of the power. Additionally, there is a significant difference between the performance of the Jetson Nano compared to Jetson Xavier NX, while the inference speed measured on Jetson AGX Xavier is only slightly higher than on Jetson Xavier NX.

Another notable difference lies in how weight quantization to the `FP16` number representation increases the inference speeds on NVIDIA Jetson devices. The highest difference in FPS between `FP32` and `FP16` on Jetson nano was for the YOLOv8-small model, where the inference speed after quantization increased by 50.13 % from 11.19 FPS to 16.79 FPS. On the other hand, the inference speed for the same model on Jetson Xavier NX increased by 165.84 % from 44.12 FPS to 117.29 FPS. Furthermore, the difference is even higher for YOLOv8-medium, where the quantization to `FP16` resulted in a 228.33 % increase in inference speed. However, it's important to note that for smaller models, these gains are lower and more comparable between the two devices.

For a complete table of inference speeds of all models (including quantized ones) on all devices with a batch size of 1, please see Table B.1 in the appendix.

[1] List of tests that failed because the model could not fit into the memory of a device when performing inference with a specific batch size: YOLOv8-medium on NVIDIA Jetson Nano with batch size 32, YOLOv8-medium on NVIDIA MX150 with batch sizes 16 and 32, YOLOv8-small on NVIDIA MX150 with batch size 32.

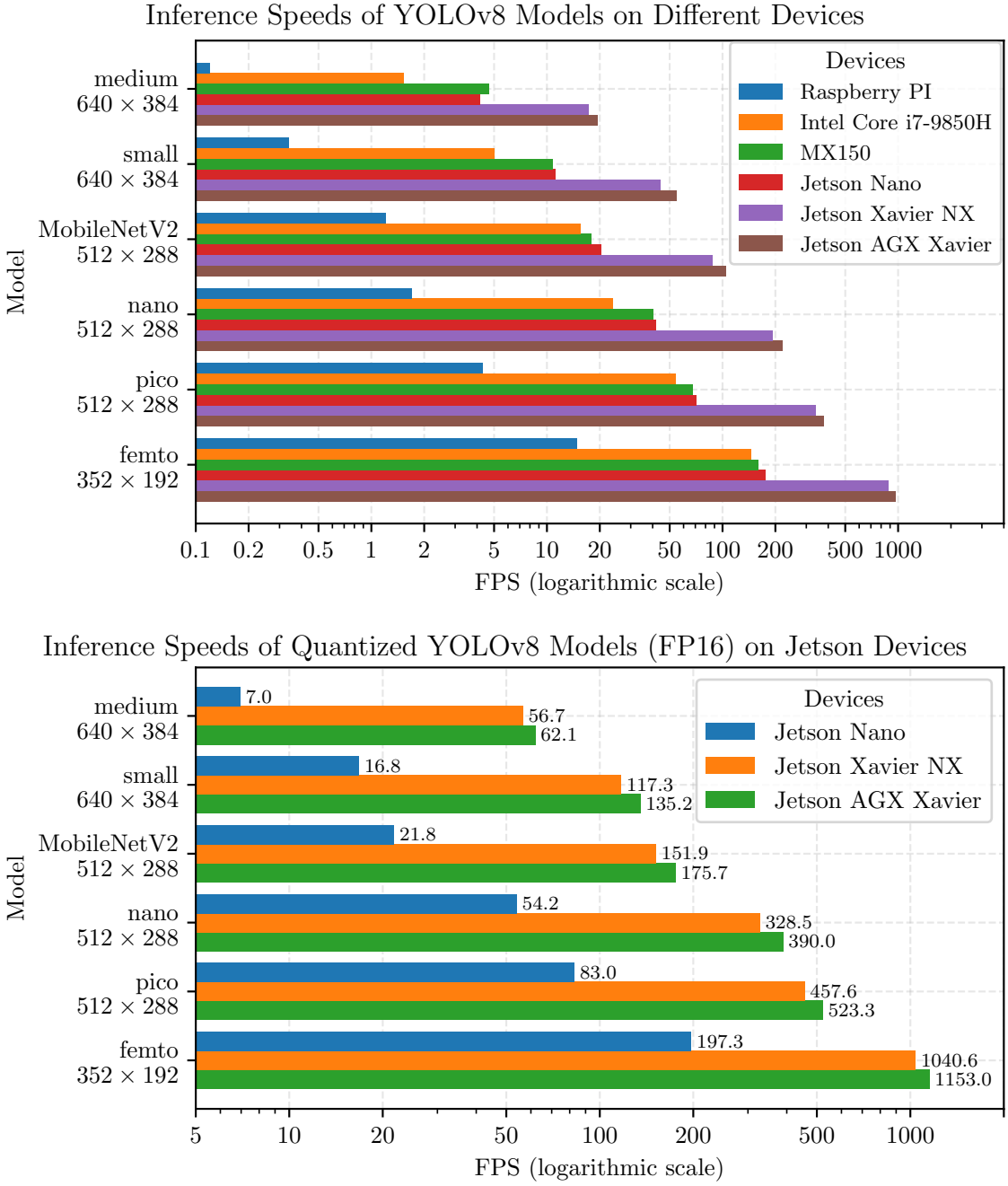


Figure 6.8: Benchmark of inference speeds (FPS) on different devices featuring all model architectures. The second plot compares models quantized to `FP16` precision on NVIDIA Jetson devices. Please note that in both subplots, the x-axis is logarithmic and that for comparison, YOLOv8-nano model with 512×288 input resolution and YOLOv8-femto model with 352×192 input resolution were selected. A batch size of 8 was selected for tests featuring larger models – YOLOv8-medium, YOLOv8-small and YOLOv8 MobileNetV2 — and for the rest of the models, a batch size of 32 was used. The TensorRT inference backend was utilized when testing on NVIDIA Jetson devices, while the ONNX Runtime backend was used on the rest of the devices. Additionally, we remind the reader that for the tests on CPUs, only a single thread was used, utilizing just one of the CPU's cores.

Chapter 7

Conclusion

In this paper, we have demonstrated the potential of the state-of-the-art YOLOv8 object detector for vehicle detection on various embedded devices, with a focus on optimizing the models for real-time performance. Our experiments involved training several models of different sizes on a large, diverse dataset of surveillance-type images.

We have shown that the entry-level embedded device NVIDIA Jetson Nano can run reasonably accurate vehicle detectors like the YOLOv8-nano in real-time. On higher-performing devices, namely the Jetson Xavier NX or Jetson AGX Xavier, even the largest of the trained models — YOLOv8-medium — operated at high frame rates when quantized to `FP16` precision. We found that on these devices, quantizing models to `FP16` results in higher inference speeds with no real impact on accuracy, while quantizing to `INT8` rarely brings an advantage over using a smaller model.

In contrast, the Raspberry Pi 4B can only run real-time inference of the smallest models, which exhibit inferior performance, particularly when detecting small objects. Therefore, vehicle detection on this device is only viable in special cases where the accuracy requirements are not as high and the vehicles present in the input images have a large area.

To compare these embedded devices with processing units commonly found in laptops or desktop computers, we also benchmarked the models on the Intel Core i7-9850H CPU and the NVIDIA MX150 GPU. Additionally, we studied the effects of other factors that influence the accuracy and inference speed of the models, including different input resolutions, batch sizes during inference, and the differences between the ONNX Runtime and TensorRT inference backends on NVIDIA Jetson devices.

In conclusion, our study successfully demonstrates the real-time capabilities of state-of-the-art vehicle detectors on various embedded devices, including the popular NVIDIA Jetson Nano and the low-performance Raspberry Pi 4B. Our research provides valuable insights for practical applications in the field of vehicle detection and establishes a foundation for future work aimed at refining object detection models to run on embedded systems.

Chapter 8

Future Work

In future studies, we suggest experimenting with additional network optimization and compression techniques, such as knowledge distillation and network pruning. These methods have the potential to improve the efficiency and accuracy of object detection models when deployed on low-performance devices.

Additionally, since this paper focuses on embedded devices specifically designed for machine learning, it is essential to investigate further optimizations to draw more comprehensive conclusions about vehicle detection on devices like the Raspberry Pi 4B. This includes examining the effect of weight quantization on CPU inference times and exploring the use of specialized inference frameworks such as NCNN, ARM NN, or TensorFlow Lite.

Furthermore, to enable a more extensive comparison with related research papers in the field, future studies may also consider benchmarking other object detection models, such as the Single Shot MultiBox Detector (SSD) and earlier YOLO versions like the YOLOX. This would also demonstrate how the recent advances in object detection contribute to the development of more efficient and accurate vehicle detection systems.

Appendices

Appendix A

Additional Precision-Recall Curves

Additional precision-recall curves, which were not included in Figure 6.1, are shown here, in Figure A.1 for completeness.

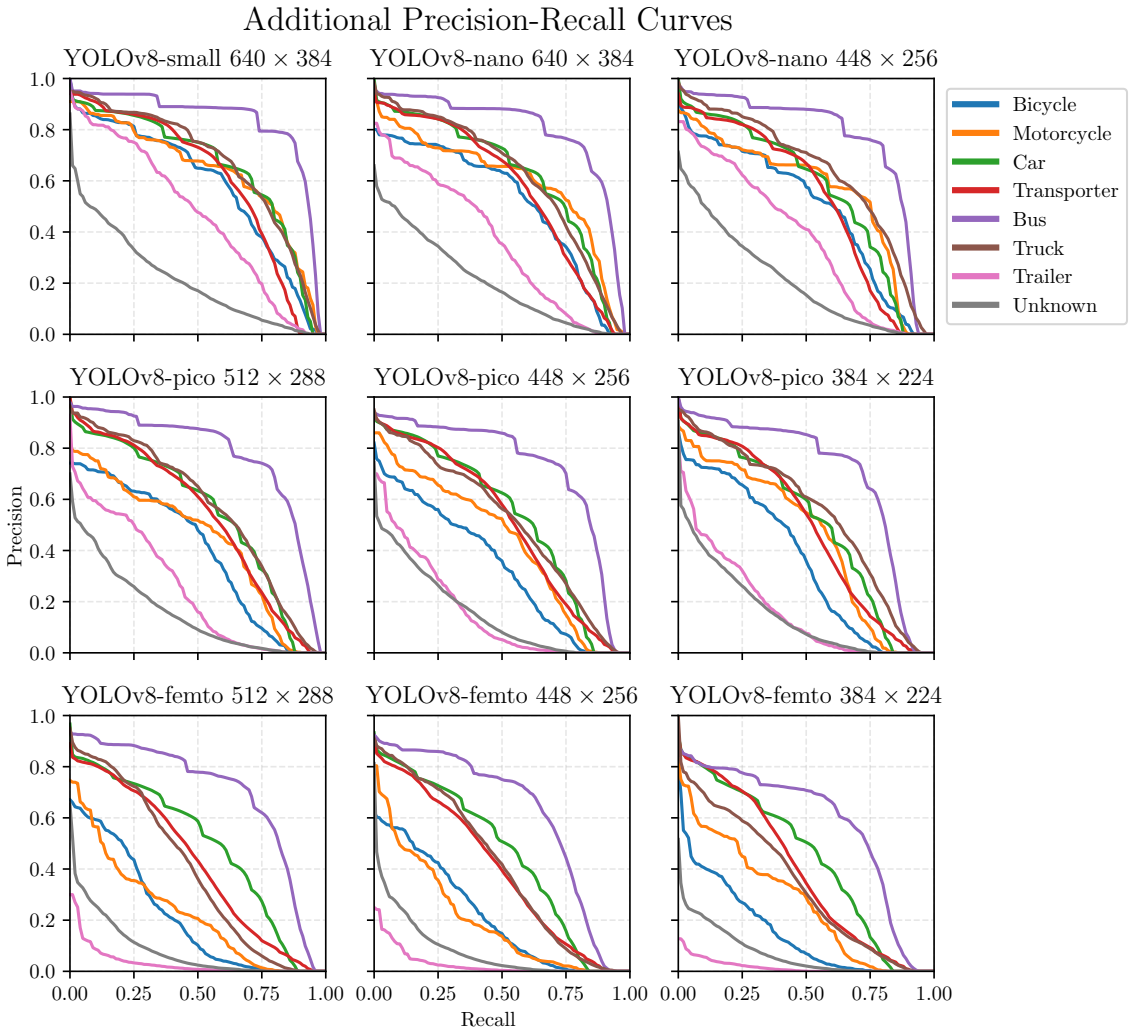


Figure A.1: Precision-Recall curves of the models which were not included in Figure 6.1.

Appendix B

Complete Benchmark of All Models on All Devices

In this appendix, the complete inference speeds of all trained and quantized models on all six devices utilized in this paper are provided. Two tables are shown — one for tests that were run with a batch size of 1 (Table B.1) and one with a batch size of 32 (Table B.2). However, not all models could be benchmarked on all devices with the large batch size, so for tests of YOLOv8-medium and YOLOv8-small on NVIDIA MX150 and YOLOv8-medium on NVIDIA Jetson Nano, a batch size of 8 was used instead. This, however, shouldn't significantly affect the values, as demonstrated by Section 6.5.

The tests were run using the TensorRT inference backend on NVIDIA Jetson devices, while for the rest of the devices, ONNX Runtime was used. Additionally, weight quantization was only performed using the TensorRT backend, while the `INT8` precision was not used on NVIDIA Jetson Nano. Furthermore, we remind the reader that on CPUs, only a single CPU thread was utilized when using the ONNX Runtime inference backend.

Model (YOLOv8)	Input Resolution	Precision	mAP (%)	Inference Speed with a batch size of 1 (FPS)					
				Raspberry Pi	Intel Core i7 9850H	NVIDIA MX150	NVIDIA Jetson		
							Nano	Xavier NX	AGX Xavier
medium	640 × 384	FP32	60.3	0.12	1.62	4.04	4.15	16.36	17.48
		FP16	60.3	–	–	–	6.82	47.49	49.29
		INT8	47.1	–	–	–	–	48.62	53.37
small	640 × 384	FP32	57.8	0.34	5.12	9.54	10.91	39.48	46.18
		FP16	57.8	–	–	–	16.27	52.01	57.44
		INT8	54.4	–	–	–	–	70.68	64.25
MobileNet V2	512 × 288	FP32	54.8	1.19	15.82	15.34	18.55	51.59	66.72
		FP16	54.8	–	–	–	19.93	53.30	58.66
		INT8	45.9	–	–	–	–	69.95	55.24
nano	640 × 384	FP32	53.0	1.02	14.78	20.40	25.08	54.30	57.50
		FP16	53.0	–	–	–	31.90	63.13	55.33
		INT8	43.0	–	–	–	–	59.78	64.21
	512 × 288	FP32	49.2	1.70	23.67	30.58	37.56	57.25	63.53
		FP16	49.2	–	–	–	48.29	70.01	71.38
		INT8	35.6	–	–	–	–	81.91	83.87
	448 × 256	FP32	51.1	2.18	29.55	36.21	49.25	67.73	58.17
		FP16	51.1	–	–	–	54.04	77.80	83.35
		INT8	37.2	–	–	–	–	61.81	91.60
pico	512 × 288	FP32	45.4	4.23	50.48	43.96	54.02	69.78	71.11
		FP16	45.4	–	–	–	57.14	59.80	87.29
		INT8	29.7	–	–	–	–	64.16	94.32
	448 × 256	FP32	41.5	5.39	61.38	52.89	57.37	78.47	81.29
		FP16	41.5	–	–	–	59.12	66.16	98.80
		INT8	35.6	–	–	–	–	71.28	101.98
	384 × 224	FP32	41.7	7.07	77.38	59.04	60.39	62.16	88.68
		FP16	41.8	–	–	–	61.85	77.94	108.55
		INT8	32.5	–	–	–	–	80.42	113.24
femto	512 × 288	FP32	31.8	7.11	62.40	55.16	58.78	55.81	84.13
		FP16	31.8	–	–	–	60.28	91.62	93.83
		INT8	30.0	–	–	–	–	65.07	97.91
	448 × 256	FP32	30.0	9.03	81.19	63.29	61.44	62.33	95.11
		FP16	30.0	–	–	–	58.12	70.06	104.58
		INT8	26.4	–	–	–	–	72.38	107.26
	384 × 224	FP32	28.1	11.87	103.49	76.33	66.01	72.29	103.47
		FP16	28.1	–	–	–	57.66	82.98	113.06
		INT8	23.5	–	–	–	–	83.54	115.56
	352 × 192	FP32	25.9	14.82	121.83	84.24	64.84	78.18	109.02
		FP16	25.9	–	–	–	65.91	89.41	120.63
		INT8	21.4	–	–	–	–	88.15	123.08

Table B.1: Benchmark of all trained and quantized YOLOv8 models on all available devices. For these tests, a batch size of 1 was used and all models were of a dynamic shape in the batch size dimension. Additionally, the TensorRT inference backend was used on NVIDIA Jetson devices, while the ONNX Runtime backend was used for tests on the rest of the devices. Models were only quantized to `FP16` precision on NVIDIA Jetson devices and `INT8` precision was only used on Jetson Xavier NX and Jetson AGX Xavier.

Model (YOLOv8)	Input Resolution	Precision	mAP (%)	Inference Speed with a batch size of 32* (FPS)					
				Raspberry Pi	Intel Core i7 9850H	NVIDIA MX150	NVIDIA Jetson		
							Nano	Xavier NX	AGX Xavier
medium	640 × 384	FP32	60.3	0.12	1.57	4.68*	4.15*	16.97	19.11
		FP16	60.3	–	–	–	6.96	58.20	63.91
		INT8	47.1	–	–	–	–	86.30	102.17
small	640 × 384	FP32	57.8	0.34	4.98	10.70*	11.23	44.73	54.78
		FP16	57.8	–	–	–	16.79	121.36	140.62
		INT8	54.4	–	–	–	–	191.18	206.82
MobileNet V2	512 × 288	FP32	54.8	1.19	15.46	17.17	20.36	89.98	108.48
		FP16	54.8	–	–	–	22.00	159.10	185.10
		INT8	45.9	–	–	–	–	261.10	290.35
nano	640 × 384	FP32	53.0	1.03	14.38	21.22	26.30	115.28	139.49
		FP16	53.0	–	–	–	33.69	205.36	251.42
		INT8	43.0	–	–	–	–	264.79	305.43
	512 × 288	FP32	49.2	1.70	23.71	40.13	41.86	192.31	219.17
		FP16	49.2	–	–	–	54.16	328.51	390.02
		INT8	35.6	–	–	–	–	416.75	466.65
	448 × 256	FP32	51.1	2.21	30.16	51.74	55.73	246.03	283.35
		FP16	51.1	–	–	–	71.49	409.57	483.80
		INT8	37.2	–	–	–	–	524.52	575.99
pico	512 × 288	FP32	45.4	4.30	53.92	67.29	70.51	338.69	378.36
		FP16	45.4	–	–	–	83.02	457.63	523.29
		INT8	29.7	–	–	–	–	511.57	596.65
	448 × 256	FP32	41.5	5.48	68.66	83.70	92.82	432.56	484.74
		FP16	41.5	–	–	–	107.79	563.96	641.45
		INT8	35.6	–	–	–	–	631.91	740.18
	384 × 224	FP32	41.7	7.18	89.89	105.58	118.47	564.88	648.23
		FP16	41.8	–	–	–	138.83	763.25	888.08
		INT8	32.5	–	–	–	–	878.58	1016.77
femto	512 × 288	FP32	31.8	7.10	70.84	85.43	87.42	440.21	474.64
		FP16	31.8	–	–	–	98.72	527.16	579.90
		INT8	30.0	–	–	–	–	582.56	589.16
	448 × 256	FP32	30.0	9.04	93.97	106.62	113.09	556.71	600.81
		FP16	30.0	–	–	–	127.27	656.74	682.84
		INT8	26.4	–	–	–	–	679.27	726.82
	384 × 224	FP32	28.1	11.92	125.96	133.62	145.79	732.24	813.68
		FP16	28.1	–	–	–	158.38	843.47	1004.03
		INT8	23.5	–	–	–	–	1009.65	1034.69
	352 × 192	FP32	25.9	14.80	145.94	158.72	175.98	878.53	968.72
		FP16	25.9	–	–	–	197.29	1040.57	1152.97
		INT8	21.4	–	–	–	–	1117.36	1237.66

Table B.2: Benchmark of all trained and quantized YOLOv8 models on all available devices. For these tests, a batch size of 32 was used. However, for special cases marked by `*`, where the model did not fit into the device's memory with the large batch size, a batch size of 8 was used instead. Models tested were of a dynamic shape in the batch size dimension. Additionally, the TensorRT inference backend was used on NVIDIA Jetson devices, while the ONNX Runtime backend was used for tests on the rest of the devices. Models were only quantized to `FP16` precision on NVIDIA Jetson devices and `INT8` precision was only used on Jetson Xavier NX and Jetson AGX Xavier.

Appendix C

Complete Mean Average Precision Metrics for YOLOv8 Models

This appendix provides a complete table of the six mean Average Precision (mAP) metrics for each model trained and quantized in this paper. The data can be seen in Table C.1. Please note that the quantization process was performed by the TensorRT library individually for each of the utilized NVIDIA Jetson devices. The results shown in the table were measured on the NVIDIA Jetson AGX Xavier.

Model (YOLOv8)	Input resolution	Precision	mAP (%)					
			mAP	IoU:50	IoU:75	small	medium	large
medium	640 × 384	FP32	60.3	80.5	69.5	31.6	60.6	77.0
		FP16	60.3	80.5	69.3	31.6	60.7	77.0
		INT8	47.1	67.2	55.6	16.4	47.6	66.3
small	640 × 384	FP32	57.8	78.6	66.0	27.3	58.5	74.4
		FP16	57.8	78.6	66.0	27.4	58.5	74.2
		INT8	54.4	76.3	62.5	24.5	54.9	70.3
MobileNetV2	512 × 288	FP32	54.8	75.9	61.8	23.5	55.0	71.5
		FP16	54.8	75.9	61.9	23.6	54.9	71.8
		INT8	45.9	65.5	52.3	21.2	45.3	60.7
nano	640 × 384	FP32	53.0	74.9	60.8	22.9	52.9	68.5
		FP16	53.0	74.9	60.7	23.0	52.9	68.3
		INT8	43.0	65.0	48.4	18.2	45.3	53.0
	512 × 288	FP32	49.2	69.9	56.9	21.0	48.3	67.3
		FP16	49.2	69.8	56.9	20.9	48.2	66.9
		INT8	35.6	53.5	39.8	15.0	35.9	50.0
	448 × 256	FP32	51.1	72.3	59.2	20.6	50.8	68.1
		FP16	51.1	72.3	59.2	20.5	50.8	68.3
		INT8	37.2	55.5	41.7	14.0	37.1	51.6
pico	512 × 288	FP32	45.4	66.4	50.2	15.1	44.8	61.4
		FP16	45.4	66.4	50.0	15.1	44.8	61.2
		INT8	29.7	44.6	32.9	6.5	26.0	49.7
	448 × 256	FP32	41.5	62.1	46.8	13.6	40.8	55.2
		FP16	41.5	62.1	46.8	13.6	40.9	55.1
		INT8	35.6	55.2	39.6	12.2	35.5	47.4
	384 × 224	FP32	41.7	61.5	46.5	12.8	40.5	58.9
		FP16	41.8	61.5	46.5	12.8	40.5	58.8
		INT8	32.5	49.0	36.1	10.1	30.8	48.6
femto	512 × 288	FP32	31.8	48.8	35.1	7.2	29.8	41.6
		FP16	31.8	48.8	35.0	7.2	29.8	41.7
		INT8	30.0	46.6	32.6	7.1	27.0	42.0
	448 × 256	FP32	30.0	47.7	31.8	7.0	25.9	42.9
		FP16	30.0	47.7	31.9	7.0	25.9	43.0
		INT8	26.4	41.3	28.7	6.1	21.9	40.4
	384 × 224	FP32	28.1	45.2	30.2	7.0	26.3	38.6
		FP16	28.1	45.3	30.2	7.0	26.4	38.7
		INT8	23.5	40.9	24.2	5.9	21.1	33.1
	352 × 192	FP32	25.9	43.0	27.5	5.4	22.7	37.0
		FP16	25.9	43.0	27.5	5.4	22.6	37.0
		INT8	21.4	34.8	23.5	4.4	18.2	32.3

Table C.1: Comparison of multiple mean Average Precision metrics for each trained model and input resolution, including models quantized to `FP16` and `INT8`. Results were measured on NVIDIA Jetson AGX Xavier.

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] Gou, J., Yu, B., Maybank, S. J. and Tao, D. Knowledge Distillation: A Survey. *International Journal of Computer Vision*. Springer Science and Business Media LLC. mar 2021, vol. 129, no. 6, p. 1789–1819. DOI: 10.1007/s11263-021-01453-z.

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

[15] 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.

[16] 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.

[17] Jocher, G., Chaurasia, A. and Qiu, J. YOLO by Ultralytics. [online]. 2023. Accessed: May 5, 2023. Available at: https://github.com/ultralytics/ultralytics.

[18] 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.

[19] Labelbox. *Labelbox* [online]. 2022. Accessed: May 5, 2023. Available at: https://labelbox.com.

[20] 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.

[21] 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.

[22] Liu, W., Anguelov, D., Erhan, D., Szegedy, C., Reed, S. et al. SSD: Single Shot MultiBox Detector. Springer International Publishing. december 2015, p. 21–37. DOI: 10.1007/978-3-319-46448-0_2.

[23] 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.

[24] Maity, M., Banerjee, S. and Chaudhuri, S. S. Faster R-CNN and YOLO based Vehicle detection: A Survey. In: *2021 5th International Conference on Computing Methodologies and Communication (ICCMC)*. IEEE, Apr 2021. DOI: 10.1109/iccmc51019.2021.9418274.

[25] 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.

[26] OpenMMLab. Dive into YOLOv8: How does this state-of-the-art model work? [online]. 2023. Accessed: May 5, 2023. Available at: https://openmmlab.medium.com/dive-into-yolov8-how-does-this-state-of-the-art-model-work-10f18f74bab1.

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

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

[29] Redmon, J., Divvala, S., Girshick, R. and Farhadi, A. You Only Look Once: Unified, Real-Time Object Detection. arXiv. june 2015. DOI: 10.48550/ARXIV.1506.02640.

[30] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L.-C. MobileNetV2: Inverted Residuals and Linear Bottlenecks. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2018, pp. 4510-4520*. arXiv. january 2018. DOI: 10.48550/ARXIV.1801.04381.

[31] Shah, D. *Mean Average Precision (mAP) Explained: Everything You Need to Know* [online]. 2022. Accessed: May 5, 2023. Available at: https://www.v7labs.com/blog/mean-average-precision.

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

[33] 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.

[34] 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.

[35] Wang, H., Yu, Y., Cai, Y., Chen, X., Chen, L. et al. A Comparative Study of State-of-the-Art Deep Learning Algorithms for Vehicle Detection. *IEEE Intelligent Transportation Systems Magazine*. Institute of Electrical and Electronics Engineers (IEEE). 2019, vol. 11, no. 2, p. 82–95. DOI: 10.1109/mits.2019.2903518.

[36] 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.

[37] 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.

[38] 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.

[39] Wu, T.-H., Wang, T.-W. and Liu, Y.-Q. Real-Time Vehicle and Distance Detection Based on Improved Yolo v5 Network. In: *2021 3rd World Symposium on Artificial Intelligence (WSAI)*. IEEE, Jun 2021. DOI: 10.1109/wsai51899.2021.9486316.

[40] Zhang, A., Lipton, Z. C., Li, M. and Smola, A. J. *Dive into Deep Learning* [online]. 2021. Accessed: May 7, 2023. Available at: https://d2l.ai.

[41] 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.