Universidade de Aveiro Departamento de Engenharia Mecnica
2022

Pedro Andrade Azevedo

Deteção e seguimento de alvos múltiplos a bordo do ATLASCAR2 com Deep Learning

Detection and tracking of multiple targets aboard ATLASCAR2 with Deep Learning

Universidade de Aveiro Departamento de Engenharia Mecnica
2022

Pedro Andrade Azevedo

Deteção e seguimento de alvos múltiplos a bordo do ATLASCAR2 com Deep Learning

Detection and tracking of multiple targets aboard ATLASCAR2 with Deep Learning

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

DOCUMENTO PROVISÓRIO

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

agradecimentos / acknowledgements

Aos meus pais e irmão pelo amor todo que me deram desde que nasci. Por me ensinarem a ter orgulho, não no resultado das coisas que fazemos, mas no esforço e dedicação que colocamos na mesma, agredeço por me ensinarem a respeitar e a amar.

Ao meu orientador Vitor Santos por inspirar a exigencia e rigor que uma dissertação merece. Por toda a empatia e concelhos que transmitiu ao longo deste trabalho, isto, especialmente nos momentos em que as coisas não estavam a correr bem e um empurrãozito era necessário. A etica de trabalho que me impôs fez me entregar a melhor tese que eu poderia ter feito. Tenho orgulho no trabalho, que com só com a ajuda do meu orientador, conseguia fazer. Muitos, dizem que dois orientadores não chega, eu tinha um e valeu-me por mil.

Ao Tiago Almeida pela ajuda que prestou , pelas boas ideas que sempre tinha e pelo seu prévio trabalho com Jetson AGX Xavier que funcionou como meu guia.

Aos meus grandes amigos (ou animais depende de a quem perguntam), João Veloso, Gonçalo Silva, João Marques e Marcelo Manteigas por me levantarem quando caía, empurarem quando saltava, voarem quando eu quando me perdia e mais importante de tudo, por me fazerem cuspir o pulmão a rir durante estes 5 anos.

keywords Object Detection; Object Tracking; Deep Learning; ATLASCAR2; MOT; Jetson AGX Xavier; Artificial Intelligence.

abstract There are multiple tasks involved in the development of a fully autonomous driving system. One of the most important is the perception of the environment. To achieve this perception, multiple techniques are used. A special emphasis is given on computer vision namely for the tasks of road detection, object detection and tracking; this includes objects such as vehicles, traffic signs and pedestrians. This thesis utilizes recent Deep Learning techniques and algorithms to deploy object detection and tracking models to be utilized aboard ATLASCAR2, a self-driving car project developed by the Department of Mechanical Engineering in University of Aveiro. Different SOTA detectors such as Scaled-YOLOv4, YOLOR, YOLOv5 were trained on the BDD100K dataset and evaluated in terms of inference speed and accuracy. This was followed by an implementation and evaluation of multiple SOTA trackers such as NvDCF DeepSORT and BYTE utilizing DeepStream technology for optimized camera inference. The final solution presents a YOLOR-CSP architecture with 70.50% mAP@50 using a DeepSORT tracker with FP32 precision, achieving 17 FPS with a detection interval of zero and 33 FPS with an interval of one.

palavras-chave

Deteção de objetos; Seguimento de alvos; Deep Learning; ATLASCAR2; MOT; Jetson AGX Xavier; Inteligencia Artificial.

resumo

Há múltiplas tarefas envolvidas no desenvolvimento de um sistema de condução totalmente autónomo. Um dos principais é a percepção do ambiente. Para alcançar a percepção, são utilizadas múltiplas técnicas e é dada muita ênfase a visão computacional. Nomeadamente para as tarefas de detecção de estradas, detecção de objectos e seguimento de alvos (tais como veículos e pedestres). Esta tese utiliza as técnicas e algoritmos de Deep Learning para implementar modelos de detecção e seguimento de objectos para serem utilizados no ATLASCAR2, um projecto de um carro autonomo desenvolvido pelo Departamento de Engenharia Mecânica da Universidade de Aveiro. Diferentes detectores estado da arte como o Scaled-YOLOv4, YOLOR, YOLOv5 foram treinados no dataset BDD100K, mais tarde, foi feita uma análise, medindo a velocidade de inferência e precisão destes modelos. Seguiu-se uma implementação e avaliação de múltiplos algoritmos estado da arte para seguimento de alvos , tais como NvDCF DeepSORT e BYTE, utilizando a tecnologia DeepStream para optimizar a inferência da câmara. A solução final apresenta uma arquitectura YOLOR-CSP com 70,50% mAP@50 com um algoritmo de seguimento DeepSORT usando precisão FP32, atingindo 17 FPS com um intervalo de detecção de zero e 33 FPS com um intervalo de um.

Contents

List of Tables

Intentionally blank page.

List of Figures

Intentionally blank page.

List of Acronyms

IDF1	Identification F1 Score. The fraction of correctly identified detections over the average number of true and computed detections.
CUDA	Compute Unified Device Architecture - Software layer that gives direct access to the GPU's virtual instruction set and parallel computational elements, for the execution of compute kernels
cuFFT	CUDA Fast Fourier Transform
cuBLAS	CUDA Basic Linear Algebra Subroutine
AVs	Autonomous Vehicles
BB	Bounding Box
BDD100K	Berkeley DeepDrive 100K Images Dataset
BFLOPS	Billions of FLOPS
HOG	Histogram of Oriented Gradients
CNN	Convolutional Neural Network
COCO	Common Objects in Context (Dataset with 80 Classes used to measure the accuracy of different object detectors)
RPN	Region Proposal Networks
CSP	Cross Stage Partial Networks
FPN	Feature Pyramid Network
BiFPN	Bi-Directional Feature Pyramid Network
SORT	Simple Online Real-Time Tracking
DCF	Discriminative Correlation Filter
LIDAR	Light Detection and Ranging
DIoU NMS	Distance IoU Non Maximal Supression
DETR	DEtection TRansformer

DL	Deep Learning
FF NN	Feed Forward Neural Network
SAT	Self Adversarial Training
BFLOPS	Billions Floating Point Operations Per Second
FPS	Frames Per Second
IoU	Intersection over Union
MOT	Multi Object Tracking
MOTP	Multi Object Tracking Precision
MOTP	Multi Object Tracking Accuracy
GPU	Graphics Processing Unit
IoU	Intersection over Union
LAR	Laboratory of Automation and Robotic
mAP	Mean Average Precision
ML	Machine Learning
MOTA	Multiple Object Tracking Accuracy
MOT	Multiple Object Tracking
HOTA	Higher Order Tracking Accuracy
NMS	Non Maximal Suppression.
NN	Neural Network
NW	Not Working
PAN	Path Integral Based Convolution for Deep Graph Neural Networks
PGIE	Primary GIE (The primary detector in a DeepStream pipeline)
PNG	Portable Network Graphics (type of image file format)
PRO	Portuguese Robotics Open
RCNN	Region-based Convolutional Neural Network
RE-ID	Re-Identification
ROI	Region of Interest
SDK	Source Development Kit
SSD	Single Shot Detector

SOTA	State of the Art
SPP	Spatial Pyramid Pooling
SSD	Solid State Drive
SSH	Secure Shell protocol
W&B	Weights and Bias
YOLO	You Only Look Once.

Intentionally blank page.

Chapter 1

Introduction

1.1 Background and Motivation – The ATLAS Project

This dissertation occurs in the scope of the ATLAS Project, that started in 2002/2003 in the Laboratory of Automation and Robotics (LAR) in the Department of Mechanical Engineering of the University of Aveiro with the main objective of researching technology to be employed in autonomous driving subjects. The first success of this project was the small-scale prototype that enrolled in the Portuguese Robotics OPEN (PRO). After multiple awards in the various years of this competition, in 2010 the project team started the development of a real-scale vehicle the ATLASCAR1, a Ford Escort Station Wagon from 1998 containing several state-of-the-art sensors. This project later evolved into the ATLASCAR2 with a more modern vehicle as seen in Figure 1.1, a Mitsubishi i-MiEV, an electric car equipped with multiple LIDAR sensors and cameras, among others, needed for the development of this work.

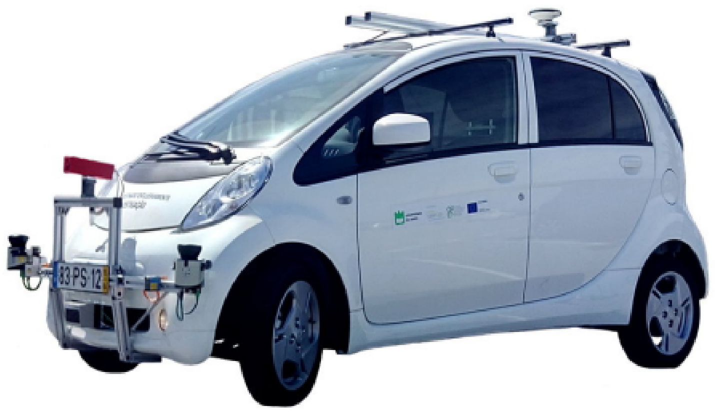


Figure 1.1: The ATLASCAR2 vehicle [1].

This dissertation focuses on the perception on the car, the ability of the car to understand it's surroundings, more specifically on object tracking and detection with the use of Deep Learning. Deep Learning is a machine learning technique that teaches computers to do what comes naturally to humans: learn by example. Deep Learning is a key technology behind driverless cars, enabling them to recognize a stop sign, or to distinguish a pedestrian from a trafic sign [2]. This technology will be described in detail in the next chapters and a brief introduction to Deep Learning can be found on Appendix A for readers new to the topic. Previous work in the area of Deep Learning was developed on LAR, the authors from [3] used Deep Learning in the classification of road and objects with semantic segmentation. Some specific work was developed in the object detection and tracking area, the authors from [4] used Deep Learning for object detection of pedestrians and vehicles, the authors from [5] utilized LIDAR and Velocity Obstacles for multi-object tracking. In the work of the authors from [4] a few studies were done on the 2018 SOTA algorithms, since that time, a lot of progress has been made in Deep Learning research and on the hardware available for deployment of edge devices. This dissertation aims to bring new technological developments to complement and enhance previous work, achieving a robust solution.

1.2 Problem description

A fully autonomous vehicle has a multitude of tasks to perform. In a simplified way we can reduce it to these simple topics/tasks. Computer vision, Environment perception, Sensor Fusion, Localization, Path planning and Control, Figure 1.2 shows an example on how these can be intertwined.

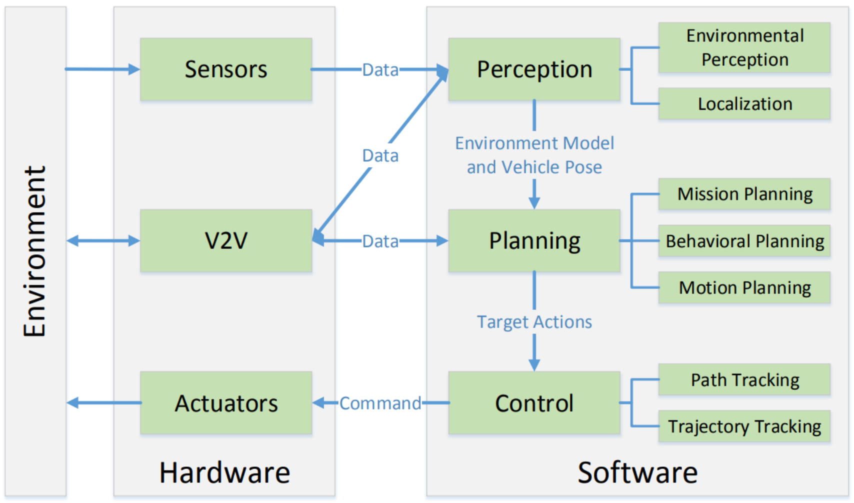


Figure 1.2: A typical autonomous vehicle system overview, highlighting core competencies [6].

To achieve environment perception with the aid of computer vision, some algorithms are needed, mainly for object and road detection. This information is later fused with different cameras/sensors and used to localize the car in the road; after that, a path is planned, and control algorithms are used to drive the car itself. While humans generally have no problem assimilating their surroundings to carry out basic functions, computers can have difficulties with this task despite the existence of the optical systems, and a lot of computational power and learning algorithms are required to do so. Many techniques used to attack this problem lay on the common heading of image recognition, some of the key aspects of this include object detection, that is, processing an image to detect and localizing objects within in [7].

With the current development of new technologies, new possible solutions have emerged to tackle the problem of perception of the car. One of these technologies was Deep Learning. This technology, a subset of machine learning and a form of AI, lays on the back of neural networks which have been showing important results in multiple tasks such as image classification, object detection, multiple class object tracking, and image segmentation. Training such neural networks with a lot of data is a computationally heavy process , this process however happens offline, so an algorithm can be trained previously and deployed on the hardware without major problems. On the other hand, the inference stage relies heavily on the hardware because that is where the segmentation and object detection is done in real time, so a great GPU power is needed. Therefore, specific hardware must be chosen and integrated into the vehicle system to run a Deep Neural Network with enough speed. The new processing unit from NVIDIA, Jetson AGX Xavier, is a good candidate and has numerous applications due to its high capacities and versatility. Due to these capabilities, this hardware was provided for this dissertation.

1.3 Objectives

The main aim of this dissertation is to train a neural network to perform object detection and object tracking with the help of Deep Learning techniques. Then implement these algorithms on the ATLASCAR2 followed by controlled real scenario tests. This includes:

- Configuring the hardware
- Selecting the right detection algorithms to deploy
- Train the algorithms
- Write deployment pipelines for the models
- Implement the tracker on the models
- Test them on the ATLASCAR2

This thesis will be composed of 5 chapters, this first one gives an overview of the problem and the motivation behind it. The second chapter describes all the related work and state of the art. In this chapter, topics like Deep Learning, object detection algorithms and object tracking will be described in great detail, and an analysis of urban driving public datasets will be made; furthermore, a brief introduction to NVIDIA's hardware provided for this thesis will be described. In the third chapter, the experimental infrastructure will be described; here, all the methodologies and technology stacks adopted will be discussed in greater detail. The fourth chapter is about the proposed solution where all the tests, data processing, model training results, and tracker implementations will be described. The final chapter contains the conclusions regarding the solution obtained as well as preparation for future developments.

Intentionally blank page.

Chapter 2

Related work and State of the Art

Many technologies and algorithms are used to solve multiple object detection and tracking. These algorithms were traditionally made with human-written feature extractions and have since evolved to Deep Learning techniques . This chapter details in depth the related work and state of the art. To explain the connection between the traditional techniques and Deep Learning techniques a small section comparing the two was provided in this chapter. After this introduction in a different sub-section, object detection algorithms will be discussed in greater detail. This sub-chapter starts by introducing what object detection is and how Deep Learning has improved the accuracy of object detectors, afterwords the difference between 2D and 3D object detectors will be explained followed by an analysis of the SOTA of object detection. After this sub-section, in a similar manner, object tracking will be approaches going in depth about the SOTA trackers.

2.1 Deep Learning vs Machine Learning

Machine Learning is a branch of artificial intelligence (AI), in a simple way it means learning patterns from examples or sample data. The machine is given data and learns from it. Data could be labeled, unlabeled, or a combination of both [8].

When distinguishing traditional machine learning from Deep Learning, in the case of computer vision, we can say that Machine Learning extracts hand-crafted features from images and performs a classification, on the other hand, Deep Learning techniques extract the features and classify them in one step, as seen in Figure 2.1 [7].

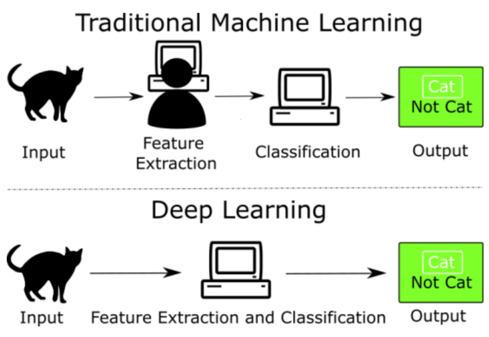


Figure 2.1: Differences between Traditional Machine Learning and Deep Learning feature extraction [7].

Deep Learning architectures have been used in multiple different areas like computer vision, speech recognition, natural language processing and even in the field of medicine. Recent developments in this area have allowed the deployment of faster algorithms that can be used in object tracking and detection. The details on how Deep Learning affects each of these specific subjects are described in the following sections.

2.2 Object detection

Since autonomous vehicles share the road with many other traffic participants such as cars, pedestrians etc, fast and reliable detection of these objects is crucial [9]. The detection pipeline typically starts with a prepossessing of the input images, followed by a region of interest detector and, finally, a classifier that identifies the object detected.

The evolution of object detectors began with the Viola Jones detector [10] that was used for detection in real-time. Traditionally, object detection algorithms used hand-crafted features to capture relevant information from images and a structured classifier to deal with spatial structures.

An example of this would be the work done in [11] where the authors propose an algorithm based on Histogram of Orientation Gradients (HOG) and Support Vector Machine (SVM). The algorithm is shown in Figure 2.2. Essentially, it passes the input image through prepossessing, computes HOG features over the sliding detection window, and uses a linear SVM classifier for detection. This algorithm captures object appearance by purposefully designed HOG features and depends on linear SVM to deal with highly nonlinear object articulation [9].

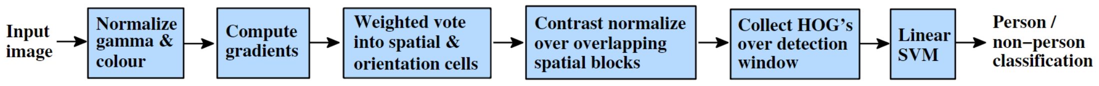

Input image
Normalize gamma & colour
Compute gradients
Weighted vote into spatial & orientation cells
Contrast normalize over overlapping spatial blocks
Collect HOG's over detection window
Linear SVM
Person / non−person classification

Figure 2.2: Overview of feature extraction and object detection chain [11]

However, these traditional approaches are not able to fully exploit the extremely large data volume and deal with endless variations of object appearance and shape. Even though they do not require historical data for training and are unsupervised in nature, these techniques have many restrictions, especially when dealing with complex scenarios like illumination effect, occlusion effect and clutter effect [7]. To fully understand the impact of the new techniques some key performance metrics used in object detection need to be introduced first.

2.2.1 Key Performance indicators

To establish a fair comparison between different detectors many metrics were defined over the years, the most dominant one being mean Average Precision (mAP). A brief introduction to other metrics is necessary to fully understand mAP so an explanation on those is provided.

Definition of Terms:

- True Positive (TP) - Correct detection
- False Positive (FP) - Incorrect detection

- False Negative (FN) - A ground truth missed (not detected) by the detector

Intersection over Union - IoU

IoU metric evaluates the division between the area of overlap and the area of union. In other words, it evaluates the degree of overlap between ground truth (gt) and predictions (pd). It ranges from 0 to 1, where 1 would be a perfect overlap between the ground truth and the prediction. [12]. In Figure 2.3 we can see the difference between a good IoU and a bad one.

$$IoU(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{2.1}$$

Where "A" and "B" are the predicted and ground truth bounding boxes.

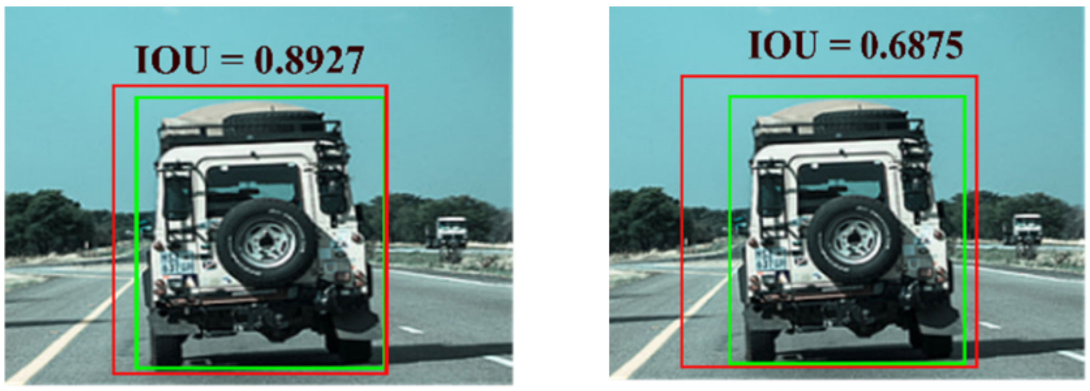


Figure 2.3: Visual example of different IoU values. Green box: ground truth; red box: prediction [13].

Precision and Recall

Precision attempts to answer the question "What proportion of positive identifications was correct?". Recall relates the proportion of actual positives that were correctly identified.

$$Precision = \frac{TP}{TP + FP} \tag{2.2}$$

$$Recall = \frac{TP}{TP + FN} \tag{2.3}$$

So, for example, analysing a vehicle object detector, the precision contain information of "from all the cars predicted by the model, how many of them were actually cars"? And recall would say "how many cars the model identified".

Average Precision (AP)

When plotting the precision recall curve evaluated at an IoU threshold, we get the Average Precision.

$$AP@\alpha = \int_0^1 p(r)dr \tag{2.4}$$

Where "α" is the threshold value, p the precision and r the recall.

Mean Average Precision (mAP)

When considering a multi-class object detector the mAP gives us the mean AP across all classes.

$$\text{mAP} = \frac{\sum_{q=1}^{Q} \text{AP}(q)}{Q} \tag{2.5}$$

Where "AP" is the average precision of each "q" class and "Q" is the number of classes.

In recent years, object detection performance has improved significantly, with an increase from 30% mean Average Precision (mAP) to more than 90% in 2018 on the PASCAL VOC benchmark (measured on the PASCAL VOC object detection public dataset) [14]. The main driver of these improved results was the use of Deep Learning. With the development of new technologies that allow faster and easier pipelines and the availability of large-scale open datasets, the development of powerful models has become a reality [7].

2D vs 3D Object Detectors

In the context of autonomous driving, both 2D and 3D object detectors are important. 2D object detectors give bounding boxes with four Degrees of Freedom (DOF) in the form of [xmin,ymin,xmax,ymax] [13] as seen in Figure 2.4.

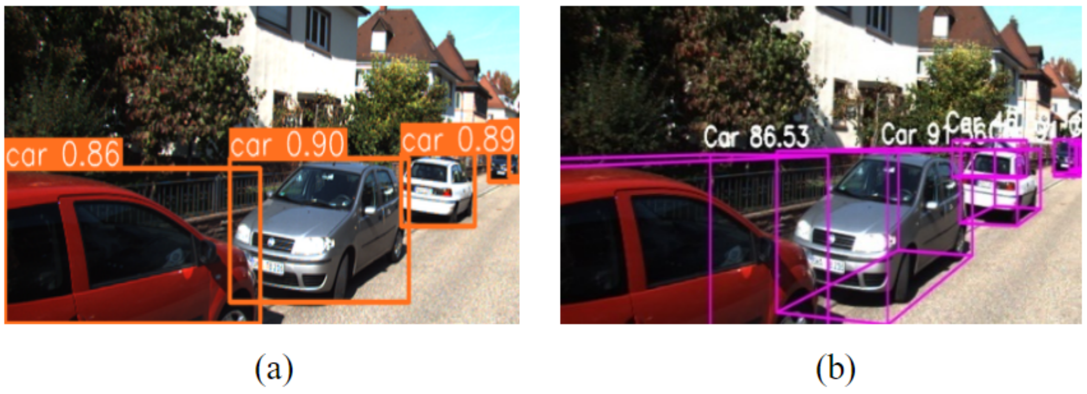


Figure 2.4: 2D vs 3D bounding box. [13]

This gives information of the position of the objects in the 2D plane but lacks information about the depth of the object. This information is crucial to predict important factors like shape, size and position of the objects in order to develop other self-driving tasks like path planning or collision avoidance [13]. While 2D detectors use input from images, 3D detectors use data from camera, LIDAR or radar to generate 3D bounding boxes. There are multiple approaches to do so, but some of these detectors take advantage of their 2D counterparts and project LIDAR/radar information, fusing it in order to obtain depth and a 3D bounding box like [15]. Lately, some authors have been leveraging monocular image-based to perform 2D to 3D lifting and create 3D object detection results like [16]. A taxonomy summary can be seen in Figure 2.5.

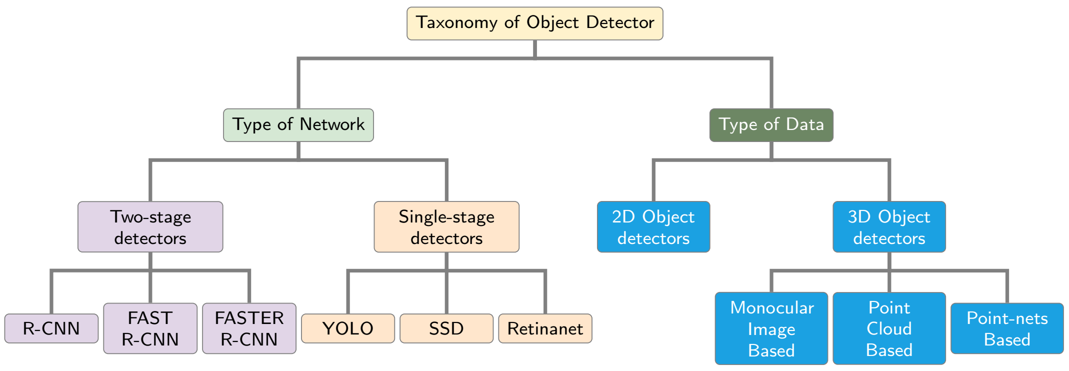


Figure 2.5: Taxonomy of object detectors with some example models (Adapted from [13]).

In the context of this dissertation the focus will be on 2D object detection. However, the work from this thesis can be fused with other sensors in future projects to generate 3D bounding boxes.

2.2.2 Types of Object Detectors

Object detection tries to answer the question, "Are there any objects in the image?". If yes, "what objects and where?". We can divide object detectors into two main types: One-stage and Two-stage detectors. Object detectors extract features from the input image/video frame in two main tasks; first they find the objects (and their bounding boxes) then they classify them. The architecture of both types can be seen in Figure 2.6.

In a two staged architecture these steps are separated; first it gets an object region proposal then classifies it based on the features extracted from the region proposed. These architectures achieve a very high accuracy rate but are rather slow, which makes them unfit for real-time applications like self-driving vehicles.

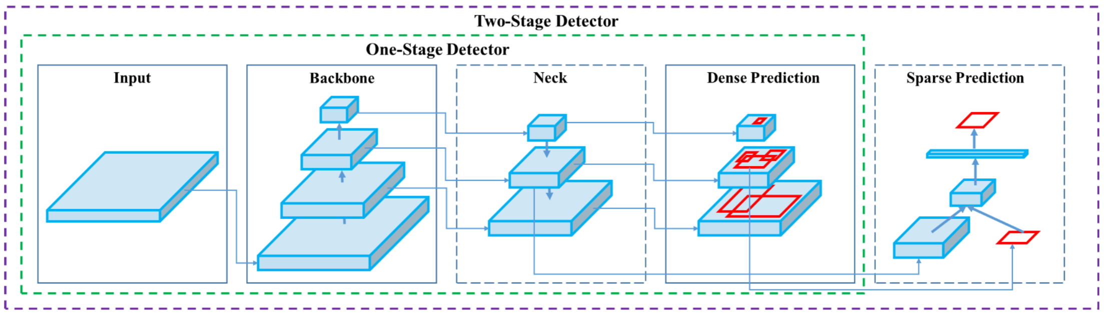


Figure 2.6: Architecture of Object Detector(Adapted from [17]).

Some examples of two-stage detectors include RCNN [18], Fast-RCNN [19] and Mask-RCNN [20]. Usually, object detectors are made with a backbone. This is typically a network that acts as a feature generation network for object detection [8]. CNNs are usually chosen for this task. To achieve the best accuracy and efficiency, when building a model, we must choose the most adequate backbone. More accurate backbones are generally tied to how deep and dense they are, some example of backbones are ResNet [21] and ResNetXt [22]. On the other hand, if inference speed or computational power is a concern, a lightweight backbone might be a better choice, especially in mobile applications. When referring to real-time detection systems, we must often adapt the detection backbone and make a fair trade-off between accuracy and speed. In short, deeper and densely connected backbones replace the shallower and sparsely connected ones to obtain more detection accuracy [8].

A one-stage detector predicts the bounding box over the images without a region proposal step, achieving greater detection speeds. A sample architecture of each type of detector can be seen in Figure 2.7; here we can see that the ROI generation step differes from two-staged to one-stage.

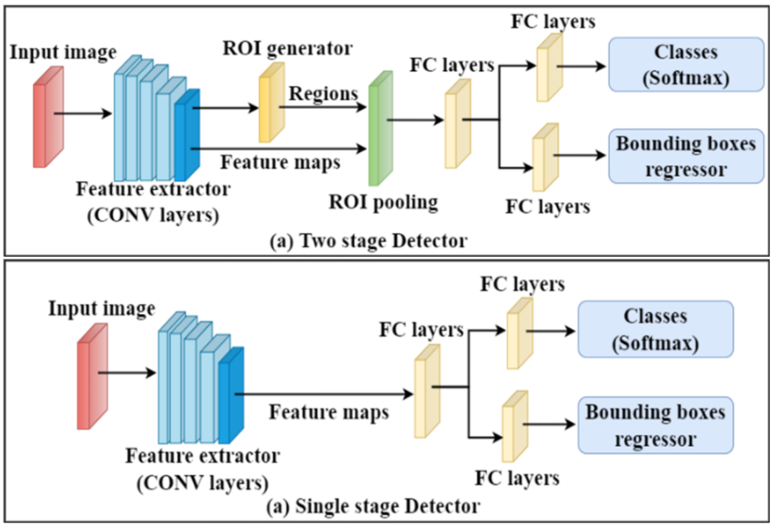


Figure 2.7: Two-staged vs One-stage Detectors Diagram [13].

Some examples of one-stage detectors are YOLO [23], SSD [24] and RetinaNet [22] EfficientDet-D7 [25].

Figure 2.8 shows an overview of the various real-time top performing models in terms of mAP at the COCO dataset for the years between 2017 and 2022. It is important to keep in mind that even though there is a big speed difference between one-stage to two-stage, some two-stage detectors can actually inference in real time like Mask RCNN [20], this is why we can see both types of detectors in Figure 2.8. The current top performing model is a version of YOLO, there is a series of versions of this algorithm which will be further detailed in the following sub-sections.

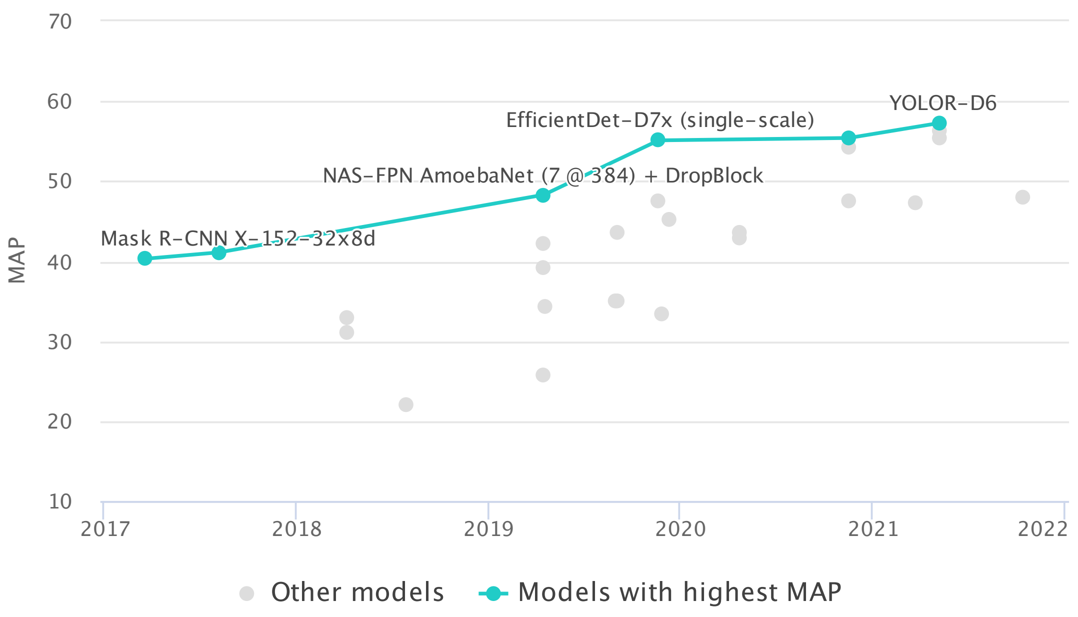


Figure 2.8: mAP of various models preforming real-time object detection on COCO (Adapted from [26]).

2.2.3 A shift in the computer vision paradigm

The models presented on Figure 2.8 present the real time detectors; as discussed before the inference speed of a detector on the model and architecture used. Many of the improvements of real-time object detectors are ported from non-real time detectors once new papers are published, so these can be seen as a "future" of where real-time object detection could be in terms of accuracy. This makes it important to analyse not only real-time detectors but also non-real time detectors. The backbone of traditional detectors are typically CNN or RCNN based; with the development of new technologies such as Transformer Neural Networks the future of object detection is taking a shift. In the 2022, there seems to be a shift in paradigm to the use of Swin Transformers as a backbone for object detectors [27], reaching SOTA results like DINO [28] which significantly reduces its model size and pre-training data size while achieving better results compared to the previous SOTA. However, at the time of writing this thesis, there is no current solution for the use of Swin-Transformers use in real-time applications in object detection, since its inference speed is still low (due to being a recent breakthrough). Mentioning this type of algorithms is important, since it sets our eyes on a possible better solution in the future; for example some sort of YOLO type detector with a transformer backbone. The future of computer vision seems to be transformer based. Figure 2.9 shows the top performing detectors (regardless of their inference speed).

In the following subsections multiple popular object detectors will be discussed.

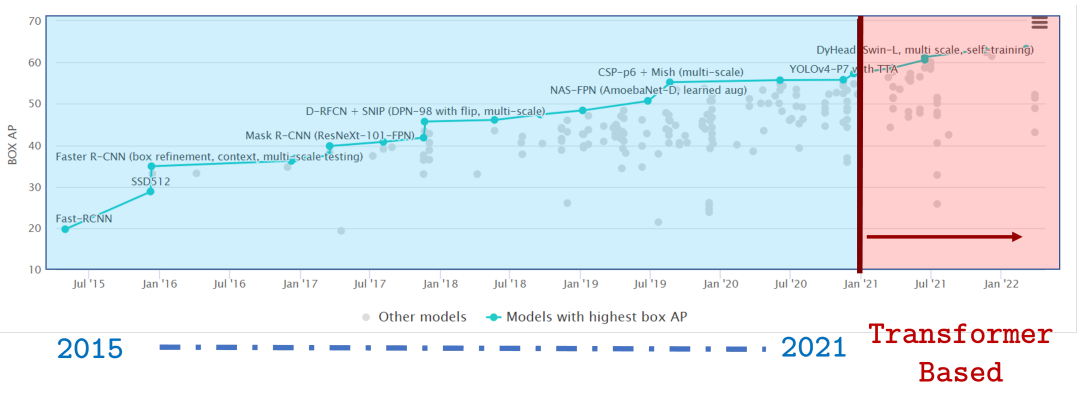


Figure 2.9: mAP of top performing models in COCO dataset (regardless of being capable of real-time inference ot not) (Adapted from [29]).

In summary: object detection model architectures

Object detectors can either be 2D or 3D; this thesis focuses on 2D object detection. The 2D object detectors can either be one-staged or two-staged; one-stage being faster than two-staged.

These object detectors are measured against a benchmark on different datasets. The most popular benchmark and metric in recent papers is mAP measured in the COCO dataset.

The current SOTA algorithms for real-time object detection are the YOLO series.

2.2.4 Faster R-CNN

R-CNN was one of the first Deep Learning-based object detectors and used an efficient selective search algorithm for ROI proposals as part of a two-stage detection [18]. Fast R-CNN improved R-CNN low inference speed and accuracy. In this model the input image is fed to a CNN, generating a feature map and ROI projection. These ROIs are then mapped to the feature map for prediction using ROI pooling, but instead of inputting the ROI to the CNN layers, Fast-RCNN uses the entire image directly to process the feature maps to detect objects [30].

Faster R-CNN used a similar approach, but instead of using a selective search algorithm for the ROI proposal, it employed a separate network that fed the ROI to the ROI pooling layer and the feature map, which were then reshaped and used for predictions [13]. The Faster R-CNN introduces Region Proposal Networks (RPN) that share full-image convolutional features with the detection network, this enables nearly cost-free region proposals. An RPN is a fully convolutional network that simultaneously predicts object bounds and object scores at each point. The RPN is trained end-to-end to generate high-quality region proposals which are used by Fast R-CNN for detection. The authors merged RPN and Fast-RCNN into a single network by sharing their convolutional features with "attention" mechanisms. Essentially the RPN component tells the unified network

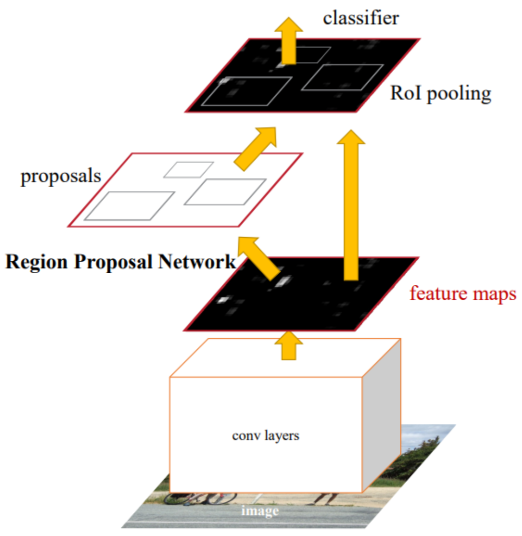


Figure 2.10: Faster R-CNN is a single, unified network for object detection. The RPN module serves as the "attention" of this unified network [19].

where to look [19]. In Figure 2.10 we can how this RPN is incorporated.

2.2.5 Single-Shot Detector

Similar to YOLO, SSD is a one-stage detector. It discretizes the output space of bounding boxes into a set of default boxes over different aspect ratios and scales per feature map location. At prediction time, the network generates scores from the presence of each object category in each default box and makes adjustments to the box to better match the shape of the object. The authors of SSD managed to achieve a better mAP outperforming Faster R-CNN [24].

2.2.6 EfficientDet

EfficientDet is a detector whose authors propose a weighted bi-directional feature pyramid network (BiFPN), which allows easy and fast multi-scale feature fusion. In addition to this, they propose a compound scaling method that uniformly scales the resolution, depth, and width for all backbone, feature network, and box/class prediction networks at the same time [31]. It achieves good accuracy but not as fast as YOLO algorithms.

2.2.7 The YOLO Series

The Original YOLO

The authors of YOLO [23] proposed a different approach to the existing object detectors of the time. Although the models increased in accuracy each year, object detectors lacked the speed to allow them to perform in real time. YOLO presented a new approach, instead of re-purposing a classifier to perform object detection, YOLO [23] instead framed object detection as a regression problem to spatially separated bounding boxes with associated class probabilities. By doing so the entire object detection pipeline was turned into a single network that could be optimized end-to-end directly on detection performance (instead of having to maximize a classifier then perform classification with it).

YOLO, however, suffered from a many of limitations specially dealing with small objects. Due to their size it is hard to separate them when they are present in groups. YOLO works like a grid, and each grid can only detect a single object, so when objects are presented in groups, it becomes harder to detect them. It also sacrifices accuracy for inference speed so it is not as accurate as some of the other past SOTA models.

YOLO divides images into regular grids, as seen on Figure 2.11, and performs detection and localization on those very same grids (Residual Blocks). These grids return three things; the bounding box coordinate with respect to their cell coordinates, the object label, and the probability that the object is in the cell grid [23].

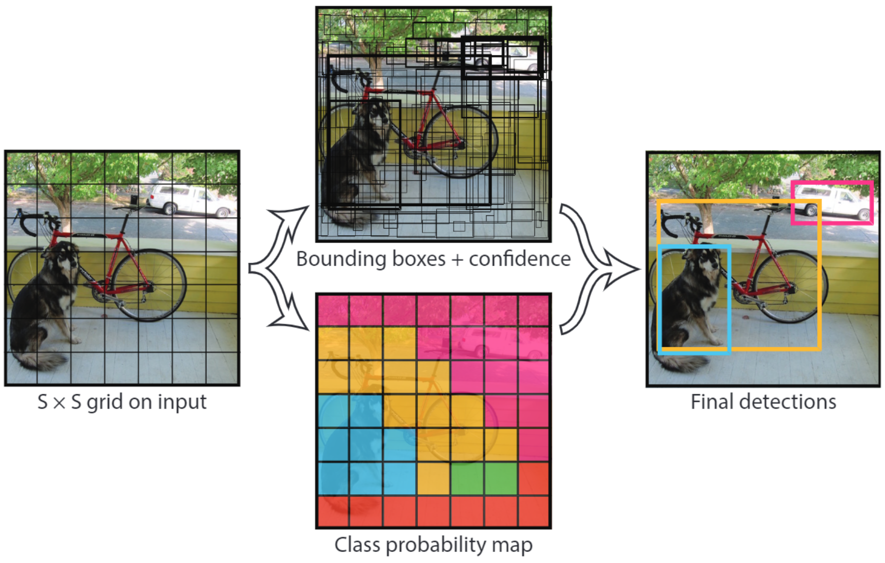


Figure 2.11: YOLO [23] divides the image into an S × S grid and for each grid cell predicts B bounding boxes.

This makes the algorithm fast and lowers computation cost since detection and recognition is made in a single shot by the cells. One drawback of this approach is that it creates duplicate predictions because each cell predicts a bounding box. So, many times the same object is being predicted with multiple different bounding boxes. All this "noise" is passed through a Non-Maximal suppression algorithm, suppressing the bounding boxes that have lower probability scores. In summary, it divides the image into grids of equal size, performs object detection and classification, and eliminates noise with Non-maximal suppression. [32] That is, choosing the highest probability score and suppress all the bounding boxes having the largest IoU with the current high probability bounding box, then it repeats these stages in a loop until the final bounding boxes are obtained as seen on Figure 2.12.

The architecture consists of three key components: the head, neck, and backbone. The backbone is the part of the network made from convolutional layers to extract key features from an image. These first layers are trained on a large dataset with low resolution like ImageNet, then the neck uses those features with fully connected layers to make predictions on probabilities and bounding boxes. Finally, there is the head of

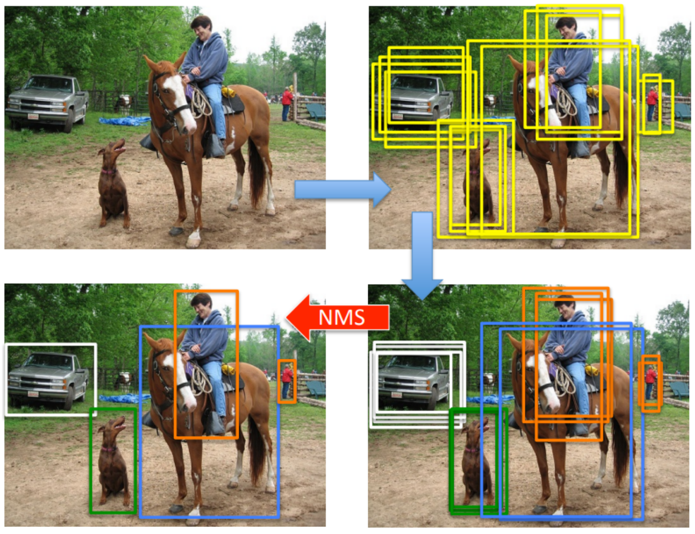


Figure 2.12: Example of Non-maximal suppression [32].

the detector that can be interchanged with other layers with the same input shape for transfer learning.

In summary: YOLO

YOLO introduced a one-stage detector capable of real time object detection with resonable accuracy.

YOLOv2

Batch normalization is a technique used to train neural networks that prevents the weights in the network from becoming imbalanced with extremely high or extremely low values, since it adds normalization in the gradient process itself. This helps solve the problems of exploding gradient and vanishing gradient, it also increases training speed and reduces the ability of outlying large weights that will over influence the training process. To fight the original YOLO's low performance on small objects in groups, YOLOv2 introduces batch normalization; this lead to improvements in convergence while eliminating the need for other forms of regularization. On top of this, YOLOv2 also introduces anchors. In the original YOLO if more than one object is located within the cell YOLO would not be able to classify them both since one cell is only able to perform one classification. On YOLOv2 [33] a single cell can perform multiple predictions since it predicts five bounding boxes for each cell.

In summary: YOLOv2

Introduces anchors and batch normalization. This improved the YOLO performance, especially on small objects.

YOLO900

YOLOv2 was trained on COCO, a dataset with 80 classes in very diverse scenarios that became the standard metric to compare object detection models. In order to expand the number of classes YOLOv2 could detect the authors of YOLO9000 [33] used labels from both ImageNet and COCO, merging the classification and detection tasks to only perform detection. It makes use of hierarchical classification where classes and their sub-classes are represented in a tree-node based format. It provides a lower mAP than YOLOv2, but can detect more than 9000 classes, making it a powerful algorithm.

In summary: YOLO900

Expands on the work of YOLOv2, allowing it to perform in many more classes.

YOLOv3

YOLOv3 [34] seeks to improve YOLOv2's work by implementing modern CNNs that use residual networks and skip connections. YOLOv3 used DarkNet-53 instead of DarkNet-19 as a backbone. This architecture allows it to predict at three different scales, having feature maps extracted from these layers. This further increases the performance of YOLOv2 to detect smaller objects. YOLOv3 predicts 3 bounding boxes per cell (compared to the five YOLOv2 predicts) but these are made at 3 different scales, so multiplying, it adds up to a total of 9 anchor boxes.

In summary: YOLOv3

Improved YOLOv2 architecture using modern CNNs and multiple-scale predictions.

YOLOv4

YOLOv4 adds weighted residual connections, Cross Mini Batch Normalization , cross stage partial connections, self adversarial training and mish activation function to the modern methods of regularization and data augmentation.

YOLOv4 authors initially considered the following backbones:

- CSPResNext50
- CSPDarknet53
- EfficientNet-B3

The first two are cross stage partial networks (CSP) based on DenseNet [35]. DenseNet aimed to solve the problem of the vanishing gradient by establishing extra partial connections between layers, as seen on Figure 2.13, to bolster feature propagation and to encourage the network to reuse features (reducing the number of parameters).

EfficientNet [25] was created by Google Brain to study the scaling of ConvNets (depth, input size, width etc.) as seen in Figure 2.14. This network managed to outperform other networks of comparable size on image classification at the time. However, in the scenario

Table 2.1: Initial YOLOv4 backbone considerations [17].

Backbone model	**Input network resolution**	**Receptive field size**	**Total parameters**	**Average size of layer output (W×H×C)**	**BFLOPs (512×512 network resolution)**	**FPS (GPU RTX 2070)**
CSPResNext50	512×512	425×425	20.6M	**1058 K**	31 (15.5 FMA)	62
CSPDarknet53	512×512	725×725	**27.6M**	950 K	52 (26.0 FMA)	**66**
EfficientNet-B3	512×512	**1311×1311**	12.0M	668 K	11 (5.5 FMA)	26

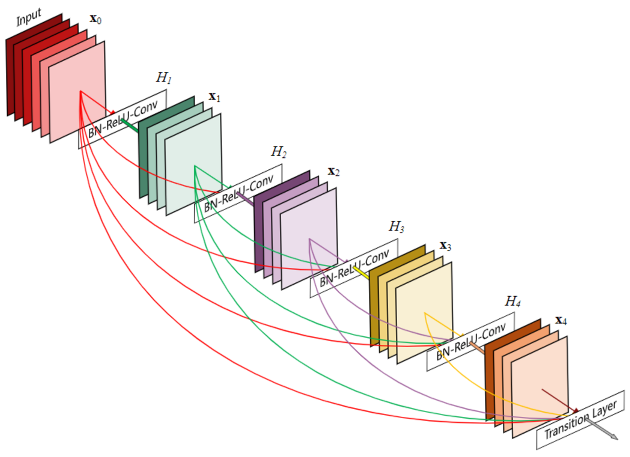


Figure 2.13: DenseNet Architecture [35].

of object detection, the authors of YOLOv4 opted to implement CSPDarknet53 for the backbone network.

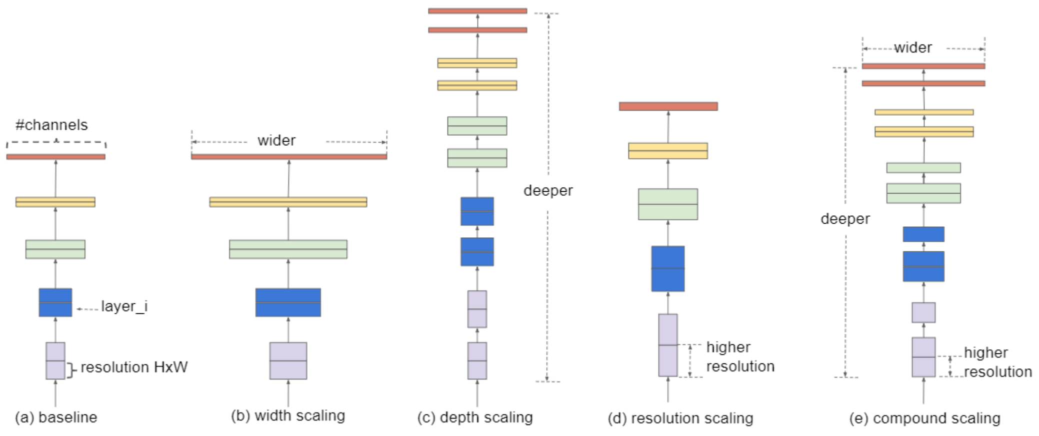


Figure 2.14: EfficientNet Scaling [25].

To use the features provided by the backbone, YOLOv4 neck proposes the use of PAN for feature aggregation and an additional SPP block to increase the receptive field (region of space that a neuron or unit is exposed to in the input data) and separate out the most important features from the backbone. A similar mechanism can be seen in Figure 2.15.

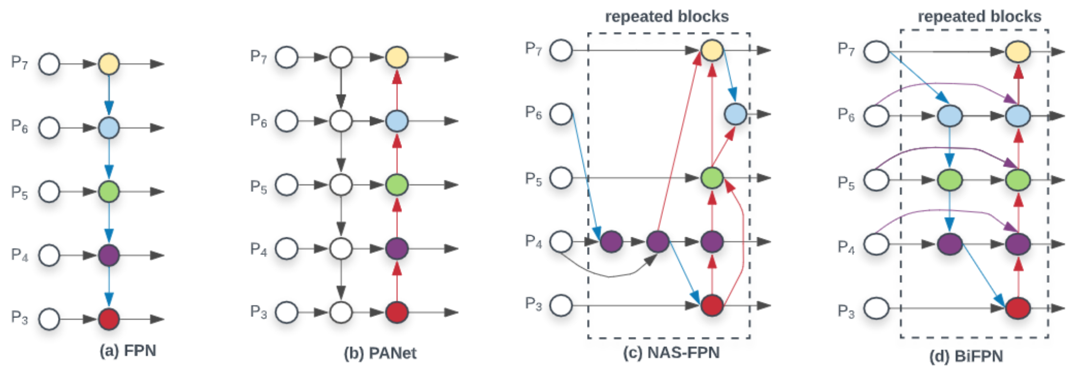


Figure 2.15: Efficient Det FPN. In the case of YOLOV4 each of those entries P' would refer to one layer of the Neural Network [31].

The YOLOv4 head deploys the same head as YOLOv3 with the anchor-based detection steps and three levels of detection granularity.

YOLOv4 adopts what the authors call a "Bag of Freebies" that is a bunch of changes that directly improve the performance of the network without adding inference time in production. Most of these changes have to do with data augmentation.

Since many of these techniques were already well known to the computer vision community, the main contributions were related to the mosaic data augmentation as seen on Figure 2.16. This type of augmentation tiles together four images, improving the ability of the model to detect smaller objects. Aside from this type of augmentation, the authors also added Self-Adversarial Training (SAT), which seeks to find the portion of the image that the network most relies on during training.

When designing a neural network (or any type of object detector), a compromise between inference speed and model complexity should be made. With this in mind, the authors have provided what they call "Bag of Specials", these significantly increase performance time while adding marginal increases to inference time, so they were deemed worth the trade-off.

One of the changes made concerns the activation functions. Due to the nature of the YOLOv4 architecture while passing the features from one layer to another, the problem of vanishing gradient becomes relevant.

This makes it difficult to pass feature creations to their optimal input. As an alternative, the authors have suggested the use of the Mish activation function. This function a more smooth profile compared to ReLU as seen in Figure 2.17 and Figure 2.18 [36]. Given by the equation:

$$\mathrm{mish}(x) = x \cdot \tanh(ln(1 + e^{x})) \tag{2.6}$$

To separate the predicted bounding boxes DIoU NMS is used. This differs from

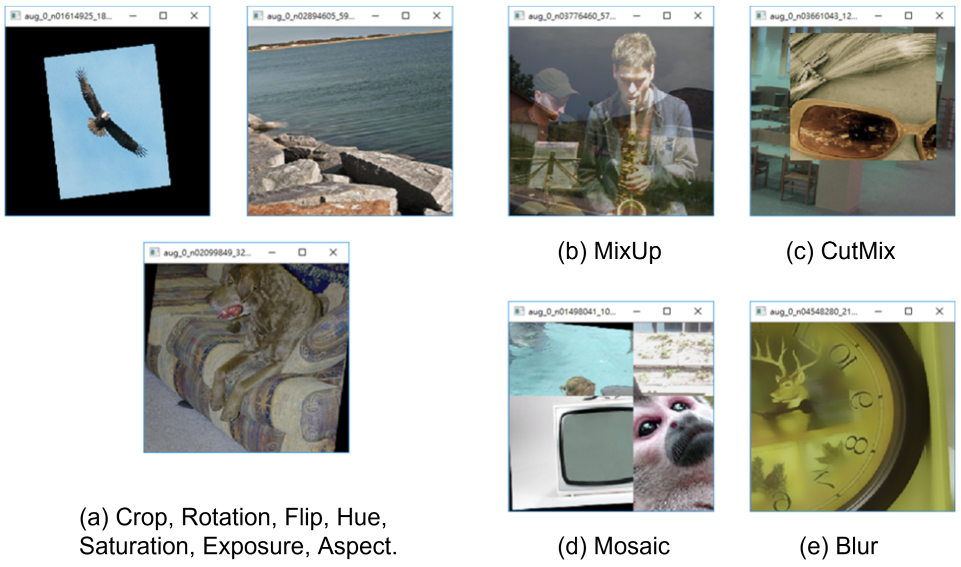

(a) Crop, Rotation, Flip, Hue, Saturation, Exposure, Aspect.
(b) MixUp
(c) CutMix
(d) Mosaic
(e) Blur

Figure 2.16: YOLOv4 Image augmentation examples [17].

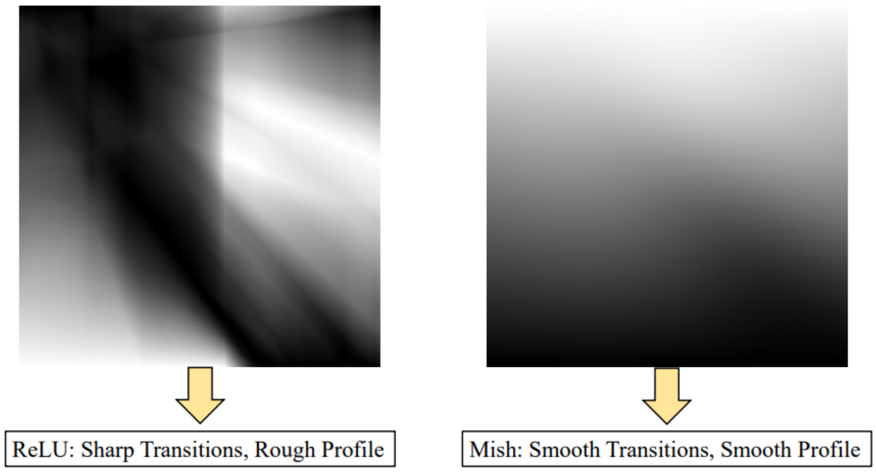


Figure 2.17: Mish vs ReLU [36].

the original NMS since it considers not only the overlap area but also the central point distance between boxes. This improves the algorithm performance for cases with occlusion, since only using the overlap area produces false suppressions for cases with occlusion [37].

For batch normalization the authors use Cross Mini-Batch Normalization and finally they use DropBlock regularization; with the DropBlock, some sections of the image are hidden from the first layer, forcing the network to learn additional features.

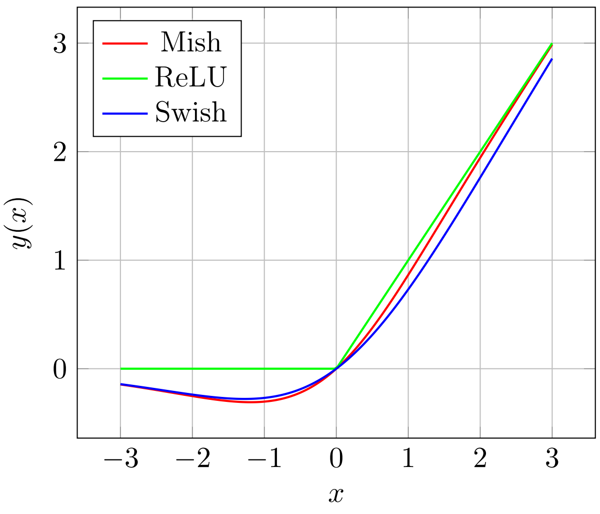


Figure 2.18: Comparison between Mish and various activation functions.

In summary: YOLOv4

Shows a big increase in accuracy over YOLOv3. Uses CSP Networks as a backbone. Utilizes "Bag of Frebies" and "Bag of Specials" to increase the model performance.

YOLOR

With the huge success of YOLOV4 more papers followed, more specifically YOLOR [38]. The goal of this network is to, with the same model, to be able to perform multiple

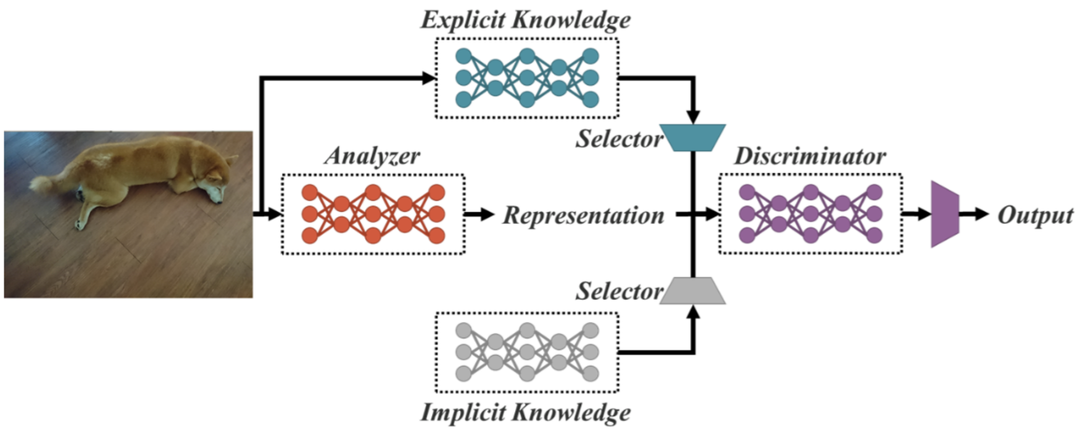


Figure 2.19: YOLOR -Multi purpose single unified network model [38].

tasks such as object detection, instance segmentation and panoptic segmentation with a single network. To achieve this, the authors relied on what they call Explicit and Implicit Knowledge fusing it on to a unified model, as seen on Figure 2.19. Explicit knowledge is connected to the shallow layers of the neural network, which is directly correlated with the observations that are made. Implicit knowledge is obtained by features in the deep layers. For the explicit learning, the authors used a DETr (Detection Transformer), Non Local Networks and a method of Kernel Selection. For implicit learning, they opted for techniques such as:

- Manifold Space Reduction
- Kernel Alignment
- Offset refinement
- Anchor refinement

With these techniques, YOLOR achieved similar results in terms of accuracy but an outstanding increase in performance speed compared to the other SOTA models as seen in Figure 2.20 and Table 2.2.

Table 2.2: Comparison between State-Of-The-Art models and YOLOR on COCO test dataset [38].

Method	**pre.**[a]	**seg.**[b]	**add.**[c]	AP^{test}	AP_{50}^{test}	AP_{75}^{test}	FPS^{V100*}
YOLOR				55.40%	73.30%	60.60%	30
ScaledYOLOv4				55.50%	73.40%	60.80%	16
EfficientDet	✓			55.10%	74.30%	59.90%	6.5
SwinTransformer	✓	✓		57.70%	-	-	-
CenterNet2	✓		✓	56.40%	74.00%	61.60%	-
CopyPaste	✓	✓	✓	57.30%	-	-	-

[a] pre. : large dataset image classification pre-training.
[b] seg. : training with segmentation ground truth.
[c] add. : training with additional images.
[*] Measured on a TeslaV100 Graphics card

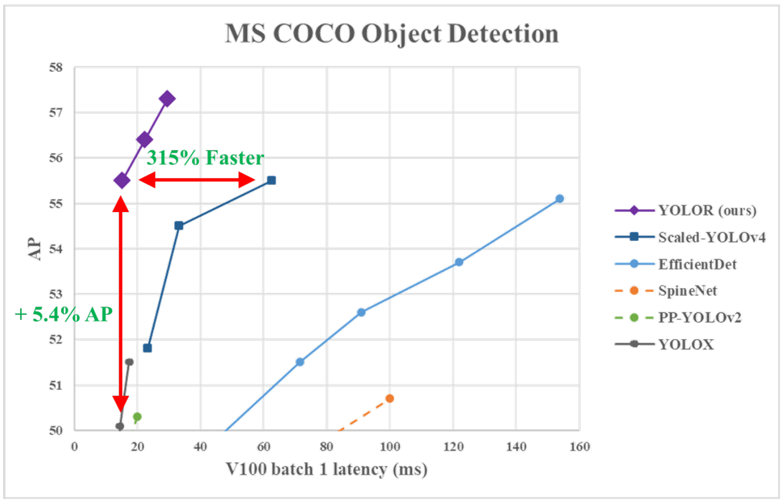


Figure 2.20: YOLOR comparison on MS-COCO Dataset [38].

YOLOv5

YOLOv5 [39] is a family of object detection architectures and models pre-trained on the COCO dataset. Its architecture is similar to YOLOv4; having a backbone, neck and head. For the backbone CSP (Cross Stage Partial Networks) are used (CSP-Darknet-53) [40]. For the neck a version PANet is used [41] to extract feature pyramids, these, help the model generalize on object scaling (big and small); in this case a CSP-PAN and SPPF are used. The model head is identical to YOLOv3. It uses similar image augmentation techniques including mosaic, copy paste random affines, etc. The authors choose Leaky ReLU and Sigmoid activation function in the middle/hidden layers. There is no official paper for the YOLOv5, however, some authors made some comparisons between YOLOv4 and YOLOv5. The conclusions are close in terms of accuracy and speed, although they vary from paper to paper and the size of the model. In the scope of this dissertation, inference speed is a big factor and it is highly dependent on hardware. The trade-off between inference speed and accuracy is the reason why some authors like [42] opted for YOLOv5 over YOLOv4 for their autonomous aerial vehicle, since it provided them with a better mAP with a similar inference speed. The work of [43] directly compared YOLOv4, YOLOv5 and YOLOX. Here YOLOv5 outperformed YOLOv4-CSP on terms of accuracy with the model of 640×640 resolution, however, this conclusion varied depending on the resolution of the detector; as model resolution is increased YOLOv4 seemed to outperform YOLOv5. In the context of this dissertation, perhaps it is more fair to compare the lighter (smaller) models where YOLOv4 achieved higher values in terms of accuracy, but YOLOv5 in terms of inference speed.

It is not clear in the literature which architecture (YOLOv4/v5) is better as they differ on the dataset, hardware, size of the model and inference speed. A further discussion of these topics will be made in the experiment section of this thesis. With the creation of YOLOv5, significant value is added through its translation of the Darknet research framework to the PyTorch framework. This, combined with the open source development, has made YOLOV5 extremely easy to test with, train and deploy, making its usage very friendly compared to other versions of YOLO.

In summary YOLOv5

Has similar performance to YOLOv4 but it is easier to use/train/deploy, which allows for faster experimentation and quicker deployment.

YOLOX

YOLOX is an anchor-free version of YOLO with a simpler design but better performances. The main difference between this model and traditional YOLO is that YOLOX is an anchor-free algorithm that was built together with advanced detection techniques, i.e, a decoupled head and SimOTA. This model achieves higher performance than the YOLOv4/v5. A comparison between YOLOX and other algorithms can be seen in Figure 2.21.

Currently, additions of an anchor-free system are being made to YOLOR [38] to create an even better model. As the results of the YOLOX and YOLOR papers were published at a similar time, a fair comparison is not established by the authors in the papers. However, by analyzing the mAP values in the published papers, YOLOR offers

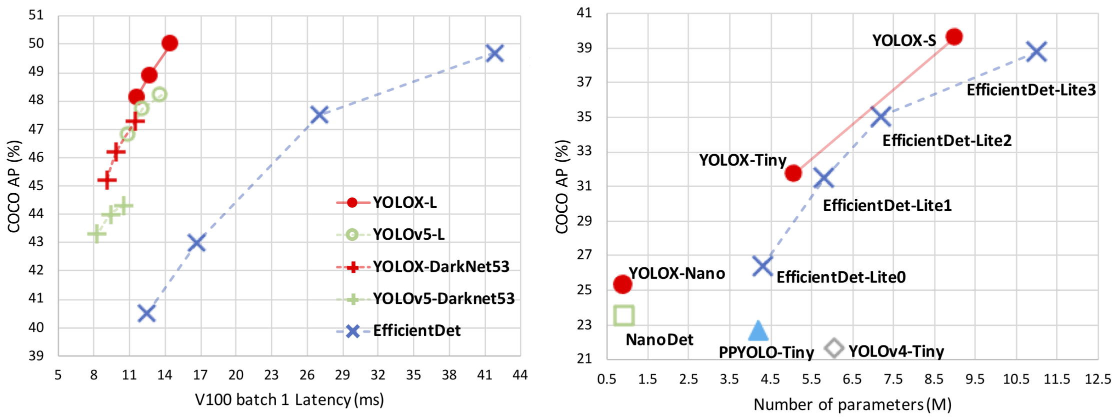


Figure 2.21: Comparison of YOLOX with different SOTA algorithms [43].

higher values on the larger model. YOLOX, however, seems to perform very well on edge devices using smaller model like YOLOX-Tiny and YOLOX-Nano. [43]

This version of YOLO seems to be improving, and the authors are currently working on a version with a Swin Transformer-based backbone; another recent paper [44] has shown that this is possible with important results. Currently, at the time of writing this dissertation, the only transformer-based YOLO version is VitYOLO [44], but its performance is very limited and does not compare to the other YOLO versions. However, both YOLOX and YOLOR authors are running experiments with these transformer based systems as it seems to be the direction were the future of computer vision is headed.

In summary YOLOX

Offers a boost in performance over YOLOv5 and YOLOv4 by using anchor-free system and advanced detection techniques

Scaled-YOLOv4

Scaled-YOLOv4 [45] offers an extension to the traditional YOLOv4 allowing it to scale effectively maintaining a good performance. First, the authors redesigned YOLOV4 proposing YOLOv4-CSP and later developed the scaled version.This scaling allows it to be used in a range of different applications having a good compromise between speed and accuracy, being able to operate in real time, as well as in embedded devices. The authors designed a powerful scaling method for small models that can balance systematically the computation cost and memory bandwith of a shallow CNN; and designed a simple yet effective strategy for scaling a large object detector and analyzed the relations among all model scaling factors.

In summary Scaled-YOLOv4

Extends the original YOLOv4 paper by giving it good scaling capabilities capable of performing in different scenarios.

YOLO Series Summary

All these models and improvements can be hard to grasp and compare so, a small simplified summary, is shown in Figure 2.22.

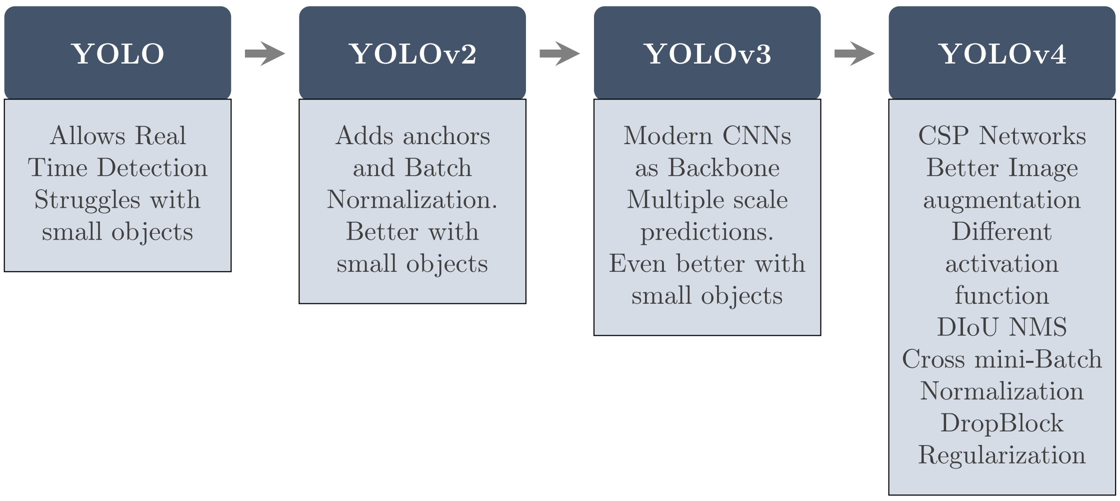

YOLO
Allows Real Time Detection Struggles with small objects
YOLOv2
Adds anchors and Batch Normalization. Better with small objects
YOLOv3
Modern CNNs as Backbone Multiple scale predictions. Even better with small objects
YOLOv4
CSP Networks Better Image augmentation Different activation function DIoU NMS Cross mini-Batch Normalization DropBlock Regularization

Figure 2.22: An evolution of the YOLO series from YOLO to YOLOv4.

For the current SOTA of real time object detection, a small summary was also made in Figure 2.23. The SOTA mark added refers to the largest and highest-performing model of each algorithm (accuracy). There are many details to keep in mind when comparing all these, especially regarding model size and inference speed, which is not addressed by this simple diagram.

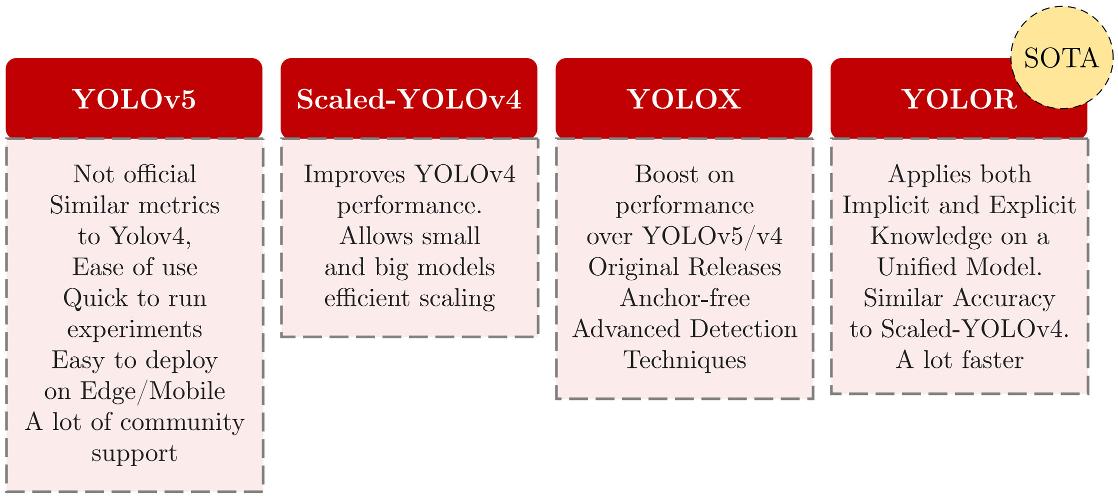

YOLOv5
Not official Similar metrics to Yolov4, Ease of use Quick to run experiments Easy to deploy on Edge/Mobile A lot of community support
Scaled-YOLOv4
Improves YOLOv4 performance. Allows small and big models efficient scaling
YOLOX
Boost on performance over YOLOv5/v4 Original Releases Anchor-free Advanced Detection Techniques
YOLOR
SOTA
Applies both Implicit and Explicit Knowledge on a Unified Model. Similar Accuracy to Scaled-YOLOv4. A lot faster

Figure 2.23: Comparison between more recent YOLO models.

A table comparing some of the models described in this chapter is provided by [13] and shown in Table 2.3. The important thing to keep in mind is that all these inference speed tests are not made on the same hardware and the author gives little descriptions on how these were measured.

Table 2.3: Comparison between different 2D-3D models of object detection [13].

Name	Year	Type	Dataset	mAP	Inference rate (fps)
R-CNN	2014	2D	Pascal VOC	66%	0.02
Fast R-CNN	2015	2D	Pascal VOC	68.80%	0.5
Faster R-CNN	2016	2D	COCO	78.90%	7
YOLOv1	2016	2D	Pascal VOC	63.40%	45
YOLOv2	2016	2D	Pascal VOC	78.60%	67
SSD	2016	2D	Pascal VOC	74.30%	59
RetinaNet	2018	2D	COCO	61.10%	90
YOLOv3	2018	2D	COCO	44.30%	95.2
YOLOv4	2020	2D	COCO	65.70%	62
YOLOv5	2021	2D	COCO	56.40%	140
YOLOR	2021	2D	COCO	74.30%	30
YOLOX	2021	2D	COCO	51.20%	57.8
Complex-YOLO	2018	3D	KITTI	64.00%	50.4
Complexer-YOLO	2019	3D	KITTI	49.44%	100
Wen et al.	2021	3D	KITTI	73.76%	17.8
RAANet	2021	3D	NuScenes	62.00%	

2.2.8 Datasets for object detection

When training Deep Learning models, choosing the correct dataset is a crucial task. With the increasing interest in self-driving technologies, many new high-quality datasets started to show up. Some of the most popular ones are KITTI, Apollo and BDD100k. Some authors like [46] made a comparison between these datasets and found that the best performing one was BDD100k. This is because there are more diverse situations on the BDD100k compared to KITTI or most other datasets, since KITTI mostly uses daily time pictures, the model fails to generalize in the real world. Table 2.4 offers a good overview of BDD100k compared to other datasets [47].

Table 2.4: BDD100K Comparison with other street scene datasets [47].

	KITTI	City-Scapes	Apollo-Scape	Mapil-lary	BDD100k
# Sequences	22	∼50	4	N/A	**100 000**
# Images	14 999	5000 (+2000)	143 906	25 000	**120×10^6**
Multiple Cities	No	Yes	No	Yes	Yes
Multiple Weathers	No	No	No	Yes	Yes
Multiple Times of Day	No	No	No	Yes	Yes
Multiple Scene types	Yes	No	No	Yes	Yes

In addition to this, BDD100k also offers a fair distribution of classes as seen in Figure 2.24, which prevents over-fitting the model and a better generalization, this is a problem in some datasets like KITTI where the data distribution is not as good.

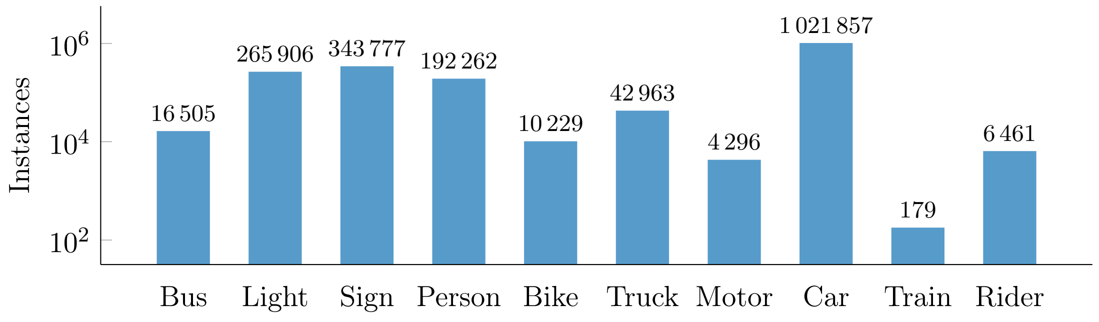


Figure 2.24: Statistical Distribution of object classes on BDD100k [47].

Deployment of Object Detectors in Autonomous Vehicles

Deep Learning-based detectors have many hardware challenges, mainly linked to the fact that AVs typically use on-board embedded computers which have limited memory availability and reduced processing capabilities due to stringent power caps, and high susceptibility to faults [13]. Some techniques, however, have been proven to work in improving the deployment efficiency like pruning, which is a widely used method for reducing the model's memory footprint and computational complexity. These techniques been shown successful by some authors such as Wang et al. [48] who proposed a sparse normalization of SSD models followed by a pruning of the channels with small scaling factor and consequently by fine-tuning. Also, Zhao et al [49] successfully pruned YOLOv4 with some different techniques.

2.3 Multiple Object Tracking (MOT)

In this section the topic of object tracking is discussed. First, an introduction is made to the key metrics used in the literature to compare different trackers; after this introduction, a discussion on the SOTA algorithms of object tracking is conducted.

2.3.1 Key Metrics

MOTP - Multiple Object Tracking Precision

The Multiple Object Tracking Precision is the average dissimilarity between all true positives and their corresponding ground truth targets. MOTP thereby gives the average overlap between all correctly matched hypotheses and their respective objects and ranges between 50% and 100% [50].

$$\text{MOTP} = \frac{\sum_{t,i} d_{t,i}}{\sum_t c_t} \tag{2.7}$$

where c_t denotes the number of matches in frame t and $d_{t,i}$ is the bounding box overlap of target i with its assigned ground truth object.

MOTA - Multiple Object Tracking Accuracy

The MOTA [51] is perhaps the most widely used metric to evaluate a tracker's performance

$$\text{MOTA} = 1 - \frac{\sum_t (FN_t + FP_t + IDSW_t)}{\sum_t GT_t} \tag{2.8}$$

Combining three sources of errors, where t is the index of frame and GT is the ground truth object count. FN is the number of false negatives, $IDSW$ is number of identity switches the and FP is the number of false positives [50].

HOTA and IDF1

In addition to both of these metrics, HOTA (Higher Order Tracking Accuracy) and IDF1 are also used. HOTA is a higher order metric for evaluating Multi-Object Tracking composed of a family of sub-metrics. There are multiple sub-metrics each with multiple equations required to calculate them. In order to save the reader from going unnecessarily in depth into this one metric, these equations will not be shown. HOTA scores typically align better with human visual evaluation of tracking performance [52]. The metric IDF1 is the ratio of correctly identified detections over the average number of ground-truth and computed detections.

2.3.2 IOU Tracker

The Intersection-Over-Union (IOU) tracker uses the IOU values among the detector's bounding boxes between two consecutive frames to perform the association between them or assign a new target ID if no match is found. This tracker includes a logic to handle false positives and false negatives from the object detector. However, this can be considered as the bare-minimum object tracker, which may serve as a baseline only to compare other trackers [53].

2.3.3 NvDCF Tracker

NvDCF tracker is a visual tracker based on the discriminative correlation filter (DCF) [54] which learns target-specific correlation filters and uses it to localize the same target in the next frames. This correlation filter learning and localization is usually carried out on a per-object basis in a typical MOT implementation, creating a potentially large number of small CUDA kernel launches when processed on GPU. This inherently poses challenges in maximizing GPU utilization, especially when a large number of objects from multiple video streams are expected to be tracked on a single GPU. [53]

NvDCF addresses this issues since its GPU-accelerated operations are designed to execute in batch processing mode to maximize the GPU utilization despite the nature of small CUDA kernels in per-object tracking model.

The batch processing mode is applied in the entire tracking operations, including the bounding box cropping and scaling, visual feature extraction, correlation filter learning, and localization. This can be viewed as a similar model to the batched cuFFT (CUDA Fast Fourier Transform) or batched cuBLAS (CUDA Basic Linear Algebra Subroutine) calls, but it differs in the fact that the batched MOT execution model spans many operations in a higher level. The batch processing capability is extended from multi-object batching to the batching of multiple streams for even greater efficiency and scalability.

This tracker is compatible with NVIDIA development kit, which makes it easier to stack multiple detectors; this means taking the detection from a primary detector (PGIE) and feed it to a secondary detector (SGIE). This will be further discussed in the section talking about the NVIDIA Hardware and Software. Thanks to its visual tracking capability, the NvDCF tracker can localize and keep track of the targets even when the detector in PGIE misses them (i.e., false negatives) for potentially an extended period of time caused by partial or full occlusions, resulting in more robust tracking. Shadow Tracking is when a target is still being tracked in the background for a period of time even when the target is not associated with a detector object. The enhanced robustness characteristics of NvDCF allow the use of a higher maxShadowTrackingAge (max time a object is being tracked in the background) for longer-term object tracking. In addition to this, NvDCF also allows for PGIEs detection interval to be higher, only at the cost of slight degradation in accuracy. In addition to the visual tracker module, the NvDCF tracker employs a Kalman filter-based state estimator to better estimate and predict the states of the targets [53].

NvDCF tracks each target by defining a search region around its predicted location in the next frame large enough for the same object to be detected in that region according to the following equations.

$$\text{SearchRegion}_{\text{ width }} = w + \text{ searchRegionPaddingScale } \times \sqrt{w \times h} \tag{2.9}$$

$$\text{SearchRegion}_{\text{ height }} = h + \text{ searchRegionPaddingScale } \times \sqrt{w \times h} \tag{2.10}$$

Where the h and w are height and width of the previous frame object boundig box. This tracker is proposed by NVIDIA and is part of the NVIDIA DeepStream SDK; this SDK will be approached in greater detail in later chapters, this makes its integration with NVIDIA hardware much simpler than a normal tracker, on the other hand, there is currently no published paper with any metrics to compare it to other SOTA trackers; however, it is described to have comparable performance to DeepSORT according to NVIDIA Documentation.

2.3.4 DeepSORT Tracker

Simple Online and Real-time Tracking (SORT) is a pragmatic approach to multiple object tracking with a focus on simple effective algorithms [55]. It performs Kalman filtering in image space and frame-by-frame data association using the Hungarian method with an association metric that measures bounding box overlap, achieving favorable performance at high frame rates. This way, SORT manages to combine location and motion cues in a simple way [56]. Despite this high performance, SORT returns a high number of identity switches. DeepSORT authors improved on the classical SORT by integrating appearance information into the SORT algorithm; by doing this, they are able to track objects through longer periods of occlusions, reducing the number of identity switches. This is done through an offline pre-training stage where a deep association metric is learned on a large scale person re-identification dataset. These additions to the original SORT improved identity switches by 45%.

2.3.5 BYTE and ByteTrack Tracker

Most methods obtain identities by associating detection boxes whose scores are higher than a threshold. The objects with low detection scores, e.g. occluded objects, are simply town away, which brings non-negligible true objects, missing and fragmented trajectories. The authors of [56] propose an association method that tracks by associating almost every detection box instead of only the high score ones. Utilizing similarities with tracklets (small set of paths associated with individual detections in consecutive frames) to recover true objects and filter out the background detection. This method achieved an improvement in tracker performance on a large number of SOTA trackers. The authors in [56] proposed a new tracker named ByteTrack, that achieved state-of-the-art performance on MOTA17 and MOTA20 with 30 FPS running on a V100 GPU, as shown in Figure 2.25. ByteTrack also achieved SOTA results on MOT20, HiEve and BDD100k. This tracker is built by equipping the high-performance detector YOLOX [43] with the association method BYTE.

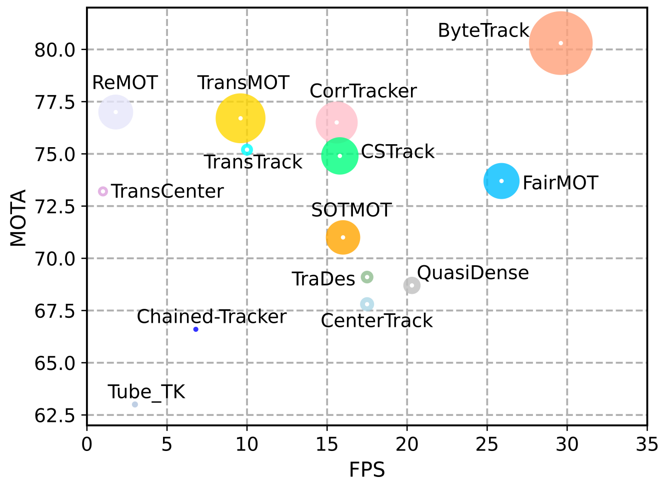


Figure 2.25: MOTA-IDF1-FPS comparisons of different trackers in the test set of MOT17. The horizontal axis is FPS (running speed), the vertical axis is MOTA, and the radius of circle is IDF1. ByteTrack achieves 80.3 MOTA, 77.3 IDF1 on MOT17 test set with 30 FPS running speed, outperforming all previous trackers [56].

One of the great contributions of this paper come from the way it performs the data association. To efficiently track, data association is an essential task that computes the similarity between tracklets and detection boxes, leveraging different strategies to match them according to the similarity. SORT combines location and motion cues by adopting a Kalman filter to predict the location of the tracklets in the new frame, then computes the IoU between the detection boxes and the predicted boxes as the similarity. DeepSort [55] adopts a stand-alone RE-ID model to extract appearance features from the detection boxes. [56]. After similarity computation, the matching strategy assigns identities to the

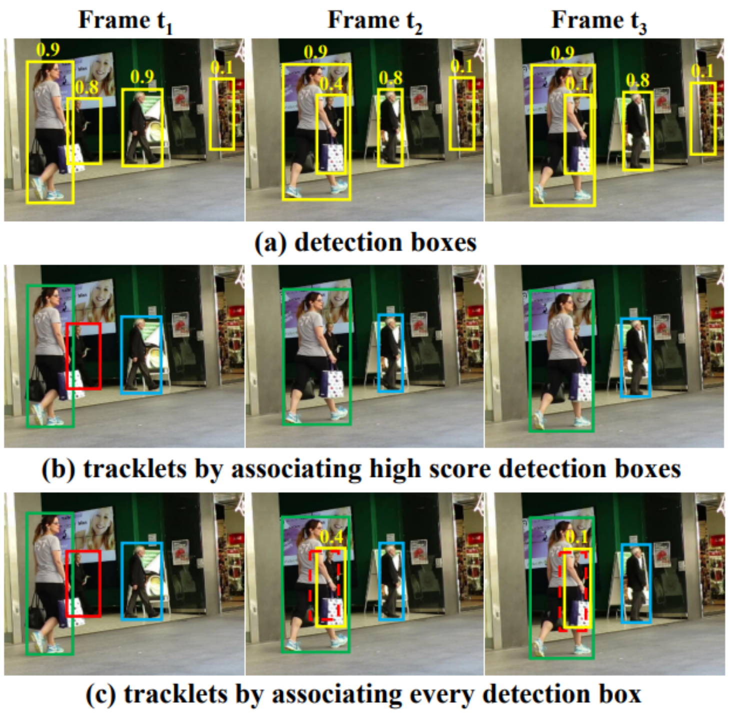

Frame t_1
Frame t_2
Frame t_3
0.9
0.8
0.9
0.1
0.4
0.8
0.1
(a) detection boxes
(b) tracklets by associating high score detection boxes
(c) tracklets by associating every detection box

Figure 2.26: Examples of the method proposed in [56] which associates every detection box. (a) shows all the detection boxes with their scores. (b) shows the tracklets obtained by previous methods which associates detection boxes whose scores are higher than a threshold, i.e. 0.5. The same box color represents the same identity. (c) shows the tracklets obtained by BYTE method. The dashed boxes represent the predicted box of the previous tracklets using Kalman Filter. The two low score detection boxes are correctly matched to the previous tracklets based on the large IoU. [56].

objects. This can be done by the Hungarian Algorithm [57] or greedy assignment [58]. DeepSORT [55] proposes a cascaded matching strategy that first matches the detection boxes to the most recent tracklets and then to the lost ones. Many authors proposed even different methods to perform the matching, these methods, however, focus on how to design better association methods. The authors of [56] argue that, the way detection boxes are utilized, determines the upper bound of data association, so the authors focus, instead, on how to make full use of detection boxes from high scores to low ones. The authors call this new data association method BYTE. Using all the detection boxes; they first associate the high score detection boxes to the tracklets. Figure 2.26 illustrates the difference this data association method provides. Doing so, some tracklets get unmatched because they do not match to an appropriate high score detection box, which usually happens when occlusion, motion blur or size changing occurs. The authors then associate the low score detection boxes and these unmatched tracklets to recover the objects in low score detection boxes and filter out background, simultaneously. BYTE is very flexible and can be implemented in combination with other trackers/detectors. In the paper the authors compared DeepSORT with BYTE by using a light YOLOX model with a modified backbone; the data association method BYTE combined with the detector

YOLOX is the foundation of what the authors call ByteTrack. This paper shows high improvement over the current SOTA and opens new doors to explore tracking with other SOTA detectors such as YOLOv5 and YOLOR.

2.3.6 OC-SORT

Most of current motion models in Multiple Object Tracking (MOT) typically assume that the object motion is linear in a small time window and needs continuous observations, so these methods are sensitive to occlusions and non-linear motion, requiring high frame-rate videos. The work in [59] shows that a simple motion model can obtain state-of-the-art tracking performance without other cues such as appearance. OC-SORT [59] provides multiple innovations to SORT, it adds an Observation-centric Online Smoothing (OOS) strategy to alleviate the error accumulation in the Kalman Filter due to lack of observations. In addition to this, the authors also incorporated the direction consistency of tracklets in the cost matrix for better matching between tracklets and observations. Finally to deal with the case of objects being untracked due to occlusion in a short time window, the authors proposed to recover them by associating their last observations with the new observations, which they refer to as Observation-Centric Recovery (OCR). The authors named this method Observation-Centric SORT or OC-SORT. It remains simple, online, and real-time but improves robustness over occlusion and non-linear motion, achieving values of 63.2 and 62.1 HOTA on MOT17 and MOT20 [59].

2.3.7 StrongSORT

The authors of [60] revisited the classical tracker DeepSORT and upgraded it from various aspects, i.e, detection, embedding and association. The resulting tracker, called StrongSORT sets new HOTA and IDF1 records on MOT17 and MOT20. In addition to this the authors also propose two plug-and-play algorithms to further refine the tracking results AFLink and Gaussian-smoothed interpolation (GSI). Applying both of this to StrongSORT originated StrongSORT++ which ranks first on MOT17 and MOT20 in terms of HOTA and IDF1 metrics as shown on Figure 2.27 and Table 2.5 [60].

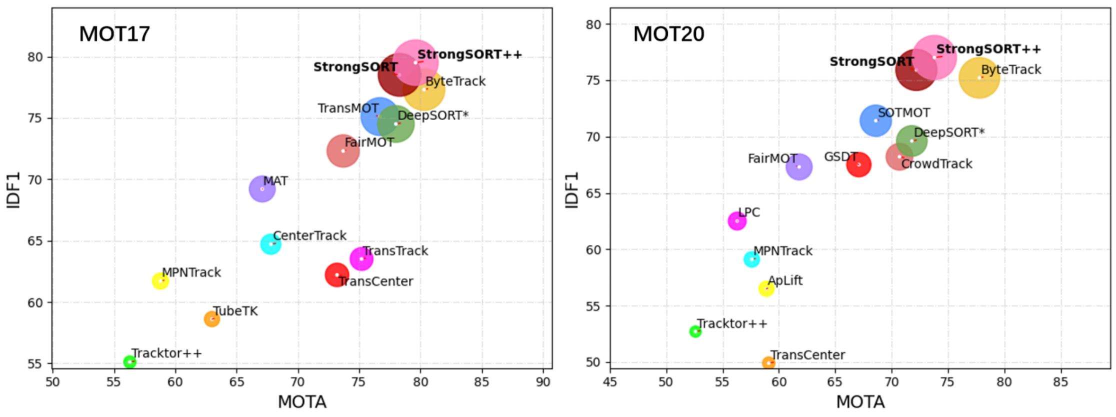


Figure 2.27: IDF1-MOTA-HOTA comparisons of state-of-the-art trackers with StrongSORT and StrongSORT++ on MOT17 and MOT20 test sets [60]. The radius of the circle is HOTA.

Table 2.5: Comparison with state-of-the-art MOT methods on the MOT20 test set. The "*" represents [60] reproduced version. The two best results for each metric are bolded and highlighted in red and blue [60].

Method	**Ref.**	**HOTA** (↑)	**IDF1** (↑)	**MOTA**(↑)	**AssA** (↑)	**DetA** (↑)	**IDs** (↓)	**FPS** (↑)
SORT	ICIP2016	34	39.8	43.1	31.8	37	4,852	**143.3**
DAN	TPAMI2019	39.3	49.5	52.4	36.3	43.1	8,431	6.3
TPM	PR2020	41.5	52.6	54.2	40.9	42.5	1,824	0.8
DeepMOT	CVPR2020	42.4	53.8	53.7	42.7	42.5	1,947	4.9
Tracktor++	ICCV2019	44.8	55.1	56.3	45.1	44.9	1,987	1.5
TubeTK	CVPR2020	48	58.6	63	45.1	51.4	4,137	3
ArTIST	CVPR2021	48.9	59.7	62.3	48.3	50	2,062	4.5
MPNTrack	CVPR2020	49	61.7	58.8	51.1	47.3	**1,185**	6.5
CenterTrack	ECCV2020	52.2	64.7	67.8	51	53.8	3,039	3.8
TransTrack	arxiv2021	54.1	63.5	75.2	47.9	61.6	3,603	59.2
TransCenter	arxiv2021	54.5	62.2	73.2	49.7	60.1	4,614	1
GSDT	ICRA2021	55.5	68.7	66.2	54.8	56.4	3,318	4.9
PermaTrack	ICCV2021	55.5	68.9	73.8	53.1	58.5	3,699	11.9
MAT	NC2022	56	69.2	67.1	57.2	55.1	1,279	11.5
CSTrack	arxiv2020	59.3	72.6	74.9	57.9	61.1	3,567	15.8
FairMOT	IJCV2021	59.3	72.3	73.7	58	60.9	3,303	25.9
ReMOT	IVC2021	59.7	72	77	57.1	62.8	2,853	1.8
CrowdTrack	AVSS2021	60.3	73.6	75.6	59.3	61.5	2,544	140.8
CorrTracker	CVPR2021	60.7	73.6	76.5	58.9	62.9	3,369	15.6
RelationTrack	arxiv2021	61	74.7	73.8	61.5	60.6	1,374	8.5
TransMOT	arxiv2021	61.7	75.1	76.7	59.9	63.7	2,346	1.1
GRTU	ICCV2021	62	75	74.9	62.1	62.1	1,812	3.6
MAATrack	WACVw2022	62	75.9	79.4	60.2	64.2	1,452	**189.1**
BytcTrack	arxiv2021	63.1	77.3	**80.3**	62	**64.5**	2,196	29.6
DeepSORT*	ICIP2017	61.2	74.5	78	59.7	63.1	1,821	13.8
StrongSORT		63.5	78.5	78.3	63.7	63.6	1,446	7.5
StrongSORT+		**63.7**	**79**	78.3	**64.1**	63.6	1,401	7.4
StrongSORT++		**64.4**	**79.5**	**79.6**	**64.4**	**64.6**	**1,194**	7.1

Chapter 3

Experimental Infrastructure

In this chapter, the experimental infrastructure used in the development of this dissertation is discussed; both software and hardware components utilized are described. It starts with an introduction to the NVIDIA hardware utilized and its associated SDKs, followed by an analysis of all the software tools utilized to train and deploy the models.

3.1 NVIDIA SDK

Self-driving cars and other autonomous machines benefit from portable compact edge devices, these are, physical devices where the algorithms run at the local level (where the data is being collected, as opposed to a cloud remote service). It is not easy to fit a big computer into many types of autonomous systems, which is especially true when talking about cars; so, high-capability portable solutions are needed to deploy algorithms that can run in real-time. For this dissertation a specific hardware was provided. This hardware comes with several advantages and disadvantages which are explored in this section.

Jetson AGX Xavier

The NVIDIA Jetson AGX Xavier developer kit shown in Figure 3.1 is a solution made by NVIDIA to easily deploy end-to-end AI robotics applications for robots, drones, and other autonomous machines. It is supported by NVIDIA JetPack and DeepStream SDKs, as well as CUDA, CUDNN and TensorRT. It offers the performance of a high-end GPU workstation, in under 30 W, capable of more than 30 trillion operations per second, perfect for Deep Learning and computer vision tasks.

JetPack

JetPack SDK is the SDK pre-packaged with any Jetson NVIDIA device; it includes:

- Sample Linux filesystem with NVIDIA drivers
- AI and Computer Vision libraries and APIs
- Developer tools

JetPack also comes with following Libraries:

Figure 3.1: NVIDIA Jetson Xavier AGX Development Kit.

- TensorRT and cuDNN for high performance Deep Learning applications
- CUDA for GPU accelerated applications across multiple domains
- Multimedia API package for camera applications and sensor driver development
- VisionWorks and OpenCV for visual computing applications

JetPack uses L4T (Linux for Tegra), an NVIDIA Jetson Linux driver package with Linux operating system based on Ubuntu 18.04 and comes with CUDA-X accelerated libraries and APIs for Deep Learning, Computer Vision, and Accelerated Computing.

JetPack includes TensorRT which is an SDK for high performance Deep Learning inference; it includes a Deep Learning inference optimizer that delivers low latency and high throughput, this can be used for example, in image classification, segmentation, and object detection Neural Networks. NVIDIA TensorRT-based models perform up to 36 times faster than CPU-only platforms during inference [61]. Figure 3.2 shows the TensorRT pipeline in the deployment of a Neural Network.

The cuDNN (CUDA Deep Neural Network library) provides high-performance primitives for Deep Learning frameworks. It provides highly tuned implementations for standard routines such as forward and backward convolution, pooling, normalization, and activation layers. Along with these features JetPack also comes with the support of DeepStream SDK.

DeepStream SDK

The NVIDIA DeepStream SDK is a streaming analytics toolkit developed by NVIDIA. The DeepStream SDK accelerates development of scalable applications, making it easier for developers to build core deep learning networks instead of designing end-to-end applications from scratch. The SDK is an extensible collection of hardware-accelerated plugins that interact with low-level libraries to optimize performance. Figure 3.3 shows a simplified scheme of the DeepStream pipeline.

DeepStream introduces Deep Neural Networks and other complex processing tasks directly into a stream processing pipeline. This enables near real-time analytics on video and other sensor data. A DeepStream "application" is a set of modular plugins connected

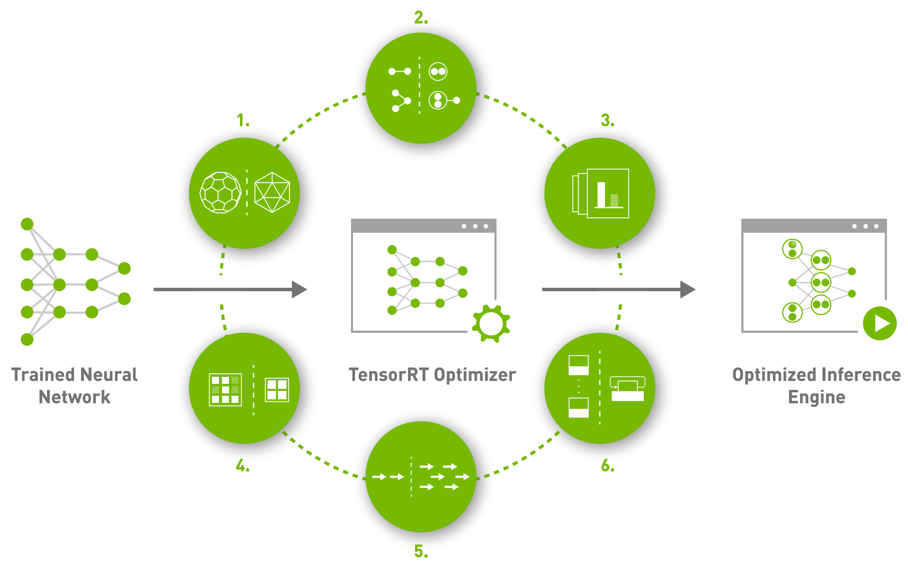


Figure 3.2: NVIDIA TensorRT pipeline.

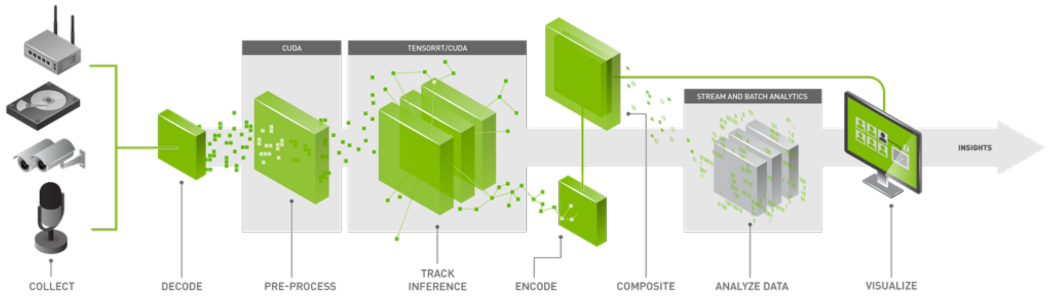


Figure 3.3: Deepstream Workflow [53].

to form a processing pipeline. Each plugin represents a functional block. Examples of these functional blocks include multi-stream batching, inference using TensorRT, and decoding. Hardware accelerated plugins interact with the underlying hardware to deliver maximum performance [53].

As an added bonus, the DeepStream SDK comes packed with integrated trackers. These trackers are made compatible with DeepStream by NVIDIA. This brings numerous advantages since it means they are optimized to work on the DeepStream pipeline with inference speed and accuracy in mind; in addition to this, it makes the integration of different trackers with different detectors much easier. Considering the amount of detectors and trackers required to be tested to take meaningful conclusions, DeepStream is a great candidate technology to develop the final solution.

Since all the connections between these low-level libraries, SDKs and hardware technicalities, can be hard to understand, a simplified summary of the of the Jetson hardware can be seen in Figure 3.4.

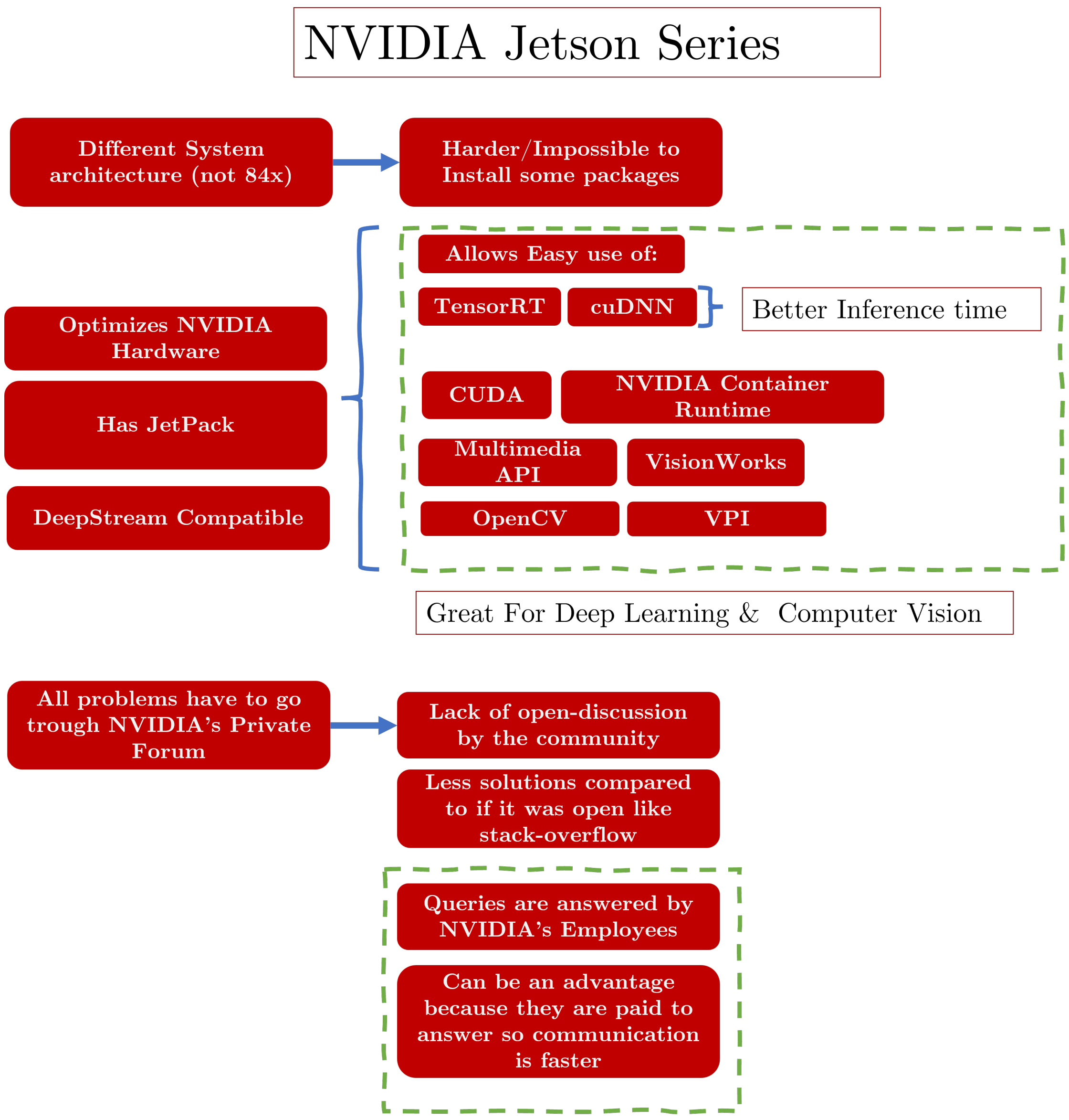

NVIDIA Jetson Series
Different System architecture (not 84x)
Harder/Impossible to Install some packages
Optimizes NVIDIA Hardware
Has JetPack
DeepStream Compatible
Allows Easy use of:
TensorRT
cuDNN
Better Inference time
CUDA
NVIDIA Container Runtime
Multimedia API
VisionWorks
OpenCV
VPI
Great For Deep Learning & Computer Vision
All problems have to go trough NVIDIA's Private Forum
Lack of open-discussion by the community
Less solutions compared to if it was open like stack-overflow
Queries are answered by NVIDIA's Employees
Can be an advantage because they are paid to answer so communication is faster

Figure 3.4: Jetson systems summary scheme.

TAO Toolkit

The NVIDIA TAO Toolkit is a CLI (command line interface) and Jupyter notebook based solution, that abstracts away the AI/Deep Learning framework complexity, allowing the fine-tuning on high-quality NVIDIA pre-trained AI models with only a fraction of the data compared to training from scratch [53]. With all these topics introduced a simple summary of the DeepStream SDK is provided in Figure 3.5.

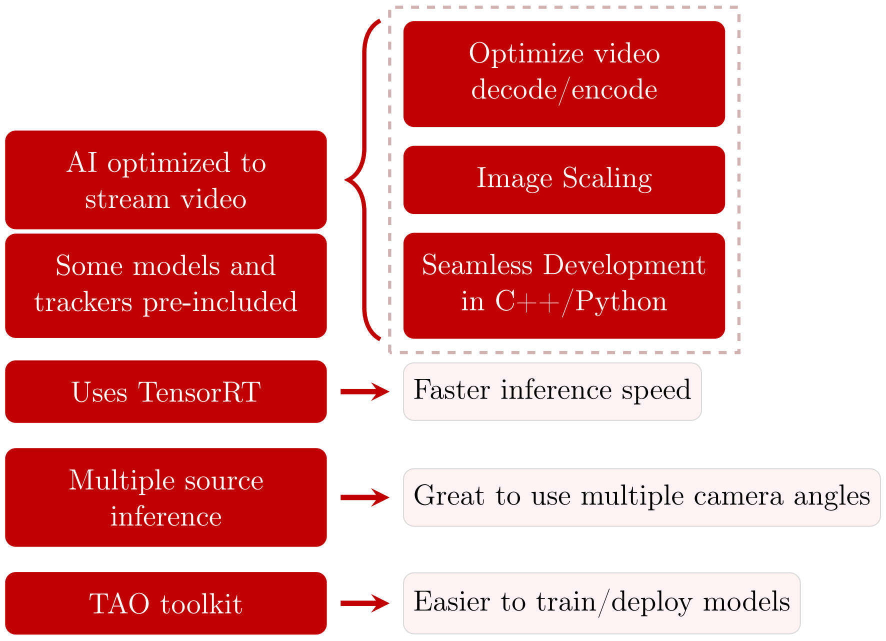

Optimize video decode/encode
Image Scaling
Seamless Development in C++/Python
AI optimized to stream video
Some models and trackers pre-included
Uses TensorRT
Faster inference speed
Multiple source inference
Great to use multiple camera angles
TAO toolkit
Easier to train/deploy models

Figure 3.5: DeepStream SDK simplified summary scheme.

3.2 Camera Hardware and Software Tools

JupyterLab

JupyterLab is a web-based interactive development environment for notebooks, code, and data. When training Deep Learning models and deployment pipelines, a lot of re-occurring commands and data plots need to be done. Jupyter-Notebooks with Python snippets allow for a quick seamless interaction with Python code and Python plots.

DeepLar

The Laboratory of Automation and Robotics (LAR) has a computer, shown in Figure 3.6, explicitly crafted for Deep Learning tasks; because of its high GPU power, this computer allows for the training of large Deep Learning models.

The computer specs are the following:

- 4x NVIDIA RTX2080TI
- AMD Thradripper 2850 Extreme
- 128Gb DDR4 RAM
- 512Gb SSD + 4Tb HDD

Cameras

Multiple tests were done with different cameras, some of them were performed with the e-CAM130-CUXVR. Four Synchronized 4K Cameras for Jetson AGX Xavier shown in Figure 3.7. These cameras have up to four 13 MP 4-Lane MIPI CSI-2 camera boards, and

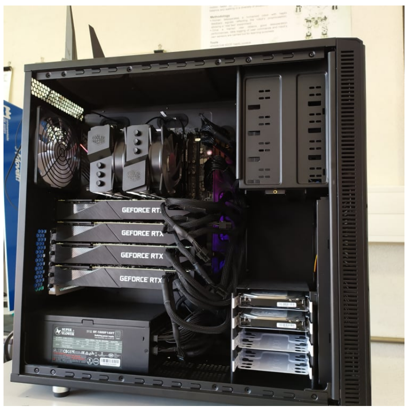


Figure 3.6: DeepLar Computer.

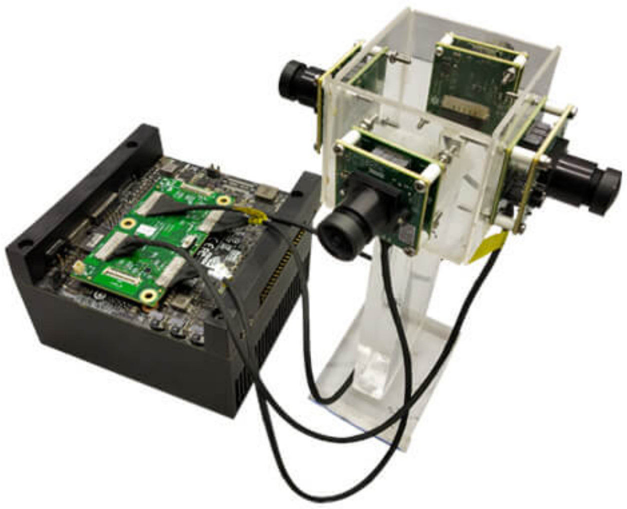


Figure 3.7: e-CAM130-CUXVR plugged into NVIDIA Jetson AGX Xavier.

4-lane MIPI camera modules that can be synchronously streamed in 4K resolution, which will be the best fit for a high-end multi-camera solution; they also offer Gstreamer-1.0 support for video recording and network streaming [62].

SSD

To adress the AGX Xavier low available memory storage problem, a NVMe M2 SSD was installed and mounted.

PyTorch

PyTorch is an open-source machine learning framework that accelerates the path from research prototyping to production deployment. This framework was chosen due to its "dominance" in the research field. According to a study data from "paperswithcode", in March 2022, 63% of the papers submitted utilized PyTorch while only 7% utilized TensorFlow [63]. Most recent papers have PyTorch implementations, therefore PyTorch was selected over TensorFlow as the framework of choice. This came with the added bonus of having containers developed by NVIDIA that can be used as a base to deploy the final solution.

Docker and NVIDIA Docker

Installing libraries, frameworks, packages or compilers sometimes can be difficult, especially when working with different computer architectures. Docker uses OS-level virtualization to deliver software in packages called containers. These containers can be used to quickly deploy code across machines and often eliminate the problem of complicated installations when setting up a development environment. Docker proved especially helpful when training multiple Deep Learning algorithms, which required different dependencies / different PyTorch versions and different compilers. These are problems that could not be easily solved with typical virtual environments. It also allowed quick deployment and testing on Jetson AGX Xavier and DeepLar. To utilize the power of DeepLar's GPUs inside the Docker containers, nvidia-docker had to be installed.

Notion

Notion is a project management and note-taking software designed to help members of a company or organization coordinate deadlines, objectives, and assignments for the sake of efficiency and productivity. In the context of this dissertation, it was used as a tool to communicate between the student and the professor. Multiple documentation files were written on it, including guides, notes, and a database with the model's results. This was chosen over a conventional blog due to its ease of use, snippet of code integration, and automatic web deployment/markdown/HTML exports. For further reference check the "Journal"[1] created.

WSL2

Windows Subsystem for Linux 2 (WSL2) lets developers run a GNU/Linux environment including most command-line tools, utilities, and applications directly on Windows, unmodified, without the overhead of a traditional virtual machine or dual-boot setup. Since the laptop used on this dissertation had some hardware limitations and incompatibilities with the old Ubuntu versions needed to work with the Jetson (due to Unity / NVIDIA incompatibilities), WSL2 was used. Doing so, avoided the need for dual booting, allowing rapid development, seamlessly switching through Ubuntu versions, and sharing of the Windows/Linux file system.

SSH Workflow

A good workflow needed to be established to efficiently train multiple models with multiple dependencies. Remote connection to the DeepLar was required, as well as docker containers with the numerous packages needed to train each algorithm. A remote connection through the Host and Remote PC was established through SSH, where the

[1] https://narrow-sled-25f.notion.site/d86245b298de4d6c90620a76004c87d4?v=9531e3a47bb04e8fba0672138e26e524

Remote PC had the GPUs, which were accessed with Docker. A visual chart of this can be seen in Figure 3.8.

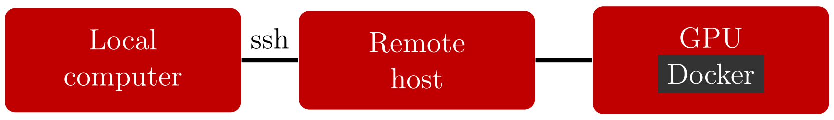


Figure 3.8: SSH Remote Connection Workflow.

W&B – Weights and Bias

With the current DeepLar setup it was impossible to analyse the model metrics in real-time; first, the user would have to wait until the training was finished, then create the plots, and then transfer them over to a local PC to open the plots. To allow researchers to collaborate and respond to the model metric easily, a W&B integration was done on DeepLar. This platform allows the user to access the plots of the model in real time, check GPU usage and allow multiple users to collaborate on interactive plots (with zoom and editable axis capability) maintaining all the information that would be lost in the typical image file plot. For further references on how to set up a Weights and Bias work environment check the official W&B documentation[2].

[2]https://wandb.ai/site

Chapter 4

Solution, Experiments and Results

This chapter contains all the tests and procedures used to develop the final solution; this includes an in-depth description of all the choices taken based on the results of the tests. The chapter starts with the preliminary tests and set-ups, which includes an explanation of the technology stacks used and the set-up of the workflow environment. In addition to this, since there are several dependencies limitations and conflicts that may appear with the hardware utilized, some preliminary tests were performed. The aim of these tests was to determine which algorithms could be successfully deployed in the NVIDIA Jetson AGX Xavier and evaluate their inference speed.

With the results of these tests, some model architectures were selected based on their inference speed. The training of these models and all the data pre-processing required to train them are presented in the following subsections. Finally, the results of the deployment of these models is shown in the last section, this includes the creation of the necessary engines, the implementation of the tracking algorithms and a qualitative/quantitative analysis of the solution proposed.

4.1 Preliminary Tests

When designing a solution for the problem of object detection in tracking in ATLASCAR2, multiple things have to be considered. The desired properties of the solution for this thesis would have in mind the following:

- Find a solution that optimizes the unique features of the hardware used.
- Find an optimal trade-off between accuracy and speed.
- Find a solution with longevity (allowing future updates).
- Find a solution that allows easy sharing of metadata (for further development of other robotic systems).
- Within the scope of a given solution, optimize it and fine-tune it for better performance.

4.1.1 Memory Problems

With these purposes in mind, the first step was to analyze the characteristics of the Jetson AGX Xavier. A summary of this analysis is shown in Figure 4.1. Since the

Jetson AGX Xavier has some unique characteristics and custom NVIDIA optimization technologies, performing an analysis on its characteristics is important when deciding which technologies to use and which development methodologies to use.

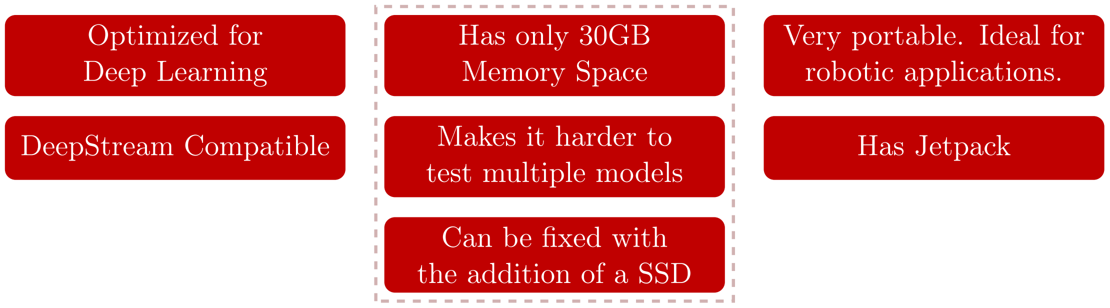

Optimized for Deep Learning
DeepStream Compatible
Has only 30GB Memory Space
Makes it harder to test multiple models
Can be fixed with the addition of a SSD
Very portable. Ideal for robotic applications.
Has Jetpack

Figure 4.1: Summary of Jetson AGX Xavier characteristics.

One of the problems linked with this equipment is the low storage memory, this makes it harder to compile and test multiple models or even impossible to install the dependencies needed to run some of them. In addition to this, due to the different system architecture of the Jetson, even if there is enough space to install the dependencies, their installation is still difficult or sometimes impossible due to incompatibilities. To address this issue, first, an SSD was installed and mounted on the Jetson AGX Xavier. After this installation, some issues remain. Even though model files can be booted from the SSD, all the installation requirements (packages, libraries, etc.) are still being installed into the main storage drive and their installation is still a difficult process.

To fix this, Docker containers were used to set up all the dependencies needed, and a mapping was created from the Docker Daemon directly to the SSD storage, allowing the deployment of multiple containers with the required libraries all directly from the SSD, solving the storage problem and the dependencies installation problem.

4.1.2 Deployment Technology Decision

After addressing the storage problem, the next stage was to determine the technology stacked in which to deploy the final solution. There are multiple options for this stack, the code could be deployed directly, using raw PyTorch, or using an optimization engine such as TensorRT combined with an optimized camera streaming pipeline, such has DeepStream. This choice is important because it drastically affects performance speed, the development decisions, and the time required to implement the algorithms. With real-time optimized camera inference in mind an analysis was performed on whether or not DeepStream should be used; the results of this analysis are summarized in Figure 4.2.

The conclusion was that, to develop a flexible code with longevity, choosing DeepStream would provide great benefit. The DeepStream pipeline works like "Lego modules", which makes it easier to swap a portion of the code (for example a detector or a tracker) while keeping the rest of the pipeline intact; this abstraction lowers the code complexity and allows solutions to evolve over the years more easily without breaking the entire pipeline. As the technology progresses, more and more models will be added to the DeepStream SDK; these can be swapped with the ones used in this dissertation, and the same goes

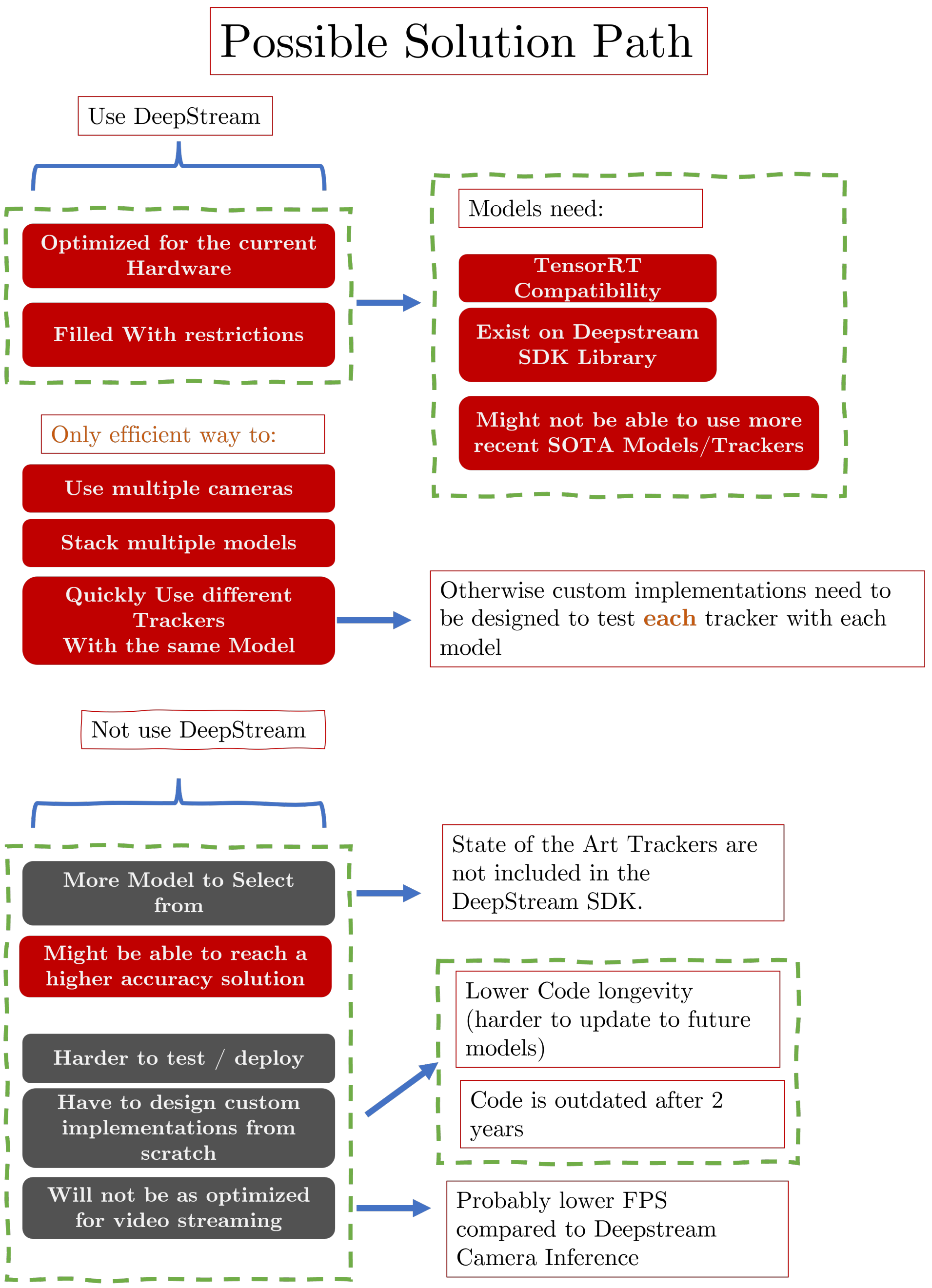

Possible Solution Path
Use DeepStream
Optimized for the current Hardware
Filled With restrictions
Models need:
TensorRT Compatibility
Exist on Deepstream SDK Library
Might not be able to use more recent SOTA Models/Trackers
Only efficient way to:
Use multiple cameras
Stack multiple models
Quickly Use different Trackers With the same Model
Otherwise custom implementations need to be designed to test each tracker with each model
Not use DeepStream
More Model to Select from
Might be able to reach a higher accuracy solution
Harder to test / deploy
Have to design custom implementations from scratch
Will not be as optimized for video streaming
State of the Art Trackers are not included in the DeepStream SDK.
Lower Code longevity (harder to update to future models)
Code is outdated after 2 years
Probably lower FPS compared to Deepstream Camera Inference

Figure 4.2: Possible solution path summary on whether to use or not to use DeepStream.

for the trackers. With this approach, the code written can be updated frequently with the most recent algorithms and models with minimal effort.

It is worth keeping in mind that many of the solutions tested in this thesis were not included in the DeepStream SDK and custom implementations were used. However, it is expected that these algorithms will eventually be implemented in the DeepStream SDK since it is in NVIDIA's best interest to keep updating its products. Since DeepStream is specifically designed to optimize the streaming of video services on NVIDIA hardware, choosing DeepStream maximizes the hardware capabilities.

Multiple camera testing was also in the scope of this dissertation; deploying multiple camera streams outside DeepStream requires additional configurations and writing additional code for each specific algorithm. In addition to this, doing the implementation manually from scratch can risk a possible bottleneck in the inference speed when the multiple camera streams are not optimized on different computer threads. Considering the time frame of this thesis, developing code for multiple camera inferences for each specific algorithm would be too time-consuming, taking time away from more important things such as optimizing the models and running different experiments to determine the best trade-offs between speed and accuracy. The DeepStream SDK also comes with pre-compatible SOTA trackers and a tracker inference plugin; this makes it easy to implement different trackers with different detectors; the three trackers included are NvDCF, IoU and DeepSORT. Since this thesis aims to find the best solution for object detection and tracking, testing different models with different trackers is necessary; this makes using DeepStream the best choice to make it possible.

4.2 Neural Network training and data pre-processing

This section addresses all the topics related to the training of the Neural Networks as well as all the data pre-processing required to train them.

4.2.1 Pre-trained models inference tests

After the decision to use DeepStream was made, the next logical step was to test the models, presented in the SOTA chapter, that have DeepStream compatibility and have reasonable accuracy. The results of the DeepStream deployment of TensorRT FP32 (floating point precision of 32 bits) models with a video stream are shown in Table 4.1.

During these tests, different inference intervals were tested; this means turning the detector off for a specific batch interval then turning it on again. A few important things to keep in mind are that the mAP presented is evaluated in the COCO dataset with the pre-trained models offered by some of the authors, and not the on an urban driving dataset like BDD100K. The Faster-RCNN with a VGG16 backbone pre-trained model had no mAP provided by the author; however, through a qualitative analysis, the model was discarded from further evaluations since many detections were missing despite its higher resolution; in addition to this, the FPS were very low. One of the models (yolov5-n6) could not infer in the DeepStream pipeline, possibly due to some incompatibility in the model architecture and the DeepStream inference plugin. Through this early analysis, it was concluded that:

- YOLOR-CSP* ,YOLOv5-m and Scaled-YOLOv4 were the best model architecture options to consider.
- Changing the inference interval drastically boosts the FPS.
- Resolutions higher than 640 pixels wide are not the best fit since they can only perform at low FPS using the Jetson AGX Xavier.
- When changing the resolution of YOLOv4, there seems to be a considerable increase in speed when dropping the resolution from $608x608$ to $512x512$.

Table 4.1: Initial Inference Speed Tests. mAP@50 is the mAP value measured on the COCO datasets of these pre-trained weights provided by the original authors. Int means the Inference interval (On-Off). NW means "Non-Working"; ND means "No Detections".

Model Name	**Algorithm**	**Reso-lution**	**FPS Int=0**	**FPS Int=2**	**mAP@50** [1]
yoloR_p6	yolor	1280	4.44	13	**73.30%**
yoloR_csp_X*	yolor	512	10.75	31.84	69.90%
yoloR_csp_X	yolor	512	10.83	31.73	69.60%
yoloR_csp	yolor	512	19.44	53.39	67.60%
yoloR_csp*	yolor	512	19.35	53.83	68.70%
yolov4-csp-x-swish	Scaled yolov4	640	19.43	54.15	69.90%
yolov4-p6	Scaled yolov4	1280	2.17	6.85	72.10%
yolov4-csp-swish	Scaled yolov4	640	11.3	32.18	68.70%
yolov4-csp	Scaled yolov4	512	19.55	52.44	64.80%
yolov4-csp	Scaled yolov4	640	10.88	31.69	67.40%
yolov4	yolov4	608	10.11	29.33	65.70%
yolov4	yolov4	512	17.15	47.4	64.90%
yolov4	yolov4	416	20.14	55.6	62.80%
yolov4	yolov4	320	27.15	60.69	60 %
yolov4-tiny	yolov4-tiny	416	**60.27**	60.93	40.20%
yolov5-n	yolov5	640	60.3	61.6	45.70%
yolov5-s	yolov5	640	53.56	60.47	56.80%
yolov5-m	yolov5	640	22.17	60.43	64.10%
yolov5-l	yolov5	640	12.1	35.31	67.30%
yolov5-n6	yolov5	1280	NW	ND	54.40%
Faster-RCNN VGG16	faster-rcnn	1920×1080	7.81	14.4	-

[1] Measured on the COCO dataset.
[*] The "*" is how the authors of [38] assigned the new training strategy they are currently developing.

4.2.2 Data pre-processing

The COCO dataset includes the traffic object classes (car, pedestrian, traffic sings, etc.) so, in theory, the COCO pre-trained weights could be used on the final solution. However, COCO was not created with the urban driving scenario in mind, due to this, it does not include the different weather conditions and extreme edge cases that are normally faced in a real life urban environment. For this reason, choosing a large dataset, with different scenarios is important for model accuracy in a real life scenario.

According to the dataset analysis made in Chapter 2, BDD100k was a great candidate for the training and validation of the models. So, this ended up being the chosen dataset for this dissertation. It is important to select a dataset that reflects an urban scenario since this is the environment ATLASCAR2 will be driving at.

The BDD100K dataset requires some preparation to be trained with YOLO algorithms; some of the YOLO implementations are on Darknet (an open-source Neural Network framework built in C by the author of the original YOLO) and the others on PyTorch;

YOLOv4 Darknet uses a different format from Scaled-YOLOv4 PyTorch, which uses a different format from YOLOv5 PyTorch and YOLOR. So, conversion scripts had to be created to pre-process these labels into the specific algorithms/frameworks formats.

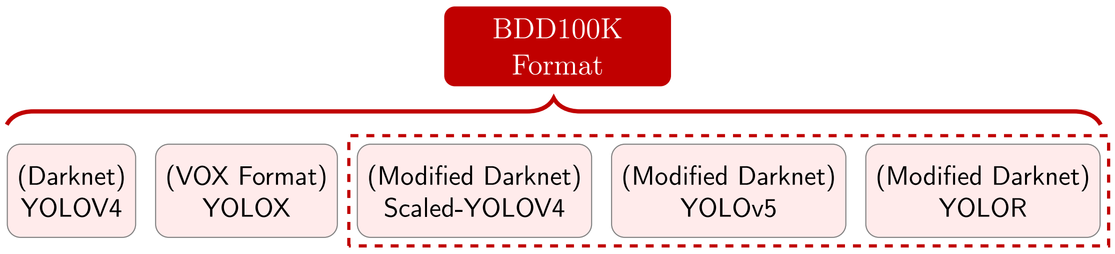


Figure 4.3: Format conversion needed to train each different YOLO from BDD100K.

As seen in Figure 4.3, each YOLO has its own format; in the case of YOLOv5, YOLOR and Scaled-YOLOv4, the format is the same; this format rises from the conversion of the Darknet framework to PyTorch by YOLOv5 Authors. A specific Darknet format must be used to train YOLOv4; on the other hand, YOLOX uses the VOX dataset format. A script to convert to each of these formats was created. With this script, the conditions to train the algorithms were set.

Portainer and Dockerized YOLOs with W&B

The training rig provided for this thesis was DeepLar, so some setup work had to be done in order to train the algorithms. DeepLar is installed with OpenSuse and some legacy versions of required packages/libraries like pip and Python. To train some of these algorithms newer versions had to be installed; on top of this, since some of the YOLOs require the installation of the mish activation function with CUDA support (to allow the GPUs to be used during training), creating a simple python virtual environment was not enough since extra NVIDIA drivers had to be installed. Installing these packages/drivers on a normal computer would be feasible; however, when dealing with a server utilized by multiple people, the odds of an installation corrupting or breaking the entire system are large. To address this issue, a Docker-Based solution was selected. With the type of virtualization provided by Docker, each individual using DeepLar could work with a PyTorch image with their respective libraries without worrying about breaking the system. After this setup, Docker containers were created with the dependencies of each of the YOLOs pre-selected before; in addition to this, Docker-Volumes were created containing the dataset; this way, the dataset could be accessed inside the containers.

DeepLar does not have a GUI, so, visualization of different plots is difficult; to address this issue a virtual JupyterLab was created for better testing and visualization (testing different batch sizes etc..). Before this, the users of DeepLar would have to create the plots with a Python script on the command line then transfer them trough SSH to their computer and, finally, open them. With JupyterLab, the plots can be opened directly on the browser of the local PC.

To create an intuitive easy to use work environment some extra tools were utilized. Portainer is a container management tool that was utilized for easier visibility of the multiple Docker containers status; the Portainer setup was performed according to the official documentation[1]. In addition to this, a W&B connection to DeepLar was established; this way, the model training plots could be accessed in real-time, through a cellphone or any other remote PC, with an interactive GUI.

4.2.3 Training of the Pre-Selected Models

This subsection goes in-depth about the decisions made during the training of multiple models.

First Trained model

YOLOv5 was the model of choice for the first run since it offered greater community support and more mature documentation. For this run, the entire BDD100K dataset was used, that is, the 100 000 images. As seen in Figure 4.4, some of the classes are severely

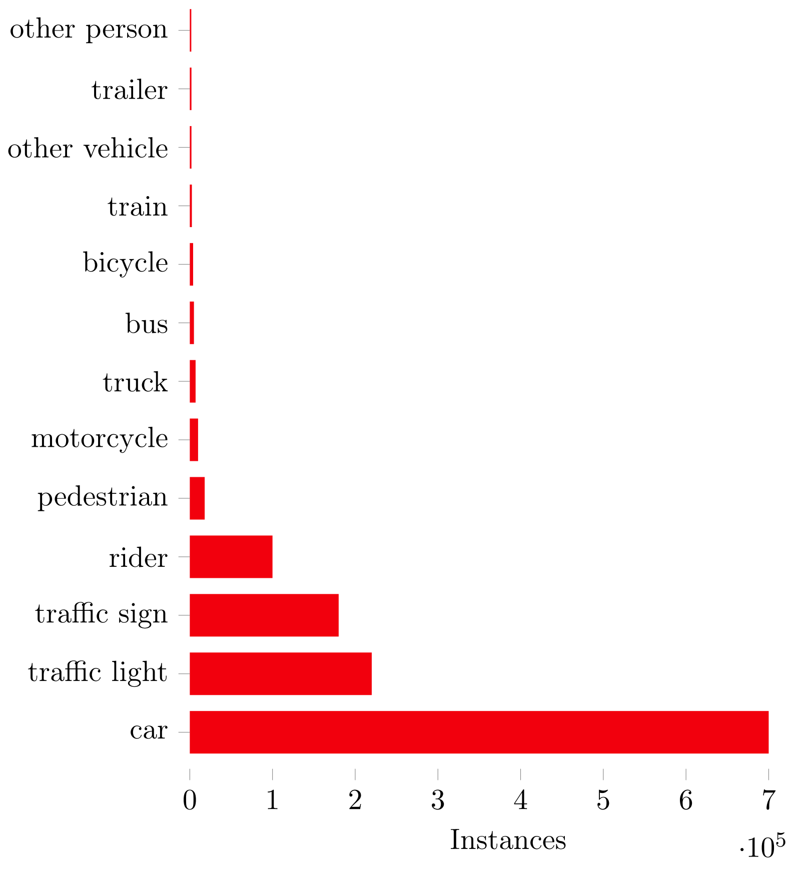


Figure 4.4: Class distribution of the 100k images.

underrepresented while others are over-represented (car). Different approaches were considered when training these models; a set with each reference is shown in Table 4.2. This table should be seen as a reference guide for easier analyse of the plots shown in the rest of this chapter.

[1]https://docs.portainer.io/start/install/server/docker/linux

Table 4.2: Mapping between each Reference number of different YOLOv5 runs.

Name	**Ref**
yolov5m-Low batch size - All GPUS	1.1
yolov5-m - 10K Images - No Regroup	1.2
yolov5-s-regrouped-high-aug	1.3
yolov5-l-high-aug	1.4
yolov5-m Re-grouped/Dropped High-Augmentation	1.5
yolov5-m Dropped-Labels	1.6
yolov5-m Full-100K Images - No Cache	1.7

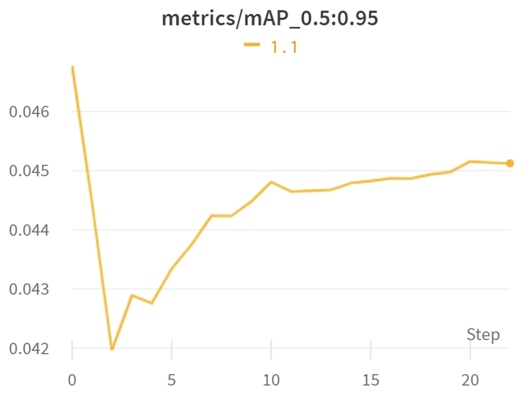


Figure 4.5: Run 1.1 mAP@05-0.95.

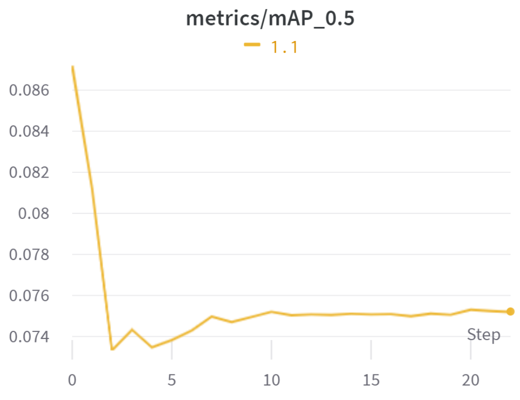


Figure 4.6: Run 1.1 mAP@05.

The first run was trained with 4 GPUs. At first, a batch size of 64 was selected. With these settings, the training kept crashing during the first epoch. After some debugging, it was found that the problem layed in the GPU memory capacity; despite having a combined memory of 27 GB, the GPUs RAM was running out, crashing the training. To fix this problem, the batch size was reduced from 64 to just 16; this allowed for the first results seen in Figure 4.5 and 4.6. This training took 5 hours to run 22 epochs, reaching a mAP@50 of 7.5%. A typical YOLOv5 training takes a minimum of 300 epochs and can reach up to 1000 or more; this means that, to fully train a model with one combination of hyperparameters, it would take at least ten days. Training with these settings makes it impossible to run any meaningful tests in the time frame of the thesis. In addition to this, 4 GPUs were used during this run; this is not always possible since multiple researchers collaborate on DeepLar, so at least one GPU must be left free. To address these issues the batch size was increased to the maximum the GPUs memory allowed. NVIDIA mandates that the batch size is a multiple of the GPUs; therefore, a total batch size of 30 was the highest possible without the training crashing.

The results of this new run are presented in Figures 4.7 and 4.8, taking 45 hours to finish 300 epochs. Even with this long training, the model could not reach 10% mAP@50. A further analysis on the training plots was performed. YOLO has multiple components to its loss function.

- box-loss — bounding box regression loss (Mean Squared Error).

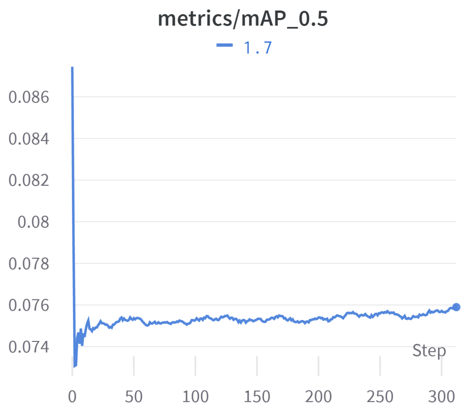


Figure 4.7: Run 1.7 mAP@0.5:0.95.

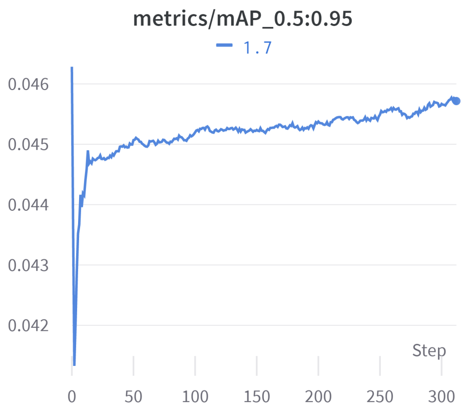


Figure 4.8: Run 1.7 mAP@0.5.

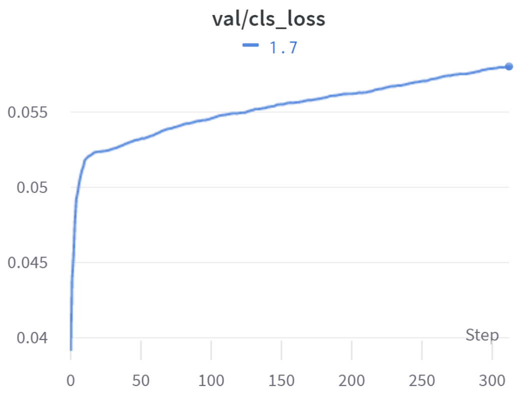


Figure 4.9: Run 1.7 Validation cls-loss.

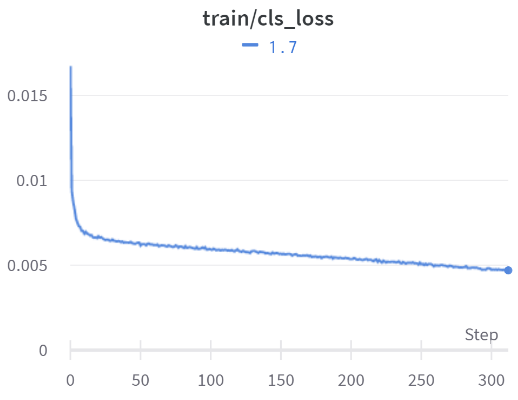


Figure 4.10: Run 1.7 Train cls-loss.

- obj-loss — the confidence of object presence (whether or not there is an object in the bounding box) is the objectness loss (Binary Cross Entropy).
- cls-loss — the classification loss (Cross-Entropy).

The model seems to be overfitting on the cls-loss as seen in Figures 4.9 and 4.10. This is seen by the patterns in the validation and training losses; if the validation loss increases while the training loss decreases it means the model is overfitting. It is important to keep in mind that the scale of these graphs and absolute values are not as relevant as the "change or evolution" of any given variable across the epochs.

The cls-loss value measures the correctness of the classification of each predicted bounding box. This means the model is having trouble in correctly classifying each class. This high validation cls-loss and the low mAP values might be related to the class distribution on the dataset; since some classes are severely underrepresented and the mAP is calculated by the mean of each class precision, the poorly represented classes like train or trailer might be affecting the mAP value too negatively. In addition to this, since the dataset is large, 300 epochs don't seem to be enough, and running experiments for 1000+ epochs is impossible within the time frame of the thesis. This training speed is restricted by the number of GPUs, the batch size and, more importantly, how the data loader loads the dataset. What seems to be happening is that the GPUs have to load the 100 000 images, from the disk, every time an epoch starts, slowing down the training process considerably. A possible solution would be to cache the data on the GPUs RAM, but due to the large dataset size, this was impossible without the training crashing.

BDD100k sub-set

To address thes issues caused by a large dataset, a different approach was taken; the BDD100K was subsampled into a sub-set of 10K images. To allow an easier replication of the results by other researchers, the validation set of the original BDD100K was used. This new sub-set was then divided into a train-val split of 80/20. This new sub-set had the distribution shown in Figure 4.11.

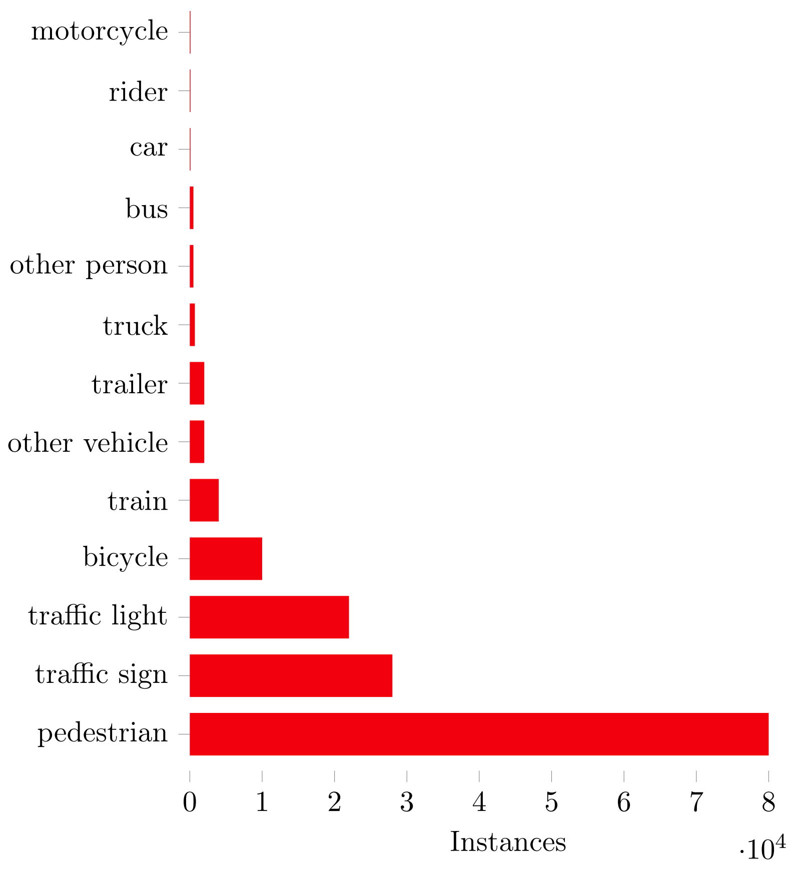


Figure 4.11: Class distribution of the new BDD100K subset made with 10K images (labels-miss-matched).

The maximum number of instances seen in Figure 4.11 is 80 000, compared to the previous 700 000. This reduction creates new problems; some of the classes are even more severely underrepresented, with classes such as motorcycle having less than 400 instances which is not enough (the recommended minimum by the YOLOv5 authors is 10 000 instances per class). Alongside this issue, there was a problem with the label-matching when performing the format conversions and sub-set division that lead to a label swap (for example pedestrians label was swapped with car).

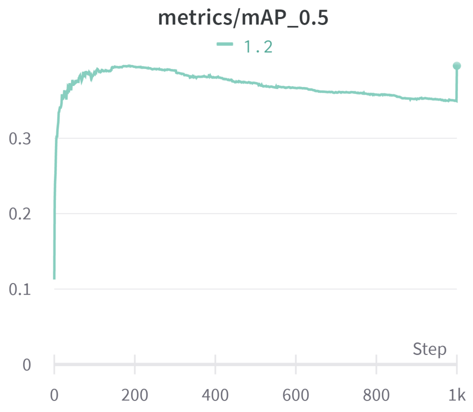


Figure 4.12: Run 1.2 mAP@50.

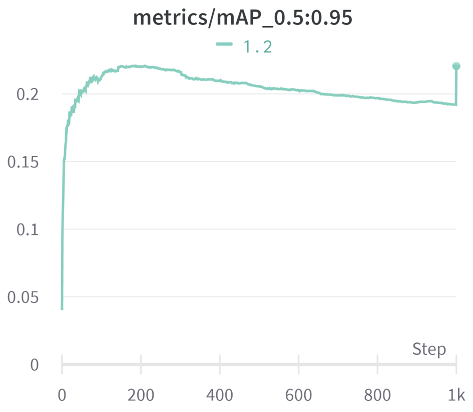


Figure 4.13: Run 1.2 mAP@50:95.

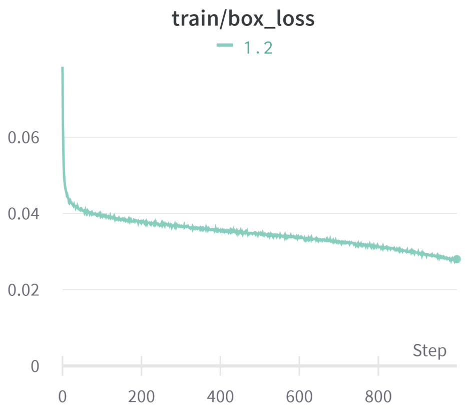


Figure 4.14: Run 1.2 Train box-loss.

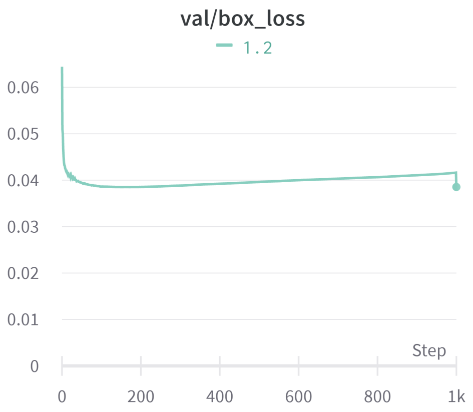


Figure 4.15: Run 1.2 Validation box-loss.

Looking at Figures 4.12 and 4.13 an increase in mAP is noticeable when comparing to the run with 100K images, more specifically an increase of over 30% in mAP@50; in addition to this, the model only took 18 hours to complete the 1000 epochs. However, the model seems to be overfitting on the obj-loss component (Binary Cross Entropy) as seen in Figures 4.18 and 4.19, and overfitting slightly on cls-loss as seen on Figures 4.16 and 4.17. This run was made with the hyperparameters shown in Appendice B; these hyperparameters are set with low augmentation parameters.

A mAP peak is seen on epoch 1000 in Figure 4.12. This peak was first diagnosed as the model leaving a local minimum. With this diagnose in mind, the following runs were trained with more than 1000 epochs in hopes of achieving a better accuracy. A second analysis was performed after all the models in this chapter were trained; it was found that this minimum was due to a PyTorch bug. This bug forced the last epoch weights to be equal to the weights with best mAP of the respective run, this resulted in a point with false measurements.

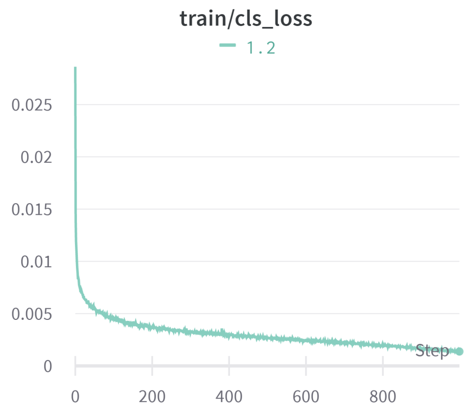


Figure 4.16: Run 1.2 Train cls-loss.

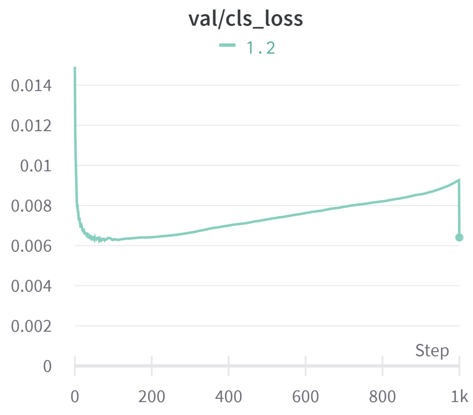


Figure 4.17: Run 1.2 Validation cls-loss.

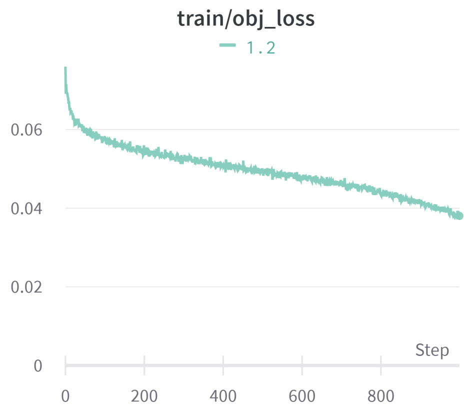


Figure 4.18: Run 1.2 Train obj-loss.

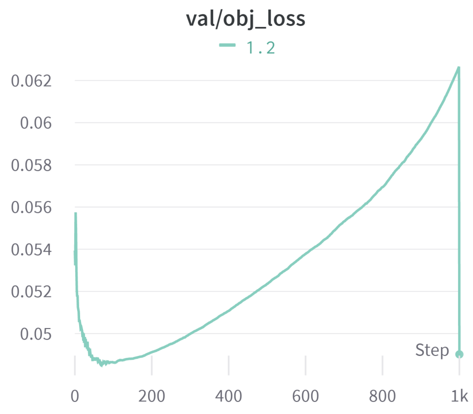


Figure 4.19: Run 1.2 Validation obj-loss.

Class dropping and Regrouping classes

To address some of the issues of the previous run, some adjustments were made. More specifically, some of the classes were dropped. The new class distribution is shown in Figure 4.20.

The new class distribution increased the mAP@50 by almost 20%, as seen in Figure 4.23. However, the model still overfitted in the same areas as the previous run. Some of the classes were still severely underrepresented and with swapped labels, so, a different approach had to be taken. Since some of the classes had a low number of instances, like motorcycle, which only has 400 instances, a regrouping was made between the classes. This regrouping is shown in Table 4.3. The car and pedestrian classes are to be seen as "super classes", a more accurate name would be vehicle or person. The name was kept the same so the same labels could still be used, facilitating the development, validation and analysis of the result.

In an ideal world, with sufficiently large and balanced data, our model would be able to detect all classes efficiently, but since the data set is imperfect and some hardware

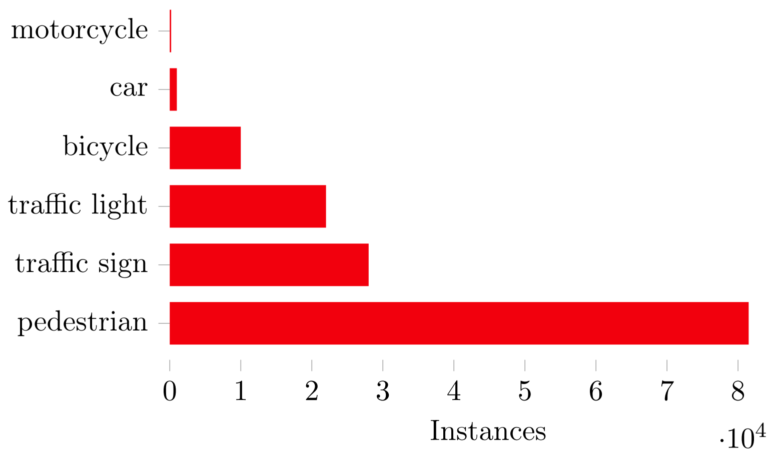


Figure 4.20: Class distribution of the class dropped BDD100K subset (with label miss-match).

Table 4.3: Class Regrouping/Dropped performed.

Groupings	
car	car truck bus motorcycle
pedestrian	pedestrian bicycle rider
dropped	other vehicle train trailer other person

limitations are imposed, a compromise must be made. The labels can either be dropped (the algorithm doesn't train on them) or re-labeled. Due to the low number of instances of classes like "other vehicle, train, trailer, other person", it would be difficult (or even impossible) for the model to generalize well enough to detect them; besides this, the presence of these classes would affect the values of the weights, possibly, converging to a worse solution. On top of this, they would make the value of mAP a miss-representation of our model's accuracy. On the other hand, it is important to keep some of the other labels to avoid missing important detections. So, classes like bus, truck and motorcycle were merged into a vehicle superclass named "car". This way, our model is still able to detect them, mitigating the effect under-represented classes have on the weights and mAP values; the same strategy was applied to pedestrians, bicycles and riders. The final class distribution after the re-labeling / label dropping is seen in Figure 4.21.

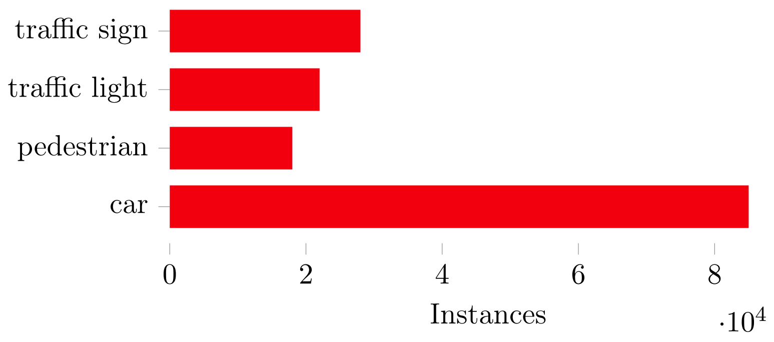


Figure 4.21: Class distribution of the class dropped and Regrouped BDD100K subset.

Dataset Manipulation Summary

Using the entire BDD100K led to poor mAP results and memory issues due to hardware limitations. It also resulted in a longer training that made it impossible to experiment in the time frame of the thesis.

A subset of the larger dataset was then used, totaling 10K images. The model trained much quicker from this sub-set, achieving better mAP values in the process; however, the model still overfitted.

The underrepresented classes were causing problems, making the model converge to a worse solution and misleading the representation of mAP, so a new dataset was made with class dropping and class regrouping.

When performing the label regrouping, the bug causing the label swapping problem was found and fixed. These adjustments to the class distribution allowed for the recommended 10 000 instances per class that YOLOv5 authors recommend. In Figure 4.22 we can see the label distribution across the xy axis of the image; since most of the BDD100K represents an urban driving scenario, often there is a car directly in front of the camera, so we see a larger density of labels in the center of the image as expected.

In addition to the changes in the dataset, more strategies were chosen to address the overfitting issue. The loss function of YOLO is composed of the three components previously described (box, obj and clss) loss; the specific "gain" or contribution of these values to the loss function can be adjusted. As seen in Figure 4.19, the runs with 10k images overfitted on obj-loss, so the "gain" of this value was manually reduced in the hyperparameters of the future runs; in addition to this, more aggressive data augmentation was used.

Figures 4.23 and 4.24 show the difference in mAP of each of these different runs. Deleting and regrouping the labels combined with more aggressive data augmentation and lowering the "gain" of obj-loss (model 1.5) led to an increase of almost 30% mAP@05 compared to the run of the raw 10K images (model 1.2), 7% compared to label-dropping with low augmentation (model 1.6), and a 58% increase since the first run with 100k images. However, as seen in Figures 4.29 and 4.30, the model is still overfitting heavily on obj-loss. The other two values of loss are still overfitting, but not as aggressively; this

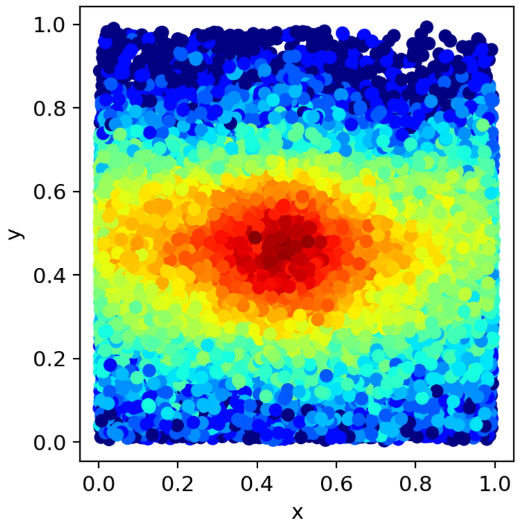


Figure 4.22: XY Label Distribution of the final dataset (with classes regrouped/dropped.

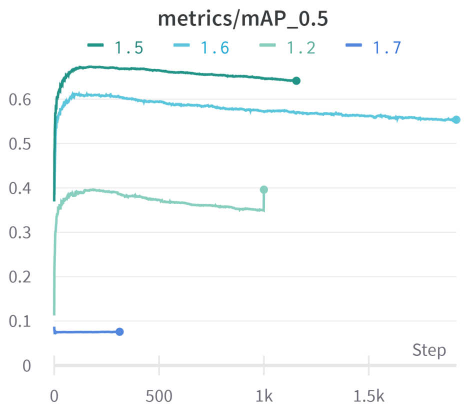


Figure 4.23: mAP@50 of all the YOLOv5-m runs.

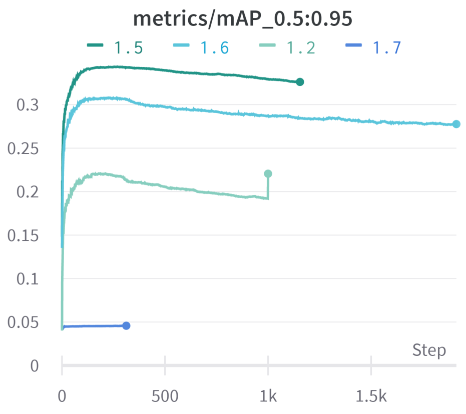


Figure 4.24: mAP@50:95 of all the YOLOv5-m runs.

can be seen in Figures 4.25, 4.26, 4.27 and 4.28.

A question then emerges of what is the best adjustment of hyperparameters and augmentation that allows the model to perform better. Running multiple runs is expensive and takes a lot of time. Despite the success of the manual adjustment made on the run 1.5, these hyperparameters are extremely hard to adjust manually, some industry experts have an intuition on which values to change, but it is a very complex task due to the large number of hyperparameters (in the case of YOLOv5 more than 13). However, these hyperparameters can be adjusted computationally using optimization algorithms. To explore the best hyperparameters for YOLOv5, a genetic algorithm was used. A fitness function (a type of objective function) had to be defined to use this algorithm. A fitness function considering the mAP@50 and mAP@50:95 was defined as seen in Equation 4.1.

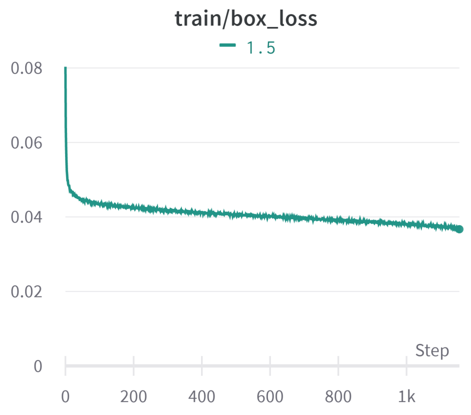


Figure 4.25: Run 1.5 Train box-loss.

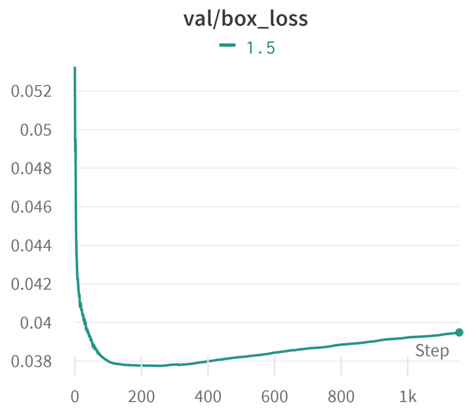


Figure 4.26: Run 1.5 Validation box-loss.

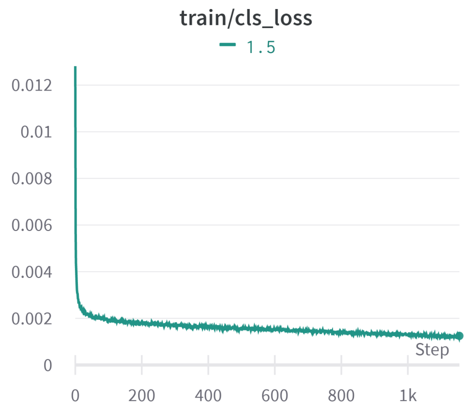


Figure 4.27: Run 1.5 Train cls-loss.

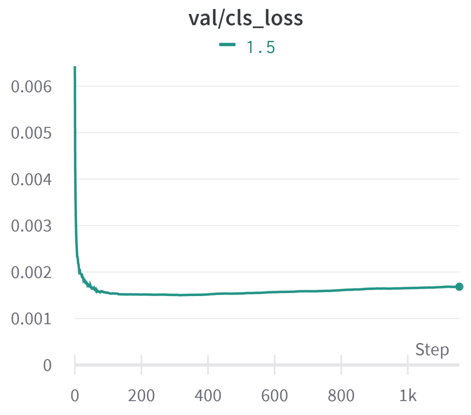


Figure 4.28: Run 1.5 Validation cls-loss.

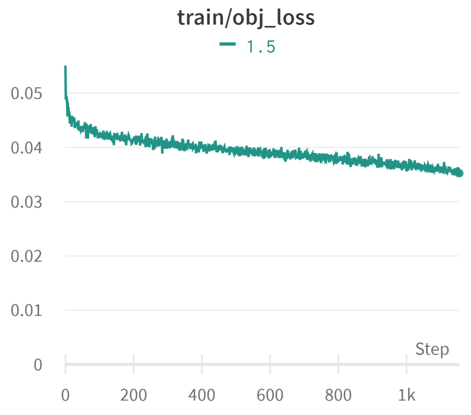


Figure 4.29: Run 1.5 Train obj-loss.

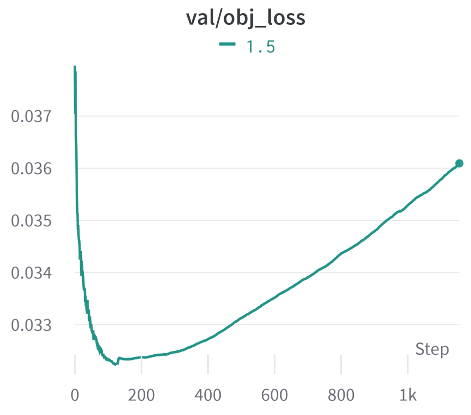


Figure 4.30: Run 1.5 Validation obj-loss.

$$fitness = 0.1\text{mAP}_{05} + 0.9\text{mAP}_{05:095} \tag{4.1}$$

The genetic algorithm runs a "base scenario" multiple times to maximize the fitness function; in this case, the base case selected was a ten epoch run of the re-grouped and re-labeled dataset. Ideally, this hyperparameter evolution should be made with a large base scenario, but since the genetic algorithm has to run the base scenario for at least 300 generations, increasing the number of epochs is too computationally and time expensive, so, a ten epoch base scenario was chosen.

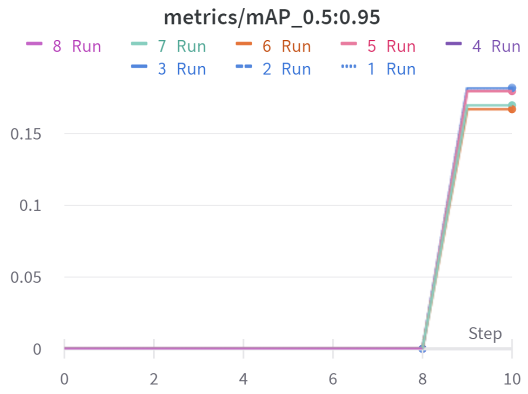


Figure 4.31: Hyperparameter evolution mAP@50:95.

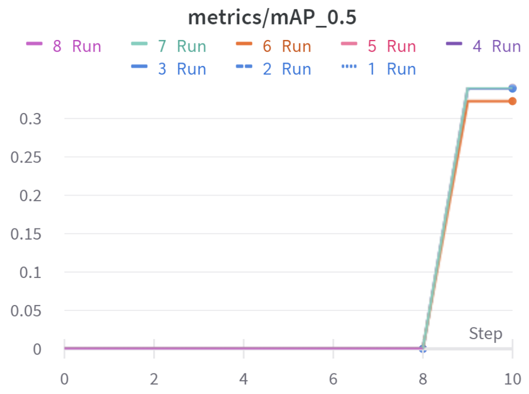


Figure 4.32: Hyperparameter evolution mAP@50.

As seen in the Figures 4.31 and 4.32, different combinations of hyperparameters offer different values of mAP, and it is only expected that these differences will grow wider as the model trains on more data. This evolution ran for three days with a total of 8 runs in which two of them crashed. Since the minimum required for meaningful evolution of hyperparameters is 300 generations [39], finishing the evolution was not possible considering the deadlines of the thesis. There is a chance the evolution took longer because two of the runs crashed; they might have crashed due to memory issues on the GPUs or something different; it is hard to pinpoint. Despite showing promise, the hyperparameter evolution was put on hold while training the other models.

Different Model size comparisons between YOLOv5s

Since all the training and deployment pipelines were already set, some more experiments were run with the same hyperparameters on different model sizes. A comparison between the small, medium and large models was then performed. The metrics for these comparisons can be seen in Figures 4.33, 4.34, 4.35 and 4.36.

As seen in Figure 4.35 an increase of almost 2% in mAP was obtained by the larger model compared to the medium one; however, the biggest difference is in the small to medium models with a difference close to 4% in mAP@50. Table 4.4 shows a summary of all the runs made on YOLOv5.

In addition to this, an analysis of the confusion matrix of one of the models (in this case, YOLOv5-Small) was performed. This matrix can be seen in Figure 4.37.

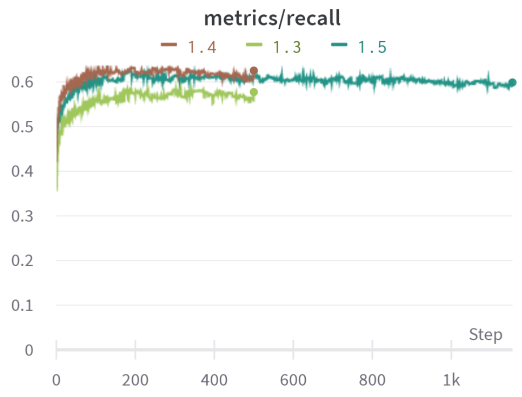


Figure 4.33: Recall of YOLOv5 small-medium-large runs.

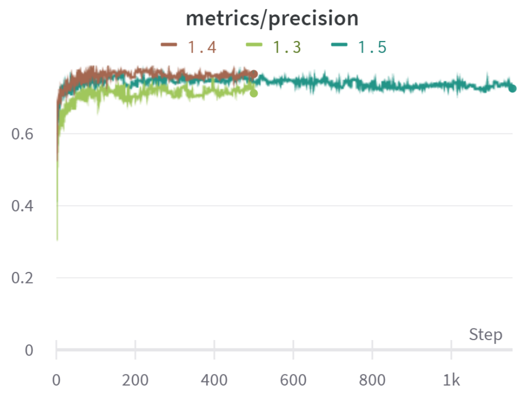


Figure 4.34: Precision of YOLOv5 small-medium-large runs.

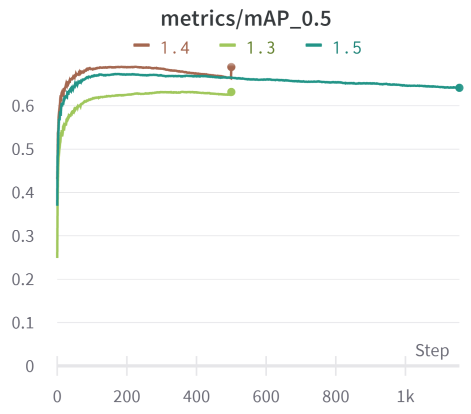


Figure 4.35: mAP@50 of YOLOv5 small-medium-large runs.

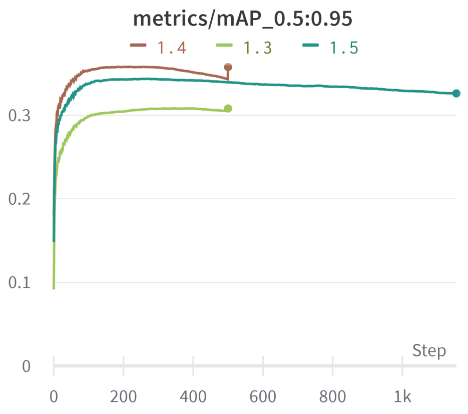


Figure 4.36: mAP@50:95 of YOLOv5 small-medium-large runs.

As expected, the classifications of pedestrians were considerably worse than the car ones; this is most likely due to the class distribution. On the other hand, the number of False Positives on cars is considerably high; this might be due to the extremely difficult scenarios found on the BDD100K dataset. Dealing with these extreme scenarios is probably problematic with only 10 000 images, for example, when cars are partially occluded or at night time, as shown in Figure 4.38.

Table 4.4: Summary of all the YOLOV5 run.

Strategy	**Runtime (hh:mm)**	**epochs**	**model weights**	**best epoch**	**best mAP @0.5**	**best mAP @0.5: 0.95**	**best precision**	**best recall**
▶ Low batch size - All GPUS	04:50	100	yolov5m.pt	0	8.7%	4.7%	0.387	0.152
▶ 10K Images - No Regroup	18:32	1000	yolov5m.pt	187	39.6%	22.1%	0.787	0.347
▶ s-Regrouped High-Aug	05:21	500	yolov5s.pt	378	63.2%	30.8%	0.712	0.578
▶ l-Regrouped High-Aug	14:04	500	yolov5l.pt	185	69.0%	35.8%	0.763	0.628
▶ m-Regrouped - High-Aug	21:16	1500	yolov5m.pt	269	67.1%	34.4%	0.749	0.611
▶ Dropped - Labels	35:00	3000	yolov5m.pt	227	60.8%	30.8%	0.726	0.555
▶ 100K Images - No Cache	45:00	500	yolov5m.pt	0	8.7%	4.6%	0.309	0.163

YOLOv5 Training summary

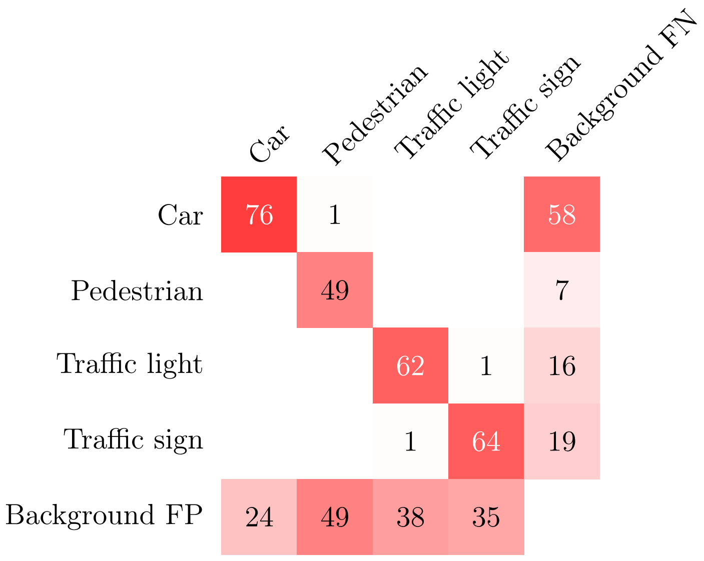
The Dataset pre-manipulation improved the mAP values but the model still overfitted.

The hyperparameters were then adjusted manually to provide more aggressive data augmentation. In addition to this, since the model was overfitting on objectness loss, the gain of this value on the loss function was changed. This adjustment, delayed overfitting leading to better mAP values.

Further manipulation and tuning of these hyperparameters (and augmentation parameters) needed to be done, so a genetic algorithm was used to provide hyperparameter evolution. However, this proved to take too long, making its use impossible with the current hardware setup and time constraints.

YOLOR

After the runs with YOLOv5, two different architectures were trained, YOLOR-CSP and YOLOR-CSP-X. These are the smallest architectures of YOLOR algorithm, YOLOR-CSP-X being the largest of the two. Using the same processed dataset selected in YOLOv5 and with similar hyperparameters. YOLOR was constructed on top of the YOLOv5 PyTorch version of Darknet code. This makes most of the code infrastructure



	Car	Pedestrian	Traffic light	Traffic sign	Background FN
Car	76	1			58
Pedestrian		49			7
Traffic light			62	1	16
Traffic sign			1	64	19
Background FP	24	49	38	35	

Figure 4.37: Confusion Matrix of the trained YOLOv5-Small (model run #1.3). The values of the matrix are the rounded percentages, Background FP is equivalent to "detection only" and Background FN is equivalent to "ground truth only"

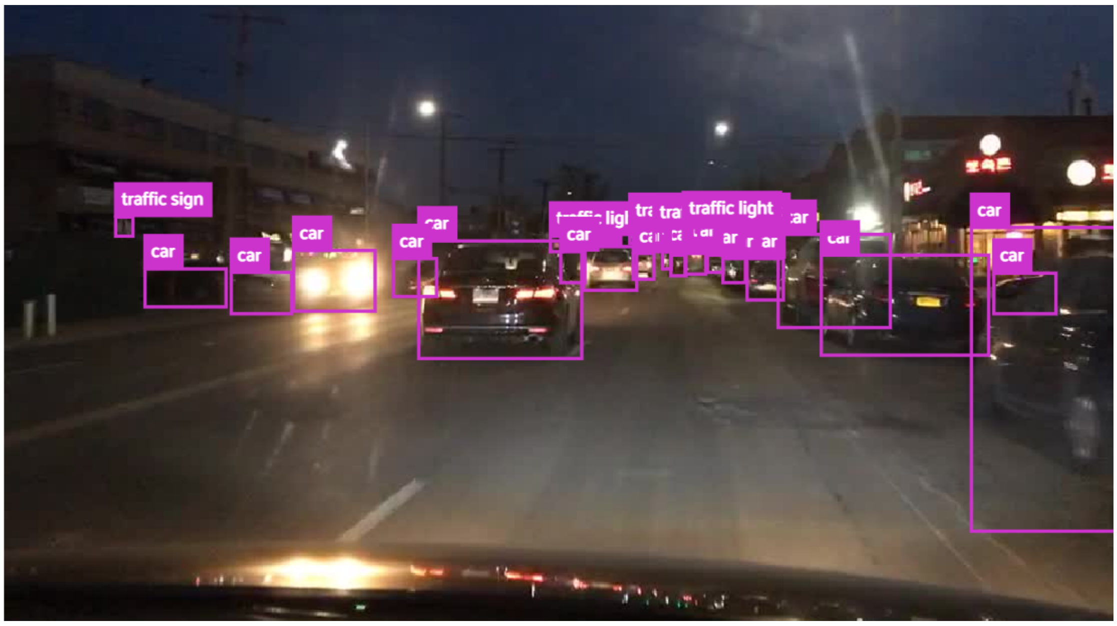


Figure 4.38: Example of a BDD100K Ground Truth Image.

similar and, as a result, the hyperparameter files are quite similar to configure.

As seen in Figures 4.40 and 4.39, using these similar hyperparameters/augmentations, YOLOR performs better than YOLOv5, achieving higher mAP values. These models also overfit less, as seen in Figures 4.43, 4.44, 4.45, 4.46 and 4.47 and achieving a slightly higher mAP. YOLOR-CSP-X shows a slight improvement in mAP values over the YOLOR-CSP version, as we can see in Figure 4.39. In addition to this, the overfitting is delayed in

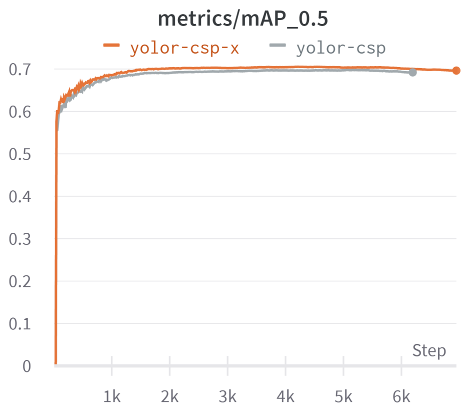


Figure 4.39: mAP@50 of YOLOR-CSP vs YOLOR-CSP-X.

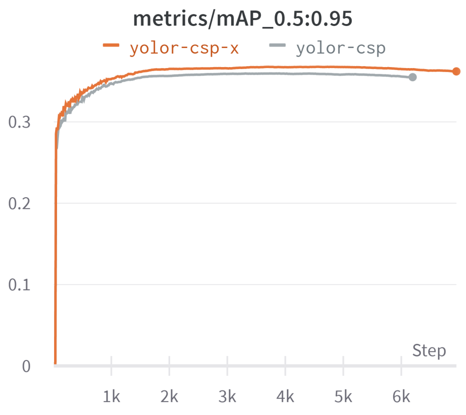


Figure 4.40: mAP@50:95 of YOLOR-CSP vs YOLOR-CSP-X.

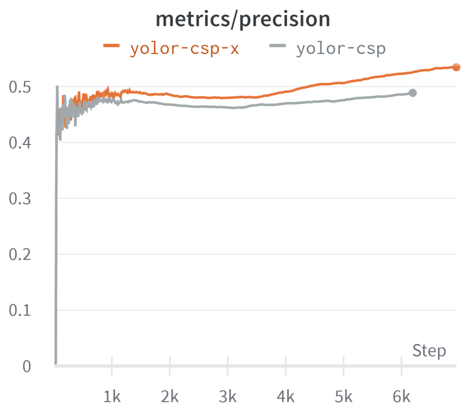


Figure 4.41: Precision plot of YOLOR-CSP vs YOLOR-CSP-X.

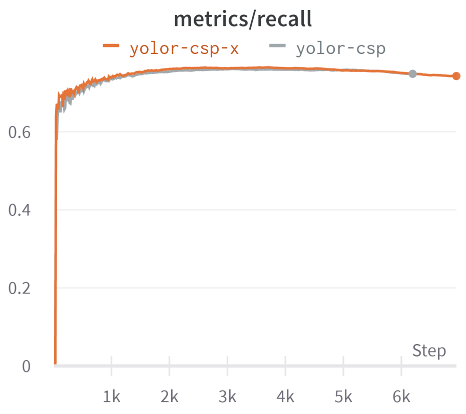


Figure 4.42: Recall plot of YOLOR-CSP vs YOLOR-CSP-X.

YOLOR-CSP-X compared to YOLOR-CSP; this change might be due to the larger model YOLOR-CSP-X architecture offers.

Scaled-YOLOv4

Similar runs with similar hyperparameters were performed on Scaled-YOLOv4 using PyTorch, and the final results of all the best models from all the different algorithms are shown in Table 4.5. YOLOR provides the best mAP results despite the lower resolution of the detector, YOLOv5-m and Scaled-YOLOv4 have similar performances. The inference speed of these models will be tested in later sections.

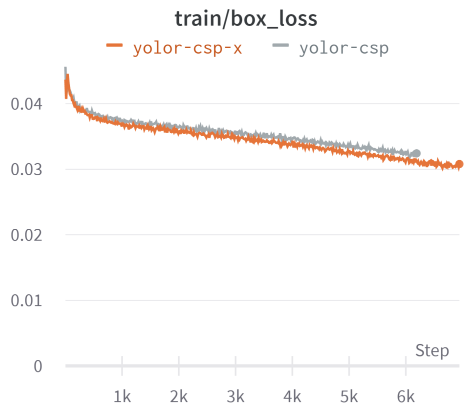


Figure 4.43: Train box-loss of YOLOR-CSP vs YOLOR-CSP-X.

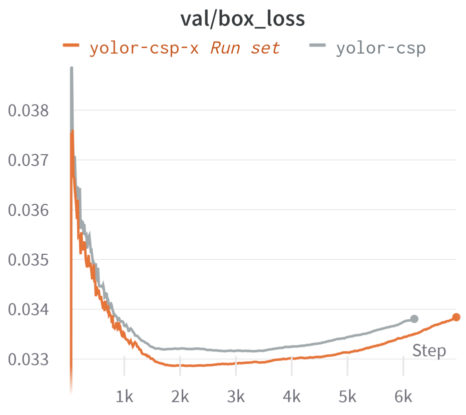


Figure 4.44: Validation box-loss of YOLOR-CSP vs YOLOR-CSP-X.

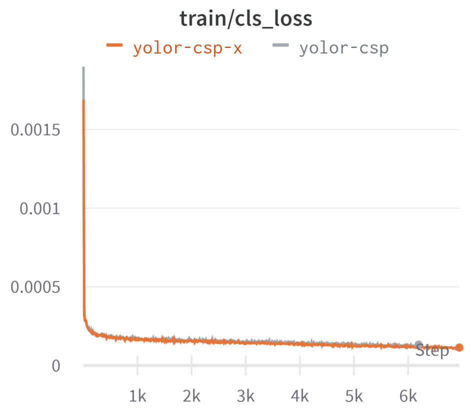


Figure 4.45: Train cls-loss of YOLOR-CSP vs YOLOR-CSP-X.

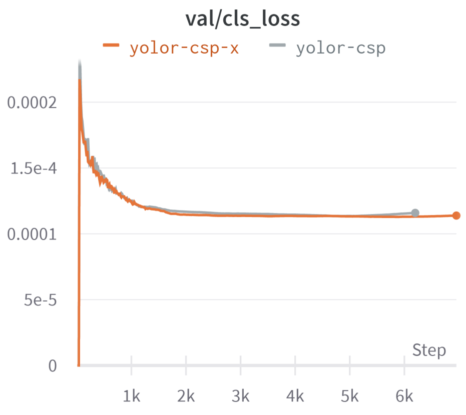


Figure 4.46: Validation cls-loss of YOLOR-CSP vs YOLOR-CSP-X.

Table 4.5: Summary of All Best Model Runs of each Architecture.

Model	**Detector Resolution**	**mAP@50**	**mAP@95**
Scaled-YOLOv4	512×512	67.2 %	33.80%
YOLOv5-S	640×640	63.19%	30.81%
YOLOv5-M	640×640	67.12%	34.37%
YOLOv5-L	640×640	68.99%	35.81%
YOLOR-CSP	512×512	69.70%	35.85%
YOLOR-CSP-X	512×512	70.50%	36.60%

Some further tests were intended with YOLOX in mind, but many dependencies problems arose because of the different system architecture of NVIDIA Jetson AGX Xavier. Because of this different architecture, even though the model could be trained at

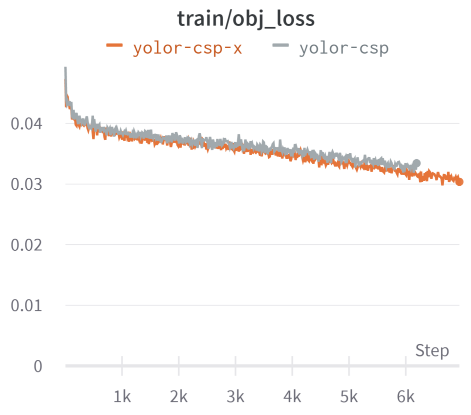


Figure 4.47: Train obj-loss of YOLOR-CSP vs YOLOR-CSP-X.

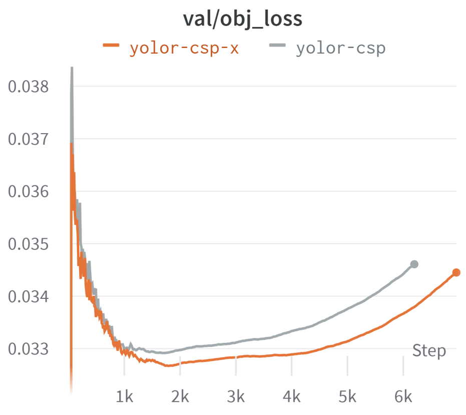


Figure 4.48: Validaiton obj-loss of YOLOR-CSP vs YOLOR-CSP-X.

DeepLar, the deployment with TensorRT needed for real-time inference was not possible; this, was due to library incompatibilities when installing YOLOX on the Jetson. Not being able to deploy with TensorRT, all further tests involving YOLOX were dropped.

Overall summary

Scaled YOLOv4 offered comparable results to YOLOv5 with the medium-sized model, and despite the lower resolution, YOLOR achieved a better mAP. The larger models performed better, with YOLOR-CSP-X achieving a mAP of over 70%.

4.3 Deployment Results

This section presents the deployment results. Starting from the creation of the optimized TensorRT engine to the implementation of the Trackers on the DeepStream pipeline and its inference results. These results are then followed by an evaluation and a real-time analysis on Aveiro's roads.

4.3.1 Creation of the TensorRT Engine

With the models trained, the next step was to deploy them; a conversion to a TensorRT engine is needed since it is a pre-requisite for the DeepStream platform. To perform this conversion, first, the weight files need to be converted from the PyTorch format to the Darknet weights format (from .pt to .weights). In the case of YOLOv5 and YOLOR, a conversion was already provided by either the research community or the authors of the algorithms; however, in the case of Scaled-YoloV4, this conversion was not provided, so a custom one had to be made. A few days were spent developing this conversion, but without in-depth knowledge of both PyTorch and Darknet frameworks, creating this conversion from scratch is challenging, especially in the time frame provided, so this idea was abandoned. Since YOLOv5 offered similar results to Scaled-YoloV4, there was no great benefit to keep investing time in it. One possible alternative would be to re-train Scaled-YoloV4 on the Darknet framework in C instead of the PyTorch framework; this way, the models could be converted to TensorRT and deployed on DeepStream since the Darknet formats are already compatible with TensorRT deployment.

4.3.2 Tracker Inference

Modern SOTA trackers perform tracking by detection. This means they leverage the powerful existing detectors to do object tracking. The previous section discussed all subjects related the training of the object detectors, followed up by the creation of the engine required to deploy them on the Jetson AGX Xavier. After the engines were created and deployed on DeepStream, the trackers needed to be implemented. Due to the modular nature of DeepStream, once the tracker is implemented, changing between model engines is fairly easy. As discussed previously, the DeepStream SDK comes with pre-integrated compatible trackers. With the tracking algorithms introduced and the DeepStream SDK explained, an overview of the SOTA trackers and their relationship with DeepStream is presented for better readability:

Object tracking with DeepStream overview

The NVIDIA DeepStream SDK offers reliable solutions for object tracking with an object tracking plugin and pre-compatible trackers (IoU, NvDCF and DeepSORT). DeepSORT and NvDCF seem to have similar performance metrics.

ByteTrack is an improvement on all these, achieving better FPS and a higher MOTA. However, it is not included in the DeepStream SDK due to being too recent.

StrongSORT and OC-SORT both improve on the SORT/DeepSORT, increasing their metrics. However, the papers are too recent, so no public code implementation is available by the authors. As a result there is no TensorRT compatibility or DeepStream integration.

The three trackers compatible with DeepStream SDK were coded and deployed (IoU Tracker, NvDCF and DeepSORT). NvDCF allows for configurations where trade-offs between accuracy and performance can be made, for example, different tracker resolutions or a different maxShadowTrackingAge. The configuration files used for this tracker are shown in Appendix B. These configurations are accuracy-based (they try to maximize the tracker's accuracy). DeepSORT allows custom Re-ID models; these are models that can be trained on large datasets to allow for a fine-tuned Re-Identification. The Re-ID model used was the one provided by the original DeepSORT authors since it was trained on a large dataset that included urban environments. When all of these trackers were successfully deployed, in hopes of bringing the solution even closer to the state of the art, a BYTE algorithm implementation was attempted on DeepStream. The previous three trackers were made compatible by NVIDIA on the DeepStream SDK; however, the use of external "more recent" trackers required custom implementation. ByteTracker, StrongSORT and OC-SORT currently offer SOTA results; however, these last two are too recent at the time of writing this thesis, so there is either no public code available or no documentation on how to run them. In addition to this, TensorRT deployment is pre-required to work with DeepStream, and none of the StrongSORT or OC-SORT authors provided a deployment pipeline for real-time inference. On the other hand, ByteTracker had a TensorRT deployment pipeline available in C++, so this was used to develop a BYTE method DeepStream implementation. There were some previous efforts to bring this Tracker to the DeepStream platform by another researcher. After some contact, this researcher made his solution to DeepStream 5.0 publically available, so minimal code adjustment was needed to port it to DeepStream 6.1 (The version chosen for this dissertation due to its improved tracker library). The BYTE engine and deployment code made in collaboration with this researcher can be found in [64].

With the deployment code finished, all that is left is to run inferences on the models. The deployment was made for every model with each different tracker. Multiple tactics were used to study different ways to boost the FPS of the tracking. A change in the precision of the floating points (testing FP32 and FP16, trading off precision for speed) and a difference in the detection interval. The detection interval dictates the interval between batches where the detector forms a detection; for example, with an interval of one, the detector would be on every other batch [ON-OFF], and with an interval of two, the detector would be [ON-OFF-OFF]. Since the NvDCF utilizes visual features to identify targets, it still works fairly well when the detection interval is different from zero. The same happens with DeepSORT (due to the Re-ID model) and BYTE due to its data association methods; the IOU tracker, on the other hand, performs poorly with an interval different of zero. By testing these different intervals, we can achieve an optimal compromise of inference speed without sacrificing much accuracy, allowing the use of more accurate detectors with bigger-sized models with greater inference speeds.

Table 4.6: Number of FPS (Frames per Second) of the DeepSORT tracker deployment with DeepStream with different Detector Intervals.

Name	Precision	Int=0	Int=1	Int=2	Int=3	Int=4	Int=5
yolor-csp-x	FP32	10.4	20.41	30.94	37.94	46.76	55.35
yolor-csp		17.06	33.3	50.65	59.58	73.52	85.08
yolov5-l		11.29	22.42	34.21	41.6	51.7	60.46
yolov5-m		20.53	39.08	59.12	66.18	84.81	99.42
yolov5-s		42.08	74.13	99.23	111.47	141.17	159.69
yolor-csp-x	FP16	30.29	61.06	90.95	67.52	82.48	96.97
yolor-csp		36.46	65.76	85.56	100.38	126.76	146.03
yolov5-l		29.53	54.97	87.4	94.07	101.95	124.31
yolov5-m		44.51	77.99	107.01	115.45	143.81	164.82
yolov5-s		68.05	90.5	165.72	160.51	193.23	211.42

All the experiments were performed with one video input stream, more specifically, a sample video of an urban driving setting that is publicly available on the DeepStream SDK; choosing a video as an input stream for the experiments offers a better metric since the hardware of the cameras does not limit it, and since the video is publicly available any researcher can reproduce the results and use them as a base for future work. Tables 4.6, 4.7, 4.8 and 4.9, show an analysis of the FPS with different detection intervals and

Table 4.7: Number of FPS (Frames per Second) of the NvDCF tracker deployment with DeepStream with different Detector Intervals.

Name	Precision	Int=0	Int=1	Int=2	Int=3	Int=4	Int=5
yolor-csp-x	FP32	10.45	19.28	26.64	33.44	39.76	46.18
yolor-csp		17.47	30.43	40.29	49.47	57.89	64.62
yolov5-l		11.76	21.38	29.39	36.76	43.37	48.74
yolov5-m		20.77	35.06	46.95	56.11	65.71	73.25
yolov5-s		43.13	64.26	76.87	87.79	97.05	103.75
yolor-csp-x	FP16	24.01	39.86	51.71	61.5	71.64	78.65
yolor-csp		38.53	58.08	70.76	80.6	89.8	97.18
yolov5-l		31.29	50.29	62.36	71.8	81.46	87.93
yolov5-m		45.34	66.5	78.98	89.7	99.68	107.34
yolov5-s		73.21	96.7	104.43	116.7	124.06	129.87

Table 4.8: Number of FPS (Frames per Second) of the BYTE method deployment with DeepStream with different Detector Intervals.

Name	Precision	Int=0	Int=1	Int=2	Int=3	Int=4	Int=5
yolor-csp-x	FP32	11.5	23.13	34.94	41.04	50.31	59.2
yolor-csp		20.62	41.57	62.28	67.3	82.15	91.93
yolov5-l		13.03	26.1	39.41	45.86	56.23	66.04
yolov5-m		25.02	50.52	75.43	79.47	93.23	109.85
yolov5-s		64.3	129.69	192.97	148.01	174.49	200.26
yolor-csp-x	FP16	30.05	60.69	91.71	88.18	105.72	125.46
yolor-csp		55.31	112.44	168.65	137.29	160.54	184.66
yolov5-l		41.92	84.93	127.29	113.99	133.24	157.31
yolov5-m		68.19	137.5	204.98	155.17	180.18	205.36
yolov5-s		117.53	230.81	307.5	233.65	259.64	290.96

precisions. The detection FPS is similar across all trackers for when the interval equals zero. A boost in speed is shown when lowering the precision, as expected, however, many detections are missed, more particularly when detecting occluded objects in the distance. In 2020, Tesla, had their object detector running at around 17-18 FPS, so, in hopes of bridging the gap between this dissertation and the industry, 17 FPS was set as the bar for minimum FPS needed for real-time inference in ATLASCAR2. With an interval of zero, YOLOR-CSP-X model reached around 10 FPS; the same goes for YOLOv5-l; this is possibly due to their larger model architecture. The two highest mAP models that could infer at least 17 FPS were YOLOR-CSP and YOLOv5-m.

After a visual analysis, it seems that an interval of five was the highest the trackers could go without losing important detections. However, an interval of two offered the best compromise between model accuracy and inference speed. When analyzing an interval of two, as shown in Table 4.6 DeepSORT offers a higher frame rate than NvDCF as shown in Table 4.7. The BYTE method offers an even higher FPS, as seen

Table 4.9: Number of FPS (Frames per Second) of the IoU tracker deployment with DeepStream with different Detector Intervals.

Name	Precision	Int=0	Int=1
yolor-csp-x	FP32	11.73	23.39
yolor-csp		21.01	41.87
yolov5-l		13.4	26.63
yolov5-m		25.37	50.52
yolov5-s		65.94	129.86
yolor-csp-x	FP16	30.38	60.82
yolor-csp		55.87	113.54
yolov5-l		42.39	84.93
yolov5-m		68.69	138.79
yolov5-s		158.71	276.68

in Table 4.8, however, a visual analysis of the inference noted that many detections were missing compared to the previous two. This is the opposite of what is expected since BYTE offered better performance metrics in the literature compared to DeepSORT. The problem seems to be related to the BYTE implementation. ByteTracker is a composition of the detector YOLOX and the method BYTE. In our inference scenario, we have an even higher performance detector YOLOR; it would be expected that this combination would lead to an improvement over the current SOTA. However, the implementation of the BYTE method was made by a third party and not the original authors; since all the processes in DeepStream have to be extremely well optimized, the current implementation seems to have some flaws performing significantly worse than DeepSORT or NvDCF. Unfortunately, without in-depth knowledge of both C++ optimization, the DeepStream platform, and the BYTE method, a better solution could not be executed in time. This leaves DeepSORT and NvDCF as the two working solutions for real-time inference. NVIDIA lacked values for evaluation metrics between the trackers; however, after some communication with the chief engineers of the tracking department, NVIDIA communicated that both DeepSORT and NvDCF have similar MOTA values with NvDCF performing slightly better; these were, however, internal numbers of the NVIDIA team so they could not be discussed in public. This leaves the problem of evaluating the trackers with quantitative metrics; in the literature, we have a rough estimate of DeepSORT MOTA values; however there is currently no way of evaluating a Tracker within the DeepStream platform other than a visual appreciation.

4.3.3 Tracker Evaluation

To address the lack of support for Tracker evaluation on the DeepStream SDK a custom implementation was created by extracting the Tracker metadata from DeepStream trough evaluation scripts. Most recent trackers are evaluated on multiple datasets on the HOTA and MOTA metric, for example, on the MOTChallenge dataset or the KITTI dataset. The evaluation code [65] for these challenges is public. To use these evaluation platforms, first the metadata was extracted; DeepStream SDK has a built in function that outputs the data in a similar format to KITTI.

One alternative would be to try to convert this metadata into another existing format from the official evaluation [65] supported formats and run it on a custom Aveiro dataset. However, currently, a labeled Aveiro dataset does not exist, and creating one would require a considerable amount of hours in labeling (hundreds or more). Another alternative would be to use an existing dataset that reflects an urban driving scenario. KITTI offers a tracking dataset in an urban driving scenario so, it can be used to evaluate the trackers discussed in the previous subsection. With this in mind, scripts were created to convert the output files from the DeepStream output format to the evaluation format for the KITTI dataset; however, a problem still needed to be solved before the tracking data could be evaluated. DeepStream only works with video streams, not image frames. The public tracking datasets that exist are available with the image frames with an associated label text file. To solve this issue, first, the image frames needed to be converted to video; this video could be used as an input stream for the DeepStream platform, allowing for the metadata extraction that could be then formatted for the evaluation platform [65]. The FPS in which the data was recorded is necessary to create a video from the frames, however, the KITTI dataset does not have this information publicly available. After reaching out to the creator of the official tracker evaluation algorithm for the KITTI dataset, it was communicated that the KITTI dataset was shot at 10 FPS per second. With this information the video could be created. After the video was created and processed in the DeepStream pipeline, the metadata was formatted with custom scripts to the new KITTI evaluation format. This pipeline is shown in Figure 4.49.

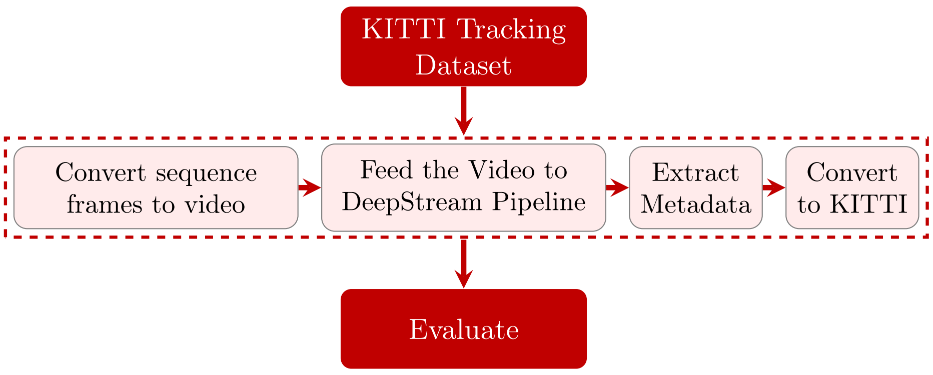

KITTI Tracking Dataset
Convert sequence frames to video
Feed the Video to DeepStream Pipeline
Extract Metadata
Convert to KITTI
Evaluate

Figure 4.49: Tracker evaluation pipeline.

KITTI tracker dataset is composed of twenty sequences, most of them on pedestrian dense environments. Since our model over-fitted strongly on pedestrian detection, it would be more fair to compare the different trackers on car dense scenarios, so the sequence "0007" of the KITTI dataset was chosen to create the video; this sequence is made of 800 frames on a car dense environment. The results of the evaluation on this sequence is shown in Table 4.10. This evaluation was performed with the YOLOR-CSP detector since it offered the best compromise between speed and accuracy.

From the results shown in Table 4.10, we can see that DeepSORT and NvDCF have similar performance metrics. When the detection interval is zero, DeepSORT offers a slightly higher MOTA and IDF1 scores, however, this is probably due to the number of detections by the model being different from the ground truth labels (in theory, with a perfect detector/tracker they should be the same). On the KITTI tracking dataset

Table 4.10: Tracker + YOLOR-CSP evaluation results on the "0007" sequence of the KITTI Tracking dataset. Gt means Ground Truth, Dets means detections.

Tracker	Int	HOTA	MOTA	MOTP	IDF1	Dets	Gt-Dets	IDs	GT-Ids
DeepSORT	Int=0	51.81	47.38	69.11	72.53	2067	1967	87	53
NvDCF		51.38	43.37	72.85	69.25	2157	1967	112	53
BYTE		48.07	37.21	77.38	61.29	2611	1967	164	53
DeepSORT	Int=1	39.33	22.36	64.86	56.45	1739	1967	91	53
NvDCF		43.81	31.88	69.30	61.68	1921	1967	121	53
BYTE		3.66	0.20	94.78	0.50	6	1967	2	53
DeepSORT	Int=2	32.28	6.35	64.51	43.41	1548	1967	97	53
NvDCF		35.98	15.20	67.51	48.39	1951	1967	168	53
BYTE		2.17	0.10	94.27	0.30	4	1967	2	53

multiple objects are labeled with the label "Don't Care", these are objects that are located very far away from the camera and are difficult to label. The authors of KITTI decided to not evaluate on these extreme scenarios, however since our model was trained on a more challenging dataset (BDD100K), it is able to detect these most extreme scenarios. Despite the fact that the model correctly detected these objects, the ground truth labeling of KITTI does not contain these challenging objects, which means that, "in the eyes" of the tracker evaluation algorithm, the DeepStream Tracker is outputting False Positives. These False Positives affect the metric results creating scenarios where better Trackers are penalized for their higher accuracy and number of detections. When changing inference intervals, the values of HOTA and MOTA drop drastically while MOTP remains very close. These type of metrics heavily penalize missing or extra detections. When the inference interval is different from zero, it is possible that detections are missed in frames where an object partially enters the scene; these missed detections heavily weigh on the HOTA and MOTA values. Although these metrics are important and provide meaningful results, tracking evaluating metrics keep evolving and are not perfect. While the metrics for HOTA and MOTA dropped when changing the inference interval, a visual (subjective) analysis show that this difference is negligible being almost impossible to distinguish between a detection interval of zero and two. Since tracking is measured on a time sequence, a video illustrates its operation better, so as an aid to the reader, videos were provided on Youtube. To further demonstrate the fragility of these metrics, the DeepSORT tracker was tested with different object detection models. A comparision was made between one of the highest performing models (YOLOR-CSP) and one of the lowest accuracy models (YOLOv5-Small). This comparison is shown in Table 4.11. As shown, despite having a better detector, the tracker with YOLOR-CSP obtained lower HOTA and MOTP scores.

Table 4.10 shows that, with an increase of the detection interval, NvDCF performs better than DeepSORT. This is possibly due to the visual features NvDCF uses when Re-Identifying the objects. This is important to know when scaling the solution to multiple cameras. With a single camera and an interval of zero, DeepSORT provides enough FPS and a more accurate solution, however, when scaling for multiple cameras, different detection intervals provide differences in inference speed, the redundant detection of

Table 4.11: Comparison of the DeepSORT tracker performance with different models.

Model	Int	HOTA	MOTA	MOTP	IDF1	Dets	Gt-Dets	IDs	GT Ids
YOLOv5-Small	0	53.317	44.992	70.877	71.44	1888	1967	80	53
YOLOR-CSP	0	51.81	47.38	69.11	72.53	2067	1967	87	53

partially overlapped cameras address the small accuracy loss created by this interval. As expected from a qualitative analysis performed previously during the inference speed tests, the BYTE implementation is flawed. While it offers the highest MOTP of all three trackers its MOTA, HOTA values and IDF1 scores are lower. The IDF1 score reflects the identity switches the tracker performs. We can see that, the total tracking IDs of BYTE are triple of the ground truth when the interval is zero, this means BYTE possibly is having trouble re-associating the IDS of objects that were occluded. In addiction to this, this implementation of BYTE can not be used with different inference intervals since a lot of detections are missed. Despite BYTE outperforming DeepSORT in the literature, since this implementation of BYTE in DeepStream was not made by the original authors, it is currently not as optimized, leading to worse results.

4.3.4 Camera Inference in Aveiro

All of the tests run so far were on video (mp4) streams. This brings the advantage of making the work reproducible by other researchers providing consistent results; in addition to this, using a video stream, instead of camera live stream, means there is no speed bottleneck originating from the cameras selected. However, real-time camera inference is also necessary to deploy it in an autonomous driving scenario. The cameras provided for this thesis were the e-cam130_CUXVR. This camera comes with many advantages, having four cameras with 4k resolution, being fabricated and specifically optimized for the NVIDIA Jetson AGX Xavier. This allows for better perception all around the car in contrast to a solution with one frontal camera. With the aim of exploring multiple camera solutions and since the hardware was already provided, the camera set-up was attempted on the NVIDIA Jetson AGX Xavier. These cameras offer the installation package in the form of an image that can be flashed. This approach had some problems; the system was being flashed with an old operating system with an older version of JetPack. This meant that to install the cameras the way the manufacturer recommended, the version of JetPack would be forced to an older one; with this older version, the most recent versions of DeepStream would not be available, therefore none of the SOTA detectors and trackers could be used. To avoid this problem, multiple attempts to install the camera drivers from source were performed; with this approach, more problems followed; dependencies conflicts that took around two weeks to solve. Each different attempt to install the camera drivers normally required the Jetson to be flashed from scratch, which took around 2-3 hours considering the time to install all the dependencies necessary to test the code.

With the dependencies problems solved, the manufacturer software, made specifically to test the cameras, was run; this software was run with the most recent version of JetPack successfully. However, even though a normal USB camera could be used in this software, the 4k cameras still did not work. At first, the problem looked to be originated from the camera hardware and not the software, so an attempt to flash the Jetson with an older version of JetPack was attempted. With this older operating system the cameras worked so the problem was not hardware-related. Knowing this, the camera manufacturers were contacted; according to them, the drivers available on their website were not the most recent ones, so the new drivers were personally sent by email. However, even with the new updated packages, the cameras still could not be detected, so the contact was kept with the camera manufacturers for around 4 weeks. During these 4 weeks, multiple attempts were tested, after having re-flashed the Jetson over 30 times trying different approaches without success, a confirmation from the manufacturers was recieved. This confirmation stated that the camera drivers of these cameras reached End-Of-Life and there were no new updated drivers. This information was exactly the opposite of the information given during the first contact. Having spent all this time without any results, a decision had to be made regarding the JetPack version to use, the advantages and disadvantages of each can be seen in Figure 4.50. To use the newest version of JetPack and DeepStream another camera would have to be used. Previous works in the ATLASCAR2 used a Logitech C170 webcam, since this camera was already available in the laboratory it was chosen as an alternative temporary solution. This camera is not as optimized for the Jetson, offers a lower resolution and does not have multiple streams (since it is just one camera). Considering the e-CAM130 has reached its End-Of-Life any future software updates on the Jetson AGX Xavier would render the code made on this dissertation useless and frozen in time. So JetPack 4.6 was opted as the operating system for this dissertation with the Logitech C170 as an intermediate solution that could be changed in future works.

The results of the real-time inference with this camera are shown in Figure 4.51. These tests were performed near the University of Aveiro in different scenarios; the inference speed was similar to the tests above, with a slight (1-2 FPS) drop when using the webcam. Figure 4.51 (a) shows a car approaching from a distance in a very crowded parking lot; despite the numerous detections performed; the model still manage to track the cars, for example Car 5056 represented in yellow, with the same ID since the moment it became visible to the moment it left the camera view. In Figure 4.51 (b) two pedestrians were crossing each other at different speeds, the one closest to the camera walking and the one further away running. Regardless of the total occlusion the pedestrian further away suffered while crossing the road, the model still manages to attach the correct IDs, maintaining the correct ones all trough-out the interaction. In Figure 4.51 (c) tracking is performed over longer time periods, with car 473 being tracked for a longer distance, leaving the camera view; in the opposite direction car 488 is being tracked at the same time, in a similar scenario, but in an opposite direction, and all of this while still performing the detection and tracking of the cars parked on the right. The tracker successfully performed tracking through occlusions, dense traffic scenarios and during long periods of time. This detection, however, exhibited a lot of jittering on the videos. When detecting distant objects often the label IDs get swaped. This is due to the lower camera resolution that makes objects further away completely blurred in the input image to detect.

To explore the possibility of multi-camera inference some extra tests were performed. The e-CAM130 cameras did not work with the most recent software but, they were used in a past dissertation [3] to collect video data. This video stream data was then passed to the DeepStream pipeline simultaneously, simulating a multi-camera inference scenario.

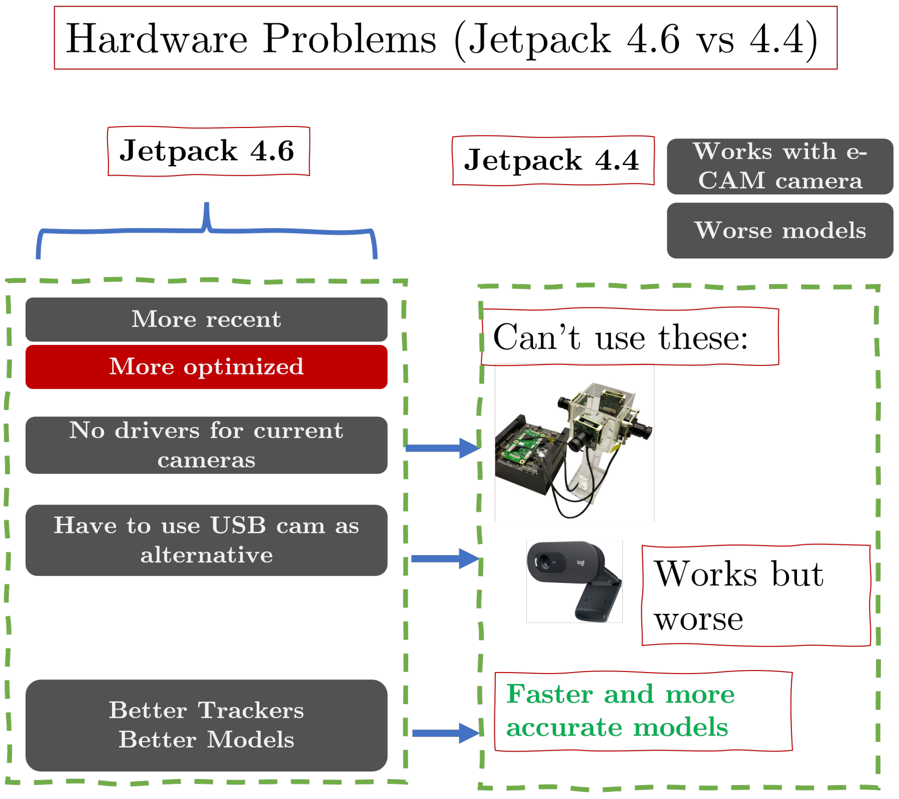
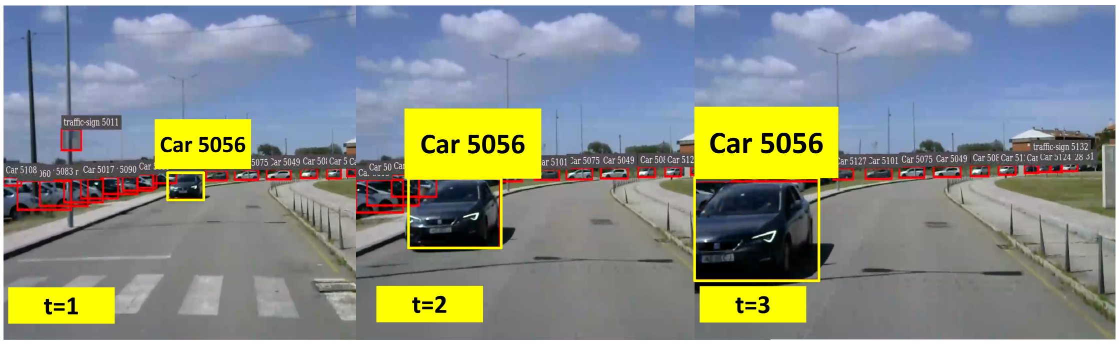
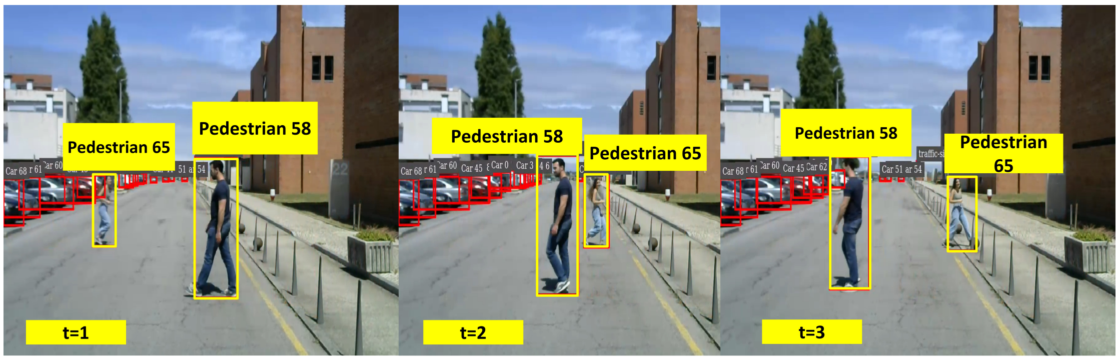

Hardware Problems (Jetpack 4.6 vs 4.4)
Jetpack 4.6
Jetpack 4.4
Works with e-CAM camera
Worse models
More recent
More optimized
No drivers for current cameras
Have to use USB cam as alternative
Better Trackers Better Models
Can't use these:
Works but worse
Faster and more accurate models

Figure 4.50: Summary of the Hardware problems that come with different JetPack versions.

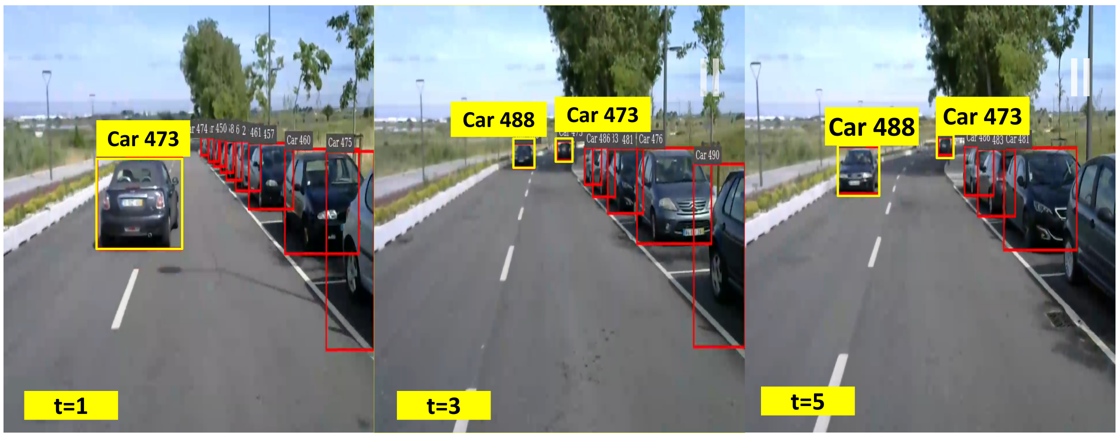
In the works of [3] the three cameras were used to create a panorama; in Figure 4.52 the real-time simultaneous inference of these three video streams can be seen. These tests prove that multi-camera inference is possible. It is also possible to see that the model can still detect motorcycles as shown in Figure 4.52 (b) (this is important to notice since this class was merged into the car class in the previous subsections to improve the detection capability). The traffic sign detection is successful even when they are situated very far away as seen in Figure 4.52 (c). This shows an improvement over the live camera inference performed previously on objects situated further away, this difference is most likely due to the difference in resolution of the cameras. A false detection seems to appear in Figure 4.52 (b) nearby the black car in the center of the image, however, the model is detecting the reflection of the car in the window. This test was performed with YOLOR-CSP, despite being the best detector, for single camera inference, when scaling to multiple cameras, the FPS drop to around 12 FPS with an interval of one and 17.5 FPS with an interval of three. This shows that perhaps, as more cameras are used the smaller the model architecture needs to be, assuming the case of three cameras, YOLOv5-m would provide better trade-off between speed and accuracy. This also opens the door for future studies for example, the resolution of the frontal camera does not need to be the same as the lateral cameras because the main purpose of these cameras is to provide redundancy. So, for example, YOLOR-CSP could be used in the frontal camera and YOLOv5-s could be used in the side cameras.



(a) Car aproaching from the distance in a parking lot with many cars



(b) Pedestrians crossing each other with complete occlusion

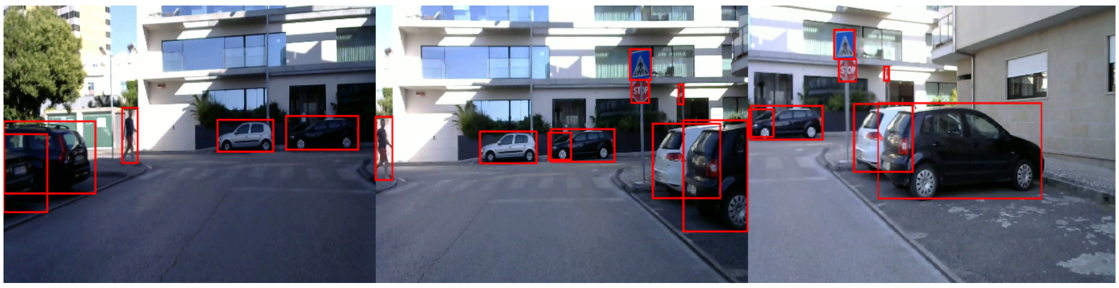


(c) Tracking of multiple cars on a large time-frame with many cars

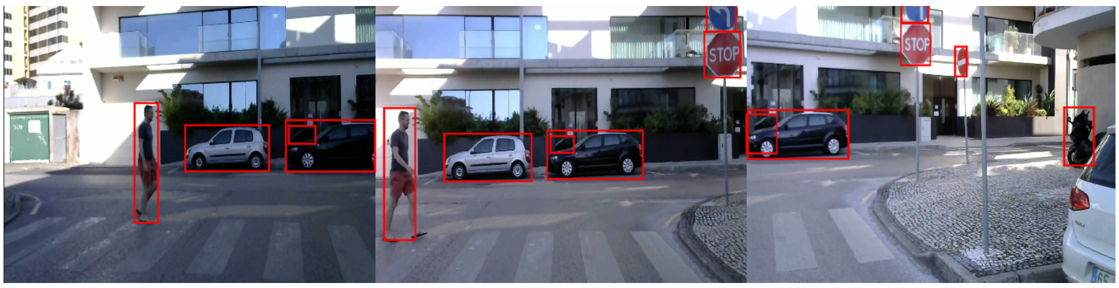
Figure 4.51: Real-Time Inference tests performed on ATLASCAR2 with Logitech C70 camera with YOLOR-CSP + DeepSORT. The bounding boxes in red are the outputs of the model, the ones in yellow were edited to illustrate the tracking to the reader.

4.3.5 Experiments with Multi-Task Neural Networks

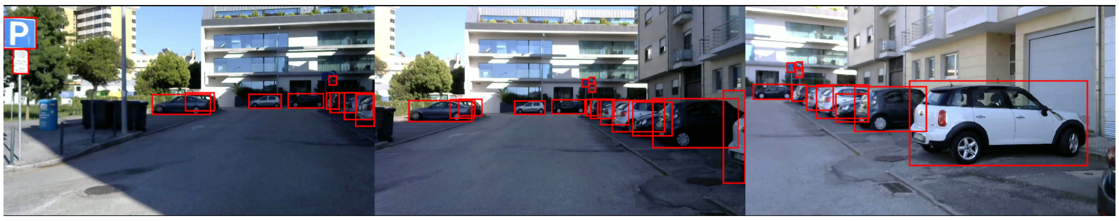
Experiments with multi-task networks were run to explore possible future paths for the visual perception on ATLASCAR2. These solutions offer outputs for multiple tasks required for autonomous driving using a single neural network. This multi-tasking is the main idea behind YOLOR paper, although, at the time of writing this thesis, its only



(a)



(b)



(c)

Figure 4.52: Inference on three video streams simultaneously. Each image (a),(b) and (c) represent a rendered tiled display with the three streams concatenated. Running at 12.5 FPS with YOLOR-CSP+NvDCF and detection interval of one.

output task is object detection. On the other hand, authors like [66] have proven that a single neural network can perform detection while outputting semantic segmentation and lane segmentation simultaneously, as seen by its architecture in Figure 4.53. The authors of [66] achieved similar results to the YOLOv5-Small detector while maintaining inference speeds of 20+ FPS on a Jetson TX2 and a Zebra camera. It was also trained on the BDD100K. This means that the pre-trained model provided by the author reflected an urban driving scenario. This specific algorithm (YOLOP) gives an inferior (less accurate) solution than the current DeepStream YOLOR + Tracker solution, but it shows a possible path for future solutions on the ATLASCAR2 system.

The results of the tests can be seen in Figure 4.54. These results were performed at the University of Aveiro campus where the goal of the Neural Network was to output car detections, segmentation of the road and detect the road lines. As shown in 4.54 (c), the model offers promising results, however, it is far from perfect. For example, the model has some trouble with speed bumps, making the semantic segmentation of the road perform worse compared to the less challenging scenarios found in Figure 4.54 (g).

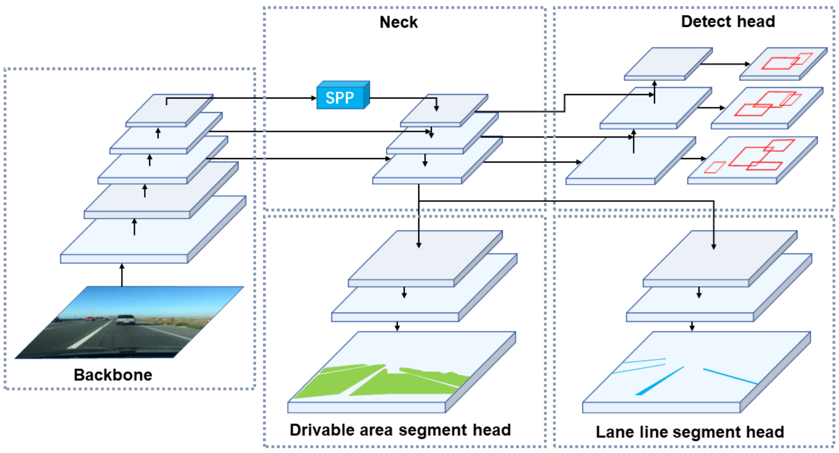
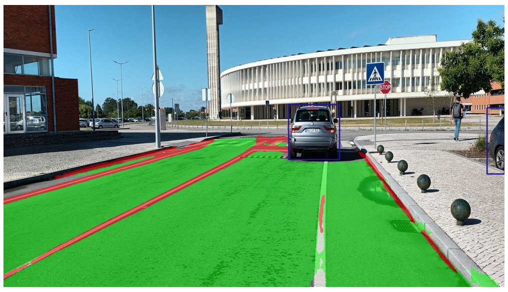


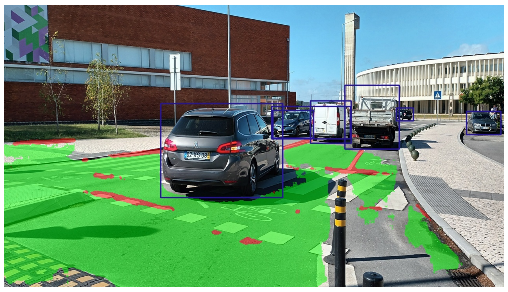
Figure 4.53: YOLOP Architecture [66].

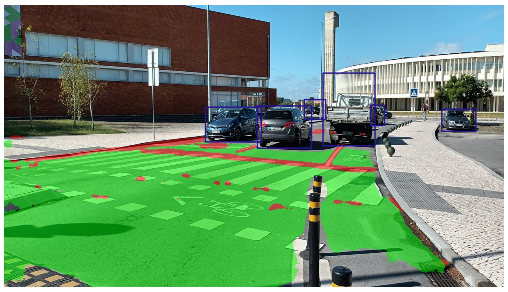
It also had troubles in Figure 4.54 (a) to detect the lines of the bicycle lane; this might be due to the difference between roads markings in north America (where BDD100K was recorded) compared to the ones in Aveiro. This specific model is only outputting car detections on this test even though it has the capability for more.

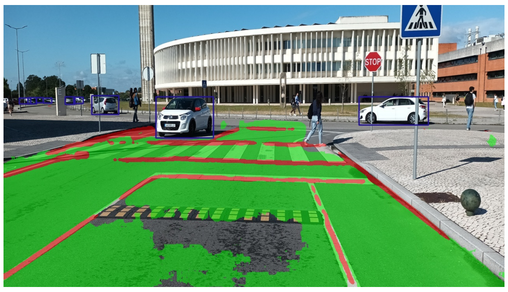
The tests were done without a TensorRT engine deployment (run on pure PyTorch), so it was not an optimized deployment, therefore, the inference speed was very poor (4 FPS). However, the results were promising, achieving notable car and lane detections. With TensorRT deployment and further optimization these Neural Networks open the doors to future works.

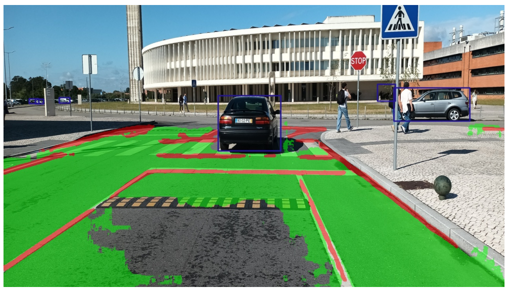
This chapter started with the training and deployment of the object detectors and all the decisions around their training. Starting with a techonlogy analysis, a dataset analysis, a model architecture and inference speed analysis and, finally, the model training and deployment. The best object detector achitecture in terms of accuracy and inference speed was the YOLOR-CSP. With the detectors working, the trackers were implemented and evaluated. With all the models and trackers tested and deployed, the one with the best compromise between speed an accuracy was the YOLOR-CSP reaching a mAP@50 of 70.50% implemented with DeepSORT in DeepStream, achieving 17 FPS with a detection interval of zero and 33.3 FPS with a detection interval of one evaluated on a single stream. When exploring multi-camera solutions YOLOR-CSP does not seem to be the best fit since there is a drop in FPS when running multiple streams. On the other hand, YOLOv5-m + NvDCF seems to be a better option since it is a lighter model and NvDCF works better with inference intervals different than zero; this is especially relevant because the redundancy achieved by multiple cameras allows for bigger inference intervals while loosing only a few detections. Multi-Task Neural Networks also showed promising results for future developments. These conclusions will be approached in greater detail in the next chapter.

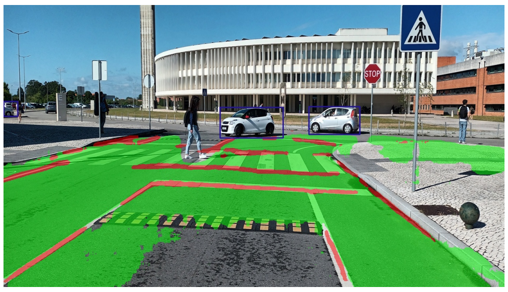

(a)

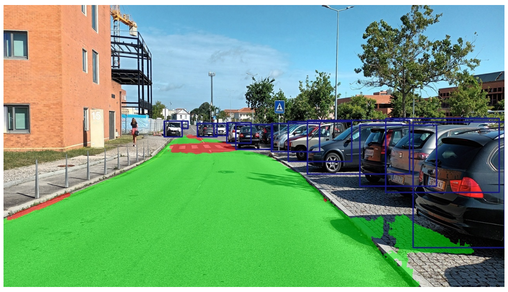

(b)


(c)


(d)


(e)


(f)


(g)

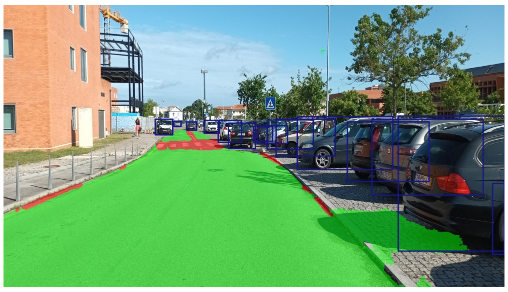

(h)

Figure 4.54: Inference tests with YOLOP on the University of Aveiro Campus; the green label is given to the road, red to the road lines and the blue bounding boxes to the cars detected.

Chapter 5

Conclusions

Thesis overview and final results

This thesis started out first with an introduction to the problem of object detection and tracking in the context of perception for a self-driving car. After this, it followed an introduction and explanation to the different SOTA technologies and algorithms for object detection and tracking. This was followed by an analysis of the experimental setting used (hardware and software) and finally the proposed solution with tests and results. In Chapter 4, first, the advantages and disadvantages of the NVIDIA Jetson AGX Xavier were analysed. Secondly, an analysis was performed regarding which technologies to use when developing the final solution. The technology of choice ended up being DeepStream since it allowed for faster development, modular integration, tracking plugins included in the SDK and, most importantly, it offered optimized solutions for video streams, allowing faster inference times and the ability to scale for multi-camera solutions.

With the technology stack chosen, multiple deployments were carried out on the Jetson AGX Xavier using many of the SOTA algorithms using the pre-trained weights. From these deployments, inference speed tests were run; these tests allowed for a pre-selection of the model architectures that could infer in real-time using the current hardware. With these architectures selected, the training dataset was chosen. Since an Aveiro dataset does not exist, an open dataset was used. Multiple datasets of urban driving scenarios were analysed; from these datasets BDD100K was chosen since it offered a larger number of scenarios with a better class distribution. To allow the models to train on this dataset, multiple format conversions were performed from the BDD100K format to the multiple formats that each model and framework required. Having the dataset in the right format, an experimental and training setup was created on DeepLar with the use of NVIDIA-Docker containers, SSH and W&B. After this, the first models were trained; however, the training was taking too long (multiple weeks to test one set of hyperparameters) and achieving poor mAP results. To combat this issue, different approaches were considered, including reducing the dataset size, caching the images on the GPUs RAM, more aggressive data augmentation and class-dropping/regrouping. Having performed all these adjustments, better accuracy was obtained in all the models trained; however, due to the smaller dataset size used, the model overfitted and was not able to detect pedestrians as accurately as cars or traffic signs. These multiple models were then deployed on the Jetson AGX Xavier but, unfortunately, since Scaled-YoloV4 could not be deployed as it was trained on PyTorch instead of Darknet, this made its weights not compatible with the TensorRT engine conversion code currently available for YoloV4. With the remaining models deployed; different trackers were implemented. Afterwords, different inference speed tests were performed with different detection intervals and both a visual and quantitative analysis of the trackers accuracy was performed. DeepSORT and NvDCF both provide similar accuracy with DeepSORT achieving slightly higher metrics, however, NvDCF scaled better for multi-camera solutions since it works better with different detection intervals. The current BYTE implementation is not as well optimized, lacking in performance when compared to the other trackers implemented. YOLOR-CSP architecture offers the best compromise between accuracy and speed, allowing for real-time detection and tracking. The visual and quantitative analysis did not always arrive to the same conclusion. For example, when comparing a model with lower accuracy YOLOv5-Small with YOLOR-CSP. There is some fragility in the current tracking evaluation metrics and they are still evolving. The initially intended hardware for deployment on ATLASCAR2 were the e-CAM130 cameras; however, these cameras had driver incompatibilities with the most recent software and ordering new ones in time for testing was not possible, so a Logitech C170 webcam was used instead. Finally some final tests were performed with Multi-Task Neural Networks. With all the models and trackers tested and deployed, the one with the best compromise between speed and accuracy was the YOLOR-CSP reaching a mAP@50 of 70.50% implemented with DeepSORT in DeepStream, achieving 17 FPS with an detection interval of zero and 33.3 FPS with a detection interval of one. The results of this solution are currently being explored for an article to be submitted in the "Autonomous Driving and Driver Assistance Systems" session which is part of the fifth Iberian Robotics Conference, ROBOT2022. The proposed solution:

- maximized the provided hardware capabilities, by utilizing the most recent optimization techniques and technologies;
- utilized the most recent SOTA algorithms and techniques, deploying it to a physical system that could infer in real time;
- through multiple tests, achieved the most optimal compromise between speed and accuracy;
- provided a modular solution for a pipeline that can evolve as time goes on, allowing for code longevity and for future upgrades as the SOTA progresses.
- provided datasets conversions of the BDD100K format to all YOLO formats so the research community can benefit from it as a whole. Speeding up the testing and development of future algorithms;
- Provided multiple inference speed tests with different floating point percisions and detection intervals in the NVIDIA Jetson AGX Xavier. This information is valuable to the research community since it tests multiple SOTA detectors on recent hardware that can be used on other diverse autonomous systems other than autonomous driving;
- Created the first DeepStream evaluation pipeline in the KITTI dataset for a DeepStream Tracker. Achieving the first public comparisons metrics between DeepSORT and NvDCF.

Future Work

Multiple future paths were open with this thesis. From a hardware standpoint, the current solution used a Logitech C170, its 720p resolution and slower bus makes it not being the best fit, especially when analysing distant objects; different camera options can be considered in the future. When considering these cameras, an analysis has to be performed on the camera(s) positioning. In addition to this, solutions involving multiple detectors running simultaneously could be explored, for example, YOLOR-CSP running on the main frontal camera and YOLO-S running on the lateral cameras. The current solution placed the NVIDIA Jetson AGX Xavier module on the car seat, this needs to be changed; a support must be created and attached to the ATLASCAR2 with the NVIDIA Jetson AGX Xavier in mind.

From a software standpoint, this 2D object detection could be used to perform 3D detection by performing sensor fusion with the rest of ALTASCAR2 sensors. In addition to this, stereo cameras could be used to gather a tracked object speed. Finishing the hyperparameter evolution would provide an accuracy increase, the same goes for using these hyperparameters to train on the full BDD100K dataset again, hopefully increasing the effectiveness of pedestrian detection. Finally, multi-tasked neural networks could be further explored and deployed on the Jetson AGX Xavier.

Appendix A

Gentle Introduction to Deep Learning

This chapter introduces some Deep Learning concepts starting with Supervised Learning, followed by an explanation on Neural Networks and linear models.

A.1 Neural Networks in Supervised Learning

Supervised Learning

Machine Learning is divided in to two main fields, supervised learning and unsupervised learning. This thesis focuses on supervised learning; supervised learning is a type of learning where the algorithm trains on labeled (supervised) data. After training, the algorithm tries to predict, given a specific input, as close to possible to the ground truth labels. Neural Networks can be explained with higher-level abstractions that facilitate their understanding. The simplest way to understand them is to start from a linear model.

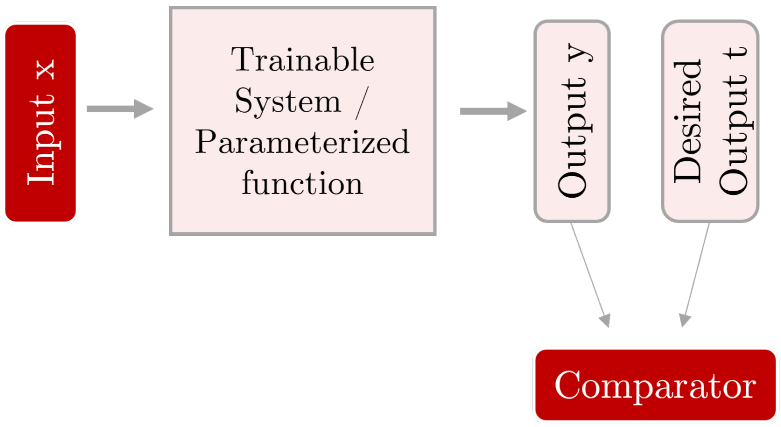


Figure A.1: Simplified example of a Neural Network.

A linear model can be thought of as a simple function where $y = mx + b$, given some input data x the model tries to obtain an output y. The transformation between output and input can be thought of as a series of mathematical operations.

In supervised learning, the output is compared with the desired output by a comparator, with the results of this comparison the weights and bias of the network are adjusted. So, for example, when considering a cat classication model (model that classifies images of cats) that receives as an input x a RGB image of a cat. The output of this model would be the probability of x being a cat. If the image provided is, in fact, an image of a cat; the desired output of our model t would be 1 for the class cat (or in practical terms 0.999...), assuming the model predicted an output y of 0.7 these two values are compared and then the parameters m and b are adjusted in order to obtain a more accurate model. A scheme to illustrate the relashionsips between input and output is shown in Figure A.1. In reality, a model of a Deep Neural Network is lot more complicated, instead of simple parameters m and b we have matrices with sometimes millions of parameters. These are usually called weights and bias.

Neural Networks

Neural networks, also known as artificial neural networks (ANNs), are a subset of machine learning. They received their name and structure from the inspiration of the human brain itself, mimicking the way biological neurons signal to each other. They are composed of layers, containing an input layer, one or more hidden layers, and an output layer. Each neuron connects to another neuron and has an associated weight [67].

Considering the model of a simple neuron shown in Figure A.2, the inputs x weights $+$ bias are all summed up originating an output. It is a simple model that works well on linearly separable problems.

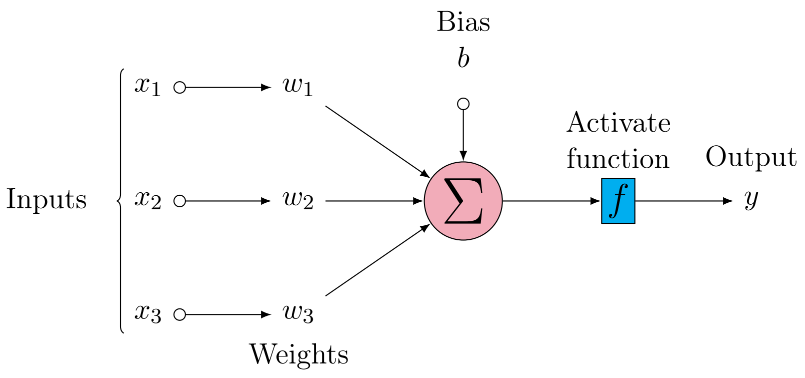


Figure A.2: Single-Neuron representation.

A geometric interpretation of a single neuron would be a line that can best separate the data clusters as shown in Figure A.3.

Neural Networks are made with different layers of neurons. They can have multiple hidden layers with millions of layers, the name Deep Neural Network derives from the depth of layers in the network. Figure A.4 shows a representation of a Deep Feed Forward Neural Network.

In the example of this simple Neural Network, neurons of the same type are grouped into layers, there are no connections between neurons in one layer, in the case of a Feed Forward Neural Networks (one of the simplest types of Neural Networks), each neuron of the previous layer is connected with each neuron of the next layer. The number of

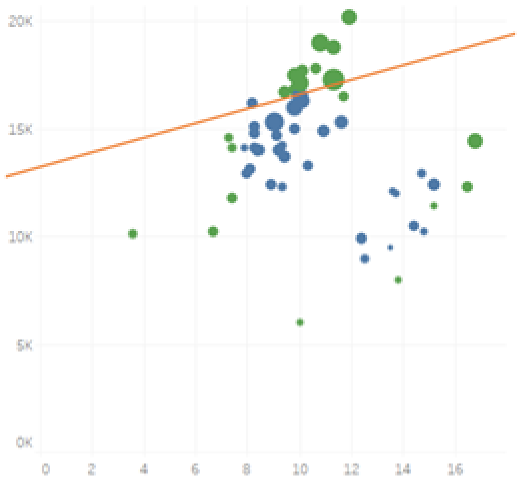


Figure A.3: Line attempt to separate the points; adapted from [68].

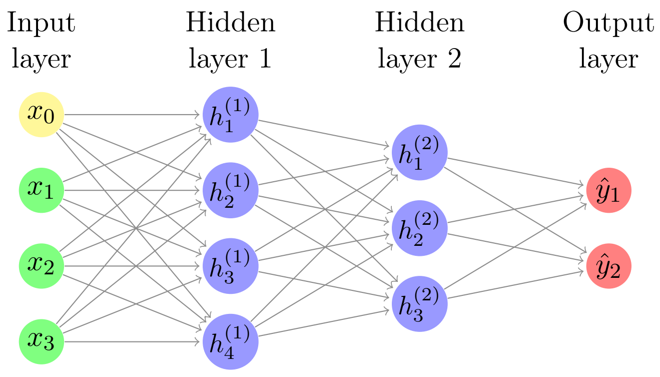


Figure A.4: Typical FF Neural Network.

inputs is dictated by the input data, the number of outputs depends on the problem at hand, the number of hidden layers is considered a hyperparameter. Working with the size and shape of a network is one of the problems in trying to find an optimal solution.

A.2 Training of a Neural Network

When connecting larger models with multiple neurons, we must define a certain threshold for the neuron, this will dictate if the neuron is active or not, for that we use activation functions.

In order to achieve a comparison between output and desired output a "mechanism" needs to be defined. The mechanism to perform this comparison is defined as a loss function.

A.2.1 Loss functions

A loss function measures the difference between the desired output values and the actual output, this loss should be as small as possible (ideally 0), this would mean that the

Neural Network outputs are equal to the target outputs.

Mean Squared Error Loss

- The standard loss function to perform regression tasks

For N training objects and one real output, MSE is given by the following:

$$\mathrm{J}(\theta)=\frac{1}{N}\sum\left(\mathrm{y}^{\mathrm{i}}-t^{i}\right)^{2} \tag{A.1}$$

Where y and t are the output and desired output as shown in Figure A.1. The same nomenculature will be used when defining following functions.

Multiclass (categorical) Cross-Entropy loss

- Standard loss function for multiclass classification tasks

For N training objects and K classes:

$$J(\theta)=\frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\left[-t_{k}^{(i)}\log\left(y_{k}^{(i)}\right)\right] \tag{A.2}$$

$$J(\theta)=\frac{1}{N}\sum_{i=1}^{N}-\mathbf{t}^{(i)}\log\mathbf{y}^{(i)} \tag{A.3}$$

This way, the model has exactly K outputs.

Binary Cross-Entropy loss

- For binary classification problems
- The type of output unit should be between 0-1

The cross-entropy formula has the following form:

$$J(\theta)=-\frac{1}{N}\sum_{i=1}^{N}\left[t^{(i)}\log\left(y^{(i)}\right)+\left(1-t^{(i)}\right)\log\left(1-y^{(i)}\right)\right] \tag{A.4}$$

A.2.2 Gradient Descent

Given some training data, a parameterized model, and a chosen loss function; the goal is to change the parameters such that the loss function achieves a global minimum. This can be thought of as finding the global minimum of a multidimensional function.

This can be achieved in a number of ways, but the most popular solution is gradient-based learning shown in Figure A.5.

Essentially, the local steepness of the function (its gradient) is calculated and a small increment step in the direction of the steepest descent is taken. This requires however that the function is differentiable in every point. A good way to think about it would be a continuous and smooth multidimensional function. So, for example, the error rate could not be chosen as a loss function because it is not smooth/differentiable.

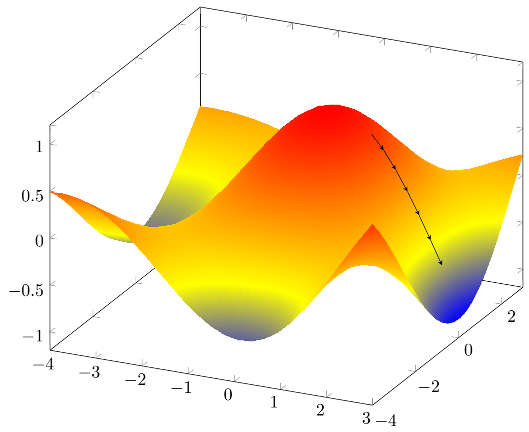


Figure A.5: Example of Gradient Descent; the black arrow represents the iteration steps.

A gradient is a vector of all partial derivatives of a function and all its arguments.

$$\nabla(J(\theta) = \frac{\partial J}{\partial \theta} = \begin{bmatrix} \frac{\partial J}{\partial \theta_1} \\ \frac{\partial J}{\partial \theta_2} \\ \vdots \\ \frac{\partial J}{\partial \theta_p} \end{bmatrix}. \tag{A.5}$$

Gradient descent is an iterative method for finding the local minimum of a function. It is represented by the following:

$$\theta^{i+1} = \theta^i - \alpha\nabla(J(\theta)) \tag{A.6}$$

α is the learning rate, a small scalar that essentially defines how fast the increment steps are taken, or, in other words, 'how fast the Neural Network learns'.

A.3 Activation Functions

When passing information to one neuron to another an activation function needs to be chosen. Activation functions transform the weighted sum of the previous neuron (or group of neurons) in to an input for the following neuron; the output of this function dictates if the neuron is active or not.

Logistic Sigmoid Activation Function

Is given by the following expression:

$$\sigma(z) = \frac{1}{1+e^{-z}} \tag{A.7}$$

The plot of this function is shown in Figure A.6.

This activation function "compacts" the output of a neuron in a regime between 0 and 1. This is useful when calculating the gradient because the large difference in weights could cause the output of each neuron to fire out of proportion causing some problems while calculating the gradient. This function is also differentiable everywhere

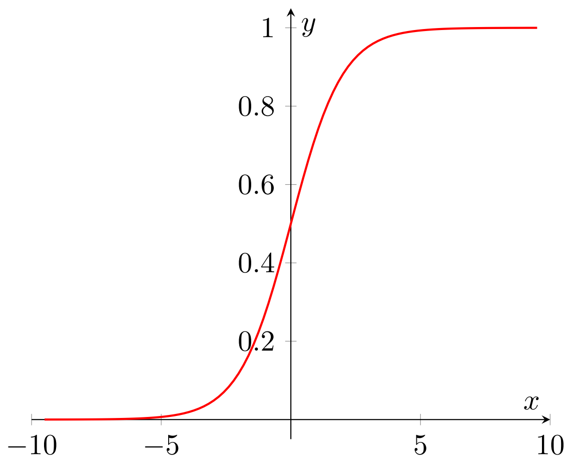


Figure A.6: Sigmoid activation function

which allows to calculate the gradient. One disadvantage is that, since it does not have negative values, when the when the values of the gradient approach zero the Neural Network the weights do not updated; this is called vanishing gradient problem. Due to it is 0-1 nature, its typically used as:

- a single output unit for binary classification problems
- as one of the output units in an output vector (for multilabel classification problems)
- In nets with sub components working as differentiable gates

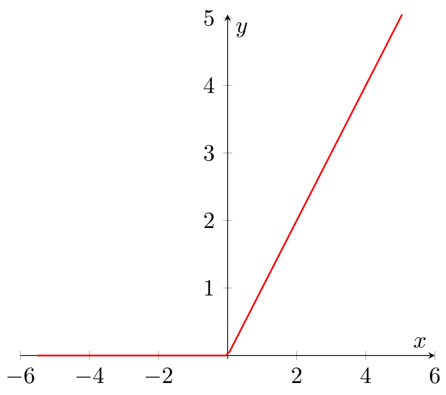
ReLu Activation Function

The ReLU activation function or Rectified Linear Unit, now, the most common (default) activation function used has some advantages in comparison to sigmoid. Its easier to compute and performs well, although it has a non-differentiable point, it is negligible.

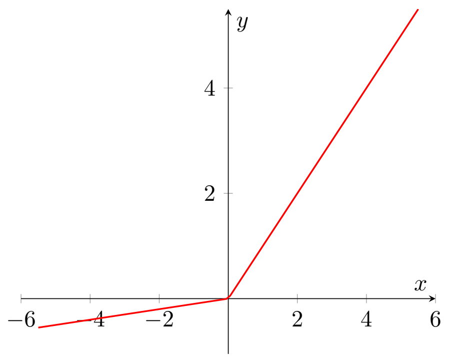
$$\text{ReLU}(z) = \begin{cases} 0 \text{ if } z < 0 \\ z \text{ if } z \geq 0 \end{cases} \tag{A.8}$$

The plot of this function is shown in Figure A.7 (a).

As demonstrated, if the neuron outputs a value less then zero it will be turned off. This type of activation function works better than sigmoid in the hidden layers of the model since it does not have the problem of the shrinking gradient. The problem with sigmoid is that its local gradient is less than 1 and often very close to zero, when preforming gradient descent often it becomes so small it vanishes. On the other hand, the local gradient of ReLU in the active region is 1. In the inactive region is 0 which presents another problem, when all examples from training data are not activating the ReLU the neural it will not change its weights and it will remain inactive, we call this problem the dying ReLU. Therefore, other activation functions started to be used in addiction to normal ReLU like Leaky ReLU. This function is represented in Figure A.7 (b).



(a) ReLU activation function



(b) Leaky ReLU

Figure A.7: ReLU and Leaky ReLU activation functions

SoftMax output activation/layer

It's mainly used for classification tasks. It can be thought of as an activation function, but instead of being applied to a single neuron, it is applied to all the neurons in the layer at once. Typically For multi-class single label classification.

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad for\ i = 1, 2, \ldots, K \tag{A.9}$$

This function is very helpful since it assures that all the outputs are positive, and the sum of all outputs is equal to 1. This allows for a visual interpretation of the outputs as probabilities. To further illustrate this point, considering a classification problem where y_k is the probability that the input belongs to class k.

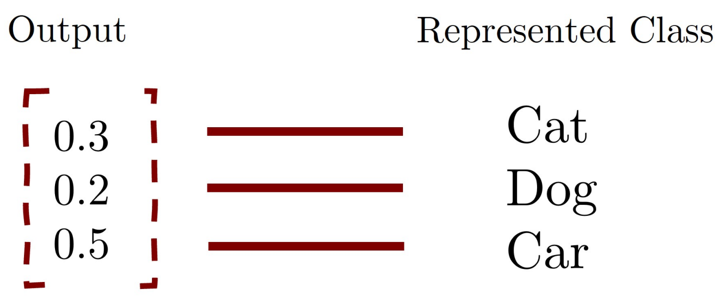


Figure A.8: Typical output of multi-class classification problem

We can directly correlate the number of the output to a probability of a class, so in the case of figure A.8 the input would be most likely a car since it has the highest probability. However, with a very low level of confidence (0.5).

Intentionally blank page.

Appendix B

Configuration Files

Here are presented the configuration files used in the training.

B.1 YOLOv5 Hyperparameters

High Augmentation Hyperparameters:

```
lr0: 0.01  # initial learning rate (SGD=1E-2, Adam=1E-3)
lrf: 0.1  # final OneCycleLR learning rate (lr0 * lrf)
momentum: 0.937  # SGD momentum/Adam beta1
weight_decay: 0.0005  # optimizer weight decay 5e-4
warmup_epochs: 3.0  # warmup epochs (fractions ok)
warmup_momentum: 0.8  # warmup initial momentum
warmup_bias_lr: 0.1  # warmup initial bias lr
box: 0.05  # box loss gain
cls: 0.3  # cls loss gain
cls_pw: 1.0  # cls BCELoss positive_weight
obj: 0.7  # obj loss gain (scale with pixels)
obj_pw: 1.0  # obj BCELoss positive_weight
iou_t: 0.20  # IoU training threshold
anchor_t: 4.0  # anchor-multiple threshold
fl_gamma: 0.0  # focal loss gamma (efficientDet default gamma=1.5)
hsv_h: 0.015  # image HSV-Hue augmentation (fraction)
hsv_s: 0.7  # image HSV-Saturation augmentation (fraction)
hsv_v: 0.4  # image HSV-Value augmentation (fraction)
degrees: 0.1  # image rotation (+/- deg)
translate: 0.1  # image translation (+/- fraction)
scale: 0.9  # image scale (+/- gain)
shear: 0.1  # image shear (+/- deg)
perspective: 0.0  # image perspective (+/- fraction), range 0-0.001
flipud: 0.0  # image flip up-down (probability)
fliplr: 0.5  # image flip left-right (probability)
mosaic: 1.0  # image mosaic (probability)
mixup: 0.1  # image mixup (probability)
copy_paste: 0.1  # segment copy-paste (probability)
```

Low Augmentation Hyperparameters:

```
lr0: 0.01  # initial learning rate (SGD=1E-2, Adam=1E-3)
lrf: 0.01  # final OneCycleLR learning rate (lr0 * lrf)
momentum: 0.937  # SGD momentum/Adam beta1
weight_decay: 0.0005  # optimizer weight decay 5e-4
warmup_epochs: 3.0  # warmup epochs (fractions ok)
warmup_momentum: 0.8  # warmup initial momentum
warmup_bias_lr: 0.1  # warmup initial bias lr
box: 0.05  # box loss gain
cls: 0.5  # cls loss gain
cls_pw: 1.0  # cls BCELoss positive_weight
obj: 1.0  # obj loss gain (scale with pixels)
obj_pw: 1.0  # obj BCELoss positive_weight
iou_t: 0.20  # IoU training threshold
anchor_t: 4.0  # anchor-multiple threshold
# anchors: 3  # anchors per output layer (0 to ignore)
fl_gamma: 0.0  # focal loss gamma (efficientDet default gamma=1.5)
hsv_h: 0.015  # image HSV-Hue augmentation (fraction)
hsv_s: 0.7  # image HSV-Saturation augmentation (fraction)
hsv_v: 0.4  # image HSV-Value augmentation (fraction)
degrees: 0.0  # image rotation (+/- deg)
translate: 0.1  # image translation (+/- fraction)
scale: 0.5  # image scale (+/- gain)
shear: 0.0  # image shear (+/- deg)
perspective: 0.0  # image perspective (+/- fraction), range 0-0.001
flipud: 0.0  # image flip up-down (probability)
fliplr: 0.5  # image flip left-right (probability)
mosaic: 1.0  # image mosaic (probability)
mixup: 0.0  # image mixup (probability)
copy_paste: 0.0  # segment copy-paste (probability)
```

B.2 YOLOR Hyperparameters

```
warmup_bias_lr: 0.1  # warmup initial bias lr
box: 0.05  # box loss gain
cls: 0.5  # cls loss gain
cls_pw: 1.0  # cls BCELoss positive_weight
obj: 0.7  # obj loss gain (scale with pixels)
obj_pw: 1.0  # obj BCELoss positive_weight
iou_t: 0.20  # IoU training threshold
anchor_t: 4.0  # anchor-multiple threshold
# anchors: 3  # anchors per output layer (0 to ignore)
fl_gamma: 0.0  # focal loss gamma (efficientDet default gamma=1.5)
hsv_h: 0.015  # image HSV-Hue augmentation (fraction)
hsv_s: 0.7  # image HSV-Saturation augmentation (fraction)
hsv_v: 0.4  # image HSV-Value augmentation (fraction)
```

```
degrees: 0.8  # image rotation (+/- deg)
translate: 0.5  # image translation (+/- fraction)
scale: 0.8  # image scale (+/- gain)
shear: 0.1  # image shear (+/- deg)
perspective: 0.0  # image perspective (+/- fraction), range 0-0.001
flipud: 0.0  # image flip up-down (probability)
fliplr: 0.5  # image flip left-right (probability)
mosaic: 1.0  # image mosaic (probability)
mixup: 0.2  # image mixup (probability)
```

B.3 Scaled YoloV4 HyperParameters

```
lr0: 0.01  # initial learning rate (SGD=1E-2, Adam=1E-3)
momentum: 0.937  # SGD momentum/Adam beta1
weight_decay: 0.0005  # optimizer weight decay 5e-4
giou: 0.05  # GIoU loss gain
cls: 0.3  # cls loss gain
cls_pw: 1.0  # cls BCELoss positive_weight
obj: 0.62  # obj loss gain (scale with pixels)
obj_pw: 1.0  # obj BCELoss positive_weight
iou_t: 0.20  # IoU training threshold
anchor_t: 4.0  # anchor-multiple threshold
fl_gamma: 0.0  # focal loss gamma (efficientDet default gamma=1.5)
hsv_h: 0.015  # image HSV-Hue augmentation (fraction)
hsv_s: 0.7  # image HSV-Saturation augmentation (fraction)
hsv_v: 0.4  # image HSV-Value augmentation (fraction)
degrees: 0.8  # image rotation (+/- deg)
translate: 0.5  # image translation (+/- fraction)
scale: 0.9  # image scale (+/- gain)
shear: 0.1  # image shear (+/- deg)
perspective: 0.0  # image perspective (+/- fraction), range 0-0.001
flipud: 0.0  # image flip up-down (probability)
fliplr: 0.5  # image flip left-right (probability)
mixup: 0.2  # image mixup (probability)
```

B.4 DeepStream Configs

```
[application]
enable-perf-measurement=1
perf-measurement-interval-sec=5
kitti-track-output-dir=/xavier_ssd/Tracker_Info
#gie-kitti-output-dir=/xavier_ssd/Tracker_Info

[tiled-display]
enable=1
rows=1
columns=1
width=1280
height=720
gpu-id=0
nvbuf-memory-type=0

#Video Source
[source0]
enable=1
type=3
#uri=file:///opt/nvidia/deepstream/deepstream/samples/streams/sample_1080p_h264.mp4
uri=file:///opt/nvidia/deepstream/deepstream/samples/streams/sample_qHD_short.mp4
num-sources=1
gpu-id=0
cudadec-memtype=0

#Camera Source
#[source0]
#enable=0
# Type - 1=CameraV4L2 2=URI 3=MultiURI
#type=1
#camera-width=640
#camera-height=480
#camera-fps-n=30
#camera-fps-d=1
#camera-v4l2-dev-node=0
#num-sources=1

#Camera Source
[source1]
enable=0
# Type - 1=CameraV4L2 2=URI 3=MultiURI
type=1
camera-width=640
camera-height=480
camera-fps-n=30
```

```
camera-fps-d=1
camera-v4l2-dev-node=1
num-sources=1

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

[sink1]
enable=1
type=3
container=1
sync=0
codec=1
bitrate=2000000
output-file=out.mp4

[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

[streammux]
gpu-id=0
live-source=0
batch-size=1
batched-push-timeout=40000
width=1920
height=1080
enable-padding=0
nvbuf-memory-type=0

[primary-gie]
enable=1
```

```
gpu-id=0
interval=0
gie-unique-id=1
nvbuf-memory-type=0
config-file=config_infer_primary_yolor.txt

[tracker]
enable=1
tracker-width=640
tracker-height=384
# -- Change Tracker height for better perform
#tracker-height=640
gpu-id=0
#ll-lib-file=/opt/nvidia/deepstream/deepstream/lib/libByteTracker.so
ll-lib-file=/opt/nvidia/deepstream/deepstream/lib/libnvds_nvmultiobjecttracker.so
#ll-lib-file=/opt/nvidia/deepstream/deepstream/lib/libnvds_nvdcf.so
#ll-config-file=/opt/nvidia/deepstream/deepstream/samples/...
.../configs/deepstream-app/config_tracker_DeepSORT.yml
#enable-past-frame=1
enable-batch-process=1


#[img-save]
#enable=1
#output-folder-path=/xavier_ssd/output
#save-img-cropped-obj=0
#save-img-full-frame=1
#frame-to-skip-rules-path=capture_time_rules.csv
#second-to-skip-interval=600
#min-confidence=0.9
#max-confidence=1.0
#min-box-width=5
#min-box-height=5


[tests]
file-loop=0
```

Glossary

Batch Size number of training examples utilized in one iteration..

Cross mini-Batch Normalization method of batch normalization that can be applied to small batches..

DeepLar Model training machine at LAR..

DeepStream SDK A Development kit made by NVIDIA to rapidly develop and deploy Vision AI applications and services..

DenseNet Type of convolutional neural network that utilises dense connections between layers..

Docker Volumes Mechanism for persisting data generated by and used by Docker containers..

DropBlock Regularization A form of structured dropout, where units in a contiguous region of a feature map are dropped together..

e-CAM130-CUXVR Set of 4 MPI Cameras that integrate on the Jetson Xavier AGX.

Edge Devices Physical devices where algorithms are run at the "edge", this means nearby the sensors in which the real data is being collected..

EfficientDet type of object detection model which utilizes several optimization and backbone tweaks and a compound scaling method that uniformly scales the resolution, depth and width..

EfficientNet convolutional neural network architecture and scaling method that uniformly scales all dimensions of depth/width/resolution using a compound coeficcient..

Epoch complete pass of the training dataset trough the algorithm..

Mask-RCNN framework for object instance segmentation that extends Faster R-CNN by adding a branch for predicting an object mask in parallel with the existing branch for bounding box recognition..

maxShadowTrackingAge The maximum time an object is being tracked after being completely occluded.

Mish Activation function used by YOLOV4.

Receptive Field Size of the region in the input that produces the feature..

ReLU Popular activation function (used for example in YoloV5).

Shadow Tracking target is still being tracked in the background for a period of time even when the target is not associated with a detector object..

Tracklets small set of paths associated with individual detections in consecutive frames..

Transformers Transformer is a Deep Learning model that uses the mechanism of self-attention, deferentially weighting the significance of each part of the input data..

Vanishing Gradient Problem caused by the derivative of the activation function used to create the Neural Network. The Networks are unable to back propagate the gradient information to the model..

References

[1] D. Figueiredo, "Remote control for operation and driving of atlascar2," *UA Master Thesis*, 2020.

[2] MathWorks. What is deep learning? | how it works, techniques & applications - matlab & simulink. Acessed: 01/06/2022. [Online]. Available: https://www.mathworks.com/discovery/deep-learning.html

[3] R. D. F. da Costa, "Detection and classification of road and objects in panoramic images on board the atlascar2 using deep learning," *UA Master Thesis*, 2020.

[4] T. S. D. Freitas and F. Osório, "Deteção de objectos para carros e pedestres através de deep learning - object detection for cars and pedestrians using deep learning," *UA Master Thesis*, 2018.

[5] R. P. L. C. Costa, "Multi target tracking and detection using lidar and velocity obstacles for real time definition of collision zones," *UA Master Thesis*, 2020.

[6] S. D. Pendleton, H. Andersen, X. Du, X. Shen, M. Meghjani, Y. H. Eng, D. Rus, and M. H. Ang, "Perception, planning, control, and coordination for autonomous vehicles," *Machines*, vol. 5, 3 2017.

[7] S. Chung, K. Shek, J. Butterfield, A. Murphy, J. Butterfield, and I. Spence, "Current state of the art in object detection for autonomous systems," 2021. [Online]. Available: https://www.researchgate.net/publication/354786902

[8] S. K. Pal, A. Pramanik, J. Maiti, and P. Mitra, "Deep learning in multi-object detection and tracking: state of the art," *Applied Intelligence*, vol. 51, pp. 6400–6429, 9 2021.

[9] S. Liu, L. Li, J. Tang, S. Wu, and J.-L. Gaudiot, "Creating autonomous vehicle systems," *Synthesis Lectures on Computer Science*, vol. 8, no. 2, pp. i–216, 2020.

[10] P. Viola and M. Jones, "Rapid object detection using a boosted cascade of simple features," in *Proceedings of the 2001 IEEE computer society conference on computer vision and pattern recognition. CVPR 2001*, vol. 1. IEEE, 2001, pp. I–I.

[11] N. Dalal and B. Triggs, "Histograms of oriented gradients for human detection," in *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, vol. 1, 2005, pp. 886–893 vol. 1.

[12] M. Mayank. Convolutional neural networks, explained | towards data science. Acessed: 08/02/2022. [Online]. Available: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939

[13] A. Balasubramaniam and S. Pasricha, "Object detection in autonomous vehicles: Status and open challenges," *arXiv preprint arXiv:2201.07706*, 2022.

[14] M. Research. Image recognition: Current challenges and emerging opportunities - microsoft research. Acessed: 08/02/2022. [Online]. Available: https://www.microsoft.com/en-us/research/lab/microsoft-research-asia/articles/image-recognition-current-challenges-and-emerging-opportunities/

[15] F. Nobis, M. Geisslinger, M. Weber, J. Betz, and M. Lienkamp, "A deep learning-based radar and camera sensor fusion architecture for object detection," *2019 Symposium on Sensor Data Fusion: Trends, Solutions, Applications, SDF 2019*, 5 2020. [Online]. Available: https://arxiv.org/abs/2005.07431v1

[16] T. Wang, X. Zhu, J. Pang, and D. Lin, "Fcos3d: Fully convolutional one-stage monocular 3d object detection," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 913–922. [Online]. Available: https://github.com/open-mmlab/mmdetection3d.

[17] A. Bochkovskiy, C.-Y. Wang, and H.-Y. M. Liao, "Yolov4: Optimal speed and accuracy of object detection," *arXiv preprint arXiv:2004.10934*, 2020. [Online]. Available: https://github.com/AlexeyAB/darknet.

[18] R. Girshick, J. Donahue, T. Darrell, and J. Malik, "Rich feature hierarchies for accurate object detection and semantic segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2014, pp. 580–587.

[19] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," *Advances in neural information processing systems*, vol. 28, 2015.

[20] K. He, G. Gkioxari, P. Dollár, and R. Girshick, "Mask r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 2961–2969.

[21] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[22] S. Xie, R. Girshick, P. Dollár, Z. Tu, and K. He, "Aggregated residual transformations for deep neural networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 1492–1500. [Online]. Available: https://github.com/facebookresearch/ResNeXt

[23] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[24] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "Ssd: Single shot multibox detector," in *European conference on computer vision.* Springer, 2016, pp. 21–37.

[25] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning.* PMLR, 2019, pp. 6105–6114.

[26] P. W. Code. Coco benchmark (real-time object detection) | papers with code. Acessed: 08/02/2022. [Online]. Available: https://paperswithcode.com/sota/real-time-object-detection-on-coco

[27] Y. Fang, S. Yang, S. Wang, Y. Ge, Y. Shan, and X. Wang, "Unleashing vanilla vision transformer with masked image modeling for object detection," *arXiv preprint arXiv:2204.02964*, 2022.

[28] H. Zhang, F. Li, S. Liu, L. Zhang, H. Su, J. Zhu, L. M. Ni, and H.-Y. Shum, "Dino: Detr with improved denoising anchor boxes for end-to-end object detection," *arXiv preprint arXiv:2203.03605*, 2022.

[29] P. W. Code. Coco test-dev benchmark (object detection) | papers with code. Acessed: 08/02/2022. [Online]. Available: https://paperswithcode.com/sota/object-detection-on-coco

[30] R. Girshick, "Fast r-cnn," in *Proceedings of the IEEE international conference on computer vision*, 2015, pp. 1440–1448.

[31] M. Tan, R. Pang, and Q. V. Le, "Efficientdet: Scalable and efficient object detection," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2020.

[32] N. Bodla, B. Singh, R. Chellappa, and L. S. Davis, "Soft-nms–improving object detection with one line of code," in *Proceedings of the IEEE international conference on computer vision*, 2017, pp. 5561–5569.

[33] J. Redmon and A. Farhadi, "Yolo9000: better, faster, stronger," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 7263–7271.

[34] J. R. A. Farhadi, "Yolov3: An incremental improvement," *arXiv preprint arXiv:1804.02767*, 2018.

[35] G. Huang, Z. Liu, L. Van Der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, pp. 4700–4708. [Online]. Available: https://github.com/liuzhuang13/DenseNet.

[36] D. Misra, "Mish: A self regularized non-monotonic activation function," *arXiv preprint arXiv:1908.08681*, 2019. [Online]. Available: https://github.com/digantamisra98/Mish.

[37] Z. Zheng, P. Wang, W. Liu, J. Li, R. Ye, and D. Ren, "Distance-iou loss: Faster and better learning for bounding box regression," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 07, 2020, pp. 12 993–13 000.

[38] C.-Y. Wang, I.-H. Yeh, and H.-Y. M. Liao, "You only learn one representation: Unified network for multiple tasks," *arXiv preprint arXiv:2105.04206*, 2021.

[39] G. Jocher, A. Chaurasia, A. Stoken, J. Borovec, NanoCode012, Y. Kwon, TaoXie, J. Fang, imyhxy, K. Michael, Lorna, A. V, D. Montes, J. Nadar, Laughing, tkianai, yxNONG, P. Skalski, Z. Wang, A. Hogan, C. Fati, L. Mammana, AlexWang1900, D. Patel, D. Yiwei, F. You, J. Hajek, L. Diaconu, and M. T. Minh, "ultralytics/yolov5: v6.1 - TensorRT, TensorFlow Edge TPU and OpenVINO Export and Inference," Feb. 2022. [Online]. Available: https://doi.org/10.5281/zenodo.6222936

[40] C.-Y. Wang, H.-Y. M. Liao, Y.-H. Wu, P.-Y. Chen, J.-W. Hsieh, and I.-H. Yeh, "Cspnet: A new backbone that can enhance learning capability of cnn," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, 2020, pp. 390–391.

[41] S. Liu, L. Qi, H. Qin, J. Shi, and J. Jia, "Path aggregation network for instance segmentation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8759–8768.

[42] U. Nepal and H. Eslamiat, "Comparing yolov3, yolov4 and yolov5 for autonomous landing spot detection in faulty uavs," *Sensors*, vol. 22, no. 2, p. 464, 2022.

[43] Z. Ge, S. Liu, F. Wang, Z. Li, and J. Sun, "Yolox: Exceeding yolo series in 2021," *arXiv preprint arXiv:2107.08430*, 2021.

[44] Z. Zhang, X. Lu, G. Cao, Y. Yang, L. Jiao, and F. Liu, "Vit-yolo: Transformer-based yolo for object detection," in *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2021, pp. 2799–2808.

[45] C. Wang, A. Bochkovskiy, and H. Liao, "Scaled-yolov4: Scaling cross stage partial network," in *2021 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. Los Alamitos, CA, USA: IEEE Computer Society, jun 2021, pp. 13 024–13 033. [Online]. Available: https://doi.ieeecomputersociety.org/10.1109/CVPR46437.2021.01283

[46] Z. Lyu, "Dataset bias analysis on autonomous driving," 2018. [Online]. Available: https://cs230.stanford.edu/projects_spring_2018/reports/8289902.pdf

[47] F. Yu, H. Chen, X. Wang, W. Xian, Y. Chen, F. Liu, V. Madhavan, and T. Darrell, "Bdd100k: A diverse driving dataset for heterogeneous multitask learning," in *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, 2020, pp. 2636–2645.

[48] Q. Wang, H. Zhang, X. Hong, and Q. Zhou, "Small object detection based on modified fssd and model compression," *arXiv preprint arXiv:2108.10503*, 2021.

[49] P. Zhao, W. Niu, G. Yuan, Y. Cai, B. Ren, Y. Wang, and X. Lin, "Achieving real-time object detection on mobiledevices with neural pruning search," *arXiv preprint arXiv:2106.14943*, 2021.

[50] A. Milan, L. Leal-Taixé, I. Reid, S. Roth, and K. Schindler, "Mot16: A benchmark for multi-object tracking," *arXiv preprint arXiv:1603.00831*, 2016.

[51] R. Stiefelhagen, K. Bernardin, R. Bowers, J. Garofolo, D. Mostefa, and P. Soundararajan, "The clear 2006 evaluation," in *International evaluation workshop on classification of events, activities and relationships*. Springer, 2006, pp. 1–44.

[52] J. Luiten, A. Osep, P. Dendorfer, P. Torr, A. Geiger, L. Leal-Taixe, and B. Leibe, "Hota: A higher order metric for evaluating multi-object tracking," *International Journal of Computer Vision*, vol. 129, pp. 548–578, 9 2020. [Online]. Available: http://arxiv.org/abs/2009.07736http://dx.doi.org/10.1007/s11263-020-01375-2

[53] NVIDIA. (2022) Deepstream 6.0.1 release documentation. Acessed: 10/04/2022. [Online]. Available: https://docs.nvidia.com/metropolis/deepstream/dev-guide/text/DS_plugin_gst-nvtracker.html#nvdcf-tracker

[54] A. Lukežič, T. Vojíř, L. Zajc, J. Matas, and M. Kristan, "Discriminative correlation filter tracker with channel and spatial reliability," *International Journal of Computer Vision*, vol. 126, pp. 671–688, 2018. [Online]. Available: https://doi.org/10.1007/s11263-017-1061-3

[55] N. Wojke, A. Bewley, and D. Paulus, "Simple online and realtime tracking with a deep association metric," in *2017 IEEE international conference on image processing (ICIP)*. IEEE, 2017, pp. 3645–3649.

[56] Y. Zhang, P. Sun, Y. Jiang, D. Yu, Z. Yuan, P. Luo, W. Liu, and X. Wang, "Bytetrack: Multi-object tracking by associating every detection box," *arXiv preprint arXiv:2110.06864*, 2021.

[57] H. W. Kuhn, "The hungarian method for the assignment problem," *Naval Research Logistics Quarterly*, vol. 2, pp. 83–97, 3 1955. [Online]. Available: https://onlinelibrary.wiley.com/doi/full/10.1002/nav.3800020109https://onlinelibrary.wiley.com/doi/abs/10.1002/nav.3800020109https://onlinelibrary.wiley.com/doi/10.1002/nav.3800020109

[58] X. Zhou, V. Koltun, and P. Krähenbühl, "Tracking objects as points," in *European Conference on Computer Vision*. Springer, 2020, pp. 474–490. [Online]. Available: https://github.com/xingyizhou/CenterTrack.

[59] J. Cao, X. Weng, R. Khirodkar, J. Pang, and K. Kitani, "Observation-centric sort: Rethinking sort for robust multi-object tracking," *arXiv preprint arXiv:2203.14360*, 2022. [Online]. Available: https://github.com/noahcao/OC_SORT.

[60] Y. Du, Y. Song, B. Yang, and Y. Zhao, "Strongsort: Make deepsort great again," *arXiv preprint arXiv:2202.13514*, 2022.

[61] NVIDIA. Jetpack sdk 4.6 release page | nvidia developer. Acessed: 10/04/2022. [Online]. Available: https://developer.nvidia.com/jetpack-sdk-46

[62] e-con Systems. Four synchronized 4k cameras for nvidia® jetson agx xavier™. Acessed: 05/06/2022. [Online]. Available: https://www.e-consystems.com/nvidia-cameras/jetson-agx-xavier-cameras/four-synchronized-4k-cameras.asp

[63] paperswithcode. (2022) Paper implementations grouped by framework. Acessed: 09/06/2022. [Online]. Available: https://paperswithcode.com/trends

[64] S. Chirag. Byte-deepstream. Acessed: 02/06/2022. [Online]. Available: https://github.com/chirag4798/Byte-Deepstream

[65] A. H. Jonathon Luiten, "Trackeval," https://github.com/JonathonLuiten/TrackEval, 2020.

[66] D. Wu, M. Liao, W. Zhang, and X. Wang, "Yolop: You only look once for panoptic driving perception," *arXiv preprint arXiv:2108.11250*, 2021.

[67] IBM. What are neural networks? | ibm. Acessed: 11/02/2022. [Online]. Available: https://www.ibm.com/cloud/learn/neural-networks

[68] S. Kadam. Neural network part1: Inside a single neuron by shweta kadam analytics vidhya - medium. Acessed: 10/02/2022. [Online]. Available: https://medium.com/analytics-vidhya/neural-network-part1-inside-a-single-neuron-fee5e44f1e