Object Detection in Embedded Systems: A Study on Deep Learning Approaches

Alejandro Fernández Luces
Vision Systems Department
Centro Tecnolóxico de Automoción de Galicia
A Coruña, Spain
alefer3327@ctag.com, alejandro.fernandez.luces@udc.es

Margarita Sáez Tort (Director)
Vision Systems Department
Centro Tecnolóxico de Automoción de Galicia
Vigo, Spain
marsae@ctag.com

José Rouco Maseda (Director)
VARPA Research Group
Universidade da Coruña
A Coruña, Spain
jose.rouco@udc.es

***Abstract*—Object detection is still a challenge in embedded devices. In this work we perform an extensive research of the capabilities of a state of the art deep learning algorithm in several embedded devices. Firstly, we conduct an integral study on the available automotive datasets to select the most appropriate one for our work. Then, we also perform a research on useful deep learning models for our case and select one to use. After that we develop a framework to train the model, clean the dataset, and convert the model into different deep learning frameworks for deployment. Finally, we deploy the model into three embedded devices: *Google Coral*, *Intel Movidius* and *NVIDIA Jetson TX2*. We check the devices inference speed capabilities as well as their impact on the object detection performance. Results show difficulties with real time object detection in general, due to the challenging real scenario dataset we selected. On the other hand, we obtained promising inference speed performance on the *Jetson TX2*.**

***Index Terms*—Deep Learning, Machine Learning, Embedded Devices, Object Detection**

I. Introduction

A. Context

Computer vision is one of the most blooming research fields in recent years. A large amount of vision algorithms and techniques were developed recently thanks to the increase in computation power, as well of wider availability of cheaper computer hardware. Because of these technological advances, deep learning approaches, which are very demanding on computation power, could begin to be developed.

Particularly, one computer vision challenge highlights among all: object detection. The problem of object detection is a highly researched topic, because of the many applications it has in different industries. Many recent deep learning algorithms, models and datasets are centered around this problem.

Object detection is an image processing task where the objective is to find any instances of predetermined classes contained in images. The techniques for object detection are mainly carried out by extracting some features from the image, and, using these features, it is possible to infer where the objects are located in the image and to what class they belong. There are several approaches to this problem, but most of the state of the art methods consist on deep learning approaches, where a model is trained with a large amount of data and it learns to generalize for the classes given to it.
Another relevant topic for this work is edge devices. They are, in general, low resource devices meant to run software on-site, usually in places where you can not delegate the computation into a machine with better resources through an internet connection. In this sense, they are the opposite of cloud approaches, where the computation is made online. It is possible for edge devices to communicate with cloud services if needed, but the critical computation is performed locally. The features of these machines allow us to deploy software in environments where installing high end hardware with high power consumption is not possible and where reliable internet connection is not always available, such as vehicles.

The main drawback of these systems is that they usually impose computational constraints on the application, which can cause that the deployed algorithms do not run fast enough for some use cases.

To speed up deep learning algorithms in edge devices, some optimizations can be implemented. For example, one of the most important of these procedures is the quantization of neural network weights and activations to integer 8-bits [1], allowing also for more efficient computer operations. This optimization implements a trade-off between computational speed and object detection performance. Another optimization widely used is the precision reduction from floating point 32-bits to floating point 16-bits. Finally, another state of the art optimization for deep learning approaches is the network pruning, which consists on the removal of certain nodes from the network to reduce the number of operations, and consequently, improve inference times.

In the automotive context, edge devices are really important for local computation. An environment as critical as a vehicle on use can not rely on a proper connection to the internet. Some cases like passing through a tunnel or through an area with no GPRS coverage will result on the system failing. To solve this problem, edge devices must be used. The power constraints of these devices must also be taken into account, since it will condition which models can be deployed in vehicles.

Currently, in the car industry, there are several sets of cameras and processing units that are able to deploy software

on them. One of the most known sets are the ones provided by *NVIDIA*. Their latest product called *NVIDIA DRIVE Hyperion* [2] is an end to end modular development platform, which contains a processing unit and several cameras included. Another similar solution is provided by *Mobileye* [3], with a product called *SuperVision* that also provides 11 cameras with a processing unit.

Our work was developed in the context of automotive industry, so edge devices with low computational resources were used. The fact that we are working with these kind of devices, in a critical environment such as real time use while driving, produces some constraints over the development of the project that we must take into account to understand the decisions made along the work.

The main restraint of our problem is that the developed deep learning model must be capable of making real time inferences, which is established at a minimum of ten frames per second. Due to this limitation, it is not possible for us to choose the most precise models, we must seek for models that have a balance between precision and inference time.

Another condition is that the models must be able to run in different embedded devices, which will not have a high computing power to make the inferences, thus making the models run slower. Lastly, we must train our models in an automotive specific dataset, to improve detection of the desired objects, rather than selecting a generic dataset with many different objects that are not useful, and potentially could waste resources, in the environment of a vehicle.

One of the principal motivations for this work is the lack of studies of deep learning approaches deployed in the embedded devices we have. We needed to know how deep learning approaches deployed into our available devices would work, and which devices would be able to support real time use. We needed as well a platform for custom training of the models with custom datasets, and a flexible conversion framework to get the models deployed in the different devices.

B. Our project

Given the context, we can introduce the project, which is composed of several parts. At first, we conducted a research on the available automotive datasets, to select one dataset that adapts well to our necessities. Then, we analyzed several deep learning models papers to choose the one that was the most adjusted to our context and development environment. After the examination, we trained the model we choose from the deep learning research. Later, we made the conversions necessary to make the model compatible with the different embedded devices we used. Afterwards, we make optimizations on the models to try to have a better balance between the inference time and the precision metrics. Finally, we test the models on the different devices and optimizations and compare the results among them.

C. Objectives

The main targets of this project are:

- **Explore datasets and models** dedicated to automotive object detection.
- Develop a **training platform for deep learning** object detection and classification models.
- **Optimize the obtained models** for balancing between inference time and precision.
- **Compare the results** of the models and optimizations on the different embedded devices.

D. Contributions

This work produces a novel comparison among three similar embedded devices: *Google Coral*, *Intel Movidius* and *NVIDIA Jetson TX2*. This comparison serves as a starting point for helping decide which of these devices can be useful in which situations.

The code developed also serves as a suite for training deep learning models with *PyTorch* [4] and deploying the obtained model into the different frameworks, which is a toolset we have not been able to find as is in the current state of the art.

There is currently another similar work going on as we worked on this project, MMDeploy [5], which also works on the model deployment into different frameworks.

II. Work plan and organization

To reach the set objectives, we designed a methodology according to the characteristics of our work. The main steps of the project are the following:

1) **Perform an extensive research on the state of the art regarding object detection and classification for the automotive context**. This includes researching which datasets are being used, what problems are being faced in the automotive computer vision world and what approaches are used to resolve these problems. Also, it is necessary to research about deep learning neural networks used in embedded devices to select an appropriate model for our specific case.
2) **Choose an automotive dataset**. It is a requisite that the dataset must contain real case scenarios to properly approach the problem. Usually, these kind of datasets represent a challenge for deep learning models, due to the high complexity of the scenes. We must be conscious about this characteristics to better understand and analyze the results of our model.
3) **Select a deep learning object detection and classification model**. We must be careful to choose a model with real time capabilities, to deploy those models in the required embedded devices.
4) **Train the adopted model with the chosen dataset**. Check the results and tune the training parameters to optimize the final results. We should compare the results with the different parameters to select those with the best results. In this step we also should set the metrics baseline for further optimizations and to have a reference with respect to the embedded devices.
5) **Convert the best obtained model into the different representations** we will need to deploy the model in the embedded devices.
6) **Deploy the model** in the selected embedded devices.
7) **Compare inference test results in the different devices** to check which machines have capabilities for real time use.

III. Datasets

In this section we will explore the different options pondered when choosing a dataset for the project development. We considered several requirements to make this selection: a high number of classes, the scene diversity, the weather diversity, the number of annotations, the number of samples,

the day/night scenes, the richness of annotation information, the code support availability and the freshness.

In the following sections we will describe the main options considered, and its advantages and drawbacks for our problem. Other datasets such as *Robotcar* [6], *KAIST* [7], *ApolloScape* [8] and *Citiscapes* [9] were considered, but they were discarded for detailed study in later stages of the project due to them not meeting most of the established requirements.

It is important to note that most of these datasets come from an automotive context. These kinds of datasets usually contain different modalities of data annotations that are not strictly needed for our work. For example, LiDAR data, which are the annotations for 3 dimensional environments, is present in most of the reviewed datasets. These extra annotations are not crucial for resolving our problem, but it is desirable for future works with the dataset related with this project.

A. *nuImages*

The dataset provided by *Motional*, *nuScenes* [10], is a public large scale automotive dataset focused on autonomous driving. Its data contains the whole sensor suite a self-driving car would have in a real world situation, including LIDAR and camera information. The dataset was collected in two locations, Boston and Singapore (a left driving city) so it provides us images from complex driving situations. This collection not only contains scenes from urban locations, but also from residential areas, industrial zones and nature locations.

Another advantage of this dataset is that it contains data from sunny, rainy and cloudy weathers, as well as day and night scenes. This diverse scenery is the main reason why we chose this dataset.

The *nuImages* dataset contains two dimensional bounding boxes annotations for object detection and classification, as well as three dimensional bounding boxes for the same purpose but with LIDAR data. It also contains surface annotations for image segmentation problems. Some of these features, in spite of not being useful for the current project, were taken into account for the decision of selecting the dataset, because they allow to further extend the scope of the project in future works.

In our project, we will only be using a subsection of *nuScenes* called *nuImages*, which is the part of the dataset dedicated for object detection and classification in two dimensional data. This collection contains a total of 93.000 images. In figure 1, we expose the classes contained in this dataset with the number of annotations of each one.

We can see that one of the main problems present in this dataset is a severe imbalance among the classes. The ratio of the lowest labeled class to the most labeled class is 1:7145, which is really worrying for properly training and classifying with any machine learning approach. In table I we can see the number of samples for each class, where we can extract this ratio. In section III-F we will discuss the proposed solutions to the dataset imbalance.

Another problem of this dataset is the fact that it contains occluded annotations. Some of the two dimensional bounding box annotations do not fully contain the class it intends to annotate. For example, we observed some cars were highly occluded by vegetation, but annotators decided to label the car as if there were no vegetation covering it. In figure 2 we can observe an example of this. Some annotation guides

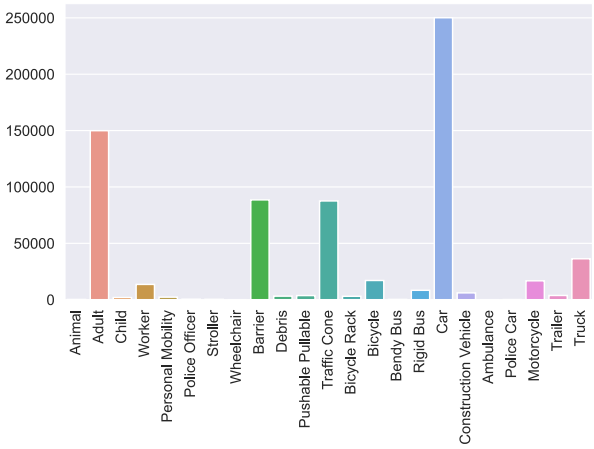

Fig. 1. Bar graph containing the number of samples in each class in the *nuImages* dataset.

TABLE I
DETAIL OF THE NUMBER OF ANNOTATIONS AND RATIO OVER TOTAL ANNOTATIONS OF EACH CLASS IN NUIMAGES DATASET

Class	Annotations	Ratio(%)
Animal	255	0.04%
Adult	149,921	21.61%
Child	1,934	0.28%
Worker	13,582	1.96%
Personal Mobility	2,281	0.33%
Police Officer	464	0.07%
Stroller	363	0.05%
Wheelchair	35	0.01%
Barrier	88,545	12.76%
Debris	3,171	0.46%
Pushable Pullable	3,675	0.53%
Traffic Cone	87,603	12.63%
Bicycle Rack	3,064	0.44%
Bicycle	17,060	2.46%
Bendy Bus	265	0.04%
Rigid Bus	8,361	1.21%
Car	250,088	36.05%
Construction Vehicle	6,071	0.88%
Ambulance	42	0.01%
Police Car	139	0.02%
Motorcycle	16,779	2.42%
Trailer	3,771	0.54%
Truck	36,314	5.23%
Total	693,783	100.00%

do not allow for these type of annotations, such as the one provided by *Waymo* [11].

Also, another downside of this dataset is that it comes pre-splitted into train, validation and test splits. This is explained because authors made sure that all splits contained similar proportions of each class, so training and testing is more accurate. But at the same time, it makes it hard to implement custom splits or make a k-fold approach on this dataset. Also, the test split is not annotated, because this data is designed for an algorithm competition. This causes that we can not use the test split for metrics calculation, leaving us only with the train and validation splits for implementing our model.

We decided to choose this dataset for our problem primarily because of the diversity of the scenes, particularly thanks to the diverse weather conditions and the day and night images. Also, another deciding factor was that the creators provide an API for easier image retrieval, which facilitates

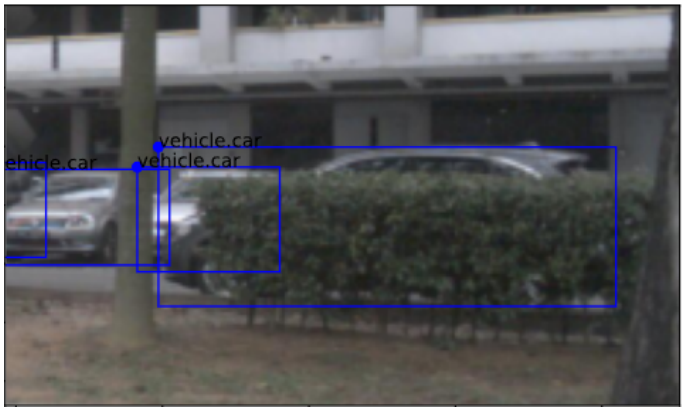

Fig. 2. Example of an occluded annotation in *nuImages*

Fig. 3. Image example from COCO dataset traffic image.

the usage of the dataset.

B. Waymo

Waymo is a company dedicated to the development of autonomous vehicles software. It released its homonymous dataset *Waymo Open Dataset* [12] on 2020.

The dataset contains two main sections: Motion and Perception. In the Perception section, which is the one useful for our project, there are 1000 scenes for training and 150 for testing, each scene containing 20 seconds of video. The scenes were recorded in six different cities from the United States. In contrast to *nuImages*, it does not contain an specific dataset for object detection and classification in two dimensional data, though it can be used for this purpose by separating the two dimensional annotations. Also, differently from *nuImages*, the dataset only consists of scenes, not separate images independent from each other. This is due to the fact that *Waymo* is mainly centered around the use of LiDAR data, being the two dimensional data used for support.

There is also a great diversity of scenes in *Waymo*. There are day and night scenes, as well as several different weather conditions.

The main downsides from this dataset that lead to its discardment for this project are the following:

1) It does not contain an specific dataset for two dimensional data, despite we can extract it from the main dataset.
2) The dataset only has 3 annotated classes: vehicle, cyclist and pedestrian. It could be enough for our project, but it lacks most of the classes that can be useful in driving situations.
3) The data is in *TensorFlow record* format, which complicates the processing of the images, since it is necessary to convert them into a *PyTorch* format.

C. Common Objects in Context

COCO [13] is the reference dataset for object detection and classification, as well as other purposes. It was released by *Microsoft* in 2014, but several releases of the dataset were made over the years. This dataset is one of the most used ones for object detection in general.

In its 2017 version, the dataset contains 123,287 images, with 886,284 annotations in total. It has 80 categories with segmentation and bounding box annotations. Most of the reviewed deep learning algorithms we will see in section IV use this dataset to train and test their implementations.

Of the reviewed datasets, this is the only one that is not from an automotive context, but rather is a generalist dataset containing several classes, including traffic classes like car, truck, pedestrian, etc. The fact that is not an automotive dataset played a key factor in discarding it for this project, but it could have been used for this work by selecting the automotive related labels only. In figure 3 we can see a traffic related annotated image.

D. Audi Autonomous Driving Dataset

The car company *Audi* released in 2020 a dataset called *Audi Autonomous Driving Dataset* [14].

This dataset contains 40,000 annotated frames with semantic segmentation and point cloud labels, recorded in the different cities. It does not contain two dimensional bounding box annotations, though they can be extracted from the segmentations. Also, it does not contain night scenes, which is not ideal for our case. These two factors, and also the fact that it does not come with a library to process the images, were decisive for discarding this dataset.

E. KITTI

The *KITTI* dataset [15] is a joint project from *Karlsruhe Institute of Technology* and *Toyota Technological Institute at Chicago*. This is the oldest reviewed dataset, released in 2013. As most of the reviewed datasets, it contains several kinds of annotations, as two and three dimensional bounding boxes, segmentation, tracking annotations, etc.

It is only recorded in one city, with only sunny and cloudy weathers, and only in daytime. Despite being a well established dataset, this lack of variety in the scenes leaded us to discard this dataset.

F. Dataset Manipulation Techniques

After selecting the *nuImages* dataset, we must apply some techniques to address the severe imbalance between the categories in the dataset. We applied the following techniques:

- **Oversampling**: we used an oversampling approach based on the rarity of the classes that appear in the image. For each class, we calculate the inverse of its number of appearances in the dataset, and we added this inverse for each item that appears in the frame to give this image a weight. Then, we pass the weights to the dataloader and it will favour the images with the highest weight.

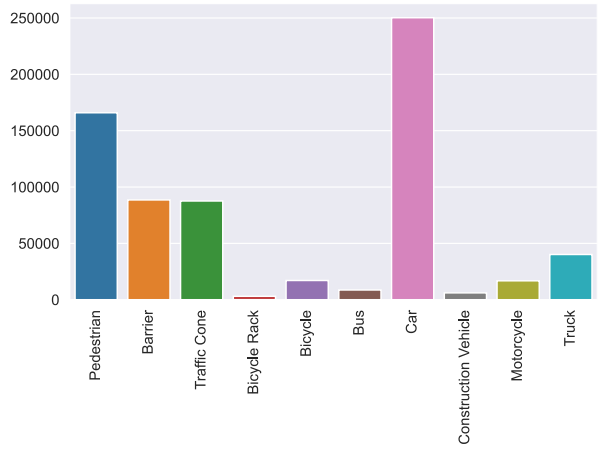

Fig. 4. Bar graph containing the number of samples in each class and the final classes after modifying the dataset.

- **Class removal**: after we have seen that class weighting was not enough, we proceeded to remove some of the classes with fewer samples. We removed the ego classes, which were not of interest for our problem, and the classes animal, personal mobility, stroller, and wheelchair, due to the low amount of annotations they contain.
- **Class joining**: some other classes, despite having few samples also, were kept, as we decided to join them into other similar classes. We joined adult, child, construction worker and police officer into pedestrian class, debris and pushable/pullable into obstacle class, bendy and rigid buses into bus, car and police car into car, and finally truck, ambulance and trailer into truck class. The obstacle class was finally removed, after reviewing the dataset and checking the wide heterogeneity of objects that were labeled as this class. In figure 4 we can see the final proportions of the classes, and in figure 5 the associations.
- **Tiny annotations**: some of the annotations were too small or too far away to be properly detected by the model. We decided to remove annotations with less than 2500 pixels of area in the original image resolution, 1600x900, since they were not of interest for our problem, and they caused trouble on the training and the testing of the model.

IV. Deep Learning Networks

In this section we will discuss the different deep learning networks we considered for this work. We will review the benefits and drawbacks of each of the considered networks and the reasons why they were selected or discarded.

We decided to stick with *YOLO* architectures because it was easier for the developed framework to train similar networks, but other paradigms were considered too.

A. YOLO Nano

YOLO Nano [16] is a deep learning network based on *YOLOv3* [17], but reduced in size to try to improve the real time use of the network. In figure 6 we can see the layers that compose this model.

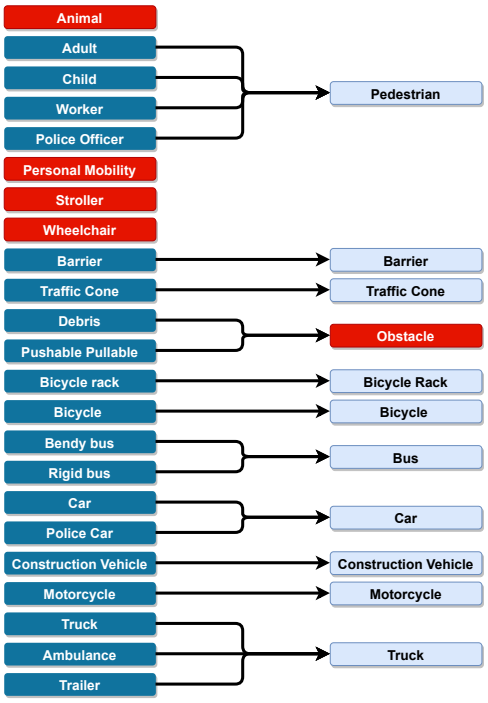

Fig. 5. Association between the original labels and the modified labels resulted from dataset fixes. In red, the removed classes. In light blue, the final classes used.

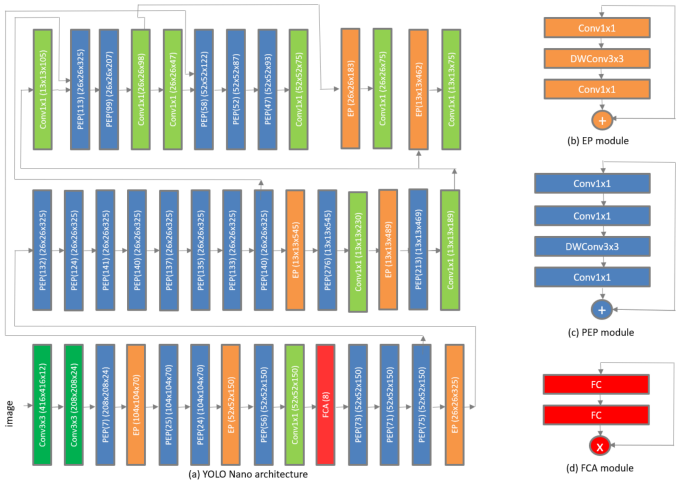

Fig. 6. Description of the layers of the YOLO Nano network.

YOLO Nano algorithm works by dividing the image into three differently sized grids. Each square of the grid is responsible for detecting any object contained inside of it. Each detection holds the following attributes: the center of the bounding box, its height and width, the predicted class and the confidence of the network. The adopted algorithm also uses non maximum suppression to select the most confident detection from a series of overlapping bounding boxes.

The main reason to choose this network over the others is that it fits our problem nicely: it is a network designed for embedded devices, which is where we will deploy our models, and also the original paper tests it in *NVIDIA Jetson AGX Xavier* hardware, which is nice for having a reference point to compare with our experiments.

In terms of detection metrics, it is not the best model to use, since it does not achieve the same mean average precision as other state of the art networks. Nonetheless, it

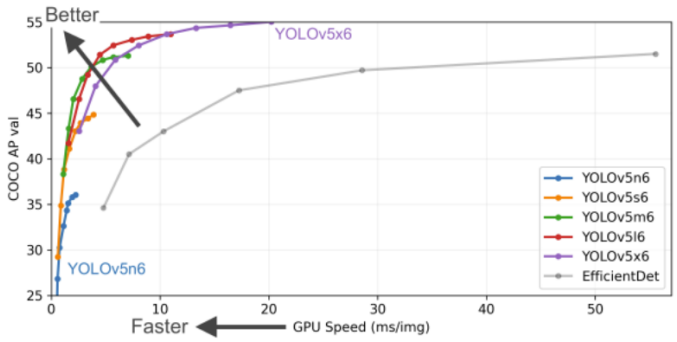

Fig. 7. Claim of the efficiency of the different YOLOv5 versions on COCO dataset.

has a good compromise between computational requirements and detection metrics, which is exactly what we are looking for. In the paper experiment they state that the model has 4.57B operations.

B. YOLOv4

The *YOLOv4* network [18] is a *YOLO* based network that we considered for this work. It is a state of the art network widely used in a diverse array of cases, particularly in cases where speed or real time use is necessary.

For our case, this network could be used, but it requires relatively high computational power for real time use, which is not ideal for this work. In general, where computation power is not a problem, the network works perfectly in real time. But we will be using devices with power constraints, so achieving real time use in embedded devices is not too feasible with this network, and so it was discarded.

C. YOLOv5

YOLOv5 [19] is a *YOLO* like architecture based on *YOLOv3*, released in 2020. It is a promising architecture that is included in *PyTorch* model repository. Authors claim very good results and adaptability to different computational capabilities, as we can see in figure 7.

Despite it looks very promising for our project, there is some drawbacks that made us discard this network.

We discovered that the calculation for the time to process an image is done in batches, instead of calculating the time for single images. This, together with the fact that there is no published paper for this network, leaded us to believe that the results of the network are not properly reviewed and contrasted, which was the main reason why we discarded this model.

Despite this controversy, the models seems to have gained reputation, as it is featured in the *PyTorch*, and the repository has a high number of contributors, issues, forks and activity in general.

D. Other Networks

There were several networks considered for its usage in this project, but due to time constrains they were not implemented. Despite not finding interesting to describe in detail all of the considered networks, we think it is necessary to discuss the considered paradigms that were not finally used, but can be utilized for future developments in this project line of work.

One of the considered paradigms is the few shot learning approach [20], where few data is used for training computer vision models. These methods rely on transfer learning, to seize the training from a similar problem and use a few samples to specialize the problem to a concrete case.

An older model called *MobileNet* [21] was also considered, because it follows a different architecture to *YOLO* while still having good results in real time.

V. Training Details

In this section we will describe the training techniques used for the model we finally selected. The original idea of the work was to train several neural networks to compare, but due to time limitations and large training times, we decided to train only one kind of neural network.

After some small trains to debug the training problems we encountered, we proceeded to use the following parameters for the training:

- **Number of threads**: to use for loading the data into the graphical processing unit memory. We used 4 despite the computer having more cores, due to stability issues.
- **Image size**: it is a really important parameter for the model, increasing its size improves detection, but also increases the processing time per image. It is a good parameter for our project, since we can use it to balance the detection performance and inference speed. We chose a relatively high image size of 608x608, due to the dataset containing many small annotations. With less resolution, too many annotations were too reduced to be detected by the model.
- **Image transformations**: we performed some transformations, apart from resizing the images. First of all, we padded the image with black pixels to make it have a square shape while preserving the original aspect ratio. We believe that this is important, since the network may learn to recognize deformed shapes that will not be valid in real cases. Also, for the training set, we implemented statistical data augmentation by randomly flipping the images horizontally with a probability of 50%. Other augmentations were not suitable for this problem, since they would make the driving scenes unrealistic.
- **Number of epochs**: we used 25 epochs to train the model. We observed that over 25 epochs, the model loss began to plateau, thus it was not improving anymore on the training set and potentially overfitting.
- **Validation interval**: we decided to carry out a validation epoch for every training epoch, so we have a better understanding of how the model is being trained, particularly to better look for overfitting.
- **Batch size**: we employed 12 images in each batch, which was the maximum allowed by the GPU we utilized.
- **Optimizer**: we used ADAM [22] optimizer. It is widely used in cases were the amount of data is very large. One of the best features of this optimizer is that it computes individual adaptive learning rates for different parameters, facilitating the proper training of the model with little tuning to the optimizer.
- **Learning rate**: we selected a learning rate of 0.001.
- **Confidence threshold**: minimum confidence the network has to have in a detection to consider it reliable. We set it at 0.8.
- **Non maximum suppression threshold**: this threshold serves for joining overlapping detections into the same

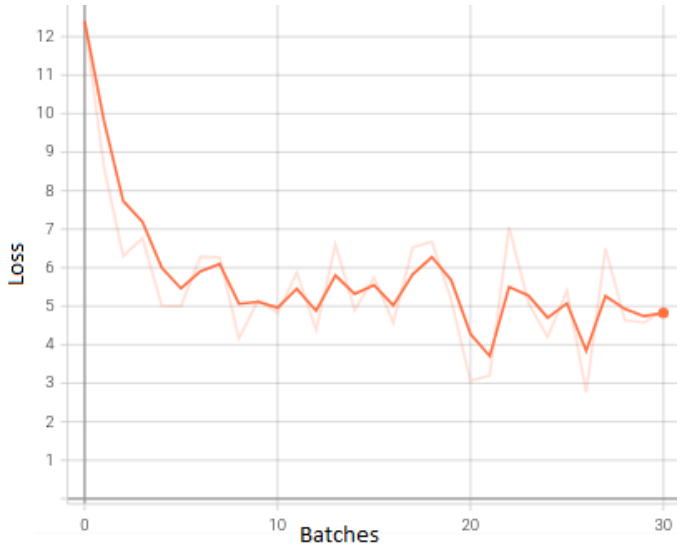

Fig. 8. Loss evolution during the training of the model.

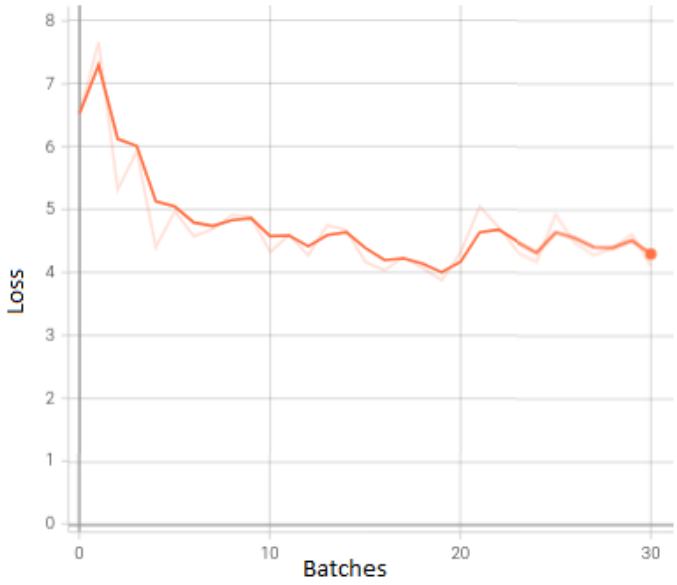

Fig. 9. Validation loss during the training of the model.

bounding box. The higher the threshold, more overlapping is allowed. We set this threshold at 0.5.

The model uses binary cross-entropy loss for the confidence and class of the prediction, and mean squared error for the bounding box predictions. Then, the final loss used for training is the sum of all the losses.

Dataset is split into train, validation and test set. Ideally, it would be better to train the network using the k-fold method [23], but due to the large amount of time needed for training deep learning models, it is impractical. Instead, we train on a fixed split, validating the model every epoch to ensure the model convergence.

Once the setup is complete, we proceeded to train the model. In figures 9 and 8 we can observe how the training and validation losses evolved, giving us an idea on how the model fits the data. The model took 43 hours and 48 minutes to train for 30 epochs in the desktop machine we describe in section VII-A.

VI. Development Framework

In this section we describe the software frameworks that we used during the development of the dissertation. It is important to discuss the effects on quality and performance of the different tools we use to produce the machine learning models as well as any support software we need.

On top of this, we must take into account that the model deployment we discuss in section VIII is produced in different devices, which entails the use of compatible frameworks for each machine. Due to this requirements, we will need the use of the following software and libraries.

A. Python

Python [24] is an interpreted, high-level programming language designed to be easy to use and readable by definition. It is the most popular language for deep learning thanks to its design philosophy. It allows for fast development, because of it easy syntax, has the automatic garbage collector that facilitates memory management and it is interpreted, which eases the testing of the code.

Thanks to these features, *Python* became popular, and now there is a lot of community support for the language, including a great variety of properly developed libraries, as well as the easiness to install *Python* libraries in any system.

Some of this libraries are *Tensorflow* and *PyTorch*, the two most important deep learning development libraries, which we will talk about in later sections.

Choosing *Python* as a programming language is a clear decision, since it has support for all of the features we need in deep learning, has great documentation and it is fast for development.

Another languages that could be selected were *C++* [25], which has good support and good documentation but is slower to develop, and *R* [26], but it is way less used for deep learning in comparison with *Python*, as well as having less programming features.

B. Tensorflow

Tensorflow [27] is a deep learning framework developed by *Google*, released in 2015. *Tensorflow* is one of the most established deep learning development frameworks to this date. It can be used in several programming languages, such as *Python*, *C++* and *Java*. In comparison to the other considered framework, *PyTorch*, which we will be discussing in section VI-C, they are very similar to each other. *Tensorflow* code writing is a bit more difficult than *PyTorch*, which makes it less ideal for prototyping and fast coding. Lack of experience in this framework also played a role in the decision to choose *PyTorch* over this one. In advanced steps of the development of the project, we discovered that *Tensorflow* has native support to some of the embedded devices we were using. It has, in general, better compatibility when converting the models into other formats. Part of this is caused by the way *PyTorch* generates its computational graphs. We will expand on this in section VI-C.

For example, *Tensorflow* can import models from *ONNX*, an open format for neural network persistence we discuss in section VIII-A, while in *PyTorch*, this feature is not implemented as of today.

Tensorflow also has a type of model representation called *Tensorflow Lite*, which is designed for cell phones, IoT and low resource computers in general. We will need this

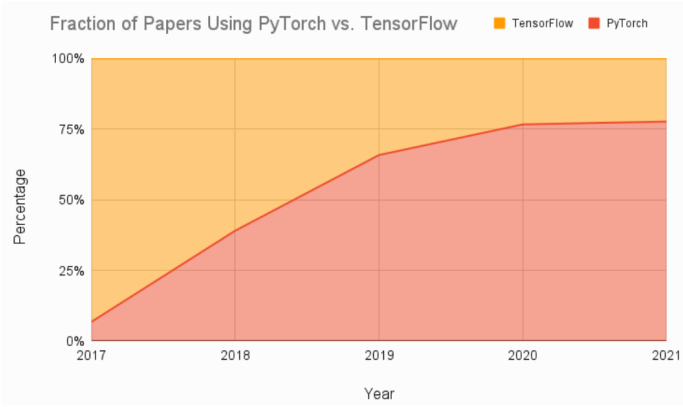

Fig. 10. Evolution of the use of *Tensorflow* and *PyTorch* in research papers from 2017 (release of *PyTorch*) to 2022 [28]

representation to use with an embedded device, *Google Coral*, which we will discuss in section VII-C. To be more precise, *Google Coral* requires the usage of *TensorFlow Lite* model to operate. Thankfully, we can convert from *ONNX* to this representation, so we did not have to reimplement the network in *Tensorflow* to achieve this representation. Also, we used *TensorBoard*, a component of *TensorFlow*, to check on the training results of the different experiments.

C. PyTorch

PyTorch [4] is a deep learning library developed by *Facebook*'s AI research lab, released in 2017 to open source. It is the main competition to *Tensorflow*, gaining popularity in research fields as we can see in figure 10. As previously stated, *PyTorch* is easier to use by the developer of the project, mainly because it is a more *pythonic* framework, so it is more comfortable to use for *Python* developers. Since both frameworks have very similar functionalities, we chose *PyTorch* because of the familiarity we are mentioning.

Nonetheless, *PyTorch* has its downsides, which we discovered as the project went on. The main issue we found is the fact that *PyTorch* has, in general, poor compatibility for deployment in the embedded devices we used.

The framework allows to export the generated models to *ONNX* format, but with some limitations. For example, if the model contains dynamic shapes, which are determined in runtime, the output *ONNX* model can not be transformed to some of the model representations we discuss in section VIII. The model can be transformed to be static through the use of the *JIT* module, an optimized compiler for *PyTorch* programs, but it is a highly difficult process which wasn't fully explored due to time constraints. Another downside is that the import of models in *ONNX* format to *PyTorch* is not yet implemented, so any transformation we want to perform in the *ONNX* model, can not be reimported to test it with the *PyTorch* framework, it can only be deployed using *ONNXRuntime*, a library that allows to make inference with a *ONNX* model representation.

Another question is the *PyTorch* quantization. This library allows us to quantize the models following different approaches. But once we obtained the quantized model, we can not export it to *ONNX* format, because this was not properly implemented in this framework. This makes quantization in *PyTorch* a bit useless, since it can not be deployed the optimized model to many devices.

In general, we can observe that *PyTorch* is a great framework for research and quick development of deep learning models, which is why we originally selected this library. But this library is not ideal for later steps like model deployment and optimization, which is a crucial step in our project. Unfortunately, we discovered these mentioned issues in the late steps of the project development, so changing the framework was not an option.

The steps we have taken to mitigate these issues will be discussed in the section VIII.

VII. Embedded Devices

In this section we will describe the different embedded devices we used to test the model, as well as the computer where the training was done. We will detail the specifications of each device and any particular feature that is of interest for this project.

A. Desktop Computer

The main computer, where the trainings were made, is a commercial desktop computer with the following hardware specification:

- **GPU**:NVIDIA GeForce RTX 3080ti, with 12gb memory. Power consumption 400W
- **CPU**:Intel(R) Core(TM) i7-6850K CPU @ 3.60GHz. Power consumption 140W
- **Memory**: 16Gb RAM Kingston DDR4 2133 MHz.

Apart from training, we will use this machine for testing the model and use it as a baseline for the results on the rest of the devices.

B. NVIDIA Jetson TX2

The *NVIDIA Jetson TX2* [29] is an embedded AI computing device released by *NVIDIA* in 2017. it is built an *NVIDIA Pascal* GPU with 8Gb of memory. It has the following hardware specifications:

- **GPU**: 256-core NVIDIA Pascal™ GPU architecture with 256 NVIDIA CUDA cores
- **CPU**: Dual-Core NVIDIA Denver 2 64-Bit CPU and Quad-Core ARM® Cortex®-A57 MPCore.
- **Memory**: 8GB 128-bit LPDDR4 Memory 1866 MHz - 59.7 GB/s
- **Power**: 15W

This device specifications are fitting for our project, since it is designed precisely for deploying deep learning models in it.

Despite this, the hardware is somewhat old, and it is not being updated anymore, which causes some issues for the installation of the necessary software in it. The operating system for the device must be installed through a host machine, which has to have installed an *Ubuntu* operating system. Installing it through a virtual machine causes connectivity problems that leads to errors during the installation. Also, some of the libraries versions installed by default are outdated, so we must carefully install the libraries needed to run the project without updating any library that would be incompatible with the hardware.

It is worth noting that this device has four power modes that allow to adjust the consumption of the device. In table II we can see a summary of the features of each mode.

TABLE II
POWER MODES OF *Jetson TX2*.

	MAXN	MAXQ	MAXP	MAXP*
Power Budget	n/a	7.5W	15W	15W
D15 CPU	2	0	2	0
A57 CPU max freq. (MHz)	2000	1200	1400	2000
D15 CPU max freq. (MHz)	2000	n/a	1400	n/a
GPU max freq. (MHz)	1300	850	1122	1122
Memory max freq. (MHz)	1866	1331	1600	1600

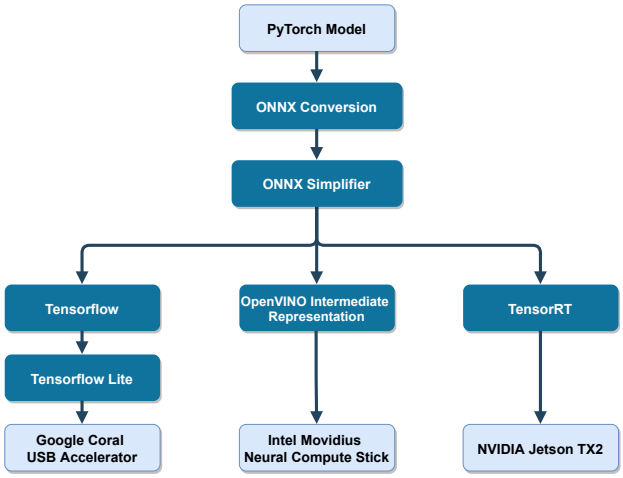

Fig. 11. Flow chart of the conversions that were done for the embedded devices.

C. Google Coral

The *Google Coral USB accelerator* [30] is an USB device that provides an edge TPU as a co-processor. It accelerates inference for machine learning models when attached to a host computer. it is capable of performing 4 trillion operations per second using 2 watts of power.

In our case, the model we developed is too demanding for this device, but we deployed the model in it to test the capabilities of the device.

D. Intel Movidius Neural Compute Stick

The *Intel Movidius Neural Compute Stick* [31] is an USB plug and play AI device designed for deep learning inference at the edge. It is based on the *Intel Movidius Myriad 2* visual processing unit. It has a clock speed of 933MHz and 4 Gb of memory. It can draw up to 1.5 watts while being able to perform 4 trillion operations per second.

This device is very similar to *Google Coral*, and it suffers from the same drawback: it does not have enough power for real time usage of our model.

VIII. DEPLOYMENT INTO PRODUCTION

Once the model was trained and the devices were ready to use, we proceeded to deploy the model into the different hardware we have available. This was a challenging step of the project, since each embedded device works with a different framework, hindering the deployment. We need to implement a series of conversions we will describe in this section, as well as the optimizations that could be done to the model to perform faster.

In figure 11 we can see a summary of the conversion process for the different devices

A. ONNX

Open Neural Network Exchange [32] is an open format to represent machine learning models. It defines a common set of operators and a common file format to enable to convert models into different frameworks. The main advantage of using *ONNX* is the interoperability between the frameworks we need to use in the different devices.

After converting our model from *PyTorch* to *ONNX*, we found that for some conversions it did not work out of the box, because some operators were not supported by the target frameworks. We decided to use a tool called *ONNX simplifier* [33] which changed the operators into compatible ones, as well as reducing the number of operations that compose the model.

The conversion to *ONNX* is a core task for this project. Without this model representation, we would not be able to deploy the model in any platform.

B. TensorFlow Lite

After converting the model into *ONNX*, we proceeded to convert it into *Tensorflow* model representation, and to *Tensorflow Lite* afterwards. *Tensorflow Lite* is a model representation included in the *Tensorflow* library designed to convert and optimize *Tensorflow* models for mobile and edge devices. It supports a wide variety of embedded environments, such as android, embedded Linux and IOS.

We used this representation to deploy the model into the *Google Coral* device.

C. OpenVINO Intermediate Representation

OpenVINO is the deep learning inference library developed by *Intel* to deploy machine learning models into *Intel* devices. It allows to convert models from *ONNX* into their representation, called *intermediate representation*. This representation consists of two files: a XML file which contains the network topology, and a binary file which will contain big values, like the weights.

This representation is required to deploy the trained model to the *Intel Movidius* device.

D. TensorRT

NVIDIA's TensorRT is a SDK built on top of CUDA designed for deep learning optimizations. It generates optimized runtime engines that are deployable in compatible hardware. Since *NVIDIA* GPUs are widely used, it is a very well established framework, which not only allows us to deploy the developed model into the *NVIDIA Jetson TX2*, but also helps us optimize the model for the desktop computer, which uses an *NVIDIA* GPU also. Of all the conversion frameworks we are using, this is the one that has the best compatibility with other technologies.

E. Performance Optimizations

Since the network is going to be deployed in machines with very low resources, it is a good idea to make some optimizations in the model to reduce its computational cost as much as possible. Sadly, since each framework has its own optimization implementations, it was hard to conduct an optimization that would be equal for all devices. There are several optimizations that can be done to improve the model inference speed. We considered the following optimization approaches:

- **Network pruning** [34] consists on reducing the size of the neural network by removing parameters from it. This process results in a model with similar accuracy, but faster inference speed. Due to time constraints, it was not possible to delve into this option.
- **Operations quantization** [35] is a reduction of the numerical precision of the operations in the neural network. Generally, models use 32 bits floating point operations. By reducing the precision to smaller types, such as 16 bits floating point operations or 8 bit integer operations, you can improve the speed of the model while keeping similar accuracy. This optimization can be done pre and post training, but if done post training it is required to calibrate the quantizer with a representative subset of the dataset. There is also a similar approach to post training quantization called dynamic quantization that obtains the calibration based on the data observed at runtime.
- **Weights quantization** reduces the numerical precision of the weights contained in the model. It helps to compress the model, but its improvement on inference speed is marginal, since operations precision are not optimized. This operation is commonly done after training the neural network.

Initially, the established plan was to quantize the operations in the model post training into 8 bit integer precision by using the *PyTorch* framework. Later, we discovered that the optimized *PyTorch* models had no support for transforming them into *ONNX* format, and *PyTorch* models could not be directly deployed anywhere in our project except for the desktop machine, so this optimization was discarded.

We tried to carry out an 8 bit integer precision dynamic quantization over the operations in the *ONNX* model, but the resulting model was not compatible for converting into the desired frameworks, so this had to be discarded due to the incompatibilities.

Later, we achieved 8 bit integer precision quantization for the weights in the *ONNX* model, which could reduce the size of the model, but it would not make much difference with respect to the speed, since the operations were still in 32 bits floating point.

In *TensorRT* framework, we tried to implement the 8 bit integer operations quantization, by using a small section of 100 images of the dataset to let the quantizer know the minimum and maximum possible values of the dataset and so it would be able to transform the 32 bits floating point operations. Sadly, this part of the framework is poorly documented and we did not have enough time to properly develop this part of the project.

We optimized the model to 16 bits floating point in *TensorRT* for the inference in *NVIDIA Jetson TX2*.

The operation quantization of the model to 8 bit integer precision would have improved vastly the speed of the model, particularly in the *Google Coral* and *Intel Movidius* devices.

IX. Experiments

In this section we will explain how we compared the results from each framework and device, in a way that we have a fair comparison between the different embedded devices.

Firstly, to check the performance of the model in the different contexts, we selected the following metrics:

TABLE III
Object detection and classification metrics obtained by the model.

Precision	43.20%
Recall	24.98%
F1	30.30%
mAP	19.25%

- **Precision** indicates the capability of the model to not label a sample as positive when the sample is negative.

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives} \quad (1)$$

- **Recall** gives us the ability of the model to find the positive samples.

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives} \quad (2)$$

- **F1** is the harmonic mean between the precision and the recall, giving us a more general metric for the model.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \quad (3)$$

- **Mean average precision** is the mean of the area under the precision-recall curve of each class. It tells us how well the bounding boxes are adjusted in comparison to the annotations in the dataset. We will get the general mean average precision as well as the mean average precision for each class. Note that for a perfectly fitted bounding box, average precision can be 0 if the label of the box is incorrect. E.g. a perfectly fitted truck labeled as car will have 0 average precision.
- **Frames per second** is the most important metric in our context. We need an average of more than 10 frames per second of performance in the devices to be available to use them in real time scenarios.

The metrics we will be giving more importance to are the frames per second and the average precision of the different classes.

Then, to obtain these metrics, we developed a testing framework which will use the test data from the dataset and make inferences with a batch size of 1 over all of the frames. This is done to have a realistic view on how the model would perform on a real case scenario. In this process we extracted the detection metrics and the frames per second in each framework and device, as well as comparing the different optimizations we were able to implement.

X. Results

After discussing how the experiments were made to obtain the metrics we considered, we can expose and discuss the results we obtained in our work.

A. Baseline results

To begin discussing the results, it is interesting to know how the model performs on the baseline machine we used to train the selected architecture. In table III we can observe the global metrics our network obtained in the testing split. We see that the metrics are oddly low, taking into account the large amount of data available and the use of a widely used architecture.

After observing these poor results, we took a look into the results per class to further analyze the causes. In table IV

TABLE IV
MEAN AVERAGE PRECISION OBTAINED IN EACH OF THE CLASSES.

Class	mAP
Pedestrian	42.05%
Barrier	34.61%
Traffic Cone	46.85%
Bicycle Rack	0.39%
Bycicle	12.18%
Bus	0.00%
Car	40.27%
Construction Vehicle	0.00%
Motorcycle	0.28%
truck	15.87%

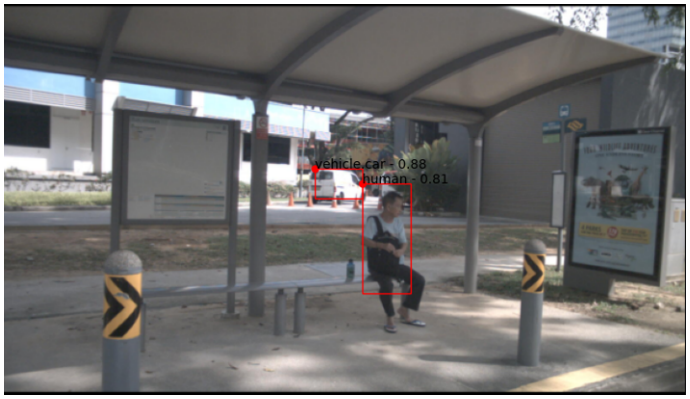

Fig. 12. Inference from a frame with a sunny weather.

we can observe the mean average precision in each of the classes. The most relevant data we can extract from the table is that in the most relevant classes, which are pedestrian and car, the model obtains acceptable results. We can also observe that there are some classes with poor performance, and some that get no detections at all. We think that this is caused due to the poor class representation in the dataset, as we will experiment in the following section.

In figures 12 and 13 we observe some examples of inferences the model can achieve. We can observe that in the most challenging situations, the model struggles to infer properly. These complicated cases can be finely adjusted to obtain better performance on object detection.

B. Impact of the dataset on the model

We hypothesized that the number of samples in each class impacted on the model detection metrics. In order to test this hypothesis, we implemented an scatter plot in figure 14 of the

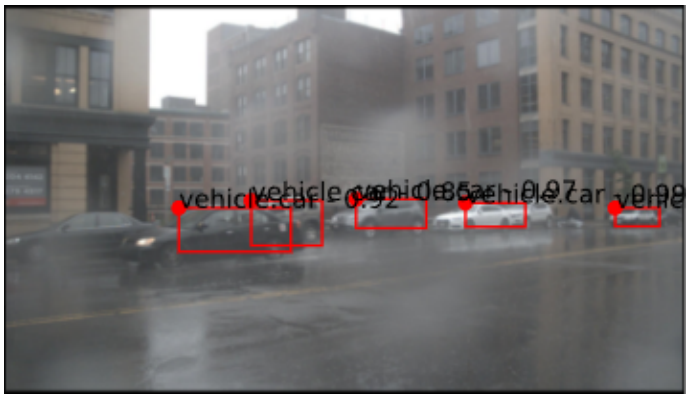

Fig. 13. Inference from a frame with a rainy weather.

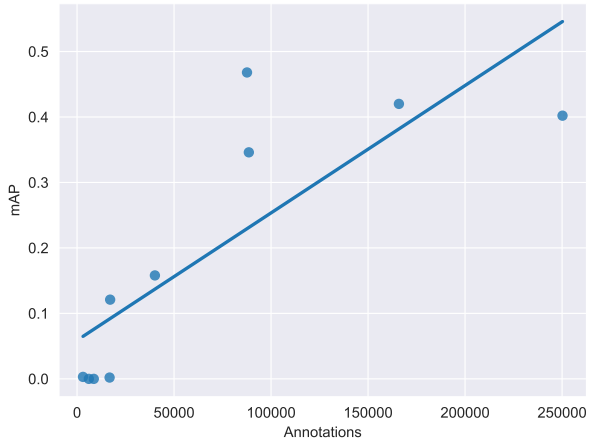

Fig. 14. Scatter plot with the number of annotations and the achieved mean average precision in each class.

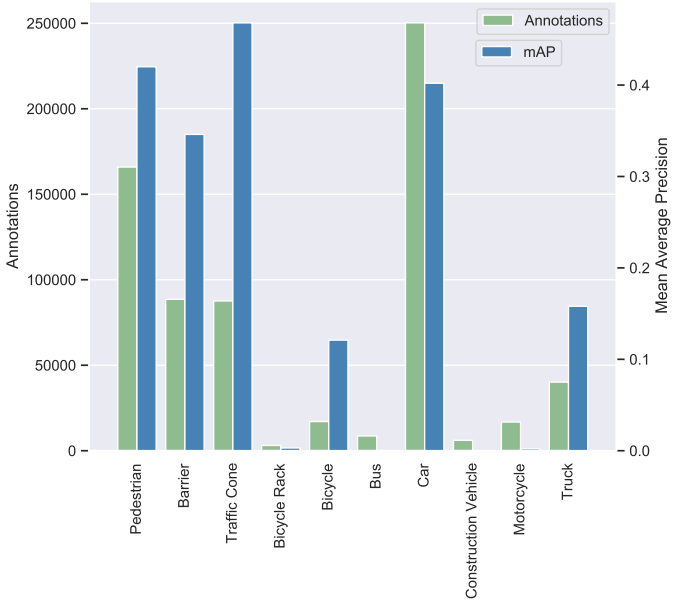

Fig. 15. Relation between the amount of annotations per class and the mean average precision per class.

number of samples on each class against its performance. We also implemented a bar plot in figure 15 with the same data to have a better view of each class. As we can observe there is a heavy correlation between the number of annotations and the mean average precision obtained in each class. We also observe that some classes obtain no detections, for example, the bus and the construction vehicle classes. We think that this is caused due not only to the few amount of samples, but also to the large variability that these classes can present. We observe that the efforts to solve the class imbalance were not enough, due to the severity of it.

C. Impact of the embedded devices on the detection

We also wanted to test how the conversions to the different frameworks and the deployments into the devices affected the detection and classification metrics. Table V shows the mean average precision of the models in each of the frameworks

TABLE V
SUMMARY OF THE MEAN AVERAGE PRECISION RESULTS IN EACH FRAMEWORK AND DEVICE.

Framework	Device	mAP
PyTorch	Desktop GPU	19.25%
TFLite	Coral	17.21%
OpenVINO	Movidius + Desktop CPU	17.20%
OpenVINO	Movidius	15.88%
TensorRT	Desktop GPU	17.21%
TensorRT	Jetson TX2	17.15%

TABLE VI
SUMMARY OF THE FRAMES PER SECOND ACHIEVED WITH EACH DEVICE AND RELEVANT OPTIMIZATION.

Framework	Device	FPS
PyTorch (baseline)	Desktop GPU	49.54 $\pm$2.46
TensorRT	Desktop GPU	82.04 $\pm$7.59
TFLite	Coral	1.77 $\pm$0.04
OpenVINO	Movidius + Desktop CPU	15.40 $\pm$1.02
OpenVINO	Movidius	3.37 $\pm$0.02
TensorRT	Jetson TX2 15W	10.07 $\pm$0.76
TensorRT	Jetson TX2 uncapped	10.41 $\pm$0.72

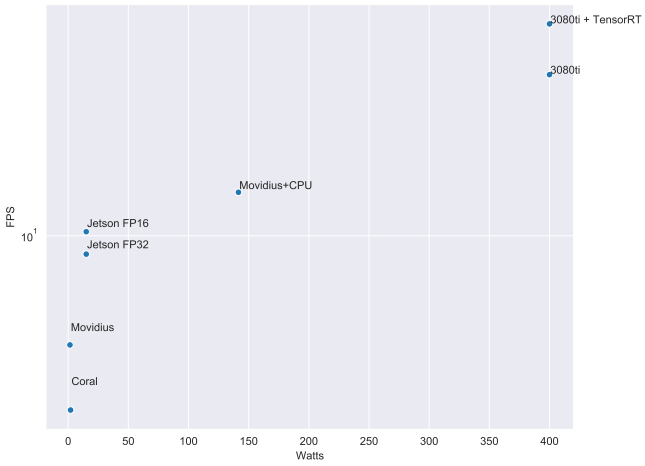

Fig. 16. Scatter plot relating the frames per second achieved to the maximum power the devices can draw. Y axis in logarithmic scale for better visualization

and devices. To summarize the results, the table V only presents the data with the described optimizations already applied to achieve the most frames per second. Note that a 2% decrease in mean average precision happens in general across all conversions and devices, except for the inference using only the *Intel Movidius* devices, which decreases the mAP by 3.4%. It was expected that the conversions would take a toll on the model detection metrics, since it is difficult for the libraries to exactly translate operations from one framework to another one.

D. Systems Throughput

The frames per second achieved in the embedded devices were more or less what we were expecting from the computational power specifications of the devices. Table VI contains the results obtained in terms of frames per second. The *Intel Movidius* and *Google Coral* devices achieved a very low performance, which was anticipated, due to its very low computational power. On the other side, *NVIDIA Jetson TX2* passed the barrier of 10 frames per second, which is the minimum value considered for real time usage.

It is relevant to see how the conversion from the *PyTorch* framework to *TensorRT* implied optimization over the model, improving the frames per second achieved by 57.96%.

Also, it is important to note the little difference between the tested power modes in *NVIDIA Jetson TX2*. We expected some improvement with the MAXN power mode, since it has no power consumption limit, but it seemed that it could not draw much more power than the 15W limited mode.

E. Power consumption analysis

We also found interesting to compare the power consumption against achieved frames per second in each device. In figure 16, the obtained data shows this relation.

We observe that there is a clear relation between the amount of power drawn by a device and the speed performance it can achieve.

XI. CONCLUSIONS

In this project we performed an extensive study on the state of the art of object detection and classification for the automotive context. We reviewed the datasets that different companies and research groups have available online. Then, we selected the dataset we believed fitted the best our problem, taking into account several features such as the variety of scenes and the amount of classes. After that, we researched some deep learning approaches that could fit the embedded devices context. We selected *YOLO Nano* due to its use on *Jetson* devices on the original paper [16], which fitted perfectly our study case. After that, we proceeded to preprocess the dataset and train the model with it. We converted the model into different frameworks so we would be able to deploy it into the embedded devices we selected.

Results show inadequate detection performance for our problem. The model did not work as we expected in terms of object detection capabilities. We believe this was partly due to the characteristics of the dataset, which had a severe imbalance of the classes, as well as some occluded annotations that could potentially harm the final result. Anyway, the model achieves acceptable performance in the most important classes, which are pedestrian and car.

In terms of speed, we obtained the expected results. Very low resource devices have not achieved real time inference. It maybe possible that a model quantized to 8 bit integer precision could have achieved a real time performance in these devices, but we were not able to setup this configuration, which may be regarded as a limitation of the study. The *Jetson TX2* has the right speed performance to achieve real time inference, which was also expected, as it is a significantly more powerful device that *Google Coral* and *Intel Movidius*.

The proposed work was very ambitious for the planned time, so we have left many work threads that can be continued in future works. First of all, different detection paradigms can be used to improve the detection results. Single shot learning, zero shot learning and transfer learning approaches could significantly improve the time spent on model training. Also, other similar problems could be checked too, for example, semantic and instance segmentation.

Another line of future work would be to test other datasets. The chosen datasets had many unforeseen problems, which partly worsened the detection results.

For future developments of this project, detection speed could also be tuned by modifying the image size. Since we chose a high image size, we understand that the processing of the image is costly. A smaller image size would worsen the detection metrics, but could significantly improve the speed of the inference.

Finally, the quantization of the network to 8 bit integer precision is the most important line of work that was left out. We believe that this modification could significantly improve the speed of the algorithm in the low resource devices, since they are designed to work the best with this precision.

Acknowledgment

Firstly, I would like to thank Margarita Sáez Tort for proposing the project and giving me the opportunity of doing this work with her team. I would like to earnestly acknowledge the sincere efforts and valuable time given by my colleagues at *CTAG Vision* team, specially to Alexandre Rodríguez Rendo and Francisco Parada Loira, who provided insight and expertise that greatly assisted this project. Without their help this project would be impossible.

I would also like to thank José Rouco Maseda for accepting the project and for the comments that greatly improved the manuscript.

I am grateful to my friends and family, whose love and guidance support me no matter what I pursue.

Most importantly, I wish to thank my loving and supportive partner, Clara, who accompanied me through this project.

References

[1] Manu Mathew, Kumar Desappan, Pramod Kumar Swami, and Soyeb Nagori. Sparse, quantized, full frame cnn for low power embedded devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 11–19, 2017.

[2] NVIDIA. Nvidia drive web page. https://developer.nvidia.com/drive, 2022.

[3] MobilEYE. Mobileye web page. https://www.mobileye.com/, 2022.

[4] Adam Paszke, Sam Gross, Massa, et al. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019.

[5] MMDeploy Contributors. OpenMMLab's Model deployment toolbox, 12 2021.

[6] Will Maddern, Geoffrey Pascoe, Chris Linegar, and Paul Newman. 1 year, 1000 km: The oxford robotcar dataset. *The International Journal of Robotics Research*, 36(1):3–15, 2017.

[7] Yukyung Choi, Namil Kim, Soonmin Hwang, Kibaek Park, Jae Shin Yoon, Kyounghwan An, and In So Kweon. Kaist multi-spectral day/night data set for autonomous and assisted driving. *IEEE Transactions on Intelligent Transportation Systems*, 19(3):934–948, 2018.

[8] Xinyu Huang, Xinjing Cheng, Qichuan Geng, Binbin Cao, Dingfu Zhou, Peng Wang, Yuanqing Lin, and Ruigang Yang. The apolloscape dataset for autonomous driving. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 954–960, 2018.

[9] Marius Cordts, Mohamed Omran, Sebastian Ramos, Timo Scharwächter, Markus Enzweiler, Rodrigo Benenson, Uwe Franke, Stefan Roth, and Bernt Schiele. The cityscapes dataset. In *CVPR Workshop on the Future of Datasets in Vision*, volume 2, 2015.

[10] Holger Caesar, Varun Bankiti, Alex H Lang, Sourabh Vora, Venice Erin Liong, Qiang Xu, Anush Krishnan, Yu Pan, Giancarlo Baldan, and Oscar Beijbom. nuscenes: A multimodal dataset for autonomous driving. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 11621–11631, 2020.

[11] Waymo. Waymo open dataset labeling specifications. https://github.com/waymo-research/waymo-open-dataset/blob/master/docs/labeling_specifications.md, 2022.

[12] Pei Sun, Henrik Kretzschmar, Xerxes Dotiwalla, Aurelien Chouard, Vijaysai Patnaik, Paul Tsui, James Guo, Yin Zhou, Yuning Chai, Benjamin Caine, et al. Scalability in perception for autonomous driving: Waymo open dataset. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 2446–2454, 2020.

[13] Tsung-Yi Lin, Michael Maire, Serge Belongie, James Hays, Pietro Perona, Deva Ramanan, Piotr Dollár, and C Lawrence Zitnick. Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer, 2014.

[14] Jakob Geyer, Yohannes Kassahun, Mentar Mahmudi, Xavier Ricou, Rupesh Durgesh, Andrew S Chung, Lorenz Hauswald, Viet Hoang Pham, Maximilian Mühlegg, Sebastian Dorn, et al. A2d2: Audi autonomous driving dataset. *arXiv preprint arXiv:2004.06320*, 2020.

[15] Andreas Geiger, Philip Lenz, Christoph Stiller, and Raquel Urtasun. Vision meets robotics: The kitti dataset. *The International Journal of Robotics Research*, 32(11):1231–1237, 2013.

[16] Alexander Wong, Mahmoud Famuori, Mohammad Javad Shafiee, Francis Li, Brendan Chwyl, and Jonathan Chung. Yolo nano: a highly compact you only look once convolutional neural network for object detection. In *2019 Fifth Workshop on Energy Efficient Machine Learning and Cognitive Computing-NeurIPS Edition (EMC2-NIPS)*, pages 22–25. IEEE, 2019.

[17] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767*, 2018.

[18] Alexey Bochkovskiy, Chien-Yao Wang, and Hong-Yuan Mark Liao. Yolov4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[19] ultralytics. Yolov5. https://github.com/ultralytics/yolov5, 2020.

[20] Yaqing Wang and Quanming Yao. Few-shot learning: A survey. 2019.

[21] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.

[22] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[23] Davide Anguita, Luca Ghelardoni, Alessandro Ghio, Luca Oneto, and Sandro Ridella. The 'k'in k-fold cross validation. In *20th European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning (ESANN)*, pages 441–446. i6doc. com publ, 2012.

[24] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[25] ISO/IEC. Programming languages — c++. Draft International Standard N4660, March 2017.

[26] R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria, 2021.

[27] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, et al. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[28] Ryan O'Connor. Pytorch vs tensorflow in 2022, Dec 2021.

[29] NVIDIA. Nvidia jetson tx2 specification. https://developer.nvidia.com/embedded/jetson-tx2, 2020.

[30] Google. Google coral usb accelerator datasheet. https://coral.ai/docs/accelerator/datasheet/, 2019.

[31] Intel. Intel movidius neural compute stick. https://www.intel.es/content/www/es/es/products/sku/125743/intel-movidius-neural-compute-stick/specifications.html, 2019.

[32] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[33] Daquexian et al. Onnx simplifier. https://github.com/daquexian/onnx-simplifier, 2019.

[34] Davis Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John Guttag. What is the state of neural network pruning? *Proceedings of machine learning and systems*, 2:129–146, 2020.

[35] Markus Nagel, Marios Fournarakis, Rana Ali Amjad, Yelysei Bondarenko, Mart van Baalen, and Tijmen Blankevoort. A white paper on neural network quantization. *arXiv preprint arXiv:2106.08295*, 2021.