

# POLITECNICO DI TORINO

Master's Degree in ICT for Smart Societies



Master's Degree Thesis

## Traffic sign recognition algorithm: a deep comparison between Yolov5 and SSD Mobilenet

**SUPERVISORS**

Prof. Stefano Malan

Dott. Davide Faverato

**CANDIDATE**

Paola Migneco

April 2024



# Summary

The current personal transportation scenario presents one of the most significant and complex challenges of our time: making the driving experience completely effortless. Autonomous driving, with its attributes of safety, mobility and sustainability, stands as a milestone on this ambitious path. In this perspective, the importance of selected algorithms plays a crucial role, enabling vehicles to make immediate and accurate decisions and opening the way to a future of innovative transportation. The constant search for increasingly accurate models is one of the most significant goals and challenges in the automotive industry. Road safety and the efficiency of autonomous driving depend largely on the ability of these algorithms to learn and adapt to situations in real time.

The research presented here focuses on the comparison of two models in the context of traffic signal detection through object detection: SSD MobileNet and YoloV5. The goal is to determine the optimal model in the context of autonomous driving, and the models will be tested on a specific hardware, the NVIDIA Jetson Nano, a compact yet high-performance computer characterized by a low power consumption. This project is part of a larger work aimed at the realization of autonomous driving of a ROSMASTERX3, that is an educational robot based on the robot operating system compatible with Jetson NANO, enabling performance simulation in the context of the overall project. In the implementation of this work, I had the opportunity to benefit from the valuable technical support provided by the MCA company. Their expertise and assistance were critical to the success of the project. The MobileNet SSD model uses convolutional neural networks for object detection and classification in images, characterizing itself as lightweight and fast. It leverages the MobileNet architecture for feature extraction and the Single Shot Detector algorithm for real-time detection, making it ideal for mobile applications and contexts where fast and efficient detection is required. The YOLO (You Only Look Once) object detection algorithm represents a significant advance in computer vision. Based on a unified convolutional neural network, it is capable of predicting bounding box positions and object classes in real time on complete images, highlighting itself for its speed and generalization capability. Version 5, considered in this project, presents an innovative approach to detection without the use of predefined bounding

boxes, providing a balance between speed and accuracy.

The project is developed through several phases, framed in the context of vehicle automation, with the ultimate goal of realizing autonomous driving on the road. The initial phase involves the creation of a specially designed training dataset for traffic signal recognition, with the thoughtful selection of a representative set of signals to compose the training set. During this process, it was necessary to study the dataset formats required by each model, namely the VOC format for the SSD Mobilenet model and the .txt format for YOLOv5. The dataset includes four basic classes: "Stop Sign", "Speed Limit", "Crosswalk" and "Traffic Light".

Next, both models were trained based on the chosen dataset. During this phase, the parameters of the neural networks are studied to maximize performance, taking into account limitations related to the available computing power, especially the processing capacity. The results obtained are compared and evaluated through simulations and empirical tests.

The entire process is implemented on an NVIDIA Jetson Nano board, equipped with a high-definition ASTRA camera to capture detailed images. This configuration enables accurate analysis of neural network performance in traffic signal detection. The metrics used for comparison during training and testing were mAP (Mean Average Precision), and FPS (Frames Per Second). MAP measure the performance of a model that focuses on object detection tasks and information retrieval on images. The mAP uses the ground-truth bounding box, compares it to the detected box and returns a score. The higher the mAP score, the more accurate the model detects and makes correct predictions. Indeed, FPS measures the performance and frame capture rate in a video.

The models, SSD MobileNet and YOLOv5, exhibited different behaviors, both in terms of performance and model portability. The initial comparison was conducted through training on Colab, a platform developed by Google that enables the use of resources running directly in the cloud. YOLOv5 clearly outperformed SSD MobileNet, especially in some classes and in terms of training time; on Jetson Nano the situation was more complex. The challenge of porting complex models to hardware with limited resources required several solutions, such as expanding RAM with a swap or having to compromise on various fronts. The MobileNet SSD model was trained, again, directly on the Jetson. Indeed, despite all efforts, the conversion to a format acceptable to Jetson itself was impractical. This resulted in limitations related to the size and heterogeneity of the dataset and the parameters adapted to Jetson RAM restrictions.

The results obtained are good, with excellent FPS reaching about 30 FPS, and a mAP that is around 95%. However, the SSD MobileNet model is incomplete, as it was trained only on "stop" signals of the four initially selected. A choice forced by the restrictions due to the limitations of the Jetson Nano. The YOLOv5 model, in contrast, was easily transferred to Jetson thanks to Ultralytics support. For this

reason, training the model on Jetson, as had been planned at the outset, was not necessary, and it was possible to proceed directly through testing. Although FPS performance might initially have seemed concerning due to the heaviness of the model on low-capacity hardware. It was necessary a conversion to .onnx format with FP16 and FP32 configurations that greatly improved the performances. The evaluation of these two formats aimed to assess the trade-off between precision and FPS. In terms of accuracy, both setups exhibited outstanding signal recognition. A decrease in image resolution was noted, aligning with predictions, yet the detection capability remained uncompromised. Then, YOLOv5 model performed extremely promisingly, with a mAP of 98%. Conversion to Jetson Nano using the DeepStream library achieved nearly 14 FPS with the FP16 configuration. Although the FPSs of the YOLOv5 model are significantly lower than those of SSD MobileNet, they are still sufficient contextualizing these results to a domain in which the rover to which the object detection model will be integrated does not reach speeds higher than 30 km/h.

Tests with the ASTRA camera attached to Jetson Nano explored various situations, such as extreme distances and obstacles partially covering the signals, to evaluate the models performance in realistic scenarios. Both models responded excellently to the empirical tests, confirming the validity of what was developed.

Finally, the model selected for future projects is YOLOv5 with *batch - size* = 32 and *epochs* = 300, that reported the best results, converted in FP16 format to .onnx model. Its consistency with the expected goals and completeness of the work makes it an ideal choice. Confidence curve and precision-recall analysis confirmed excellent results, although the dataset has some challenges in the "crosswalk" class, which, in fact, is the weakness in the dataset. This model will form the basis for future projects, integrating the newly presented survey into a prototype rover capable of moving autonomously, simulating autonomous driving. Its adaptability and high degree of accuracy provide a solid basis for further developments and practical implementations in the field of autonomous vehicles and road automation.

# Acknowledgements

I would like to express my deep gratitude to the MCA company for providing me with this valuable opportunity for professional growth. In particular, I would like to thank my corelator, Davide Faverato, for believing in me from the beginning and for his constant support and availability. I would also like to acknowledge Marta Cattaneo and Leonardo Forese for their generous assistance and patience throughout my journey.

Special thanks go to my thesis advisor, Stefano Malan, for his dedication, expertise, and for always being available to provide me with valuable guidance and feedback. His thoroughness and commitment definitely contributed to the success of this research paper.



# Table of Contents

<b>List of Tables</b>	IX
<b>List of Figures</b>	X
<b>Acronyms</b>	XIII
<b>1 Introduction</b>	1
1.1 About MCA . . . . .	1
1.2 Brief Overview . . . . .	1
1.3 Objectives . . . . .	2
1.4 Thesis Outline . . . . .	3
<b>2 State of Art</b>	4
<b>3 Neural networks for object detection</b>	10
3.1 Machine Learning and Neural Networks . . . . .	10
3.2 Convolutional Neural Network . . . . .	11
3.3 YOLO . . . . .	12
3.3.1 YOLO General Overview . . . . .	12
3.3.2 YOLOv5 . . . . .	14
3.4 SSD Mobilenet . . . . .	16
3.5 SSDmobilenetv2 vs YOLOv5 . . . . .	18
<b>4 Hardware and software analysed tools</b>	20
4.1 Environment . . . . .	20
4.1.1 NVIDIA® Jetson Nano™ . . . . .	20
4.1.2 Google Colab . . . . .	22
4.2 Dataset . . . . .	22
4.3 Training model . . . . .	25
4.3.1 SSD MobileNet . . . . .	25
4.3.2 YOLOv5 . . . . .	34



4.4	Metrics of the final chosen model . . . . .	45
<b>5</b>	<b>Test results on the analysed models</b>	<b>50</b>
5.1	SSD MobileNet . . . . .	50
5.1.1	Jetson-inference folder . . . . .	50
5.1.2	SSD MobileNet with COCO128 . . . . .	53
5.2	YOLOv5 . . . . .	57
<b>6</b>	<b>Conclusions</b>	<b>62</b>
6.1	Overview of the analysed models . . . . .	62
6.2	Improvements and future works . . . . .	63
	<b>Bibliography</b>	<b>65</b>

# List of Tables

4.1	Command-Line for training . . . . .	30
4.2	Command-Line for training . . . . .	34
4.3	Training results of YOLOv5 model in Google Colab . . . . .	35
4.4	AP YOLOv5 model <i>epochs</i> = 300 <i>batch - size</i> = 32 . . . . .	36
4.5	AP YOLOv5 model <i>epochs</i> = 300 <i>batch - size</i> = 16 . . . . .	36
4.6	Command-Line for detection . . . . .	39
4.7	YOLOv5 FPS on Jetson Nano . . . . .	39
4.8	Configuration Group in <code>deepstream_app_config.txt</code> . . . . .	41
4.9	Configuration [source] . . . . .	42
4.10	YOLOv5.onnx FPS on Jetson Nano with DeepStream . . . . .	42
4.11	YOLOv5.onnx FPS on Jetson Nano with DeepStream . . . . .	45
5.1	MAP results according to different batch-size and epochs . . . . .	51
5.2	Confidence values variation according to the distance between road sign and camera . . . . .	58

# List of Figures

3.1	General Architecture of a Neural Network . . . . .	12
3.2	YOLO Architecture . . . . .	13
3.3	YOLOv5 versions . . . . .	16
3.4	SSD architecture . . . . .	17
3.5	MobileNet-V1 vs MobileNet-V2 architecture . . . . .	19
4.1	NVIDIA® Jetson Nano™ . . . . .	21
4.2	Technical specifications NVIDIA® Jetson Nano™ . . . . .	21
4.3	Label format xml . . . . .	23
4.4	Label format txt . . . . .	23
4.5	Label format txt . . . . .	24
4.6	Label format yaml . . . . .	24
4.7	SSD Mobile Net detection of test images . . . . .	27
4.8	Docker container start . . . . .	29
4.9	Training command . . . . .	29
4.10	OutOfMemory error . . . . .	30
4.11	Conversion to ONNX model . . . . .	31
4.12	Detectnet command . . . . .	32
4.13	Class Object Detector . . . . .	33
4.14	Yolo model comparison . . . . .	35
4.15	Tests performed with $epochs = 300$ and $batch - size = 32$ . . . . .	37
4.16	DeepStream Overview grapg architecture . . . . .	40
4.17	deepstream_app_config.txt file . . . . .	43
4.18	config_infer_primary_yoloV5.txt file for FP16 vs FP32 configuration . . . . .	44
4.19	Detection of a signal FP16 vs FP32 configuration . . . . .	44
4.20	MAP0.5 and MAP0.5:0.95 of YOLOv5 model with $epochs = 300$ and $batch - size = 32$ . . . . .	46
4.21	PR curve YOLOv5 model with $epochs = 300$ and $batch - size = 32$ . . . . .	47
4.22	F1 curve YOLOv5 model with $epochs = 300$ and $batch - size = 32$ . . . . .	48
4.23	Train Loss functions YOLOv5 model with $epochs = 300$ and $batch -$ $size = 32$ . . . . .	49

5.1	Dataloader Worker killed . . . . .	50
5.2	<i>batch - size</i> = 2 and <i>epochs</i> = 30 . . . . .	52
5.3	<i>batch - size</i> = 4 and <i>epochs</i> = 10 . . . . .	52
5.4	<i>batch - size</i> = 8 and <i>epochs</i> = 10 . . . . .	53
5.5	Stop sign detection at the expense of other traffic signals . . . . .	53
5.6	Street Sign missed detection . . . . .	54
5.7	Detection stop sign closely . . . . .	55
5.8	Detection stop sign from a distance . . . . .	55
5.9	Detection stop sign next to another traffic signal . . . . .	55
5.10	Stop signal detection with obstacles . . . . .	56
5.11	Traffic Light detection . . . . .	56
5.12	Traffic Light detection with other types of semaphores . . . . .	57
5.13	Traffic light and stop sign detection . . . . .	57
5.14	Tests performed with <i>epochs</i> = 300 and <i>batch - size</i> = 32 with signals through the camera . . . . .	58
5.15	Speed Limit signal at different distances . . . . .	59
5.16	Detection of Speed Limit signal with obstacles . . . . .	59
5.17	Detection of Stop and Speed Limit signals near a sign similar in shape and color . . . . .	60
5.18	Tests with different type of traffic lights . . . . .	60
5.19	Detection of all the signals all at once . . . . .	61



# Acronyms

**ADAS**

Advanced Driver Assistance Systems

**AI**

Artificial Intelligence

**AP**

Average Precision

**CNN**

Convolutional Neural Network

**CPU**

Central Processing Unit

**DNN**

Deep Neural Network

**FP**

Single-precision floating-point format

**FPS**

Frames Per Second

**FC**

Fully Connected

**GPU**

Graphics Processing Units

**IoU**

Intersection over Union

**MAP**

Mean Average Precision

**MLP**

Multilayer Perceptron

**ONNX**

Open Neural Network Exchange

**RAM**

Random Access Memory

**ROS**

Robot Operating System

**SDK**

Software Development Kits

**SSD**

Single Shot Detector

**YOLO**

You Only Look Once

# Chapter 1

## Introduction

### 1.1 About MCA

MCA is an international high-tech engineering and consulting company, provides support to its talents in large-scale projects for major industrial groups. It was founded in France but expanded its presence worldwide, eventually establishing in Turin in 2016. Today, it provides consultancy services to the Turin region in the fields of aerospace, biomedical, energy, industrial automation, and, notably, the automotive industry.

### 1.2 Brief Overview

In a society where smart technology permeates every aspect of our lives, autonomous driving stands as one of the most promising fields for the development of technologies capable of redefining our approach to driving. What may have once seemed like a futuristic vision is now an established reality, ready to take significant, but even more decisive steps.

Autonomous driving brings with it a number of tangible benefits to everyone's lives. For example, 90% of road accidents are caused by human error, while autonomous vehicles have the potential to significantly reduce the risk of accidents by eliminating the human factor. In addition, this technology could provide people with mobility disabilities with access to mobility, opening up new prospects for inclusion.

These innovations promise a future in which roads are safer, cities less congested and the environment better preserved, improving our quality of life overall.

When addressing the topic of autonomous driving, it is inevitable to consider the Six Levels of Vehicle Autonomy. Especially in Europe, we are at level 2 - Partially Automatic Driving - which involves advanced driver assistance systems (ADAS).



The integration of cutting-edge technologies, such as advanced sensors and machine learning, represents a real breakthrough in the field of autonomous driving. These innovations not only change the way we drive, but propel us from the current Level 2 to the upper layers of the Six Levels of Automated Driving. The importance of embracing these technologies and their symbiotic relationship cannot be overstated.

To progress from Level 2 to the higher levels of automation, it is critical to recognize the crucial role of advanced sensors and machine learning. Sensors such as LiDAR, radar and cameras act as the eyes and ears of autonomous vehicles, providing a real-time, data-rich view of their surroundings. This data is then processed and interpreted by machine learning algorithms, enabling vehicles to perceive, analyze and react to complex driving scenarios with a degree of sophistication that was once unthinkable.

Machine learning in this context assumes a fundamental but highly sensitive role, as decision making in driving situations can involve significant conflicts. The constant search for increasingly accurate and precise models represents one of the most significant goals and challenges in the automotive industry. Road safety and the efficiency of autonomous driving depend largely on the ability of these algorithms to learn and adapt to situations in real time.

## 1.3 Objectives

The main goal of this project is the development and testing of two neural networks through personalized training, with the aim of recognizing a variety of road signs. This purpose is within the scope of vehicle automation, with the ultimate goal of realizing autonomous driving on the road.

The first phase of the work involves creating a specially designed training model for traffic-sign recognition, carefully selecting a set of representative signals to compose the dataset.

Next, that training model will be tested on two well-known neural networks: SSD MobileNet and YOLOv5. At this stage, the parameters of the neural networks will be studied in order to maximize performance, considering the limitations imposed by the available computing power, particularly the processing capacity of the GPU. The results obtained will then be compared and evaluated by simulations.

The entire process will be implemented on an Nvidia Jetson Nano board, equipped with a high-definition ASTRA camera to capture detailed images. This configuration will allow accurate analysis of the performance of neural networks in the area of traffic sign recognition to be conducted.

## **1.4 Thesis Outline**

The key steps of the work carried out are as follows:

- Study of the code and project architecture.
- Exploration of documentation related to the development of the two models.
- Examination, selection, and creation of a dataset for training purposes.
- Deployment of the best model in both neural networks and comparison of the results.
- Final tests.

# Chapter 2

## State of Art

In an ever-evolving smart society that strives to simplify every facet of life, the paradigm of vehicle guidance demands a profound transformation, a transformation that, as of now, remains partially realized due to a myriad of diverse challenges. To enable vehicles to truly achieve full autonomy, a comprehensive examination of techniques capable of ensuring an entirely secure and dependable driving experience becomes imperative. Recent technological advances in cloud computing and the accessibility of cloud-based high-performance Graphics Processing Units (GPUs) have greatly accelerated the development of artificial intelligence algorithms.

As evidenced by the research conducted in Autonomous Car Driving Using Deep Learning [1], the successful implementation of an autonomous driving system hinges on several critical factors. These factors include the delivery of precise results crucial for safety, reliability and cost-effectiveness. To generate the initial dataset, it was initially planned to manually collect data while driving on highways, but this resulted in fuzzy and unclear data. As a result, an alternative was successfully evaluated using the CARLA driving simulator. This simulator allows various environments to be recreated and provides open digital resources such as urban models, buildings, vehicles and more. CARLA is based on Unreal Engine 4 (UE4) and is configurable for the use of agent sensors, although it is currently limited to RGB cameras and dummy sensors for depth and semantic segmentation. In this particular study, a U-Net Network Model was employed, utilizing a synthetic dataset. U-Net is a convolutional neural network mainly developed for biomedical image segmentation, but it does not currently achieve the level of reliability required for real-world applications. This architecture is designed for semantic segmentation and uses a contractive and an expansive path. The model, since it was trained on a synthetic dataset, is not directly applicable in the real world, especially for smaller objects such as lamps and traffic signals. Architectures with dilated convolutions are being considered to address this challenge. Currently, the model makes limited decisions, but the goal of the study is to expand them to include traffic light

and lane change detection. Additional investigations, such as the one undertaken by "Moving object detection in videos using principal components pursuit and convolutional neural networks" [2] delve into the latest applications of Convolutional Neural Networks (CNNs) in computer vision and object detection. In this context, the study introduces a novel approach to address the substantial computational cost associated with video processing. Specifically, Principal Component Pursuit (PCP) is introduced as a pre-processing step, enabling the segmentation of moving objects within videos. Tests were conducted on different platforms for image classification in videos. The use of an image processing algorithm improved performance on many videos, although some problems remain, such as loss of objects with little visibility. The use of reduced images has lowered classification times on all platforms, with improvements ranging from 2% to 25%. It is important to note that PCP time depends only on image size, not on content. This innovative strategy substantially reduces the computational burden and classification time, as it necessitates processing fewer regions.

Another compelling solution emerges from the research in the paper "Autonomous Driving Designed for Embedded Automotive Platforms" [3]. This work introduces an end-to-end deep neural network tailored explicitly for autonomous driving. In this context, a neural network was trained on data collected while a human driver was driving. This network learns to control the vehicle based only on camera images, without further human interaction, this method is known as "behavior cloning". The network predicts the steering angle, allowing the vehicle to drive autonomously in the simulator. The goal is to constantly keep the vehicle on the track. Data were acquired from three cameras during several manual driving sessions in order to train the training model. In order to obtain good quality samples, the vehicle in manual mode was driven following the same protocol as for autonomous vehicle driving. Due to the limited size of the dataset, data augmentation techniques such as image mirroring and steering angle inversion were implemented. These techniques expanded the amount of data available for training and improved the ability to handle turns. In addition, a system of three cameras was used for data acquisition, each with a slightly different perspective. This choice resulted in an increase in the amount of data available and improved the vehicle's ability to maintain its trajectory during any lateral deviations. Steering angle measurements related to the side cameras were corrected to ensure proper correlation between the images and the driving data. Finally, an image cropping process was used in order to eliminate portions not relevant to autonomous driving, such as the sky, vegetation and the vehicle hood. The core idea revolves around the development of a neural network capable of seamless operation even within the constraints of limited hardware resources. The proposed J-Net deep neural network was compared with AlexNet and PilotNet, which were reimplemented in order to conduct an objective performance evaluation of the new design. The models of all three network

architectures were implemented, trained with the same dataset, and the trained models were used for processing in a simulator for autonomous driving. The results were compared in terms of performance, as the ability to drive successfully on the representative track, and in terms of network complexity, as the number of trainable parameters and size of the trained model. This endeavor aims to construct a lightweight Deep Neural Network (DNN) that excels at the task of autonomous driving on a representative track.

Furthermore, the field of autonomous driving embraces the powerful algorithm known as YOLOv4, as elucidated in the paper Object Detection for Autonomous Driving using YOLO algorithm [4]. This class of object detection algorithms play a key role in object identification. Indeed, they can contribute significantly to improving road safety by reducing accidents due to human error. This algorithm is adeptly employed to custom-train models for object detection, with data sourced from the extensive open images dataset via the OIv4 toolkit, which boasts a staggering repository of over 6 million images. Subsequent fine-tuning on a customized dataset has yielded results that are nothing short of impressive. The model that has been trained demonstrates the ability to identify various objects on the road by delineating them with a bounding box and providing a category classification. However, some reports of false positives and false negatives persist and require further reduction, as they could lead to serious consequences in practical implementation. The occurrence of false positives, in particular, can cause potentially fatal accidents, so significant resources are needed to mitigate this risk through extensive image collection and thorough training.

The study "Impact of Image Corruptions on the Reliability of Traffic Sign Recognition Using Machine Learning Technique" [5] focuses on the recognizability of traffic signs by machine learning algorithms under conditions of image corruption. Traffic signal recognition, indeed, is a complex process that requires considerable accuracy and reliability. This task involves several significant challenges. First, lighting conditions on roads can vary greatly, ranging from direct sunlight to shadows, artificial lighting, and darkness at night. This variability affects the visibility of traffic signals, making it difficult to identify them accurately. In addition, traffic signals may be partially hidden by trees, vehicles or other objects, which can hinder the detection and classification process. The perspective and scale of signs vary with their distance from the camera, which can make determining their apparent shape and size more complex. Another challenge concerns variations in the geometry and shape of the signals themselves. These may have different shapes, such as circles, triangles or rectangles, which requires considerable flexibility on the part of recognition systems. Images captured by sensors may also contain noise or be subject to distortions, such as blurring or perspective distortions, which reduce image quality and complicate signal recognition. Therefore, it can be understood that traffic sign recognition is crucial for the implementation of autonomous driving

and driver assistance systems. However, the challenges go beyond simple image corruption and include variations in weather conditions, lighting, distortions due to vehicle motion, and wear and tear on traffic signals. Many algorithms have been developed to address image correction, but it is important to assess how these corruptions affect the overall accuracy of traffic sign recognition. The used dataset includes a total of 62 categories of traffic signs, each represented by images and numerical labels. To standardize the format of the images and make them suitable for the training process using neural networks, all images were resized. The model was trained using the entire training dataset in order to minimize the overall classification error rate. This training was carried out through the use of the TensorFlow toolkit. Next, the trained model was applied to the test dataset and the classification error rates for training and testing were calculated, reporting the percentage of correct prediction as the result. In order to evaluate the impact of image corruption on the traffic sign recognition model, four different corruption methodologies were applied by manipulating some pixels of the images. This study examined how these may affect the ability of the traffic sign recognition model and discussed the impact of such corruptions on machine learning techniques, with the goal of aiding model adaptability in future contexts characterized by image corruptions. In the case of uncorrupted images, 65.6% of the images were classified correctly. The analysis showed that as the number of corrupted pixels increased, different types of corruption produced different results. Removal of a continuous section of pixels was more damaging than removal of scattered pixels, particularly as the fraction of pixels removed increased. This study provides an interesting and essential perspective on improving the correctness of traffic sign recognition by examining how each variable in image corruption can affect the correctness of its detection.

Another contribution comes from the work "Pedestrian Detection with YOLOv5 in an Autonomous Driving Scenario" [6] underscores the applicability of the YOLOv5 algorithm within the autonomous driving domain. The study focuses on pedestrian detection, an important and challenging problem in autonomous driving systems and a classic topic in the field of computer vision. Vision-based pedestrian detection must address problems such as pedestrian size, posture, weather, illumination, occlusion and truncation. Systems based on deep neural networks have been widely used in image processing and provide high accuracy. However, to be used on autonomous vehicles, the algorithms must combine high accuracy, real-time processing efficiency and good generalization ability. Here, two datasets, PASCAL VOC2012 and KITTI, were used for model training and evaluation. Several experiments were performed to select the best hyperparameters for training the model. In addition, the accuracy of pedestrian detection on test images was evaluated, showing that the model provides accurate results. The YOLOv5 model, trained with 75 epochs, proved to be the best option for pedestrian detection. It

achieved high accuracy, offering real-time processing speed. The test images were taken from the KITTI dataset, which has a different pedestrian distribution than the PASCAL VOC2012 dataset. This shows that the YOLOv5 model has good generalization ability and could be successfully applied in new environments, such as autonomous vehicles on the road. Thus, this study showed that the YOLOv5 model is highly promising for pedestrian detection in autonomous driving scenarios.

"Real-time Traffic Light Detection and Recognition based on Deep RetinaNet for Self Driving Cars" [7] focuses on the use of autonomous vehicles by concentrating on some real-time problems related to autonomous driving, such as recognition of traffic lights, traffic signals and pedestrians, are successfully addressed by state-of-the-art algorithms based on deep learning. For real-time traffic light detection and recognition, the paper proposes the use of RetinaNet, a deep neural network architecture implemented with Keras and TensorFlow on Google Colaboratory's cloud platform. This model was trained and evaluated on the Bosch Small Traffic Light dataset, containing traffic light images at four different classes and high resolution. In particular, it contains traffic light images labeled as "off," "green," "red," and "yellow". The RGB image captured by the on-board camera system is used as input to the RetinaNet network, which returns the location of the traffic lights and their class. In this case no preliminary image processing is required. The training process is described with specific parameters, including the stochastic optimization used (Adam), the number of epochs, the learning rate, and the types of losses used. The RetinaNet model achieved a weighted Mean Average Precision (mAP) of 0.54. This model shows a significant improvement in accuracy without compromising execution time. The proposed system allows traffic lights to be detected and their status classified in RGB images using a single RetinaNet network. The system is capable of real-time operation with improved accuracy, which could play a key role in real-time control of self-driving cars by achieving good detection and classification accuracy compared with other deep learning methods for real-time operations. The performance of the system can be further improved by training the network with an external dataset or creating additional image samples by augmenting the existing dataset.

All of these studies exemplify the critical role played by deep learning techniques and object detection algorithms in the journey toward fully autonomous vehicles, promising safer, more efficient and more accessible transportation systems for the future.

The article "The Future of Autonomous Driving" [8] provides a realistic picture of the prospects for autonomous driving in the near future. It points out that despite rising expectations, it will still be many years before we see fully autonomous cars driving without the supervision of a human driver. This is due to the need to bridge the transition between current assisted Level 1 and semi-autonomous Level 2 driving technologies and Level 3 highly automated driving. The latter

should already be a reality on some European roads, but unresolved legislative challenges are slowing this transition. Europe does not yet have a common set of regulations for Level 3 or higher automated driving, leaving member states to issue specific laws and regulations and giving automakers responsibility for the details of implementation. Countries at the forefront of this issue, including Sweden, Germany, France and Belgium, allow testing of autonomous vehicles on roads open to the public, while requiring the presence of a driver on board. There are also signs of progress in Italy, but the lack of operational autonomous vehicles is still a significant obstacle. Many automakers are setting ambitious short- and medium-term goals for the introduction of Level 3 autonomous driving.



## Chapter 3

# Neural networks for object detection

### 3.1 Machine Learning and Neural Networks

Machine learning represents a sub-discipline of Artificial Intelligence (AI) that has attracted considerable attention in recent years due to its transformative capabilities in a variety of fields and applications.

It is defined as a set of methods that enable the automatic identification of patterns in data and the use of those patterns to predict future events or to make other forms of decisions under conditions of uncertainty.

Currently, the two main types of ML approaches are: the predictive or supervised approach and the descriptive or unsupervised approach. The distinction between these two types is defined by the way each algorithm learns from data to make predictions.

The first type of algorithm is based on learning a mapping from inputs to outputs, given a set of labeled input-output pairs, which is called training set. This approach proves to be particularly valuable when there is a need to train a model for making predictions on unfamiliar data, relying on the information furnished by the training dataset.

In the second type of algorithm, instead, there are only inputs and the goal is to identify "interesting patterns" in the data. This poses a more complex challenge, as there is a lack of specific metrics to refer to. However, unsupervised learning can result valuable when it comes to exploring and discovering hidden information in the data, even if it is not immediately obvious what we are trying to find.

Therefore, ML automates the creation of analytical models, exploiting methods based on neural networks, statistical models and operations research to trace hidden information in data. A neural network draws inspiration from the workings of the

human brain, constituting a processing system composed of interconnected, neuron-like units. These units process information in response to external signals and transmit related information between different units, thus creating a sophisticated learning structure. [9]

Neural networks consist of interconnected nodes, or artificial neurons, organized in layers. Each neural network has a few components in common: [10]

- **Input** - Data put into the model for learning and training purposes.
- **Weight** - Real values that help at organize the variables by importance and impact of contribution.
- **Transfer function** - Also known as Summation Function, it has the function of tying weights and inputs together.
- **Activation function** - introduces non-linearity in the model.
- **Bias** - Bias shifts the value given by the activation function.

The strength of neural networks lies in their ability to process massive amounts of data, recognize intricate patterns, and make predictions or decisions based on this information. This ability makes them particularly well suited for a wide range of tasks, ranging from image and speech recognition to natural language processing and autonomous driving.

It is precisely this versatility that characterizes neural networks, as they are able to adapt and be used in heterogeneous and diverse applications. As a result, different types of neural networks are available, each designed to address a specific task or application domain.

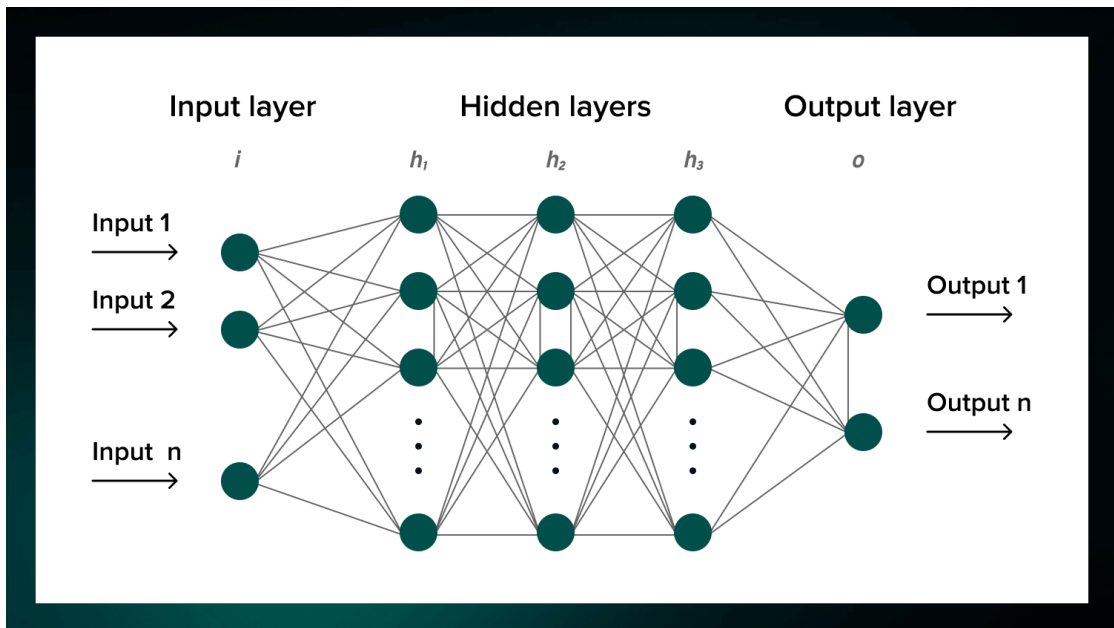
Neurons are usually organized into independent layers. One example of neural networks are feed-forward networks, where the data moves from the input layer through a set of hidden layers only in one direction.

Every node in the system is connected to some nodes in the previous layer and in the next layer. The node receives information from the layer beneath it, does something with it, and sends information to the next layer.

Every incoming connection is assigned a weight. It is a number that the node multiplies the input by when it receives data from a different node. [11]

## 3.2 Convolutional Neural Network

Prominent among the various categories of neural networks is the Convolutional Neural Network (CNN), which is particularly relevant when it comes to analyzing text or images. This type of network represents a specific variant of the Multilayer Perceptron (MLP), characterized by fully connected neurons with a type of nonlinear



**Figure 3.1:** General Architecture of a Neural Network

activation function. A distinctive aspect of the CNN is its organization into at least three layers, with the ability to distinguish data that are not linearly separable. A general example architecture can be seen in Fig.3.1.

In detail, within the CNN, hidden units are endowed with local receptive fields, similar to what is observed in the primary visual cortex. In addition, weights are shared or connected between different portions of the image in order to reduce the number of parameters. Intuitively, this sharing of spatial parameters allows useful features identified in one part of the image to be reused throughout the model without having to learn them independently. [9] [11]

## 3.3 YOLO

### 3.3.1 YOLO General Overview

The YOLO object detection algorithm, which stands for "You Only Look Once," represents a remarkable advancement in the field of computer vision. Its essence is summarized by a unified convolutional neural network that can simultaneously predict both the positions of bounding boxes - rectangles surrounding objects of interest - and the probabilities of object classes in a clear and straightforward manner. The architecture of this model can be observed in Fig.3.2. A distinctive aspect of YOLO is its training on complete images, avoiding the need for pre-processing steps

or extraction of regions of interest. This directly optimizes detection performance, contributing to highly accurate object detection and recognition. The strength of the YOLO algorithm lies in the small size of the model and its remarkable computational speed. YOLO follows a "one-stage" approach, unifying object location, identifying where they are in the image and object recognition, attributing specific classes to them, in a single end-to-end process, minimizing computational complexity. Thus, YOLO high speed enables real-time detection of objects on both images and video. In addition, YOLO is distinguished by its use of the global image in the detection process, capturing global information and reducing the error in identifying the background as an object. An additional strength of YOLO lies in its generalization capability, as it can learn highly generalized features that can be transferred to different domains [12] [13].

YOLO can be compared, in part, to a traditional CNN, using convolutional layers and max pooling for feature extraction from input images. However, there are some significant differences that distinguish YOLO. The architecture consists of as many as 24 convolutional layers. These layers work hierarchically to identify edges, shapes, textures and other relevant features in the images. This deep layering is crucial to YOLO ability to perform detection effectively. At the end of the process, there are two Fully Connected (FC) layers that play a key role in sensing. These layers are responsible for predicting the coordinates of bounding boxes surrounding objects and the probabilities of object classes. A distinctive feature of the architecture is the use of convolution kernels of size 1x1 alternating with kernels of 3x3. This approach is aimed at extracting more abstract and complex features from the input image, thus contributing to its ability to identify objects accurately [12] [13].

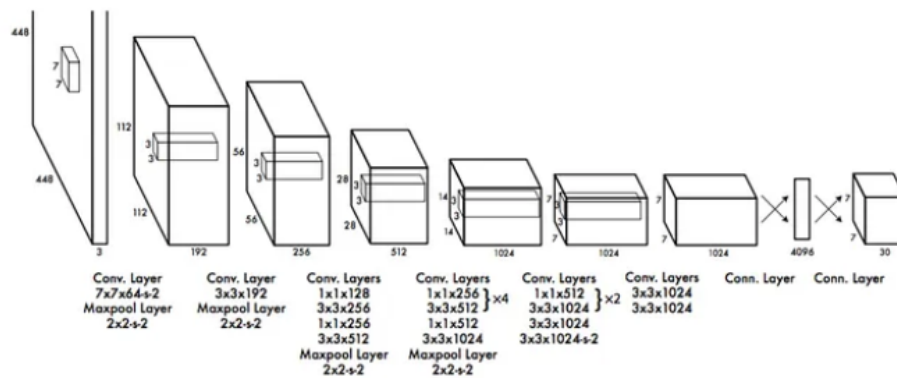


Figure 3.2: YOLO Architecture

Several versions of YOLO have been developed over the years, characterized by significant advances that have affected the accuracy, speed and versatility of the

algorithm. The most significant steps will be briefly explained below: [14]

- **YOLOv1** marked the introduction of a new paradigm in object detection, allowing the detection of objects in a single pass through a neural network. This innovation represented a breakthrough, but the architecture had some limitations, including the difficulty in detecting objects close together and the difficulty in adapting to objects of very different sizes. In addition, YOLOv1 did not impose constraints on bounding box predictions, resulting in an unstable model, especially in the early iterations of learning, as bounding box predictions could extend well beyond the edge of the reference grid cell.
- **YOLOv2** addressed the initial limitations and made significant improvements. This version introduced the concept of "bounding box priors" to handle objects of different sizes and limited the location of anchor boxes with respect to the relevant cell. In addition, it was trained on a combined dataset that included even more object classes, remaining a real-time object detection system.
- **YOLOv3** further refined object detection by adopting a deeper architecture and employing different sizes of bounding boxes to capture objects of various sizes. This version introduced multi-scale predictions and three different sizes of detection kernels. Moreover, the feature extraction network, Darknet-19, was revised through the use of additional convolutional layers and residual layers, bringing the total to 53 layers. This version of YOLO represented a significant performance improvement.
- **YOLOv4** was characterized by its increased speed and accuracy, introducing a set of optimizations, including data augmentation through the Mosaic method, a new sensing head without anchor boxes, and a new loss function. This version innovated through the use of Weighted-Residual-Connections (WRC), Cross-Stage-Partial-connections (CSP), Cross mini-Batch Normalization (CmBN), Self-adversarial-training (SAT), Mish-activation, Mosaic data augmentation, DropBlock regularization, and CIoU loss.[15]
- **YOLOv5** is known for its lightness and speed. This version introduced a lean architecture designed to ensure efficient real-time detection. In addition, YOLO v5 supports training on different image sizes and offers a number of advanced features, including hyperparameter optimization, built-in experimental tracking, and automatic export to popular export formats.

### 3.3.2 YOLOv5

YOLO version 5 represents a major achievement in the field of computer vision and machine learning, and is known for its outstanding object detection capabilities. In

this section, we will delve into the analysis of YOLOv5, as this is one of the main areas of study in this research work.

The salient features of YOLOv5 include:

- Easy installation: YOLOv5 requires only the installation of "torch" and some lightweight Python libraries. This ease of installation makes it very accessible to developers who wish to integrate machine vision technologies into their applications.
- Fast training: YOLOv5 models train extremely fast, which helps reduce experimental costs during model development.
- Working inference ports: YOLOv5 offers the ability to perform inference on single images, sets of images, video streams or via webcam ports.
- Intuitive structure of the data file system: the structure of data file folders in YOLOv5 is intuitive and easy to navigate during development, simplifying the developer's work.
- Easy implementation on mobile devices: YOLOv5 can be easily converted from PyTorch weights to ONNX weights and then to CoreML for iOS, making it suitable for implementation on mobile devices.
- "Anchor-free Split Ultralytics Head" adoption: represents a departure from traditional object detection models, which rely on predefined bounding boxes to predict the position of objects. This type of approach makes sensing more flexible and adaptable, improving performance in a variety of scenarios.

YOLOv5 also challenges the typical trade-off between speed and accuracy. It offers a calibrated balance that ensures real-time detections without compromising accuracy. This feature is especially useful in applications that require rapid responses, such as autonomous vehicles, robotics and real-time video analysis.

In addition, YOLOv5 provides a variety of pre-trained models, each of which is optimized for specific tasks such as inference, validation or training. [16]

The architecture of YOLOv5 was developed theoretically and has been made available through a repository on GitHub. Ultralytics implemented YOLOv5 using the PyTorch framework, one of the most widely used frameworks in the artificial intelligence community. This architecture is only a preliminary basis that researchers can adapt to their own needs, making adjustments such as adding layers, removing blocks, integrating additional image processing methods and varying optimization methods and activation functions.

It can be noted that YOLOv5 is an open source project that includes a set of object identification models and algorithms based on the YOLO model, pre-trained

on the COCO dataset. These models are accessible and offer a wide range of applications in different fields.

Finally, YOLOv5 is available in several versions, as is possible to see in Fig.3.3, each with increasing levels of accuracy and different training times. These versions include Nano(n), Small (s), Medium (m), Large (l) and Extra Large (x), this enables the selection of the one most suited to the project at hand.

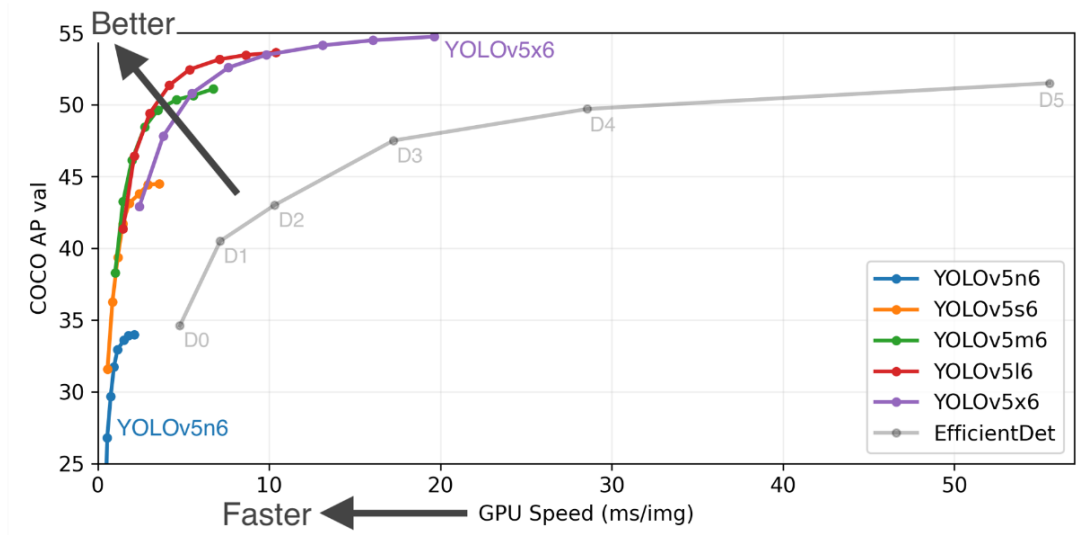


Figure 3.3: YOLOv5 versions

### 3.4 SSD Mobilenet

In the context of the SSD Mobilenet model, Convolutional Neural Networks are used for the detection and classification of objects in an input image. This model is specially designed for the use of CNN in the context of classifying objects into predefined classes. Its main feature is that it is a lightweight and fast model, which exploits the MobileNet architecture for image feature extraction and the Single Shot Detector (SSD) algorithm for real-time detection. This combination makes it particularly suitable for mobile applications and situations where fast and efficient object detection is critical.

The SSD model makes use of an architecture known as the "Single Shot Multibox Detector", shown in Fig.3.4, based on a forward convolutional network. This algorithm was developed with the goal of detecting objects in a single iteration by exploiting a deep neural network. It operates at different scales and is capable of detecting objects of varying sizes within the image. To achieve this, a basic network is employed to extract features from the image. Then, additional convolution

layers generate a series of feature maps with different sizes in order to facilitate the detection process. In addition, a nonmaximal suppression technique is applied to ensure that only a bounding box is maintained for each detected object.

The SSD approach adopts a multi-reference and multi-resolution detection methodology. In other words, it establishes a series of anchored boxes with different sizes and proportions at different positions in the image, thus providing for a detection box based on these references. This method makes it possible to detect objects at various scales and in different parts of the network. The SSD model implements an algorithm that can detect different classes of objects in images by generating a confidence score related to the presence of a category of objects in each predefined field. Further corrections are made to the bounding boxes in order to better fit them to the shape of the objects themselves. The particularity of this network lies in its adaptation to real-time situations, since it does not require the extraction of additional features based on assumptions about the bounding boxes. The entire process takes place within a single deep neural network, and object detection occurs at various scales, allowing objects of different sizes to be detected. [17] [18]

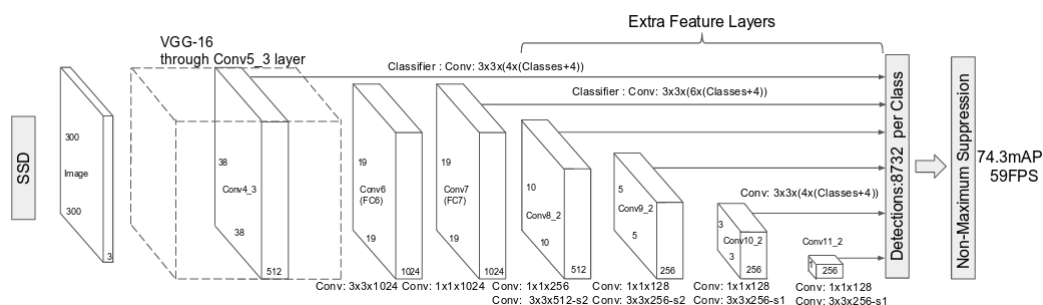


Figure 3.4: SSD architecture

In parallel, the MobileNet model is a convolutional neural network architecture designed specifically for mobile applications. Its structure is based on deep separable convolutions, with the exception of the first layer, which employs a standard convolution. Each layer is followed by batch normalization and the ReLU 6 function, with the exception of the last fully connected, nonlinearity-free layer, which feeds a softmax layer for classification. The distinguishing feature of this architecture is that it splits the standard convolution into a depth convolution and a 1x1 convolution, known as point convolution. The in-depth convolution applies a filter to each input channel, while the point convolution generates a 1x1 convolution to combine the outputs of the in-depth convolution. This partitioning contributes greatly to reducing the overall computational cost and model size. Batch normalization allows further data optimization by adjusting the learning parameters,



learning rate, dropout ratio and limiting gradient vanishing. It should also be noted that this architecture is able to take advantage of learning transfer, requiring a minimal amount of computational power for both training and application.

The innovative combination of deep convolutions and point convolutions in the MobileNet framework speeds up the learning process and reduces computation time. Note that the model also has a residual connection, an element that improves the accuracy of the architecture compared to those without residual connections. [18] [19]

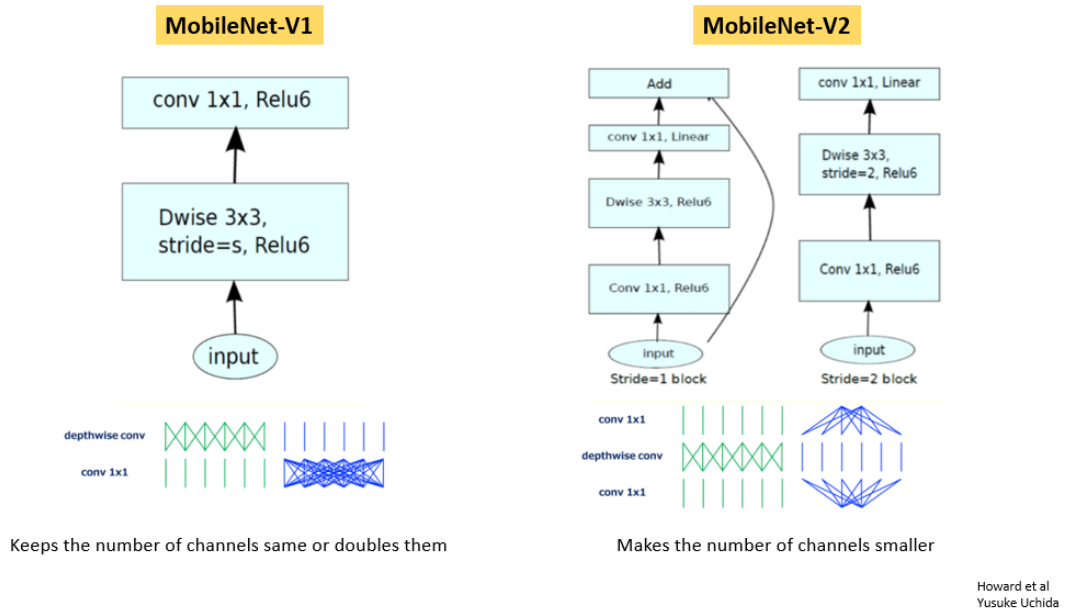
Regarding the versions of the MobileNet model there are two main variants, as can be seen from the Fig.3.5 MobileNetV1 is in conjunction with SSD, this version eliminates the last layers such as FC, Maxpool and Softmax. Instead, it exploits the outputs of the last convolution layer in MobileNet, subjecting them to further convolution operations to obtain a set of feature maps. These maps are, then, used as inputs to the object detection process. The main feature of this version is the adoption of depth-separable convolutions, which greatly helps to reduce the size and computational cost of the model, making it suitable for situations with limited computational resources. MobileNet-SSDV2 is the second edition of the MobileNet family. An inverted residue structure is introduced to improve the modularity of the model. This version has allowed the elimination of non-linearities present in the narrow layers. The structure of the residual layers has been adapted based on the ResNet architecture, allowing the accuracy of the deep convolution layers to be improved without adding significant computational load. In addition, "bottleneck" layers are used to reduce the input size, and an upper bound is applied to the ReLU layer to limit the overall complexity. [20] [19]

### 3.5 SSDmobilenetv2 vs YOLOv5

In the scientific literature, elements of convergence and divergence have emerged between the two models selected for this analysis, namely YOLOv5 and SSD MobileNetv2. Both of these models fall into the category of "one-stage detectors", in which object detection is a simple regression problem in which the model learns the probability classes and bounding box coordinates.[21]

An example study conducted in "Object Detection using YOLO And Mobilenet SSD: A Comparative Study" [22] analyzed the YOLO object detection model and the MobileNet SSD model and evaluated their performance in different scenarios. Each model has unique characteristics and is effective in their respective applications. In particular, YOLO offers higher accuracy than MobileNetSSD, while the latter stands out for higher detection speed, especially in the presence of small objects.

In another relevant contribution entitled "Comparison of Object Detection Algorithms for Objects at the Street Level" [23] a detailed analysis of object detection



**Figure 3.5:** MobileNet-V1 vs MobileNet-V2 architecture

algorithms including MobileNetv2 SSD and YOLOv5 - particularly the YOLOv5l and YOLOv5s variants - focusing on object detection in street-level contexts is presented. In this study, a modified Udacity Self Driving Car dataset was used, which was further extended through various image enhancement techniques. The results show that YOLOv5l is the most accurate algorithm, while SSD MobileNetv2 FPN-lite is one of the fastest in roadside object detection. However, further analysis indicates that YOLOv5s is the ideal algorithm for real-time applications, such as in the case of self-driving vehicles, as it offers relatively accurate results in reasonable time.

Another study titled "Comparison of YOLOv3, YOLOv5s and MobileNet-SSD V2 for Real-Time Mask Detection" [19] focuses on evaluating different methodologies for real-time mask detection. This study analyses three widely used machine learning algorithms: YOLOv3, YOLOv5 and MobileNet-SSD V2. The results show that YOLOv5s is the most suitable model for real-time applications, as it combines accuracy and speed optimally. In contrast, MobileNet-SSD V2 offers similar speed to YOLOv5s, but with a trade-off in accuracy.

These analyses highlight the importance of balancing accuracy and speed in real-time object detection applications. The choice between models depends on the specific needs of the application, with YOLO usually excelling in terms of accuracy and MobileNet SSD generally in terms of speed.

## Chapter 4

# Hardware and software analysed tools

### 4.1 Environment

To conduct a more in-depth evaluation, we chose to run the training of the YOLOv5 and SSD MobileNet models in two separate contexts in order to explore their limitations and evaluate their respective capacity.

#### 4.1.1 NVIDIA® Jetson Nano™

The selection of the development environment was mainly oriented toward the NVIDIA® Jetson Nano™, a choice motivated by its ability to support future developments related to the evolution of this thesis. The device in question, showed in Fig.4.1, represents a compact, yet high-performance computer characterized by a low power consumption of only 5 watts. Such a platform enables the simultaneous execution of several neural networks, making it particularly suitable for applications such as image classification, object detection, segmentation and language processing.

The NVIDIA Jetson Nano operates in a Linux-based environment and enjoys active support from a active community, with numerous open source projects distributed across the network. This characteristic gives the device a particular adaptability to academic approaches, providing a rich context for resources and collaborations to carry out the present research.

Fig.4.2 shows the Jetson Nano technical specifications.



Figure 4.1: NVIDIA® Jetson Nano™

TECHNICAL SPECIFICATIONS	
<b>GPU</b>	NVIDIA Maxwell architecture with 128 NVIDIA CUDA® cores
<b>CPU</b>	Quad-core ARM Cortex-A57 MPCore processor
<b>Memory</b>	4 GB 64-bit LPDDR4, 1600MHz 25.6 GB/s
<b>Storage</b>	16 GB eMMC 5.1
<b>Video Encode</b>	250MP/sec 1x 4K @ 30 (HEVC) 2x 1080p @ 60 (HEVC) 4x 1080p @ 30 (HEVC) 4x 720p @ 60 (HEVC) 9x 720p @ 30 (HEVC)
<b>Video Decode</b>	500MP/sec 1x 4K @ 60 (HEVC) 2x 4K @ 30 (HEVC) 4x 1080p @ 60 (HEVC) 8x 1080p @ 30 (HEVC) 9x 720p @ 60 (HEVC)
<b>Camera</b>	12 lanes (3x4 or 4x2) MIPI CSI-2 D-PHY 1.1 (1.5 Gb/s per pair)
<b>Connectivity</b>	Gigabit Ethernet, M.2 Key E
<b>Display</b>	HDMI 2.0 and eDP 1.4
<b>USB</b>	4x USB 3.0, USB 2.0 Micro-B
<b>Others</b>	GPIO, I <sup>2</sup> C, I <sup>2</sup> S, SPI, UART
<b>Mechanical</b>	69.6 mm x 45 mm 260-pin edge connector

Figure 4.2: Technical specifications NVIDIA® Jetson Nano™

### 4.1.2 Google Colab

In this project, another platform, Google Colab, was used to conduct a more detailed analysis of the two models.

Google Colab represents an environment in which code can be executed directly on the Cloud. To take advantage of the features of this platform, a Google account is required, through which one can log in and gain entry to the dedicated page.

Google Colab infrastructure is based on Jupyter Notebooks, interactive documents in which code can be written and executed. These documents allow code to be divided into cells, each of which can contain informational text, possibly formatted in markdown.

The use of such notebooks is convenient and widely used in the fields of data science and machine learning. Through a single document, it is possible to perform all the steps of an analytical or computing process, as well as describe its behavior in natural language. Notebooks are an effective tool for explaining or illustrating how an algorithm works.

Trough Google Colab, it is possible to import an image dataset, train an image classifier and evaluate the model, all with just a few lines of code. Colab's notepads run the code on Google cloud servers, allowing you to use the power of Google hardware, including GPUs and TPUs, regardless of the performance of your own machine. [24] [25]

## 4.2 Dataset

The initial selected dataset had four different classes, "Traffic light", "Stop", "Speedlimit", and "Crosswalk". The goal was to divide the dataset into train (80%), test (10%), and validation (10%). The dataset was downloaded from <https://www.kaggle.com/datasets/andrewmvd/road-sign-detection> and the files were in .xml format. Since SSD MobileNetV2 and YOLOv5 required different label formats, it was necessary to convert the .xml files into its corresponding .txt format file using a Python script.

In the case of SSD MobileNetV2, the .xml format, as shown in Fig.4.3, includes the following sections:

- <folder>: Name of the image folder.
- <filename>: Filename of the image file.
- <size>: Information about the size of the image (width, height, depth).
- <segmented>: Information about whether the object is part of a segmented area.

```
1 |
2 | <annotation>
3 |   <folder>images</folder>
4 |   <filename>road0.png</filename>
5 |   <size>
6 |     <width>267</width>
7 |     <height>400</height>
8 |     <depth>3</depth>
9 |   </size>
10 |   <segmented>0</segmented>
11 |   <object>
12 |     <name>trafficlight</name>
13 |     <pose>Unspecified</pose>
14 |     <truncated>0</truncated>
15 |     <occluded>0</occluded>
16 |     <difficult>0</difficult>
17 |     <bndbox>
18 |       <xmin>98</xmin>
19 |       <ymin>62</ymin>
20 |       <xmax>208</xmax>
21 |       <ymax>232</ymax>
22 |     </bndbox>
23 |   </object>
24 | </annotation>
```

Figure 4.3: Label format xml

```
Traffic light
Stop
Speedlimit
Crosswalk
```

Figure 4.4: Label format txt

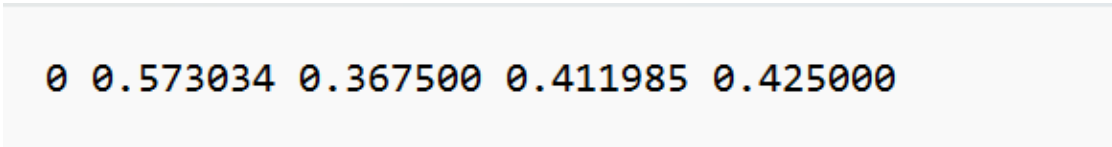
- <object>: Annotated object information, including name, pose, truncation, occlusion, difficulty and bounding box with coordinates.
- <pose>: The location of the object.
- <truncated>: Shows whether the object is partially visible or cropped in the image, 0 indicates that it is not truncated.
- <occluded>: Specifies whether the object is occluded by other objects, 0 indicates that it is not occluded.

- `<difficult>`: Means whether the object is difficult to recognize, 0 means it is not difficult.

In addition, the general file where all the labels are contained will be a .txt file, as indicated in the Fig.4.4, where there will be a simple list containing the name of each class.

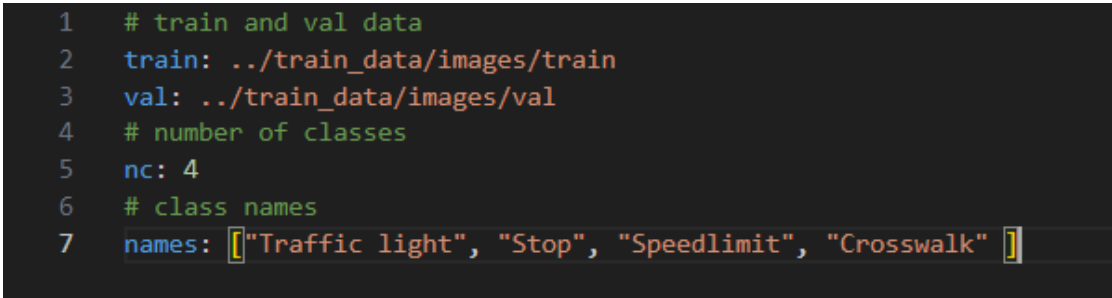
Regarding YOLOv5, as can be seen in the Fig.4.5 the .txt format includes the following information for each image:

- `<class>`: Label of the object class.
- `<x_center>`: X-coordinate of the center of the bounding box normalized to the size of the image.
- `<y_center>`: Y coordinate of the center of the bounding box normalized with respect to the size of the image.
- `<width>`: Normalized width of the bounding box relative to the image size.
- `<height>`: Normalized height of the bounding box relative to the image size.



**0 0.573034 0.367500 0.411985 0.425000**

**Figure 4.5:** Label format txt



```
1 # train and val data
2 train: ../train_data/images/train
3 val: ../train_data/images/val
4 # number of classes
5 nc: 4
6 # class names
7 names: ["Traffic light", "Stop", "Speedlimit", "Crosswalk"]
```

**Figure 4.6:** Label format yaml

The .yaml configuration file for YOLOv5 specifies the training and validation image paths, the total number of classes and the class names. Such as we can see in Fig.4.6.

## 4.3 Training model

In this section, we will proceed with the study and development of the training, as well as the implementation on Jetson Nano, of the two models considered in the analysis. The initial approach for both models involves training in a high-efficiency environment, characterized by the absence of severe constraints on CPU and GPU processing power, such as that provided by Google. Next, migration to the NVIDIA device is planned in order to evaluate its performance, including frames per second, through the use of the ASTRA camera.

During the study, especially in the case of SSD MobileNet, problems related to the conversion phase emerged, which made it necessary to develop the training directly on the device. This adaptation, however, potentially affected the overall quality of the results obtained. Deeper details regarding these issues will be covered in the next sections.

### 4.3.1 SSD MobileNet

As previously discussed, various approaches were explored both to comprehensively study the models and due to necessary modifications made throughout the study.

#### Google Colab

Initially, the model was studied on the Google Colab platform. The choice was motivated by the expectation of faster model training facilitated by Google GPU, thereby minimizing the utilization of the computer CPU. This was particularly advantageous when dealing with complex models. The initial plan was to conduct the training on Colab and subsequently convert the model for testing on the NVIDIA Jetson Nano.

The fundamental steps involved were as follows:

1. Prepare the dataset and upload it to the Google Colab folder.
2. Split images into train, validation, and test folders.
3. Create the labelmap, that is a structured mapping or dictionary that associates class labels with numerical identifiers and it defines and organizes the categories or classes that a model is trained to recognize. Each class is assigned a unique numerical label, enabling the model to output predictions with corresponding class identifiers.
4. Choose the batch size, that is the hyperparameter that defines the number of samples to work through before updating the internal model parameters.



5. Determine the number of steps; here, the term "steps" referred to the mini-batches processed during training, and the choice was not expressed in terms of epochs. The number of steps, on the other hand, refers to the number of mini-batches that the training dataset is divided into. In each step, the model processes one mini-batch of data. The number of steps is related to the size of the dataset and the batch size. It is calculated as the total number of examples in the dataset divided by the batch size. The formula for calculating steps in relation to epochs is shown in equation (4.1).

$$\text{Number of Steps} = \left( \frac{\text{Number of Examples}}{\text{Batch Size}} \right) \times \text{Number of Epochs} \quad (4.1)$$

The *Number of examples* are data points or instances in the dataset. In the case of image recognition or object detection, each example could be a distinct image. Then, the total number of examples represents the size of the dataset.

6. Conduct the model training.
7. Calculate mean Average Precision (mAP). The mAP is a metric used to measure the performance of a model that focuses on object detection tasks and information retrieval on images. The mAP uses the ground-truth bounding box, compares it to the detected box and returns a score. The higher the mAP score, the more accurate the model detects and makes correct predictions.

$$\text{mAP} = \frac{1}{N} \sum_{k=1}^N \text{AP}_k \quad (4.2)$$

Where:

mAP Average of the mean accuracies for each class.

$N$  Total number of classes.

$\text{AP}_k$  Average accuracy for the class  $k$ .

$$\text{AP} = \sum_n (R_n - R_{n-1}) P_n \quad (4.3)$$

Where  $R_n$  and  $P_n$  are the precision and recall at the  $n_{th}$  threshold.

The SSD MobileNet model, being intricate due to the composition of two networks, necessitated several hours of training.

Steps	Number of training hours	mAP
40000	6	69.16%
60000	8	74.52%

The mAP, unfortunately, does not exhibit a high value, likely attributed to the dataset utilized. Specifically, the dataset demonstrated excellent results for the "Stop" and "Speedlimit" classes, reasonably good outcomes for "Crosswalk" and notably poor performance for "Traffic Light". This discrepancy can be attributed to the abundance of data for the first two classes, accompanied by empirically superior quality in the corresponding images.

Upon scrutinizing individual classes where the number of training steps was set to 40,000, the following outcomes were observed:

Class	AP
Stop	85.53%
Speedlimit	80.73%
Crosswalk	67.61%
Traffic Light	42.77%

In the Fig.4.7 there are some test images where it is possible to notice the performance of the model application. As mentioned above, even from these two test images alone it is possible to see the goodness of the results of the "speedlimit" class and, on the other hand, the fragility of the accuracy of the "crosswalk" class.

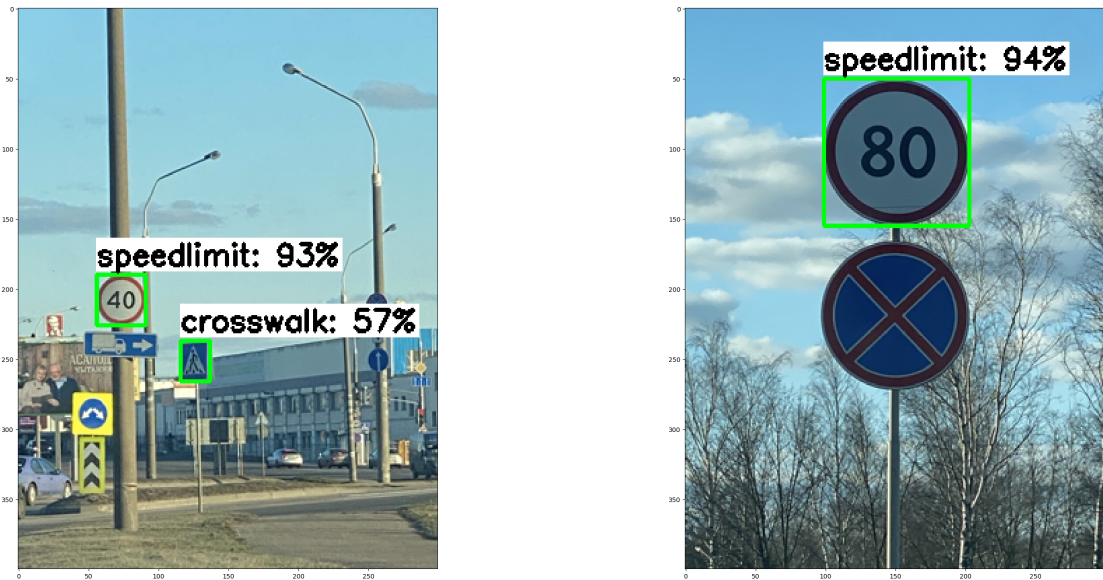


Figure 4.7: SSD Mobile Net detection of test images

The resulting model was in the .pb format. The .pb format, also known as protocol buffer (protobuf), is employed in TensorFlow to store models. Protocol buffers, developed by Google, offer a systematic approach for data storage, enhancing data transport efficiency and enforcing a structured format. The file saved\_model.pb

specifically stores the TensorFlow program or model, along with a series of named signatures, each identifying a function that takes tensor inputs and produces tensor outputs.[26]

The initial intention was to locally save this model and subsequently convert it into a .onnx model compatible with Nvidia Jetson Nano, in order to evaluate its functionality using the camera module. However, despite multiple attempts, the conversion proved unattainable. Converting a TensorFlow model into an ONNX model for Jetson Nano posed significant challenges due to the complexity of the process. Consequently, this approach had to be abandoned. Although it presented a notable advantage in terms of CPU efficiency, the inability to validate its functionalities rendered it impractical for further experimentation.

### **Jetson-inference folder**

At a later stage, it was chosen to take advantage of the functionality offered by the "jetson-inference" repository, developed by NVIDIA developer itself for educational and instructional purposes. This repository facilitates the utilization of a customized dataset for training a model, followed by direct testing on the Jetson device.

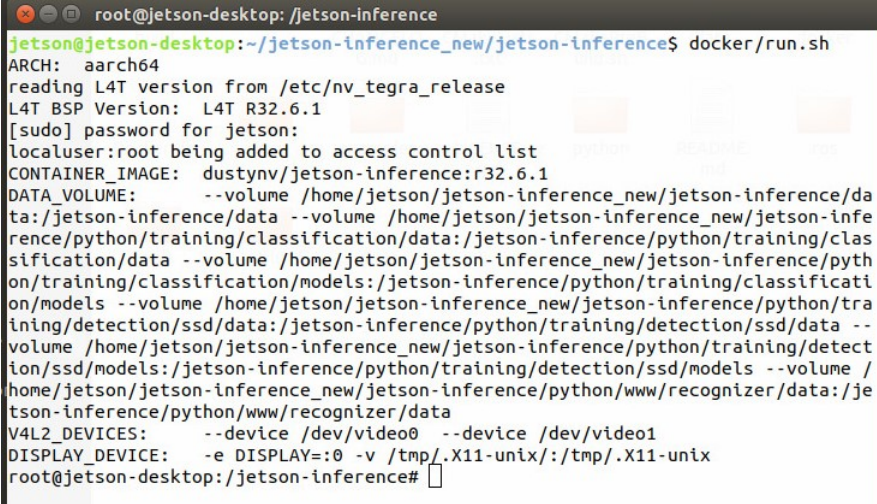
The jetson-inference repository serves as a robust inference and real-time vision Deep Neural Network (DNN) library tailored for NVIDIA Jetson devices. Leveraging TensorRT, the project efficiently executes optimized networks on GPUs, supporting both C++ and Python implementations. For model training, PyTorch is employed, enhancing the versatility of the framework. Key DNN vision primitives are supported, encompassing imageNet for image classification, detectNet for object detection, segNet for semantic segmentation, poseNet for pose estimation, and actionNet for action recognition. This comprehensive support caters to a wide array of computer vision applications on Jetson devices. Moreover, the repository includes illustrative examples demonstrating practical applications such as live camera feed streaming, web application development utilizing WebRTC, and seamless integration with ROS/ROS2. [27]

In this situation, the strategy of "transfer learning", an approach that facilitates the retraining of a DNN model on a new dataset, is adopted, requiring less time than creating a network from scratch. Through transfer learning, the weights of an already trained model are refined to classify a customized dataset. [28]

In order to compose a custom training of the model, the following steps must be followed:

1. Initially, it is necessary to start the Docker container by running the *docker/run.sh* command, as shown in Fig. 4.8. This command will automatically download the correct version of the container from DockerHub, based on the currently installed version of JetPack-L4T. In addition, the command will

mount the relevant data directories and devices, enabling the use, for example, of cams or displays within the container. [29]



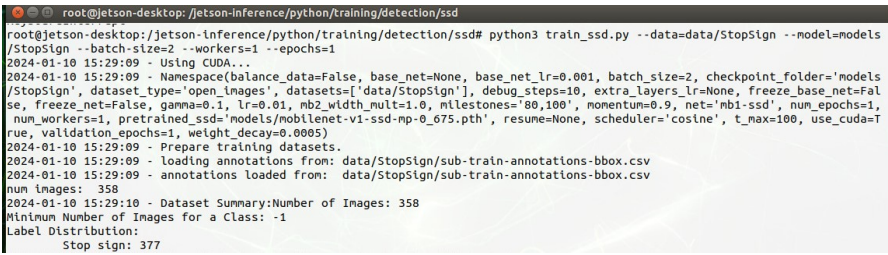
```

root@jetson-desktop: /jetson-inference
jetson@jetson-desktop:~/jetson-inference_new/jetson-inference$ docker/run.sh
ARCH: aarch64
reading L4T version from /etc/nv_tegra_release
L4T BSP Version: L4T R32.6.1
[sudo] password for jetson:
localuser:root being added to access control list
CONTAINER_IMAGE: dustynv/jetson-inference:r32.6.1
DATA_VOLUME: --volume /home/jetson/jetson-inference_new/jetson-inference/data:/jetson-inference/data --volume /home/jetson/jetson-inference_new/jetson-inference/python/training/classification/data:/jetson-inference/python/training/classification/data --volume /home/jetson/jetson-inference_new/jetson-inference/python/training/classification/models:/jetson-inference/python/training/classification/models --volume /home/jetson/jetson-inference_new/jetson-inference/python/training/detection/ssd/data:/jetson-inference/python/training/detection/ssd/data --volume /home/jetson/jetson-inference_new/jetson-inference/python/training/detection/ssd/models:/jetson-inference/python/training/detection/ssd/models --volume /home/jetson/jetson-inference_new/jetson-inference/python/www/recognizer/data:/jetson-inference/python/www/recognizer/data
V4L2_DEVICES: --device /dev/video0 --device /dev/video1
DISPLAY_DEVICE: -e DISPLAY=:0 -v /tmp/.X11-unix/:/tmp/.X11-unix
root@jetson-desktop: /jetson-inference#

```

Figure 4.8: Docker container start

2. Next, it is required to import or download the folder containing the images and their labels intended for training your model. This process involves splitting the data into training, validation and test sets.
3. By Running the python script *train\_ssd.py* from the command line, as highlighted in Fig.4.9, it will be possible to start the model training process. At this stage, you will need to specify the parameters of the number of batch sizes, workers and epochs. In Table 4.1 [30], the descriptions of the functions of each command are given concisely.



```

root@jetson-desktop: /jetson-inference/python/training/detection/ssd
root@jetson-desktop: /jetson-inference/python/training/detection/ssd# python3 train_ssd.py --data=data/StopSign --model=models/StopSign --batch-size=2 --workers=1 --epochs=1
2024-01-10 15:29:09 - Using CUDA...
2024-01-10 15:29:09 - Namespace(balance_data=False, base_net=None, base_net_lr=0.001, batch_size=2, checkpoint_folder='models/StopSign', dataset_type='open_images', datasets=['data/StopSign'], debug_steps=10, extra_layers_lr=None, freeze_base_net=False, freeze_net=False, gamma=0.1, lr=0.01, nb2_width_mult=1.0, milestones='80,100', momentum=0.9, net='mb1-ssd', num_epochs=1, num_workers=1, pretrained_ssd='models/mobilenet-v1-ssd-mp-0.675.pth', resume=None, scheduler='costine', t_max=100, use_cuda=True, validation_epochs=1, weight_decay=0.0005)
2024-01-10 15:29:09 - Prepare training datasets.
2024-01-10 15:29:09 - loading annotations from: data/StopSign/sub-train-annotations-bbox.csv
2024-01-10 15:29:09 - annotations loaded from: data/StopSign/sub-train-annotations-bbox.csv
num images: 358
2024-01-10 15:29:10 - Dataset Summary: Number of Images: 358
Minimum Number of Images for a Class: -1
Label Distribution:
Stop sign: 377

```

Figure 4.9: Training command

In this context, however, initial problems occurred due to a lack of CPU memory, as evidenced by the error visible in Fig.4.10.

Command	Description
-data	the Location of the dataset
-model-dir	Directory to output the trained model checkpoints
-batch-size	Number of training data used in a single iteration
-epochs	Number of complete runs through the entire data set when training a model
-workers	Number of data loader threads (0 = disable multithreading)

**Table 4.1:** Command-Line for training

```

root@jetson-desktop: /jetson-inference/python/training/detection/ssd
data = [self.dataset[idx] for idx in possibly_batched_index]
File "/usr/local/lib/python3.6/dist-packages/torch/utils/data/_utils/fetch.py", line 44, in <listcomp>
data = [self.dataset[idx] for idx in possibly_batched_index]
File "/usr/local/lib/python3.6/dist-packages/torch/utils/data/dataset.py", line 257, in __getitem__
return self.datasets[dataset_idx][sample_idx]
File "/jetson-inference/python/training/detection/ssd/vision/datasets/open_images.py", line 46, in __getitem__
image, boxes, labels = self._getitem(index)
File "/jetson-inference/python/training/detection/ssd/vision/datasets/open_images.py", line 40, in _getitem
image, boxes, labels = self.transform(image, boxes, labels)
File "/jetson-inference/python/training/detection/ssd/vision/ssd/data_preprocessing.py", line 34, in __call__
return self.augment(img, boxes, labels)
File "/jetson-inference/python/training/detection/ssd/vision/transforms/transforms.py", line 55, in __call__
img, boxes, labels = t(img, boxes, labels)
File "/jetson-inference/python/training/detection/ssd/vision/transforms/transforms.py", line 407, in __call__
in, boxes, labels = distort(in, boxes, labels)
File "/jetson-inference/python/training/detection/ssd/vision/transforms/transforms.py", line 55, in __call__
img, boxes, labels = t(img, boxes, labels)
File "/jetson-inference/python/training/detection/ssd/vision/transforms/transforms.py", line 167, in __call__
image = cv2.cvtColor(image, cv2.COLOR_RGB2HSV)
cv2.error: OpenCV(4.5.0) /opt/opencv/modules/core/src/alloc.cpp:73: error: (-4:Insufficient memory) Failed to allocate 167961
600 bytes in function 'OutOfMemoryError'
    
```

**Figure 4.10:** OutOfMemory error

To address this issue, several approaches were explored to alleviate the load on the Jetson CPU. Initially, it was decided to reduce the size of the dataset to a single traffic signal, with the goal of making the dataset more manageable and occupying as little memory space as possible. In particular, it was decided to focus exclusively on the detection of the "Stop" signal to avoid overloading the CPU.

Subsequently, attempts were made to address the problem by gradually reducing the number of batch sizes 32 down to 2. However, even with minimal batch sizes, problems of insufficient memory persisted.

In general, a larger batch size speeds up the training of the model for each epoch. This is because, depending on available computational resources, the machine can process much more than a single sample at a time. However, the trade-off is that even if the machine can handle very large batches, the quality of the model may degrade and it may have difficulty generalizing to previously unseen data.

Reducing the batch size reduces the computational power required by the Jetson, but this affects the performance of the algorithm. Despite these adjustments, the problem of insufficient memory persisted.

Therefore, the option of setting the number of workers to 0 was considered, even though this resulted in a reduction in model performance. However, even

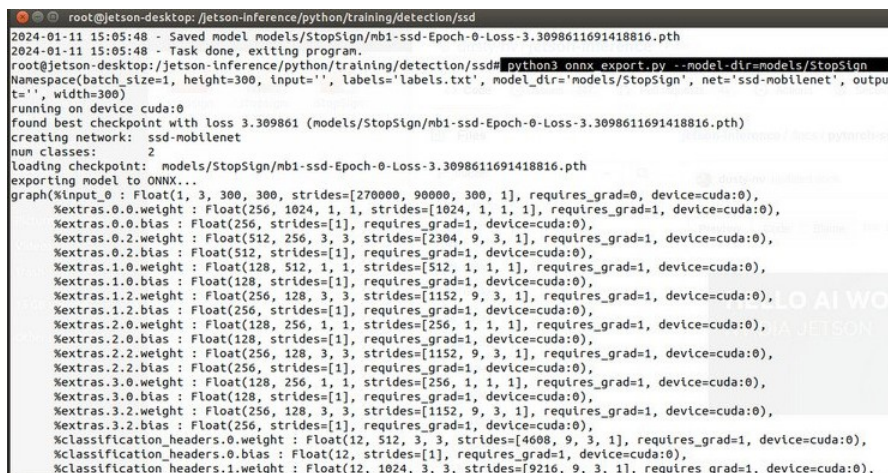


with this change, the problem remained.

The only remaining solution was to increase the memory of the Jetson. It was suggested that an additional swap space of 4 Gigabytes be mounted to provide more virtual memory during resource-intensive tasks, such as training machine learning models.

The process to increase memory included disabling ZRAM, creating a permanent 4-gigabyte swap file, and using this additional swap space to prevent running out of RAM during model training. This modification allowed successful continuation of model training.

4. Later, the model will be archived and it will be possible to convert it to the ONNX (Open Neural Network Exchange) format, as showed in Fig.4.11 that denotes an open standard created for representing machine learning models. ONNX establishes a standard file format to allow artificial intelligence developers to use models with different frameworks, tools, runtimes and compilers.[31]



```

root@jetson-desktop: /jetson-inference/python/training/detection/ssd
2024-01-11 15:05:48 - Saved model models/StopSign/mb1-ssd-Epoch-0-Loss-3.3098611691418816.pth
2024-01-11 15:05:48 - Task done, exiting program.
root@jetson-desktop: /jetson-inference/python/training/detection/ssd# python3 onnx_export.py --model_dir=models/StopSign
Namespace(batch_size=1, height=300, input='', labels='labels.txt', model_dir='models/StopSign', net='ssd-mobilenet', output='t', width=300)
running on device cuda:0
found best checkpoint with loss 3.309861 (models/StopSign/mb1-ssd-Epoch-0-Loss-3.3098611691418816.pth)
creating network: ssd-mobilenet
num classes: 2
loading checkpoint: models/StopSign/mb1-ssd-Epoch-0-Loss-3.3098611691418816.pth
exporting model to ONNX...
graph(%input_0 : Float(1, 3, 300, 300, strides=[270000, 90000, 300, 1], requires_grad=0, device=cuda:0),
  %extras.0.0.weight : Float(256, 1024, 1, 1, strides=[1024, 1, 1, 1], requires_grad=1, device=cuda:0),
  %extras.0.0.bias : Float(256, strides=[1], requires_grad=1, device=cuda:0),
  %extras.0.2.weight : Float(512, 256, 3, 3, strides=[2304, 9, 3, 1], requires_grad=1, device=cuda:0),
  %extras.0.2.bias : Float(512, strides=[1], requires_grad=1, device=cuda:0),
  %extras.1.0.weight : Float(128, 512, 1, 1, strides=[512, 1, 1, 1], requires_grad=1, device=cuda:0),
  %extras.1.0.bias : Float(128, strides=[1], requires_grad=1, device=cuda:0),
  %extras.1.2.weight : Float(256, 128, 3, 3, strides=[1152, 9, 3, 1], requires_grad=1, device=cuda:0),
  %extras.1.2.bias : Float(256, strides=[1], requires_grad=1, device=cuda:0),
  %extras.2.0.weight : Float(128, 256, 1, 1, strides=[256, 1, 1, 1], requires_grad=1, device=cuda:0),
  %extras.2.0.bias : Float(128, strides=[1], requires_grad=1, device=cuda:0),
  %extras.2.2.weight : Float(256, 128, 3, 3, strides=[1152, 9, 3, 1], requires_grad=1, device=cuda:0),
  %extras.2.2.bias : Float(256, strides=[1], requires_grad=1, device=cuda:0),
  %extras.3.0.weight : Float(128, 256, 1, 1, strides=[256, 1, 1, 1], requires_grad=1, device=cuda:0),
  %extras.3.0.bias : Float(128, strides=[1], requires_grad=1, device=cuda:0),
  %extras.3.2.weight : Float(256, 128, 3, 3, strides=[1152, 9, 3, 1], requires_grad=1, device=cuda:0),
  %extras.3.2.bias : Float(256, strides=[1], requires_grad=1, device=cuda:0),
  %classification_headers.0.weight : Float(12, 512, 3, 3, strides=[4608, 9, 3, 1], requires_grad=1, device=cuda:0),
  %classification_headers.0.bias : Float(12, strides=[1], requires_grad=1, device=cuda:0),
  %classification_headers.1.weight : Float(12, 1024, 3, 3, strides=[9216, 9, 3, 1], requires_grad=1, device=cuda:0),

```

Figure 4.11: Conversion to ONNX model

5. After the conversion is done, it will be possible to employ the detectnet command to conduct various tests, as shown in Fig.4.12, initiating access to the camera by the appropriate command-line selection. Tests and results will be discussed in the following chapter.

## SSD MobileNet with COCO128

An alternative way to ensure dataset diversity is to use the pre-trained SSD-MobileNet model, included in the jetson-inference directory, which exploits the

```

root@jetson-desktop: /jetson-inference/python/training/detection/ssd
root@jetson-desktop: /jetson-inference/python/training/detection/ssd# detectnet --model=models/StopSign/ssd-mobilenet.onnx --
labels=models/StopSign/Labels.txt --input-blob=input_0 --output-cvgscores --output-bbox-boxes --device /dev/video1
[gstreamer] initialized gstreamer, version 1.14.5.0
[gstreamer] gstCamera -- attempting to create device v4l2:///dev/video1
[gstreamer] gstCamera -- found v4l2 device: USB 2.0 Camera
[gstreamer] v4l2-proplist, device.path=(string)/dev/video1, udev-probed=(boolean)false, device.apl=(string)v4l2, v4l2.device
.driver=(string)uvcvideo, v4l2.device.card=(string)"USB\ 2.0\ Camera", v4l2.device.bus_info=(string)usb-70090000.xusb-2.3.2,
v4l2.device.version=(uint)264701, v4l2.device.capabilities=(uint)2216689665, v4l2.device.device_caps=(uint)69206017;
[gstreamer] gstCamera -- found 12 caps for v4l2 device /dev/video1
[gstreamer] [0] video/x-raw, format=(string)YUY2, width=(int)2048, height=(int)1536, pixel-aspect-ratio=(fraction)1/1, frame
rate=(fraction)30/1;
[gstreamer] [1] video/x-raw, format=(string)YUY2, width=(int)1920, height=(int)1080, pixel-aspect-ratio=(fraction)1/1, frame
rate=(fraction)30/1;
[gstreamer] [2] video/x-raw, format=(string)YUY2, width=(int)1280, height=(int)960, pixel-aspect-ratio=(fraction)1/1, frame
rate=(fraction)10/1;
[gstreamer] [3] video/x-raw, format=(string)YUY2, width=(int)1280, height=(int)720, pixel-aspect-ratio=(fraction)1/1, frame
rate=(fraction)10/1;
[gstreamer] [4] video/x-raw, format=(string)YUY2, width=(int)640, height=(int)480, pixel-aspect-ratio=(fraction)1/1, frame
rate=(fraction)30/1;
[gstreamer] [5] video/x-raw, format=(string)YUY2, width=(int)320, height=(int)240, pixel-aspect-ratio=(fraction)1/1, frame
rate=(fraction)30/1;
[gstreamer] [6] image/jpeg, width=(int)2048, height=(int)1536, pixel-aspect-ratio=(fraction)1/1, framerate=(fraction)30/1;
[gstreamer] [7] image/jpeg, width=(int)1920, height=(int)1080, pixel-aspect-ratio=(fraction)1/1, framerate=(fraction)30/1;
[gstreamer] [8] image/jpeg, width=(int)1280, height=(int)960, pixel-aspect-ratio=(fraction)1/1, framerate=(fraction)30/1;
[gstreamer] [9] image/jpeg, width=(int)1280, height=(int)720, pixel-aspect-ratio=(fraction)1/1, framerate=(fraction)30/1;
[gstreamer] [10] image/jpeg, width=(int)640, height=(int)480, pixel-aspect-ratio=(fraction)1/1, framerate=(fraction)30/1;
[gstreamer] [11] image/jpeg, width=(int)320, height=(int)240, pixel-aspect-ratio=(fraction)1/1, framerate=(fraction)30/1;
[gstreamer] gstCamera -- selected device profile: codec=mjpeg format=unknown width=1280 height=720
[gstreamer] gstCamera pipeline string:
[gstreamer] v4l2src device=/dev/video1 ! image/jpeg, width=(int)1280, height=(int)720 ! jpegdec ! video/x-raw ! appsink name
=mysink
[gstreamer] gstCamera successfully created device v4l2:///dev/video1
[v4l2] created gstCamera from v4l2:///dev/video1

```

Figure 4.12: Detectnet command

COCO128 dataset. We refer to this dataset as a subset of 128 images of the larger COCO dataset. This subset reuses the training set for both validation and testing, with the goal of demonstrating the suitability of its training process and ability to fit this small dataset.

The COCO128 dataset includes images of complex scenes containing common objects, and is annotated with bounding boxes and other relevant information. It is an appropriate choice during the initial testing phase of a new model, especially where the main problem is the lack of memory.

The full version of the dataset, in contrast, contains thousands of labeled images, covering more than 80 object categories. However, it was necessary to customize the object recognition phase so as to identify only those objects relevant to the specific investigation, through an appropriate script in python. These objects do not exactly match the initially selected dataset tested in the previous two cases. The chosen classes include: "stop sign", "traffic signal" and "traffic light".

In this circumstance, the ObjectDetector was implemented, as seen in Fig.4.13, which is designed to call the detectNet function within the jetson.inference package. The Jetson Inference library is designed to run deep neural networks on NVIDIA Jetson platforms. The detectNet class is specifically used for object detection.

In the context of this implementation, the detectNet function is called by providing the name of the pre-trained model, specifically `ssd-mobilenet-v2`, which is the model under study in this section. In addition, the threshold parameter is specified, which is intended to establish a confidence threshold for object detection. Objects with a detection confidence below this threshold are ignored. In the `detect_objects` method, the image is converted from BGR color space to RGBA using the OpenCV library. Then the array data is converted to float32. The converted array is then transferred

to the GPU memory using the Jetson Inference library. Finally, the Detect method of the detectNet object is called, passing the image, width and height. This will return a list of detections containing information about the objects found in the image. In the *GetClassDesc* method instead A mapping between class IDs and their descriptions is defined. If the class is detected, it returns the corresponding description.

```
class ObjectDetector:
    def __init__(self, threshold=0.7):
        self.net = jetson.inference.detectNet('ssd-mobilenet-v2', threshold=threshold)

    def detect_objects(self, img):
        frame = cv2.cvtColor(img, cv2.COLOR_BGR2RGBA).astype(np.float32)
        frame = jetson.utils.cudaFromNumpy(frame)
        width, height = img.shape[1], img.shape[0]
        detections = self.net.Detect(frame, width, height)
        return detections

    def GetClassDesc(self, class_id):
        # Assuming you have a class mapping
        class_mapping = {10: 'traffic light', 12: 'street sign', 13: 'stop sign'}
        if class_id in class_mapping:
            return class_mapping[class_id]
        else:
            # Handle the case where the class ID is not in the mapping
            return f"Unknown Class (ID: {class_id})"
```

Figure 4.13: Class Object Detector



## 4.3.2 YOLOv5

### Google Colab

During this phase, the initial dataset was trained, as previously outlined at the beginning of the chapter, using the GPU available on Google Colab. To optimize the training process for the traffic signals selected at the very beginning of the project, the Colab ultralytics sheet provided directly by YOLO was used and adapted for the purpose of this project.

Here are the main steps to train this model:

1. Prepare the dataset and upload it to the Google Colab folder.
2. Divide the dataset between training, validation and testing.
3. Download the folder directly from <https://github.com/ultralytics/yolov5>
4. Prepare and upload the .yaml file containing the labels.
5. Start YOLOv5s training with the Python script called train.py by setting the parameters in the Tab.4.2

Command	Description
- img	Sets the input image size during training
- batch	Number of training data used in a single iteration
- epochs	Number of complete runs through the entire data set when training a model
- data	Specifies the YAML configuration file containing information about the dataset, such as image path and labels
- weights	Specifies the initial weights of the model. In this case, the model begins training using the pre-trained weights for YOLOv5s
- cache	This parameter enables caching of data during training, which can speed up the training process

**Table 4.2:** Command-Line for training

Model selection was directed toward YOLOv5s, the second smallest and fastest configuration within the available options. This decision was guided by finding a balance between obtaining satisfactory results, network complexity, and, consequently, efficiency in training time. In detail, in Figure 4.14, a comparative analysis of the different versions of the model is provided, based on results obtained through the COCO dataset. These results include metrics such as FPS, mAP and model execution time expressed in milliseconds. [32]

Different combinations of batch-sizes and epochs were experimented with, as detailed in Table 4.3. This study was conducted by taking into consideration the training speed and simultaneously calculating the mean Average Precision.

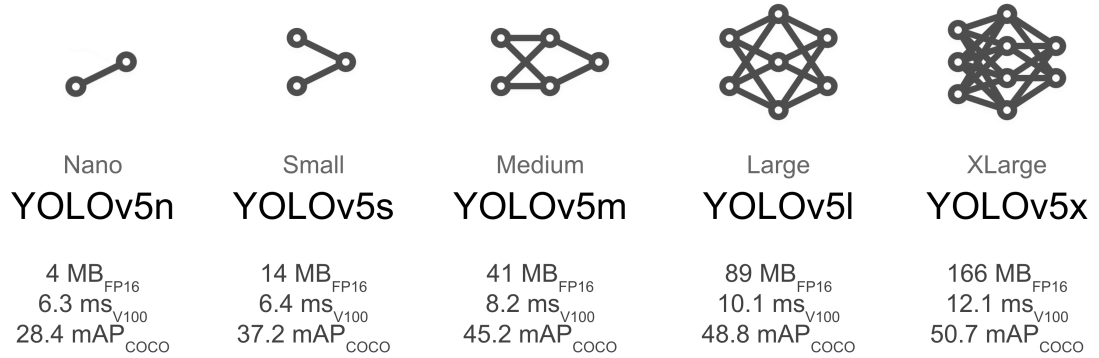


Figure 4.14: Yolo model comparison

Epochs	Batch-size	Training time	mAP
100	16	35 min	97.7%
100	32	33 min	98.2%
100	64	32 min	98.9%
100	128	–	CUDA out of memory
200	16	71 min	98.7%
200	32	65 min	97.5%
200	64	64 min	97.7%
300	16	80 min	98.2%
300	32	67 min	98.3%
300	64	97 min	97.6%

Table 4.3: Training results of YOLOv5 model in Google Colab

In the context of this analysis, significant improvements were found on all classes considered. The results obtained were more than satisfactory for both "Stop" and "Speed Limit" classes. This optimization was also extended to the other two classes, namely "Traffic Light" and "Crosswalk." Despite the fact that the latter had a lower number of images, a problem encountered in the previous model, performance was found to be almost equivalent to the first two classes. This positively affected the mean accuracy values, which were evaluated through various parameters such as batch-size and training epochs. By virtue of these adjustments, the results achieved were excellent, underscoring the effectiveness of the changes made to the model under consideration. The validity of what has been stated is supported by the outcomes found in Tables 4.4 and 4.5, which detail the AP for each class in the best combinations of batch-size and epochs identified.

Class	AP
Stop	99.5%
Speedlimit	99.5%
Crosswalk	99%
Traffic Light	94.6%

**Table 4.4:** AP YOLOv5 model  $epochs = 300$   $batch - size = 32$ 

Class	AP
Stop	99.5%
Speedlimit	99.5%
Crosswalk	98.9%
Traffic Light	94.6%

**Table 4.5:** AP YOLOv5 model  $epochs = 300$   $batch - size = 16$ 

The efficiency achieved is made possible by a faster training process, facilitated by the leaner and more advanced architecture of the YOLO model, which proved superior to the complexity of the previously used model. The default model includes 8 workers, a configuration that has been retained. This setup provides a clear idea of the power of the model, due to the parallelization of processes, which is also supported by the google GPU that enables such execution.

In the Tab.4.3 it can be seen that the batch-size increase beyond the value of 64 was prevented by the circumstance of CUDA memory exhaustion. This condition indicates that the GPU used during the training process has reached its maximum available memory capacity, preventing the execution of further operations and thus requiring the batch-size to be reduced.

Considering that the model was trained in an online environment, the camera view was impractical, so it was tested using a set of images. The confidence threshold was configured to the value of 0.75, a parameter that sets the minimum probability required for a prediction to be considered valid. The tests were performed using the optimal configuration with  $batch - size = 32$  and  $epochs = 300$ . However, it should be noted that the results would not vary significantly with other configurations, as a high level of performance was found under basically all conditions. In Fig.5.14 it can be seen that the model is able to detect signals even under suboptimal conditions, such as partially occluded signals or positioned at considerable distances.



Figure 4.15: Tests performed with  $epochs = 300$  and  $batch - size = 32$

### Yolo for Nvidia Jetson Nano

The YOLOv5 network model is implemented in the PyTorch framework, an open source machine learning infrastructure based on the Torch library used for applications such as computer vision and natural language processing. However, there was a drawback related to the Python requirements needed to run YOLOv5 on the Jetson Nano device. The documentation specifies the need for Python

3.8, while Jetson Nano is configured by default with Python 3.6. Addressing this disparity may involve creating a virtual environment or migrating the entire system to Python3.8. However, the latter option can prove problematic since changing the default Python version on the device can lead to technical difficulties and risks to system integrity. The solution adopted was to download Python3.8 and create a virtual environment with that version. This made it possible to clone the YOLOv5 directory belonging to Ultralytic, thus overcoming Python version limitations. Python virtual environments were used as a tool to isolate the working environment from other system components, providing an efficient approach to testing.

Performing object detection on traffic signals using YOLO on Jetson Nano can be approached through several methods, as was also discussed with the previous model. Indeed, one option could be to use the COCO128 dataset directly from the default YOLO folder. However, considering that it is possible to import models previously trained on Google Colab, one could opt to use a model trained specifically for the four classes of interest: "Stop", "Traffic Light", "Crosswalk", and "Speed Limit". The option of training a model directly on Jetson Nano could be considered, but it should be noted that this approach could result in lower performance than using already trained models. Moreover, training the model on Jetson could require substantial demands on resources and time, particularly attributable to the constrained RAM capacity inherent in the Nvidia Jetson Nano. For instance, the necessity to reduce the maximum number of workers from the default value of 8 to 1 is influenced by RAM restrictions, consequently leading to a notable decline in the results achieved compared to training the model on Colab. By utilizing the detect.py script provided within the official GitHub repository of Ultralytics, multiple configurable parameters become available. The tests conducted can be seen later in the 5.2. The primary parameters are outlined in the following Tab.4.6. However, a limitation can be identified in the use of the YOLO model on Jetson Nano through the camera, manifested in the reduced frame rate per second, which completely contrasts the goodness of the accuracy in detection.

It is possible to partially improve them by decreasing the quality of the camera, which will, however, irretrievably worsen the recognition of signals, but still does not reach the FPS we had in the other model. In particular, the parameter `-img-size` is expressed in multiples of 32 pixels and is set to 640 by default. With this parameter set as default we will have very good results in terms of accuracy, but poor in terms of FPS. Different configurations were tried, to see how they increased FPS, but without losing traffic sign detection, as can be seen from the Tab.4.7.

The maximum allowable limit of `-img-size = 96` was configured in order to maintain the functionality of object detection. However, it is important to note that with this setting, the ability to detect signals placed at a distance greater than 20 cm is compromised. Therefore, it is recommended to consider using a

Command	Description
<code>- img-size</code>	Image size during inference
<code>- weights</code>	Specifies the custom trained YOLOv5 weights file
<code>- conf</code>	Sets the confidence threshold
<code>- device</code>	Specifies the device for inference

**Table 4.6:** Command-Line for detection

<code>- img-size</code>	FPS
96	8
128	6
256	4
320	2.5

**Table 4.7:** YOLOv5 FPS on Jetson Nano

maximum value of `--img-size = 128` through the use of Ultralytic `detect.py` script. However, the restrictions experienced in relation to the number of frames per second do not allow for alternatives other than finding a solution aimed at improving this performance, as will be discussed in more detail in the very next section.

### FPS improvement

The most effective strategy to mitigate the issue related to low frame per second rate is to convert the model from a `.pt` to an `.onnx` format. Such conversion allows the model to be used in conjunction with the DeepStream SDK library, making use of the GitHub repository available at <https://github.com/marcoslucianops/DeepStream-Yolo.git>. This resource is explicitly recommended by both the official NVIDIA site and Ultralytic. [33]

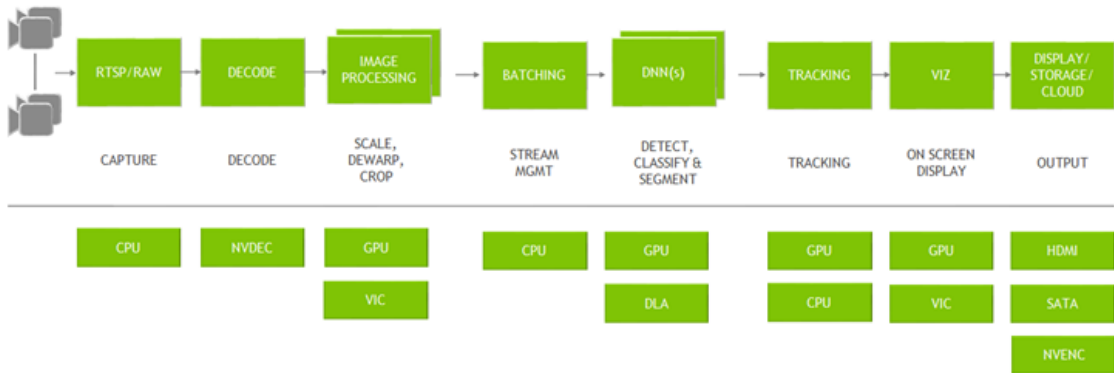
The JetPack SDK, which powers the Jetson modules, stands out as the most comprehensive solution, providing a complete development environment essential for creating end-to-end accelerated AI applications. This robust toolkit significantly reduces the time required to bring AI innovations to market.

JetPack includes Jetson Linux with bootloader, Linux kernel, Ubuntu desktop environment and a wide range of libraries dedicated to GPU computation acceleration, multimedia, graphics and computer vision. It also includes a repository of examples, comprehensive documentation, and a suite of development tools designed for both host computers and development kits. Significantly, JetPack supports advanced SDKs such as DeepStream for video analysis, Isaac for robotics, and Riva

for conversational AI. [34]

A notable feature is the unprecedented flexibility to run any Linux kernel, offering a broad spectrum of Linux-based distros from Jetson ecosystem partners. NVIDIA DeepStream is a streaming analytical toolkit for building artificial intelligence-powered applications. It uses artificial intelligence and computer vision to analyze streaming data from multiple sources, generating real-time insights.

The DeepStream SDK, with support for C/C++ and Python, offers hardware acceleration, bidirectional security between edge and cloud, and integrates libraries such as CUDA, TensorRT, Triton Inference Server. Optimized for NVIDIA GPUs, it can be implemented on Jetson, as in the case of this project, or on edge/datacenter GPUs. The graph architecture, based on GStreamer, includes more than 20 hardware-accelerated plugins. DeepStream handles output capture, decoding, preprocessing, inference, tracking, and visualization, offering output options such as streaming, local saving, or sending metadata to the cloud through various broker protocols. It is possible to visualize a schema from the Fig.4.16. [35]



**Figure 4.16:** DeepStream Overview graph architecture

The following are the main steps to enhance YOLOv5 in terms of FPS on NVIDIA Jetson Nano:

1. Download the DeepStream SDK folder from the official NVIDIA Jetson Nano website, considering compatibility with the version of JetPack currently in use. Specifically, for this study, the version of JetPack used is 4.6-b199. Therefore, it is essential to download DeepStream 6.0.1 for Jetson from the official website <https://developer.nvidia.com/embedded/deepstream-on-jetson-downloads-archived>.
2. Next, download the repository at <https://github.com/marcoslucianops/DeepStream-Yolo.git>, which facilitates the conversion of the file to the .onnx format. This conversion can be performed on the Jetson device but it was preferred to perform the conversion on Colab. This choice was made to avoid

additional load on Jetson Nano CPU, saving time and resources by taking advantage of Google GPU. As a result, the previously trained model with .pt extension is converted to .onnx.

3. At this point, move the .onnx model and the .txt file containing the labels to the previously downloaded folder on Jetson Nano.
4. Compile the CUDA library, selecting a version compatible with DeepStream SDK 6.0.1, previously downloaded. In this case, the version to compile will be CUDA 10.2.
5. After compiling the library, customize some configuration files, such as config\_infer\_primary\_yoloV5.txt, where the model name and number of classes are specified. Also, in the deepstream\_app\_config.txt file, specify the configuration file in use and modify the internal code to establish a direct connection to the camera for testing purposes. In the latter document, changes were made to several parameters in order to customize their adaptation to both the reference device and the previously developed network. The file in question is shown in Fig.4.17, while a detailed explanation of the corresponding sections is provided in Tab.4.8. In the specific context of [source0], a complete adaptation was made in order to enable the Astra camera. This adaptation involved the configuration of the parameters specified in Tab.4.9. [36] [37]

Group	Configuration Group
Application Group	Application configurations that are not related to a specific component.
Tiled-display Group	Tiled display in the application.
Source-list	Source URI provided as a list. There can be multiple sources.
OSD Group	Specify properties and modify the on-screen display (OSD) component that overlays text and rectangles on the frame.
Sink Group	Specify properties and modify behavior of sink components that represent outputs such as displays and files for rendering, encoding, and file saving. The pipeline can contain multiple sinks.

**Table 4.8:** Configuration Group in deepstream\_app\_config.txt

6. Finally, run the command `deepstream-app -c deepstream_app_config.txt`. The first time this command is run, the .engine file will be created; then the window will open from which the detection can be viewed.

The original model in .pt format was previously converted into two onnx file variants: one conforming to the FP32 standard and the other "simplified", which is characterized by a lighter structure, thus being more suitable for FP16 configuration. In addition, in order to generate the two files in .engine format, the correct setting of certain parameters, such as batch-size and network mode, to be configured in



Key	Meaning	Selected value
enable	Enables or disables the source	1
type	Choice of source type, other settings will be set or not as a result. 1: Camera (V4L2) 2: URI 3: MultiURI 4: RTSP 5: Camera (CSI)	1
camera-width	Width of frames to be requested from the camera, in pixels. Valid when type=1 or 5.	640
camera-height	Height of frames to be requested from the camera, in pixels. Valid when type=1 or 5.	480
camera-fps-n	Numerator part of a fraction specifying the frame rate requested by the camera, in frames/sec. Valid when type=1 or 5.	30
camera-fps-d	Denominator part of a fraction specifying the frame rate requested from the camera, in frames/sec. Valid when type=1 or 5.	1
camera-v4l2-dev-node	Number of the V4L2 device node. Valid when the type setting (type of source) is 1.	1

**Table 4.9:** Configuration [source]

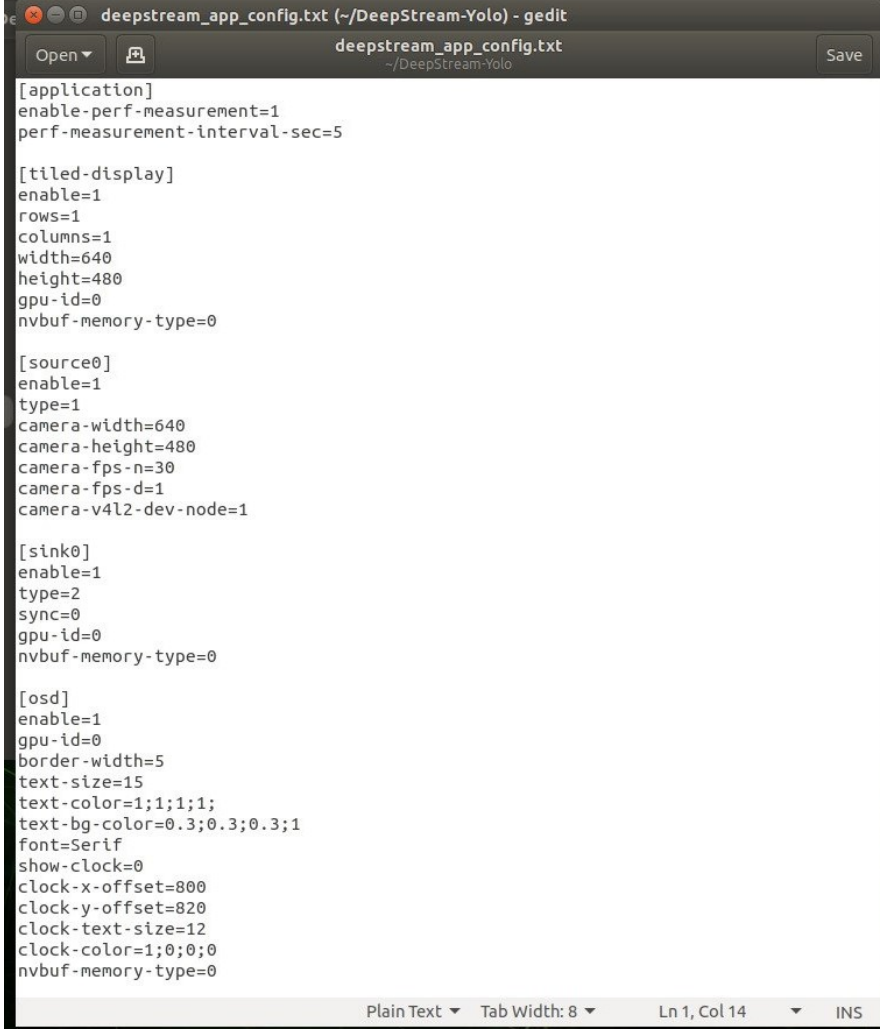
a different way, assumes particular importance. [37]The documents for the two configurations can be examined in the figure

model-engine-file	batch-size	network-mode
model_b1_gpu0_fp32.engine	1	0
model_b1_gpu0_fp16.engine	1	2

**Table 4.10:** YOLOv5.onnx FPS on Jetson Nano with DeepStream

Therefore, as mentioned just before, .engine file was generated in the FP16 and FP32 floating-point precision formats. These formats, known as half-precision and single-precision floating-point, respectively, play a central role in defining the numerical representation used during computational operations.

The FP32 format enjoys wide support and is a common choice in science and artificial intelligence applications. It employs 32 bits for the representation of a



```
deepstream_app_config.txt (~/.DeepStream-Yolo) - gedit
deepstream_app_config.txt
~/.DeepStream-Yolo
Save

[application]
enable-perf-measurement=1
perf-measurement-interval-sec=5

[tiled-display]
enable=1
rows=1
columns=1
width=640
height=480
gpu-id=0
nvbuf-memory-type=0

[source0]
enable=1
type=1
camera-width=640
camera-height=480
camera-fps-n=30
camera-fps-d=1
camera-v4l2-dev-node=1

[sink0]
enable=1
type=2
sync=0
gpu-id=0
nvbuf-memory-type=0

[osd]
enable=1
gpu-id=0
border-width=5
text-size=15
text-color=1;1;1;1;
text-bg-color=0.3;0.3;0.3;1
font=Serif
show-clock=0
clock-x-offset=800
clock-y-offset=820
clock-text-size=12
clock-color=1;0;0;0
nvbuf-memory-type=0

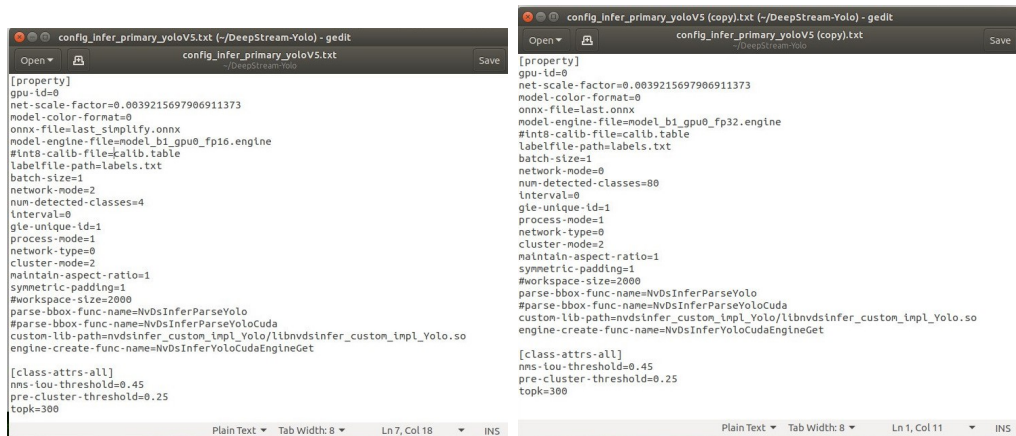
Plain Text Tab Width: 8 Ln 1, Col 14 INS
```

Figure 4.17: deepstream\_app\_config.txt file

floating-point number, distributed into 1 bit for the sign, 8 bits for the exponent, or magnitude, and 23 bits for the mantissa, or precision.

In contrast, the FP16 format is frequently adopted in deep learning applications due to its lower memory occupancy and allegedly higher computational speed compared to FP32. FP16 uses 16 bits, including 1 bit for sign, 5 bits for exponent and 10 bits for precision. [38]

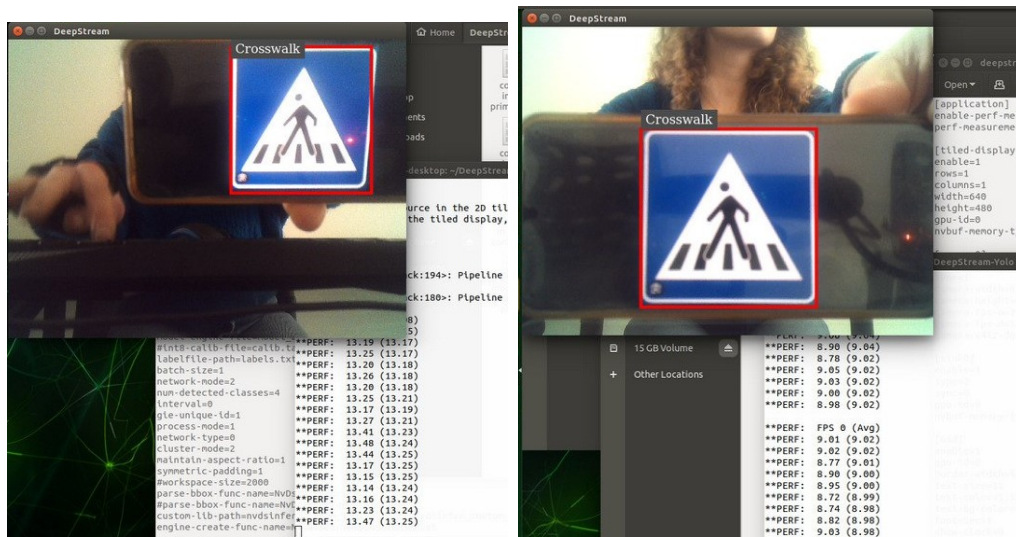
The comparison between these two formats was conducted in order to evaluate the balance between precision and FPS. Regarding accuracy, both configurations demonstrated excellent signal recognition, as illustrated in Fig.4.19. A reduction in image resolution is observed, consistent with expectations and in line with what



**Figure 4.18:** config\_infer\_primary\_yoloV5.txt file for FP16 vs FP32 configuration

was expected; however, the detection capability is not compromised. This result can probably be attributed to the robustness of the previously trained model.

Regarding FPS, the FP16 configuration manifested a superior performance in agreement with theoretical predictions, reaching nearly 14 FPS, as can be seen from the results reported in Tab.4.11.



**Figure 4.19:** Detection of a signal FP16 vs FP32 configuration

Configuration	.engine file	FPS
	FP16	13.5
	FP32	9.5

**Table 4.11:** YOLOv5.onnx FPS on Jetson Nano with DeepStream

## 4.4 Metrics of the final chosen model

Looking to design future developments on the Jetson Nano device, the YOLOv5 model comes closest to an ideal solution. Specifically, the model chosen is the one trained with a number of epochs equal to 300 and a batch size of 32, subsequently converted to .onnx and .engine formats using FP16 precision. This configuration fully meets the identified requirements, avoiding overloads on the RAM of the Jetson Nano device and ensuring adequate FPS, considering that the camera will be attached to a rover with a maximum speed of 30 km/h.

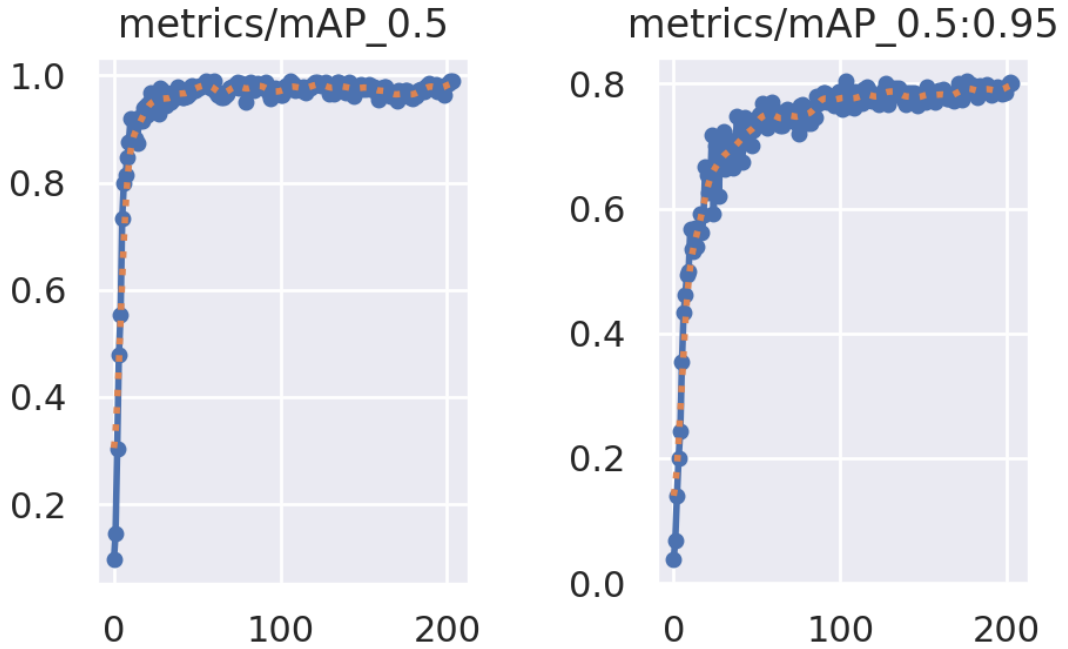
In the context of this evaluation, a detailed analysis of some metrics associated with this model is proposed. In particular, the focus is on the graphs of Mean Average Precision (mAP) at 0.5 and 0.5:0.95, presented in Figure 4.20. These graphs show stability at values of about 1 and 0.8, respectively, as early as epoch 200.

The metrics mAP0.5 and mAP0.5:0.95 are crucial in the context of object detection and evaluate the average accuracy at different Intersection over Union (IoU) thresholds. mAP0.5 represents the average accuracy at an IoU threshold of 0.5, while mAP0.5:0.95 represents the average accuracy over IoU thresholds ranging from 0.5 to 0.95. These metrics are central to evaluating the accuracy of object detection models against real data and in fact give us evidence that this model is good and fully meets these requirements. [39] [40] It is possible to have a visual feedback in the Fig.4.20.

The IoU parameter is defined as the ratio of the intersection area to the union area between the predicted and actual bounding box. Examination of this formula clearly reveals that Intersection over Union is a simple ratio. Numerator and denominator quantify the area of overlap and the area encompassed by both bounding boxes, respectively. [41]

The precision-recall curve, shown in Figure 4.21, provides a complete picture of the model performance at different thresholds. This type of curve represents accuracy versus recall, helping to visualize how the choice of threshold affects the performance of the classifier. [42]

The accuracy metric, commonly known as positive predictive value, is expressed by the following mathematical formula:



**Figure 4.20:** MAP0.5 and MAP0.5:0.95 of YOLOv5 model with  $epochs = 300$  and  $batch - size = 32$

$$Precision = \frac{TP}{TP + FP} \quad (4.4)$$

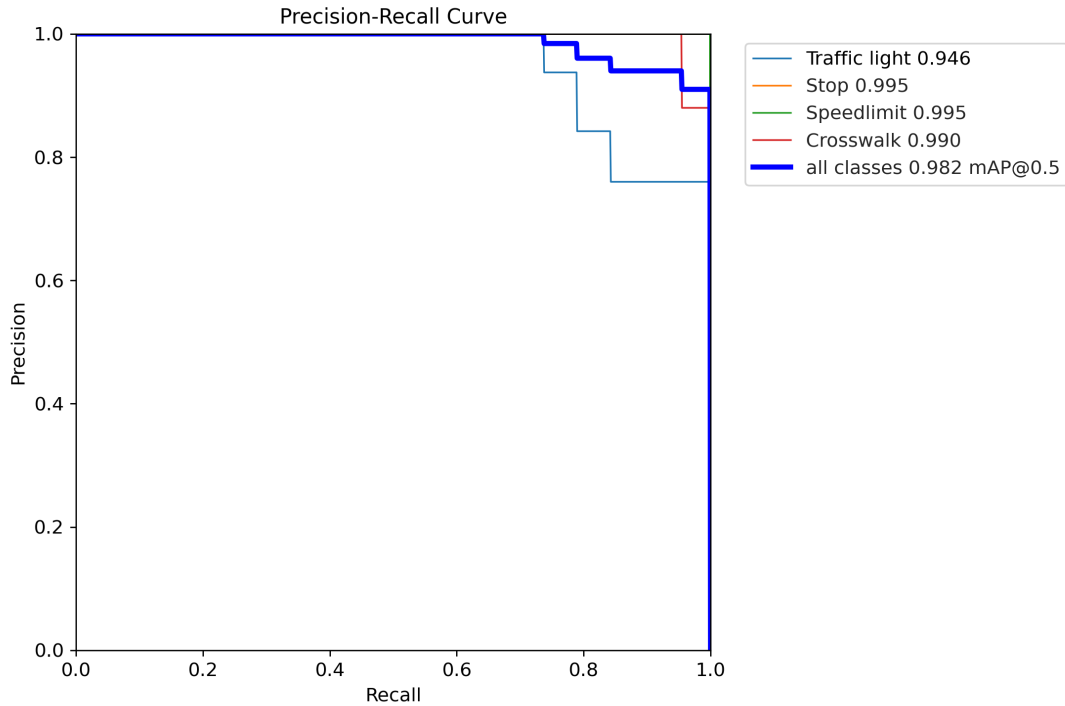
Where TP represents the number of true positives and FP indicates the number of false positives. Precision can be interpreted as the proportion of positive predictions that are actually relevant to the positive class.

Recall, also referred to as sensitivity, is defined by the following expression:

$$Recall = \frac{TP}{TP + FN} \quad (4.5)$$

Where TP is the number of true positives and FN corresponds to the number of false negatives. Recall can be interpreted as the fraction of positive predictions relative to all positive instances in the dataset.

The F1 score, as a measure of model accuracy that takes into account precision and recall, is presented in Figure 4.22. This metric, varying between 0 and 1, represents the weighted harmonic mean of precision and recall. The F1 score considers both false-positive and false-negative rates, providing an overall indicator of model performance. The formula for F1 score is:

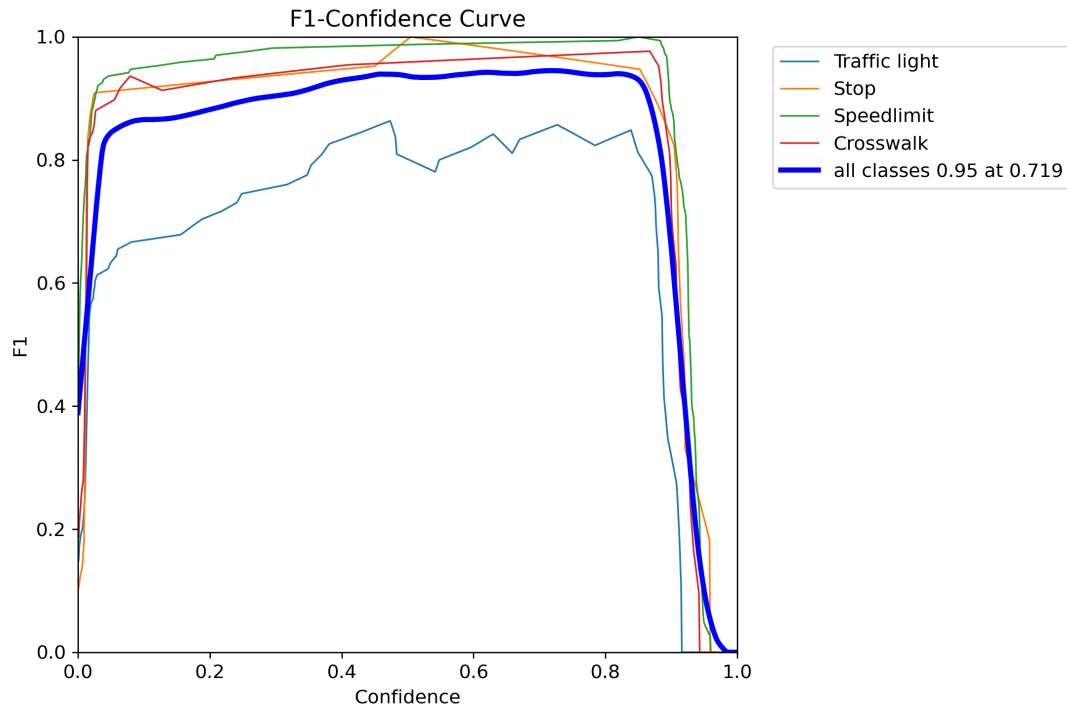


**Figure 4.21:** PR curve YOLOv5 model with  $epochs = 300$  and  $batch - size = 32$

$$F1 = \frac{2 * Precision * Recall}{Precision + Recall} \quad (4.6)$$

The overall analysis of the metrics reveals excellent results, confirming the validity of the YOLOv5 model under the specified conditions. It should be noted that some differences between classes, as can be seen in both the images 4.21 and 4.22, can be attributed to various factors, such as an imbalance in the number of training images and the inherent complexity of the classes themselves, as in the case of the light blue line, that represents the traffic light class. In fact, this class has significantly less performance than the others, although still good. However, this does not heavily affect the final result, which is still very good, partly due to the results obtained from the training of the other classes. Therefore, the overall curves demonstrate a satisfactory performance of the model, solidifying its position as a superior solution.

The final evaluation of the YOLOv5 model focuses on its loss function, which consists of three main components. The results on the training sets, displayed in Fig.4.23, show three types of loss: objectivity, bounding box and classification. Objectivity measures the probability of an object presence in a proposed region,



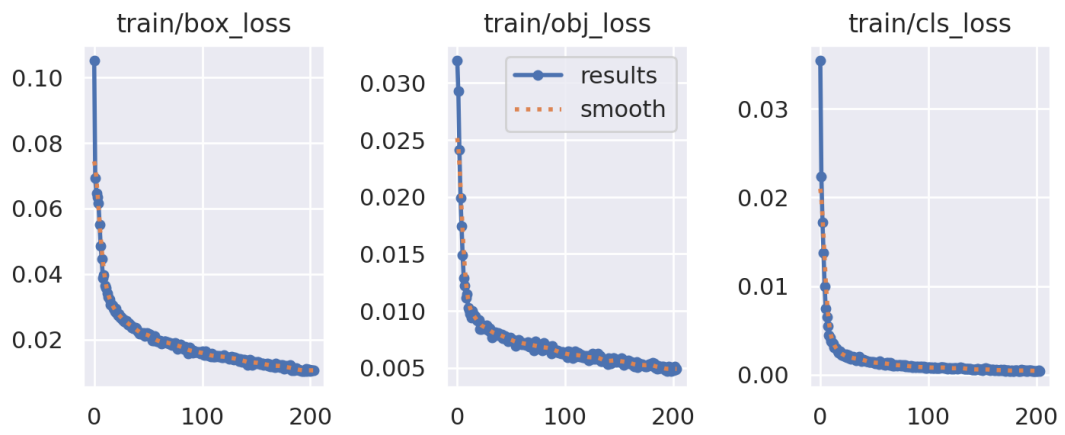
**Figure 4.22:** F1 curve YOLOv5 model with  $epochs = 300$  and  $batch - size = 32$

while bounding box loss assesses the accuracy of object location and coverage. Classification loss reflects correctness in predicting the class of the object. [43]

In examining the model loss curves, there is a general trend of convergence after the first 100 training epochs. At the initial stage, all loss functions manifest a significant decrease, highlighting rapid learning of the model from the training data.

Upon reaching 100 epochs, a stabilization of the curves is observed, suggesting the achievement of an optimal convergence point. It is relevant to note that, in the case of classification loss, the settling point occurs earlier, approximately as early as epoch 50.

The results obtained are significant, recording extremely low values for all loss function categories considered, indicating a high level of effectiveness during the training phase.



**Figure 4.23:** Train Loss functions YOLOv5 model with  $epochs = 300$  and  $batch - size = 32$



## Chapter 5

# Test results on the analysed models

This section is devoted to analyzing the tests conducted using the models reviewed in the previous chapter. The goal is to rigorously compare the two models, employing the same signals and evaluation environment for testing.

### 5.1 SSD MobileNet

#### 5.1.1 Jetson-inference folder

Several tests were performed in order to evaluate the performance of the model trained by transfer learning. As previously mentioned, the dataset employed is composed exclusively of a single class.

The "workers" parameter was consistently maintained at a value of 0. Otherwise, increasing this parameter results in an almost instantaneous crash of the process, accompanied by an error resulting from a lack of CPU resources. This issue is highlighted in the Fig.5.1.

```
__error_if_any_worker_fails()  
RuntimeError: DataLoader worker (pid 688) is killed by signal: Bus error. It is possible that data loader's workers are out of shared memory. Please try to raise your shared memory limit.  
root@jetson-desktop:/jetson-inference/python/training/detection/ssd#
```

**Figure 5.1:** Dataloader Worker killed

Therefore, different models were tried with different batch sizes and epochs, as shows the Tab.5.1 .

It would have been more appropriate to conduct network training for a significantly larger number of epochs; under normal conditions, extending the process

batch-size	epochs	training time	mAP
2	10	45 min	82%
2	30	65 min	89%
4	10	80 min	93%
8	10	120 min	97%
16	10	killed process	not available

**Table 5.1:** MAP results according to different batch-size and epochs

to 100 epochs would have been justified. However, the limitations of the Jetson’s CPU had to be taken into account, despite the expansion of RAM. Nevertheless, the use of the transfer learning method produced satisfactory results, especially considering the limited amount of epochs employed.

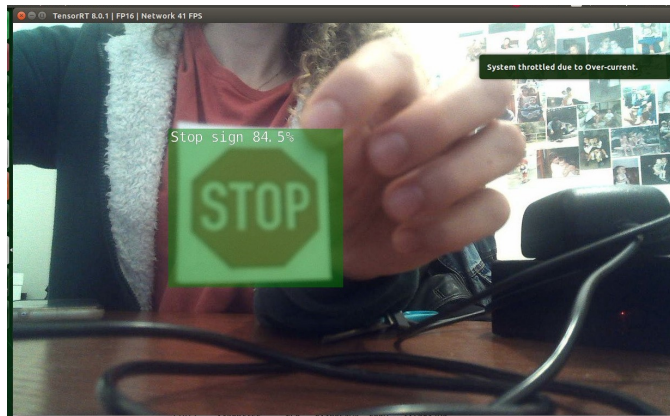
The tests were performed empirically, using standard paper signals placed in front of the camera. In this context, the frame per second of the camera was remarkably satisfactory, ranging between 38 and 40 FPS. FPS represents the rate at which a system, such as a camera or graphics software, can generate, display or process frames in one second. In the context of video and images, a higher FPS value generally results in a smoother and more realistic experience.

The number of traffic signals was also increased to test the system and evaluate its ability to correctly distinguish between different signals. In all cases, the network demonstrated adequate response.

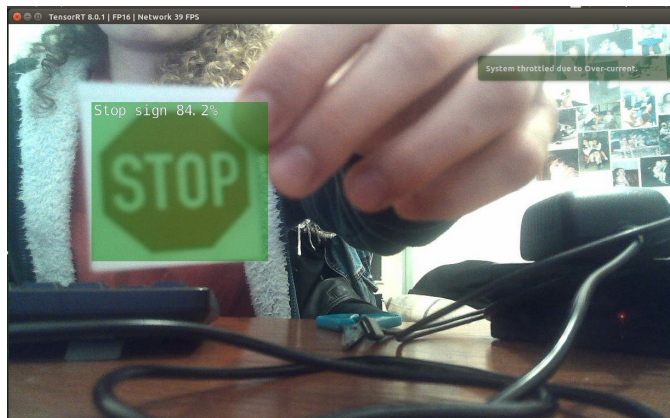
A series of images illustrating the detection of stop signals, obtained by using models with variable parameters, follows. In accordance with expectations, improved results are observed with higher batch sizes. Regardless, the detection is implemented promptly. Even experimenting with right-to-left movements or changing the distance between the signal and the camera, signal recognition persists, keeping the detection procedure stable over time. This robustness is attributable, in part, to the smoothness of the acquisition provided by the camera.

The images 5.2, 5.3 and 5.4 are the result of tests using models trained with different parameters. Specifically, the first model was trained with *batch-size* = 2 and *epochs* = 30, the second with *batch-size* = 4 and *epochs* = 10, and the last with *batch-size* = 8 and *epochs* = 10. As highlighted in the Tab.5.1, the optimal configuration in this context manifests with *batch-size* = 8 and *epochs* = 10, characterizing itself by an improved mAP. This is evidenced in the image 5.4, where the confidence level is greater than 90%.

In various experiments, the use of the stop signal along with other signals has been conducted in order to demonstrate the ability to accurately identify only the said signal, without incurring confusion with the others. The results obtained were remarkable, as the system is consistently able to correctly identify the stop signal



**Figure 5.2:** *batch - size = 2* and *epochs = 30*



**Figure 5.3:** *batch - size = 4* and *epochs = 10*

while managing to ignore the other signals with high confidence, as it is possible to see in Fig.5.5.

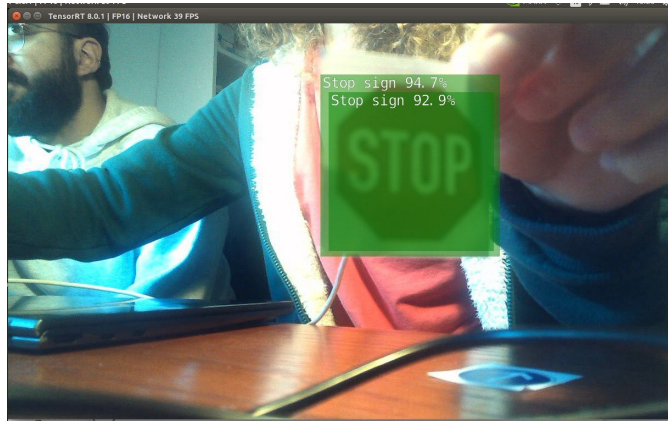


Figure 5.4: *batch - size = 8* and *epochs = 10*

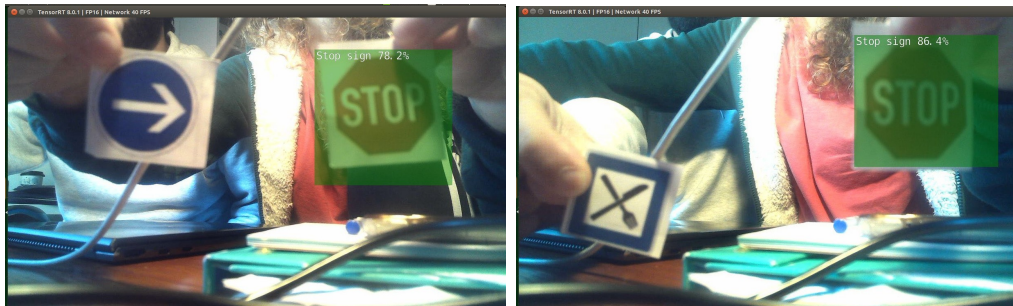


Figure 5.5: Stop sign detection at the expense of other traffic signals

### 5.1.2 SSD MobileNet with COCO128

Regarding this section, expectations were geared toward excellent results, considering the use of a pre-trained dataset such as COCO128. However, an additional issue emerged, likely attributable to the limited memory capacity of the Jetson.

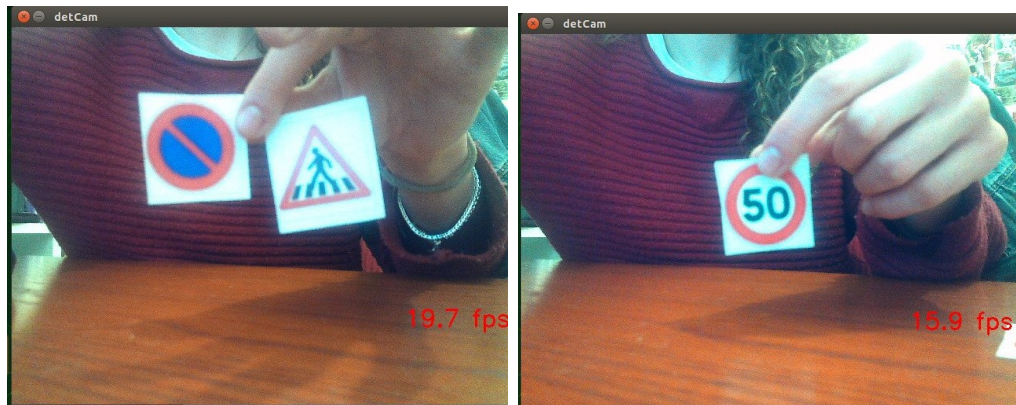
It was observed that categories related to "street signs", such as "One Way Sign", "No Parking Sign" or "Do Not Enter Sign", were missing from the COCO dataset. Despite several attempts with different street signs, the model proved unable to detect them. In contrast, the recognition of signs such as "stop sign" and "traffic light" was remarkably efficient, maintaining a good level of confidence level even under different lighting conditions.

In this context, there is a decrease in the FPS rate compared to the previous section, standing between 15 and 20 FPS. However, this result is still considered satisfactory for an autonomous driving application.

As previously outlined, the adoption of this approach was motivated by the prospect of taking advantage of the three classes relevant to the street category

offered by the COCO dataset. However, after an initial series of tests, it became apparent that the "street sign" section was only present among the labels, without translating into concrete results. Although numerous attempts were made using a variety of street signs falling into this category, none were detected. Some tries are shown in the Fig.5.6. This finding suggests that the model in question did not receive adequate training for the "street sign" class.

This judgment is supported by the fact that, in contrast, the other two classes, namely "stop sign" and "traffic light", are detected quickly and with remarkable confidence.



**Figure 5.6:** Street Sign missed detection

Using the above approach, the identification of the stop signal is achieved with timeliness and clarity, without encountering specific problems. An example can be found in the Fig.5.7, in which is possible to see a confidence level that can reach 98%.

In addition, it is possible to observe, as evidenced in Figures 5.7 and 5.8, the system ability to clearly identify the signal, both when it is in close proximity and when it is located at a greater distance, approximately 30 cm from the camera.

In addition, the model demonstrates the ability to discern the stop signal from other signals that may have similarities, as shown in Fig.5.9, thus lending an additional degree of accuracy to its performance.

A further experiment conducted is illustrated in Fig5.10, in which obstacles were placed between the camera and the traffic signal under consideration. In the case where only a limited portion of the signal is covered, the ability to detect the signal persists; however, once recognition has occurred and subsequently one of the letters of the "STOP" signal is fully covered, the ability to identify the signal is impaired.

Traffic light detection is effective under a variety of lighting conditions, also demonstrating remarkable accuracy in identifying a variety of traffic light types.





Figure 5.7: Detection stop sign closely



Figure 5.8: Detection stop sign from a distance

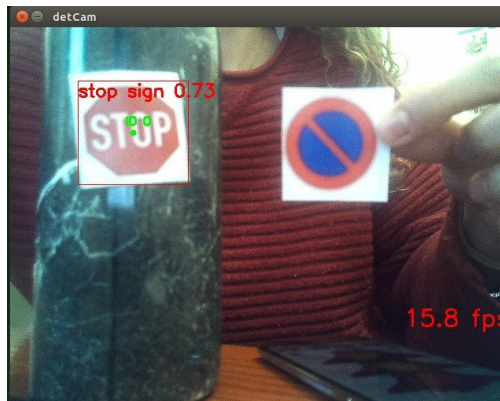
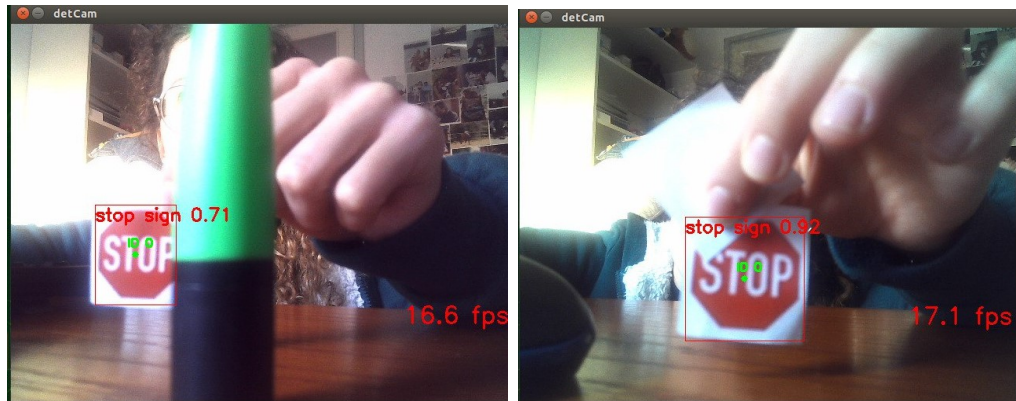
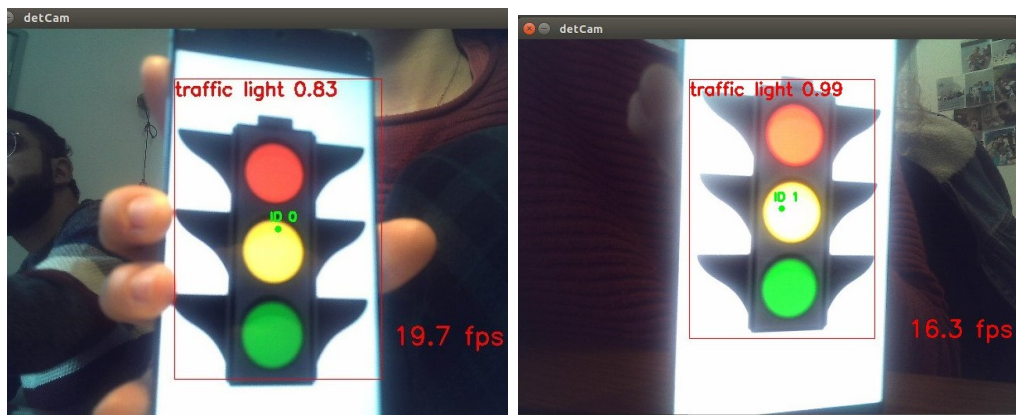


Figure 5.9: Detection stop sign next to another traffic signal



**Figure 5.10:** Stop signal detection with obstacles

Its confidence level remains high regardless of variations in light or diversity of traffic light patterns considered, as it is possible to see in the pictures in Fig.5.11 and Fig.5.12.



**Figure 5.11:** Traffic Light detection

In Fig.5.13 is presented a test in which both targets were included. Again, the model demonstrates effectiveness in detecting both the stop signal and the traffic light with timely and remarkable confidence levels.

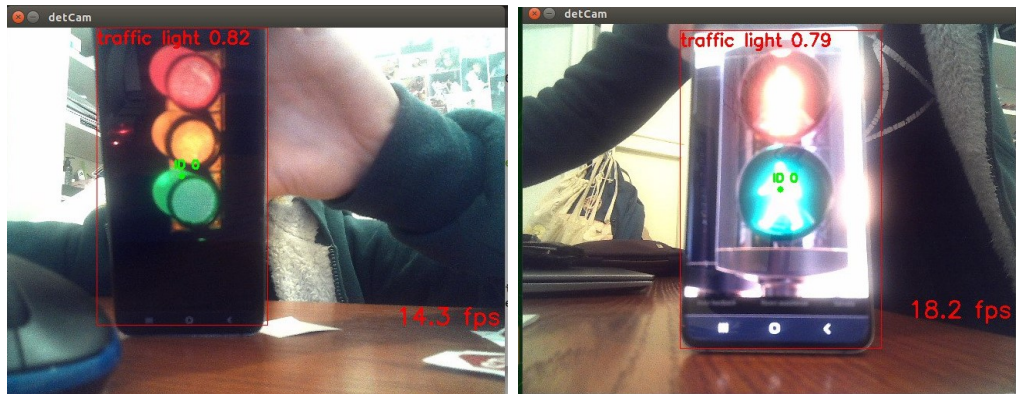


Figure 5.12: Traffic Light detection with other types of semaphores

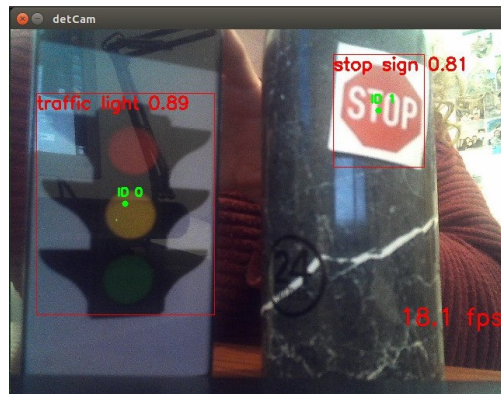


Figure 5.13: Traffic light and stop sign detection

## 5.2 YOLOv5

This section is reserved for examining the tests conducted on the customized YOLOv5 model using the Astra Camera on Jetson Nano, employing standard traffic signals in order to evaluate the effectiveness of the system. At the initial stage, individual testing of each signal belonging to the four reference classes was performed in order to evaluate its detection capability specifically and separately of the model that reported the best mAP result, i.e.  $epochs = 300$  and  $batch - size = 32$ , as can be seen in Fig.5.14.

Next, the confidence level of the model in its ability to identify the "Speed Limit" traffic signal was evaluated at two different distances from that seen in Fig.5.14: one close, approximately  $3cm$  from the camera, and one further away, approximately  $40cm$  from the camera. In both cases, there was appreciable accuracy in identifying the signal. However, as expected, the accuracy at a greater distance was lower and showed a gradual increase as the camera approached the signal. The confidence



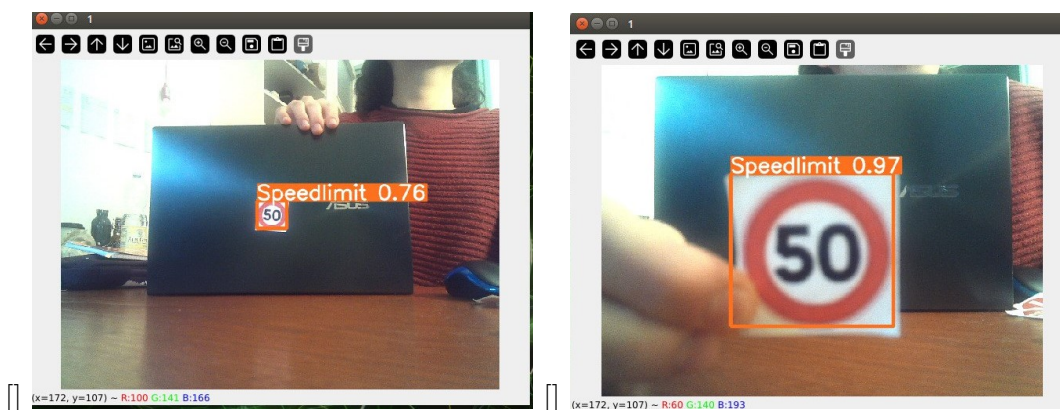


**Figure 5.14:** Tests performed with  $epochs = 300$  and  $batch - size = 32$  with signals through the camera

levels recorded at the three different distances, visible in Fig.5.14 and Fig.5.15 are detailed in Table 5.2.

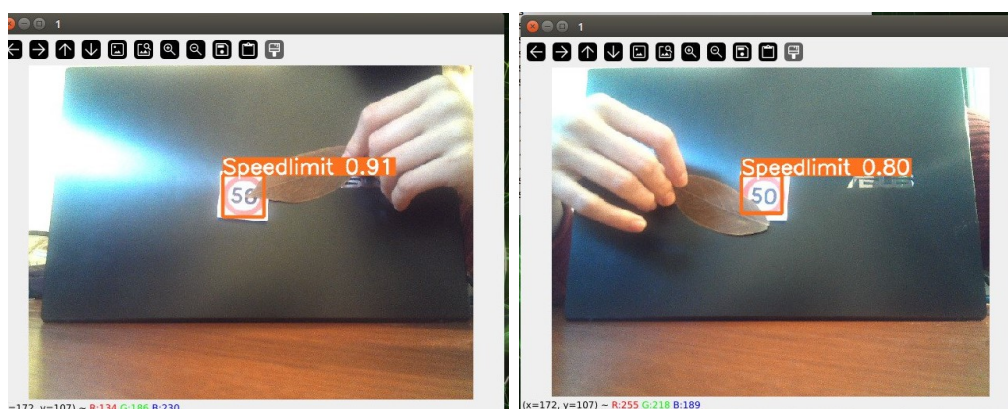
Distance between the road sign and the camera	Confidence value
3cm	97%
20cm	89%
40cm	76%

**Table 5.2:** Confidence values variation according to the distance between road sign and camera



**Figure 5.15:** Speed Limit signal at different distances

Further experiments aimed at evaluating the accuracy of the model were conducted by introducing obstacles placed near the traffic signal. The goal of this approach is to simulate real situations in which recognition might be compromised by replicating conditions of partial visibility of the signal. In this context, the results obtained are encouraging, since despite covering a significant portion of the signal, confidence levels between 80% and 95% could be achieved, as can be seen from Fig.5.16.



**Figure 5.16:** Detection of Speed Limit signal with obstacles

A further procedure adopted to evaluate the accuracy of the model was the introduction of an additional signal, not included in the dataset but characterized by considerable similarities in color and shape to the "Stop" and "Speed Limit" signals. As highlighted in Fig.5.17, the identification of the appropriate signals was remarkably effective. The model was able to detect the two classified signals with high confidence, completely ignoring the additional signal.



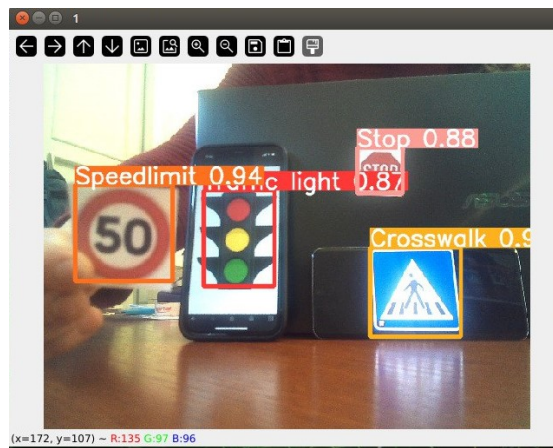
**Figure 5.17:** Detection of Stop and Speed Limit signals near a sign similar in shape and color

A further test conducted involved the analysis of traffic lights of less conventional design and with various inclinations, in order to explore any limitations or difficulties of the model in identifying them. As depicted in Fig.5.18, the results obtained in this context were highly positive, showing a remarkable ability of the model in dealing with situations where traffic lights have unconventional features or particular inclinations.



**Figure 5.18:** Tests with different type of traffic lights

The last test performed involved simulating the simultaneous detection of different classes of traffic signals by placing these signals at various distances from the camera. Also in this circumstance, the model ability in signal detection manifested highly positive results, with confidence values ranging from 88% percent to 96%.



**Figure 5.19:** Detection of all the signals all at once



# Chapter 6

## Conclusions

### 6.1 Overview of the analysed models

The objective of this research was the development and comparison of two models, SSD MobileNet and Yolov5, for the realization of an object detection system targeted at traffic signals. The experimentation was conducted using the NVIDIA Jetson Nano device, as part of a larger project aimed at developing a drone capable of simulating autonomous driving.

The dataset employed includes four classes of traffic signals: "Stop," "Speed Limit," "Traffic Light," and "Crosswalk." However, the RAM and memory limitations of the NVIDIA device posed a significant challenge in implementing complex models.

In response to these limitations, differential approaches had to be taken for the two models examined. The initial approach involved training the models on an online GPU, specifically Google Colab, followed by adapting them to the context of the Jetson Nano to ensure effective handling of external stimuli, expressed through the key parameters of frames per second and mean average precision.

Regarding SSD MobileNet, an attempt to convert the model developed on Google Colab into an NVIDIA device-compatible .onnx format was unsuccessful. Therefore, a simplified model was trained on Jetson Nano, focusing exclusively on the "Stop" class. Limited resources, particularly RAM, imposed compromises on the number of epochs and other parameters; however, the simplified model demonstrated satisfactory efficiency, with a speedup of about 40 FPS. Unfortunately, the issue concerning operation restricted to only one category of traffic signals persists. Despite achieving satisfactory mAP values of around 98%, the outcomes initially desired during the study could not be fully achieved.

As for Yolov5, more positive results were obtained. The model was successfully trained on Google Colab, and the results were extended to all classes. However, transferring the model to the NVIDIA device generated FPS below the minimum

requirements. Therefore, it was necessary to convert the model to .onnx format, exploring FP32 and FP16 configurations. The first with a definitely better resolution but reporting lower FPS, the other with a slightly lower resolution but reporting higher FPS, which are acceptable in a context where the ultimate goal is autonomous driving of a drone reporting a maximum speed of  $30\text{km}/h$ . Moreover, the resolution of the second model, although lower, did not go to compromise the detection of road signals in any way. Therefore, the final choice was an FP16 .engine model with  $epochs = 300$  and  $batch - size = 32$ , providing an acceptable balance between resolution and speed, approximately 14 FPS.

In conclusion, this research has successfully addressed the challenges associated with implementing an object detection system on an embedded device, presenting promising results, although some limitations related to device resources persist. The considerations that emerged provide a basis for the future development of applications related to autonomous driving.

## 6.2 Improvements and future works

The present work is a starting point that is susceptible to further development and improvement. One option for enhancing the robustness of the model could involve expanding the dataset to include a more diverse range of traffic signals, although the potential impact on the computational complexity of the dataset itself should be carefully considered.

In addition, methodologies to further increase the frames per second of the YOLO model can be explored, possibly considering the use of alternative versions, while still maintaining high definition to ensure efficient object detection. It is crucial to take into account the limitations imposed by the hardware, accepting the fact that the performance achievable in a simulated environment may differ from what is actually achievable.

As previously mentioned, the present work is part of a larger context geared toward the development of a drone capable of using object detection for navigation in a road driving environment. Therefore, future research directions could include:

- Implement a Cruise Control and Lane Keeping system that manifests synergistic coherence with signal recognition.
- Connect the NVIDIA Jetson Nano to a device that can provide mechanical pulses for wheel management, implementing low-level control systems for eMotors, sensors and actuators.
- Improve overall camera management and integrate the use of a LIDAR to track the distance to potential obstacles, addressing the challenges of estimating the distance and angle of objects.

- Develop a Traction Control system (including possibly ASR and/or Torque Vectoring) for traction and braking.
- Implement the State of Health (SoH) and State of Charge (SoC) of the battery and develop the interface with the Battery Management System and Advanced Driver Assistance System functions.

# Bibliography

- [1] Narayana Darapaneni, Pratosh Raj R, Anwesh Reddy Paduri, Emmanuel Anand, Kumar Rajarathinam, Prem Thomas Eapen, Sethuraman. K, and Sharath Krishnamurthy. «Autonomous Car Driving Using Deep Learning». In: *2021 2nd International Conference on Secure Cyber Computing and Communications (ICSCCC)*. 2021, pp. 29–33. DOI: 10.1109/ICSCCC51823.2021.9478090 (cit. on p. 4).
- [2] Jelena Kocić, Nenad Jovičić, and Vujo Drndarević. «An End-to-End Deep Neural Network for Autonomous Driving Designed for Embedded Automotive Platforms». In: *Sensors* 19.9 (2019). DOI: 10.3390/s19092064. URL: <https://www.mdpi.com/1424-8220/19/9/2064> (cit. on p. 5).
- [3] Enrique D. Tejada and Paul A. Rodriguez. «Moving object detection in videos using principal component pursuit and convolutional neural networks». In: *2017 IEEE Global Conference on Signal and Information Processing (GlobalSIP)*. 2017, pp. 793–797. DOI: 10.1109/GlobalSIP.2017.8309069 (cit. on p. 5).
- [4] Abhishek Sarda, Shubhra Dixit, and Anupama Bhan. «Object Detection for Autonomous Driving using YOLO [You Only Look Once] algorithm». In: *2021 Third International Conference on Intelligent Communication Technologies and Virtual Mobile Networks (ICICV)*. 2021, pp. 1370–1374. DOI: 10.1109/ICICV50876.2021.9388577 (cit. on p. 6).
- [5] Boyu Ao, Jiachang Ren, and Chen Guo. «Impact of Image Corruptions on the Reliability of Traffic Sign Recognition Using Machine Learning Technique». In: ICCDA 2019. Kahului, HI, USA: Association for Computing Machinery, 2019, pp. 6–8. ISBN: 9781450366342. DOI: 10.1145/3314545.3314547. URL: <https://doi.org/10.1145/3314545.3314547> (cit. on p. 6).
- [6] Xianjian Jin, Zhiwei Li, and Hang Yang. «Pedestrian Detection with YOLOv5 in Autonomous Driving Scenario». In: *2021 5th CAA International Conference on Vehicular Control and Intelligence (CVCI)*. 2021, pp. 1–5. DOI: 10.1109/CVCI54083.2021.9661188 (cit. on p. 7).



- 
- [7] A N Aneesh, Linu Shine, R Pradeep, and V Sajith. «Real-time Traffic Light Detection and Recognition based on Deep RetinaNet for Self Driving Cars». In: *2019 2nd International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*. Vol. 1. 2019, pp. 1554–1557. DOI: 10.1109/ICICICT46008.2019.8993293 (cit. on p. 8).
- [8] Fabio Gemelli. «Guida autonoma: a che punto sono le varie Case auto». In: *Moto1.com* (2022). URL: <https://it.motor1.com/news/569720/guida-autonoma-case-auto-2022> (cit. on p. 8).
- [9] Kevin P. Murphy. *Machine Learning: A Probabilistic Perspective*. 1st. Cambridge, Massachusetts: The MIT Press, 2012. ISBN: 978-0262018029 (cit. on pp. 11, 12).
- [10] Kuruva Satya Ganesh. «What’s The Role Of Weights And Bias In a Neural Network?» In: (July 2020) (cit. on p. 11).
- [11] Yulia Gavrilova. «Convolutional Neural Networks for Beginners». In: (Aug. 2021) (cit. on pp. 11, 12).
- [12] Chang-Yu Cao, Jia-Chun Zheng, Yi-Qi Huang, Jing Liu, and Cheng-Fu Yang. «Investigation of a Promoted You Only Look Once Algorithm and Its Application in Traffic Flow Monitoring». In: *Applied Sciences* 9.17 (2019). ISSN: 2076-3417. DOI: 10.3390/app9173619. URL: <https://www.mdpi.com/2076-3417/9/17/3619> (cit. on p. 13).
- [13] Peiyuan Jiang, Daji Ergu, Fangyao Liu, Ying Cai, and Bo Ma. «A Review of Yolo Algorithm Developments». In: *Procedia Computer Science* 199 (2022). The 8th International Conference on Information Technology and Quantitative Management (ITQM 2020–2021): Developing Global Digital Economy after COVID-19, pp. 1066–1073. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2022.01.135>. URL: <https://www.sciencedirect.com/science/article/pii/S1877050922001363> (cit. on p. 13).
- [14] Do Thuan. *EVOLUTION OF YOLO ALGORITHM AND YOLOV5: THE STATE-OF-THE-ART OBJECT DETECTION ALGORITHM*. Bachelor’s Thesis. 2021 (cit. on p. 14).
- [15] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. *YOLOv4: Optimal Speed and Accuracy of Object Detection*. 2020. arXiv: 2004.10934 [cs.CV] (cit. on p. 14).
- [16] Erich KOHMANN. *Tecniche di deep learning per l’object detection*. Corso di Laurea in Informatica. 2019 (cit. on p. 15).
- [17] Vidish Mehta. *Object Detection using SSD Mobilenet V2*. 2021. URL: <https://vidishmehta204.medium.com/object-detection-using-ssd-mobilenet-v2-7ff3543d738d> (cit. on p. 17).

- [18] Gaurav Singhal. *Transfer Learning in Deep Learning Using Tensorflow 2.0*. 2020. URL: <https://www.pluralsight.com/guides/transfer-learning-in-deep-learning-using-tensorflow-2.0> (cit. on pp. 17, 18).
- [19] Rakkshab Iyer, Kevin Bhensdadiya, and Priyansh Ringe. «Comparison of YOLOv3, YOLOv5s and MobileNet-SSD V2 for Real-Time Mask Detection». In: *International Journal of Research in Engineering and Technology* (July 2021), pp. 2395–0056 (cit. on pp. 18, 19).
- [20] *What is MobileNet SSD v2?* URL: <https://roboflow.com/model/mobilenet-ssd-v2> (cit. on p. 18).
- [21] «Object Detection Using YOLO And Mobilenet SSD». In: (Sept. 2022) (cit. on p. 18).
- [22] N Sabina, M. P Aneesa, and P. V Haseena. «Object Detection using YOLO And Mobilenet SSD: A Comparative Study». In: *INTERNATIONAL JOURNAL OF ENGINEERING RESEARCH & TECHNOLOGY (IJERT)* 11 (06 June 2022) (cit. on p. 18).
- [23] Martinus Grady Naftali, Jason Sebastian Sulistyawan, and Kelvin Julian. *Comparison of Object Detection Algorithms for Street-level Objects*. 2022. arXiv: 2208.11315 [cs.CV] (cit. on p. 18).
- [24] *Google Colab per il Machine Learning: cos'è e come si usa*. 2019. URL: <https://www.html.it/articoli/google-colab-per-il-machine-learning-cose-e-come-si-usa/> (cit. on p. 22).
- [25] *Un benvenuto a Colaboratory*. –. URL: <https://colab.research.google.com/?hl=it#scrollTo=ufxBm1yRnruN> (cit. on p. 22).
- [26] *Utilizzo del formato SavedModel*. 2022. URL: [https://www.tensorflow.org/guide/saved\\_model?hl=it](https://www.tensorflow.org/guide/saved_model?hl=it) (cit. on p. 28).
- [27] *Jetson-inference*. 2019. URL: <https://github.com/dusty-nv/jetson-inference> (cit. on p. 28).
- [28] *Jetson-inference documentation - Transfer Learning*. 2019. URL: <https://github.com/dusty-nv/jetson-inference/blob/master/docs/pytorch-transfer-learning.md> (cit. on p. 28).
- [29] *Jetson-inference documentation - Docker*. 2019. URL: <https://github.com/dusty-nv/jetson-inference/blob/master/docs/aux-docker.md> (cit. on p. 29).
- [30] *Jetson-inference documentation - Training the SSD-Mobilenet Model*. 2019. URL: <https://github.com/dusty-nv/jetson-inference/blob/master/docs/pytorch-ssd.md> (cit. on p. 29).

- [31] *ONNX Model Zoo*. —. URL: <https://github.com/onnx/models> (cit. on p. 31).
- [32] *Dati personalizzati per la formazione*. 2023. URL: [https://docs.ultralytics.com/it/yolov5/tutorials/train\\_custom\\_data/](https://docs.ultralytics.com/it/yolov5/tutorials/train_custom_data/) (cit. on p. 34).
- [33] *Distribuire su NVIDIA Jetson utilizzando TensorRT e DeepStream SDK*. 2023. URL: [https://docs.ultralytics.com/it/yolov5/tutorials/running\\_on\\_jetson\\_nano/#deepstream-configuration-for-yolov5](https://docs.ultralytics.com/it/yolov5/tutorials/running_on_jetson_nano/#deepstream-configuration-for-yolov5) (cit. on p. 39).
- [34] *JetPack SDK*. 2024. URL: <https://developer.nvidia.com/embedded/jetpack> (cit. on p. 40).
- [35] *Welcome to the DeepStream Documentation*. 2023. URL: [https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS\\_Overview.html](https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_Overview.html) (cit. on p. 40).
- [36] *Configuration Groups*. 2023. URL: [https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS\\_ref\\_app\\_deepstream.html#configuration-groups](https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_ref_app_deepstream.html#configuration-groups) (cit. on p. 41).
- [37] *How to use custom models on deepstream-app*. 2023. URL: <https://github.com/marcoslucianops/DeepStream-Yolo/blob/master/docs/customModels.md> (cit. on pp. 41, 42).
- [38] *FP16 vs FP32 – What Do They Mean and What’s the Difference?* 2022. URL: <https://bytexd.com/fp16-vs-fp32-what-do-they-mean-and-whats-the-difference/> (cit. on p. 43).
- [39] *Small-Area Population Estimation: an Integration of Demographic and Geographic Techniques*. 2013 (cit. on p. 45).
- [40] «Lightened Context Extraction Network for Object Detection». In: (2022). DOI: 10.1109/iccwamtip56608.2022.10016559 (cit. on p. 45).
- [41] *Intersection over Union (IoU) for object detection*. 2022. URL: <https://pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection/> (cit. on p. 45).
- [42] *Understanding F1 Score, Accuracy, ROC-AUC, and PR-AUC Metrics for Models*. 2023. URL: <https://deepchecks.com/f1-score-accuracy-roc-auc-and-pr-auc-metrics-for-models/> (cit. on p. 45).
- [43] Ambreen Hussain, Bidushi Barua, Ahmed Osman, Raouf Abozariba, and Taufiq Asyhari. «Low Latency and Non-Intrusive Accurate Object Detection in Forests». In: Dec. 2021, pp. 1–6. DOI: 10.1109/SSCI50451.2021.9660175 (cit. on p. 48).