Universidade de Aveiro
2023

Gonçalo Manuel
Cordeiro Ribeiro

Perceção com Redes Neuronais Multi-tarefa no ATLASCAR2

Perception using Multi-Tasked Neural Networks on ATLAS-CAR2

Universidade de Aveiro
2023

Gonçalo Manuel Cordeiro Ribeiro

Perceção com Redes Neuronais Multi-tarefa no ATLASCAR2

Perception using Multi-Tasked Neural Networks on ATLASCAR2

Dissertação apresentada à Universidade de Aveiro para cumprimento dos requisitos necessários à obtenção do grau de Mestre em Engenharia Mecânica, realizada sob orientação científica de Doutor Vítor Manuel Ferreira dos Santos, Professor Associado com Agregação, do Departamento de Engenharia Mecânica da Universidade de Aveiro.

Esta dissertação foi desenvolvida com recursos do IEETA - Instituto de Engenharia Eletrónica e Informática de Aveiro, no âmbito do Projeto UIDB/00127/2020, e usou fundos comparticipados pelo DEM/TEMA - Centro de Tecnologia Mecânica e Automação no contexto do Projeto UIDB/00481/2020, ambos financiados pela FCT - Fundação para a Ciência e Tecnologia.

Agradecimentos / Acknowledgements

Ao Professor Vítor Santos, pela orientação e conhecimentos que me transmitiu ao longo do ano. Um agradecimento especial por me ajudar a manter o foco no que era importante, por me ter apoiado e motivado em momentos de fraqueza e pela ajuda incansável que me deu durante todo este tempo.
Ao pessoal do LAR pela ajuda e companhia.
Á minha família, por estarem sempre lá me ajudar, apoiar e motivar em todos os momentos.
Aos meus colegas de casa, pelo apoio e por não me deixarem passar sede.
Aos mestres Afonso Mesquita, Diogo Canha, Francisco Custódio, João Duarte, João Reis e João Sousa pela amizade e companheirismo e pelas noites de diversão ao longo destes anos.

Keywords Deep Learning, Artificial Intelligence, Multitask, Multiple Models, Object Detection, Image Segmentation, ATLASCAR2

Abstract

Efficient perception is a fundamental requirement for ADAS and ADS, with implications for safety, accuracy, and speed. The choice between multi-tasked and single-tasked deep learning networks can significantly impact the performance of these systems and their ability to understand and respond to the complex driving environment.
This dissertation explores the comparison between multi-tasked neural networks and multiple single-tasked networks. It investigates car perception, focusing on object detection and image segmentation, covering car detection, road segmentation, and lane marking.
To make the comparisons possible and also to implement different kinds of models in the ATLASCAR2's inference unit, a versatile software system designed to seamlessly run multiple deep-learning models with distinct tasks was developed for this dissertation.
Single-tasked networks like YOLOv5, YOLOv7, and YOLOv8 were evaluated for object detection, while road segmentation was evaluated with Mask2Former, UPerNet, and SegFormer. Lane marking was analyzed using RESA, O2SFormer, and UFLDv2. The multi-tasked networks evaluated included YOLOP, YOLOPv2, and TiwnLiteNet.
The dissertation findings indicate that combining multiple single-tasked models can lead to synchronization challenges and slower inference speeds. Multi-tasked networks outperform multiple single-tasked models in terms of efficiency, although their performance benefits are more pronounced when handling tasks that share a closer relationship.

Palavras-chave *Deep Learning*, Inteligência artificial, Multi-tarefa, Multiplos modelos, Deteção de objetos, Segmentação de imagens, ATLASCAR2

Resumo

A perceção eficaz é um requisito fundamental para sistemas avançados de assistência à condução e de condução autónoma, com implicações para a segurança, precisão e velocidade. A escolha entre redes de *deep learning* multi-tarefa e mono-tarefa pode afetar significativamente o desempenho destes sistemas e a sua capacidade de compreender e responder ao complexo ambiente de condução.
Esta dissertação explora a comparação entre redes neurais multitarefa e múltiplas redes unitarefa. Investiga a perceção automóvel, centrando-se na deteção de objectos e na segmentação de imagens, abrangendo a deteção de carros, a segmentação de estradas e a marcação de faixas de rodagem.
Para tornar as comparações possíveis e também para implementar diferentes tipos de modelos de *deep learning* na unidade de inferência do ATLASCAR2, foi desenvolvido um sistema de *software* versátil concebido para executar sem problemas vários modelos de *deep learning* com tarefas distintas.
Redes de tarefa única como a YOLOv5, a YOLOv7 e a YOLOv8 foram avaliadas para a deteção de carros. A Mask2Former, UPerNet e a SegFormer foram avaliadas na segmentação de estradas. Já as redes RESA, O2SFormer e UFLDv2 foram avaliadas na marcação de faixas de rodagem. As redes multi-tarefa avaliadas incluíram a YOLOP, a YOLOPv2 e a TiwnLiteNet.
Os resultados da dissertação indicam que a combinação de vários modelos de tarefa única pode levar a desafios de sincronização e a velocidades de inferência mais lentas. As redes multitarefa superam a utilização de vários modelos de tarefa única em simultâneo em termos de eficiência, embora os seus benefícios de desempenho sejam mais pronunciados quando lidam com tarefas mais relacionadas entre si.

Contents

List of Tables

Intentionally blank page.

List of Figures

Intentionally blank page.

List of Acronyms

ADAS Advanced Driver-Assistance System

ADS Automated Driving System

AI Artificial Intelligence

ANN Artificial Neural Network

AP Average Precision

API Application Programming Interface

BIFPN Bi-directional Feature Pyramid Network

CNN Convolution Neural Network

DETR DEtection TRansformer

E-ELAN Extended Efficient Layer Aggregation Network

FCN Fully Convolutional Network

FN False Negative

FP False Positive

FPN Feature Pyramid Network

FPS Frames Per Second

IoU Intersection over Union

LAR Laboratory for Automation and Robotics

mAP Mean Average Precision

MIMO Multi-Input Multi-Output

mIoU Mean Intersection over Union

MISO Multi-Input Single-Output

MiT	Mix Transformer
mPA	Mean Pixel Accuracy
MTL	Multi-Task Learning
O2SFormer	One-to-Several Transformer
PA	Pixel Accuracy
R-CNN	Region-based Convolutional Neural Network
RESA	REcurrent Feature Shift Aggregator
ROS	Robot Operating System
RPN	Region Proposal Network
SDK	Software Development Kit
SIMO	Single-Input Multi-Output
SOTA	State-of-the-art
SSD	Single Shot MultiBox Detector
TP	True Positive
UPerNet	Unified Perceptual Parsing Network
ViT	Vision Transformer
YOLACT	You Only Look At CoefficienTs
YOLO	You Only Look Once
YOLOP	You Only Look Once for Panopitic Driving Perception
YOSO	You Only Segment Once
UFLDv2	Ultrafast lane detection v2

Chapter 1

Introduction

The domains of Automated Driving System (ADS) and Advanced Driver-Assistance System (ADAS) have fostered diverse research endeavors. The full realization of the envisioned goals with ADS holds the potential to substantially reduce road accidents worldwide and alleviate drivers from the daily grind of commuting, thereby enhancing their overall quality of life.

For this, vehicles must have sensors to extract real-world information. The data from the sensors has to be analyzed to get useful information. The analyses can be done using deterministic algorithms, and when that is proven unfeasible, machine learning methodologies, such as artificial neural networks, come into play.

The study undertaken within this dissertation constitutes a contribution to the ATLAS project in the context of perception. It encompasses exploring and implementing single and multi-tasked networks tailored to perception-related tasks.

1.1 ATLAS project

The study and developments conducted in this dissertation are integrated in the ATLAS project. This project started in the Laboratory for Automation and Robotics (LAR) at the Department of Mechanical Engineering of the University of Aveiro. The main goal is to develop and enable the proliferation of advanced sensing and active systems designed for implementation in automobiles and affine platforms. The project started with controlled environments but has been dealing with autonomous navigation in real road scenarios for several years. The prototype used for research is currently ATLASCAR2, a Mitsubishi i-MiEV (figure 1.1), equipped with multiple sensors and related hardware.

1.2 Problem description

The perception of the environment in ADAS and ADS involves multiple tasks and can be accomplished using various sensors. However, some of these tasks are highly complex and are hard to solve deterministically, so machine learning algorithms such as neural networks are a good alternative.

ATLASCAR2 is currently equipped with an NVIDIA Jetson Xavier AGX, a processing unit capable of running Deep Learning networks. In previous work, some networks were tested on this device.

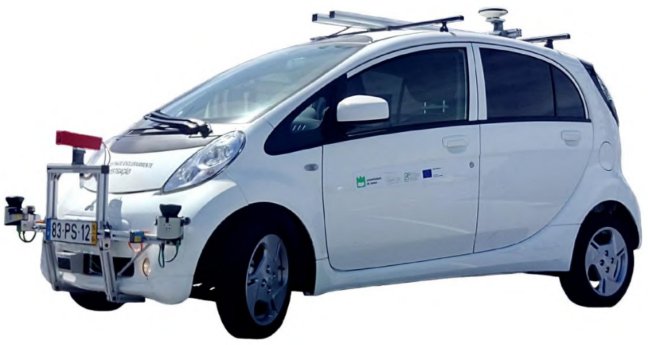


Figure 1.1: The ATLAS prototype: ATLASCAR2 [1]

In practical terms, for ADAS and ADS, several tasks must be performed simultaneously, and the inference results must be communicated to other devices in the car, either to inform the driver or to make decisions autonomously. The execution of multiple tasks simultaneously can be done through various single-task networks or one multi-task network. This raises the challenge of finding if the most advantageous way to achieve a reliable and efficient perception system is through multi-tasked neural networks or multiple single-tasked networks.

1.3 Objectives

The work developed in the scope of this dissertation has two main objectives. The first is to create an infrastructure capable of performing inferences with multiple kinds of deep learning models and sharing the results with other devices onboard. The second is to use that infrastructure to evaluate and find the most advantageous way to execute multiple tasks. To accomplish these objectives, the plan is to execute the following tasks:

- To choose a set of perception tasks to analyze.
- To search multi and single-task models to perform the tasks.
- To develop software capable of seamlessly running different deep learning models
- To test the software created on the Jetson and another device.
- To evaluate some recent multi and single-task networks.

1.4 Document structure

This dissertation is composed of 6 chapters:

- Chapter 1: Presents the introduction, which is an overview of the problem and its context.
- Chapter 2: Presents the state of the art, some techniques used in vehicle perception, concepts related to multi-task neural networks, deep learning models for perception tasks, and evaluation metrics.

- Chapter 3: Describes the available hardware and software infrastructure.
- Chapter 4: Describes the proposed solution and the addressed challenges within its development.
- Chapter 5: Presents the tests done on the developed solution and the evaluation results.
- Chapter 6: Presents the conclusions of this dissertation and suggestions for future works.

Intentionally blank page.

Chapter 2

Related work and state of the art

To build ADAS and ADS, it is essential to employ fast and reliable perception algorithms. This chapter starts by exploring some of the principles and techniques of perception. Furthermore, concepts associated with multi-task neural networks and the latest and State-of-the-art (SOTA) models are presented. In the chapter, there are also some evaluation metrics to facilitate model evaluation and comparison.

2.1 Perception

The perception of surroundings is essential for ADAS and ADS. There are different types of sensors, such as camera, LiDAR, radar, and ultrasonic sensors, that can enable or improve the perception capabilities of ADAS and ADS [2]. Figure 2.1 shows some of the state-of-the-art ADAS sensors.

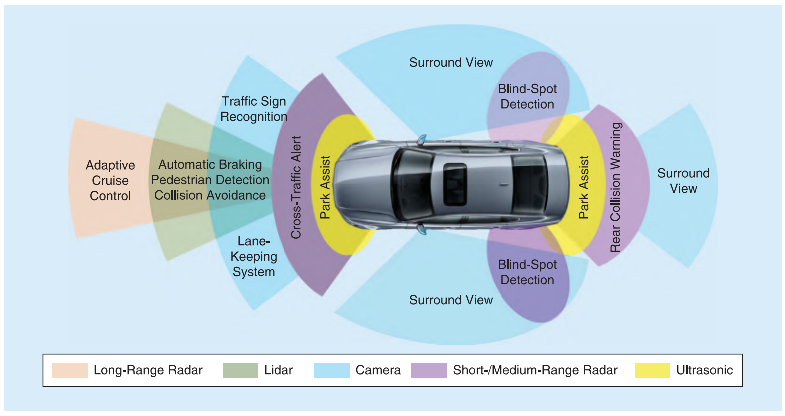


Figure 2.1: Most common state-of-the-art sensors used in ADAS [2]

Since the previous work, presented in [3], has been primarily about computer vision, for continuity purposes and time management, this dissertation focuses on computer vision as well.

By using cameras it is possible to extract useful information through computer vision principles, such as image acquisition, preprocessing, segmentation, object detection, object tracking, and depth estimation [2].

2.1.1 Object detection

Object detection is a key computer vision component, enabling machines to locate and identify objects within images. Traditional methods use handcrafted features, efficient but less capable in complex situations. Deep learning excels in capturing fine object details, especially in challenging environments. This dissertation explores the deep learning methods, focusing on two-stage and one-stage object detectors [4].

Two-stage detectors

In this kind of detector, two stages are used to detect objects. The first stage involves generating proposals to identify regions where objects may be present. In the second stage, predictions are made for each proposal using classification methods to determine whether an object is present or not and what its class is [4]. Figure 2.2 represents a deep learning model that uses these two stages format.

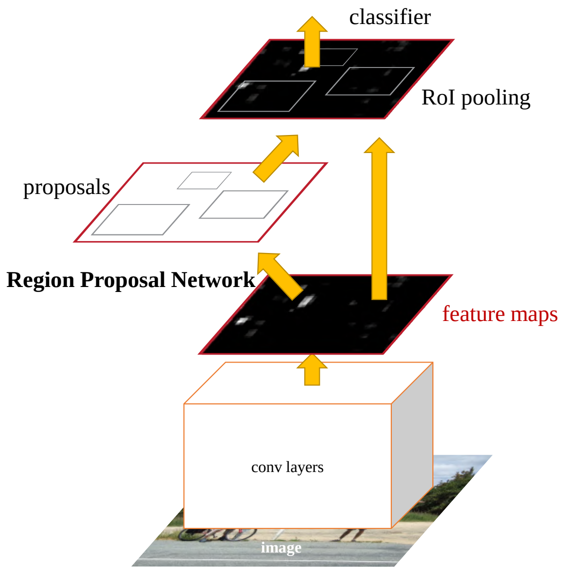


Figure 2.2: Faster Region-based Convolutional Neural Network (R-CNN) architecture [5]

One of the first two-stage object detectors was R-CNN, and, after that, improvements were made, leading to fast and faster R-CNN [5]. Faster R-CNN includes a Region Proposal Network (RPN), which generates proposals with various scales and aspect ratios, merged and sharing convolutional computations with Fast R-CNN. The beauty of this is that RPN uses "attention" mechanisms, telling the detection module (Fast R-CNN) where to look. Another area where Faster R-CNN improves upon previous models is by introducing anchor boxes as references at multiple scales and aspect ratios, and, instead of relying on pyramids of images or filters, it utilizes a pyramid of regression references based on the anchor box concept. This way, Faster R-CNN avoid the need to enumerate images or filters of various scales or aspect ratios [5].

Faster R-CNN was also the foundation for Mask R-CNN for instance segmentation (section 2.1.2).

One-stage detectors

One-stage detectors do not have an intermediate task. Therefore, conceptually, they have a simpler and faster architecture than two-stage detectors [4]. The first technique of one-stage detection was presented with You Only Look Once (YOLO), which uses a single neural network applied to the whole image to detect objects but has a worse accuracy if the input image size is different from the training image size. YOLOv2 solved this issue and Single Shot MultiBox Detector (SSD) improved the performance. Another way of improving the performance was presented with Retina-Net, using focal loss instead of the existing standard cross-entropy loss. Later, YOLOv3, using Feature Pyramid Network (FPN), managed to outperform the fastest versions of Retina-Net, getting similar accuracy at a much higher inference speed [4].

Since one-stage detectors are more suitable for real-time applications, it is more worthwhile to explore them. Therefore, in section 2.3 are presented newer one-stage detectors.

2.1.2 Image segmentation

Image segmentation is a vital computer vision technique that partitions an image into distinct regions based on pixel characteristics. This process helps identify and isolate objects or areas of interest within an image.

There are three categories of deep learning-based segmentation: Semantic, instance, and panoptic segmentation [6].

Semantic segmentation

The basic idea of this type of segmentation is to associate each pixel of an image with a class label. A way to illustrate this is to assign a color to each class and then color each pixel according to its class, as shown in figure 2.3 [6].

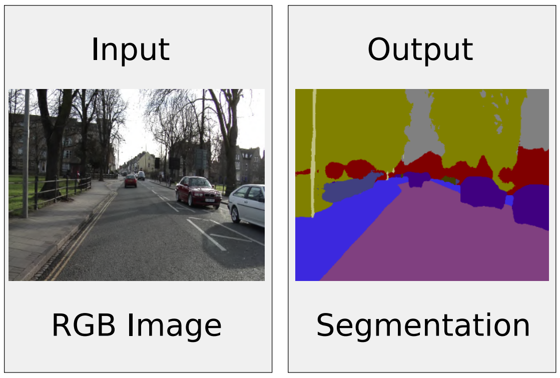


Figure 2.3: Semantic segmentation example (adapted from [7])

One way of obtaining semantic segmentation is through a Convolution Neural Network (CNN) to get a heat map like in figure 2.4. However, the result is a coarse segmentation compared to the input resolution [8]. To solve this problem, it is necessary to increase the resolution, keeping the global context, which can be done through a deconvolution structure, a decoder [7–9].

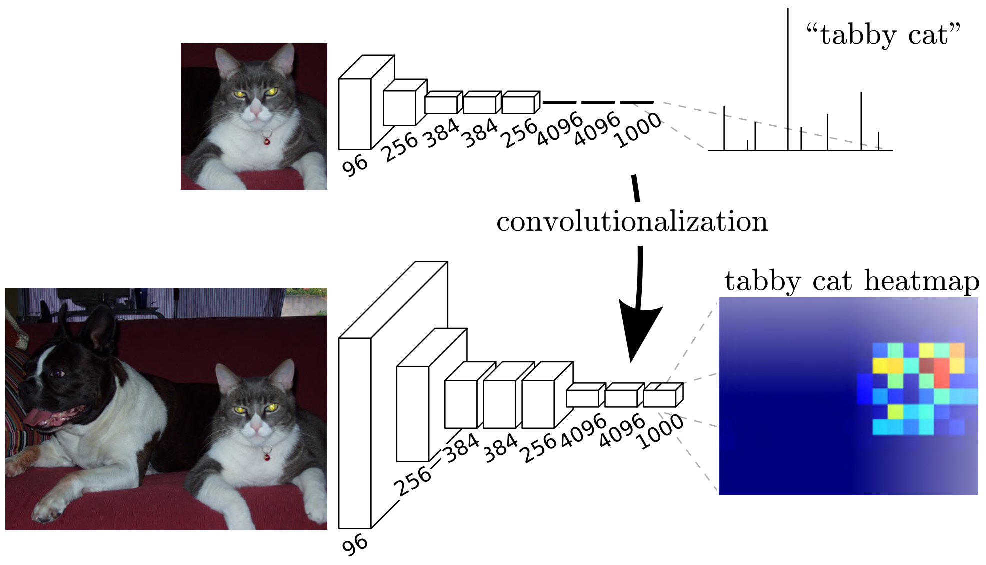


Figure 2.4: Image to heatmap using a CNN [8]

The encoder-decoder solution works but produces bad results, as shown by the authors of [8]. To address that, the authors added links combining the final prediction and previous layers with finer strides, as shown in figure 2.5.

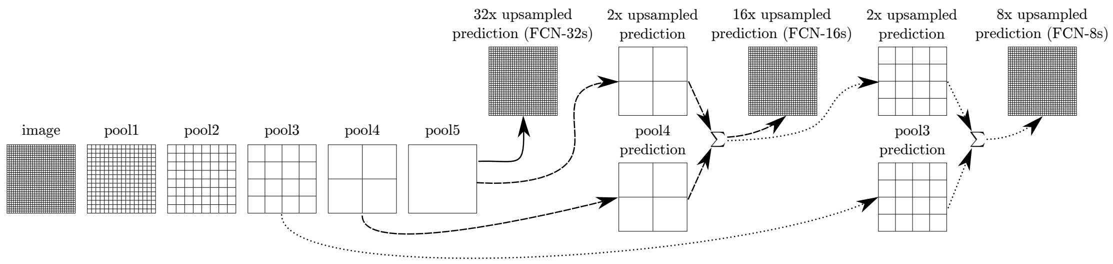


Figure 2.5: CNN combining the final prediction with previous layers [8]

In this encoder-decoder structure, there are other approaches, like SegNet and U-net. In SegNet, only the max-pooling indices are transferred from the encoder to the decoder [7], and in U-net, the entire feature maps are transferred [9].

More recently, transformers are being used within the networks' architecture with attention mechanisms for global context understanding, resulting in higher precision than traditional CNN-based architectures.

Transformers are a type of artificial neural network architecture designed for sequence-to-sequence transformations in deep learning. Introduced in 2017 by the authors of [10], they have gained traction in various domains. Transformers rely on the attention mechanism, discerning the relative importance of different segments within the input data. They excel in learning context and understanding through sequential data analysis, employing techniques such as attention or self-attention. Recently, they started to be implemented in computer vision, with Vision Transformer (ViT). ViT divides images into patches, converting each into a vector, which is then processed by a Transformer encoder [11].

Instance segmentation

The goal of instance segmentation is to predict the object class label and the pixel-specific object instance mask [6], as shown in figure 2.6. Like object detection, instance segmentation has one and two-stage methods [12].

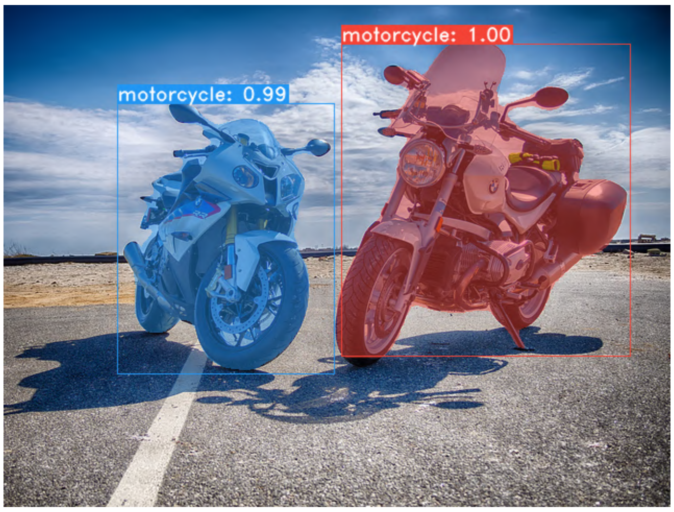


Figure 2.6: Instance segmentation example [12]

Two-stage instance segmentation can be done through both proposal-based (R-CNN driven) and proposal-free (Fully Convolutional Network (FCN) driven) methods. Proposal-based methods usually detect a bounding box as the object proposal and then segment out the foreground object in the box region. A limitation of this method is the segmentation of long and thin instances like the lane line of the road. On the other hand, proposal-free methods follow a scheme of representation learning and then clustering [13].

Mask R-CNN is a two-stage network based in Faster R-CNN (section 2.1.1) that benefits from a proposal-based method and a FCN, as it uses the region proposal from Faster R-CNN (proposal-based), but instead of using RoIPool, which is not suitable to extract pixel-wise precise masks, it uses RoIAlign. In parallel to predicting the class and box offset, Mask R-CNN applies a FCN in order to get the segmentation of the instance in that region [14]. Figure 2.7 represents the Mask R-CNN framework.

Two-stage instance segmentation is usually more accurate, but the inference speed is not suitable for real-time applications like ADAS or ADS [12].

One-stage instance segmentation methods are used to produce maps sensitive to objects' position in an image. These maps are then combined either through position-sensitive pooling or by combining the predictions of semantic segmentation and direction to create final masks for the objects. The problem with these approaches is that, although conceptually faster than two-stage methods, their architectures are speed-limiting, placing them far from some real-time [12] applications.

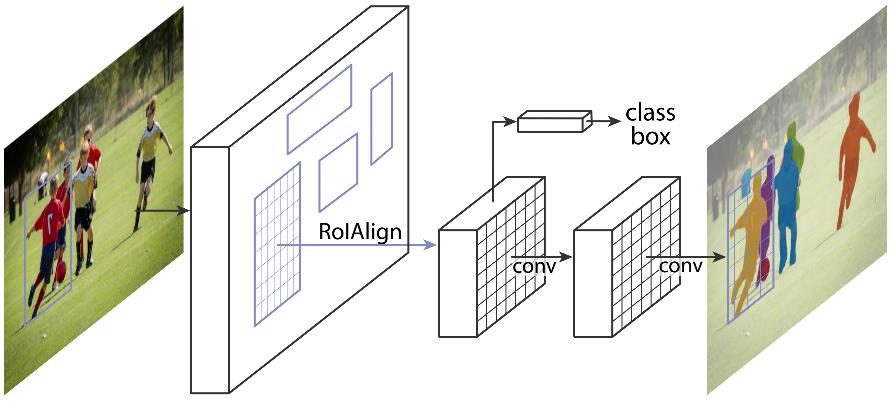


Figure 2.7: The Mask R-CNN framework [14]

As of 2019, new instance segmentation networks suitable for real-time inference started to appear. One of the first was You Only Look At CoefficienTs (YOLACT) [12], which is based on YOLO networks for object detection, and its architecture can be seen in figure 2.8.

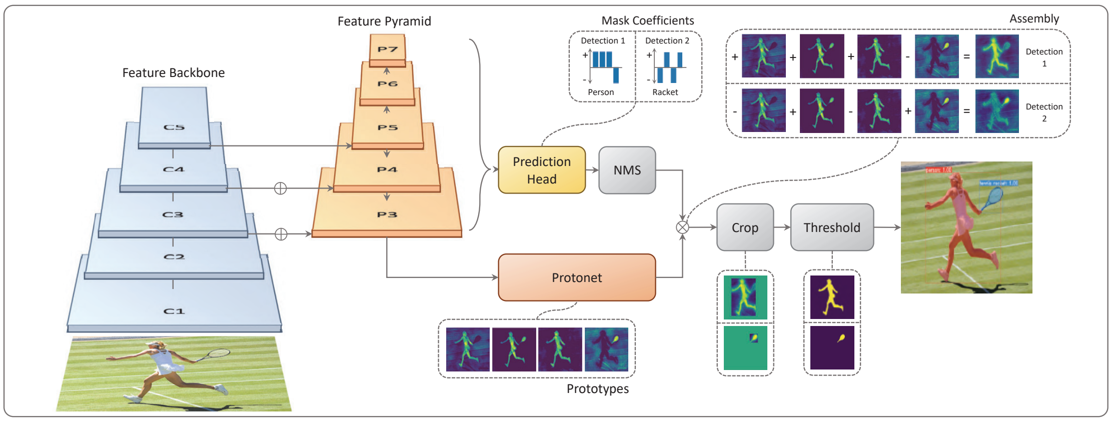


Figure 2.8: YOLACT architecture [12]

To achieve instance segmentation, YOLACT employs a Protonet to produce mask prototypes from the FPN. In parallel to that, and also from the FPN, YOLACT generates mask coefficients that can be positive or negative. Using the mask prototypes and coefficients, YOLACT combines them to create a mask for each detection. Finally, the resulting masks are cropped to fit within their respective bounding boxes [12].

Panoptic segmentation

Semantic segmentation assigns a semantic label to each pixel of an image. In contrast, instance segmentation separates each instance within a class without giving labels to uncountable objects such as roads, skies, etc. Panoptic segmentation aims to combine semantic and instance segmentation by assigning a semantic label and, if possible, an instance label to every pixel in the image [6, 15]. Figure 2.9 illustrates semantic, instance, and panoptic segmentation.

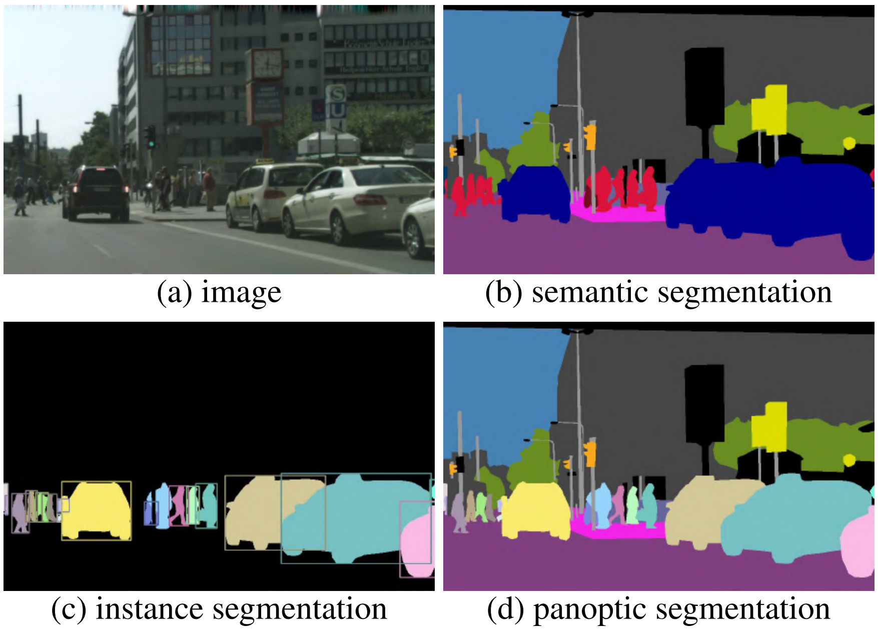

(a) image
(b) semantic segmentation
(c) instance segmentation
(d) panoptic segmentation

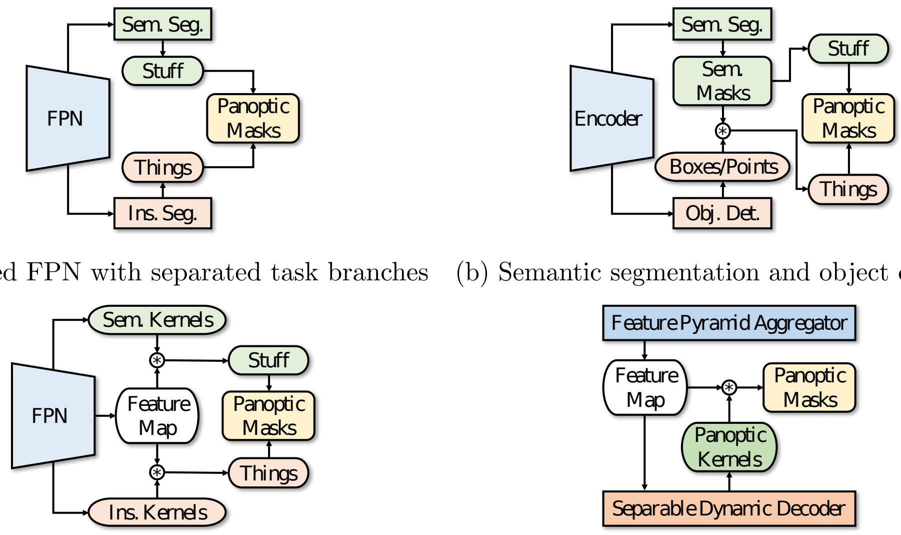
Figure 2.9: The main kinds of image segmentation [15]

There are several methodologies for accomplishing panoptic segmentation. One approach involves the utilization of a FPN shared between semantic and instance segmentation tasks, illustrated in figure 2.10a. Alternatively, segmentation masks can be generated and subsequently employed in object detection for instance recognition, as schematized in figure 2.10b. Another method is to use an FPN to generate semantic and instance kernels for convolution with image feature maps, illustrated in figure 2.10c [16].

The authors of [16] claim that the methodologies above are not suitable for real-time applications and present a new solution, schematized in figure 2.10d, in which panoptic segmentation is treated as a single task. To do such a thing, You Only Segment Once (YOSO) unifies the semantic and instance tasks by predicting masks via dynamic convolutions between panoptic kernels and image feature maps. This innovative approach allows YOSO to perform instance and semantic segmentation tasks in one go.

Summary - Perception

Perception in the context of ADAS and ADS is a complex subject with many possible ramifications.
The goal of object detection is to localize and classify objects. Object detectors are composed of either one or two stages. One-stage detectors are rather interesting since some of them are feasible for real-time application.
Semantic, instance and panoptic segmentation are the three main categories of image segmentation. Semantic segmentation labels each pixel with a class. Instance segmentation can distinguish instances in every countable class but can not label pixels of uncountable classes. Panoptic segmentation is a combination of semantic and instance segmentation.



(a) Shared FPN with separated task branches (b) Semantic segmentation and object detection

(c) Shared FPN for semantic and instance kernels (d) You Only Segment Once

Figure 2.10: Panoptic segmentation methodologies [16]

2.2 Multi-task neural networks

Deep learning has become a core technology for achieving smart and intelligent systems. This technology is based on the concept of Artificial Neural Network (ANN), as it is mainly composed of many connected processing elements called neurons.

Artificial Intelligence (AI) incorporates human behavior and intelligence into systems. In this domain, Machine Learning covers learning methods from data or past experiences. One of these methods is deep learning, which uses multi-layer neural networks to perform computations [17].

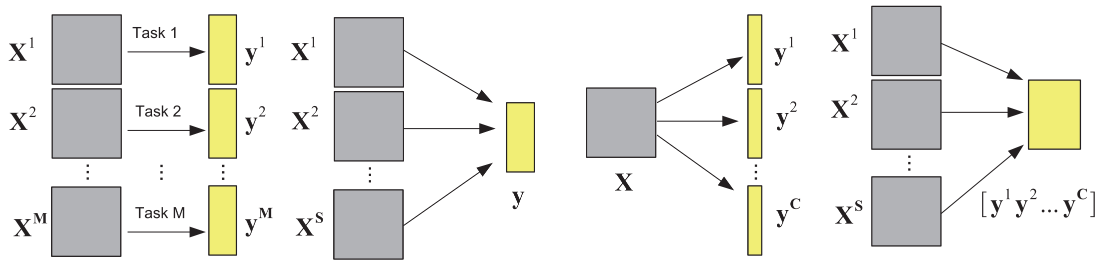
Traditional single-task neural networks use one model to perform a single task. Multiple models should be inferred simultaneously if more than one task is to be performed. Multi-task networks excel by performing more than one task with a single model [18].

When trained for tasks that are related or derived from others, multi-task networks have the potential to work faster since the same data can be used to train multiple tasks simultaneously, rather than training one task individually for a dataset [18], and to be more generalized [19].

One challenge in these networks is that they add problems that do not exist in single-task networks: different tasks may have different learning needs, harming the outcome of some tasks by benefiting others. This phenomenon is known as destructive interference, and minimizing this phenomenon is one of the main goals when using methods involving multi-task neural networks. This minimization is not, however, a trivial process [18].

2.2.1 Multi-task Architectures

Multi-task neural network architectures are rooted on Multi-Task Learning (MTL) [18]. In terms of inputs and outputs, they can be categorized into Multi-Input Single-Output (MISO), Single-Input Multi-Output (SIMO), and Multi-Input Multi-Output (MIMO), but for deep learning, the SIMO is the most typical structure [20]. Figure 2.11, in which $\mathbf{X}^M$ and $\mathbf{y}^M$ are a unique input and output, respectively, depicts these categories. These outputs can be in multiple output layers or in a single output layer that concatenates the output nodes [18, 20].



(a) MTL **(b)** MISO **(c)** SIMO **(d)** MIMO

Figure 2.11: General form of MTL, and its special cases [20]

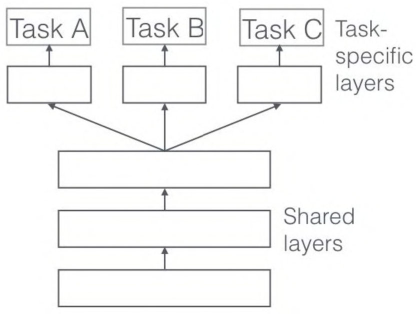
Multi-task architectures can also be partitioned into four groups: architectures for a particular task domain, multi-modal architectures, learned architectures, and conditional architectures. Multi-modal architectures tackle different domains in the same network, like computer vision and natural language processing for visual question answering. Single-domain architectures are designed to deal with only one domain. On the other hand, learned architectures and conditional architectures differ in the way they behave for a given piece of data: learned architectures are fixed, and conditional architectures change their architecture depending on the data [18].

Based on the way parameters are shared, multi-task neural network architectures are often divided into two groups: hard parameter sharing, which is the practice of sharing model weights between multiple tasks, and soft parameter sharing, where different tasks have individual task-specific models with separate weights, but the difference between parameters of different tasks is added to the joint objective function. Since the nature of multi-task methods has grown in the past years, these two categories alone are not broad enough to accurately describe the entire field [18].

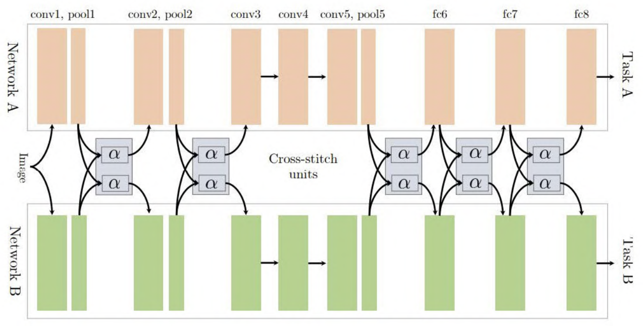
The main multi-task architectures in computer vision are Shared Trunk, Cross-Talk, Prediction Distillation, and Task Routing [18]. Shared Trunk has a traditional multi-head structure composed of a global feature extractor made of convolutional layers shared by all tasks, followed by an individual output branch for each task (figure 2.12a). In Cross-Talk, each task has a separate network, but with information flow between parallel layers (figure 2.12b) [18, 19, 21]. In prediction distillation, preliminary predictions are made in sub-tasks, and then these predictions are re-combined and used to compute final, refined predictions for the output tasks. Task routing is more flexible than shared trunk and cross-talk, allowing for fine-grained parameter sharing between tasks that occur at the feature level instead of the layer level [18].

In the context of the shared trunk architecture, it is common to use some terms to identify specific regions of the network:

- Backbone: This is the part of the network responsible for extracting features from the input. In CNN, for example, the backbone is often a pre-existing CNN (like ResNet) that has been pre-trained on a large dataset. The backbone processes the



(a) Shared Trunk [19]



(b) Cross-Talk [18]

Figure 2.12: Some computer vision multi-task architectures

input data and outputs a set of feature maps [22].

- Neck: The “neck” of a network is the part that connects the backbone and the head. It usually performs some form of aggregation or transformation on the feature maps output by the backbone before passing them to the head. An example of a “neck” is the FPN, which aggregates feature maps at different scales to create a multi-scale feature pyramid [22].
- Head: The “head” of a network is the part that makes predictions based on the features extracted by the backbone and processed by the neck. For example, in an object detection network, the head might consist of several layers that predict bounding boxes and class probabilities [22].

2.2.2 What is and what is not multi-task

At first glance, a multi-tasking network can be seen as one that manages multiple tasks. Although true, the meaning of task does not exactly correspond to our day-to-day definition. It is worth delving a little deeper to better understand the concept.

In machine learning, a task typically involves learning an output target from a single input source [20, 23]. This definition shows that a task varies with different inputs and/or different outputs. Since the task is also related to the learning process, it also varies with the loss function. More formally, in supervised learning, a task, $\mathscr{T}$, can be defined as a set composed by a distribution over input $\mathbf{x}$, $p(\mathbf{x})$, a distribution over output $\mathbf{y}$ given $\mathbf{x}$, $p(\mathbf{y} \mid \mathbf{x})$, and a loss function, $\mathscr{L}$:

$$\mathscr{T}_i \triangleq \{p_i(\mathbf{x}), p_i(\mathbf{y} \mid \mathbf{x}), \mathscr{L}_i\} \tag{2.1}$$

By this definition, in order to have different tasks, either $p_i(\mathbf{x})$, $p_i(\mathbf{y} \mid \mathbf{x})$ or $\mathscr{L}_i$ has to vary.

In a neural network, various types of tasks can be distinguished, including main tasks, sub-tasks, and auxiliary tasks. The main tasks refer to the primary objectives for which the network was designed. Auxiliary tasks are employed to enhance the performance of the main task(s) [19, 21], and usually, they are only used at the training stage [24]. On the other hand, sub-tasks are necessary components contributing to completing a main task, such as object classification and localization for object detection [25, 26]. An example of sub-tasks can be seen in the architecture suggested by the authors of [27] for panoptic segmentation, where the final loss function is given by $\mathscr{L} = \lambda_{\mathrm{i}} (\mathscr{L}_{\mathrm{c}} + \mathscr{L}_{\mathrm{b}} + \mathscr{L}_{\mathrm{m}}) + \lambda_{\mathrm{s}} \mathscr{L}_{\mathrm{s}}$. This equation is composed of two weighted losses: a loss for instance segmentation, $(\mathscr{L}_{\mathrm{c}} + \mathscr{L}_{\mathrm{b}} + \mathscr{L}_{\mathrm{m}})$, and a loss for semantic segmentation, $\mathscr{L}_{\mathrm{s}}$. In this case, is quite visible the composition of instance segmentation by three sub-tasks, each one with its loss: classification ($\mathscr{L}_{\mathrm{c}}$), bounding-box ($\mathscr{L}_{\mathrm{b}}$) and mask loss ($\mathscr{L}_{\mathrm{m}}$).

Tasks like panoptic segmentation can either manifest as a single-task, employing architectures like YOSO, as depicted in figure 2.10d, or it may be constructed as a multi-task encompassing two distinct sub-tasks: one for semantic segmentation and another for instance segmentation, as exemplified in figure 2.10a.

In the context of ADAS and ADS, the required tasks align with the main tasks of the analyzed models. Therefore, for the purpose of this dissertation, networks with multiple main tasks will be classified as multi-task, while the remaining ones will be considered single-task. In this sense, examples of tasks are object detection and panoptic segmentation. Instance and semantic segmentation may be main or sub-tasks depending on the network or the intended use.

2.2.3 Real-world implementation example

According to the author of [28], there is the example of Tesla (figure 2.13) where a common model is used in the network architecture, whose function is to obtain the features for different resolutions of the same image in an efficient way since it uses a Bi-directional Feature Pyramid Network (BIFPN), and using the features obtained by this model, shares them between tasks, without the need to repeat this process for each task.

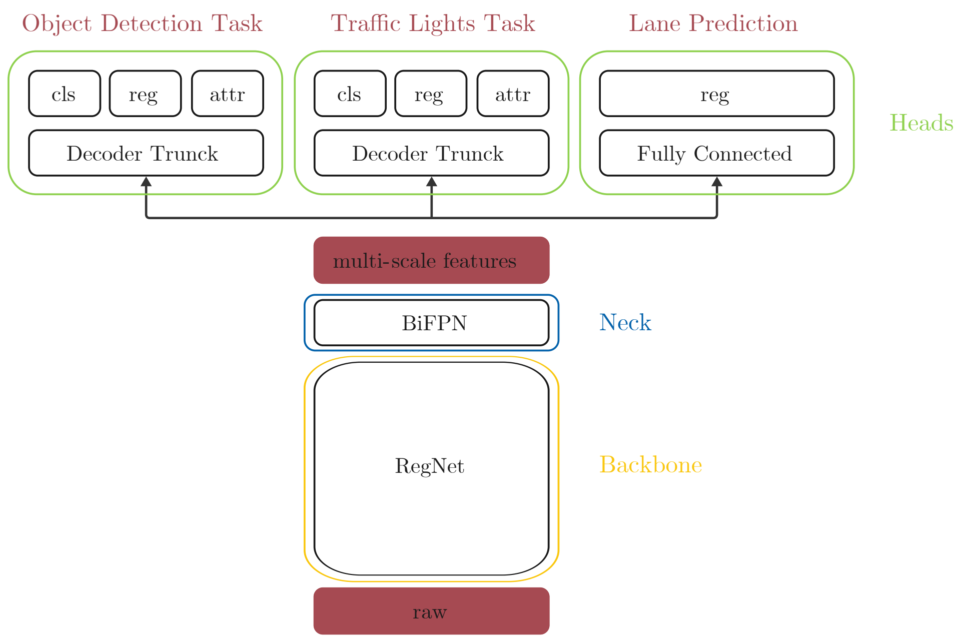


Figure 2.13: Architecture used by Tesla (adapted from [28])

Also, in figure 2.13, it is possible to see main and sub-tasks. For example, the traffic lights detection task is composed of a classification, a regression, and an attribute recognition task. As explained in the previous section, for the purposes of this dissertation, the sub-tasks are not considered tasks, making the traffic light detection a single task.

Summary - Multi-task neural networks

Multi-task neural networks are architectures based on MTL, and there are several approaches for different problems to design them. The key benefits of these networks are the potential for better generalization and the simultaneity of tasks. However, these architectures require precaution in the training stage to avoid losing too much performance of individual tasks.
There are main, sub, and auxiliary tasks, but for the scope of this dissertation, the main tasks are the most relevant ones.

2.3 Deep learning models

In order to test the solutions explained later in chapter 4 in this dissertation, some models are needed. The authors of [3] investigated the effectiveness of multi-tasked neural networks by deploying a You Only Look Once for Panopitic Driving Perception (YOLOP) model. This dissertation uses this deep learning model as a starting point to explore multi-tasked neural networks.

The YOLOP network performs three tasks: object detection, drivable area segmentation, and lane lines segmentation. The networks presented in the following sections were chosen because they perform similar tasks to YOLOP and have implementations available. These networks are separated by their architecture of single or multi-task.

2.3.1 Single-task models

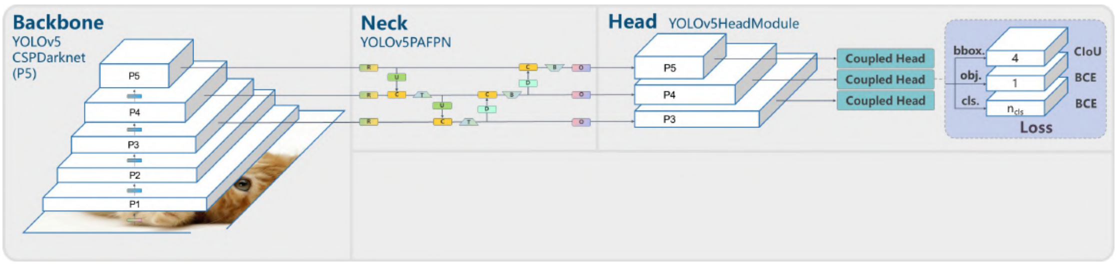
YOLOv8 and YOLOv5

YOLOv5 and YOLOv8 are part of the YOLO family and are both SOTA models for object detection, instance segmentation, and image classification tasks [29].

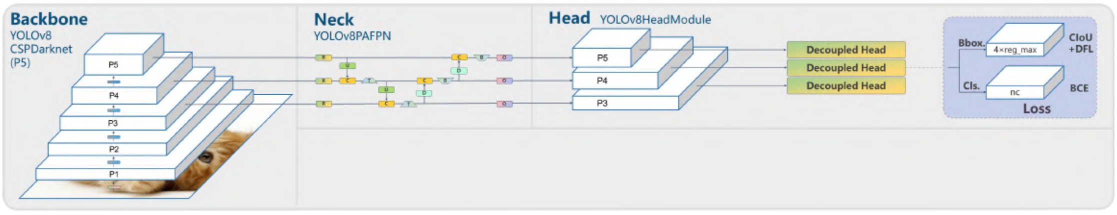
YOLOv5, developed by Ultralytics, builds upon the success of previous YOLO versions and introduces new features and improvements to boost performance and flexibility [29] further.

YOLOv8 is the latest model in the YOLO family, introduced in 2022 by Ultralytics. It is built on the YOLOv5 framework and includes several architectural and developer experience improvements. According to Ultralytics, it is faster and more accurate than YOLOv5 [29].

YOLOv5 employs a CSPDarknet architecture as its backbone, complemented by a neck featuring SPPF and CSP-PAN structures. In the case of YOLOv8, adjustments have been introduced not only to the YOLOv5 backbone and neck but also, more significantly, to the head. While YOLOv5 retains the same head as YOLOv3 and YOLOv4, YOLOv8 introduces a novel head design. In YOLOv8, the head is decoupled, anchor-free, and does not produce an objectness output. The loss function has also been revamped [29–31]. The architecture of YOLOv5 and YOLOv8 are represented in figures 2.14a and 2.14b respectively.



(a) YOLOv5



(b) YOLOv8

Figure 2.14: YOLOv5 and YOLOv8 architectures (adapted from [31])

YOLOv7

YOLOv7 stands as the latest official iteration within the YOLO family. Developed with Alexey Bochkovskiy collaboration, who carried forward the pioneering work of Joseph Redmon, one of the original authors of the YOLO series, YOLOv7 distinguishes itself by surpassing all previous versions in terms of both speed and accuracy.

The architecture of YOLOv7, is based on previous YOLO versions like YOLOv4. Some of the improvements are the following [32]:

- Extended Efficient Layer Aggregation Network (E-ELAN) as the computational block of the backbone - The E-ELAN architecture of YOLOv7 enables the model to learn better.
- Compound Model Scaling - allows to adjust key attributes of the model to generate models that meet the needs of different application requirements.
- Adicional head - YOLOv7 is not limited to one single head. The head responsible for the final output is called the lead head, and the head used to assist training in the middle layers is named auxiliary head.

Mask2Former

Mask2Former takes a unified approach to tackle instance, semantic, and panoptic segmentation. It achieves this by predicting a set of masks along with their associated labels. All three tasks are handled as if they were instance segmentation. Mask2Former outperforms the prior SOTA model, MaskFormer, in terms of performance and efficiency. This improvement is achieved by replacing the pixel decoder with a more advanced multi-scale deformable attention Transformer. Additionally, it adopts a Transformer decoder with masked attention to enhance performance without introducing extra computational overhead. Moreover, training efficiency is enhanced by calculating loss on subsampled points rather than the entire masks [33]. The architecture of Mask2Former can be seen in figure 2.15.

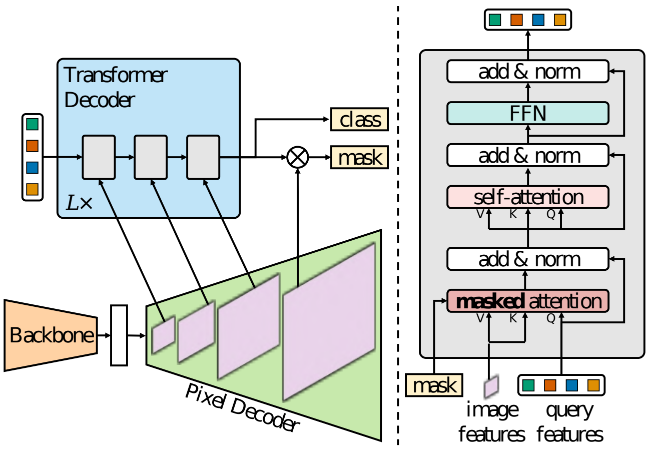


Figure 2.15: Mask2Former architecture [33]

In this dissertation, Mask2Former has been used with the Swin backbone. The Swin Transformer, a ViT variant, combines a hierarchical structure similar to CNN with efficient computation using Shifted windows. The shifted windowing scheme brings greater efficiency by limiting self-attention computation to non-overlapping local windows while also allowing for cross-window connection [34].

UPerNet with ConvNeXt backbone

The Unified Perceptual Parsing Network (UPerNet) is a powerful tool for semantic segmentation. It is designed to recognize a wide range of visual concepts in images, from scene categories and objects to parts, materials, and textures [35]. The useful task of this network for this dissertation is the semantic segmentation that corresponds to object recognition in this case. The remaining tasks can be discarded, resulting in the architecture illustrated in figure 2.16.

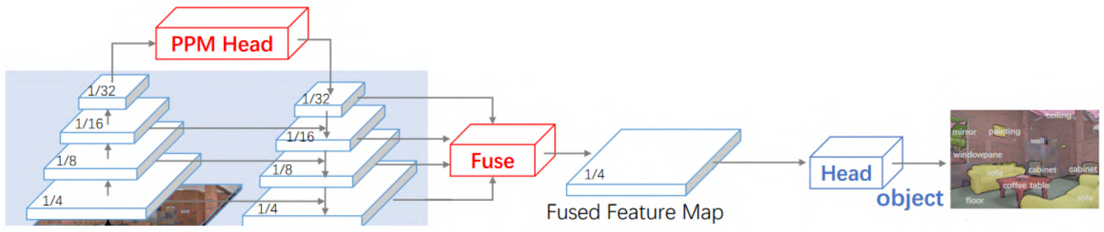


Figure 2.16: UPerNet architecture for semantic segmentation (adapted from [35])

In this dissertation, UPerNet has been used with the ConvNeXt backbone. Constructed entirely from standard ConvNet modules and influenced by ViT, ConvNeXts compete favorably with Transformers in terms of accuracy and scalability, achieving remarkable performance on a wide variety of vision tasks. They maintain the simplicity and efficiency of standard ConvNets while achieving state-of-the-art performance [36].

SegFormer

SegFormer comprises two main components: a hierarchical Transformer encoder and a lightweight all-MLP decoder head. The core of SegFormer, known as Mix Transformer (MiT), is the hierarchical Transformer encoder [37].

The authors of SegFormer initiated their approach by pre-training the Transformer encoder on ImageNet-1k for image classification. Subsequently, they removed the classification head and replaced it with the all-MLP decoder head. The model was then fine-tuned on diverse datasets, including ADE20K, Cityscapes, and COCO-stuff, to adapt its performance to semantic segmentation tasks [37]. This architecture is represented in figure 2.17.

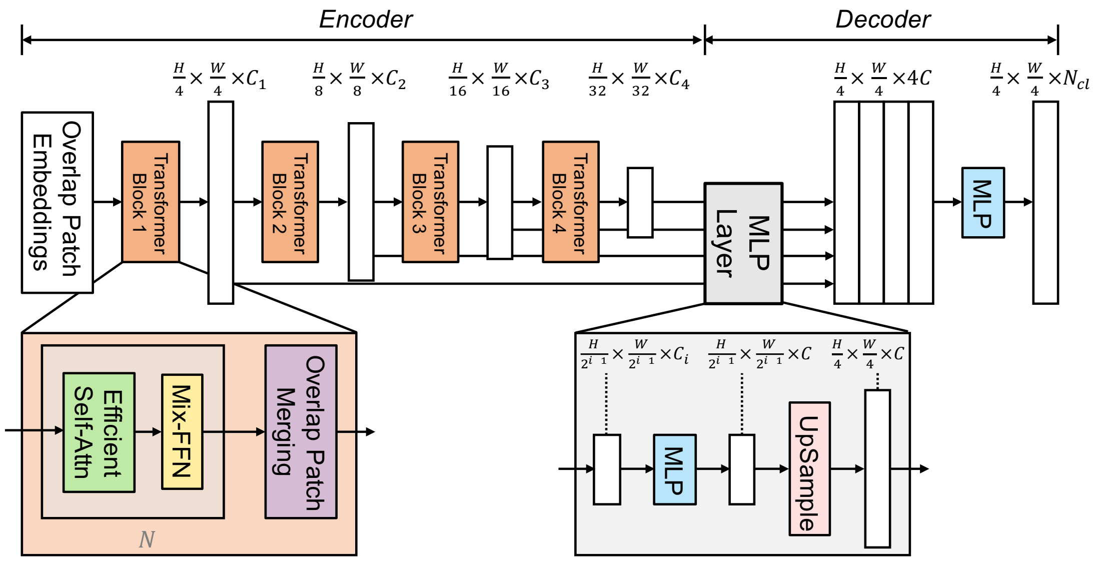

Encoder
Decoder
$\frac{H}{4}\times\frac{W}{4}\times C_1$
$\frac{H}{8}\times\frac{W}{8}\times C_2$
$\frac{H}{16}\times\frac{W}{16}\times C_3$
$\frac{H}{32}\times\frac{W}{32}\times C_4$
$\frac{H}{4}\times\frac{W}{4}\times 4C$
$\frac{H}{4}\times\frac{W}{4}\times N_{cl}$
Overlap Patch Embeddings
Transformer Block 1
Transformer Block 2
Transformer Block 3
Transformer Block 4
MLP Layer
MLP
Efficient Self-Attn
Mix-FFN
Overlap Patch Merging
N
$\frac{H}{2^{i+1}}\times\frac{W}{2^{i+1}}\times C_i$
$\frac{H}{2^{i+1}}\times\frac{W}{2^{i+1}}\times C$
$\frac{H}{4}\times\frac{W}{4}\times C$
MLP
UpSample

Figure 2.17: SegFormer architecture for semantic segmentation [37]

RESA

The REcurrent Feature Shift Aggregator (RESA) is a module designed for lane detection. It was developed to address the challenges of complex scenarios such as severe occlusion, ambiguous lanes, and the sparse supervisory signals inherent in lane annotations [38].

RESA works by enriching the lane feature after the feature extraction done by the backbone. It leverages the strong shape priors of lanes and captures spatial relationships of pixels across rows and columns. This is achieved by recurrently shifting the sliced feature map in vertical and horizontal directions, enabling each pixel to gather global information [38]. The researchers proposed a Bilateral Up-Sampling Decoder that combines coarse-grained and fine-detailed features in the up-sampling stage. Following upsampling by the decoder, the resulting feature map is employed to make predictions regarding the presence of each lane and its corresponding probability distribution. The existence prediction involves utilizing a fully connected layer for binary classification. Concurrently, pixel-wise lane probability distribution prediction is performed. This architecture is represented in figure 2.18 and is used with a ResNet34 as the backbone (encoder).

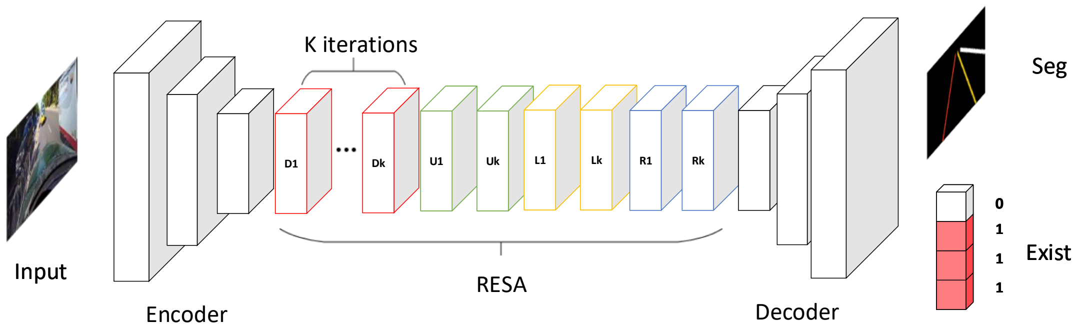


Figure 2.18: RESA architecture for lane detection [38]

O2SFormer

Lane detection techniques have demonstrated remarkable performance in real-world scenarios. However, many of these methods rely on post-processing steps that may lack the desired robustness. To address this challenge, end-to-end detectors like DEtection TRansformer (DETR) have been introduced to lane detection. Nonetheless, due to label semantic conflicts, DETR's one-to-one label assignment can hamper training efficiency. Additionally, its positional query lacks the capacity to provide explicit positional priors, making optimization challenging [39].

In response to these concerns, the authors of One-to-Several Transformer (O2SFormer) propose a novel approach involving one-to-several label assignment. This approach combines elements of both one-to-many and one-to-one label assignment strategies to mitigate label semantic conflicts while maintaining end-to-end detection. To enhance optimization in one-to-one assignment, they introduce the concept of layer-wise soft labels. These soft labels dynamically adjust the positive weight assigned to positive lane anchors across different decoder layers. Finally, the authors introduce a dynamic anchor-based positional query, which incorporates lane anchors into the positional query mechanism, facilitating the exploration of positional priors [39]. This architecture is represented in figure 2.19 and is used with a ResNet18 as the backbone.

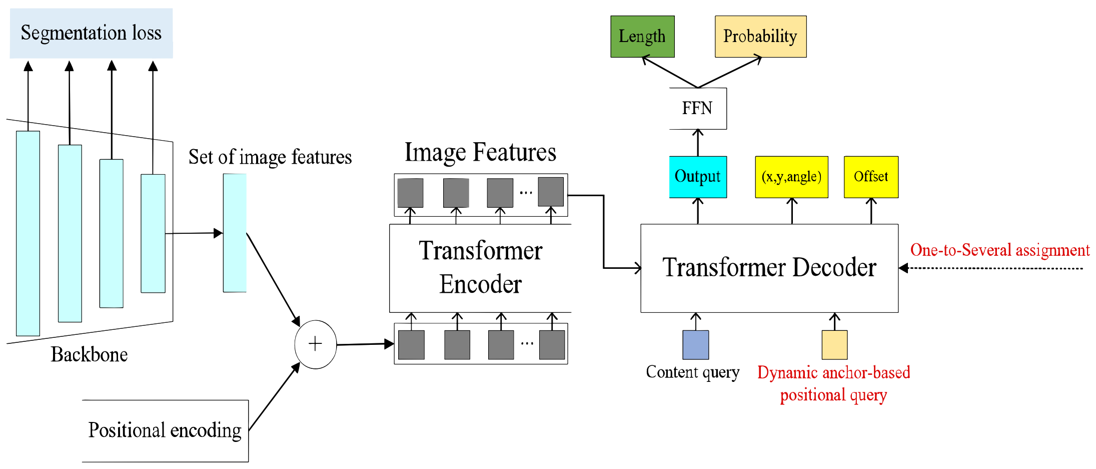


Figure 2.19: O2SFormer architecture for lane detection [39]

Ultrafast lane detection v2

Ultrafast lane detection v2 (UFLDv2) introduces an innovative approach to lane detection, framing it as an anchor-driven ordinal classification problem leveraging global features. This approach draws inspiration from human perception, where the identification of lanes, even in challenging scenarios with occlusions and adverse lighting, relies heavily on contextual and global information [40].

In this method, lanes are characterized by sparse coordinates associated with a hybrid set of anchors spanning both rows and columns. Consequently, the lane detection task is redefined as an ordinal classification problem to determine lane coordinates. This anchor-driven representation substantially reduces computational cost [40].

In terms of architecture, illustrated in figure 2.20, the initial step involves forwarding the input image through a backbone network, which for this dissertation is a ResNet18, to extract deep features. These deep features are subsequently flattened and directed into a classifier with dual output branches. The first branch, the localization branch, is responsible for acquiring coordinates on the hybrid anchors using a classification-based representation. Meanwhile, the second branch, known as the existence branch, predicts the presence or absence of each coordinate on the hybrid anchors. Upon obtaining the localization output, the process employs expectation, rather than argmax, to derive the lane coordinates [40].

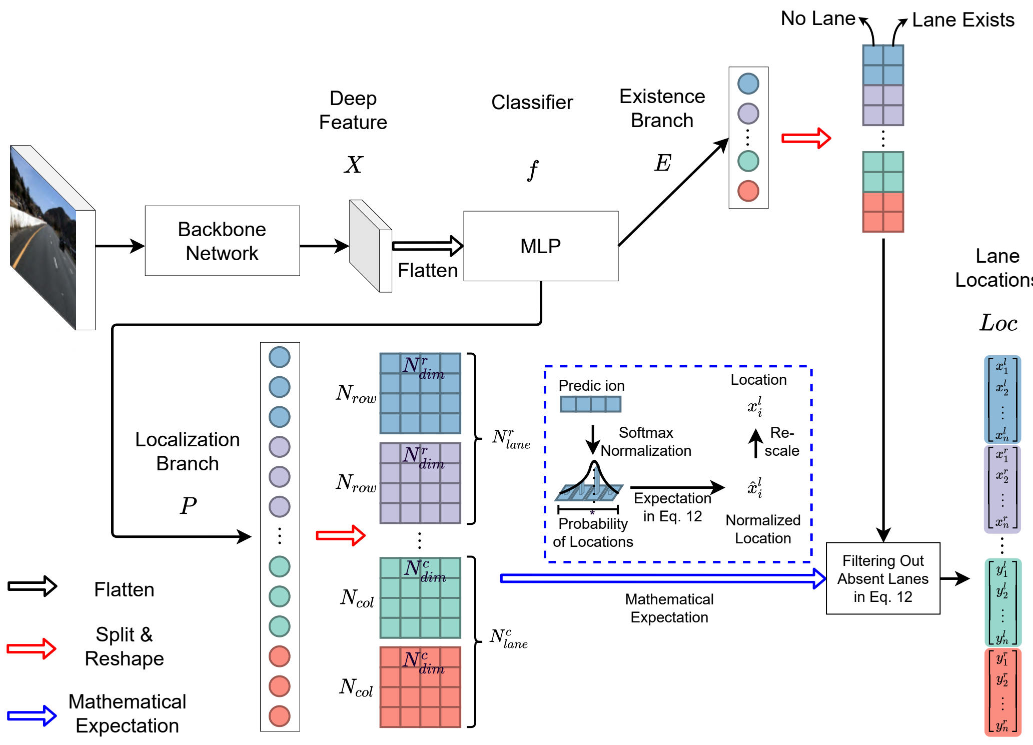


Figure 2.20: Ultrafast lane detection v2 architecture for lane detection [40]

2.3.2 Multi-task models

YOLOP and YOLOPv2

The core objective of YOLOP is to simultaneously execute three critical tasks: traffic object detection, drivable area segmentation, and lane detection. The architectural framework, illustrated in figure 2.21, encompasses a single encoder responsible for feature extraction and deploys three dedicated decoders for task-specific processing. YOLOP represents a pioneering achievement as the first solution capable of real-time, concurrent processing of these three visual perception tasks on an embedded device like the Jetson TX2 while maintaining excellent accuracy [41].

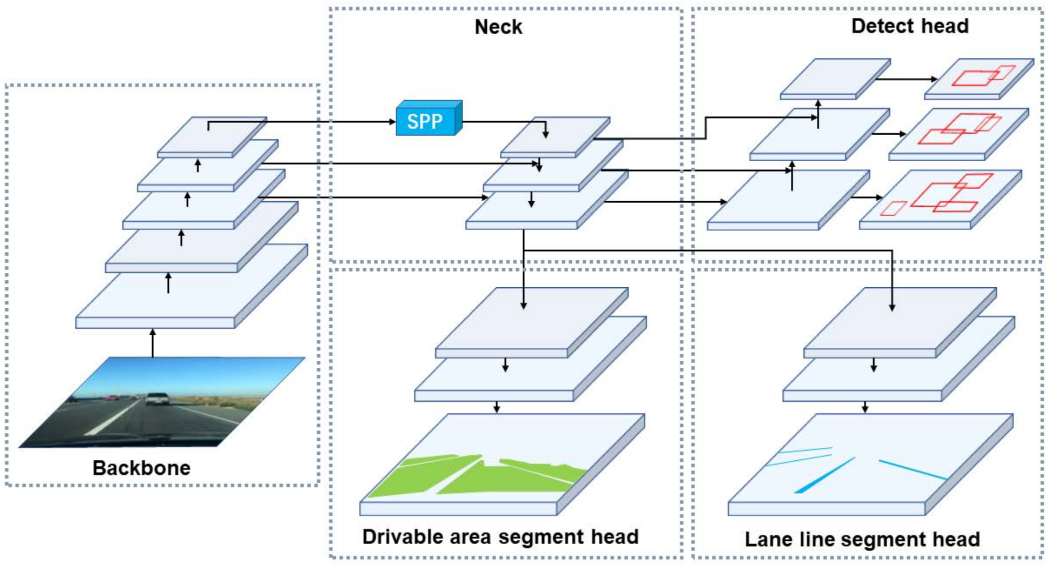


Figure 2.21: YOLOP multitask architecture [41]

YOLOPv2, on the other hand, is an improved version of YOLOP. It was designed to be better, faster, and stronger for panoptic driving perception. Inspired by YOLOv7 architecture, it has a better feature extraction backbone, more efficient structures for reasonable memory allocation, and a stable network design with powerful robustness for adapting to various scenarios [42].

TwinLiteNet

TwinLiteNet is an efficient and lightweight model designed for drivable area and lane segmentation. It utilizes ESPNNet-C as an information encoding block, efficiently generating feature maps. It integrates Dual Attention Modules within the network to capture global dependencies in both spatial and channel dimensions, thereby enhancing its contextual awareness. The ensuing feature map is subsequently channeled through two dedicated encoder blocks, each tasked with a specific objective: Driveable Area Segmentation and Lane Detection [43]. Figure 2.22 illustrates the architecture of this network.

TwinLiteNet has been developed with inference speed as the primary focus. The cost of this approach is worse accuracy results compared to other SOTA models like YOLOP and YOLOPv2.

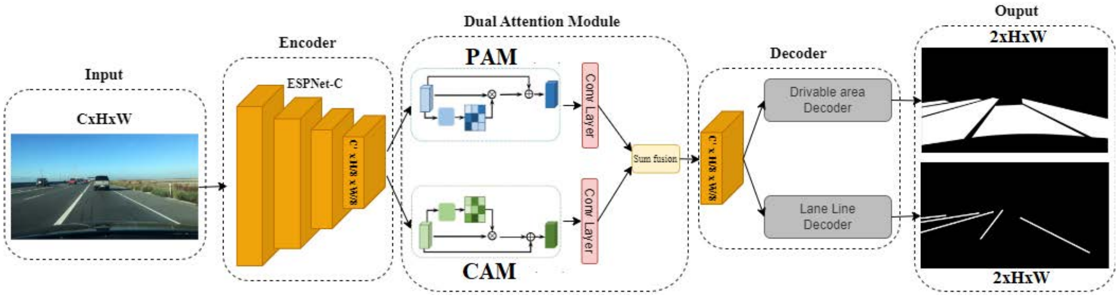


Figure 2.22: TwinLiteNet multitask architecture [43]

Summary - Deep learning models

In the scope of this dissertation, some single-task and some multi-task models for object detection, lane marking, and drivable area detection were analyzed.
The multi-task models for all tasks are the YOLOPv2 and the YOLOP.
TwinLiteNet is a multi-task model for lane marking and drivable area detection.
The single-task models for object detection are the YOLOv8, YOLOv7, and YOLOv5.
The single-task models for drivable area detection are the Mask2Former, the UperNet, and the SegFormer.
The single-task models for lane marking are the RESA, the O2SFormer, and the Ultrafast lane detection v2.

2.4 Evaluation metrics

In order to evaluate if a model performs better or worse than others, ways to measure performance are needed. A model performs better if it is faster and makes better predictions. A good way to measure speed, for comparison purposes, in computer vision is to evaluate how much time a model takes to process a given number of frames [4]. This measure comes in Frames Per Second (FPS):

$$\text{FPS} = \frac{\text{Frame count}}{\Delta t}. \tag{2.2}$$

A dataset containing ground truth for various tasks is required to assess whether a model outperforms others in terms of predictions. In this regard, BDD100k [44] serves as a suitable choice once it is focused on driving data and due to its versatility and widespread usage.

The BDD100k dataset is composed of two groups of labeled images. One group is used for object detection, lane marking, and drivable area segmentation, and the authors named it “100K Images”. The other group is used for semantic, instance, and panoptic segmentation, and its name is “10K Images” [45].

In the previous section, it was pointed out that this dissertation focuses on analyzing object detection, lane marking, and drivable area detection. To achieve this goal, both groups within the BDD100k dataset are well-suited for conducting experiments. This suitability arises from the fact that, for the "10K Images" group, panoptic segmentation includes lane marking and road segmentation, while object detection can be attained through instance segmentation. Given the "100K Images" group's greater alignment with the dissertation's scope, it is the one selected for application.

In this dissertation, lane marking and drivable area are treated as challenges of image segmentation, providing a means to harmonize various model outputs. The subsequent subsections will discuss the evaluation metrics for both object detection and image segmentation.

2.4.1 Object Detection

In object detection, the model predictions can be compared to the ground truth data in two key aspects: detections and localizations. Regarding detection, the predictions can be categorized as correct, represented by True Positive (TP); incorrect, represented by False Positive (FP); or missed, indicated by False Negative (FN). In terms of localization, the comparison involves assessing the agreement between the predicted areas (represented as "BB_p" in this section) and the ground truth areas (represented as "BB_g" in this section).

Precision and Recall

Both precision and recall are important metrics, and they are often used together. Achieving a balance between high precision and high recall is crucial for ADS.

Precision (equation 2.3) is a measure of the accuracy of positive predictions. A higher precision value indicates fewer false positives [4].

$$\text{Precision} = \frac{\text{TP}}{\text{TP} + \text{FP}} \tag{2.3}$$

Recall (equation 2.4), also known as sensitivity, measures the ability to identify all positive instances correctly. A higher recall value indicates fewer false negatives [4].

$$\text{Recall} = \frac{\text{TP}}{\text{TP} + \text{FN}} \tag{2.4}$$

Intersection over Union (IoU)

Localization is the process of accurately identifying and locating different objects by predicting bounding boxes around them. The IoU metric (equation 2.5) is commonly employed to assess the accuracy of predicted bounding boxes. IoU calculates the overlapping area between the ground truth and the predicted bounding box, providing a measure of their similarity [4].

$$\text{IoU}(BB_p, BB_g) = \frac{|BB_p \cap BB_g|}{|BB_p \cup BB_g|} \tag{2.5}$$

F_1 Score

Since both precision and recall are important metrics, F_1 score (equation 2.6) combines these metrics with a harmonic mean [4].

$$\mathrm{F}_1 = 2 \times \frac{\text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{2.6}$$

Macro-averaged F_1 Score

In order to get a value for all classes, thus the detector itself, one method is to average the F_1 values of all classes, as given by equation 2.7.

$$\mathrm{mF}_1 = \frac{\sum_{q=1}^{Q} \mathrm{F}_1(q)}{Q} \tag{2.7}$$

Average Precision (AP)

By plotting precision on one axis and recall on the other, a precision-recall curve can be created [4]. In this curve, specific values can be selected based on detections with an IoU greater than a threshold α. The AP (equation 2.8) is then calculated as the area under this precision-recall curve [4]. In equation 2.8, k is the number of inferences.

$$\mathrm{AP@}\alpha = \sum_{i=2}^{k} (r_i - r_{i-1}) \times p_i \tag{2.8}$$

Mean Average Precision (mAP)

The mAP (equation 2.9) is a metric used to measure the detector's accuracy in all the classes and is given by the average of all AP calculated for each class [4]. This is the metric used in the platform "Papers With Code"[1] for object detection.

$$\mathrm{mAP} = \frac{\sum_{q=1}^{Q} AP(q)}{Q} \tag{2.9}$$

2.4.2 Image segmentation

In this dissertation, lane marking and drivable area detection are employed as distinct categories within the context of image segmentation, and their performance is assessed separately. Consequently, similarly to the outputs generated by multi-task networks, two masks are evaluated: one for lane marking and one for the drivable area. Furthermore, when the network has the ability to differentiate between multiple instances within a class, these instances will be consolidated and treated as a form of semantic segmentation. Therefore, in this section, the explored metrics are for this kind of segmentation.

[1]https://paperswithcode.com

Pixel Accuracy (PA)

PA (equation 2.10) is the simplest metric, merely calculating the ratio of correctly classified pixels to the total number of pixels [46].

$$\mathrm{PA} = \sum_{i=0}^{k} \frac{p_{ii}}{\sum_{j=0}^{k} p_{ij}} \tag{2.10}$$

Where k is the number of classes -1 and p_{ij} is the number of pixels of class i inferred to belong to class j. If $j = i$, it means that the inferred class is correct, therefore, p_{ii} represents the TP pixels.

Mean Pixel Accuracy (mPA)

mPA (equation 2.11) is a slight improvement on PA, as it computes the ratio of correct pixels on a per-class basis and then averages this ratio over all classes [46].

$$\mathrm{mPA} = \frac{1}{k+1} \sum_{i=0}^{k} \frac{p_{ii}}{\sum_{j=0}^{k} p_{ij}} \tag{2.11}$$

Mean Intersection over Union (mIoU)

mIoU (equation 2.12) serves as the standard metric for segmentation tasks. It determines the ratio between the intersection and union of two sets: the ground truth and the predicted segmentation. This ratio can be expressed as the number of TP (intersection) divided by the sum of TP, FN, and FP (union). mIoU is calculated for each class and then averaged. It is important to note that this IoU calculation differs from the IoU in equation 2.5; this is a pixel-wise IoU, not a bounding-box IoU.

$$\mathrm{mIoU} = \frac{1}{k+1} \sum_{i=0}^{k} \frac{p_{ii}}{\sum_{j=0}^{k} p_{ij} + \sum_{j=0}^{k} p_{ji} - p_{ii}} \tag{2.12}$$

This is the metric used in the platform “paperswithcode” for image segmentation.

Summary - Evaluation metrics

There are multiple options for evaluating object detection, and image segmentation. In this dissertation, the chosen metrics are those that are also used in the “papers with code” website: for object detection, the mAP, for image segmentation, the mIoU.
The car detection will be evaluated using the object detection metrics.
The lane marking and the detection of the drivable area will be evaluated using image segmentation metrics.

Chapter 3

Experimental infrastructure

This chapter describes the tools used in this dissertation, both in hardware and software. The hardware includes computing and data acquisition devices. The software comprises the available deep learning libraries and Software Development Kit (SDK).

3.1 Hardware

The hardware outlined in the next sections was used throughout the development and deployment of the dissertation's software, alongside with data acquisition.

3.1.1 Software development and deployment

Three devices were accessible for software development, with one also for deployment. These devices include an Nvidia Jetson Xavier AGX and a laptop.

3.1.2 Nvidia Jetson AGX Xavier

The NVIDIA Jetson Xavier AGX, illustrated in figure 3.1, is a powerful and advanced single-board computer designed specifically for AI and edge computing applications. It is often used in robotics, autonomous vehicles, industrial automation, and other scenarios where real-time AI processing is essential.

Figure 3.1: Nvidia Jetson Xavier AGX Development Kit

The Jetson Xavier AGX integrates multiple components with excellent energy efficiency and performance, enabling it to achieve over 30 trillion operations per second while consuming under 30 watts of power. This remarkable capability positions it as an excellent choice for onboard utilization, facilitating AI computations to occur near the data source rather than depending on remote servers. The main specifications of this device are in Table 3.1 [47].

Table 3.1: Jetson Xavier AGX specifications

Specifications	
CPU	8-core NVIDIA Carmel Armv8.2 64-bit
GPU	Nvidia Volta architecture with 512 Nvidia CUDA cores and 64 Tensor cores
RAM & VRAM	32GB 256-bit LPDDR4x
Storage	32GB eMMC 5.1 + 250GB SSD

This board is supported by Nvidia JetPack SDK, which provides a full development environment for hardware-accelerated AI-at-the-edge development, as well as higher level SDK such as DeepStream for streaming video analytics, Isaac for robotics, and Riva for conversational AI [48].

The laptop

The provided laptop is a Gigabyte Aero 15 KD, illustrated in figure 3.2. This platform stands in stark contrast to the Jetson, allowing for testing the developed software across diverse environments. The main specifications of this laptop are presented in Table 3.2.

Figure 3.2: Gigabyte Aero 15 KD

Although the GPU might not be the top-tier GPU in Nvidia's lineup, it still offers substantial AI capabilities and a good balance between performance and power efficiency.

3.1.3 Data acquisition

To obtain the data for inference, the LAR has a set of four synchronized cameras, the e-CAM130-CUXVR from e-con Systems, illustrated in figure 3.3, capable of getting 4k images at 30 fps.

Table 3.2: Gigabyte Aero 15 KD specifications

Specifications	
CPU	Intel Core i7-11800H
GPU	Nvidia RTX 3060 laptop
RAM	16GB DDR4
VRAM	6GB
Storage	1TB SSD

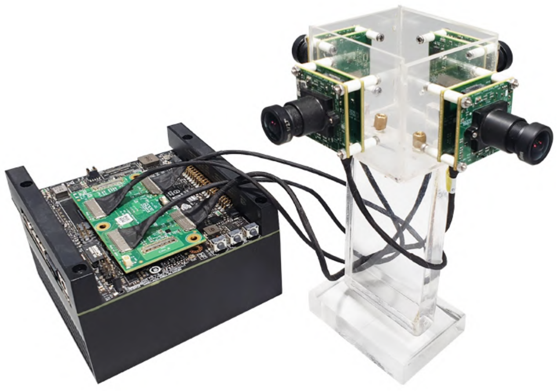


Figure 3.3: e-CAM130-CUXVR plugged into Nvidia Jetson AGX Xavier

Nonetheless, there are compatibility challenges with the camera drivers when using the latest Jetpack versions, as detailed by the author in [49]. As the gathered data serves qualitative purposes exclusively within the scope of this dissertation, prioritizing image quality isn't paramount. To circumvent the driver issue, the Logitech C270 HD webcam, accessible at LAR, will be employed. This webcam is visualized in figure 3.4.

Figure 3.4: Logitech C270 HD

3.2 Software

The following software descriptions provided the base upon which the programs for the final solutions were built. This software is indispensable for introducing modularity to the solutions and simplifying the complexities of the deep learning aspect.

3.2.1 Robot Operating System

Robot Operating System (ROS) is an open-source framework designed to facilitate robot software development. It provides tools, libraries, and conventions that help software developers create and manage complex robotic systems. ATLASCAR2 employs ROS as part of its software architecture [50].

Its functionalities encompass communication, enabling data exchange between different segments of a robot's software system, and modularity, achieved by encapsulating diverse functionalities into discrete software modules referred to as "nodes". With communication capabilities in play, these nodes can transmit and receive data using ROS topics. Figure 3.5 shows a schematic example of communication between two nodes using ROS.

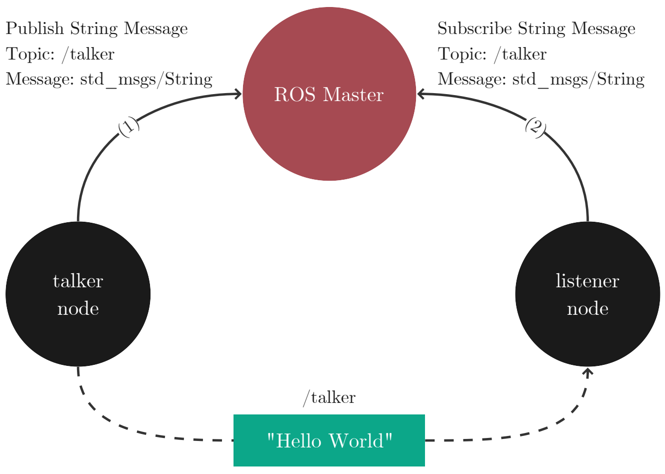


Figure 3.5: Schematic example of the use of ROS

3.2.2 Deep learning frameworks

Utilizing a deep learning framework brings advantages as it handles intricate aspects of neural network implementation while also capitalizing on performance optimization through hardware acceleration for faster computations. Furthermore, most existing pre-trained models have been developed using deep learning frameworks [51].

The most popular frameworks are Tensorflow, Keras, and PyTorch. While Keras boasts user-friendliness, it falls short in terms of performance. PyTorch and TensorFlow, on the other hand, exhibit comparable performance, yet PyTorch is often noted for its user-friendly nature since it is more "Pythonic" [51]. Moreover, as per "Papers With Code", PyTorch emerges as the framework with a more extensive range of implementations [52]. Hence, for the primary deep learning framework within the developed software of this dissertation, PyTorch stands as the selected preference.

3.2.3 Nvidia tools

Nvidia delivers software tools to both deep learning tasks and Jetson devices. Specifically for Jetson, there is JetPack; for deep learning inference, there is DeepStream.

JetPack

JetPack functions as a software stack, providing developers with tools, libraries, and frameworks to design and implement AI and deep learning applications on Jetson devices. Among its components, JetPack features a customized Linux distribution optimized for Jetson hardware. Additionally, it encompasses inference-oriented tools such as the DeepStream framework for real-time video analytics and the TensorRT inference optimizer [53].

DeepStream SDK

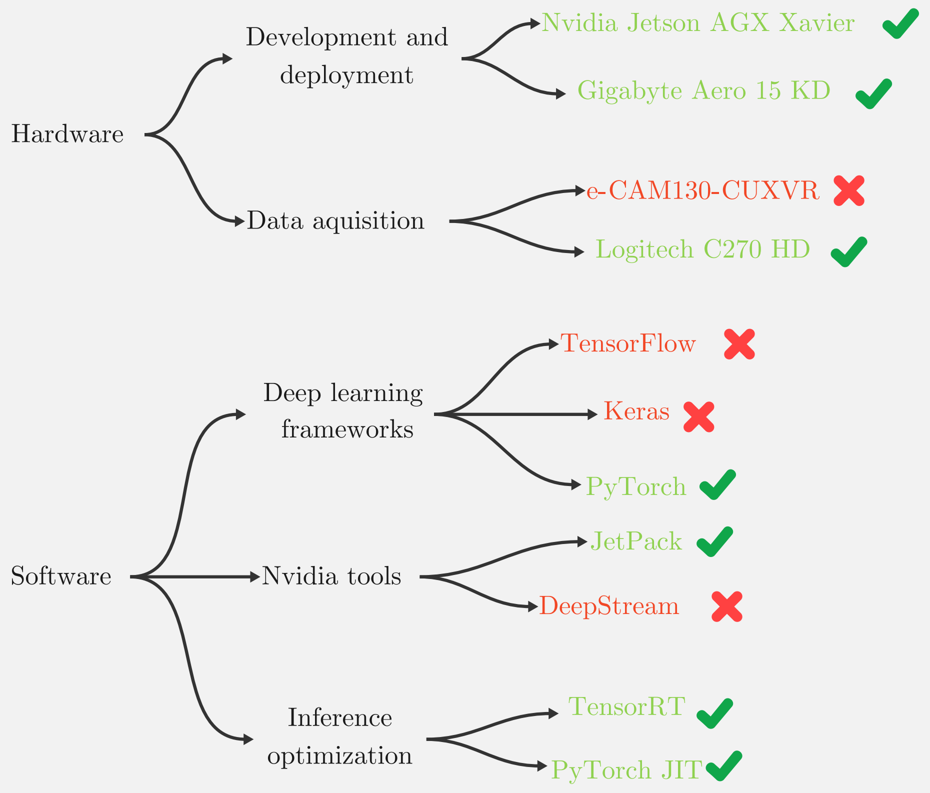
DeepStream is specifically developed to facilitate the construction and deployment of real-time AI-powered video analytics applications. It offers a framework and a range of tools that allow developers to establish intelligent video processing pipelines that can handle multiple video streams. Additionally, DeepStream encompasses a collection of Application Programming Interface (API), including support for ROS, providing developers with enhanced flexibility and integration capabilities [54].

3.2.4 Model optimization for inference

Model optimization is crucial in deep learning as it aims to enhance the efficiency and speed of neural network inference. Two prominent approaches for optimizing models are TensorRT and PyTorch JIT, and both are valuable tools for optimizing neural network models. TensorRT excels in optimizing models for Nvidia GPUs with a focus on performance, while PyTorch JIT offers a more integrated and flexible approach for optimizing PyTorch models across different hardware platforms. The optimization provided by TensorRT, however, relies on specific hardware, implying that if a model is optimized for a particular GPU model, it may not be compatible with other GPUs. On the other hand, TensorRT optimizes ahead of time, meaning that when it is time for inference, the model is immediately ready, which is not true for PyTorch JIT, in which the optimization is done during a few initial inferations, often referred to as warmup.

Summary - Experimental infrastructure

This chapter covered the software and hardware resources available. Below is a summary diagram with both topics for a better overview of what was used and what was not.



Chapter 4

Proposed solution

This chapter focuses on a pivotal component of the dissertation – a versatile application designed to run multiple deep-learning models with distinct tasks seamlessly. This application serves two primary functions: facilitating the deployment of deep-learning models on ATLASCAR2's inference unit and evaluating the performance of these models. It plays a central role in the project, enabling the fulfillment of the dissertation's objectives. As there is no pre-build solution with the necessary versatility and functionalities, the chapter proceeds to explore the application's development process, highlighting its adaptability for dual deployment in both the ATLASCAR2 integration and model performance evaluation contexts. The source code is available at GitHub[1].

4.1 Inference architecture

The ATLASCAR2 software architecture has been developed using the ROS framework. As explained in section 3.2.1, this framework has a fairly versatile inter-process communication system. As such, the inference software's architecture should incorporate this framework to communicate with other onboard devices.

The developed architecture has to receive the data sent by ATLASCAR2's sensors through ROS topics and perform the inference with a specific deep learning model. After that, the architecture has to send the results through ROS topics. The general idea is illustrated in figure 4.1. This architecture comprises four main blocks, as described next in more detail.

4.1.1 Data sender

The Data sender block represents ROS nodes that send the data from ATLASCAR2's sensors. To have more freedom while developing this architecture, and since only image data was explored for the scope of this dissertation, this node is not from any ATLASCAR2 device or sensor but a local one that only reads images or videos and sends them. The reason for this is to iterate the application without using ATLASCAR2. This could be done through ROS bags, but using this node allows a broader variety of data and image sizes and resolutions.

[1]https://github.com/GoncaloR00/perception_with_multi-task_neural_networks/

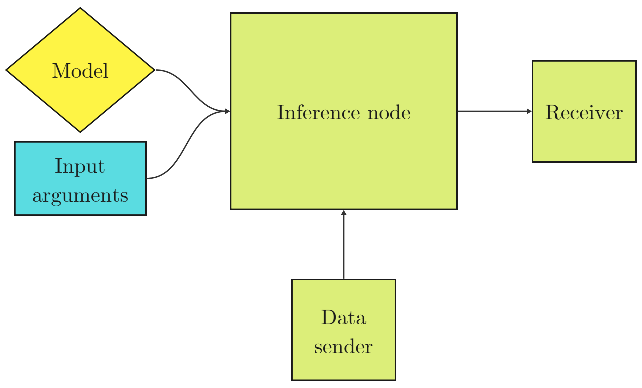


Figure 4.1: Basic architecture for deep learning based inference unit

4.1.2 Model and Input arguments

These blocks do not represent ROS nodes. The yellow rhombus labeled "Model" refers to the neural network model file containing its weights. The blue rectangle labeled "Input arguments" represents the needed parameters/arguments for inference, such as the model file path.

4.1.3 Inference node

The inference node is the main block of this architecture and is where the inference is done. This has to receive the data and send the inference results. These results should always have the same structure so that they can be transmitted through ROS topics consistently. This way, the work of a receiver is much easier since the results format is always the same, no matter the used model.

To have more flexibility, this node is divided into two parts: an inference solution and an inference manager (figure 4.2).

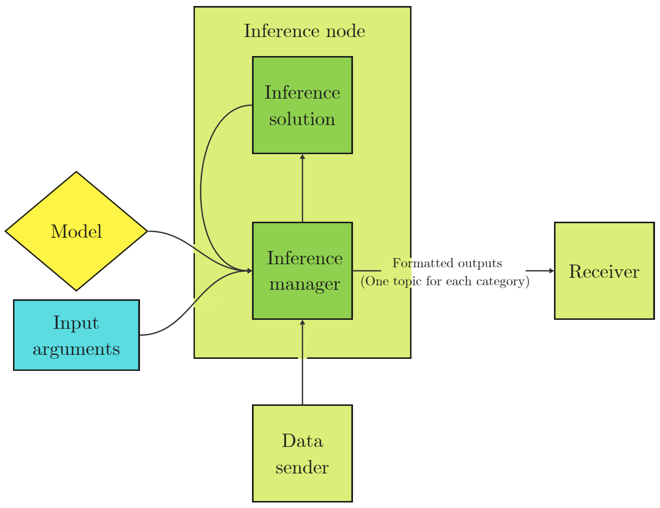


Figure 4.2: Internal parts of the inference node - inference solution and inference manager

The inference solution is a Python Class with a constructor and two more methods. The constructor loads the model using the model file, the inference parameters, and a sample of data for inference to know its shape. One of the other methods is "load_image", which loads the image and executes all the needed transformations. The other method is called "infer" and returns the inference output of the loaded image in a standardized way. The inference could be made in a few different ways, as shown in section 4.3.

The inference manager receives the model weights and inference parameters and subscribes to the ROS topic in which data is being sent. In the first execution, it starts an instance of the inference solution, loading the model weights, the inference parameters, and a sample of data, as some model builders require the original data shape to load the model. After that, each time a new ROS data message arrives, the inference solution executes the inference and outputs the results in a standardized format. These results are then sent through ROS topics.

4.1.4 Receiver

The Receiver block represents the nodes that receive the inference results and make something with that data. Since there are no nodes ready to receive the inference results in the ATLASCAR2, two nodes were made to show this architecture's capabilities and provide examples for future developments. One version shows every result that arrives, and the other discards results that are too much delayed in relation to the original data.

The current receivers are capable of displaying semantic, instance, and panoptic segmentation, as well as 2D object detection.

4.2 Architecture for performance evaluation

To fulfill the goals of this dissertation, it is necessary to develop an additional architecture to evaluate deep learning models.

This evaluation architecture uses the inference node from the previous architecture and combines the data sender and receiver in the same node with the name "Inference eval", as illustrated in figure 4.3. However, the data sender and receiver are slightly modified in this architecture.

The modified data sender sends images from the dataset, which is, for this dissertation, the BDD100k, as explained in section 2.4, when the modified receiver requests. The reason is that when multiple models are used in parallel, the models' inference times are different. The data sender can only send data when every model finishes the inference, and all results arrive at the receiver.

The modified receiver does not show the results in real-time. Instead, it saves the results in a similar structure to the BDD100k dataset with the time stamps before and after the inference.

The actual evaluation is done with post-processing by comparing the BDD100k files with the files created by the receiver. The used metrics are explained in section 2.4, and the results of this evaluation are saved in a spreadsheet.

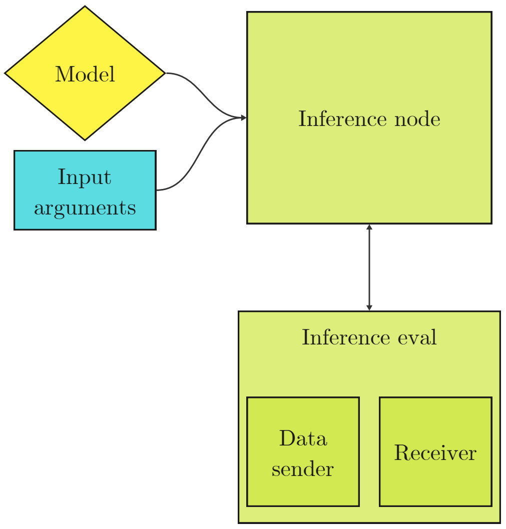


Figure 4.3: Basic architecture for performance evaluation

Summary - Architectures

For the scope of this dissertation, two similar architectures were developed. One is targeted at ATLASCAR2, and the other is for evaluation purposes. Both architectures are capable of working with multi-tasked and single-tasked models, as well as multiple models simultaneously.

4.3 Inference solutions

As mentioned in section 4.1.3, the inference solution could be implemented in a few distinct ways. Since these architectures are used in the Nvidia Jetson Xavier AGX, it is interesting to research solutions made by Nvidia and other alternatives.

4.3.1 Nvidia solutions

At the time of writing this dissertation, Nvidia had developed some solutions to perform inference in devices like Jetson. The following sections explore the up and downsides of some solutions.

DeepStream SDK

As explained in section 3.2.3, DeepStream has a collection of API and was used in previous works. This is a highly optimized solution since it was developed by Nvidia staff with lots of experience.

The downside of DeepStream is that, in addition to being a closed source software, its API for ROS is only compatible with ROS2 and only works for image classification and object detection [55]. This could be solved by adapting the API and deepstream_python_apps, [56], to have compatibility and more possible tasks. But beyond this limitation, utilizing non-supported models is not a trivial process, requiring significant time to make the needed adaptations or use unofficial implementations like [57], when available.

Jetson-inference

Jetson-inference is part of an introductory project called "Hello AI World" and has a submodule for ROS, the ros_deep_learning ROS package. It is easy to use for its purpose, but the main goal of this project is to introduce people to AI. As such, it is made to work with specific deep learning networks, and adapting this module to work with other networks could be very difficult [58, 59].

ISAAC SDK

ISAAC SDK is designed to create sophisticated robotic systems that can perceive, reason, and act autonomously. It includes modules for various perception tasks such as camera input processing, image and point cloud analysis, object detection, tracking, and mapping. In addition to that, and since its purpose is to create robotic systems, ISAAC SDK has a good amount of API for ROS [60].

This solution could be used in two different ways. One is using API for DeepStream integration and API for ROS, working as a bridge between DeepStream and ROS, avoiding the compatibility problem. The other way is to build a new ISAAC module for each non-supported network.

The downside of ISAAC SDK is similar to DeepStream and Jetson-inference, which is the difficulty of using non-supported networks.

4.3.2 PyTorch framework

The alternative to using an Nvidia solution is to use a deep learning framework. As discussed in section 3.2.2, the chosen framework was PyTorch. This is a solution built more from scratch, which gives much more flexibility but is not as efficient as an Nvidia solution since it is thought to reach the level of optimization that took years to achieve.

The PyTorch solution was chosen due to time management. Even being the easiest path, some challenges, explored in section 4.4, had to be overcome due to the required versatility and compatibility.

Summary - Inference solutions

Nvidia offers some inference solutions that could be used in the proposed architectures. Still, none was used due to the difficulty of using non-supported networks and issues of time management. Instead, a custom solution was made based on the PyTorch framework.

4.4 Versatility and compatibility challenges

The chosen inference solution led to some challenges during its development. The following subsections explain each problem and the approach to solve them.

4.4.1 Different deep-learning networks

There are two problems when dealing with distinct deep-learning networks: one related to the model file and the other due to the input and output data from the model's inference:

Model file problem

Although this architecture mainly focuses on PyTorch models, there are different ways to store the network parameters. Therefore, there are also multiple ways of loading a model.

A model can be saved by mapping each layer to its parameter tensor. This way, the model's structure is not saved. Thus, to load the model, the Python class with the structure is necessary to initialize the model.

Some models can also be compiled, representing the model's structure with its parameters. Saving this compilation avoids the need for a Python class to initialize the model when loading.

The parameters can furthermore be saved with different precisions. The most common ones are FP32 and FP16.

Inference's data problem

Another problem with performing the inference of different deep-learning networks is that the input data shape and required transformations are not always the same. Also, different networks may not have the same output format.

The solution

Since the way models are loaded and the data is sent and received could differ for each model, the loading process and data transformations should depend on the used model. To make this happen, the approach was to create special modules, one for each model, and model loaders. Here is where the input arguments came in handy because they allowed passing to the inference solution not only the model location but also the module and model loader names. This way, the inference solution's main script can call the right module and model loader for a specific model file.

The modules are Python scripts with the purpose of solving the inference data problem: Different models' input requirements and different models' output formats. Therefore, to address the inference data problem, the modules perform two critical jobs: first, with a function called "transforms", which converts the received data into the necessary format for the model's input, and second, with a function called "output_organizer", which restructures and adapts the model's output to meet the requirements of the inference manager.

For the model file problem, one of the input arguments is the model loader name, which the inference solution can access, and then call the respective loader. This model loader receives the model path, loads the model, and returns the variable with the model, as well as parameters for inference like the model framework (eg.: PyTorch, ONNX...) and the precision (eg.: float32, float16...). This approach allows the use of other frameworks other than PyTorch. In figure 4.4 is an illustration of this solution.

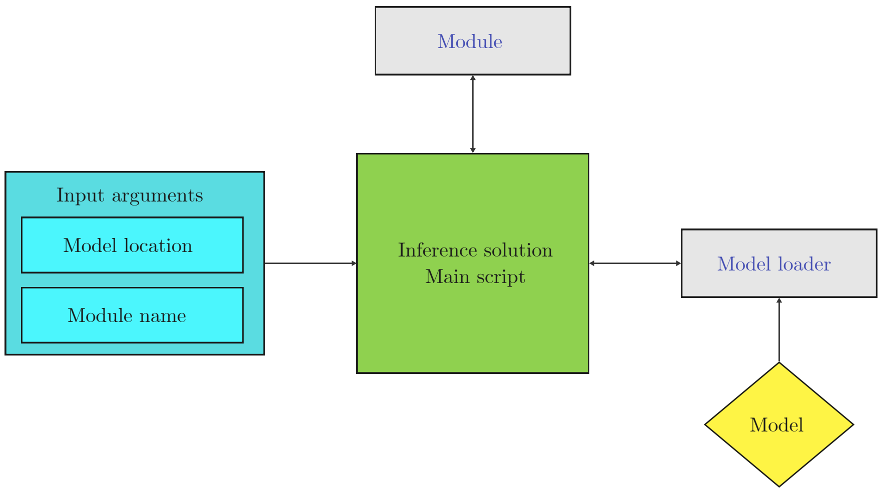


Figure 4.4: Modular design of the inference solution (one of the blocks in figure 4.2): abstraction of data processing and model loading from the main script to improve versatility

4.4.2 Different datasets

Certain models are trained using a specific dataset, while others are trained using a different dataset. This generates a problem related to the labels, which are generally different or are in a distinct order from dataset to dataset.

This issue can be solved in advance by changing the output layer to give the desired labels in the desired order, freezing all the remaining weights, and retraining the model. Another solution is to create conversion tables between datasets to avoid this process and keep the model as it is.

The chosen dataset for standardization was the BDD100k. Therefore, all the inference solution outputs are defined by this dataset labels.

To make the dataset conversions, YAML files were generated, containing labels and their corresponding IDs for each dataset. Additionally, files were created to establish the correspondences between labels across different datasets. As an illustrative example, Table 4.1 presents a segment of labels from both COCO and BDD100k for object detection, along with the corresponding label ID conversions from COCO to BDD100k.

When, for example, a model pre-trained in the COCO dataset outputs a 2, the model module accesses the conversion file and returns an 8. Then it accesses the BDD100k labels file, looks for the id=8, and returns “bicycle”.

4.4.3 Road lines representations

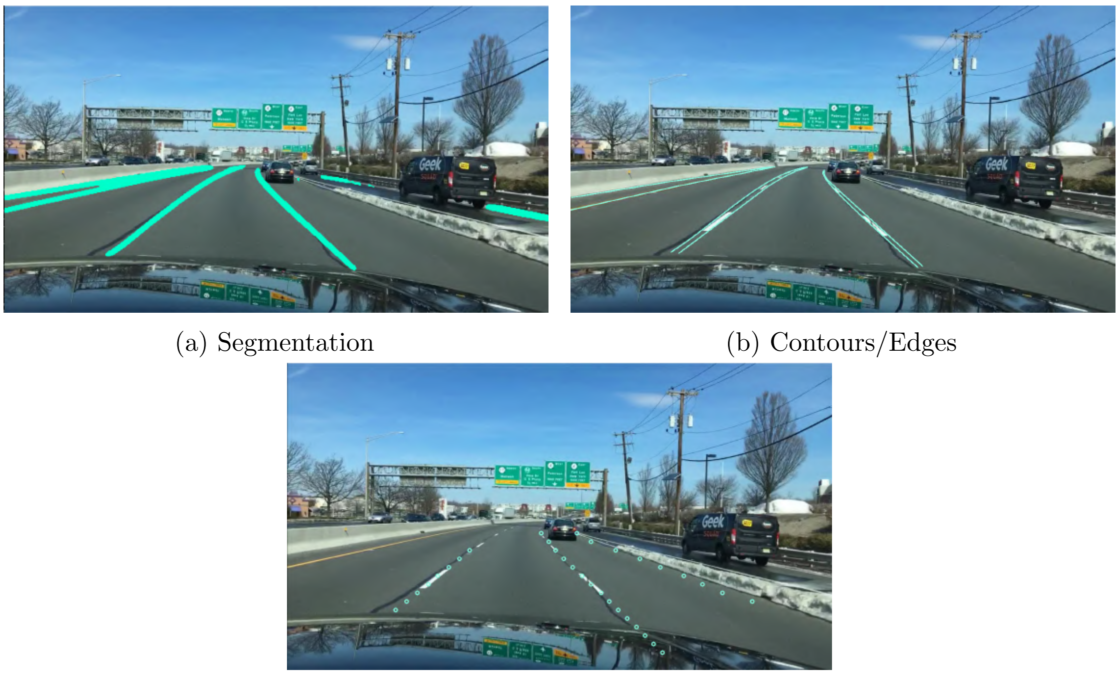
Three ways of representing road lines exist for the used models: the entire pixels of the road line through segmentation, contours, and points (figure 4.5).

The representation of road lines should be consistent to compare these different types of models. The easiest way is to transform all into pixels of the lines.

In order to transform the pixels of the edges into filed lane markers, two steps are made. The first is to do a dilate morphological operation to join the two lines of the

Table 4.1: Some COCO and BDD100k labels for object detection and conversions from COCO to BDD100k IDs

ID	COCO labels	BDD100k labels	Conversion
1	person	pedestrian	1: [1, 2]
2	bicycle	rider	2: [8]
3	car	car	3: [3]
4	motorcycle	truck	4: [7]
5	airplane	bus	5: [999]
6	bus	train	6: [5]
7	train	motorcycle	7: [6]
8	truck	bicycle	8: [4]
9	boat	traffic light	9: [999]
10	traffic light	traffic sign	10: [9]
11	fire hydrant	-	11: [999]
12	stop sign	-	12: [10]
999	-	other	-

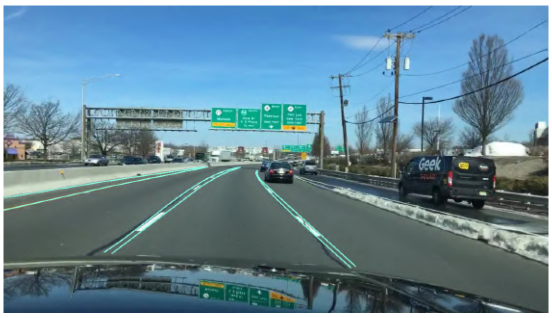

(a) Segmentation (b) Contours/Edges

(c) Points

Figure 4.5: Diverse approaches to road line representation across various datasets and deep learning models

edges, and the second is to do an erode process to remove thickness. The result of these operations can be seen in figure 4.6.

To transform the points into the pixels of the filed lane markers, there are two methods. In the first, the points are interpolated, and then more points are added. Then, for each


(a) Original image

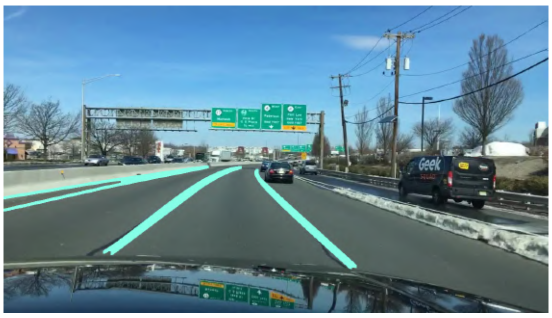
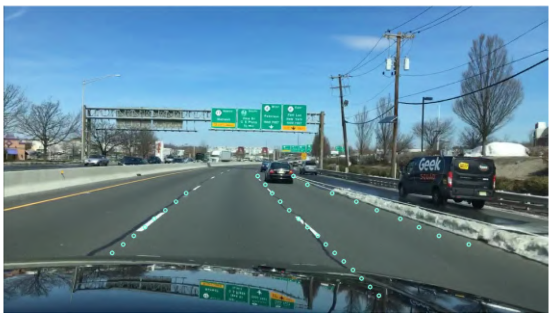

(b) Transformed image

Figure 4.6: Segmentation of road lines from contours in figure 4.5b

point, a circle is drawn, where the radius decreases with the height position of the point. The second method is to draw lines between sequential points, where the line thickness decreases with the height position of the points. The result of these operations can be seen in figure 4.7.


(a) Original image

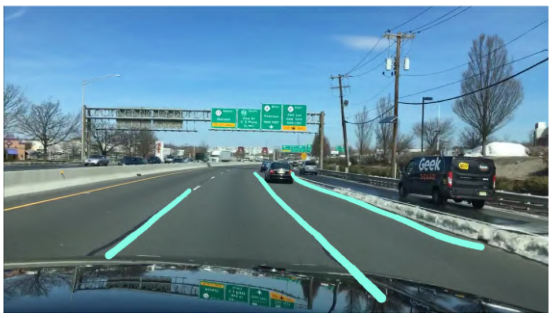

(b) Transformed image

Figure 4.7: Segmentation of road lines from points in figure 4.5c

Summary - Versatility and compatibility challenges

In order to get the inference solution to work properly, some problems had to be solved. Addressing these problems has given the architecture compatibility with a wider variety of pre-trained deep-learning models.

4.5 Final solutions

The diagrams of the architectures can now be more detailed with the inference solution completed and the associated problems solved. The final solution to implement in ATLASCAR2 is represented in figure 4.8, and for evaluation in figure 4.9.

These two solutions can be used with and without a Jetson device since no Jetson tools were used. In addition to that, everything is built within the ROS framework, which means that not all nodes need to be on the same machine. This feature allows the use of multiple devices for inference in the case of using various models simultaneously.

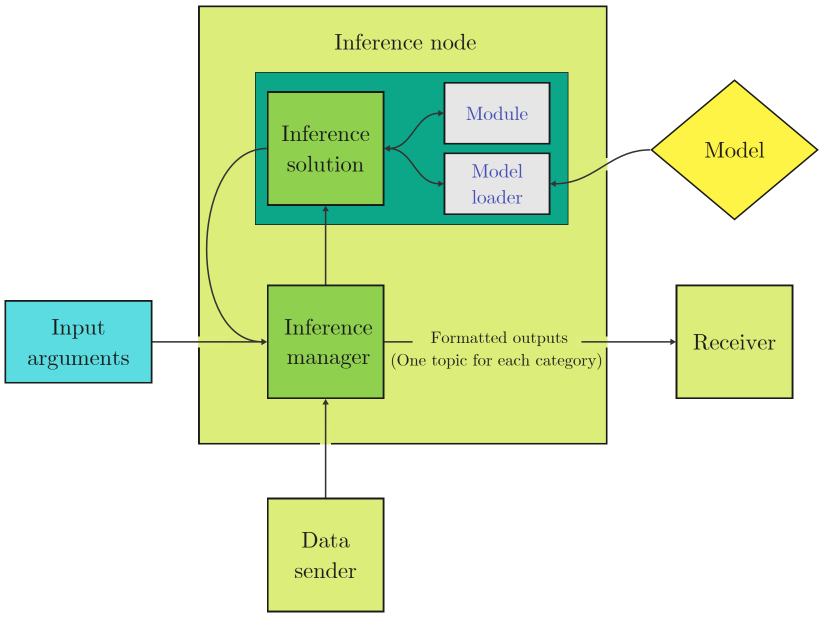


Figure 4.8: Final solution to implement in ATLASCAR2

The following sub-sections will explore the workflow of implementing models for inference and explain how to execute the nodes.

4.5.1 Implementation of deep-learning models

As discussed in section 4.4, the chosen inference solution requires a module and a model loader. Additionally, the model can be optimized to improve the inference speed by compiling it into TensorRT to ensure that the models are in the best condition for inference.

Module and model loader

Each model has its own module, but all the used modules should follow the same structure. For the main script of the inference solution to work, modules should have two functions.

The first function is “transforms”, which transforms the input data into a format compatible with the model. It takes as input the data, a string with the device’s name where the inference will occur (e.g., “cpu”), and a boolean value indicating if the data needs to be in half precision. Since, for the scope of this dissertation, only images are used, the outputs are the transformed image, the original image’s shape, and the shape of the transformed image. Listing 4.1 provides an example of this function.

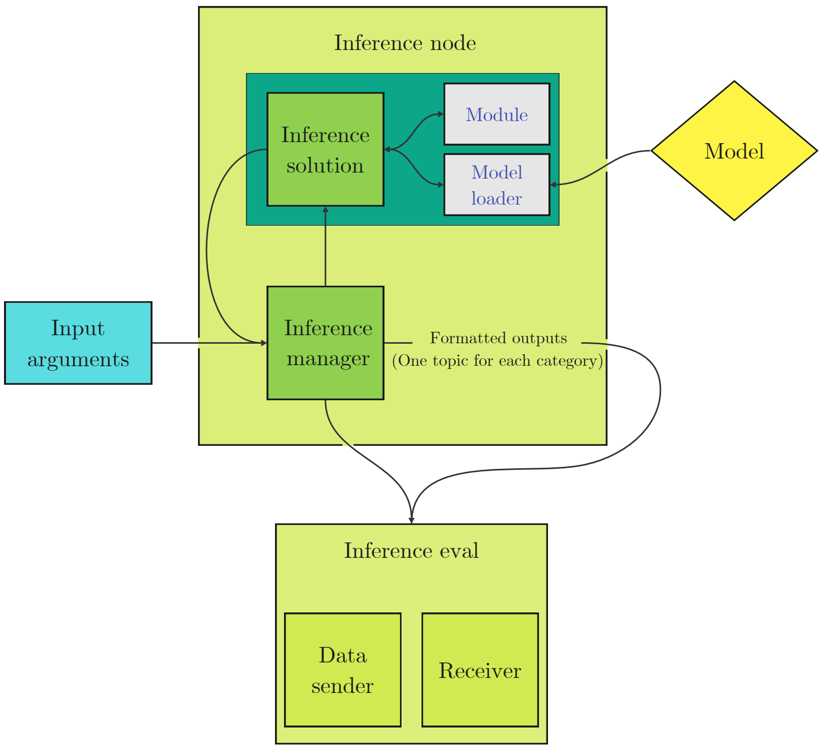


Figure 4.9: Final solution for performance evaluation

Listing 4.1: Example of a "transforms" function

```
import torch
import cv2
model_img_size = (640, 640)
def transforms(image, device: str, half: bool):
    original_img_size = (image.shape[0],image.shape[1])
    img_0 = cv2.resize(image, (model_img_size[1], model_img_size[0]))
    img = torch.Tensor(img_0)
    img = img.permute(2,0,1)
    img = img.unsqueeze(0)
    img = img.to(device)
    if half:
        img = img.half()
    img /= 255
    return img, original_img_size, model_img_size
```

The other function is "output_organizer", which receives and organizes the output from the inference to a specific format. It takes the output from the inference as input, the original image shape, and the transformed image shape. The output is a list, and its content depends on the intended use of the architecture. Listing 4.2 provides an example of this function.

Listing 4.2: Example of a “output_organizer” function

```
from module_utils import pred2bbox
from module_utils import pred2seg
model_img_size = (640, 640)
def output_organizer(original_output, original_img_size, model_img_size):
    pred = original_output
    det2d_class_list, det2d_list = pred2bbox(pred, original_img_size,
                                             model_img_size)
    seg_class_list, seg_list = pred2seg(pred, original_img_size,
                                        model_img_size)
    detections = (det2d_class_list, det2d_list)
    segmentations = (seg_class_list, seg_list, "semantic")
    return detections, segmentations
```

Currently, only 2D object detection and segmentation are made, but if, for example, 3D object detection needs to be added, the output list will have one more element. The catch is that all the modules need the same output structure, and the inference manager adapted accordingly.

The model loader receives as input the shape of the original image, the shape of the transformed image, and the model path. After loading the model, it returns the loaded model, three boolean variables to check if the model is in CUDA, is in half-precision, and was optimized with TensorRT, and one string storing the name of the framework of the model. Listing 4.3 provides an example of this function.

Listing 4.3: Example of a “load” function

```
import torch
def load(original_img_size, model_img_size, model_path):
    model = torch.jit.load(model_path)
    model.to('cuda')
    model.half()
    cuda = 1
    half = 1
    engine = 0
    framework = 'torch'
    return model, cuda, half, engine, framework
```

TensorRT optimization

Since the TensorRT optimization depends on the used hardware, the optimized model only works on the hardware where it was optimized. Therefore, in order to convert a model to TensorRT, the first step is to install all TensorRT dependencies.

Two distinct pathways are available to perform the conversion process with PyTorch models. The first approach involves the utilization of an additional package, such as “Torch-TensorRT”, offering remarkable convenience by enabling direct conversion from a TorchScript model to TensorRT compilation. This package features a dedicated function called “compile”. The disadvantage of this method is that it is more restrictive in terms of compatibility. Alternatively, the second method requires an initial conversion of the model to the ONNX framework, followed by compilation into TensorRT. Various options exist to facilitate the transition from ONNX to TensorRT; however, in the context of this dissertation, the preferred choice is the “Polygraphy” framework [61] due to its convenience of use.

Figure 4.10 is a representation of the two methods used to optimize models with TensorRT, and listings 4.4 and 4.5 are illustrative examples of the use of the “Torch-TensorRT” and “Polygraphy” frameworks, respectively.

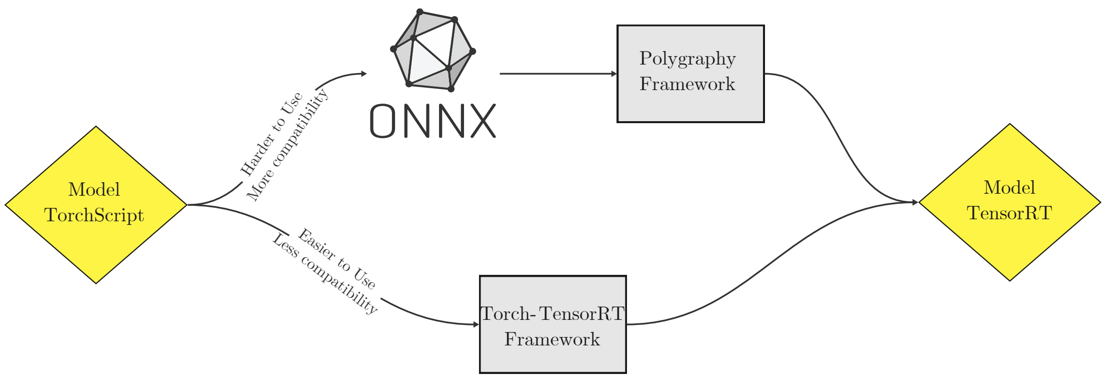


Figure 4.10: The methods used to optimize a model with TensorRT

Listing 4.4: Example of the use of “Torch-TensorRT”

```python
import torch
import torch_tensorrt
def model2trt(load_model, model_img_size):
    load_model = load_model.half()
    traced_model = torch.jit.trace(load_model,
                       [torch.randn((1, 3, model_img_size[0],
                          model_img_size[1])).to("cuda")])
    trt_model = torch_tensorrt.compile(
        load_model,
        inputs = [torch.randn((1, 3, model_img_size[0],
                              model_img_size[1]), dtype=torch.float16)],
        enabled_precisions = {torch.float16},
        device = torch_tensorrt.Device("cuda"),
        workspace_size=4194304
        )
    return trt_model
```

Listing 4.5: Example of the use of "Polygraphy"

```python
from polygraphy.backend.trt import EngineFromNetwork, NetworkFromOnnxPath
from polygraphy.util import save_file as save_engine

def engine_builder(ONNX_model_path, save_path):
    pre_engine = EngineFromNetwork(NetworkFromOnnxPath(ONNX_model_path))
    build_engine = pre_engine.call_impl()
    save_engine(engine.serialize(),save_path)
```

4.5.2 Launching nodes

The best way to start the nodes is through ROS launch files. Having launch files for the inference applications not only automates the process of starting every node, but also allows full integration within the ATLASCAR2 framework since other higher-level launch files, present in ATLASCAR2, can start these launch files.

For the scope of this dissertation, two launch files were created: one for illustration, using the architecture for ATLASCAR2, and the other for evaluation, using the architecture for evaluation. These two, however, follow the same structure.

At the beginning of each launch file are defined the arguments that will be sent to the inference node. In the architecture diagrams, figures 4.8 and 4.9 correspond to the "Input arguments" block. These arguments are the ROS topic by which the data is sent, the model loader and module's names, and the model's directory. A generic example is shown in listing 4.6.

Listing 4.6: Inference node input arguments in a launch file

```xml
<arg name="data" default="data_topic"/>
<arg name="fn_model" default="module_name"/>
<arg name="ml_model" default="model_loader_name"/>
<arg name="mp_model" default="model_directory"/>
<arg name="n_images" default="10000"/>
```

After the arguments definition, it is defined which nodes will be initialized and their namespaces.

In the case of the launch file for illustration, a sender node is needed. In the launch file, that is achieved using something similar to listing 4.7. This node does not need a namespace.

Listing 4.7: Data sender node in a launch file for illustration

```xml
<node pkg="basic_sender" name="image_sender" type="sender_node.py"/>
```

The other nodes are within a namespace. This namespace has a required format for the receiver to work correctly, and it is the data source, followed by a "/" followed by the used models separated by an "|". A namespace could be "frontcamera/model1|model2".

For each group, either an evaluation or a receiver node is initialized. Therefore, if there is more than one inference node within the same group, more than one inference will be evaluated or displayed. However, more than one group can be initialized simultaneously, which is especially important for illustration since this allows watching multiple inference options simultaneously. For illustration, the nodes can be set as in listing 4.8.

Listing 4.8: Inference and receiver nodes in a launch file

```
<group ns='data/model1|model2'>
    <node pkg="basic_receiver" name="image_plotter"
    type="sync_receiver_node.py" output="screen"/>
    <node pkg="inference_manager" name="inference_node_model1"
    type="inference_node.py" output="screen"
    args="-fn $(arg fn_model1) -ml $(arg ml_model1)
    -mp $(arg mp_model1) -sr $(arg data)"/>
    <node pkg="inference_manager" name="inference_node_model2"
    type="inference_node.py" output="screen"
    args="-fn $(arg fn_model2) -ml $(arg ml_model2)
    -mp $(arg mp_model2) -sr $(arg data)"/>
</group>
```

For evaluation, the nodes can be set as in listing 4.9. The additional parameters "obj", "l", and "da" are boolean and are used to activate object detection, lane detection, and drivable area, respectively, which are the ones evaluated in this dissertation.

Listing 4.9: Inference and evaluation nodes in a launch file

```
<group ns='data/model1'>
    <node pkg="inference_eval" name="eval"
    type="inference_eval.py" output="screen"
    args="-fn FolderName -nimg $(arg n_images) -obj 1 -l 1 -da 1"/>
    <node pkg="inference_manager" name="inference_node_model1"
    type="inference_node.py" output="screen"
    args="-fn $(arg fn_model1) -ml $(arg ml_model1)
    -mp $(arg mp_model1) -sr $(arg data)"/>
</group>
```

Overall summary

This chapter introduces two similar architectures designed for ATLASCAR2 and for evaluation purposes. These architectures are versatile, capable of accommodating both multi-tasked and single-tasked models, and they enable the simultaneous use of multiple models. During development, the need for an inference solution arose. Although Nvidia offers inference solutions, they were not employed due to challenges related to unsupported networks and time constraints. Instead, a custom solution was developed using the PyTorch framework.
To ensure the proper functioning of the inference solution, challenges due to different model formats, input requirements, and output formats were addressed, resulting in improved compatibility with a wider array of pre-trained deep-learning models. In addition, the application developed interfaces with ROS topics for data receiving and streaming and employs a modular architecture to enhance robustness and simplicity, with different components designed to the features of each model.
The installation steps of the application developed in this dissertation are available in appendix D.

Chapter 5

Tests and results

The purpose of this chapter is to test the application outlined in Chapter 4, demonstrating its capabilities, while also evaluating the neural networks introduced in Chapter 2 and analyzing the corresponding outcomes. To achieve this, the chapter initiates with an overview of the procedure for obtaining the model files for testing, followed by the application's testing process, and concludes with the presentation of results and their subsequent analysis.

5.1 Model implementations

The dissemination of deep learning models typically occurs through platforms such as "GitHub," "Papers With Code," and "Hugging Face." These platforms were handy for sourcing pre-trained models in the context of this dissertation. However, a challenge arises because each author employs its own approach to model implementation. Consequently, adjustments are necessary to align with the architecture outlined in Chapter 4.

This section elucidates the workflow for making these adaptations, highlighting the challenges that should be addressed to test and evaluate each model.

5.1.1 Models from Hugging Face

Hugging Face models can be employed directly via a dedicated "model loader" using the Python "Transformers" API. However, for optimization purposes, an alternative approach involves utilizing the API to load the model in a format compatible with TorchScript and subsequently exporting it as TorchScript or ONNX.

Through this approach, the models obtained were SegFormer, Mask2Former, and UperNet.

5.1.2 Models from GitHub

Some implementations offer valuable tools for seamlessly exporting the model file into various formats. In such cases, the process is straightforward, allowing for direct export to TorchScript and ONNX or, in some cases, to a file optimized with TorchRT. This convenience applied to models like YOLOv5, YOLOv8, YOLOP, TwinLiteNet.

Conversely, some implementations lack these tools or rely on different frameworks, such as MMCV [62] on top of PyTorch. In such scenarios, it becomes necessary to navigate the repository to locate the required Python functions and classes for model loading, make necessary adaptations, and craft a script for model export. This requirement applied to models such as YOLOPv2, YOLOv7, RESA, UFLDv2, and O2SFormer.

5.1.3 Optimization

The optimization of models for inference could not be universally applied to every model. This limitation is attributed to the construction of the models, as certain models incorporate operators unsupported by the ONNX and/or TensorRT compatibility matrixes. Furthermore, some models employ functions incompatible with half-precision (float 16).

Table 5.1 contains the information regarding which optimization options were possible for each model.

Table 5.1: Optimizations made to each model for evaluation - The first three models are multi-tasked, and the remaining are single-tasked

	Jit - Fp32	Jit - Fp16	Trt - Fp32	Trt - Fp16
YOLOPV2	✓	✓	✓	✓
YOLOP	✓	✓	✓	✓
TwinLiteNet	✓	✓	✓	✓
YOLOV8	✓	✓	✓	✓
YOLOV7	✓	✓	✗	✗
YOLOV5	✓	✓	✗	✓
Mask2Former	✓	✗	✗	✗
UperNet	✓	✓	✗	✗
SegFormer	✓	✓	✓	✓
RESA	✓	✓	✗	✗
O2SFormer	✓	✗	✗	✗
UFLDv2	✓	✗	✓	✓

Summary - Model implementations

In order to conduct application testing and model evaluation, it is imperative to acquire weight files and establish a method for loading these files to create functional models for subsequent inference execution. These weight files were downloaded from both “GitHub” and “Hugging Face”.
Whenever feasible, optimizations were applied to the models.

5.2 Architecture functionalities

The application's architecture, as presented in Chapter 4, is designed to accommodate both single-task and multi-task models and concurrently use multiple models. To assess its functionality, some models from Table 5.1 were employed for inference tasks. During the application execution, ROS graphs were generated to visualize the inter-node communication across various configurations. Additionally, screen captures were taken to record the output for documentation.

Due to their size, the detailed ROS graphs have been included in Appendix A, while this section showcases simplified adaptations for better visualization. These adaptations follow a structure similar to figure 5.1.

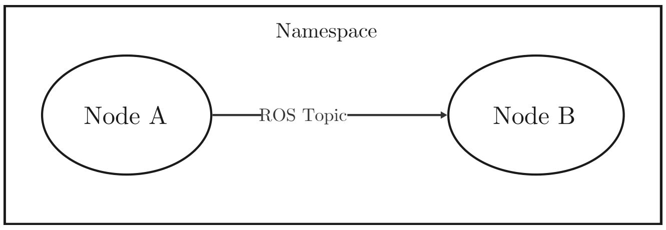


Figure 5.1: ROS Graph adaptation structure for illustration purposes in this dissertation

5.2.1 Single-tasked networks

This subsection demonstrates the range of tasks the architecture can handle effectively and presents a modified ROS graph to depict the communication flow.

The architecture supports tasks such as object detection, as depicted in figure 5.2, and segmentation tasks, showcased in figures 5.3 and 5.4.

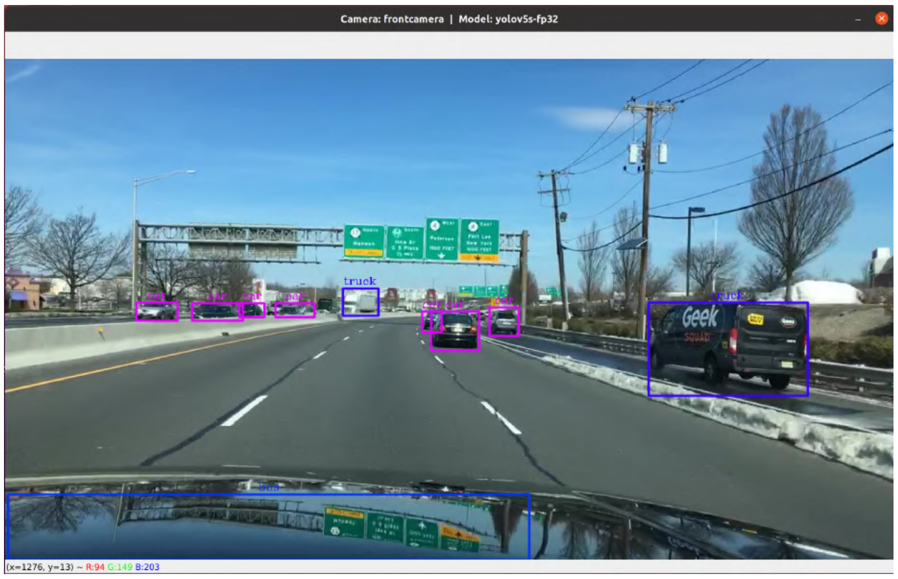

Camera: frontcamera | Model: yolov5s-fp32
truck
(x=1276, y=13) ~ R:94 G:149 B:203

Figure 5.2: Inference output of a single-tasked neural network for object detection. Bounding boxes identify objects

In the adapted ROS graph shown in figure 5.5, two communication channels are observable, both directed toward the image plotter. One is dedicated to object detection, while the other serves image segmentation. This might appear redundant, but it is important to note that the inference manager lacks knowledge of the model's capabilities, thus generating topics for all potential tasks. However, it only transmits messages to the channel corresponding to the active task.

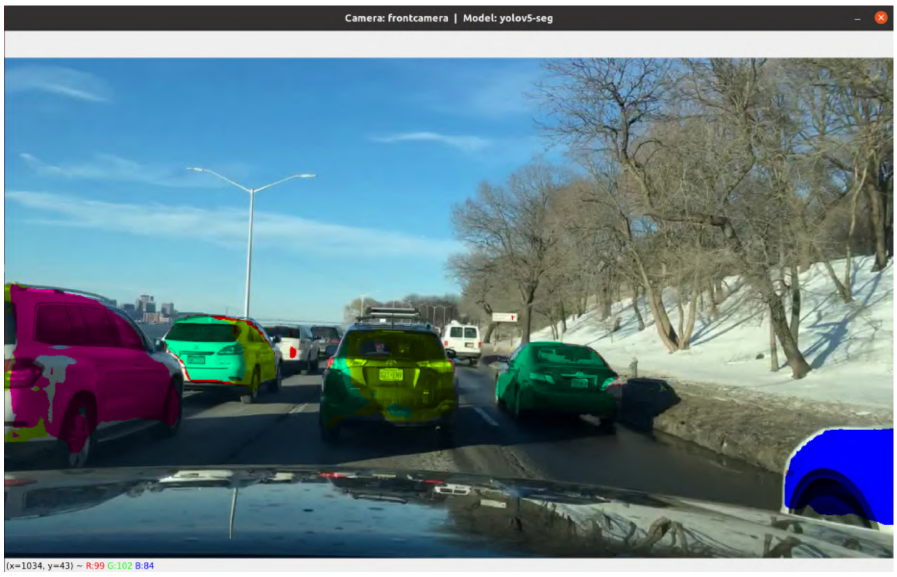

Camera: frontcamera | Model: yolov5-seg
(x=1034, y=43) ~ R:99 G:102 B:84

Figure 5.3: Inference output of a single-tasked neural network for instance segmentation. Overlay colors identify object instances and their pixels

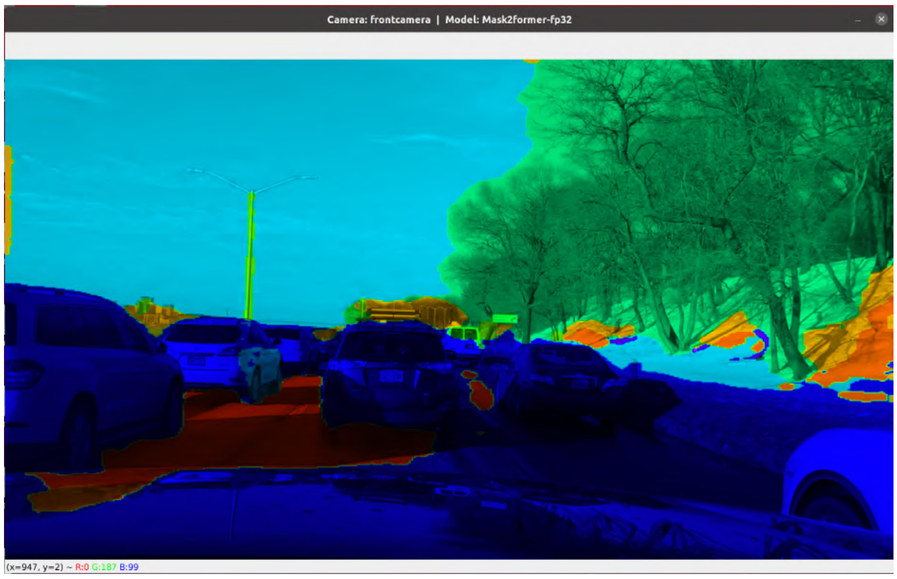

Camera: frontcamera | Model: Mask2former-fp32
(x=947, y=2) ~ R:0 G:187 B:99

Figure 5.4: Inference output of a single-tasked neural network for semantic segmentation. Overlay colors identify classes of pixels

5.2.2 Multi-tasked networks

To illustrate its multi-task capabilities, figure 5.6 displays the application's output, featuring a multi-task neural network capable of performing car detection and segmentation of drivable areas and lane lines.

The ROS graph for a multi-task network closely resembles a single-task network, with the key difference being that in this one, both topics receive messages. Figure 5.7 contains a visual representation of the ROS graph.

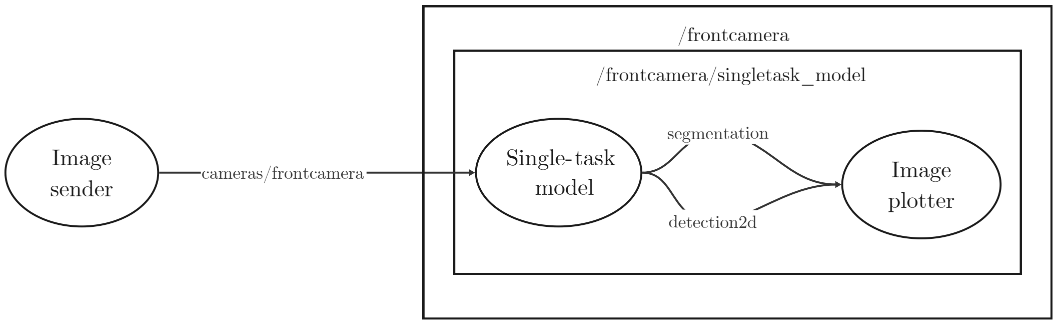


Figure 5.5: ROS graph of a single-tasked neural network with one input and one output (adapted from figure A.1)

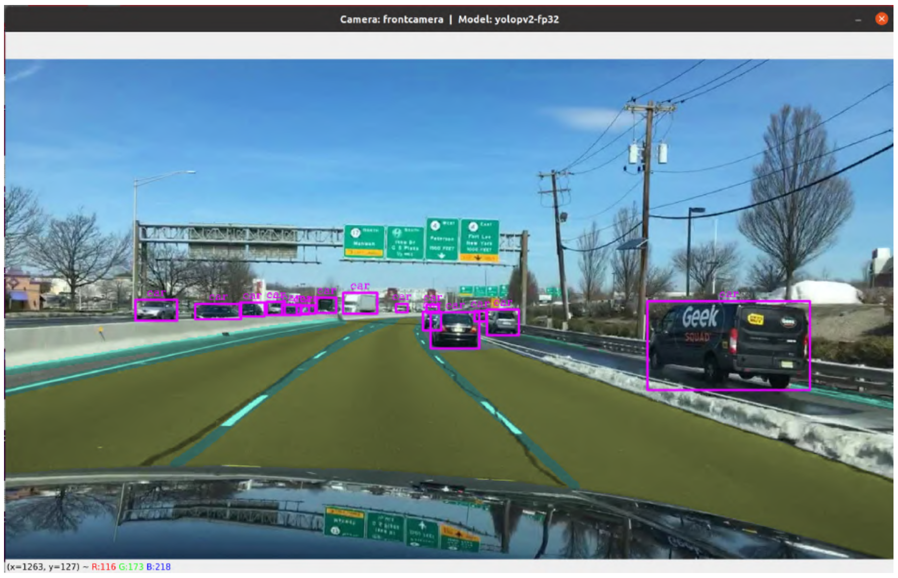

Camera: frontcamera | Model: yolopv2-fp32
car
Geek
(x=1263, y=127) ~ R:116 G:173 B:218

Figure 5.6: Inference output of a multi-tasked neural network for car detection, drivable area segmentation, and lane marking. It is a "do-it-all" approach (multitask)

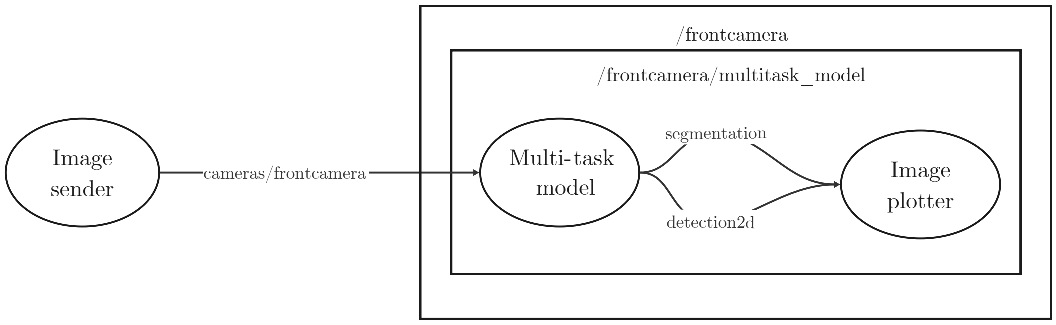


Figure 5.7: ROS graph of a multi-tasked neural network with one input and one output (adapted from figure A.2)

5.2.3 Multiple networks

The developed application exhibits the ability to operate with multiple models concurrently. Figure 5.8 shows an image featuring detected objects and segmented road and lane lines. To achieve this, individual models dedicated to each specific task were employed. This capability is also depicted in a ROS graph, as shown in figure 5.9.

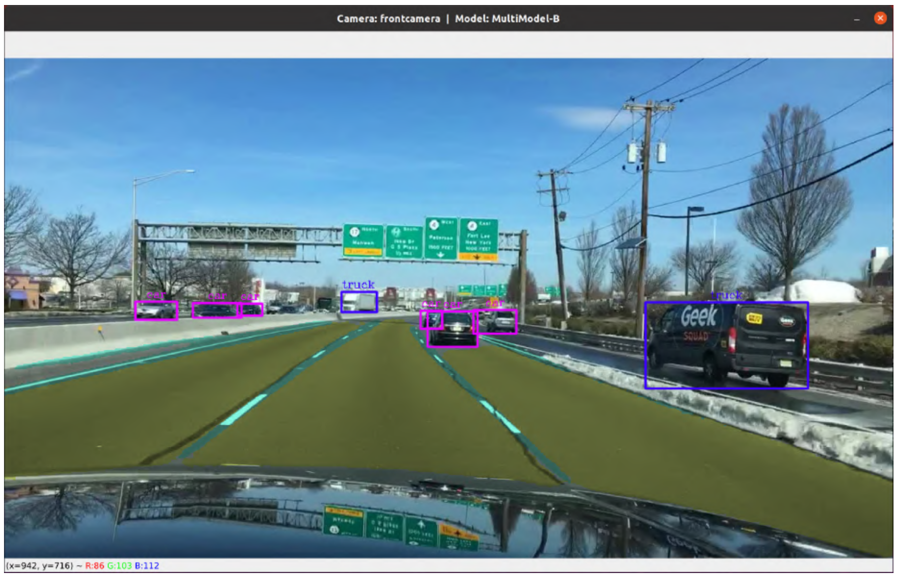

Camera: frontcamera | Model: MultiModel-B
truck
truck
(x=942, y=716) ~ R:86 G:103 B:112

Figure 5.8: Inference output of multiple single-tasked neural networks: one for object detection, a second one for drivable area segmentation, and a third one for lane marking

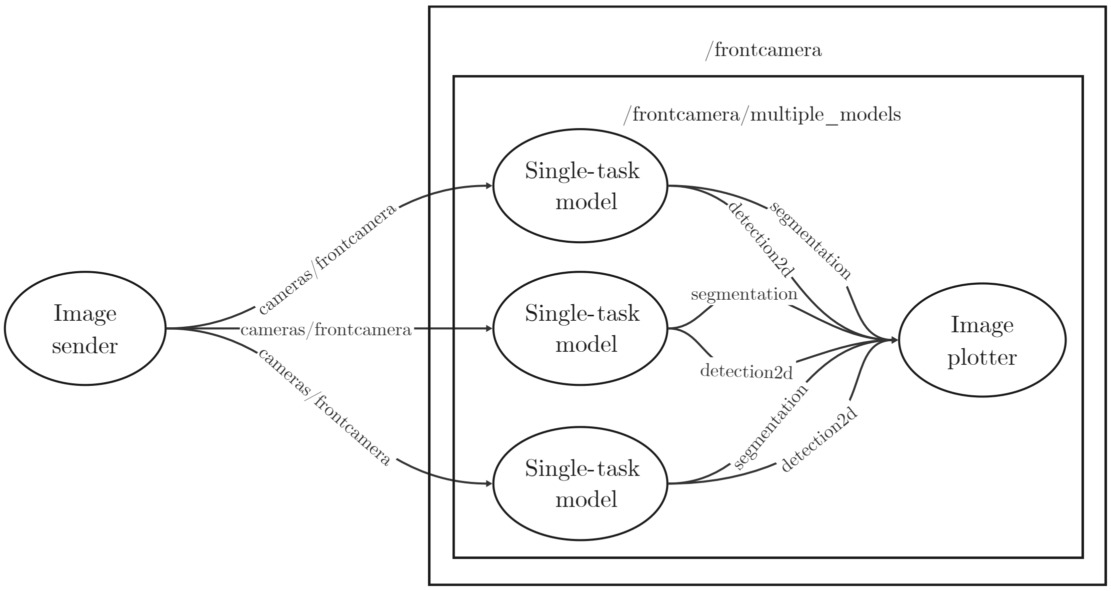


Figure 5.9: ROS graph of multiple neural networks with one input and one output (adapted from figure A.3)

Beyond the utilization of multiple models, it is also feasible to employ multiple parallel outputs, allowing for simultaneous inferences on the same image using different models, as demonstrated in figure 5.10. Moreover, as illustrated in figure 5.11, the system supports using multiple inputs, each associated with at least one corresponding output.

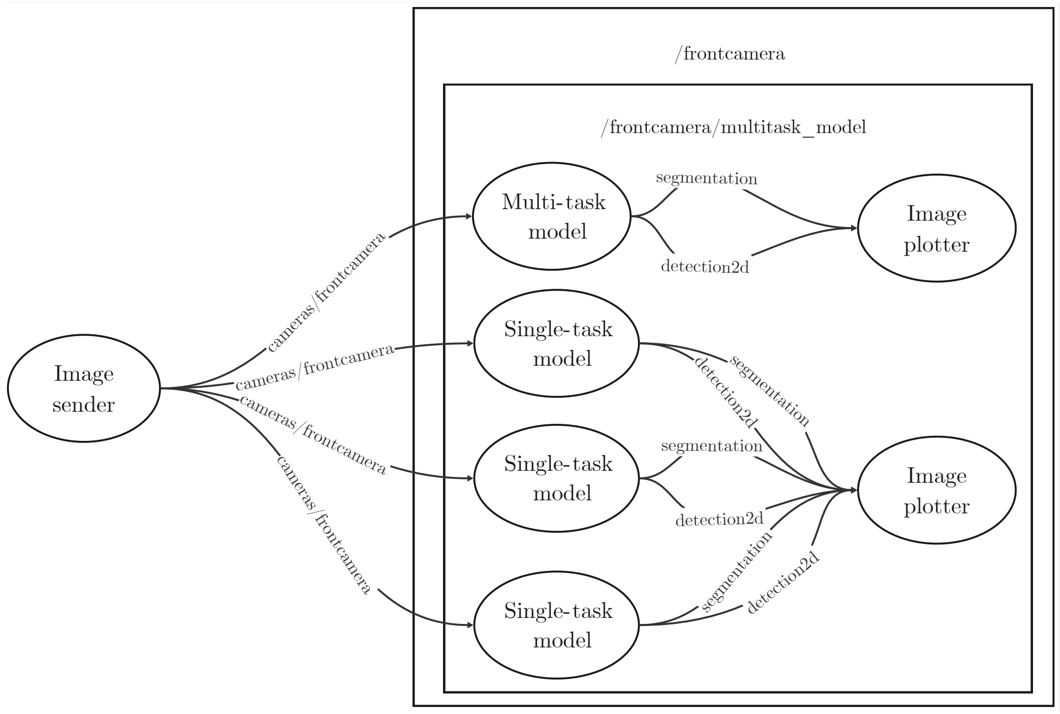


Figure 5.10: ROS graph of multiple neural networks with two inputs and one output (adapted from figure A.5)

Summary - Architecture functionalities

The application explored in Chapter 4 is capable of handling both single-tasked and multi-tasked networks, as well as multiple models. This section substantiates these capabilities by showcasing the application's output and employing ROS graphs to depict the nodes and the communication channels between them.

5.3 Performance comparisons

In order to assess the performance of the models in this study, the tests were divided into two main phases. The first phase involved identifying, for each task, the best-performing single-task and multi-task models. The second phase focused on comparing combinations of the best single-task models with the best multi-task models.

Throughout the subsequent sections, two types of graphs are employed. One type evaluates the AP with IoU thresholds of either 50 (AP@50) or 75 (AP@75) and the inference speed, primarily for car detection. The other type assesses lane marking and drivable area segmentation, employing IoU as the evaluation metric instead of AP.

All model names in the graphs adhere to a consistent format: "ModelName-Precision" for Jit optimization and "ModelName-Precision-trt" for TensorRT optimization. Each

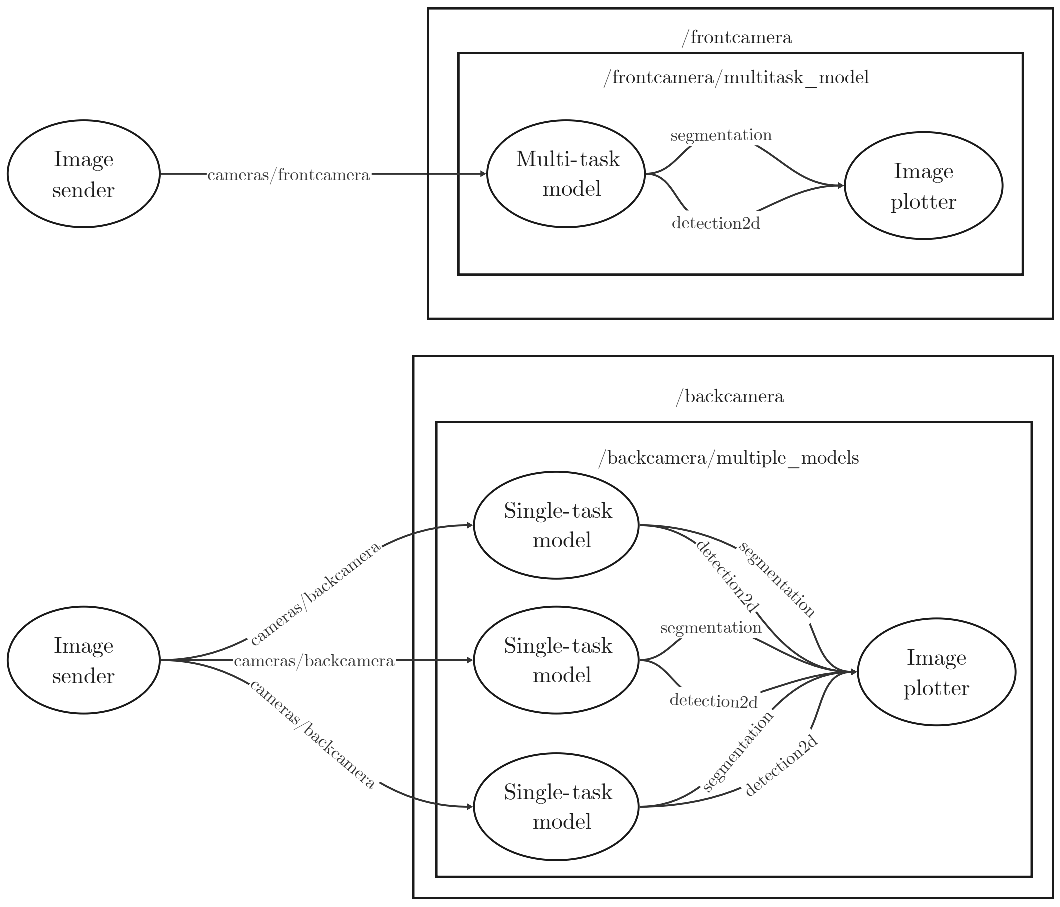


Figure 5.11: ROS graph of multiple neural networks with two inputs and two outputs (adapted from figure A.4)

model is represented by a circular marker with a central dot, with the circle's size directly proportional to the model's file size.

The evaluations were conducted using the BDD100k dataset on the validation set, as the labeled test set is not publicly available. It is worth noting that in the context of car detection, some models categorize vehicles such as trucks and buses as "cars". To ensure a fair comparison, when the evaluation script encounters classes like "truck" or "bus", they are treated as "car". Furthermore, BDD100k distinguishes between regions of the road where the vehicle can or cannot navigate. However, certain models cannot make this distinction, and, to address that, these regions were combined to equate the drivable area with the entire road.

The results in this section were obtained using the laptop Gigabyte Aero 15. The result values can be found in the appendix B. Table 5.2 presents a summary of the goals of the tests covered in this section.

Table 5.2: Overview of the executed steps in the following subsections

Tests	Goals
Single-tasked (JIT and TRT)	To assess what is the best single-tasked model for car detection[*]
	To assess what is the best single-tasked model for drivable area
	To assess what is the best single-tasked model for lane marking
Multi-tasked (JIT and TRT)	To assess what is the best multi-tasked model for car detection[*] + drivable area + lane marking (YOLOPs)
	To assess what is the best multi-tasked model for drivable area + lane marking (TwinLiteNets)
Multiple models and multi-tasked	To combine the best single tasked models to form a MultiModels
	To compare the best multi-tasked models with MultiModels

[*] Tested for AP@50 and AP@75.

5.3.1 Single-tasked models

Figure 5.12 displays the AP with an IoU threshold greater than 50 for object detection. Similarly, figure 5.13 illustrates AP with an IoU threshold greater than 75 for the same task. Notably, the model "YoloV8s-Fp16" emerged as the top-performing choice, excelling in precision while maintaining a high inference speed.

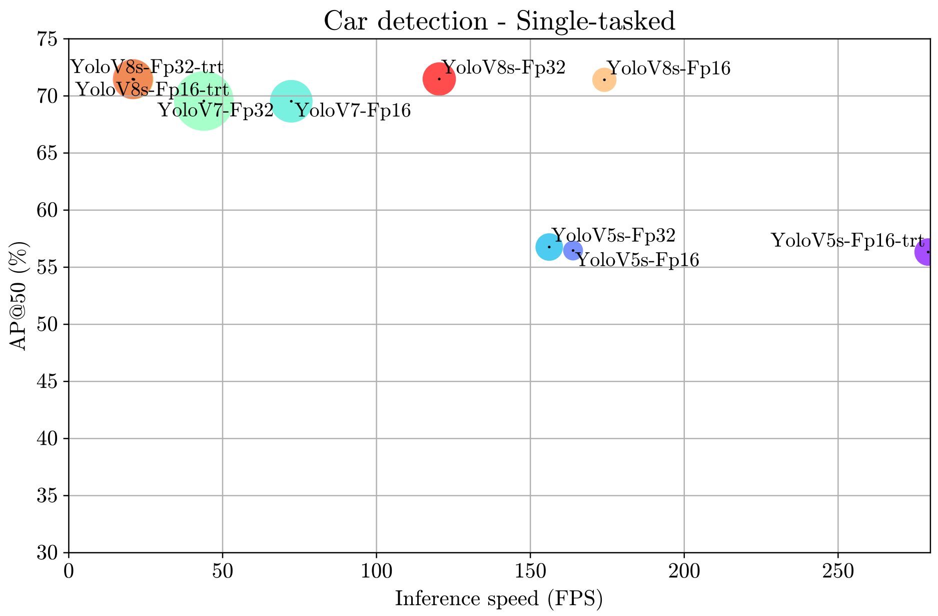


Figure 5.12: Evaluation of car detection with single-tasked neural networks at AP@50

Figure 5.14 depicts the evaluation of IoU in the drivable area segmentation task. Regardless of their performance, several models with unacceptably slow inference speeds are excluded from consideration. Among the models, "Segformer-Fp32" and "Segformer-Fp16" achieve comparable IoU scores, with "Segformer-Fp16" being selected due to its superior inference speed.

Figure 5.15 presents the evaluation of IoU in the lane segmentation task. Unfortunately, all models exhibited poor performance, but the selected model, in this case, is "UFLDv2-Fp16-trt" due to its superior speed.

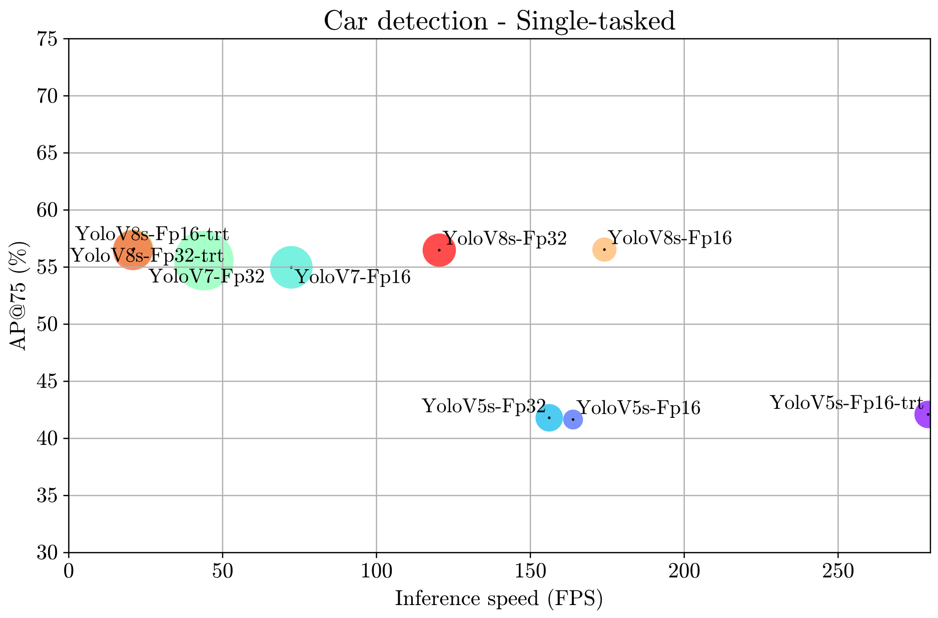


Figure 5.13: Evaluation of car detection with single-tasked neural networks at AP@75

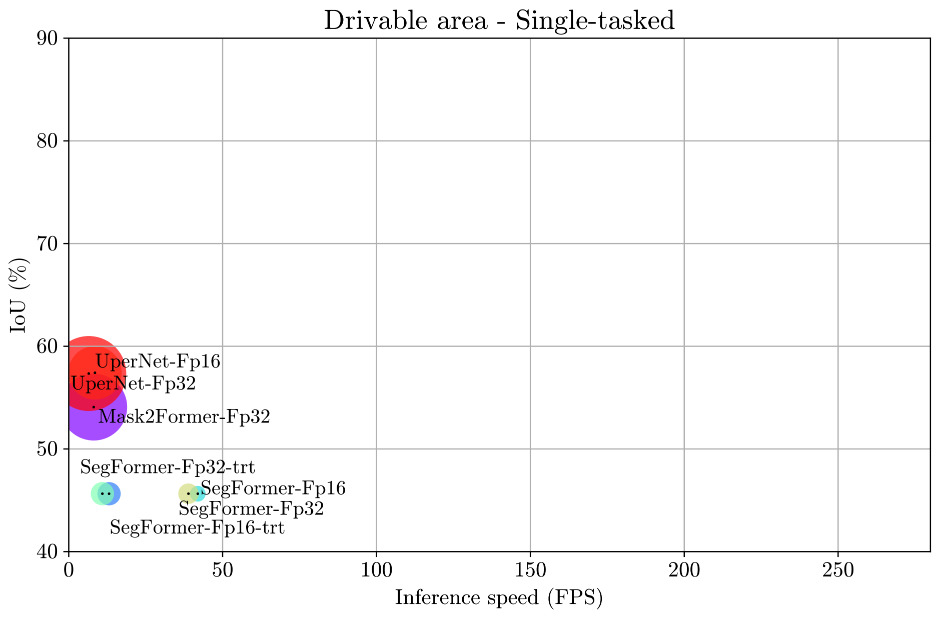


Figure 5.14: Evaluation of drivable area segmentation with single-tasked neural networks

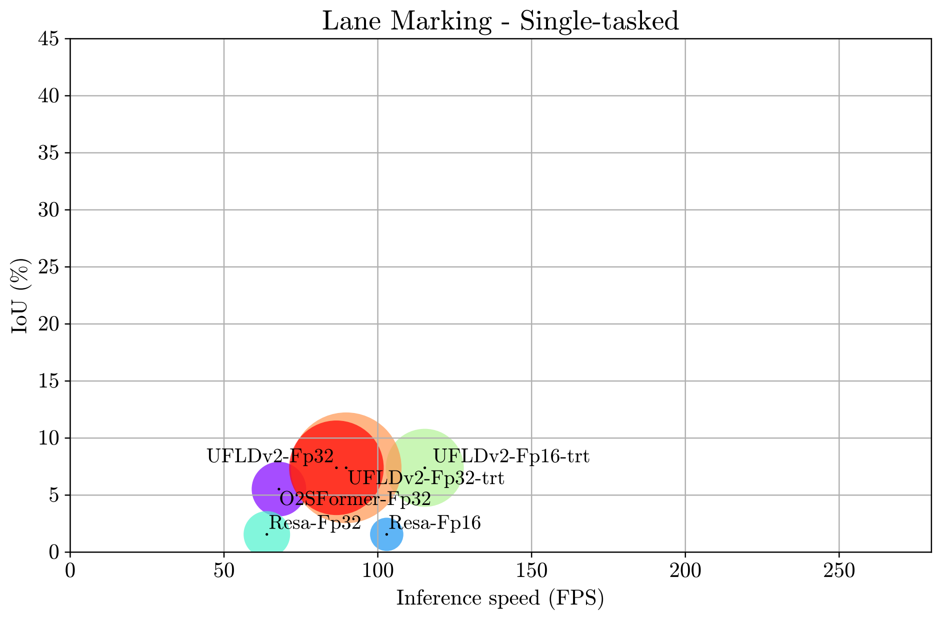


Figure 5.15: Evaluation of lane marking with single-tasked neural networks

5.3.2 Multi-tasked models

Within the scope of this dissertation, two distinct types of multi-task models are examined. YOLOP and YOLOPv2 undertake three tasks, while TwinLiteNet focuses on two. Due to their differing task sets, this subsection conducts separate evaluations for each.

For object detection, an examination of the results displayed in figures 5.16 and 5.17 reveals a cluster of models delivering superior precision. In this group, the choice falls upon the fastest option, “YolopV2-Fp16-trt”.

Comparatively, the models exhibit analogous behavior to that observed in object detection, both in the context of drivable area segmentation (figure 5.18) and the lane marking task (figure 5.19). Consequently, based on the analysis of these graphs, the model of choice remains “YolopV2-Fp16-trt”.

TwinLiteNet receives a separate evaluation due to its specialization in two tasks. While this comparison exclusively considers TwinLiteNet, it encompasses various precisions and optimizations. In this context, the disparity in IoU between models is minimal, evident in both drivable area segmentation (figure 5.20) and lane marking (figure 5.21). Consequently, inference speed is the decisive factor for determining the superior model. Taking this into consideration, the optimal choice is “TwinLiteNet-Fp16-trt”.

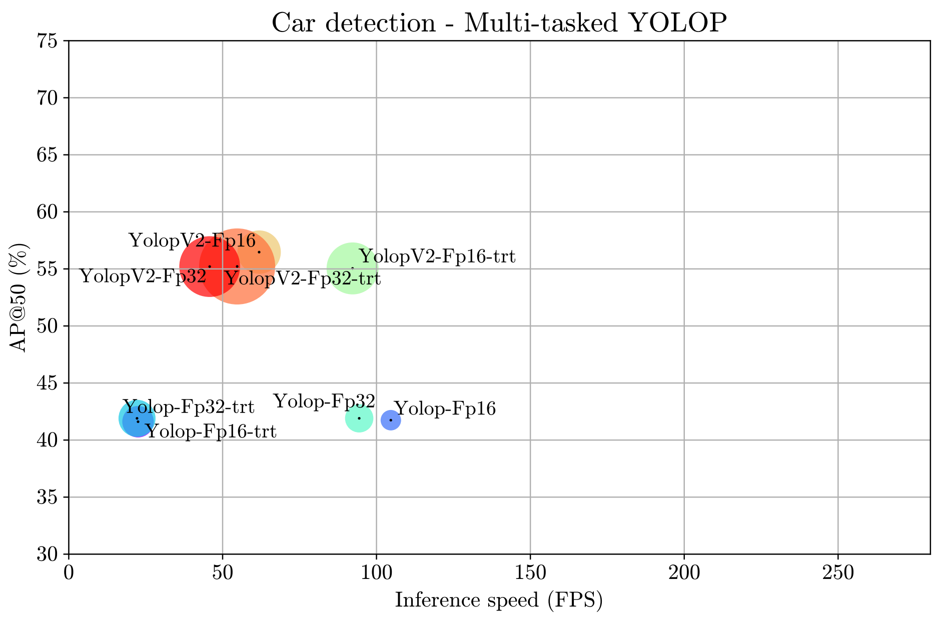


Figure 5.16: Evaluation of car detection with YOLOP at AP@50

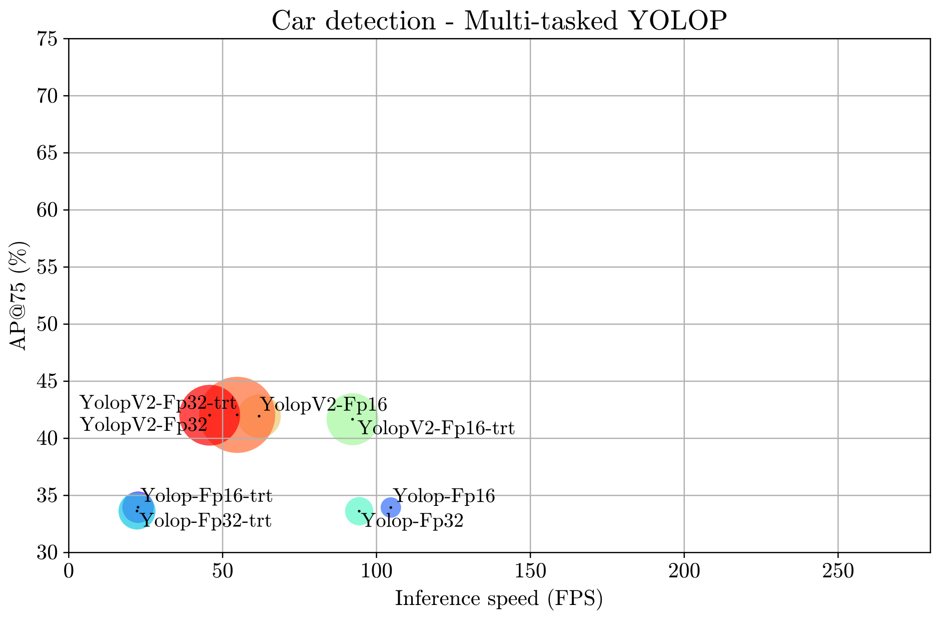


Figure 5.17: Evaluation of car detection with YOLOP at AP@75

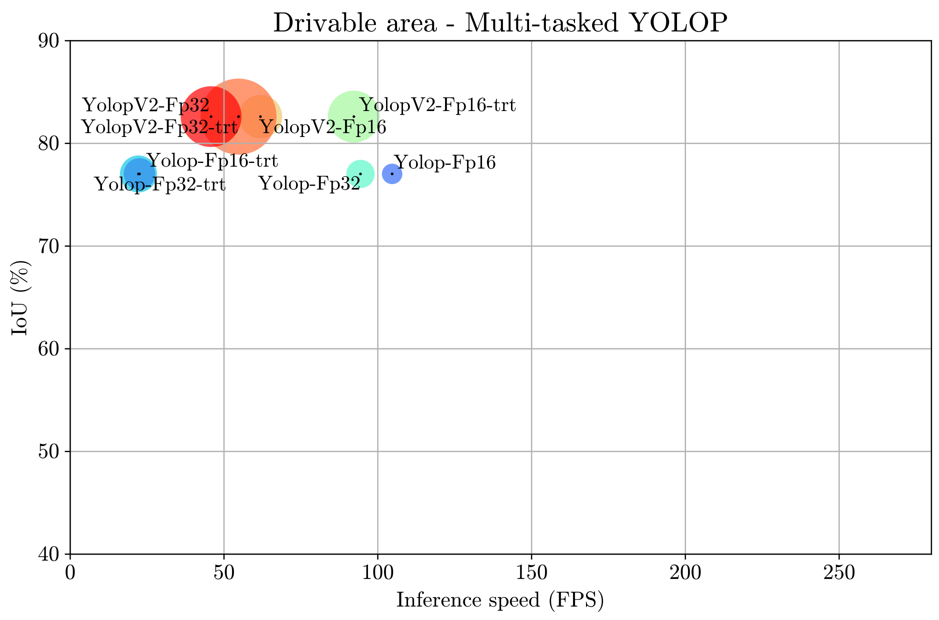


Figure 5.18: Evaluation of drivable area segmentation with YOLOP

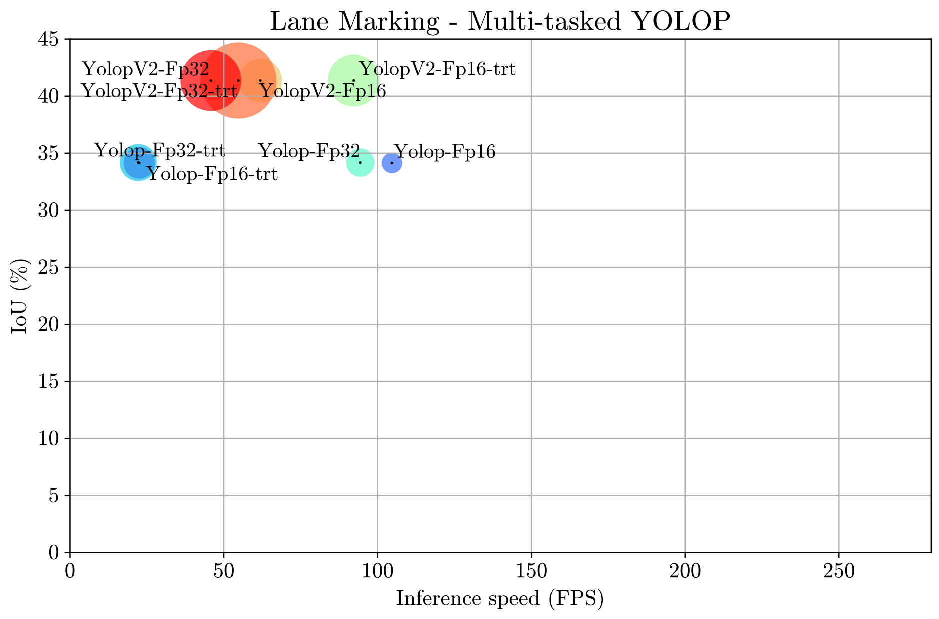


Figure 5.19: Evaluation of lane marking with YOLOP

5.3.3 Multi-tasked and multiple models

This subsection compares the best single-task neural networks and the best multi-task models. While two multi-task networks were selected in the previous subsection, a detailed evaluation was only conducted for one. In the case of segmentation tasks, comparing

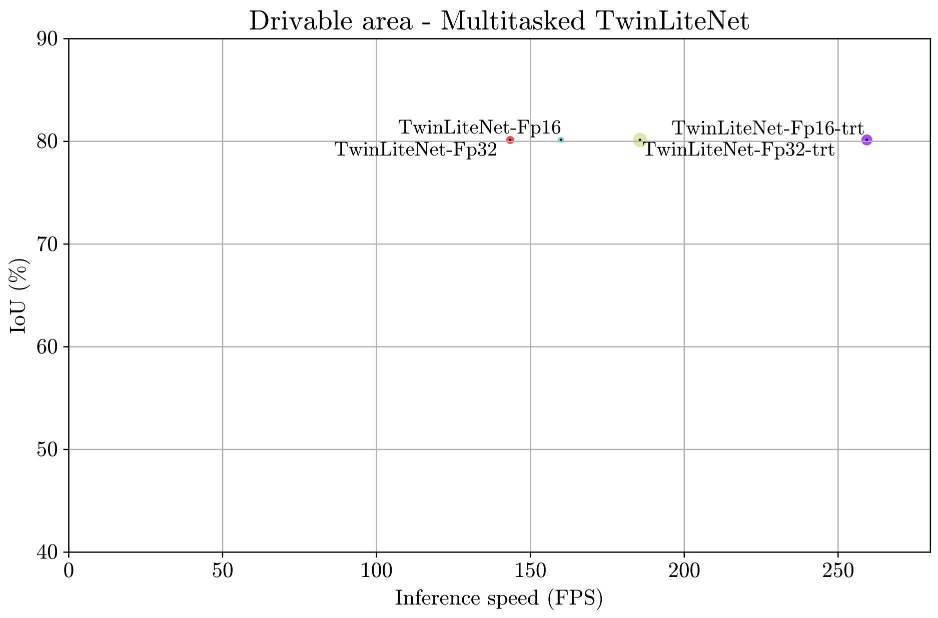


Figure 5.20: Evaluation of drivable area segmentation with TwinLiteNets

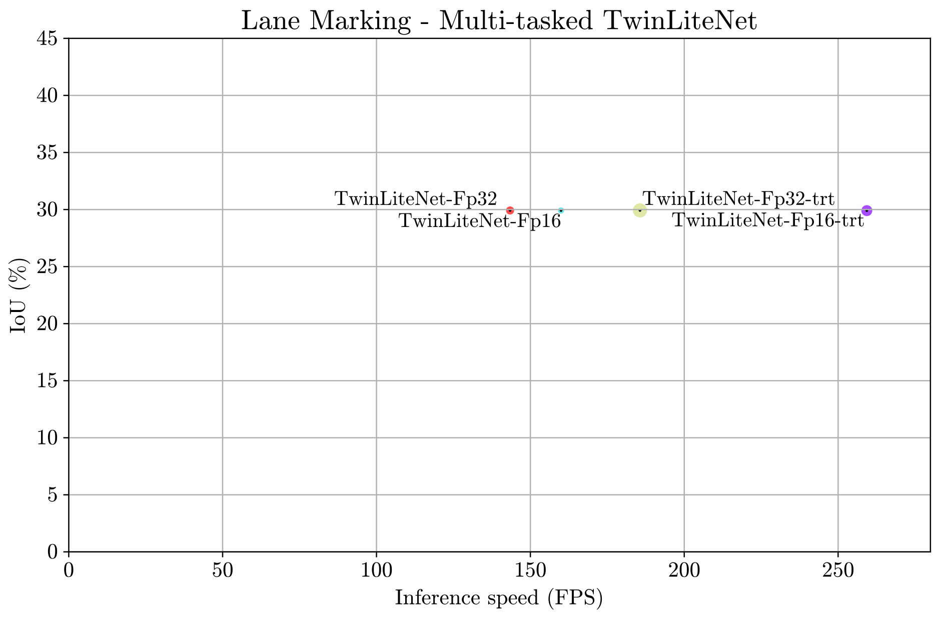


Figure 5.21: Evaluation of lane marking with TwinLiteNets

TwinLiteNet to single-task models is redundant as the single-task models yield inferior results in both speed and IoU.

Given the unsatisfactory performance of single-task models in both segmentation tasks, this subsection introduces an additional study involving two instances of TwinLiteNet, each employed as a single-task model. For that, two special “modules” (section 4.5.1) were made. One only outputs drivable area segmentation by ignoring the lane marking, while the other only outputs lane marking by ignoring the drivable area segmentation. This approach compensates for the absence of competitive single-task models, facilitating a more equitable comparison between multi and single-task models.

The combinations of the best single-tasked neural networks (“YOLOv8-Fp16”, “Segformer-Fp16”, and “UFLDv2-Fp16-trt”), are denoted as “MultiModel-A”, and the combinations of two instances of TwinLiteNet with the best car detection model are labeled as “MultiModel-B”, simplifying the interpretation of the comparison graphs.

TensorRT optimization and tasks’ synchronization

During each test, the timestamps for the start and end of each task are obtained using the ROS time functions. This data provides information on the speed of execution of the tasks within MultiModel and the level of synchronization between the outputs of each model within MultiModel.

Some of the best single-tasked models were optimized with TensorRT. This optimization, as explained in section 3.2.4, takes the hardware into account. Running multiple models simultaneously might result in unexpected behavior, as the hardware is used differently than during the optimization. To ensure that the MultiModels are in the best condition possible, it is worth comparing the use of optimized models with TensorRT and PyTorch JIT since PyTorch JIT’s optimization does not take the hardware into account.

Figures 5.22 and 5.23 display the inference speed of “MultiModel-A” and “MultiModel-B”, respectively, both with and without TensorRT optimization (indicated by the “-trt” suffix). Additionally, the figures show the speed of tasks compatible with TensorRT, both with and without TensorRT optimization, during the MultiModel execution.

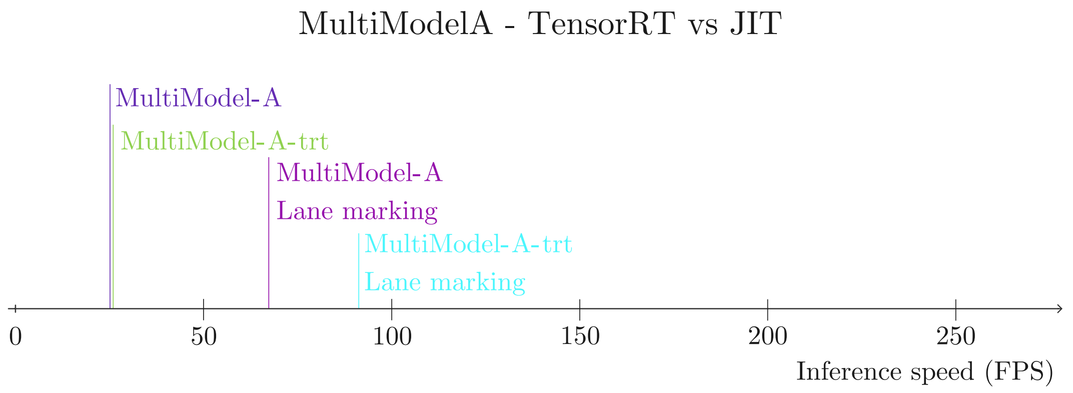


Figure 5.22: Comparison between the optimized tasks with TensorRT and Pytorch JIT in the MultiModelA and overall performance of the MultiModel

Both figures 5.22 and 5.23 show that, although the individual tasks still benefit from the TensorRT optimization, the overall speed of the MultiModel is slightly improved.

On the utilized hardware, it was not possible to execute the tasks in a fully simultaneous manner, which harmed the synchronization of outputs. This issue becomes more pronounced when working with models optimized using TensorRT. This can be observed

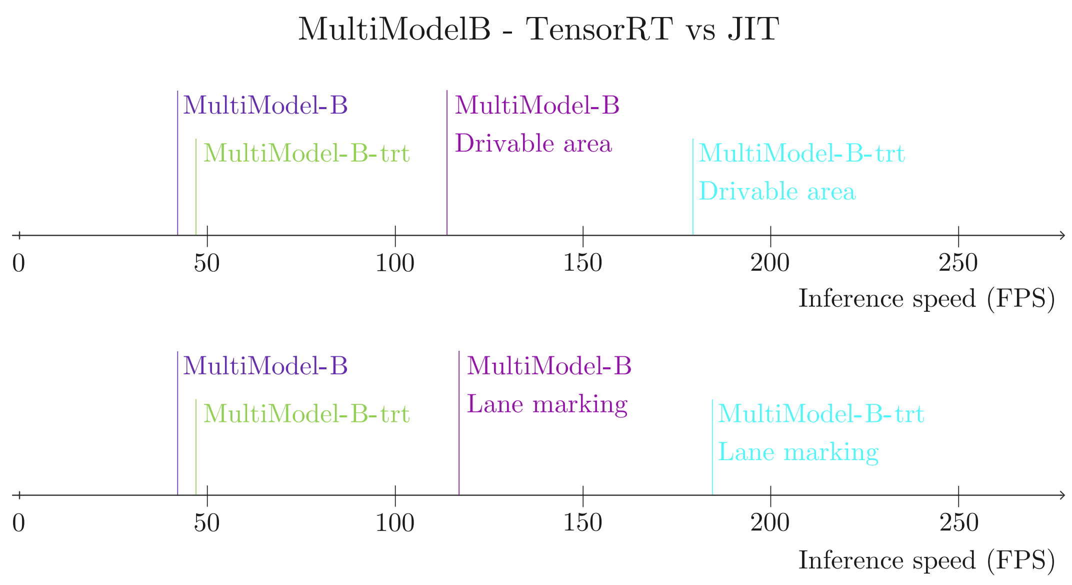


Figure 5.23: Comparison between the optimized tasks with TensorRT and Pytorch JIT in the MultiModelB and overall performance of the MultiModel

by reordering the tasks in temporal order for each processed image and computing the time difference between the completion of subsequent tasks.

Despite the worst synchronization, the overall inference time for both MultiModels is shorter when using models optimized with TensorRT, rendering "MultiModel-A-trt" and "MultiModel-B-trt" as the preferred options for comparison with the multi-tasked neural network.

Comparison between MultiModels and Multi-tasked models

Given that not all models within the MultiModels are optimized with TensorRT, to ensure a fair evaluation, the multi-tasked model is compared using both TensorRT and PyTorch JIT optimizations.

Across all tasks analyzed, "MultiModel-A-trt" and "MultiModel-B-trt" exhibit slower speed when considering all tasks in the analysis, compared to the dedicated multi-task model. The car detection task excels in precision for the MultiModels (figures 5.24 and 5.25), while segmentation tasks favor YOLOPv2, with MultiModel-B delivering competitive IoU scores (figures 5.26 and 5.27).

The observation that object detection, particularly car detection, yields superior results when executed as a single task, compared to a multi-task setting, aligns with concerns related to potential issues arising from destructive interference (section 2.2) during the training of multi-task models. Object detection, in particular car detection, is somewhat related to road and lane line segmentation, but not enough to not harm its performance.

On the other hand, road and lane line segmentation tasks exhibit a much stronger interconnection. The inclusion of multiple tasks forces the encoder to preserve information about that task, and information about the road is very important to detect the lanes.

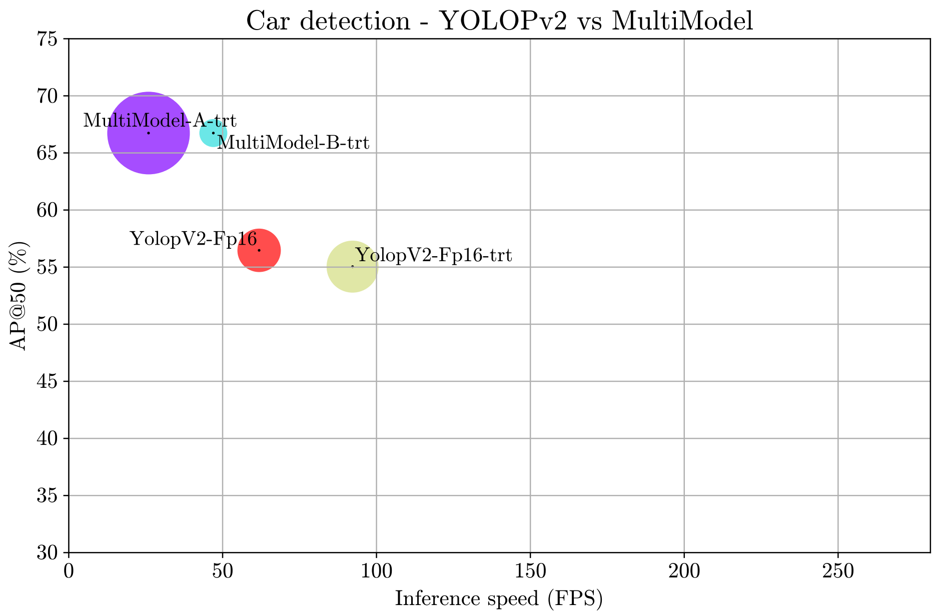


Figure 5.24: Comparison between a multi-task model with multiple single-task models in car detection

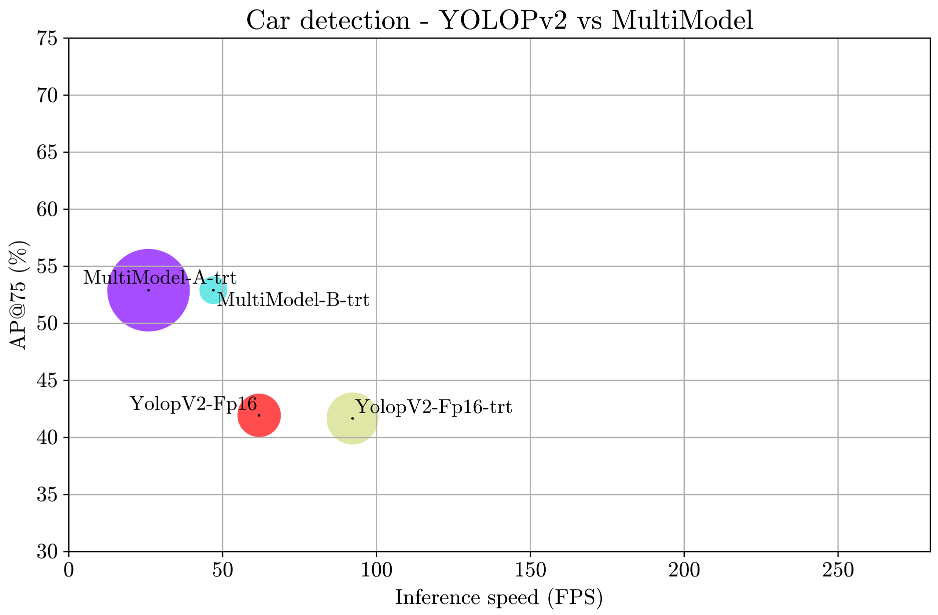


Figure 5.25: Comparison between a multi-task model with multiple single-task models in car detection

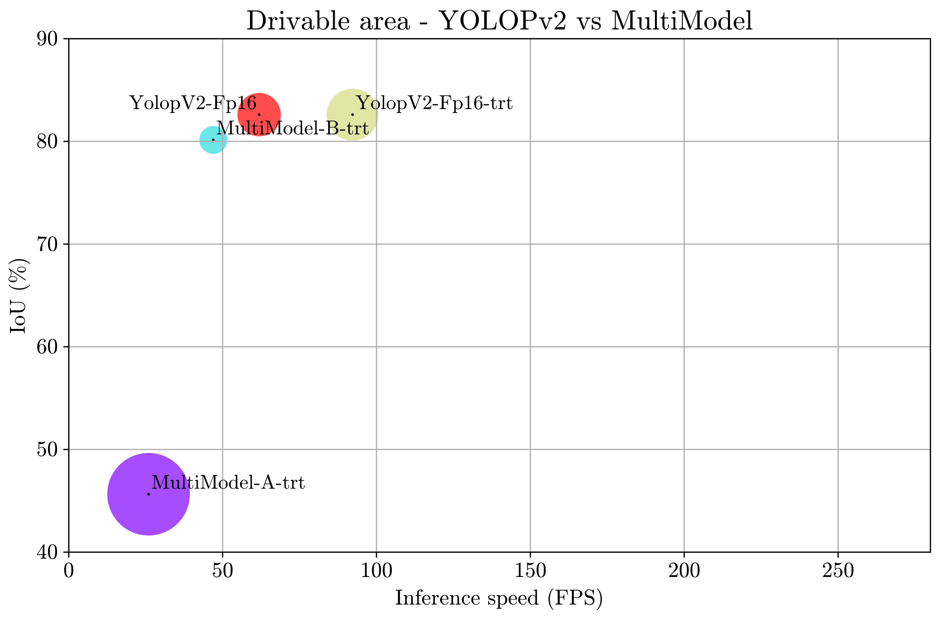


Figure 5.26: Comparison between a multi-task model with multiple single-task models in drivable area segmentation

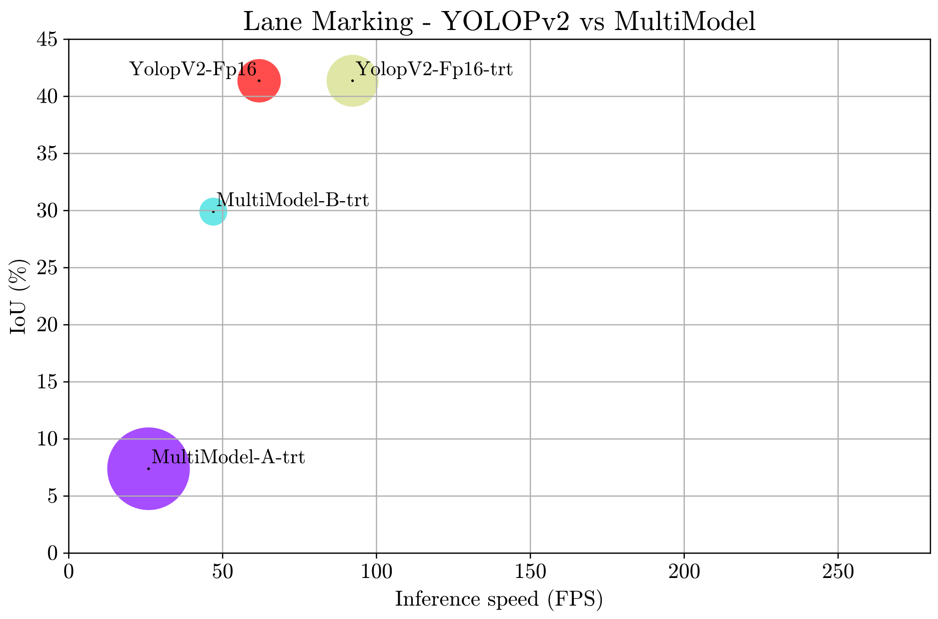


Figure 5.27: Comparison between a multi-task model with multiple single-task models in lane marking

Despite the trade-off in precision for car detection, YOLOPv2 strikes a great balance between precision and inference speed, positioning it as a commendable neural network.

TwinLiteNet shines as a well-crafted implementation of multi-task neural networks, efficiently encapsulating highly related tasks within a single network.

In the context of synchronicity, multi-tasked networks demonstrate exceptional performance due to the presence of a singular input with concurrent output generation. In contrast, MultiModels exhibit different framerates across tasks, and, complicating matters further, the tasks do not start simultaneously. The implementation of MultiModels thus presents an additional challenge that should be addressed, a challenge that is automatically resolved in the case of multi-tasked networks.

In this study, the MultiModels were assessed in the optimal scenario, wherein once a model completes its image evaluation, it pauses until the others also finish. In the application version where evaluations are not executed, models operate whenever possible, further reducing the frame rate. To address this, an additional controller would be necessary to oversee the active nodes and permit a node to restart the execution only after all nodes have completed their tasks. An illustration of this possible solution is illustrated in figure 5.28.

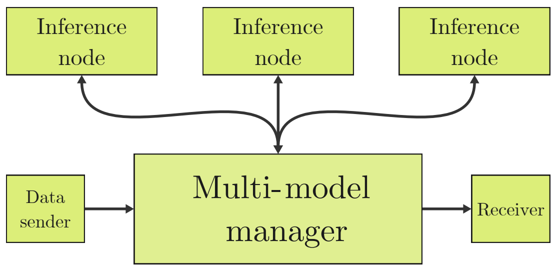


Figure 5.28: Possible solution to manage the inference nodes in order to avoid decreasing the overall framerate

Summary - Performance comparisons

The best single-task models encompass “YoloV8s-Fp16” for car detection, “Segformer-Fp16” for drivable area segmentation, and “UFLDv2-Fp16-trt” for lane marking. Among the multi-task models, for three tasks, “YolopV2-Fp16-trt” excels in handling all tasks analyzed, and for two tasks, “TwinLiteNet-Fp16-trt” demonstrates the best performance in the segmentation tasks.
To evaluate multiple models simultaneously, the best single-task models were combined. However, due to suboptimal results observed with “Segformer-Fp16” and “UFLDv2-Fp16-trt”, a secondary test was conducted, employing two instances of “TwinLiteNet-Fp16-trt” to compensate for these outcomes.
The results of this section conclude that the inference speed of combined models is much slower than the speed of multi-tasked networks. When multiple models are concurrently executed, the framerate of each task is different, leading to worse synchronization. The addition of object detection done by the authors of YOLOP leads to some destructive interference, decreasing the performance when compared with other standard YOLO, but a good balance between accuracy and speed is still achieved.

5.4 Tests and results on the Jetson device

The application functionalities tested on the laptop are compatible with the Jetson device. Nonetheless, on the Jetson device, an additional test was conducted to simulate its onboard usage within ATLASCAR2. This supplementary test involved utilizing the Jetson solely for executing the inference node, while the laptop was responsible for sending the images and presenting the results. A visual representation of the communication setup is depicted in figure 5.29.

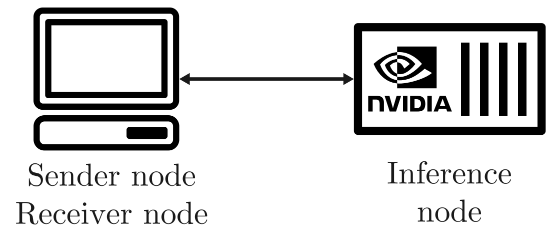


Figure 5.29: Diagram illustrating the laptop and the Jetson in a ROS environment

The evaluation of the performance of the models was done in a similar configuration, as shown in the scheme in figure 5.30.

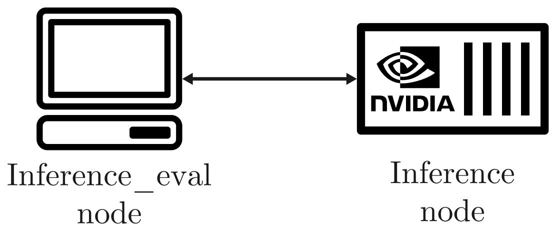


Figure 5.30: Diagram illustrating the laptop and the Jetson in a ROS environment to evaluate the performance of deep learning models

Nevertheless, it is important to note that this evaluation is less extensive compared to the one in section 5.3. In this section, only the models that demonstrated superior performance in the previous section 5.3 are evaluated, primarily serving as a benchmark of their performance on the Jetson platform (the detailed results are presented in appendix C). The only exception to this approach is the comparison involving the utilization of TensorRT in the MultiModels due to its somewhat unpredictable behavior. The best models' performance for both multi-tasked and single-tasked is presented in the graphs of figures 5.31, 5.32, 5.33 and 5.34 for object detection at AP@50 and AP@75, drivable area segmentation and lane marking, respectively.

Figure 5.39 shows a comparison between the MultiModels optimized with TensorRT and JIT. Similarly to the comparison in section 5.3, the TensorRT optimization results in a slightly better inference speed.

The comparisons depicted in figures 5.35, 5.36, 5.37, and 5.38 closely resemble the comparisons made between MultiModels and multi-tasked models in section 5.3. The primary distinction lies in the fact that the JIT-optimized version of YOLOPv2 exhibits slower performance than “MultiModel-B-trt”. Still, the TensorRT optimization remains faster, and “MultiModel-B” utilizes two instances of TwinLiteNet, which is a multi-tasked model. This leads to similar conclusions as the previous section.

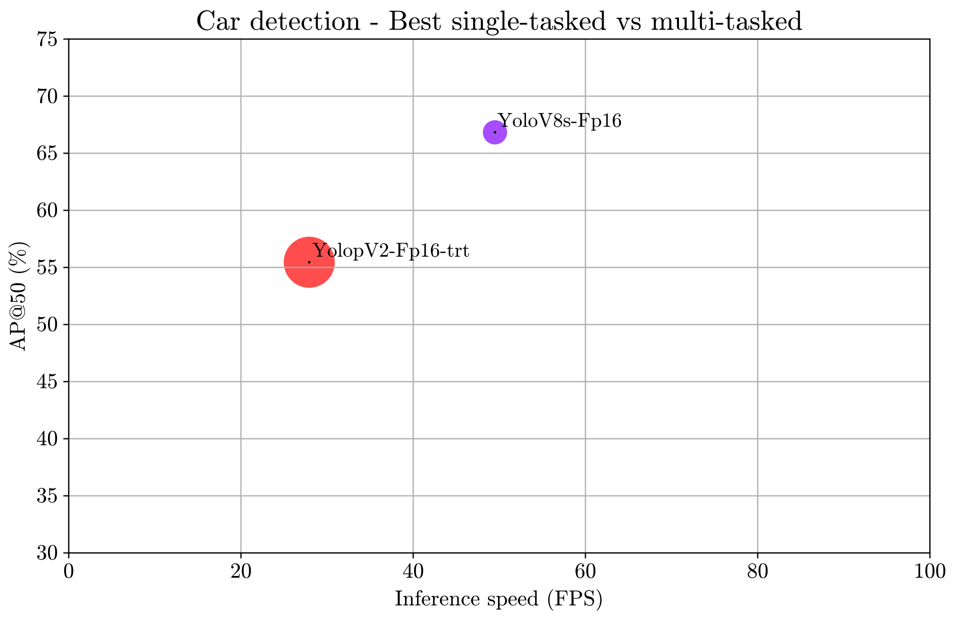


Figure 5.31: Best multi-task and single-tasked models’ performance for object detection at AP@50

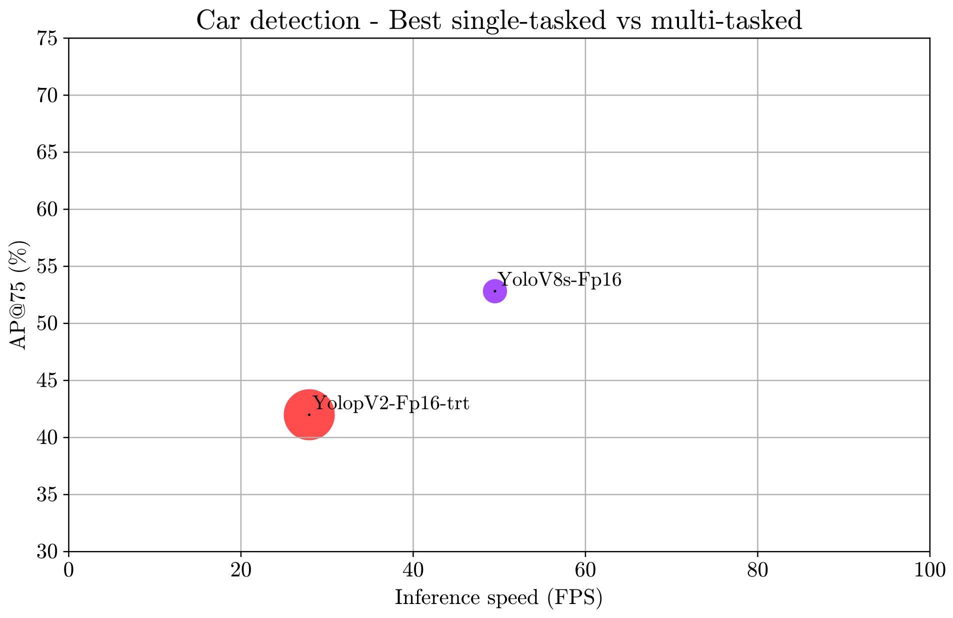


Figure 5.32: Best multi-task and single-tasked models’ performance for object detection at AP@75

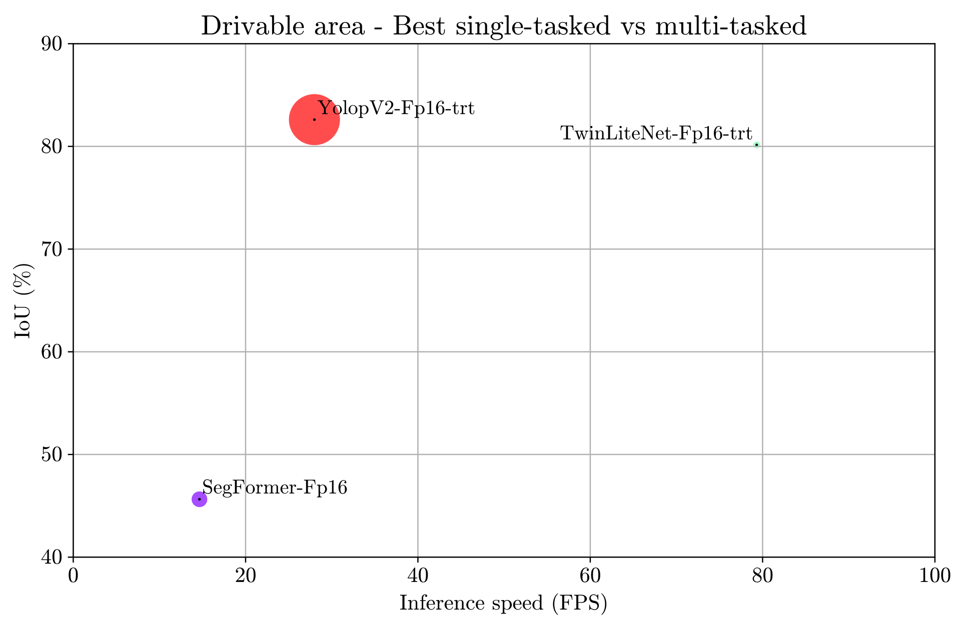


Figure 5.33: Best multi-task and single-tasked models' performance for drivable area segmentation

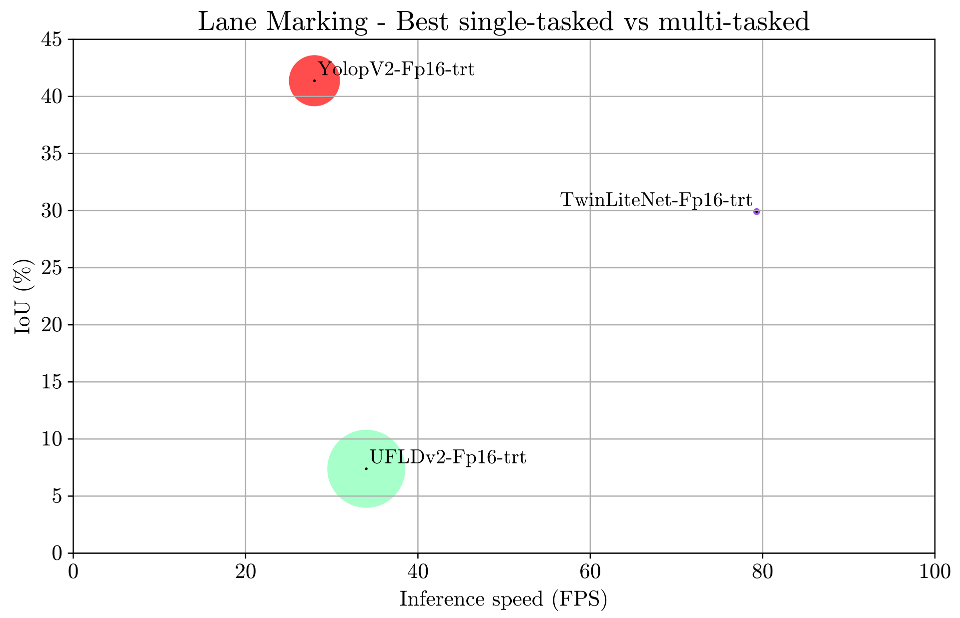


Figure 5.34: Best multi-task and single-tasked models' performance for lane marking

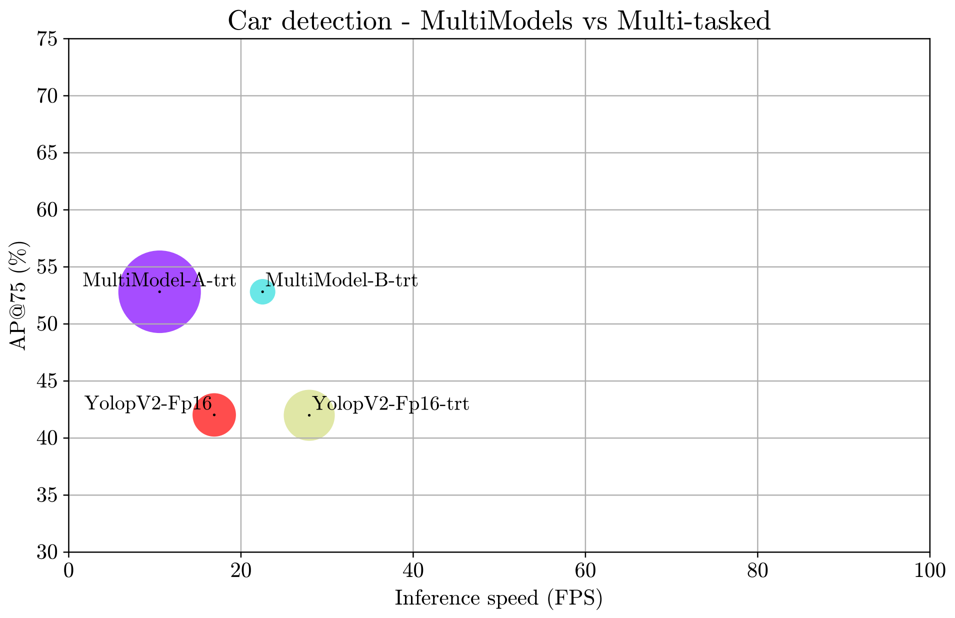


Figure 5.35: Comparison between a multi-task model with multiple single-task models in car detection at AP@50

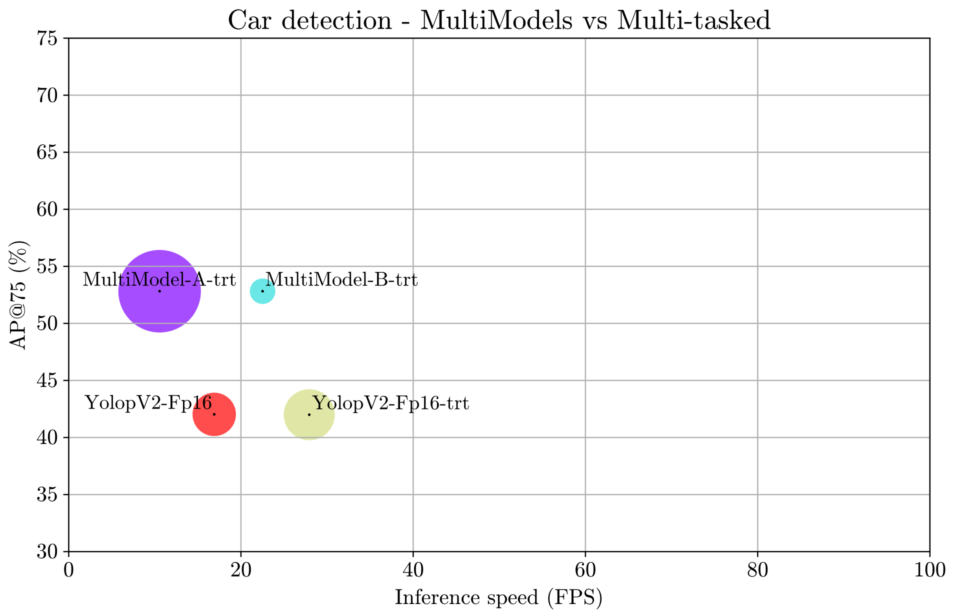


Figure 5.36: Comparison between a multi-task model with multiple single-task models in car detection at AP@75

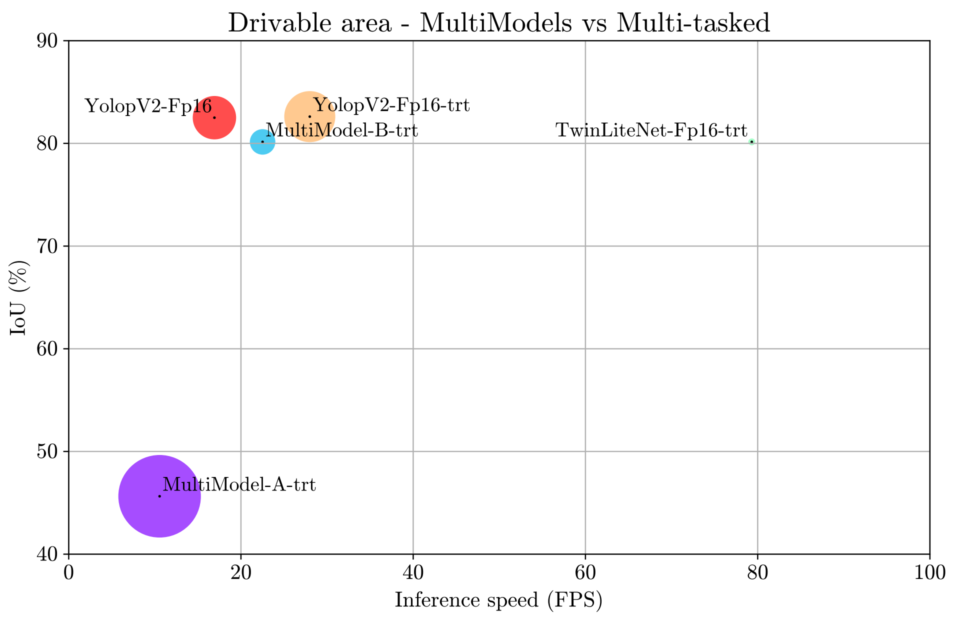


Figure 5.37: Comparison between a multi-task model with multiple single-task models in drivable area segmentation

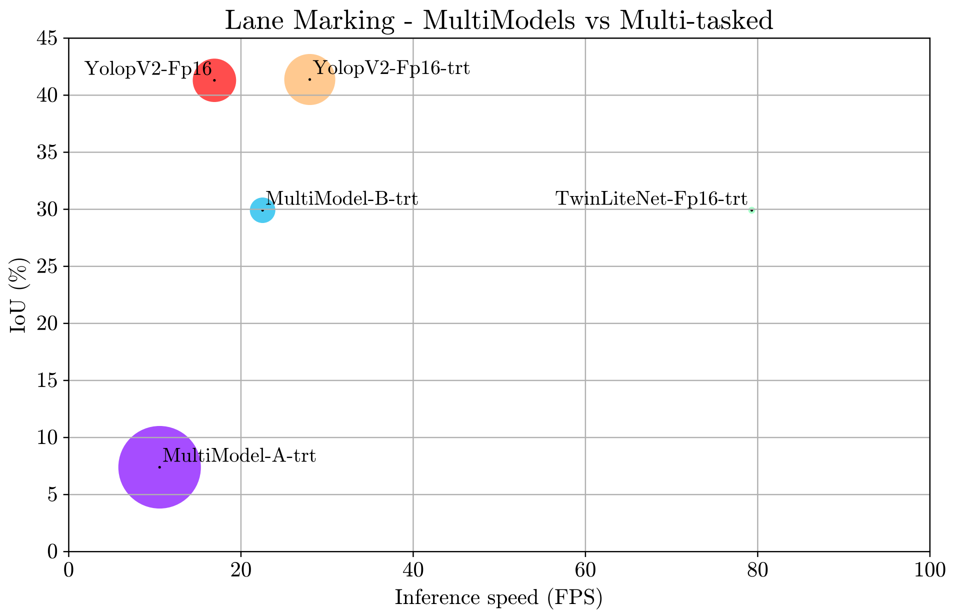


Figure 5.38: Comparison between a multi-task model with multiple single-task models in lane marking

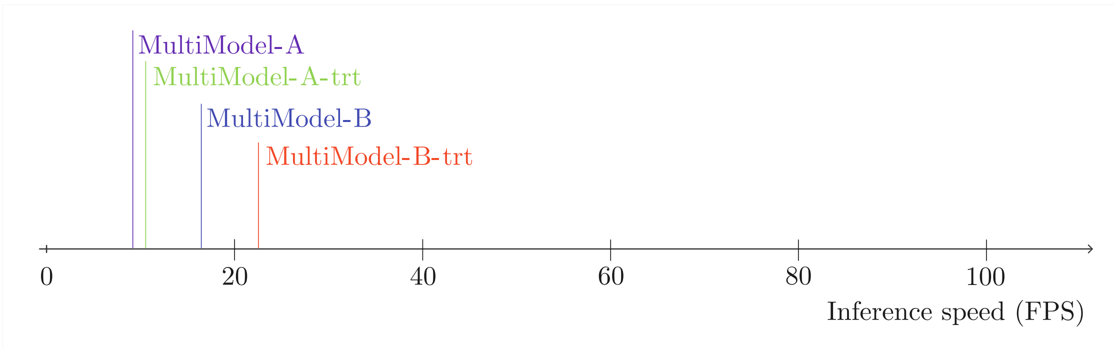


Figure 5.39: Comparison between the MultiModels with models optimized with TensorRT and without

5.5 Tests in a self-distributed dataset

The purpose of this section is to assess the efficacy of the top-performing multi-task neural networks when confronted with images sourced not from common datasets and to give a better idea of how networks would behave in ATLASCAR2 when implemented. To accomplish this, a bespoke dataset consisting of six videos has been created. These videos were recorded using the Logitech C270 HD webcam while driving a car in various cities in Portugal, including Peniche, Santarém, and Rio Maior. The chosen vehicle for this endeavor was a Nissan Leaf, distinct from ATLASCAR2, and the webcam was positioned in proximity to the rear-view mirror. Each video contributed a randomly selected frame for analysis in this section.

Due to time constraints, the images were not labeled, therefore there is no ground truth. Consequently, this section opts for a qualitative evaluation approach. The resultant images are presented in figure 5.40, where the images on the left represent the outcomes of employing TwinLiteNet, while those on the right depict the results of utilizing YOLOPv2.

Car detection Only the YOLOPv2 is capable of performing car detection. In general, it performed well, but it is possible to see a FP in the second-to-last row of images where a bush is identified as a car. In the remaining cases, all cars are correctly identified.

Drivable Area Segmentation It is possible to see that both neural networks exhibit a commendable ability to effectively segment the drivable area, but TwinLiteNet did better. They indeed distinguish the drivable region of the road from the non-drivable.

Lane Marking The YOLOPv2 did better in general for giving more consistent lines. However, it is interesting to see that TwinLiteNet is somewhat capable of identifying the limits of the road (last row of images), showing a big benefit of the multi-task approach for a better understanding of the context of the road.

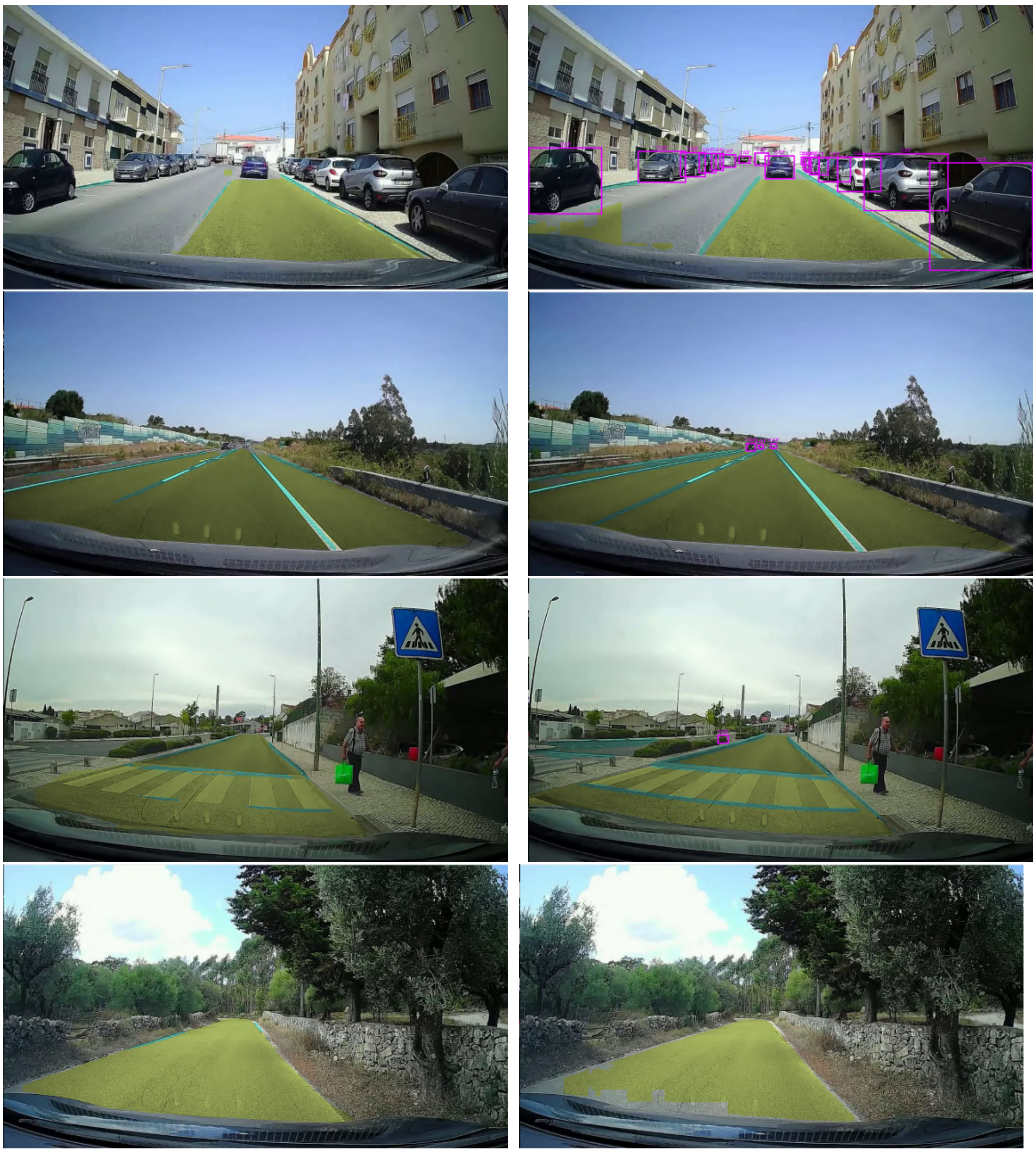


Figure 5.40: Inference of multi-tasked models in images not from common datasets: TwinLiteNet in the left and YOLOPv2 in the right

Overall summary

This chapter started by elucidating the process of acquiring the model files for subsequent testing and evaluation. After that, each functionality of the application, as detailed in Chapter 4, was showcased through the application's output and the ROS graphs generated with ROS. Following that, performance comparisons among the models were undertaken. These comparisons revealed that utilizing multiple models leads to a relatively slow inference speed and introduces synchronization challenges inherently addressed by multi-tasked networks. Furthermore, the chapter shows the influence of destructive interference in the object detection task by comparing its performance in the single and multi-tasked configurations. Concluding this chapter, the best models were benchmarked using the Jetson, and some qualitative evaluations were done on the multi-tasked networks using images obtained using the Logitech webcam.

Intentionally blank page.

Chapter 6

Conclusions and future work

This chapter contains the summary and the conclusions taken with the development of this work, as well as some of the future paths that can be followed using this work as a starting point.

6.1 Conclusions

This dissertation started with the challenge of comparing multi-tasked networks with multiple single-tasked networks in the context of ADAS and ADS perception and creating software to do that while being able to communicate within a ROS environment.

In tackling these challenges, the dissertation initiated by exploring the concept of car perception and the feasible pathways in this domain. Regarding perception, the dissertation delved into object detection and image segmentation. It then delved into the concepts of multi-tasked neural networks to select appropriate models for single and multi-task networks and better analyze the networks' behavior compared to each other. The tasks under this study encompassed car detection, road segmentation, and lane marking.

For object detection, three single-tasked neural networks, namely YOLOv5, YOLOv7, and YOLOv8, were chosen. For road segmentation, the selections were Mask2Former, UperNet, and SegFormer, while lane marking was approached with RESA, O2SFormer, and UFLDv2. The multi-tasked networks scrutinized included YOLOP, YOLOPV2, and the TwinLiteNet. Various evaluation metrics were explored to assess model performance, with AP for object detection and IoU for image segmentation being the selected ones. Speed was quantified in terms of FPS.

The developed application was designed to interface with ROS topics for data streaming, route the data through the model for inference, and send the results through other topics. However, complications arose during development due to different model formats, input requirements, and output formats. Consequently, the application needed to accommodate diverse image transformations and model formats and standardize output for ease of handling on receiver devices. The application adopts a modular architecture for robustness and simplicity, utilizing different components based on the model's features. Two versions of the application were created: one for illustration and application testing and another for model evaluation.

In the evaluation phase, models were tested under various optimization settings to maximize their chance of being in the best format possible. The analysis revealed that the best single-task models were YOLOv8 for car detection, SegFormer for drivable area segmentation, and UFLDv2 for lane marking. Among the multi-task models, for three tasks, YOLOPv2 excels in handling all tasks analyzed, and for two tasks, TwinLiteNet demonstrates the best performance in the segmentation tasks. To evaluate multiple models simultaneously, the best single-task models were combined. However, due to suboptimal results observed with SegFormer and UFLDv2, a secondary test was conducted, employing two instances of TwinLiteNet to mitigate these outcomes.

The results lead to the conclusion that the inference speed of combined models is slower than the speed of multi-tasked networks. When multiple models are concurrently executed, the framerate of each task is different, leading to worse synchronization. The addition of object detection done by the authors of YOLOP leads to some destructive interference, which can happen during the training stage due to the implementation of tasks with different learning needs, decreasing the performance when compared with other standard YOLO, but a good balance between accuracy and speed is still achieved.

Overall, in comparing the multiple models with multi-tasked networks, the main takeaways are that utilizing multiple models leads to a relatively slow inference speed and introduces synchronization challenges inherently addressed by multi-tasked networks. However, multi-tasked models, when using not-so-related tasks (with different learning needs), suffer from destructive interference while training, leading to worse performance. In conclusion, multi-task neural networks are indeed better than using multiple models. Still, the set of tasks is something to consider because the more related the tasks, the better the chance of the model having excellent performance.

6.2 Future work

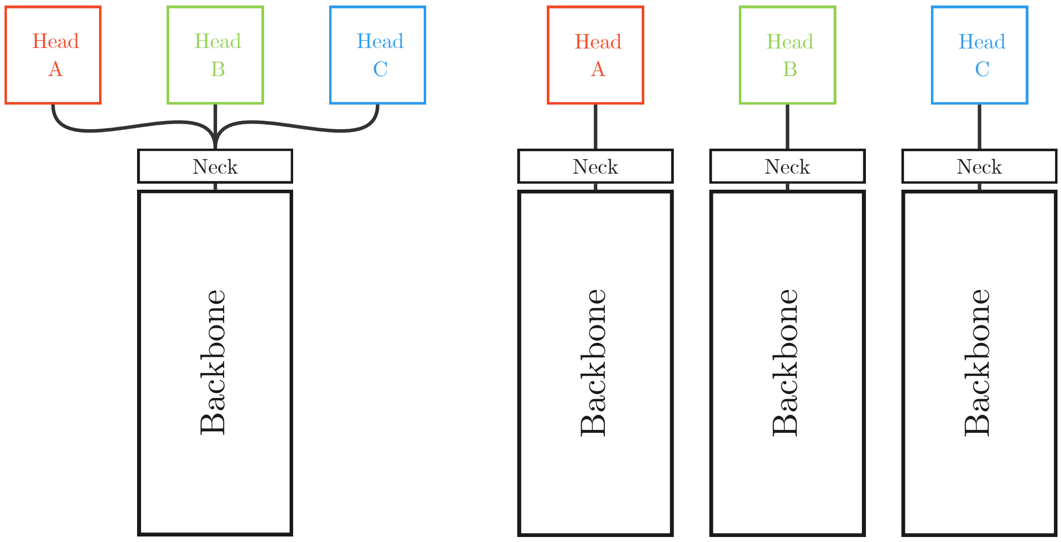
This dissertation leaves multiple possible paths for the future. These paths can be divided into two categories regarding improvements of the developed application and model architectures.

Application

Utilizing the established application as a foundation is feasible to enhance its capabilities by introducing additional tasks, such as 3D object detection, tracking, or even tasks related to other kinds of data, like LiDAR. Another path for application improvement involves investigating methods to increase the overlap of tasks and assess whether this enhances inference speed. Furthermore, the application's performance could be potentially enhanced by exploring alternatives to the current "Inference solution", such as Nvidia's solutions. This has the potential to improve the application. If the result is not an improvement, it also serves as a valuable means to gauge the effectiveness of alternative solutions.

Models

This study evaluated various neural networks with readily available implementations. An interesting avenue for further exploration, focused on the model's architecture, involves the use of multi-tasked models with multiple heads, as in figure 6.1a, to generate single-tasked models, by removing heads, as schematized in figure 6.1b. These single-tasked models can then be retrained to evaluate whether there are performance advantages. In this approach, network architectures remain consistent, differing primarily in the heads, facilitating an assessment of the efficacy of the multi-task learning approach.



(a) Generic multi-headed neural network (b) Generic multiple models

Figure 6.1: Multiple models from a multi-tasked neural network

Another path of research would involve the examination of networks such as the YOSO, which can perform tasks typically accomplished through subtasks while avoiding using them.

Finally, instead of optimizing each model individually for optimizing multiple models, a newer Nvidia solution[1] designed to optimize multiple models together could be used as it is a better-suited solution for this kind of optimization.

[1]https://www.nvidia.com/en-us/on-demand/session/gtcspring22-s41169/

Intentionally blank page.

Appendix A

ROS Graphs

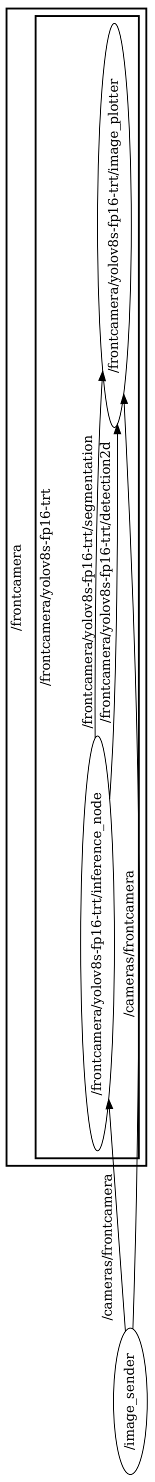

/frontcamera
/frontcamera/yolov8s-fp16-trt
/frontcamera/yolov8s-fp16-trt/image_plotter
/frontcamera/yolov8s-fp16-trt/segmentation
/frontcamera/yolov8s-fp16-trt/detection2d
/frontcamera/yolov8s-fp16-trt/inference_node
/cameras/frontcamera
/cameras/frontcamera
/image_sender

Figure A.1: ROS graph of a single-tasked neural network with one input and one output

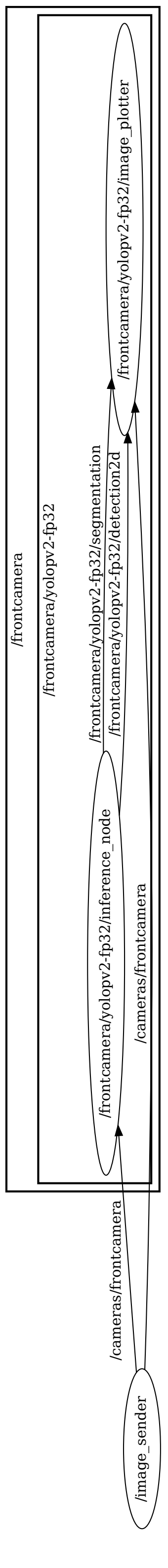

/frontcamera
/frontcamera/yolopv2-fp32
/frontcamera/yolopv2-fp32/image_plotter
/frontcamera/yolopv2-fp32/segmentation
/frontcamera/yolopv2-fp32/detection2d
/frontcamera/yolopv2-fp32/inference_node
/cameras/frontcamera
/cameras/frontcamera
/image_sender

Figure A.2: ROS graph of a multi-tasked neural network with one input and one output

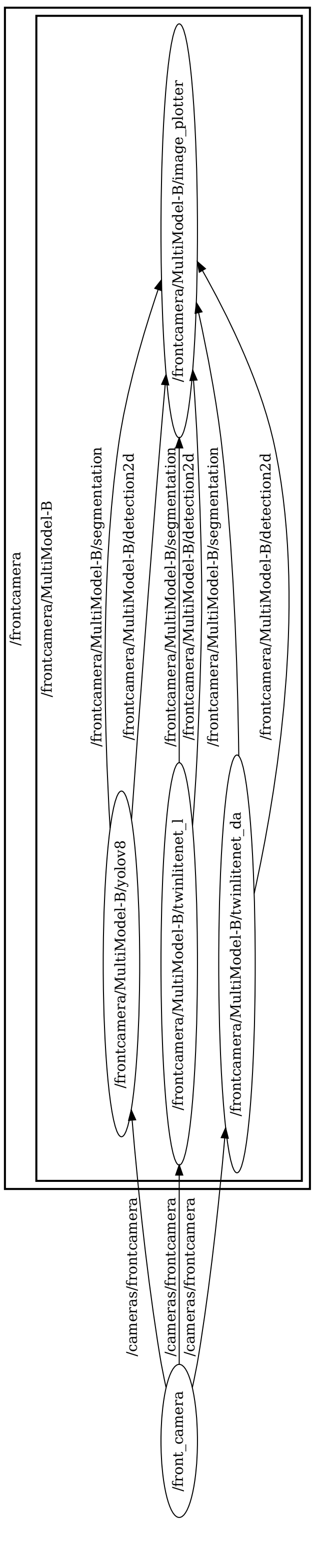

/frontcamera
/frontcamera/MultiModel-B
/frontcamera/MultiModel-B/image_plotter
/frontcamera/MultiModel-B/segmentation
/frontcamera/MultiModel-B/detection2d
/frontcamera/MultiModel-B/segmentation
/frontcamera/MultiModel-B/detection2d
/frontcamera/MultiModel-B/segmentation
/frontcamera/MultiModel-B/detection2d
/frontcamera/MultiModel-B/yolov8
/frontcamera/MultiModel-B/twinlitenet_l
/frontcamera/MultiModel-B/twinlitenet_da
/cameras/frontcamera
/cameras/frontcamera
/cameras/frontcamera
/front_camera

Figure A.3: ROS graph of multiple neural networks with one input and one output

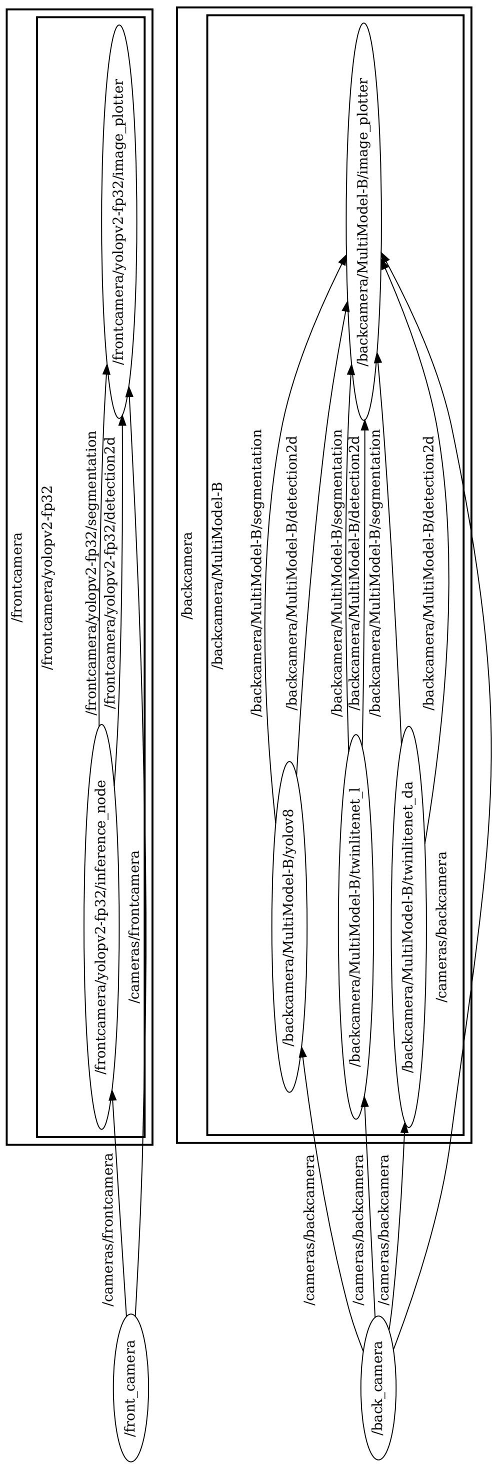

/frontcamera
/frontcamera/yolopv2-fp32
/frontcamera/yolopv2-fp32/image_plotter
/frontcamera/yolopv2-fp32/segmentation
/frontcamera/yolopv2-fp32/detection2d
/frontcamera/yolopv2-fp32/inference_node
/cameras/frontcamera
/cameras/frontcamera
/front_camera
/backcamera
/backcamera/MultiModel-B
/backcamera/MultiModel-B/image_plotter
/backcamera/MultiModel-B/segmentation
/backcamera/MultiModel-B/detection2d
/backcamera/MultiModel-B/segmentation
/backcamera/MultiModel-B/detection2d
/backcamera/MultiModel-B/segmentation
/backcamera/MultiModel-B/detection2d
/backcamera/MultiModel-B/yolov8
/backcamera/MultiModel-B/twinlitenet_l
/backcamera/MultiModel-B/twinlitenet_da
/cameras/backcamera
/cameras/backcamera
/cameras/backcamera
/cameras/backcamera
/back_camera

Figure A.4: ROS graph of multiple neural networks with two inputs and two outputs

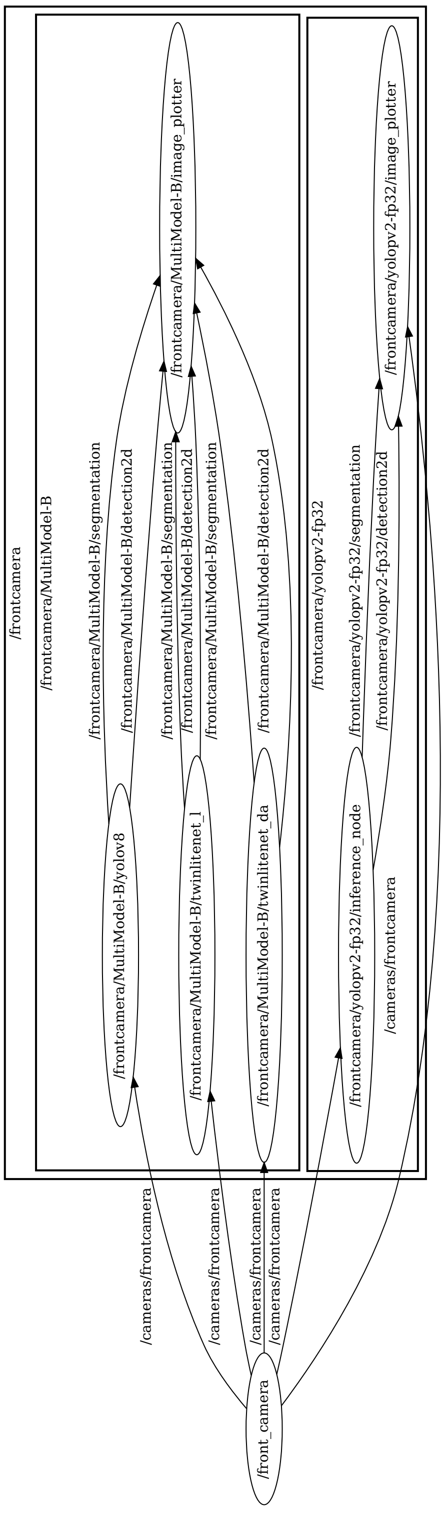

/frontcamera
/frontcamera/MultiModel-B
/frontcamera/yolopv2-fp32
/frontcamera/MultiModel-B/image_plotter
/frontcamera/yolopv2-fp32/image_plotter
/frontcamera/MultiModel-B/segmentation
/frontcamera/MultiModel-B/detection2d
/frontcamera/MultiModel-B/segmentation
/frontcamera/MultiModel-B/detection2d
/frontcamera/MultiModel-B/segmentation
/frontcamera/MultiModel-B/detection2d
/frontcamera/yolopv2-fp32/segmentation
/frontcamera/yolopv2-fp32/detection2d
/cameras/frontcamera
/frontcamera/MultiModel-B/yolov8
/frontcamera/MultiModel-B/twinlitenet_l
/frontcamera/MultiModel-B/twinlitenet_da
/frontcamera/yolopv2-fp32/inference_node
/cameras/frontcamera
/cameras/frontcamera
/cameras/frontcamera
/cameras/frontcamera
/front_camera

Figure A.5: ROS graph of multiple neural networks with two inputs and one output

Appendix B

Results on the laptop

Table B.1: Detailed results of object detection - Laptop

	AP@50	AP@75	FPS	Model size (MB)
YoloV5s-Fp16-trt	0.5633	0.4210	279.26	30
YoloV5s-Fp16	0.5646	0.4165	163.88	15
YoloV5s-Fp32	0.5677	0.4181	156.13	30
YoloV7-Fp16	0.6953	0.5498	72.31	75
YoloV7-Fp32	0.6956	0.5557	43.89	150
YoloV8s-Fp16-trt	0.7144	0.5660	21.18	64
YoloV8s-Fp16	0.7141	0.5653	174.07	23
YoloV8s-Fp32-trt	0.7147	0.5648	20.84	66
YoloV8s-Fp32	0.7148	0.5648	120.40	45
Yolop-Fp16-trt	0.4161	0.3397	22.52	40
Yolop-Fp16	0.4174	0.3394	104.66	16
Yolop-Fp32-trt	0.4190	0.3364	22.18	56
Yolop-Fp32	0.4191	0.3362	94.38	32
YolopV2-Fp16-trt	0.5504	0.4167	92.22	113
YolopV2-Fp16	0.5647	0.4194	61.86	78
YolopV2-Fp32-trt	0.5522	0.4206	54.72	247
YolopV2-Fp32	0.5520	0.4203	45.79	156
MultiModel-A	0.7141	0.5653	25.93	292
MultiModel-A (task)	0.7141	0.5653	163.00	292
MultiModel-B	0.7141	0.5653	46.98	31
MultiModel-B (task)	0.7141	0.5653	160.18	31

Table B.2: Detailed results of drivable area segmentation - Laptop

	IoU	FPS	Model size (MB)
Mask2Former-Fp32	0.5408	8.11	190
SegFormer-Fp16-trt	0.4564	13.10	21
SegFormer-Fp16	0.4564	41.93	9
SegFormer-Fp32-trt	0.4565	10.93	21
SegFormer-Fp32	0.4565	38.90	16
TwinLiteNet-Fp16-trt	0.8014	259.33	4
TwinLiteNet-Fp16	0.8015	159.96	1
TwinLiteNet-Fp32-trt	0.8014	185.62	7
TwinLiteNet-Fp32	0.8014	143.41	2
UperNet-Fp16	0.5742	8.51	120
UperNet-Fp32	0.5734	6.50	240
Yolop-Fp16-trt	0.7703	22.52	40
Yolop-Fp16	0.7703	104.66	16
Yolop-Fp32-trt	0.7702	22.18	56
Yolop-Fp32	0.7703	94.38	32
YolopV2-Fp16-trt	0.8261	92.22	113
YolopV2-Fp16	0.8261	61.86	78
YolopV2-Fp32-trt	0.8261	54.72	247
YolopV2-Fp32	0.8261	45.79	156
MultiModel-A	0.4564	25.93	292
MultiModel-A (task)	0.4564	37.73	292
MultiModel-B	0.8014	46.98	31
MultiModel-B (task)	0.8014	179.29	31

Table B.3: Detailed results of lane marking - Laptop

	IoU	FPS	Model size (MB)
O2SFormer-Fp32	0.0553	67.86	125
Resa-Fp16	0.0156	102.90	45
Resa-Fp32	0.0156	63.93	90
TwinLiteNet-Fp16-trt	0.2990	259.33	4
TwinLiteNet-Fp16	0.2991	159.96	1
TwinLiteNet-Fp32-trt	0.2992	185.62	7
TwinLiteNet-Fp32	0.2992	143.41	2
UFLDv2-Fp16-trt	0.0739	115.28	260
UFLDv2-Fp32-trt	0.0740	89.71	530
UFLDv2-Fp32	0.0740	86.55	385
Yolop-Fp16-trt	0.3415	22.52	40
Yolop-Fp16	0.3414	104.66	16
Yolop-Fp32-trt	0.3418	22.18	56
Yolop-Fp32	0.3418	94.38	32
YolopV2-Fp16-trt	0.4137	92.22	113
YolopV2-Fp16	0.4137	61.86	78
YolopV2-Fp32-trt	0.4137	54.72	247
YolopV2-Fp32	0.4137	45.79	156
MultiModel-A	0.0739	25.93	292
MultiModel-A (task)	0.0739	91.26	292
MultiModel-B	0.2991	46.98	31
MultiModel-B (task)	0.2991	184.50	31

Intentionally blank page.

Appendix C

Results on the Jetson device

Table C.1: Detailed results of object detection - Jetson

	AP@50	AP@75	FPS	Model size (MB)
MultiModel-A-trt	0.6683	0.5282	10.55	292
MultiModel-A (task)	0.6683	0.5282	9.20	417
MultiModel-B-trt	0.6683	0.5282	22.51	25.6
MultiModel-B (task)	0.6683	0.5282	16.47	25
YoloV8s-Fp16	0.6683	0.5282	49.50	23
YolopV2-Fp16-trt	0.5546	0.4200	27.93	109
YolopV2-Fp16	0.5506	0.4203	16.90	78

Table C.2: Detailed results of drivable area segmentation - Jetson

	IoU	FPS	Model size (MB)
MultiModel-A-trt	0.4564	10.55	292
MultiModel-A (task)	0.4564	9.20	417
MultiModel-B-trt	0.8014	22.51	25.6
MultiModel-B (task)	0.8015	16.47	25
SegFormer-Fp16	0.4564	14.65	9
TwinLiteNet-Fp16-trt	0.8014	79.31	1.3
YolopV2-Fp16-trt	0.8261	27.99	109
YolopV2-Fp16	0.8250	16.92	78

Table C.3: Detailed results of lane marking - Jetson

	IoU	FPS	Model size (MB)
MultiModel-A-trt	0.0739	10.55	292
MultiModel-A (task)	0.0740	9.20	417
MultiModel-B-trt	0.2992	22.51	25.6
MultiModel-B (task)	0.2992	16.47	25
TwinLiteNet-Fp16-trt	0.2992	79.31	1.3
UFLDv2-Fp16-trt	0.0739	34.01	260
YolopV2-Fp16-trt	0.4138	27.99	109
YolopV2-Fp16	0.4130	16.92	78

Appendix D

Installation instructions

D.1 Requirements

For the utilization of a 64-bit machine, an Nvidia GPU is a requisite. The developed software has not been prepared or subjected to testing on alternative platforms.

The operating system in use should either be Ubuntu or a distribution based on Ubuntu. Throughout this dissertation, the software underwent testing on Ubuntu 20.04 and PopOS 20.04.

D.2 Dependencies installation

For both Jetson and other 64-bit devices, certain software prerequisites must be installed prior to the installation of the software developed within this dissertation. It is important to note that the steps for installing these dependencies differ between the Jetson device and a generic 64-bit machine.

D.2.1 Jetson AGX Xavier

On a Jetson device, the operating system needs to be flashed into the memory. Given the limited internal storage capacity of the Jetson device, it is advisable to flash it onto an external storage device with greater capacity. This task, however, is not as straightforward as it may seem, which is why the instructions for flashing the Jetson onto an additional storage device are provided within this section.

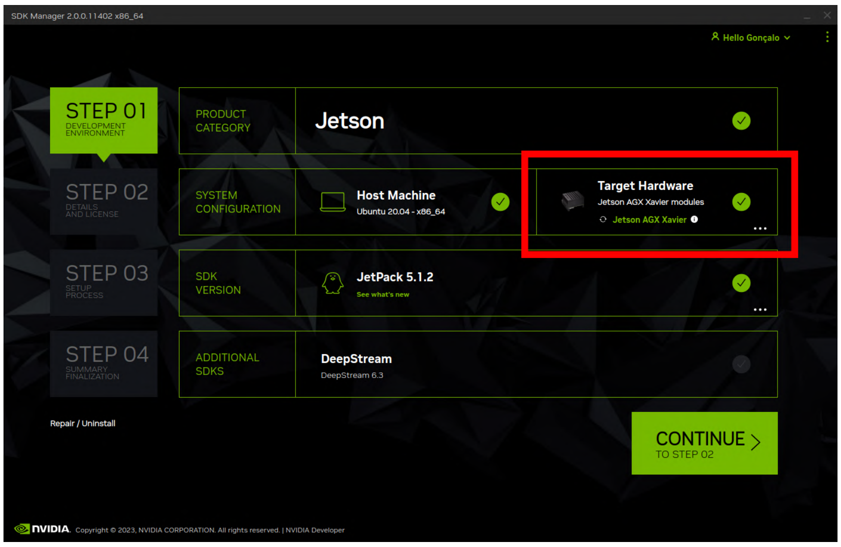
Flashing the Jetson device

To initiate the flashing process for the Jetson, an additional computer with approximately 60 GB of available storage is essential. On this computer, install the Nvidia SDK Manager[1] and follow the provided instructions for installation on the Nvidia website.

Next, it is time to place the Jetson device into recovery mode. Ensure that the Jetson is plugged in but powered off. Additionally, connect the Jetson to both a monitor and the computer via the USB-C port adjacent to the GPIO pins. To enter recovery mode, press and hold the recovery button for an extended duration while simultaneously pressing and holding the power button. A blinking white dash should be observed on the screen.

[1]https://developer.nvidia.com/sdk-manager

On the computer, launch the Nvidia SDK Manager and confirm whether the Jetson is detected. The Nvidia manager interface should resemble the one depicted in figure D.1.


SDK Manager 2.0.0.11402 x86_64

Hello Gonçalo

STEP 01 DEVELOPMENT ENVIRONMENT
STEP 02 DETAILS AND LICENSE
STEP 03 SETUP PROCESS
STEP 04 SUMMARY FINALIZATION

PRODUCT CATEGORY: Jetson ✓

SYSTEM CONFIGURATION: Host Machine, Ubuntu 20.04 - x86_64 ✓ | Target Hardware, Jetson AGX Xavier modules, Jetson AGX Xavier ✓

SDK VERSION: JetPack 5.1.2, See what's new ✓

ADDITIONAL SDKS: DeepStream, DeepStream 6.3

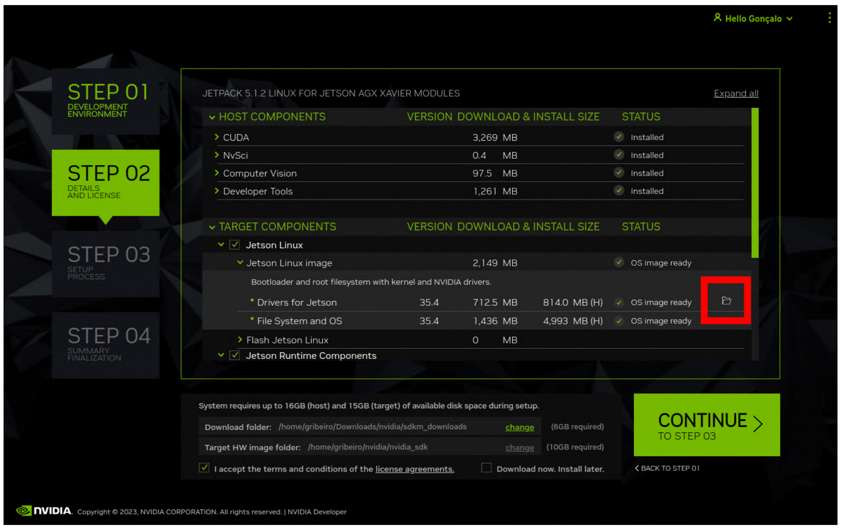
Repair / Uninstall

CONTINUE TO STEP 02

Copyright © 2023, NVIDIA CORPORATION. All rights reserved. | NVIDIA Developer

Figure D.1: Jetson device recognized in the Nvidia Manager SDK

Once the Jetson is identified, proceed by selecting the ”CONTINUE TO STEP 02” option. Subsequently, ensure that each checkbox is marked, choose ”Download Now. Install later”, and agree to the terms and conditions. Following the download, reopen the Nvidia SDK Manager and re-enter step 2. Within the ”TARGET COMPONENTS” tree, expand the ”Jetson Linux image” section and select the button highlighted in figure D.2 (labeled ”Drivers for Jetson”).


Hello Gonçalo

STEP 01 DEVELOPMENT ENVIRONMENT
STEP 02 DETAILS AND LICENSE
STEP 03 SETUP PROCESS
STEP 04 SUMMARY FINALIZATION

JETPACK 5.1.2 LINUX FOR JETSON AGX XAVIER MODULES — Expand all

HOST COMPONENTS	VERSION	DOWNLOAD & INSTALL SIZE		STATUS
CUDA		3,269 MB		✓ Installed
NvSci		0.4 MB		✓ Installed
Computer Vision		97.5 MB		✓ Installed
Developer Tools		1,261 MB		✓ Installed

TARGET COMPONENTS	VERSION	DOWNLOAD & INSTALL SIZE		STATUS
✓ Jetson Linux				
Jetson Linux image		2,149 MB		✓ OS image ready
Bootloader and root filesystem with kernel and NVIDIA drivers.				
Drivers for Jetson	35.4	712.5 MB	814.0 MB (H)	✓ OS image ready
File System and OS	35.4	1,436 MB	4,993 MB (H)	✓ OS image ready
Flash Jetson Linux		0 MB		
✓ Jetson Runtime Components				

System requires up to 16GB (host) and 15GB (target) of available disk space during setup.

Download folder: /home/gribeiro/Downloads/nvidia/sdkm_downloads change (6GB required)

Target HW image folder: /home/gribeiro/nvidia/nvidia_sdk change (10GB required)

[✓] I accept the terms and conditions of the license agreements.   [ ] Download now. Install later.

CONTINUE TO STEP 03

BACK TO STEP 01

Copyright © 2023, NVIDIA CORPORATION. All rights reserved. | NVIDIA Developer

Figure D.2: Folder button in the Nvidia Manager SDK

This action will open a directory housing the ”Linux_for_Tegra” folder. Within this directory, locate the file named ”flash_l4t_nvme.xml”. This particular file contains the definition of the ”num_sectors” variable, identified as ”NUM_SECTORES”. It’s important to note that the Nvidia SDK Manager is presently unable to automatically modify the value of "NUM_SECTORES". Nevertheless, it is feasible to manually adjust "NUM_SECTORES" to the correct value.

The number of sectors can be calculated using equation D.1, assuming the default sector size of 512 bytes.

$$\text{Number of sectors} = \frac{\text{Disk size} \times 1024^3}{\text{Sector size}} \tag{D.1}$$

After setting the right sector size, save the document and run the following terminal command in the "Linux_for_Tegra" directory to flash the Jetson device:

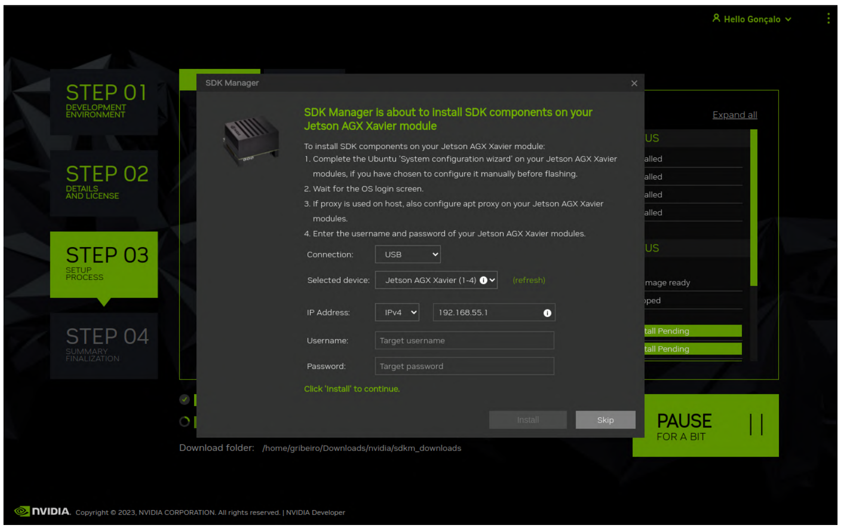
```
sudo ./tools/kernel_flash/l4t_initrd_flash.sh --external-device nvme0n1
-c ./tools/kernel_flash/flash_l4t_nvme.xml -S 232GiB --showlogs --erase-all
jetson-agx-xavier-devkit nvme0n1p1
```

Where "232GiB" should be adjusted according to the used disk.

Setup and package installation

Following the flashing process, the Jetson will initiate the Ubuntu operating system and prompt for final configuration settings, such as the selection of a username and password. Once these settings are configured, establish a network connection between the Jetson and the computer. Subsequently, reopen the Nvidia SDK Manager.

Skip the flashing process, and the installation menu will pop up (figure D.3). Input the Jetson's IP address, username, and password into the designated fields, then proceed to initiate the installation. Upon the completion of this installation, the Jetson can be disconnected from the computer, rendering the computer no longer necessary.


Hello Gonçalo
STEP 01 DEVELOPMENT ENVIRONMENT
STEP 02 DETAILS AND LICENSE
STEP 03 SETUP PROCESS
STEP 04 SUMMARY FINALIZATION
SDK Manager
Expand all
Install
Skip
PAUSE FOR A BIT
NVIDIA. Copyright © 2023, NVIDIA CORPORATION. All rights reserved. | NVIDIA Developer

SDK Manager is about to install SDK components on your Jetson AGX Xavier module

To install SDK components on your Jetson AGX Xavier module:
1. Complete the Ubuntu 'System configuration wizard' on your Jetson AGX Xavier modules, if you have chosen to configure it manually before flashing.
2. Wait for the OS login screen.
3. If proxy is used on host, also configure apt proxy on your Jetson AGX Xavier modules.
4. Enter the username and password of your Jetson AGX Xavier modules.

Connection: USB
Selected device: Jetson AGX Xavier (1-4) (refresh)
IP Address: IPv4 192.168.55.1
Username: Target username
Password: Target password

Click 'Install' to continue.

Download folder: /home/gribeiro/Downloads/nvidia/sdkm_downloads

Figure D.3: SDK components installation menu in the Nvidia Manager SDK

Now, for the installation of PyTorch, two methods are available. The first method involves searching Nvidia's website for "whl" files containing the desired PyTorch version specifically tailored for Jetson devices. The second method, which should only be employed in cases where Nvidia does not provide the requisite "whl" file for the intended PyTorch version, is to compile PyTorch from its source code. Instructions for this can be found within the PyTorch repository, but it is advisable to export a "whl" file as a backup option. The PyTorch version utilized in this dissertation (v2.0.0) is accessible through Nvidia's website[2].

Torchvision also necessitates installation from source, with the requirement of using the specific version v0.15.2 to ensure compatibility with the PyTorch version being utilized.

Installing Torch-TensorRT can be particularly challenging, as it lacks compatible versions for the most recent JetPack releases. An examination of the versions of TensorRT and CUDA within JetPack reveals a lack of compatibility with Torch-TensorRT. The workaround, although not pretty, is effective: download the Torch-TensorRT version that aligns with the PyTorch version in use and has the closest match to the TensorRT and CUDA versions required (Torch-TensorRT v1.4.0 in this dissertation). Installing this package demands some adjustments due to conflicting TensorRT and CUDA versions. For TensorRT, the solution starts by executing the following command in the package folder directory:

```
grep -r TensorRT
```

With this, it is possible to identify the files containing version restrictions and make the necessary edits to incorporate the specific TensorRT version in use. After this adjustment, access the 'WORKSPACES' file, where all instances of "http_archive" should be commented out, and the "new_local_repository" entries uncommented. Within these "new_local_repository" entries, paths for locally installed packages are defined. Ensure that these paths are accurately adjusted to match the appropriate locations. In the 'WORKSPACE' file, also replace occurrences of "/usr/local/cuda-12.1/" with "/usr/local/cuda/" to accommodate any installed CUDA version. Once these adjustments are completed, follow the installation instructions provided in the official repository.

The remaining packages, including numpy, opencv2, and scipy, can be installed directly using the pip package manager.

D.2.2 64-bit devices

On a 64-bit device, the first step involves updating the Nvidia drivers, a process that can be accomplished by following the instructions provided by Nvidia. Running the command `nvidia-smi` allows one to ascertain the value of "CUDA version." It's important to note that this value represents the maximum version compatible with the currently installed drivers, rather than the version of the CUDA software.

The CUDA software (version 11.8) can be installed by following Nvidia's instructions, but it's advisable to include additional functionalities, such as the ability to check the installed version. The CUDA software is typically installed in the "/usr/local/" directory, with one folder named "CUDA" and at least one additional folder following the structure "CUDA-XX.X." If multiple "CUDA-XX.X" folders are present, it's recommended to refer to Nvidia's official instructions to uninstall unintended versions. Regardless, the system utilizes the software contained within the "CUDA" folder. To view its version, the following line should be added to the ".bashrc" file using the command:

```
"export PATH=/usr/local/cuda-xx.x/bin${PATH:+:${PATH}}" >> ~\.bashrc
```

For this to take effect, run `source ~\.bashrc`.

[2]https://forums.developer.nvidia.com/t/pytorch-for-jetson/72048

TensorRT (version 8.6) can be installed by following the Nvidia official instructions, as well as Torch-TensorRT (v1.4.0).

The installation of PyTorch (v2.0.0) and Torchvision (v0.15.2) is a straightforward process. Simply follow the instructions provided in the official repositories.

For the remaining packages, including numpy, opencv2, scipy, and Polygraphy, direct installation can be performed using pip.

D.3 Software installation

The initial step involves the installation of ROS Noetic and cv-bridge, followed by the creation of a catkin workspace. To achieve this, it is recommended to refer to the official ROS instructions.

The subsequent step is to clone the dissertation's dissertation's GitHub repository[3], into the "src" folder within the catkin workspace. Finally, within the workspace directory, execute the command `catkin_make`. To complete the process, create a folder named "models" inside the "perception_with_multi-task_neural_networks" folder and proceed to place the models within it.

[3]https://github.com/GoncaloR00/perception_with_multi-task_neural_networks

Intentionally blank page.

References

[1] V. Santos, "ATLASCAR: A sample of the quests and concerns for autonomous cars," *Lecture Notes in Electrical Engineering*, vol. 495, pp. 355–375, 2020, Publisher: Springer Verlag ISBN: 9783030112912, ISSN: 18761119. DOI: 10.1007/978-3-030-11292-9_18/COVER. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-11292-9_18 (visited on 10/31/2023).

[2] V. K. Kukkala, J. Tunnell, S. Pasricha, and T. Bradley, "Advanced driver-assistance systems: A path toward autonomous vehicles," *IEEE Consumer Electronics Magazine*, vol. 7, no. 5, pp. 18–25, Sep. 2018, ISSN: 2162-2248. DOI: 10.1109/MCE.2018.2828440.

[3] P. Azevedo and V. Santos, "Comparative analysis of multiple YOLO-based target detectors and trackers for ADAS in edge devices," *Robotics and Autonomous Systems*, vol. 171, p. 104 558, Jan. 1, 2024, Publisher: North-Holland, ISSN: 0921-8890. DOI: 10.1016/J.ROBOT.2023.104558. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0921889023001975 (visited on 10/29/2023).

[4] J. Kaur and W. Singh, "Tools, techniques, datasets and application areas for object detection in an image: A review," *Multimedia Tools and Applications*, vol. 81, no. 27, pp. 38 297–38 351, Nov. 23, 2022, ISSN: 1380-7501. DOI: 10.1007/s11042-022-13153-y.

[5] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-CNN: Towards real-time object detection with region proposal networks," Jun. 4, 2015. arXiv: 1506.01497.

[6] J. Sushma and M. K. Pandey, "A review on image segmentation," in *Rising Threats in Expert Applications and Solutions*, R. V. Singh, N. Dey, P. Vincenzo, B. Rosalina, P. Zdzislaw, and T. J. M. R. S, Eds., Singapore: Springer Singapore, 2021, pp. 233–240, ISBN: 978-981-15-6014-9.

[7] V. Badrinarayanan, A. Kendall, and R. Cipolla, "SegNet: A deep convolutional encoder-decoder architecture for image segmentation," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 12, pp. 2481–2495, Dec. 1, 2017, ISSN: 0162-8828. DOI: 10.1109/TPAMI.2016.2644615.

[8] J. Long, E. Shelhamer, and T. Darrell, "Fully convolutional networks for semantic segmentation," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2015, pp. 3431–3440, ISBN: 978-1-4673-6964-0. DOI: 10.1109/CVPR.2015.7298965.

[9] O. Ronneberger, P. Fischer, and T. Brox, "U-net: Convolutional networks for biomedical image segmentation," in 2015, pp. 234–241. DOI: 10.1007/978-3-319-24574-4_28.

[10] A. Vaswani *et al.*, “Attention is all you need,” *Advances in Neural Information Processing Systems*, vol. 30, 2017. (visited on 10/29/2023).

[11] A. Dosovitskiy *et al.*, “An image is worth 16x16 words: Transformers for image recognition at scale,” *ICLR 2021 - 9th International Conference on Learning Representations*, Oct. 22, 2020, Publisher: International Conference on Learning Representations, ICLR. arXiv: `2010.11929`. [Online]. Available: `https://arxiv.org/abs/2010.11929v2` (visited on 10/29/2023).

[12] D. Bolya, C. Zhou, F. Xiao, and Y. J. Lee, “YOLACT: Real-time instance segmentation,” in *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, IEEE, Oct. 2019, pp. 9156–9165, ISBN: 978-1-72814-803-8. DOI: `10.1109/ICCV.2019.00925`.

[13] Y.-C. Hsu, Z. Xu, Z. Kira, and J. Huang, “Learning to cluster for proposal-free instance segmentation,” Mar. 17, 2018. arXiv: `1803.06459`.

[14] K. He, G. Gkioxari, P. Dollar, and R. Girshick, “Mask r-CNN,” in *2017 IEEE International Conference on Computer Vision (ICCV)*, IEEE, Oct. 2017, pp. 2980–2988, ISBN: 978-1-5386-1032-9. DOI: `10.1109/ICCV.2017.322`.

[15] A. Kirillov, K. He, R. Girshick, C. Rother, and P. Dollar, “Panoptic segmentation,” in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2019, pp. 9396–9405, ISBN: 978-1-72813-293-8. DOI: `10.1109/CVPR.2019.00963`.

[16] J. Hu, L. Huang, T. Ren, S. Zhang, R. Ji, and L. Cao, “You only segment once: Towards real-time panoptic segmentation,” *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 17 819–17 829, Jun. 1, 2023, Publisher: IEEE Computer Society ISBN: 979-8-3503-0129-8. DOI: `10.1109/CVPR52729.2023.01709`. [Online]. Available: `https://ieeexplore.ieee.org/document/10204006/` (visited on 09/14/2023).

[17] I. H. Sarker, “Deep learning: A comprehensive overview on techniques, taxonomy, applications and research directions,” *SN Computer Science*, vol. 2, no. 6, p. 420, Nov. 18, 2021, ISSN: 2662-995X. DOI: `10.1007/s42979-021-00815-1`.

[18] M. Crawshaw, “Multi-task learning with deep neural networks: A survey,” Sep. 10, 2020. arXiv: `2009.09796`.

[19] S. Ruder, “An overview of multi-task learning in deep neural networks,” Jun. 15, 2017. arXiv: `1706.05098`.

[20] K.-H. Thung and C.-Y. Wee, “A brief review on multi-task learning,” *Multimedia Tools and Applications*, vol. 77, no. 22, pp. 29 705–29 725, Nov. 8, 2018, ISSN: 1380-7501. DOI: `10.1007/s11042-018-6463-x`.

[21] S. Ruder, J. Bingel, I. Augenstein, and A. Søgaard, “Latent multi-task architecture learning,” *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, no. 1, pp. 4822–4829, Jul. 17, 2019, ISSN: 2374-3468. DOI: `10.1609/aaai.v33i01.33014822`.

[22] M. Shroff. “Know your neural network architecture more by understanding these terms — by megha shroff — medium.” (), [Online]. Available: `https://medium.com/@shroffmegha6695/know-your-neural-network-architecture-more-by-understanding-these-terms-67faf4ea0efb` (visited on 09/14/2023).

[23] A. C. Ian Goodfellow Yoshua Bengio, *Deep Learning.* MIT Press, 2016, Publication Title: MIT Press, ISBN: 978-0-262-03561-3.

[24] T. Standley, A. Zamir, D. Chen, L. Guibas, J. Malik, and S. Savarese, "Which tasks should be learned together in multi-task learning?" In *37th International Conference on Machine Learning, ICML 2020*, vol. PartF168147-12, 2020.

[25] C. Feng, Y. Zhong, Y. Gao, M. R. Scott, and W. Huang, "TOOD: Task-aligned one-stage object detection," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, IEEE, Oct. 2021, pp. 3490–3499, ISBN: 978-1-66542-812-5. DOI: `10.1109/ICCV48922.2021.00349`.

[26] R. Tang *et al.*, "Task-balanced distillation for object detection," *Pattern Recognition*, vol. 137, p. 109 320, May 2023, ISSN: 00313203. DOI: `10.1016/j.patcog.2023.109320`.

[27] A. Kirillov, R. Girshick, K. He, and P. Dollar, "Panoptic feature pyramid networks," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2019, pp. 6392–6401, ISBN: 978-1-72813-293-8. DOI: `10.1109/CVPR.2019.00656`.

[28] Jason Zhang. "Deep understanding tesla FSD part 1: HydraNet." (Oct. 18, 2021), [Online]. Available: `https://saneryee-studio.medium.com/deep-understanding-tesla-fsd-part-1-hydranet-1b46106d57` (visited on 04/19/2023).

[29] Glenn Jocher, Sergiu Waxmann, Ayush Chaurasia, and Laughing. "Ultralytics." (), [Online]. Available: `https://docs.ultralytics.com/` (visited on 09/19/2023).

[30] Jacob Solawetz and Francesco. "What is YOLOv8? the ultimate guide." (), [Online]. Available: `https://blog.roboflow.com/whats-new-in-yolov8/` (visited on 09/19/2023).

[31] open-mmlab. "GitHub - open-mmlab/mmyolo: OpenMMLab YOLO series toolbox and benchmark." (), [Online]. Available: `https://github.com/open-mmlab/mmyolo/` (visited on 09/19/2023).

[32] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 7464–7475, Jun. 1, 2023, Publisher: IEEE Computer Society ISBN: 979-8-3503-0129-8. DOI: `10.1109/CVPR52729.2023.00721`. [Online]. Available: `https://ieeexplore.ieee.org/document/10204762/` (visited on 09/20/2023).

[33] B. Cheng, I. Misra, A. G. Schwing, A. Kirillov, and R. Girdhar, "Masked-attention mask transformer for universal image segmentation," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2022, pp. 1280–1289, ISBN: 978-1-66546-946-3. DOI: `10.1109/CVPR52688.2022.00135`.

[34] Z. Liu *et al.*, "Swin transformer: Hierarchical vision transformer using shifted windows," in *2021 IEEE/CVF International Conference on Computer Vision (ICCV)*, IEEE, Oct. 2021, pp. 9992–10 002, ISBN: 978-1-66542-812-5. DOI: `10.1109/ICCV48922.2021.00986`.

[35] T. Xiao, Y. Liu, B. Zhou, Y. Jiang, and J. Sun, "Unified perceptual parsing for scene understanding," in 2018, pp. 432–448. DOI: `10.1007/978-3-030-01228-1_26`.

[36] Z. Liu, H. Mao, C.-Y. Wu, C. Feichtenhofer, T. Darrell, and S. Xie, "A ConvNet for the 2020s," in *2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, IEEE, Jun. 2022, pp. 11 966–11 976, ISBN: 978-1-66546-946-3. DOI: 10.1109/CVPR52688.2022.01167.

[37] E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez, and P. Luo, "SegFormer: Simple and efficient design for semantic segmentation with transformers," *Advances in Neural Information Processing Systems*, vol. 34, pp. 12 077–12 090, Dec. 6, 2021. (visited on 09/21/2023).

[38] T. Zheng *et al.*, "RESA: Recurrent feature-shift aggregator for lane detection," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 4, pp. 3547–3554, May 18, 2021, Publisher: Association for the Advancement of Artificial Intelligence ISBN: 9781713835974, ISSN: 2374-3468. DOI: 10.1609/AAAI.V35I4.16469. arXiv: 2008.13719. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/16469 (visited on 09/18/2023).

[39] K. Zhou and R. Zhou, "End-to-end lane detection with one-to-several transformer," May 1, 2023. arXiv: 2305.00675. [Online]. Available: https://arxiv.org/abs/2305.00675v4 (visited on 09/05/2023).

[40] Z. Qin, P. Zhang, and X. Li, "Ultra fast deep lane detection with hybrid anchor driven ordinal classification," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2022, Publisher: IEEE Computer Society, ISSN: 19393539. DOI: 10.1109/TPAMI.2022.3182097. arXiv: 2206.07389. (visited on 09/05/2023).

[41] D. Wu *et al.*, "YOLOP: You only look once for panoptic driving perception," *Machine Intelligence Research*, vol. 19, no. 6, pp. 550–562, Dec. 7, 2022, ISSN: 2731-538X. DOI: 10.1007/s11633-022-1339-y.

[42] C. Han, Q. Zhao, S. Zhang, Y. Chen, Z. Zhang, and J. Yuan, "YOLOPv2: Better, faster, stronger for panoptic driving perception," Aug. 24, 2022. arXiv: 2208.11434. [Online]. Available: https://arxiv.org/abs/2208.11434v1 (visited on 09/24/2023).

[43] Q. H. Che, D. P. Nguyen, M. Q. Pham, and D. K. Lam, "TwinLiteNet: An efficient and lightweight model for driveable area and lane segmentation in self-driving cars," Jul. 20, 2023. arXiv: 2307.10705. [Online]. Available: https://arxiv.org/abs/2307.10705v4 (visited on 09/24/2023).

[44] F. Yu *et al.*, "BDD100k: A diverse driving dataset for heterogeneous multitask learning," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 2633–2642, 2020, Publisher: IEEE Computer Society, ISSN: 10636919. DOI: 10.1109/CVPR42600.2020.00271. arXiv: 1805.04687. (visited on 09/21/2023).

[45] "BDD100k documentation." (), [Online]. Available: https://doc.bdd100k.com/ (visited on 10/09/2023).

[46] A. Garcia-Garcia, S. Orts-Escolano, S. Oprea, V. Villena-Martinez, P. Martinez-Gonzalez, and J. Garcia-Rodriguez, "A survey on deep learning techniques for image and video semantic segmentation," *Applied Soft Computing*, vol. 70, pp. 41–65, Sep. 1, 2018, Publisher: Elsevier, ISSN: 1568-4946. DOI: 10.1016/J.ASOC.2018.05.018. (visited on 10/10/2023).

[47] "Jetson xavier series — NVIDIA." (), [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/ (visited on 10/20/2023).

[48] Nvidia. "Embedded systems developer kits & modules from NVIDIA jetson." (), [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/ (visited on 09/05/2023).

[49] Pedro Azevedo, "Deteção e seguimento de alvos múltiplos a bordo do ATLASCAR2 com deep learning," 2022, Place: Aveiro Publisher: Universidade de Aveiro. [Online]. Available: https://ria.ua.pt/handle/10773/34537 (visited on 10/20/2023).

[50] S. C. Pombinho, "Integrated software architecture in ATLASCAR2," Jul. 18, 2022. [Online]. Available: https://ria.ua.pt/handle/10773/34924 (visited on 10/20/2023).

[51] "The ultimate guide to deep learning frameworks." (), [Online]. Available: https://www.arkasoftwares.com/blog/guide-to-deep-learning-frameworks/ (visited on 10/20/2023).

[52] "Papers with code : Trends — papers with code." (), [Online]. Available: https://paperswithcode.com/trends (visited on 09/05/2023).

[53] Nvidia. "JetPack SDK — NVIDIA developer." (), [Online]. Available: https://developer.nvidia.com/embedded/jetpack (visited on 09/05/2023).

[54] Nvidia. "DeepStream SDK — NVIDIA developer." (), [Online]. Available: https://developer.nvidia.com/deepstream-sdk (visited on 09/05/2023).

[55] A. Kulkarni and A. Bhide. "GitHub - NVIDIA-AI-IOT/ros2_deepstream: ROS 2 package for NVIDIA DeepStream applications on jetson platforms." (), [Online]. Available: https://github.com/NVIDIA-AI-IOT/ros2_deepstream (visited on 09/05/2023).

[56] R. Paliwal. "GitHub - NVIDIA-AI-IOT/deepstream_python_apps: DeepStream SDK python bindings and sample applications." (), [Online]. Available: https://github.com/NVIDIA-AI-IOT/deepstream_python_apps/ (visited on 09/05/2023).

[57] M. Luciano. "GitHub - marcoslucianops/DeepStream-yolo: NVIDIA DeepStream SDK 6.3 / 6.2 / 6.1.1 / 6.1 / 6.0.1 / 6.0 / 5.1 implementation for YOLO models." (), [Online]. Available: https://github.com/marcoslucianops/DeepStream-Yolo (visited on 09/05/2023).

[58] D. Franklin. "GitHub - dusty-nv/jetson-inference: Hello AI world guide to deploying deep-learning inference networks and deep vision primitives with TensorRT and NVIDIA jetson." (), [Online]. Available: https://github.com/dusty-nv/jetson-inference (visited on 09/05/2023).

[59] D. Franklin. "GitHub - dusty-nv/ros_deep_learning: Deep learning inference nodes for ROS / ROS2 with support for NVIDIA jetson and TensorRT." (), [Online]. Available: https://github.com/dusty-nv/ros_deep_learning (visited on 09/05/2023).

[60] Nvidia. "Isaac software development kit — NVIDIA developer." (), [Online]. Available: https://developer.nvidia.com/isaac-sdk (visited on 09/05/2023).

[61] Nvidia. “Polygraphy.” (), [Online]. Available: https://docs.nvidia.com/deeplearning/tensorrt/polygraphy (visited on 10/30/2023).

[62] OpenMMLab. “MMCV.” (), [Online]. Available: https://mmcv.readthedocs.io/ (visited on 10/30/2023).