POLITECNICO DI TORINO

Master degree course in Computer Engineering

Master Degree Thesis

# Vehicles detection and tracking using Convolutional Neural Networks on edge-computing platforms


**Supervisors**
prof. Bartolomeo Montrucchio

**Candidates**
Alessandro Marchetti
matricola: 242374

**Internship Tutor**
**CSP Innovazione nelle ICT**
ing. Ferdinando Ricchiuti

Academic year 2018 – 2019

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

Image classification and detection tasks have been approached in numerous ways throughout the years. Traditional approaches used conventional machine learning techniques dealing with hand-crafted vectors of features which had to be carefully extracted from the images with considerable domain-expertise. Deep learning algorithms have overcome this limitation by providing architectures able to work on raw data while outperforming traditional techniques. A deep learning architecture which proved to be very successful in the context of computer vision is the Convolutional Neural Network, an architecture able to achieve previously unreachable performances, making it, nowadays, a de-facto standard in classification and detection tasks. However, a drawback of deep learning architectures is their high computational footprint which has limited, until recently, their adoption only to high-performance servers and workstations.

The availability of fast and efficient CNN models together with the release of AI accelerating low-powered platforms has enabled the use of CNNs on the edge, empowering a shift from a server-centric scenario to a fog scenario, lowering data bandwidth requirements for connected nodes and enabling smart features on any connected device.

## 1.1 An edge computing approach to Object Detection and Classification

This work addresses the design and implementation of a CNN based framework for performing road traffic analysis through vehicles detection, classification and tracking, able to efficiently run on a low-powered edge-computing

platform. The resulting platform can be positioned in close proximity to the node, enabling these smart-features on any traditional IP camera.

The first efforts of this work went into selecting the most promising CNN architectures and fine-tune them on a vehicles specific dataset, consisting of around 5.5 hours of annotated videos, shot at various locations around Beijing and Tianjin, in China.

In order to perform an in-depth evaluation and to validate the results on a real use case, additional effort was put in producing a 15 minutes annotated sequence, shot in Turin in c.so Castelfidardo with the company's traffic camera, with more than 16,000 frames and 88,000 vehicle annotations.

After having produced and identified an efficient CNN detector, the second efforts went into the development of a tracking module that uses a very lightweight association metric to track the vehicles detected by the CNN. The obtained tracks can be used to enrich the road-traffic analysis with flow information, and can be matched against multiple counting segments, producing vehicles crossing counts for each of the available lanes.

With a strong and efficient model and the addition of tracking, the last part of the thesis focused on wrapping all up by developing a modular and extensible framework with three main components: a detection pipeline, a REST server and a React web app.

The design of the detection pipeline, which is the main component of the framework architecture, embraced the flow-based programming paradigm, keeping a constant eye on modularity and extensibility and providing extendable plugin-like interfaces making it easily tailorable on any different scenario.

The last two components were developed to control and interact with the detection pipeline remotely and with a user-friendly GUI. The web application is developed in React and runs as a single-page app on the client, interacting with the REST server through AJAX calls. The REST server is developed in Python as a lightweight app using Flask.

# Chapter 2

# Deep Learning and Computer Vision

> *The navy last week demonstrated the embryo of an electronic computer named Perceptron [...]. Dr. Frank Rosenblatt, [...], designer of the Perceptron, conducted the demonstration. The machine, he said, would be the first electronic device to think as the human brain. Like humans, Perceptron will make mistakes at first, "but it will grow wiser as it gains experience", he said. [...] Later Perceptrons, Dr. Rosenblatt said, will be able to recognize people and call out their names. Printed pages, longhand letters and even speech commands are at his reach.*
>
> – 1958, *New York Times*

This article is about the first attempt at synthesizing an artificial neural network. Perceptron was composed of a linear single layer neural unit with connected inputs with trainable weights, and, although the model was later revised and applied successfully in adaptive signal filtering [19], it was still far from achieving what the article mentioned. Nowadays, though, we've reached a stage in which machine learning techniques, and deep learning algorithms in particular, have outclassed previously used techniques and we can finally affirm that these "later perceptron" are able to call out our names by looking at us or even by just listening at our voices, or detect and recognize hundreds of different objects, synthesize speech based on some samples, drive cars, understand and write articles, and many other incredible things.

In the next paragraphs we will concentrate on machine learning tehcniques

applied to Multiple Object Detection, a Computer Vision task in which Convolutional Neural Networks have proven to excel.

## 2.1 Deep Learning algorithms as powerful feature extractors

For decades, conventional machine learning techniques were only able to deal with hand-crafted vectors of features, carefully extracted from the data with considerable domain-expertise; they were thus limited in their ability to deal with raw data such as the pixels of an image [10]. A notable example of image based hand-crafted features used in a detection task is from Viola and Jones face detector in 2001 [17] where they used 4 types of subtractions of rectangle sums of image pixel areas as input features for their cascade of classifiers. The resulting number of features which could be extracted from a 24x24 window was huge, 160000, but only the most promising were selected for further inspection. This features, combined with a clever intermediate image representation called "integral image" which sped up their calculation and a cascade of AdaBoost trained classifiers, made it possible to detect frontal faces with a very high precision at 15fps, almost real-time, on a 700Mhz Pentium 3.

Viola's and Jones's results were astounding at that time but what if we wanted to detect not only faces, but tens, hundreds or even thousands of different classes? Could we use Paul and Jones Haar-like features, or should we use more modern HOG features, or should we extract a completely different set of features from the image? And what about other kinds of data, such as sound-waves or sentences? Deep learning algorithms, a class of representation-learning algorithms, tried to solve the issue by learning how to extract representative features with a deep hierarchical sequence of non-linear modules which, starting from the raw input, proceed, module after module, to produce higher and more abstract levels of representation able to model very complex functions.

## 2.2 Convolutional Neural Networks

A deep-learning architecture that proved to be very successful in dealing with images and that is now widely adopted by the computer vision community is the Convolutional Neural Network, a feedforward network that is designed

to process data presented in the form of multiple arrays, such as the three 2D arrays containing the pixel intensities of the three colour channels of an image.

Convolutional Neural Networks have their roots in 1980's Neocognitron [2], a neural network model inspired by an earlier research on the cat's visual cortex [7], and their first practical use in computer vision dates back to 1990, when Yann LeCun applied the backpropagation algorithm to train a CNN to recognize handwritten digits. Nevertheless they became very popular in computer vision only after Alex Krizhevsky, a PhD student at the university of Toronto, won the 2012 ILSVRC (ImageNet Large Scale Visual Recognition Challenge), one of the most important classification and detection challenges, with a CNN which will later be known as AlexNet; his solution marked a milestone, beating the second submission with a gap of 10% on the error rate. In the following years many architectures emerged, further increasing classification precision, and the most important ones will be later discussed.

### 2.2.1 A standard architecture

The basic blocks of a standard convolutional neural network are layers of neurons that, not differently from a traditional neural network, start from an input to which they apply a dot product, followed by a non-linearity activation, some pooling and finally in the last layer, combine their output with the outputs of other neurons of that same last layer, in a fully-connected manner, contributing to the network output, a differentiable class score. The difference lies in how those neurons are made and how they are connected to the input, since CNNs are designed to work with images they exploit this fact to overcome the limitations of fully connected neural networks in dealing with such inputs. In a fully connected neural network we would have weights for each connection between an input, in this case a single channel of a pixel, and a neuron, which, for a 300x300x3 image would mean 270000 parameters for each neuron of the network; this scenario obviously can not scale well so convolutional neural networks borrow the concept of receptive field of brain cells by having neurons look at a small portion of the image and slide across it to calculate their output. These convolutional neurons are usually called convolutional kernel, or filters, and they compose the convolutional layers of a standard CNN architecture

### Convolutional layers

Convolutional layers are made up of convolutional neurons, usually called kernels or filters, which are 3d arrays of weights that can be viewed as a series of matrices, where the 1st and 2nd dimension define row and cols while the 3rd dimension, the number of matrices, matches the depth of the input. Common sizes for convolutional kernels range between $1 \times 1 \times D_{in}$ to $5 \times 5 \times D_{in}$, although they are rarely bigger than $3 \times 3 \times D_{in}$. Each kernel slides across the input and for each position it produces one single output consisting of the sum of the products of the values times the weights on all the input depth channels.

The following image 2.1 shows two convolutional layers (kernels) applied to an input image with the three RGB channels, sliding by 1 position each time, and producing a $3 \times 3 \times 2$ output volume:

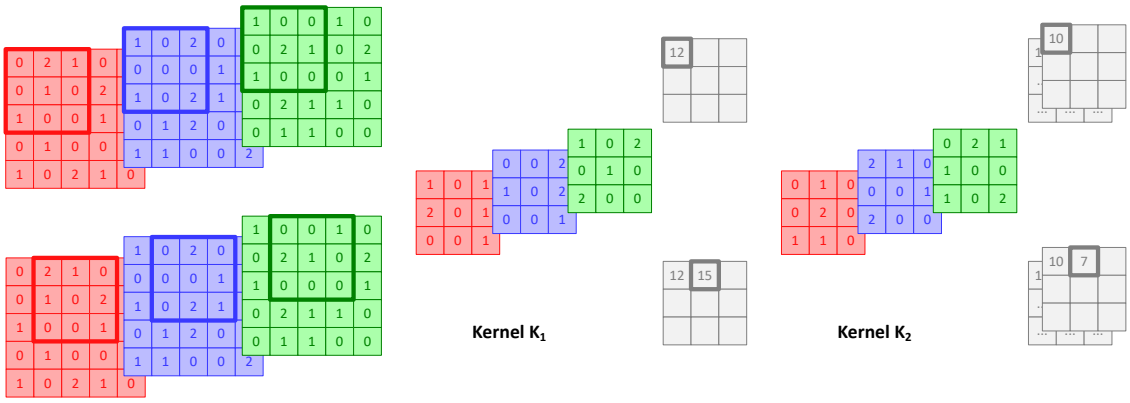



Figure 2.1. Example of a convolutional layer composed of 2 convolutional kernels (without considering bias matrices) sliding with a stride=1. The depth of the kernels matches the depth of the input volume and the number of kernels defines the depth of the output volume

For simplicity, the image does not show nor consider the bias matrix in the calculation, which is an additional matrix for each kernel with weights that get added to the final output independently of the inputs (like the constant terms in polynomials). The kernel weights will be learned during the training process in order to extract the most distinctive features, with regard to the function they have to model, from the input samples.

An example that may be useful to better understand how they work regards the filters used in Computer vision. In fact, border extraction, sharpening or blurring of an image can be obtained by convoluting special kernels over the image.

### Activation Layers

Activation layers are usually placed after convolutional layers and they simply apply a non-linear activation function to each unit of the output volume of those layers. They are used to add non-linear properties to the network, without which we would have a Linear Regression Model with limited ability to model complex functions on the input distribution. Among the activation functions, the one that's more used is the ReLU and it's derivatives, thanks to its simplicity and computational efficiency and thanks to the fact that it helps mitigating the problem of vanishing gradients, i.e., differently from the sigmoid or the tanh, whose derivative tends to zero for high activation values, ReLU's derivative is always one for any value greater than 0.

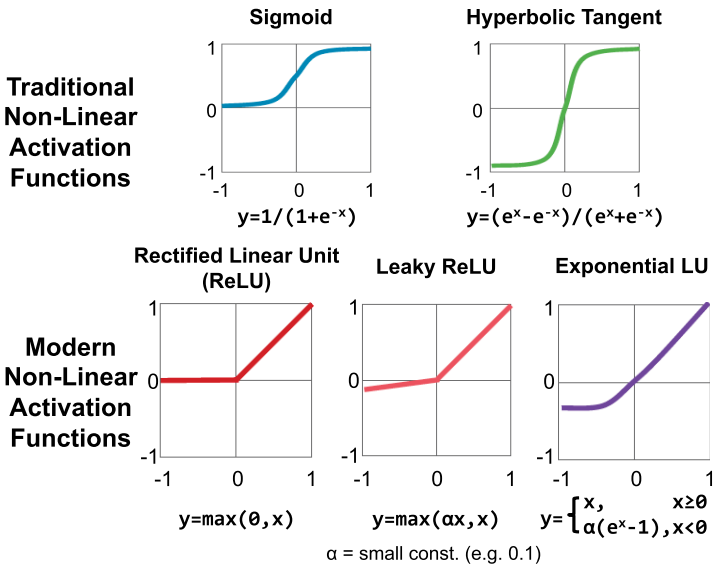



Figure 2.2. Various non-linear activation functions, taken from [14]

## 2.3 CNN Architectures

This paragraph presents an overview of the popular convolutional neural networks models. This section is divided between models developed to achieve the maximum precision and models developed to achieve good precision with fast computational performance and small memory footprint.

### 2.3.1 Towards precision

The various challenges with their metrics imposed a search for better and better (improving) architectures, able to generalize better and score higher than previous ones.

#### AlexNet

#### VGG

VGG (Visual Geometry Group, University of Oxford) and their very deep architecture

#### Zeiler-Fergus

VGG similar results with a smaller footprint. Also interesting article on visualizing CNNs

#### GoogleNet - Inception v1/v2

Going deeper with convolution [15]

#### ResNet

Residual connections between layers [6]

### 2.3.2 Towards speed

Achieving the highest possible precision is not always the main aim since in specific cases, such as the case of this work, size of the model and speed of detection are also an issue. Finding the right compromise between classification performance and speed/size performance is not an easy task but some research teams at Google came up with architectures that could outperform previous models at a fraction of the cost.

**SqueezeNet**

**MobileNet v1/v2**

## 2.4 Object Detection through CNNs

Standard convolutional neural networks are used for classification problems while detection needs a different, more complex approach. A typical object detection algorithm needs to produce a vector of bounding boxes that provide the location of detected objects and a vector of class scores for the corresponding bounding boxes, which determines what kind of object was detected.

To achieve such a result, the very first naive implementations were using sliding windows of different sizes and ratios, performing a classification for every extracted portion of the image, and such an approach was anything but performant. Naturally reasearchers came up with algorithms that sped up the detection process and in the following paragraph some of them will be analyzed.

### 2.4.1 Detection algorithms

In this paragraph we will briefly analyze the key points of some of the most used detection architectures which were considered for this thesis.

#### R-CNN

Regions with CNN features [5], this is the long name, is an object-detection algorithm made up of three modules, a region proposal module, a CNN module and a classifier module. The region proposal module uses selective search [16], with shallow hand-crafted features, to propose a certain number of regions, 2000 in the article, and passes these regions, warped to a fixed size, to the CNN module. The CNN backbone, the one chosen in the article was AlexNet [9], executes a forward pass for each proposed region and extract a feature vector, which is handled to the third module, a series of linear support vector machines, one for each class, that classify the corresponding region (car? y/n - airplane? y/n - ...). Finally, a class-specific linear regression model is applied to each output to predict a more precise bounding box. The results in object detection challenges were higher than other contemporary approaches but the architecture had some disadvantages [3]: firstly the training is a multistage process, since the CNN, the SVMs and

the bounding-box regressors have to be trained individually, furthermore the computational performances were not brilliant, not even close to something usable on real-time video analysis.

### Fast R-CNN

After less than a year, R. Girshick came up with an improved version of their detection framework, simpler and faster to train and a lot faster at test time [3]. Without delving into details, the two main improvements are the CNN computation of only one shared feature vector instead of one feature vector for each region proposal and the substitution of the N class-specific SVMs with a single softmax classifier; the resulting architecture has a single stage training and is, as the name suggests, faster. The single feature-vector from the CNN forward pass is extracted from the whole image, max-pooled according to the selected region, fed to two fully connected layer and finally to two final sibling FC layers, one for the softmax classifier and one for the bounding-box regressor. The article tested the architecture with different configurations and different CNN backbones, including the very deep VGG16. Resulting precision with the VGG16 is higher than the previous R-CNN architecture and it's also 9× faster at training and 213× faster at test-time

### Faster R-CNN

The third incarnation of the R-CNN algorithm [13] further speeds up the computation by substituting the selective search used for region proposal with a Region Proposal Network, RPN for short, that shares the computational layers with the main CNN. The idea is that the convolutional features may be useful not only for the classification but also for the creation of the region proposals. The RPN shares a part of the architecture with a part of the main CNN and, on top of that, uses a mini-network which sildes on the feature map and generates, for each position, the predicted coordinates for a number k of anchors, at different scales and ratios, and their objectness score (the probabilities of being objects or background), doing the work of both the box regressors and the selective search from the previous model. The proposed regions are then fed, as in the Fast R-CNN, to the final detection layers to classify the detected objects.

### YOLO - You Only Look Once

YOLO [12] stands for You Only Look Once and the algorithm takes its name from the fact that the detection task is merely a forward pass of a convolutional neural network, the CNN, indeed, predicts both the classes and the precise bounding boxes without any additional processing. The backbone is a custom model inspired by GoogLeNet [15]

# Chapter 3

# Implementation details and choices

This thesis focuses on developing a framework which leverages an AI solution to enable video analytics from a live camera, capable of running on a low powered edge device. Many variables have to be considered, starting from the hardware platform which has to be capable of handling the bilions of operations required by a single forward pass of a CNN and with enough memory to store the millions of weights, to all the software stack required to develop and deploy the project, like the AI framework used to deploy the neural network, the algorithms used for object detection and tracking, the whole design, the communication methods, the control part, and many other details.

Some frameworks that will be briefly introduced in this and the following chapters were considered and tested in the first months, albeit later discarded, due to the fact that the project started with a proposed hardware platform which was changed during the course of this work.

## 3.1 Edge deployment platform

Convolutional neural networks models, even the smallest, require doing billions of floating-point operations for each frame they have to process and their weights may need from several hundreds of MBs up to GBs to be loaded in memory. A low-powered board with a general purpose processor can not perform well in a similar scenario and this factor have been limiting the use of AI on the edge until now. In fact, in the last few years new solutions arose

and among them, Nvidia created a family of products built exactly for this purpose, the Nvidia Jetson.

The Nvidia Jetson is a family of products built to enable powerful AI applications with a low power profile and relatively small form factor on the edge, which makes them a perfect candidate, from a capabilities point of view, for the project of this thesis. The Jetson family includes full development boards and the corresponding modules for the final system, as shown in figure 3.1, but the thesis work concentrated on building a fully working prototype on the development board.

### The first solution - Nvidia Jetson TK1

The first selected deployment scenario consisted on developing the smart framework for the Nvidia Jetson TK1, the first installment in the Nvidia Jetson family, since the company had some units in house. The first months of the development, thus, concentrated on testing frameworks which were capable of performing object detection with a good precision a computational performance but also compatible with the TK1 hardware and software stack.

The constraints were too stringent since the TK1 was released in April 2014 and Nvidia itself dropped support for the board and discontinued its commercialization in April 2018. The situation was also worsened by the fact that with the successive generation the architecture changed from arm32 to aarch64 so all the new Nvidia libraries were not retro-compatible with the old hardware. This meant that CUDA stopped at v6 and modern NN frameworks such as Tensorflow v1.0 onwards, Pytorch and Caffe2 could not be used. A first solution was developed both with a fork of YOLOv2 [] (the original framework and YOLOv3 weren't working on the platform) and with the original deprecated Caffe implementation of the FasterRCNN [13, 4], and the results weren't satisfactory. All these constraint led to evaluate a platform-change in favour of the latest installment in the Jetson family, the Nvidia Jetson TX2.

### The final solution - Nvidia Jetson TX2

The final solution chosen for the project is the latest installment in the Nvidia Jetson family, the Jetson TX2 development board and the hardware specs are highlighted in the following table:

| Hardware Specs | |
|---|---|
| Processor: | HMP Dual Denver 2/2 MB L2 + Quad ARM A57/2 MB L2 |
| Architecture: | aarch64 |
| Memory: | LPDDR4 8 GB 128-bit 59,7 GB/s |
| GPU: | NVIDIA Pascal, 256 CUDA cores |
| CUDA compute cap.: | 6.2 |
| Video Memory: | Unified with main memory |

This module was released in March 2017 and, differently from the TK1, it is fully supported by Nvidia and it is compatible with all the latest libraries, as shown in the following table:

| Software Specs - as of Jetpack SDK v3.2.1 | |
|---|---|
| OS: | Ubuntu 16.04 aarch64 |
| CUDA support: | v9.0 |
| cuDNN: | v7.0.5 |
| TensorRT: | v3.0 |
| Gstreamer: | v1.8.1 with HW enc/dec acceleration support |
| OpenCV: | v3.3.1 but the provided version does not support HW video acceleration and it needs to be rebuilt |

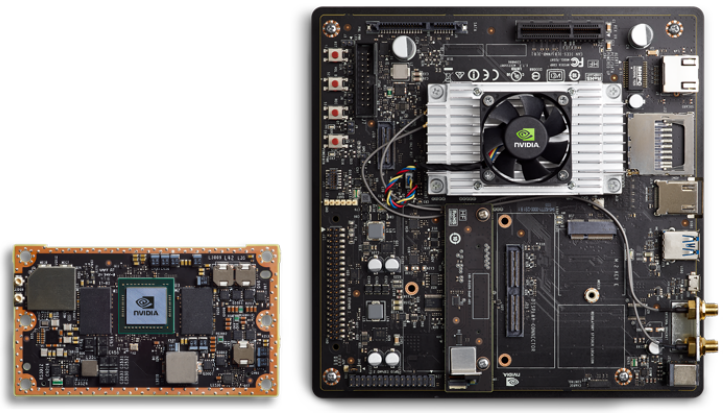


Figure 3.1. NVIDIA Jetson TX2 - Module on the left, Development board on the right

## 3.2 Neural Networks framework and Detection algorithms

An important choice regards the framework used to deploy the neural network and it is linked not only to platform compatibility but also to the detection algorithm and to the model chosen for the task, since different frameworks provide different implementations and some algorithms may not be found for certain frameworks. A key objective for analyzing live data is achieving good detection performance by keeping the computational cost as low as possible, and to achieve such a result single-pass detection algorithms such as YOLO [12] and SSD [11] were favoured to more complex ones such as Faster R-CNN [13] (for an overview of the key features of the state of the art detection algorithms the reader can refer to section 2.4.1).

The first testing embryo of the detector was developed with Darknet [8] for the first platform, the Nvidia Jetson TK1, but the official framework was working only on the development laptop while it was outputting empty detection vectors on the board. The only way it could work on the board was by using a fork which only supported up to YOLOv2, but this solution was later dropped in favour of a different framework, Tensorflow, offering more flexibility and a better support.

### 3.2.1 Tensorflow Object Detection API

The final choice was to use Tensorflow, a more modern, flexible and robust framework, developed and mantained by Google. A key point that led to the adoption of Tensorflow was the availability of an Object Detection API [6] developed by Tensorflow researchers which includes premade models combining some of the state-of-the-art detection algorithms with many of the main CNN backbone models discussed in section 2.3.
The available detection algorithms are the following:

- SSD 300/600
- Faster R-CNN
- Mask R-CNN
- R-FCN

The available backbone architectures are the following:

- MobileNet v1/v2
- Inception v2/v3
- ResNet 50/101
- Inception Resnet v2

Pretrained models of the above combinations are available in the API repository model-zoo. The provided models are trained on the MS COCO dataset with 90 classes and can be downloaded to be used directly for inference tasks or to be used as a starting-point for fine-tuning on different datasets.

The authors of the API also provided a very neat chart which compares the average detection precision, defined according to the COCO mAP metric (a description of the metric can be found in section, against the inference time (i.e. the time to produce the detections from a single frames) for many of the provided combinations. Reproducing all these tests would be time-consuming and not very useful so the graph will be reported as is from the original article:

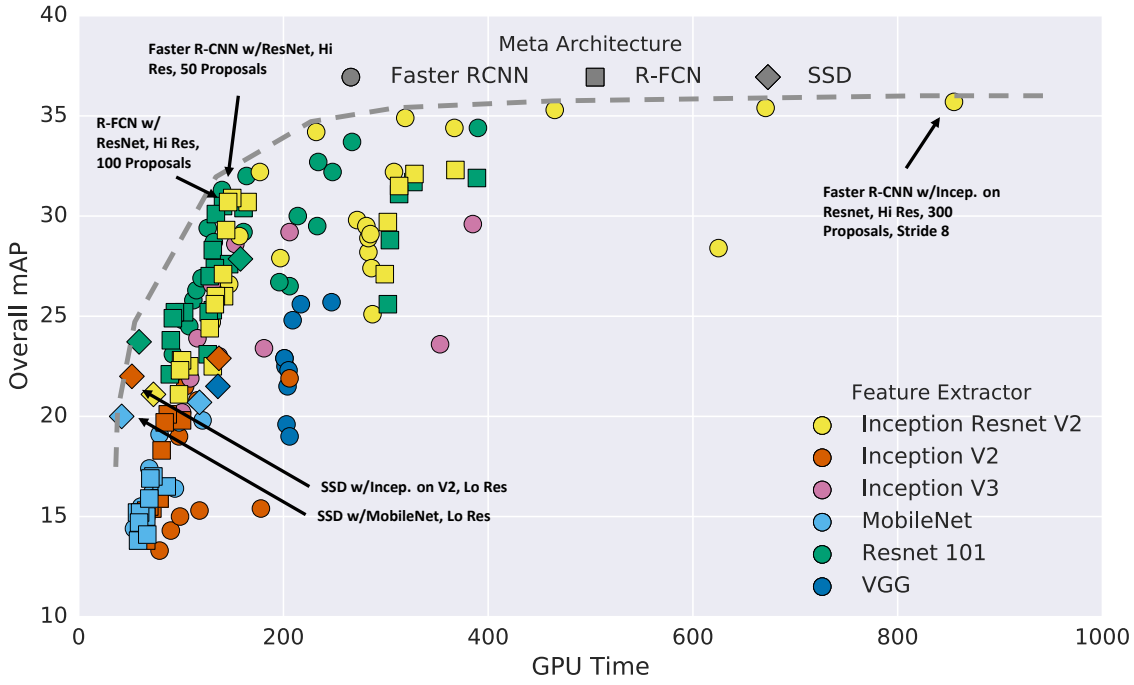



Figure 3.2. Comparison between the various combinations of detection meta-architectures and CNN backbones, from [6]

As shown in the graph, there is a frontier, reported with a gray dashed line, that marks the best trade-off between precision and inference time that was achieved with some combinations by the research team and the models trained, analyzed and used in this work have been chosen among them (refer to section 4.4).

## 3.3 Software environment choices

The framework developed in this work is written in Python, due to the fact that it permits rapid prototyping, compared to other languages like C, and due to the fact that a large number of open source libraries that could aid during the development process are available. This choice, though, doesn't come for free, in fact Python is slow compared to C, and in order to guarantee almost real-time performance, there has been a high stress on the profiling part, with heavy use of multiprocessing and numpy vectorized operations whenever possible. The framework was profiled with cProfiler, first on the development machine and then remotely on the card, in order to eliminate almost all the bottlenecks that could harm performance.

### Additional libraries and frameworks

For the development part of the work, some open source libraries have been used:

- **OpenCV** has been used for any imaging function written for the framework, from masking, to drawing boxes, polygons, texts or tracks. The initially used box-drawing function used PIL and plain Python operations and was quite slow. The final version, rewritten using Numpy operations for preprocessing and OpenCV for actual drawing, was 219x faster than the first one.

- **Protobuf** has been used to provide a serialization mechanism between the modules composing the framework. It's a lot faster, compared to serialization mechanisms such as Pickle.

- **Supervisord** has been used to provide a mechanism of process control. As it will be shown in section 5.1.2, the modules of the framework run in separate processes and supervisord is internally used to control their running state.

- **PyZMQ** has been used to provide communications between the modules, providing, among others, Publisher/Subscriber and Request/Response patterns through IP sockets or Unix sockets (the Unix ones were used). The Publisher/Subscriber pattern, in particular, has been used very useful for the framework design. One additional advantage of using PyZMQ over plain Unix sockets is that it provides recovery mechanisms for free, so if a module goes down and then gets relaunched, other connected modules won't notice a thing. The last note regards the framework design being independent of this choice, in fact, PyZMQ sockets are wrapped and modules use "Pipes" interfaces designed so that changing the underlying communication component could be done at almost no cost.
- MQTT Paho
- CVAT
- Flask
- React
- Telegraf
- InfluxDB
- Grafana

### 3.3.1 Built from source

Some libraries had to be built from source for various reasons. Here are the main ones

#### OpenCV

Nvidia JetPack 3.2.1 comes with OpenCV 3.3.1 directly built for the board so everything seemed to be fine on this regard. Nonetheless, while testing the 4K stream from the camera provided for this work some preformance problems were encountered and one of the factors (though not the only one, as it will be shown in the profiling section was that OpenCV was not using hardware accelerated decoding. Surprisingly, the version that came prebuilt with Nvidia SDK was not built with Gstreamer support for hardware acceleration, so it had to be rebuilt with its support. Moreover, two pipelines

had to be defined in order to make the interface work with OpenCV. More details on the subject can be found in section 5.2.1.

Building OpenCV needs various requirements and the actual build may greatly vary depending on the configuration that the user may set to customize it. In order to ease the build and the installation processes a 400 lines interactive script was written and provided with the project sources. As a reference, the cmake build configuration will be provided here:

```
cmake \
      -D CMAKE_BUILD_TYPE=Release \
      -D BUILD_TESTS=OFF \
      -D BUILD_PERF_TESTS=OFF \
      -D BUILD_EXAMPLES=OFF \
      -D WITH_FFMPEG=ON \
      -D WITH_GSTREAMER=ON \
      -D BUILD_opencv_java=OFF \
      -D BUILD_opencv_python2=ON \
      -D BUILD_opencv_python3=ON \
      -D ENABLE_NEON=ON \
      -D WITH_CUDA=ON \
      -D CUDA_ARCH_BIN="6.2" \
      -D CUDA_ARCH_PTX="" \
      -D WITH_CUBLAS=ON \
      -D WITH_GTK=ON \
      -D WITH_TBB=ON \
      -D ENABLE_FAST_MATH=ON \
      -D CUDA_FAST_MATH=ON \
      -D WITH_LIBV4L=ON \
      -D WITH_QT=ON \
      -D WITH_OPENGL=ON \
      -D INSTALL_C_EXAMPLES=ON \
      -D INSTALL_TESTS=OFF \
      -D OPENCV_EXTRA_MODULES_PATH=$SOURCE_DIR/opencv_contrib/
    modules \
      $SOURCE_DIR/opencv_src
```

# Chapter 4

# Fine tuning the models

The pretrained models available in Tensorflow OD API model zoo are trained on the MS COCO dataset, a dataset that includes more than 200000 annotated images on over 80 classes, including cars and buses. The most sophisticate models perform well on the vehicles detection task even out-of-the-box but the main problem is that those sophisticated models have a very high inference time and fail to provide the near real-time detection needed for the tracking component of the project. When evaluating smaller and faster models such as MobileNet or Inception out-of-the-box, they fail to provide usable detection performance and they need to be fine-tuned on datasets specialized on the task. While for a "first-person" view, such as the one from an onboard camera of an autonomous vehicles, there are many valid datasets, the only large dataset that covers vehicle detection from a surveillance-camera point of view, available for free but only for academic-purposes (a registration with a valid academic e-mail or a valid proof of the status has to be provided), is the UA-DETRAC dataset.

The term "fine tuning" indicates a typical workflow in which the weights of the network to be trained are initialized from an existing model which was already fit on a different dataset. Sometimes, the term "transfer learning" is also used, although the latter usually indicates that the model was fit on a dataset from a different field of application. The practices of fine tuning or transfer learning, opposed to training the model from scratch with randomly initialized weights, cut the training time from days to hours and can also help overcome some problems while training models. For example, in R-CNN [5], the weights are first trained on the ImageNet classification dataset then, after the model converges, the final classification layer is substituted with one for the final $N+1$ classes (+1 for the background) and the resulting

network is fine tuned on the detection dataset, granting a better mAP while also reducing the risk of overfitting the smaller detection dataset.

## 4.1 UA-DETRAC dataset

The UA-DETRAC dataset [18] provides more than 10 hours of videos of road traffic in multiple loaction of Beijing and Tianjin in China, captured from a traffic surveillance camera point of view in multiple times of the day and various weather conditions. The dataset provides annotations for four different classes: cars, buses, vans and others (which include trucks or other kinds of vehicles). The dataset provides other information for each detection, such as scale, occlusion ratio and truncation ratio, that are not considered in the proposed solution, neither for training nor for evaluation.

The dataset is is divided into a training set, composed of 60 sequences, and a test set, composed of 40 sequences which present similar traffic conditions but are shot from different locations. The organization that created and maintains the dataset also hosts a challenge and uses the test set for the challenge evaluation so, unfortunately, it does not provide the ground-truth annotations for the test sequences. This means that only a maximum of 6 hours of videos are available for training purpose and, as it will be shown later, the train data will be reduced even further since a part of this training data will be used as a test set for the training process.

Figure 4.1 shows frames from some of the sequences in the dataset, with the type of vehicles in the labels, the ignored non-annotated regions greyed out and a box color indicating their occlusion ratios.

The resulting models have a minor tendency on confusing some detected samples between cars and vans, this may be due to some correlation between the training samples of the two classes.

## 4.2 The training process

A typical training process of a convolutional neural network is made up of two alternating phases, the train phase and the validation phase, finally followed by a third and last test phase, and these three phases are typically associated to three different sets of the training data, the train-set, validation-set and the test-set respectively.

The train phase is when the actual training is performed so, for each training step, a batch of images if extracted from the train-set and used to


Figure 4.1. Frames of some of the sequences of the UA-DETRAC dataset, showing vehicles classes, ignored regions, and truncation-ratios as box colours. Taken from [18]

update the weights through the backpropagation algorithm with stochastic gradient descent. A train phase usually comprises a certain number of train steps, after which the train process switches to the validation phase

After a number of train steps are performed, or after a certain time in the train phase has passed, a rapid evaluation of the model is performed. This phase outputs logs with selected measurements, such as the value of the loss function and, in case of a detection network, AP values to measure the detection performance, that need to be tracked during the training process. The samples used for this phase are taken from the validation set, which should be different from the train set in order to highlight the generalization capabilities of the model. The outputs of the validation phase are generally used to evaluate the whole process and to select a promising model configuration as a possible final model.

At the end of the training process, when a promising final model is selected, a test phase is typically performed by evaluating the performances of the selected models against a different, possibly uncorrelated, set of samples, the test set. Since the models are chosen by selecting high performances on the validation set, having a separated test sets is important to enable a more

precise evaluation of how a model will perform on unseen data.

### 4.2.1 Adopted process

The adopted process follows the structure that was just introduced, alternating between a train phase and an evaluation phase until the detection performance on the validation set seems to stop its ascent and keeps oscillating around the a maximum value and, in parallel, the total loss function seem to stop its descent and oscillate around a minimum value. The stop is thus manual and the best checkpoint, i.e. the one with the higheast mAP over the validation set, is then chosen. Note that a thorough evaluation on the test set and on some additional test data is done and presented in the results chapter.

The conditions that toggle the switch between the train and the evaluation phases are the following:

- **Train → Evaluation:** when 10 minutes of training are passed
- **Evaluation → Train:** when the evaluation is complete (more on this in the following paragraph)

Since, as it will be shown in section 4.3.1, the evaluation set is quite large, consisting of 5 sequences for a total of 6246 images, a full evaluation every 10 minutes is unfeasible (the training would spend a lot more time evaluating rather than fitting) so the evaluation configuration is set to perform the eval step on 1 sample every 20 samples.

Image 4.2 depicts the process and summarizes what has been said in this section.

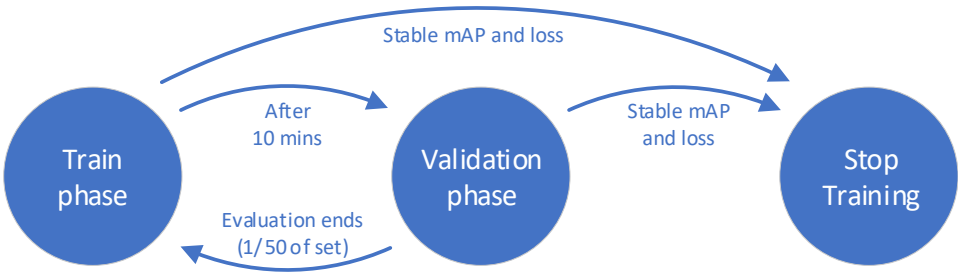



Figure 4.2. Schematic view of the training process and conditions

## 4.3 Dataset preparation

### 4.3.1 Defining train and validation sets for the UA-DETRAC dataset

As noted in the dataset description, the dataset comes with a train set of 60 sequences and a test set of 40 sequences but only the former comes with ground-truth annotations files. Albeit the test set does not have ground truth boxes, it can be used for evaluation with their provided toolkit, with a limitation though: it evaluates detection performance of vehicles in general, without discerning between classes. That's why new test data will be introduced in the results chapter. Another limitation of this subdivision is that there is no explicit validation set so the validation sequences had to be extracted from the train set. The chosen subdivision comprises 55 sequences for the train set and 5 sequences for the validation set and it is shown in figure 4.3.


Figure 4.3. Summary of the chosen subdivision between train and validation sets

The rationale behind the above 55-5 subdivision is to keep the train set as large and diverse as possible in order to reduce the risk of overfitting, considering that only 60% of the whole dataset (approximately 6 hours) is annotated for the public and that the 60 sequences are not from 60 different situations, with many sequences being just different clips from the same shots. That's why the clips extracted for the validation set only contain shots which have direct siblings in the train set and no unique shot was extracted. It can be a risk since having no unseen setting in the validation set will eventually bias the evaluation results but keeping the most possible diverse train set was preferred.

### 4.3.2 Preparing the samples

The annotated sequences are composed of a set of frames and an XML file containing the annotations for the frames of the sequence, and they have to be parsed in order to be used by the Tensorflow estimator (the API object responsible for the training). An XML sample UA-DETRAC annotation file follows this structure:

```
<sequence name="MVI_20011">
    <sequence_attribute camera_state="unstable" sence_weather=
    "sunny"/>
    <ignored_region>
        <box left="778.75" top="24.75" width="181.75" height="
    63.5"/>
        <!-- Possibly other regions... -->
    </ignored_region>
    <frame density="7" num="1">
        <target_list>
            <target id="1">
                <box left="592.75" top="378.8" width="160.05"
    height="162.2"/>
                <attribute orientation="18.488" speed="6.859"
    trajectory_length="5" truncation_ratio="0.1" vehicle_type=
    "car"/>
            </target>
            <!-- Other targets in the frame... -->
        </target_list>
    </frame>
    <!-- Other frames of the sequence... -->
</sequence>
```

The XML root is a *sequence* object which has the following children:

a *sequence_attribute* child that specifies camera stats and weather, an *ignored_region* child which contains boxes of regions that have not been annotated (i.e. far segments of a road, parking areas) and N $frame$ children that contain the annotations for each of the N images in the sequence. Each $frame$ object has a $target\_list$ with a series of $target$ children, each containing a *box* with the coordinates of the rectangle that surrounds the object and an *attribute* object with data like the category (car, bus, van or other) or the speed of the vehicle. The objects in the annotations are tracked between the frames and the target id can be seen as a track id which the object during its passage in the scene.

The annotated data that has to be fed to the training process in TensorFlow needs to be parsed to $n$ *tfrecord* files, where $n$ is configurable and corresponds to the number of *shards*. Each TFRecord file will contain a series of Tensorflow *Example* objects, one for each frame sample, that contain not only the annotations but also the encoded image data as JPEG. Setting the number of *shards* to 1 produces 1 big tfrecord file containing the samples of the whole dataset, while setting it to something greater than 1 will produce multiple files, providing speed benefits by enabling concurrent readers and facilitating the shuffling operations while reading the samples. A form of very simple shuffling is performed during the write operations by simply iterating through the record files while saving the frame samples. In this work, the number of shards was set to 20.

### Dealing with the ignored regions

A problem that has to be addressed is how to manage the ignored regions, since Tensorflow od api doesn't have a built-in way to manage them. Leaving them be is not an option, since the loss function would penalize the network if any detection in those non-annotated regions is found (they would be treated as false positives), so the simple adopted solution was to mask them with a uniform black overlay, as it can be seen in the following image.

A limit of this approach is that some context information is lost due to the masking but it is also true that in the data augmentation step random crops are performed so the loss should not be too noticeable. It would be interesting to analyze the performance differences, if any, between masking the image versus managing the ignored regions directly in the training process but this would require modifications to the algorithm itself, which is why this comparison was discarded. The mask solution was thus chosen since it works out-of-the-box with accurate detection performance.

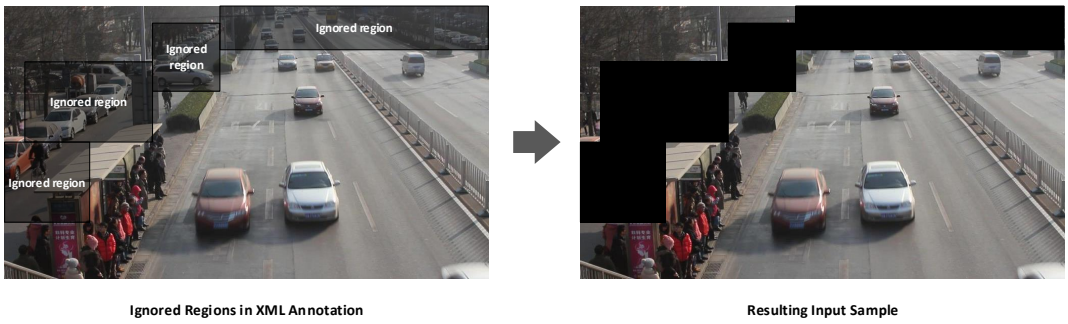



Figure 4.4. Cutout of ignored regions in input samples

### 4.3.3 Data augmentation

The smaller the training dataset, the higher the risk to overfit it. A way of dealing with the overfitting problem is to virtually enlarge the dataset by creating new samples starting from the original ones; the process is typically referred to as data augmentation. In this work data augmentation is done online at training time by a preprocessor offered by the object detection api. In this way, no additional input data has to be created and stored. The preprocessor has been configured to mimic the augmentations done by the Faster R-CNN and SSD papers.

For the Faster R-CNN architectures the input samples are just randomly horizontally flipped, while for the SSD architectures the inputs are randomly flipped and also randomly cropped. The random crop method follows the one defined in the SSD paper, where the input sample is randomly processed in one of the following ways:

1. Left as is

2. Cropped with a IOU threshold of 0.1, 0.3, 0.5, 0.7, or 0.9 with at least one ground truth object

3. Cropped without caring of the above constraint (implemented by setting IOU threshold to 0.0)

The image crop is eventually executed by keeping a random size between 0.1 and 1 of the original image size and an aspect ratio between 0.5 and 2.

## 4.4 Chosen models and hyperparameters

As seen in section 3.2.1, the Object Detection API offers a multitude of combinations of algorithms and models and the research team behind the project produced a graph (figure 3.2) plotting the detection precision against the inference speed. Four of what seemed to be the most promising combinations of algorithms and models were chosen and fine tuned in this work:

- SSD with MobileNet v2 backbone
- SSD with MobileNet v2 backbone and small anchors
- SSD with Inception v2 backbone
- Faster R-CNN with ResNet-50 backbone

Three out of four combinations use the SSD algorithm since it appears to give good results with a smaller computational cost, the Faster R-CNN, instead, was fine tuned to provide a slow but more precise model and see how it compares to the others.

### 4.4.1 Training hyperparameters

The training hyperparameters are all those parameters which are set before the training and influence the outcome of the training process. Since this work has a great emphasis on the framework development part, the hyperprarameters were set following the ones provided by the researchers behind the Object Detection API and the only modification that has been done regards the size of anchor boxes of the SSD MobileNet v2, thus the distinction between SSD MobileNet v2 and SSD MobileNet v2 with small anchors.

The models were trained with stochastic gradient descent and mini-batches of size 24, except for the Faster R-CNN with ResNet-50 for which the size was set to 1 for memory reasons. The learning rate has been set to 0.004 for the SSD models and to 0.0003 for the Faster R-CNN (it should counteract the effects of the less precise weight updates due to the batch size of 1). The optimizers choices follow the ones made by the research team to obtain their results, so the RMSprop was used for the SSD models while Momentum was used for the Faster R-CNN Resnet-50.

## 4.5 Training evaluation

As previously seen, the training phase is alternated, after a certain number of training steps or time, to evaluation phases in which values of the loss function (and it's components) together with some measures on the model detection precision are produced (the discussion on the evaluation metrics will be presented in the next section). This continuous evaluation is useful to assess the training performance and to provide a means to decide for an early stopping of the training process (one can see when the loss function and the detection performance seem to have settled on some values or even if the precision starts decreasing due to overfitting on the training set).

The results of the evaluations regarding the detection precision use the COCO mAP metric, where mAP stands for mean Average Precision, so the next section will describe the metric and give a simplified description on how it is calculated. Finally the evaluation results will be shown.

### 4.5.1 Evaluation metrics

Assessing the detection performance of a model is not a simple task, for example some models might be better at classifying the detected objects, some others might be more precise at localizing the object (i.e. they produce a tighter bounding box), moreover, models could be biased and perform better on some specific classes rather than others.

In order to quantify the detection precision the mAP, mean Average Precision, along with it's variations, is the standard most used metric. In particular, the COCO mAP Detection evalutation metrics will be used. This section will describe the metrics definitions and calculations, starting from it's basic concepts.

The first concept that will be introduced is the IOU which is used to calculate the distance between two detection boxes to determine if the detection is a hit or a miss. After the IOU, the concepts of Precision and Recall, which are the components used to calculate the mAP, will be introduced. Finally, the mAP and it's variations will be described.

#### Intersection Over Union

IOU is an acronym that stands for Intersection Over Union and, as the name suggests, it measures the similarity between two boxes as the area of the intersection of two detection boxes with respect to the area of the union of

the two:

$$IOU(A, B) = \frac{Area(A \cap B)}{Area(A \cup B)}$$

A visual representation of the IOU is shown in figure 4.5.

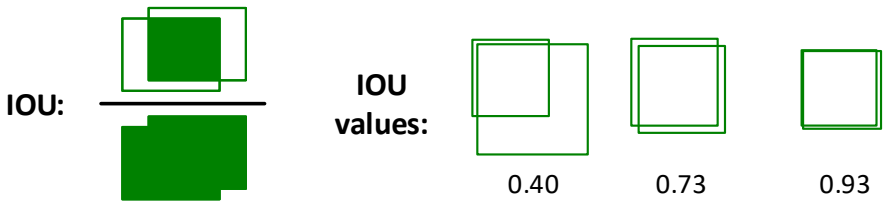



Figure 4.5. Visual representation of the IOU distance with some examples

In the context of evaluation, it is used to match detections with ground-truth objects. Models with precise localization, producing boxes which are enclosing the ground-truth object more precisely than other models, will have a higher IOU for their detected boxes and it will be seen that a specific set of metrics (the COCO mAPs) favours this aspect.

The link between this metric and the evaluation will be explained in the following paragraph.

### Hit or Miss? Defining True and False Positives and False Negatives

How does the COCO mAP evaluation algorithm determine whether a detection is a hit or a miss?

Firstly let's fix a few parameters to make things simpler. Let's consider one class for the detection task, let's say cars, choose a IOU threshold, for example a IOU threshold of 0.50, and a confidence threshold, let's say 0.5 (the confidence is an output produced for each detection model that tells how sure the model is that the detection is correct and regards the predicted class, see figure 5.2).

Based on the three parameters considered above, the algorithm cycles all the cars detections with a score greater than the confidence threshold (at least 50% sure it's a car) and tries to find the best ground-truth box of the same class that can match that detection.

The best matching ground-truth box $gt_j$ is the one that has the highest IOU score with the considered detection $d_i$:

$$BestMatch(d_i) = \{gt_j | \max_{gt_j \in GT_{boxes}} IOU(d_i, gt_j)\}, d_j \in D_{boxes}$$

If the IOU between the detection $d_i$ and the best match $gt_j$ is greater than the considered IOU threshold, the detection is considered a hit, technically called *True Positive*. Instead, if the best-match $gt_j$ has a IOU with $d_i$ lower than the threshold or if no matches are found (i.e. all IOUs between detection and ground truth boxes are zero), the detection is considered a miss, technically called *False Positive*.

This process is carried on for all detections, removing ground truth boxes while they are matched, and after all detections are processed the remaining unmatched ground-truth boxes are considered as *False Negatives* (e.g. undetected cars).

### Precision and Recall

Precision and recall are the two main components of the detection evaluation. They are strictly linked to the concepts of True Positives, TP for short, False Positives, FP for short, and False Negatives, FN for short, as defined in the previous paragraphs.

The *precision* measures how reliable are the model's positive detections and it's defined as follows:

$$Precision = \frac{TruePositives}{TruePositives + FalsePositives}$$

A model that produces a very low number of wrong detections, even if the detections are few, will have a high precision. For example, a model that detects only 2 cars out of 10 (2 TP) but does not produce any wrong detection (0 FP) will have a precision of 1.0, even if it left out many undetected cars. Thus, in order to assess the detection performance a second metric is introduced, it's the *recall*.

The *recall* measures how good is the model at detecting ground-truth objects and it's defined as follow:

$$Recall = \frac{TruePositives}{TruePositives + FalseNegatives}$$

A model that detects many ground-truth objects will have a high recall. For example, a detector that detects 10 cars out of 10 but also detects 2 trees, 4 bushes and 2 patches of road as cars, would have a recall of 1.0.

To summarize, the first detector (2 out of 10 cars and no other detections) would have a high precision but a low recall, the second detector (10 out of 10 cars but also trees, bushes, background detected as cars) would have a high recall but a low precision, an optimal detector would have both precision and recall at 1.0 (10 out of 10 cars and nothing else detected as car).

### Average Precision

In the previous paragraphs, among the fixed parameters, the confidence threshold was set to an arbitrary value of 0.5, but what happens if we change that threshold? Let's say we consider detections with a score threshold of 0.95, we would have few but possibly very precise detections, leading to a low recall and a high precision. Instead, if we consider detections with a threshold of 0.05, we would have tons of detections with a lot of errors, possibly leading to a a high recall but a low precision.

Starting from a threshold value of 1.0, the more we move the threshold towards low confidence values the more detections are included and this should lead to an overall decrease of precision and increase of recall.

Technically, instead of moving the threshold, what it's done is that the detections for the class are ranked from most confident to least confident, then precision and recall values are calculated on an increasing list of detections, starting from a list containing only the first detection (the most confident) down to the list that contains all detections. The couples of values $(precision, recall)_i$ obtained at each step are plotted (with an interpolation algorithm not described here for brevity) in a curve like graph, and an approximation of the area under the curve is taken as *Average Precision*.

### COCO mAP @ 0.50 IOU (a.k.a. Pascal VOC metric)

The average precision, as described in the last paragraph, is the main metric that defines the detector performance on a class of objects but a model usually detects more than one class. The models trained on the DETRAC dataset should also detect buses, vans and others. The mean average precision, mAP, calculated with a IOU threshold of 0.50 (as defined in the "Hit or Miss" paragraph), is the simple arithmetic mean of the APs over all the classes, and assesses (though with some limitations that will be highlighted in the results section) the general detection performance of the model. It is used in Pascal VOC Challenges and it is one of the components of the COCO mAP metric.

### COCO mAP

The reason why the above metric has the "@ 0.50" part is that it is computed by considering hits or misses while matching detections with ground-truth boxes with a fixed IOU threshold of 0.50 between the two . Having a fixed threshold does not assess the localization performance of the detectors and

that's why the official COCO mAP metric considers the arithmetic mean of the mAPs at all thresholds from 0.05 to 0.95 with a step of 0.05.

The COCO tools, together with the Tensorflow OD API, may produce the mAP, the mAP@0.50 IOU, the mAP@0.75, for all classes and , with a slight modification also for each single class.

### 4.5.2 Evaluation results

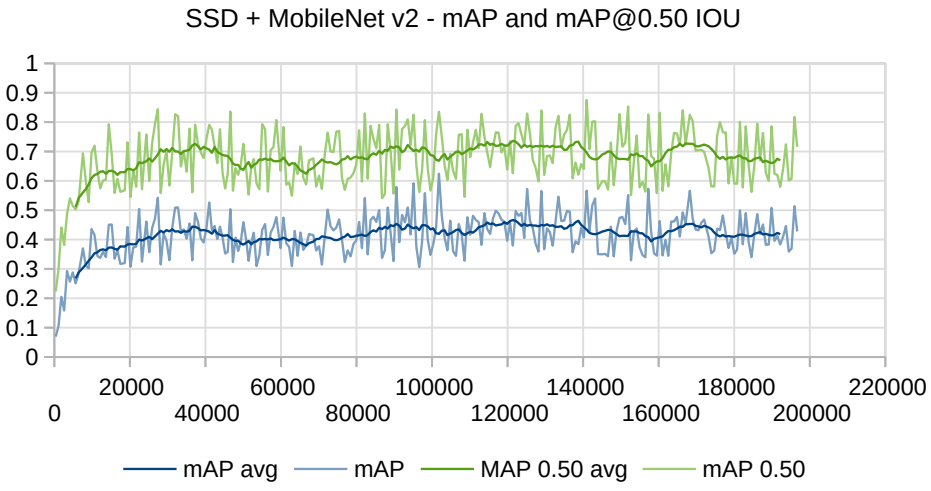
This section will show the results of the evaluation phases done during the training process. The results are the values of the loss function and it's components and the mAP metrics, calculated on 1/20th of the DETRAC eval set. For the sake of brevity only the total loss function and some of the mAPs will be shown.

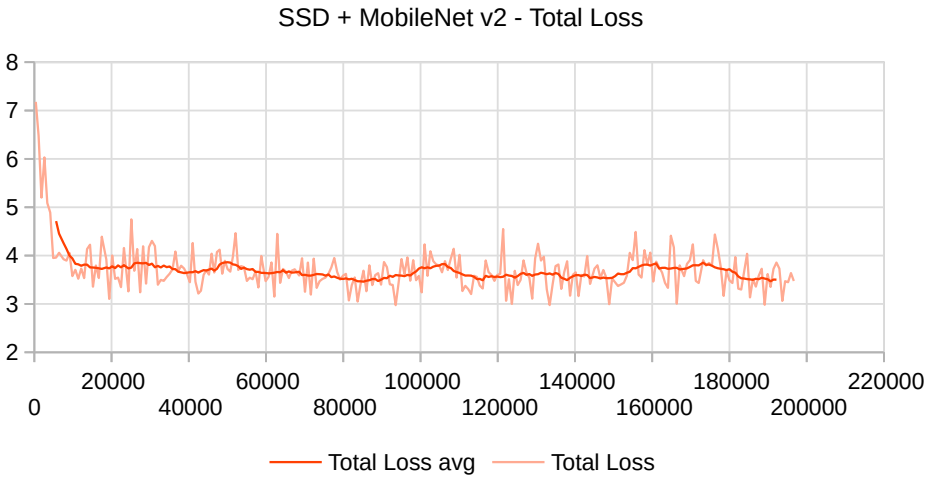
The results are plotted against the increasing training steps on the x-axis, where each training step represents an update of the neural network's weight according to the batch of samples, thus steps and training time of the models are proportional.

The evaluation runs every 10 minutes of training and each time an evaluation is run, the model, which has reached step $k$, is saved as a *checkpoint* and the loss and mAPs are produced.

#### SSD with MobileNet v2

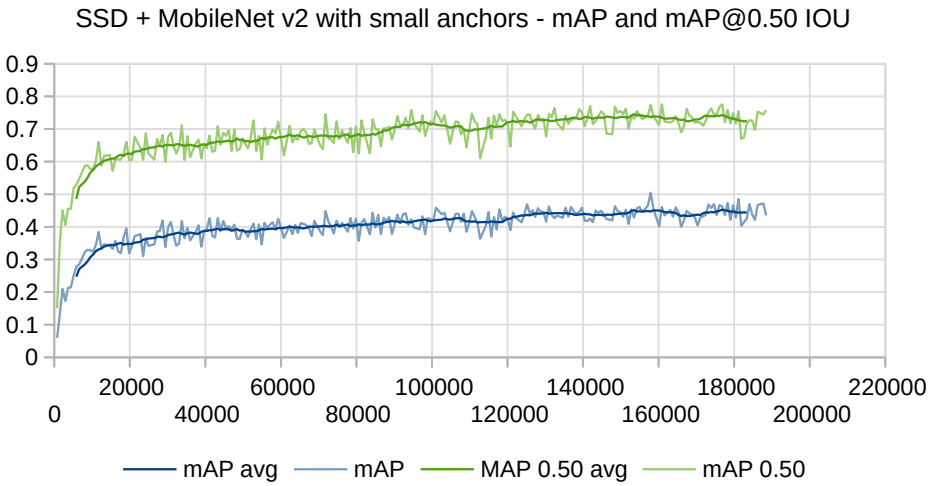
The following graphs show the evolution of the mAP, mAP at 0.50 IOU and total loss of the SSD with MobileNet v2 model during the training steps:

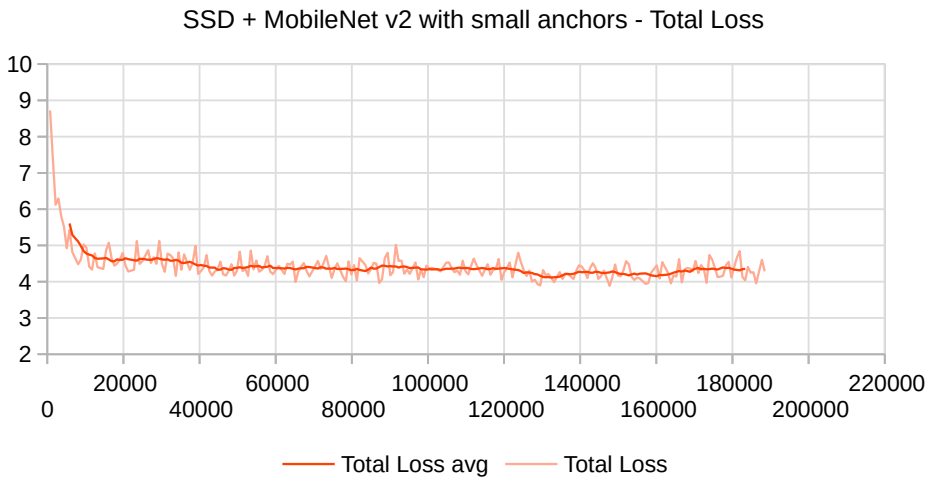





### SSD with MobileNet v2 and small anchors

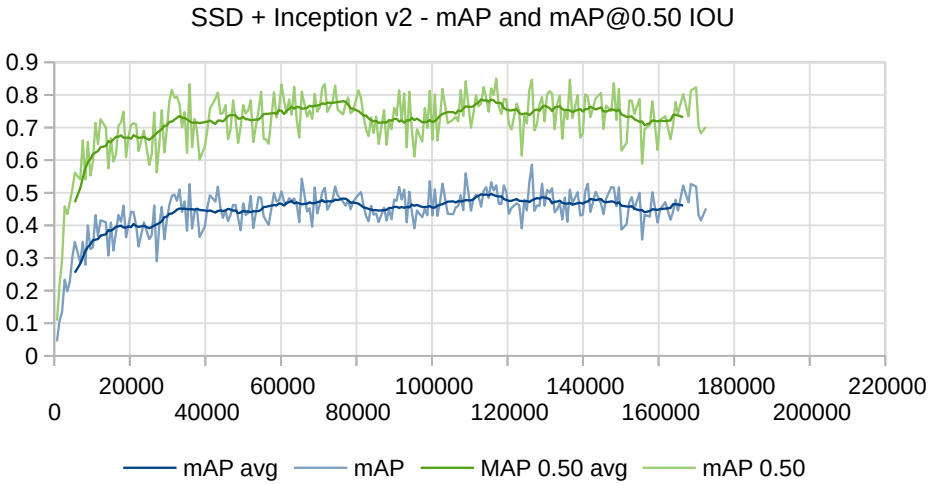
The following graphs show the evolution of the mAP, mAP at 0.50 IOU and total loss of the SSD with MobileNet v2 model with small anchor boxes during the training steps:

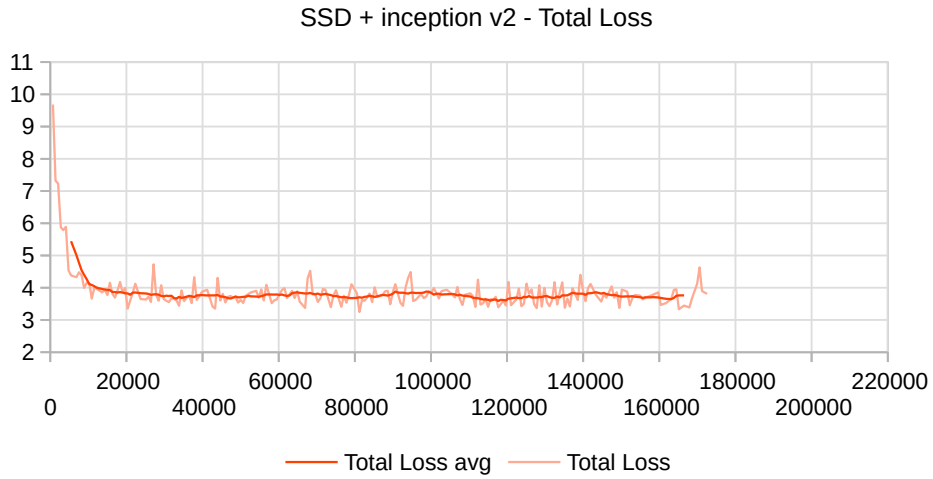





### SSD with Inception v2

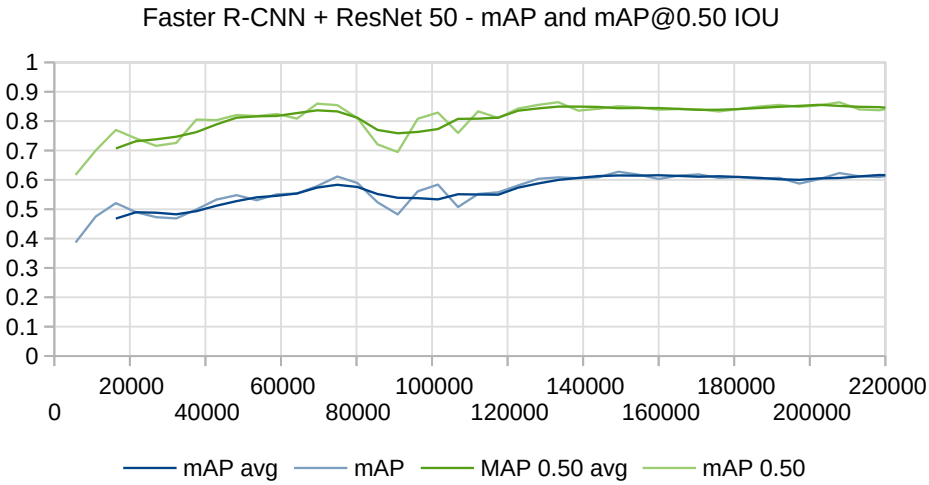
The following graphs show the evolution of the mAP, mAP at 0.50 IOU and total loss of the SSD with Inception v2 model during the training steps:

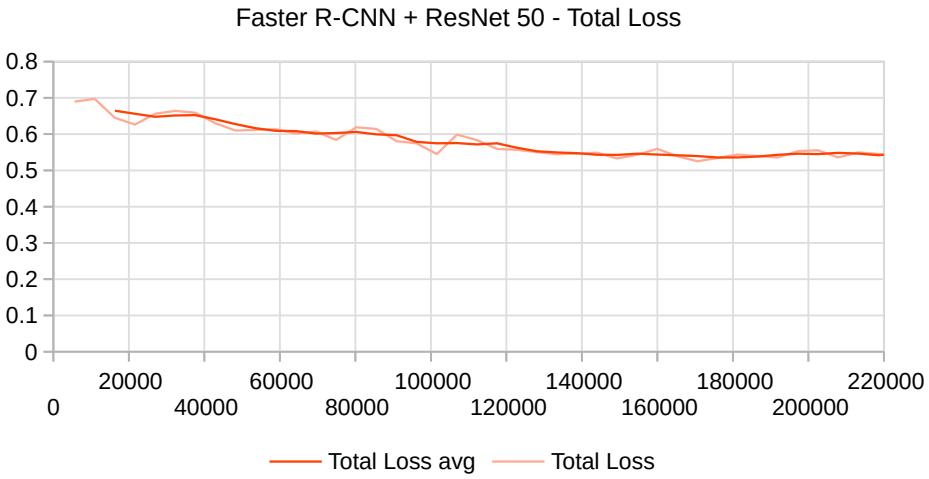





### Faster R-CNN with ResNet-50

The following graphs show the evolution of the mAP, mAP at 0.50 IOU and total loss of the Faster R-CNN with ResNet-50 model during the training steps:




Faster R-CNN + ResNet 50 - Total Loss
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
0
20000
40000
60000
80000
100000
120000
140000
160000
180000
200000
220000
Total Loss avg
Total Loss

# Chapter 5

# Framework Implementation

This chapter focuses on the software products of this work. The second part of the work in fact, possibly the most consistent, assessed the creation of a framework to perform road traffic analysis using vehicle detection and tracking data extracted from a connected IP camera's live feed.

The development work, actually, goes beyond this described task since the first idea of the company, as already mentioned in the introductory chapter, was to create a prototype of "smart adapter", i.e. a fully-configurable edge platformized for many different scenarios, from road traffic analysis, to pedestrian counting, or industrial appliances. In order to pursue these principles, the design of the framework follows the flow-programming paradigm with a modular plugin architecture. The advantages of this architecture together with a discussion on how this eases the shift between different scenarios will be presented in the following sections. Nonetheless, for the sake of giving a more thorough insight on the performance of the framework, the development and the analysis focused on a single scenario and, as the company had a live traffic camera available, the road traffic scenario was finally chosen.

## 5.1 Detpipe

Detpipe is the name chosen for the main component of the framework, the detection pipeline. The framework, in fact, consists of three components, the detection pipeline, a REST server and a dashboard web app, although the two latter components act merely as a control interface for the pipeline itself.

### 5.1.1 Architectural design

Detpipe's design follows the *flow-based programming* paradigm and it's inspired by frameworks like IBM's Node-RED and Spotify's Luigi. The first is a JavaScript framework, mostly used in IoT, which provides a visual-programming interface to connect hardware devices, APIs and online services to perform real time data feeds collection and processing, the latter is a Python framework that provides tools to manage big-data batch processing. They differ in their structure and intent but they share the same principle, they are both based on a pipeline to define the flow of operations applied to the data.

In Detpipe, the data-flow is a sequence of frames extracted from one or more connected cameras, that are processed with various modules in order to output annotated frames and statistics such as vehicles classification and counting. The operational configuration is represented by a JSON pipeline file, which defines a series of modules and a series of connections between them, called pipes. Source modules produce data that flows through pipes through other processing modules up to some final sink modules that will display frames or send statistics to a database.

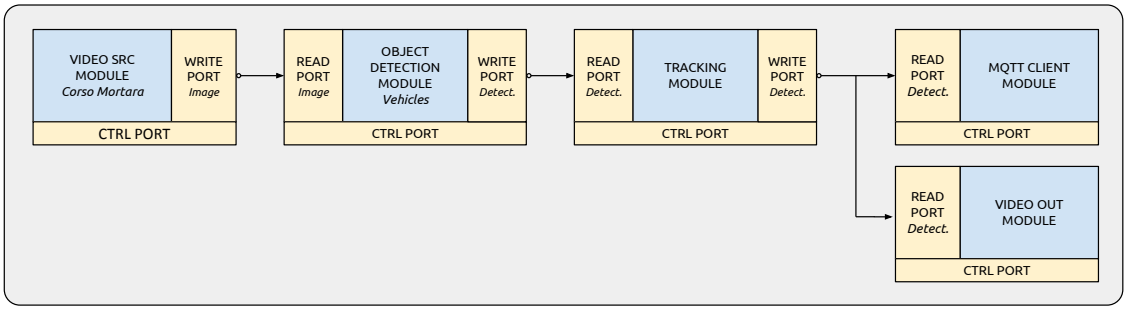



Figure 5.1. A simple pipeline configuration for detecting vehicles from a camera source module down to the output modules

The above image shows a streamlined pipeline with two camera sources connected to detection modules, tracking modules and finally two output modules.

#### Modules

Modules are the building blocks of the pipeline, and are composed of a main function block together with input and output ports. Depending on their characteristics, the modules can be divided into three categories: source

modules, such as a live camera feed, that only have output ports to which to write fresh data; sink modules, such as a video output or a database client, that have no output and simply show or collect the processed data; processing modules, such as an object detector or a tracker, which have both types of ports and will read data, apply come functions and forward the processed data to a subsequent module. This categorization is presented only for the sake of clarity since the framework sees all the modules the same way.

### Ports

Every module can define i/o ports on which it will receive/send data. They have to be defined in the module's configuration file and, once the framework is started, the relevant structures will be automatically created such that the module will transparently call the provided read and write functions without caring of the lower implementation. The ports are identified by a name, used in the read and write functions in order to select from where to read the data or to which port to write it, a port type, used to select the port implementation (more details on the implementations will follow), a message type and an encoding. The two latter fields are used to provide matching between connected ports (in order to identify configuration problems before the framework is started) and to automatically parse messages according to a dynamically imported type. In the current version of the framework, the message type refers to a Protobuf message class and the encoding can be either BINARY or JSON, according to the respective encodings defined in Protobuf library.

### Pipes

In Detpipe, pipes indicate the connections between modules' ports ad their implementation depends on the port type parameter specified for the in/out ports which they have to connect. The current version of the framework supports only one type of port/pipes which is ZMQ/xxx type, based on the ZMQ open source library, implemented in c++ with also a python interface, which uses TCP/IP or UNIX sockets as the underlying implementation to provide easy messaging between processes. In this specific case UNIX sockets are being used due to their reduced communication overhead. Each pipe is identified by an alphanumerical id which will be translated to a Unix socket file in a folder specified in Detpipe's main configuration file, according to a template specified in the same config.

The xxx part of the pipe type specifies the subtype, and the current versions of Detpipe supports three subtypes, a REQ subtype, a REP subtype and a PUBSUB subtype, directly mapped to the corresponding ZMQ REQ/REP and PUB/SUB socket types. REQ and REP pipes are bidirectional pipes that support a server/client behaviour, they are used for modules' control ports in the python control interface. A REP pipe acts like a server, it binds to a specific socket and waits for requests from a REQ pipes, then it replies to the requests and goes back to the listen state. A REQ pipe acts like a client, sending requests to a REP pipes and reading the received reply. Each REQ or REP pipe must alternate between reading and writing otherwise an error is raised. A PUBSUB pipe will be mapped to a PUB pipe if it's requested as write-only or to a SUB pipe if it's requested as read-only (they can not be bidirectional). They provide an MQTT like behaviour, i.e. a process writes to a PUB pipe and any connected SUB pipe receives the message, and they are used to implement the connection between modules. It's important noticing that, differently from REQ/REP pipes, PUB and SUB pipes are only non-blocking, so any message that is not read and that overflows the buffer size specified in Detpipe configuration will be dropped. This means that a source can inject frames in the pipeline and slower modules will not harm faster modules connected to the same PUB pipe but this also limits the syncing ability between parallels modules.

### 5.1.2 Running model

In order to get the best performance a multi-process approach had to be pursued and the following two solutions, which share in common the fact that each module runs in a separate process, were evaluated:

1. **Python multiprocessing:** This solution implies having a single main process launching child processes for the modules, and it was the first considered option. Such a solution would enable the use of communications methods in the official multiprocessing Python library but it would also require the implementation of a control mechanism in order to start and stop modules, and to restore them in case of errors.

2. **Separate processes + Supervisord:** This solution implies having separate completely independent processes, one for each modules, launched and controlled by the external application Supervisord. The advantage of this solution is that the all the process control is done by supervisord, which also exposes a cmd line interface and an XML-RPC interface for

starting stopping and interrogating the modules and a logging function for the stdout and stderr of the modules.

The chosen solution is the second one, so the running model is built around the use of supervisord. Supervisord requires the definition of what it calls "programs", i.e. the processes that need to be launched, in a configuration file which lists the name of the program, which is mapped to the id of the module in the pipeline, and the shell command which will be executed, which is mapped to a script that loads the virtualenv and then launches a python script with the module name and parameters. The supervisord configuration file is written by the framework each time a new pipeline configuration is loaded.

Once supervisord configuration is written and loaded, the modules can be started, stopped and interrogated for the status from supervisorctl shell interface, with supervisorctl utility, or directly in python with detpipe's own control interface which internally uses supervisord XML-RPC api.

### Module options

Each module can define a set of options representing runtime parameters such as the URI of a video and the crop and scale ratios for the video source module, the CNN model for the detection module, the thresholds, params and counter lines for the tracking module, and various other options. Each option can be changed through the python interface or through the web app and they can also be changed at runtime through the modules' CTRL ports.

### Pipeline initialization and start

When the pipeline is started, every module of the pipeline is initialized according to the pipeline JSON file, which contains the definitions of the pipes between the modules ports and the runtime options passed to each module. Before starting the modules run functions, Detpipe first performs some validation checks against connected pipes, checking if the ports connected by each pipe have the same port type and msg type, and against the options, which have to be parsable according to the types declared in the modules' properties. The modules will start only after these checks are performed and after the options have been parsed.

#### Module stopping and signal handling

Each module runs in a separate daemon child thread and should perform it's functionality in a cycle, checking against the "self.stopped" variable. Whenever the main threads receives a stop signal it sets the variable to False and waits 10 seconds for the child thread to exit. If the child is still running, the main thread exits and the child gets killed. Running the module's main function in a child thread saves it from the problem of recovering from interrupted function calls. Indeed, the signaling is managed by the parent.

### 5.1.3 Extensibility

Extensibility is a key component of the framework design, which provides a plugin-like architecture which eases the creation and integration of new modules or pipe types into the framework.

#### Modules

The framework already provides the modules needed for a full detection pipeline but new modules can be easily built and plugged in thanks to a series of defined interfaces and configuration schemes. Any module in Detpipe must extend a *BaseModule* class, which transparently provides all the base functionality ranging from module initialization to i/o read/write and option parsing, override a *run* function with the module's main loop and eventually override some control functions if advanced custom behaviour is needed. A module also has to provide a configuration file defining input/output ports and their respective types (pipes and port types will be discussed later), accepted options and their types. Finally, in order to be seen by the framework, any module has to be defined in the main Detpipe configuration file, with a section that defines the module's name and its packages for dynamic import.

#### Pipes

Different pipes implementation can also be provided, they need to be registered by calling a register function provided by the framework and provide a name with which they will be identified in the modules' properties. Currently, the provided PyZMQ pipes are registered with types "ZMQ/PUB-SUB', "ZMQ/REQ"' and "ZMQ/REP", alternatives implementations will have to provide the three basic pipe types implementations, some protobuf encoder and decoders, and that's all.

## 5.2 Detpipe modules

### 5.2.1 Source module

The video source module is responsible for decoding the video stream and injecting the frames into the pipeline. It supports RTSP live streams, such as the stream coming from a connected IP camera, but it can also be used with recorded video files for performance testing or debugging purposes. The source module uses OpenCV VideoCapture object and can be set from the main configuration to work in two different ways, with gstreamer support or without it.

Gstreamer is a media applications framework, similar to the ffmpeg library, which is used by OpenCV's VideoCapture as an alternative backend library to capture and decode the frames. It is used on the Jetson board to enable the hardware accelerator for stream encoding and decoding, which is quite important to get better speed performances. In order to use it, OpenCV has to be built with gstreamer support and, although Nvidia L4T came with hw accelerated gstreamer and prebuilt OpenCV, a manual build had to be performed since, unexpectedly, the latter wasn't built with the former's support.

Moreover, to use gstreamer hw acceleration the VideoCapture object has to be initialized with a special syntax, a gstreamer pipeline, which is gstreamer's way of defining the flow of plugins that will produce the videoframes, starting from a source down to OpenCV's frame buffer. The pipeline templates used for RTSP streams and files, located in Detpipe main configuration file for easy modification and tweaking, are the following:

```
1 # Gstreamer rtsp pipeline
2 rtspsrc location={location} latency=2000 protocol=
    GST_RTSP_LOWER_TRANS_TCP ! rtph264depay ! h264parse !
    omxh264dec ! nvvidconv{crop_str} ! video/x-raw, format=(
    string)BGRx{scale_str} ! videoconvert ! appsink
3 # Gstreamer file pipeline
4 filesrc location={location} ! qtdemux ! h264parse !
    omxh264dec ! nvvidconv{crop_str} ! video/x-raw, format=(
    string)BGRx{scale_str} ! videoconvert ! appsink
```

The *location* will be filled with the stream's URI or the file's path, while *crop_str* and *scale_str* will eventually contain the top, left, bottom, right parameters if cropping is needed, and width and height if scaling is needed, according to the runtime options passed to the module. Cropping is very useful when the available stream has a very large field of view, like in the

case of the stream provided for this work, and scaling can enhance speed performance, as it will be seen, and performing the two operations directly in gstreamer is more convenient.

In case gstreamer is not supported or it has been turned off in Detpipe configuration, the standard VideoCapture is used, and the cropping and scaling, whether defined as options, will be still applied programmatically.

### 5.2.2 Object Detection module

The object detection module is the core of the detection pipeline, it receives non-annotated frames and adds annotations for all the objects it finds using a neural network, called graph in Tensorflow, produced with the Tensorflow OD API. The options accepted by the modules are presented in section **??**.

The module loads the Tensorflow graph specified in the *frozen_graph_path* option which is a *.pb* protobuf binary file, as per tensorflow specifications, and also looks for a label map file named which specifies the mapping between the integer class indexes produced by the graph and the names of the corresponding classes. Once the model is loaded, the module starts reading frames, feeding the neural network, processing the outputs and sending them to the next connected module.

The output of the graph is composed of 3 vectors, also called tensors: a boxes vector, a scores vector and a classes vector. All the three vectors have size $N$, where $N$ indicates the number of detections, and for each detection $i$ there is a box, a score and a class from the correspondent vectors. The box is itself a vector of with the 4 elements *ymin*, *xmin*, *ymax* and *xmax*, the *score* indicates the confidence of the prediction while the *class* indicates the type of object detected. The output for detection $i$ is depicted in figure 5.2.

While experimenting, an attempt in using the TensorRT inference engine included in Tensorflow was made and, although the memory footprint doubled, the speed performance did not change. A possibile explanation is that TensorRT scripts included in Tensorflow are a work in progress and since Tensorflow 1.11 (the version used in this work) they have changed quite a bit. Nonetheless the option to run the model with tensorrt engine has been left.

The detection module also offers the possibility to select one or more ROIs. They are passed as options in the form of a string of polygons. If the polygons string is provided, the image is cropped down to exactly fit all the polygons and, if mask_image is true, the regions outside of the polygons is filled with a black mask. The resulting cropped and eventually masked image is then

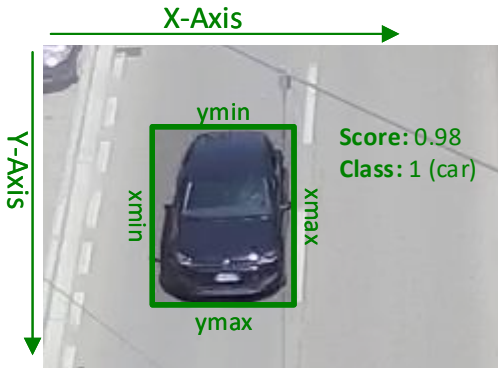



Figure 5.2. Output box, score and class for detection $i$

fed to the network which will produce the detections, which will be scaled back to the original image size and written to the following module.

### 5.2.3 Tracking module

The tracking module is responsible for identifying detections by associating boxes surrounding the same object from different frames throughout time. The desired output of the tracker is a unique *track_id* for each object, from when it enters the scene to when it leaves it. The tracker algorithm has to perform in an online scenario so it should produce track_ids every time it receives a frame, based solely on the current detections and on his past history. As usual, the configuration options of the tracker module are presented in section **??**.

The problems that an online tracking algorithm based on detections may encounter are mainly due to poor detectors, occlusion, dense detections with very close objects and missed detections between frames. Some online tracking algorithms, such as the algorithm used in this work, are also susceptible to low framerates, since in an online scenario this would mean high dislocation of the objects between a frame and the succeding one. Fortunately, for the vehicles use case, having a high camera point of view helps to mitigate the occlusion and dense detections problems, the other problems, instead, are mitigated by using a fast and precise detector.

For the tracking module an implementation of the IOU tracking algorithm [1], adapted to work in an online scenario, was developed. The IOU tracker exploits the recent advancements in the detection field in order to favour simplicity over complexity, granting results able to compete with the more

sophisticated trackers at a fraction of the cost. The IOU tracker, in fact, is highly dependent on the detector performance and, in order to give good results, it needs precise detection at a relatively high framerate.

Since the framework has to run on a low powered edge-device, computational performance was one of the main factors that led to the choice of the IOU tracking algorithm. One of the key elements that makes the tracker so performant, in terms of computational footprint, is that it does not require any kind of visual information from the image and performs the associations based only on the set of detection boxes, on which a nearest neighbour search with a very lightweight measure is performed. The distance between two boxes is measured with their IOU distance, hence the name of the tracker, and the definition is shown below.

### The algorithm

Technically, the tracker simplicity stands in the fact that all it does when it receives a new frame is to perform a nearest neighbour search between the detections on the new frame and the last detections corresponding to active tracks, based on the IOU distance (the concept of active tracks will be clarified in the following paragraphs).

In the algorithm the first note regards the distinction between active tracks $T_{act}$ and final tracks $T_{fin}$. The set of active tracks contains all the tracks for which the objects is supposed to be still in the scene, and it's the set on which the search is performed. Every time an active track can not be matched (i.e. the closest detection found has a IOU lower than a configurable threshold $\sigma_{IOU}$), a missed frames counter associated to the tracked is incremented. If the counter is greater than a $\sigma_{miss}$ threshold, the active tracks is closed and moved to the set of finalized tracks. A second note regards the tracks themselves, since the tracker keeps a history of the detections of the track (the history length is configurable) every time a search is made it is obviously made on the last available detection in the track.

One of the limitations of the tracker is the fact that it uses a greedy heuristic for the nearest neighbour search, other trackers like SORT [20] use the Hungarian algorithm to perform the best set of associations. Another limitation is that it does not consider the objects velocities to predict the position to be matched. Albeit having these limitations, the tracking performance proved to be satisfactory in the real world use case and having a simpler implementation helps reducing the impact on the whole pipeline performance, which is completely unnoticeable in the current version of the framework.

**Algorithm 1** Tracker module

**Inputs:**
  $D_{new} = \{d_1, \ldots, d_n\}$ ▷ New frame's detections
  $T_{act} = \{t_1, \ldots, t_m\}$ ▷ Active tracks

**Initialization:**
  $T_{act} \leftarrow \varnothing$
  $T_{fin} \leftarrow \varnothing$

**function** NewFrame($D_{new}, T_{act}$)
  $D_f \leftarrow \{d_i | d_i \in D_{new}, d_i \geq \sigma_l\}$
  **for** $t_i \in T_{act}$ **do**
    $d_{last} \leftarrow lastdet(t_i)$ ▷ Last detection of the track
    $d_{best} \leftarrow d \mid IOU(d, d_{last}) = \max_{d_j \in D_f} IOU(d_j, d_{last})$
    **if** $IOU(d_{best}, d_{last}) \geq \sigma_{IOU}$ **then**
      add $d_{best}$ to $t_i$
      remove $d_{best}$ from $D_f$
    **else**
      $missed_{t_i} \leftarrow missed_{t_i} + 1$
      **if** $missed_{t_i} \geq \sigma_{miss}$ **then**
        move $t_i$ from $T_{act}$ to $T_{fin}$
      **end if**
    **end if**
  **end for**
  **for** $d_k \in D_f$ **do** ▷ Add remaining dets to new tracks
    $t_{new} \leftarrow newtrack(d_k)$
    add $t_{new}$ to $T_{act}$
  **end for**
  **return** $T_{act}$
**end function**

### Counters

The tracker module does not only generate track ids, it also implements vehicles counting through counter segments. Each segment is represented by two points and indicates a crossing line, producing two sets of counters, one per direction, which keep track of the number of vehicles crossing in both directions, both total and by class.

### 5.2.4 MQTT Client module

A framework for performing road traffic analysis, or any other kind of analysis, has to provide a means to store the produced analytic data but, in order to keep the board as light as possible, it's been decided to send the data to an external database. In order to do so the framework includes a module that sends messages with the relevant data to a server with a broker and a database at regular intervals, using the MQTT protocol.

MQTT is a messaging protocol based on a publisher/subscriber pattern where a publisher sends messages with a topic to an intermediary peer called broker which, in turn, will forward the message to all the connected subscriber peers who subscribed to the same topic. Here the module acts as the publisher while a subscriber will read the data and store it on the database.

The behaviour of the module is quite simple, it parses and accumulates the data received from the pipeline, including detected objects by class, counted objects by class and by counter segment and performance data such as avg detection and loop times, and, after a certain time has passed, it creates a JSON message with all the cumulative data and sends it with a Paho MQTT client library. The stack and the behaviour of the receiving side is described in section 5.5. Obviously the publish time interval, broker address, topic and other options are configurable and a list of the module options can be found in section **??**.

### 5.2.5 Video output module

The video output module has a double function, the first function regards outputting the annotated video frames to a window, if a screen is connected to the host, the second function regards the output of the same annotated video on an RTP stream with an OpenCV VideoWriter using a Gstreamer pipeline with a udpsink.

The drawing functions for the boxes and for the tracks use OpenCV and numpy vectorized operations in order to run with an unnoticeable computational cost; for a comparison, the current box drawing functions runs more than 294 times faster than the equivalent function in tensorflow object detection api which uses PIL.

Regarding the UDP stream, it's a push stream produced by an OpenCV VideoWriter initialized with the gstreamer pipeline below. Since it's a push stream, where the framework directly sends UDP packets to the receiver,

the stream can not be viewed if the receiver is behind a NAT, unless port-forwarding is set.

```
# Gstreamer UDP stream pipeline
appsrc ! videoconvert ! omxh264enc ! mpegtsmux ! rtpmp2tpay !
    udpsink host={host} port={port}
```

The UDP stream feature, though, is experimental since in the current implementation each frame is pushed to the VideoWriter as soon as it arrives while a proper final implementation would need a separate thread with queues and a buffer, writing at a constant predefined framerate.

## 5.3 REST API

The REST API is built as a side project to provide an alternative control interface to be used directly, through a browser and the built-in Swagger UI, or indirectly through the dashboard web app. It's written in Pyhton using Flask and Flask-restplus, the latter being an open-source module to ease the development of a REST API by offering some additional functionality and decorators for the API docs. The choice of the framework used for the server development was firstly constrained by the fact that detpipe's control interface is in Python so using the same language for the server would have been easier and more straight-forward and finally constrained by the performance factor, which, for a simple REST API, made the decision between the two most used Python web frameworks, Flask and Django, easily fall to the first one.

The Swagger-UI is a very nice addition provided by the Flask-restplus library and consists of a web UI, accessible by navigating to the root of the rest server, automatically mapped to all the REST resources methods, with an easy Try it out - Execute interface which is helpful for debugging and for performing simple tasks without going through the web app.

The REST Flask application can be run in two modes:

1. **Debug mode:** The REST API is run on a Werkzeug server instance, the debug server distributed with Flask, and does not require any particular configuration apart from selecting a port and running the main *app.py* file. The debug deployment setting though is highly discouraged in a production environment due to performance and security issues.

2. **Deploy mode:** The REST API is run with a WSGI entry point (Web Server Gateway Interface is a protocol which describes communications

between a server and a Python web application) on a supported server. The current implementation uses uWSGI server, set to accept connections on a Unix socket and reversed-proxied through NginX, which is configured to forward all HTTP requests on port 8080 to the socket on which uWSGI is listening.

On the TX2, the REST API runs in deploy mode, and all the additional software and configurations were put in place.

## 5.4 React Control Dashboard

One of the interests of the company was to have a web UI to control the detection framework in a user friendly way, so that a non-developer could modify options such as the detection model, the camera source URI, the visualization options and others in a form-like structure. This requirement collided with the initial idea of developing a flow-based interface with automatically loaded modules and options but the company preferred the user friendly interface solution which led to the final version of the control dashboard.

The dashboard is written in JavaScrpipt ES6 with Facebook's React framework. The advantage of using React is that it uses a web programming paradigm which is based on the concept of states, where each view of the web app is linked to one or more state variable and gets automatically re-rendered if a new state triggers a change in the relevant view. A simple example might be the update of a *<ul>*: with traditional Javascript you would have to select the ul and manually add or remove children whenever you fetch an updated version of the list; with React, instead, you would have to define a render method and link it to the component's state (e.g. having a javascript list in the state and outputting an *<li>* for each member of the list would do the trick), then simply substituting the state with a new state with the updated list would automatically trigger an update of the <ul> DOM element.

The web app is designed as a client side single page application with all the necessary javascript modules automatically bundled during the build of the deploy version. The app is downloaded by visiting the index page and communicates asynchronously with the REST server using the Fetch API; this, in contrast with a dynamic-pages approach, slightly reduces the burden on the Jetson board (less requests) while offering a smoother user experience on the client (no page loadings).

### Flows

Implementing the dashboard on top of detpipe's pipeline configuration required an additional level of abstraction, so the concept of flows is introduced. A flow is a series of connected modules that identifies all the modules of a detection stream, from the video source module through the detection module, the tracking module and finally to the output modules like MQTT and video output.

Each flow is identified by the camera source that feeds the flow, and this is reflected in the dashboard design, where a side menu lists all the camera streams in the pipeline. Selecting a camera screen loads the dashboard body, which presents the options, divided into sections, where each section is mapped to a module (source, detection, tracking, mqtt, window). The tracking, mqtt and video output functions can be disabled through the dashboard and the REST application will trigger a modification in the flow, eliminating the disabled modules and adjusting the pipe connections.

New flows can be added by clicking on the "Add new source" option in the side menu and following a wizard which prompts for the options required for the new source in a series of modals.

### Dynamic dashboard

The dashboard body, with the sections and all the options, is actually a controlled form and all the fields are dynamically generated based on a JSON description which is downloaded when the app is loaded. The JSON object that represents the dashboard is divided in sections and each section has a list of options which get translated into inputs, check-boxes and drop-downs and radio-groups based on their type property and provide defaults, placeholders and tooltips based on other corresponding properties. The advantage of this approach is that the interface can be changed instantly without having to rebuild and redeploy the web application.

The dashboard JSON additionally passes through the Jinja templating engine in order to dynamically populate the lists of models and files (more info on this in the following paragraph).

### Usability

The design aspect was not a requirement in this work but usability was actually a main concern while developing the web app so some UI choices were taken in order to enhance it.

Firstly, a UI framework was used in order to provide UI elements with a modern behaviour, as it would be expected by an end user, and the chosen framework is Semantic UI React, especially thanks to it's simplicity. Simplicity, though, comes at a cost, in fact, some UI components used in the dashboard had to be developed by creating new components extending the behaviour of some Semantic UI simpler components. An example are the modals used in the dashboard, which extend the provided modals by adding multi-panel navigation, multiple-choice dialogs, loading screens that can not be discarded and custom confirmation messages.

Secondly, since the end user is supposed to be a non-developer but a developer might want to use the web app too, without resorting to the python control interface, the options in the dashboard JSON can define advanced or develop flags, which directly modify the way they are rendered in the dynamic dashboard. Setting the advanced property to true for an option hides it from the user, unless the advanced toggle in the dashboard is activated, while setting the develop property to true hides it even if the advanced toggle is set. To show all the options, including the development ones, the user can load the app with the devel flag set in the query string.

Lastly, not all options might be clear so placeholder and tooltips properties can be added in the dashboard JSON. The placeholder property can be defined for options of type input and it's showed if the text input is empty. Tooltips, instead, can be defined for any option field and are shown whenever the user hovers the corresponding elements with the mouse cursor.

## 5.5 Integration with InfluxDB (or other)

## 5.6 Profiling

Some words about the profiling work, the yappi profilers integrated and called with custom messages (maybe citing the problems with udp source), the use of numpy, the use of rewritten drawing function (first rewritten with opencv, then rewritten with numpy)

# Chapter 6

# Results

This chapter will describe the results of this work, starting from the test dataset annotation, following on to the testbed description, and closing with the actual results in terms of speed, detection performance and tracking performance on a real use case.

## 6.1 Test sequence annotation

In order to provide an insight on the real use case performance and to evaluate the models on unseen samples with a very different setting from the training data, a 15 minutes sequence was recorded and annotated using a combination of tools, a script written for the purpose and almost two days of handwork. The sequence is a 720p crop of a 4k stream from a CSP Hikvision camera on c.so Castelfidardo, Turin. It was recorded on a Friday at 12:55pm.

The software used to annotate the sequence is CVAT, an open-source software available as a docker application while the script used for the cleaning step is written in Python and works on the XML annotations dumped from CVAT.

The product of the annotation is a sequence of *16669 frames* with a total of *88439 annotated ground truth boxes*, and the process followed three steps that will be described hereafter.

#### Step 1: TensorFlow auto-annotation

The first step consisted of performing a TensorFlow auto annotation of the video directly with CVAT. The model used for this first processing step is a Faster R-CNN with Inception ResNet v2 Atrous backbone, trained on the

COCO dataset, coming from the Object Detection API Model Zoo. It's a very slow but very precise model but it is still not perfect. It produces a considerable number of wrong detections and it's predicting boxes for 80 different classes.

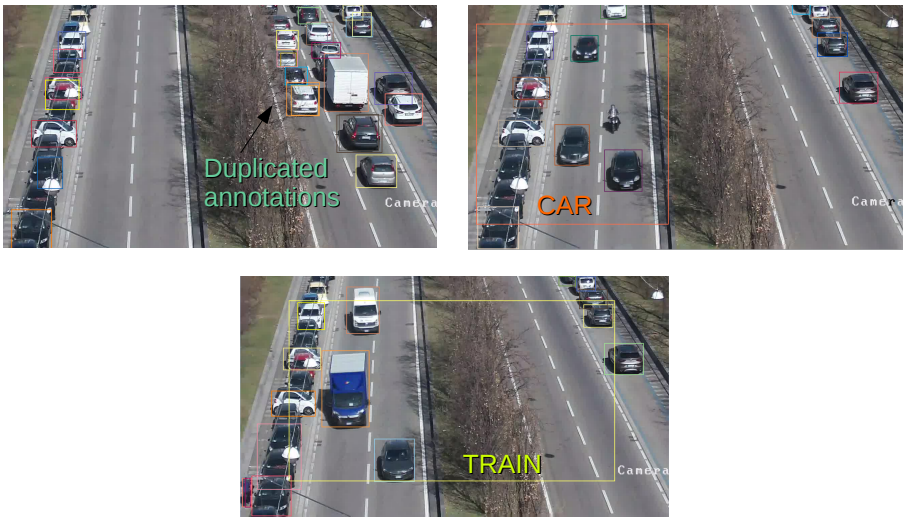



Figure 6.1. Errors from the TF auto annotation step

### Step 2: Cleaning script

Eliminating all wrong detections by hand on more than 16 thousands frames, right after the auto annotation process, was too time-consuming so a script to perform a cleaning step was written. The script parses the CVAT XML annotations and execute three types of filtering. Firstly, it finds special polygons called ignored regions and eliminates all detections inside those regions (in this case it eliminates all parked cars from all 16669 frames). Secondly, it also filters detections by label, eliminating unwanted class labels. Lastly, it also cleans any annotation bigger than a certain area threshold.

Since the majority of vehicles are cars, any annotation which is not a car was eliminated, even if labels regarded other vehicles (bus, trucks, etc.). This really sped up the manual process since checking each annotation on each frame against a multitude of vehicles classes was too time-consuming and error-prone (for each detection in each frame, is that box a car, or did the auto-annotation label it as a van or a truck or vice-versa?). There were very big cars and other unwanted objects detected in the middle of some frames

and they were eliminated by this last step. The cleaned xml annotations were then loaded back into CVAT for the final manual step.

### Step 3: Manual corrections and annotations of non-cars

The output of the cleaning step is a video with annotations of cars only, with remaining errors that had to be addressed manually. Errors regarded duplicated and missing detections when cars were lining up, and this happened quite often due to semaphores in close proximity, missing detections when cars were partially covered by tree branches, other vehicles labeled as cars and wrongly localized boxes. This manual last step took almost two days and involved deleting all duplicated or wrong detections, correcting wrongly localized boxes and adding annotations for undetected cars and for all the remaining vehicles. More than 8000 boxes were deleted one by one by hand, and almost 12000 boxes were added, partly by hand and partly with interpolation methods.

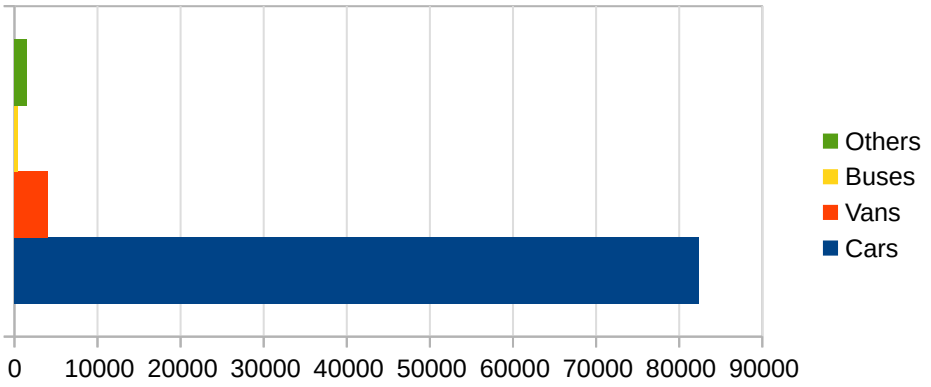



Figure 6.2. Number of annotations, divided by class, in the annotated test dataset

The resulting sequence has 88439 annotation boxes with 82418 annotations of cars, 4063 annotations of vans, 437 annotations of buses and 1521 annotations of other (trucks, special vehicles and pickups). This imbalance towards cars, which make up more than 93% of the whole traffic, led to the adaptation of the COCO mAP metric used in the detection performance evaluation, and it's implication will be further analyzed in the following sections.

## 6.2 Benchmark

The following table shows the running speeds of the models on the Nvidia Jetson TX2 board.

FPS of the models at two different resolutions

| - | 720p | 300x300 |
|---|---|---|
| Mobilenet v2 | 5.60 | 6.93 |
| Inception v2 | 14.20 | 19.82 |
| Faster Rcnn | 0.6 | 0.7 |

## 6.3 Detection performance analysis

This section will concentrate on the performance side of the testing of the models. It will analyze all the considered models and their evolution during their training (more on this in the paragraphs below), resulting in an extensive set of tests, the definition of a new metric, derived from the existing ones, and the presentation of various consideration on some particular conditions that were highlighted during the process.

The starting point is the set of evaluations presented in section 4.5.2. In fact, during the training process the models were evaluated every 10 minutes on $1/20^{th}$ of DETRAC Eval dataset and the resulting precision metrics, the mAPs, were produced and plotted. Those evaluations, though, were only done on a fraction of an already small dataset. The eval dataset (described in detail in section 4.3.1) contains only 6246 frames, few compared to the 16669 of the annotated test set, and $1/20th$ of it only comprises 312 frames. An evaluation executed on only 312 frames is quite unattendable so a new set of detection tests, both on $1/5th$ of the DETRAC eval set and on $1/5th$ of the hand-annotated Turin sequence, was performed.

### 6.3.1 Checkpoints and performance metrics

Before getting right into the test results, with tables and graphs containing step values, mAPs, mAPs @ 0.50 and weighted mAPs, a recap is definitely in order. The first concept to be clarified is the one of *checkpoints* and *steps*. It will then be followed by a recap on the used *mAPs* and some *important considerations* about the ones used throughout the tests.

### Checkpoints

During the training process the model's weight and parameters are updated at each training step and every time an evaluation is made (once every 10 minutes) the model is saved. Each save produces what in TensorFlow jargon is called *checkpoint* which is a representation of the model's state at the training step at which it was produced. A final neural network model is simply one of the checkpoints, exported to a binary format used for Tensorflow inference at runtime. Choosing one "best" checkpoint for all of the considered models is the final objective of the two analysis (detection and tracking) presented in this chapter.

Checkpoints with larger steps represent the models after a longer training time and one may think that they ought to be more precise. This is not the case though, since, as seen in the charts in section 4.5.2, even if the values of the moving average of the mAPs tend to stabilize, the exact values continue to fluctuate with differences that can exceed 20 percentage points for very close steps of some models (e.g. SSD + MobileNet v2 at step 156,476 has a mAP of 0.33 while at step 157,236 has a mAP of 0.57). The fact that the precision values have this high fluctuations is the main reason that motivated a more extensive set of tests on the detection performance

### Performance metrics

Throughout the tests, the COCO mAP metrics were used to quantify the detection performance of the models' checkpoints. The reader should familiarize with the description of the metrics and the simplified overview of its calculation provided in section 4.5.1, especially with the concept of IOU and detections matching, since this paragraph will add some considerations and also provide a new metric derived from the original ones.

The first consideration regards the IOU matching and the fact that the so called COCO mAP@0.50 IOU is the main metric chosen to represent the models performance throughout the tests. This metric, as described in the metrics section, considers matchings at a fixed IOU threshold of 0.5 and the reason why it is used in place of the standard COCO mAP is that localization precision is not the main concern of this analysis. The main aim of this work is to provide fast models to analyze and count traffic so calculating matchings at IOUs greater than 0.50 is good enough. Additionally, matching against stricter IOU thresholds would need very precise ground truth boxes around the objects, which is true for the UA-DETRAC annotations and less true for the annotated Castelfidardo sequence.

The second consideration regards the *introduction of a weighted derived metric*, introduced due to the traffic bias towards cars and the fact that the standard COCO metrics consider the total mAP values as a simple arithmetic means of the mAPs by class, without considering the numbers of samples on which they are calculated:

$$mAP = \frac{mAP_{car} + mAP_{bus} + mAP_{van} + mAP_{other}}{4}$$

This means that if a specific model checkpoint has a very high mAP(@0.50) on cars but lower mAP(@0.50) on buses, vans and others, it will still have a low general mAP(@0.50), even if the cars represented the most numerous class. Since 93% of the traffic in the 15 minutes of annotated sequence in Turin was made up of cars, the counting performance should prioritize that class.

The derived metric will be called Weighted COCO mAP and in the tests will be called WmAP for short, and it considers the number of samples used for the evaluation:

$$WmAP = \frac{mAP_c \times n_c + mAP_b \times n_b + mAP_v \times n_v + mAP_o \times n_o}{n_c + n_b + n_v + n_o}$$

### 6.3.2 Training evaluations revisited

As already said, the mAPs collected during the training process regard $1/20_{th}$ of the DETRAC-Eval set, consisting of only 312 samples, so the first extensive evaluation tests were run on the same DETRAC-Eval set, this time evaluating $1/5_{th}$ of the dataset, in order to highlight differences, if any, on mAPs evaluated on a bigger portion of the same data. The tests were run on the Top 50 available checkpoints of the four models, according to the standard mAP evaluated during the training evaluations, and the Top 5 for SSD + MobileNet v2 and SSD + Inception v2 are reported here.

| | | DETRAC $1/20_{th}$ | | DETRAC $1/5_{th}$ | |
|---|---|---|---|---|---|
| Model name | Step | mAP | mAP@0.5 | mAP | mAP@0.5 |
| SSD + MobileNet v2 | 95078 | **0.591** | 0.825 | **0.459** | **0.727** |
| SSD + MobileNet v2 | 90539 | 0.578 | 0.842 | 0.427 | 0.710 |
| SSD + MobileNet v2 | 168256 | 0.566 | 0.825 | 0.402 | 0.668 |
| SSD + MobileNet v2 | 98096 | 0.558 | 0.806 | 0.433 | 0.695 |
| SSD + MobileNet v2 | 27412 | 0.542 | **0.844** | 0.415 | 0.676 |

| Model name | Step | DETRAC $1/20_{th}$ mAP | DETRAC $1/20_{th}$ mAP@0.5 | DETRAC $1/5_{th}$ mAP | DETRAC $1/5_{th}$ mAP@0.5 |
|---|---|---|---|---|---|
| SSD + Inception v2 | 126461 | **0.586** | **0.846** | 0.458 | 0.728 |
| SSD + Inception v2 | 108932 | 0.559 | 0.841 | 0.472 | 0.762 |
| SSD + Inception v2 | 99423 | 0.535 | 0.812 | 0.498 | **0.776** |
| SSD + Inception v2 | 115686 | 0.532 | 0.820 | 0.479 | 0.766 |
| SSD + Inception v2 | 125791 | 0.532 | 0.815 | **0.512** | 0.774 |

### 6.3.3 CVAT evaluations

It has just been shown that the training evaluation on $1/20_{th}$ of DETRAC-Eval have large gaps compared to the same evaluations run on the larger dataset, so basing the choice of the best checkpoints solely on those evaluations is not attendable. In this respect, the following evaluation tests were performed on $1/5_{th}$ of the annotated Turin-Castelfidardo sequence (a total of 3,333 frames) for the same Top 50 checkpoints from the training phase. As a side not, some smaller tests were performed for some checkpotins on the whole Turin dataset (16,669 frames) and showed no difference with the $1/5_{th}$ partition.

| Model name | Step | DETRAC $1/20_{th}$ mAP | DETRAC $1/20_{th}$ mAP@0.5 | C.SO CASTELF. mAP | C.SO CASTELF. mAP@0.5 |
|---|---|---|---|---|---|
| SSD + MobileNet v2 | 95078 | **0.591** | 0.825 | 0.189 | 0.358 |
| SSD + MobileNet v2 | 90539 | 0.578 | 0.842 | 0.198 | 0.424 |
| SSD + MobileNet v2 | 168256 | 0.566 | 0.825 | 0.218 | 0.427 |
| SSD + MobileNet v2 | 98096 | 0.558 | 0.806 | 0.252 | 0.480 |
| SSD + MobileNet v2 | 27412 | 0.542 | **0.844** | **0.257** | **0.551** |

| Model name | Step | DETRAC $1/20_{th}$ mAP | DETRAC $1/20_{th}$ mAP@0.5 | TURIN C.so Castelf. mAP | TURIN C.so Castelf. mAP@0.5 |
|---|---|---|---|---|---|
| SSD + Inception v2 | 126461 | **0.586** | **0.846** | 0.177 | **0.380** |
| SSD + Inception v2 | 108932 | 0.559 | 0.841 | 0.183 | 0.351 |
| SSD + Inception v2 | 99423 | 0.535 | 0.812 | **0.191** | 0.355 |
| SSD + Inception v2 | 115686 | 0.532 | 0.820 | 0.141 | 0.343 |
| SSD + Inception v2 | 125791 | 0.532 | 0.815 | 0.189 | 0.362 |

### 6.3.4 Considerations on Turin results and Weighted mAP

The previous section, highlights very large gaps between the mAPs over the DETRAC data and the Turin Castelfidardo data. Since the second part of this testing chapter regards vehicles tracking and counting performance on the c.so Castelfidardo camera, a *new ordering* for the checkpoint steps was needed. In fact, the maximum mAP@0.50 IOU obtained on the Turin dataset were quite higher than the ones showed for the Top 5 on DETRAC (e.g. the top SSD + MobileNet v2 checkpoint on DETRAC, of step 95078, has an mAP@0.50 over TURIN of 0.380 while the highest mAP@0.50 on TURIN for the same model is 0.551, for checkpoint 27412)

Moreover, while in general the mAPs for bus, van and other were lower on the Turin dataset compared to the DETRAC dataset, the mAPs for car were higher. Following this consideration, the Weighted mAPs were calculated for the models and will be showed in the following tables.

#### TURIN mAPs ALL and BY CLASS vs WEIGHTED

The next tables will show the *Top 5* with respect to the *mAP@0.50 IOU* calculated on the *Turin Castelfidardo* dataset, it will highlight the *components* on the four vehicles classes, and will finally show the recalculated *Weighted mAP@0.50 IOU*.

SSD + Mobilenet v2 - TURIN C.so CASTELF.

| Step | mAP.5$_{all}$ | mAP.5$_{car}$ | mAP.5$_{bus}$ | mAP.5$_{van}$ | mAP.5$_{oth}$ | WmAP.5 |
|---|---|---|---|---|---|---|
| 27412 | **0.551** | 0.909 | 0.550 | 0.235 | **0.510** | 0.869 |
| 183045 | 0.532 | 0.901 | 0.464 | **0.298** | 0.467 | 0.863 |
| 169771 | 0.531 | 0.921 | 0.495 | 0.265 | 0.442 | 0.880 |
| 167118 | 0.527 | **0.927** | **0.663** | 0.290 | 0.229 | **0.884** |
| 72355 | 0.512 | 0.911 | 0.609 | 0.248 | 0.282 | 0.868 |

SSD + Inception v2 - TURIN C.so CASTELF.

| Step | mAP.5$_{all}$ | mAP.5$_{car}$ | mAP.5$_{bus}$ | mAP.5$_{van}$ | mAP.5$_{oth}$ | WmAP.5 |
|---|---|---|---|---|---|---|
| 112344 | **0.444** | 0.926 | 0.209 | **0.332** | 0.307 | 0.885 |
| 131145 | 0.432 | 0.926 | 0.209 | 0.247 | **0.345** | 0.882 |
| 26467 | 0.422 | **0.945** | 0.336 | 0.316 | 0.093 | **0.898** |
| 18335 | 0.418 | 0.892 | 0.225 | 0.281 | 0.276 | 0.850 |
| 13317 | 0.408 | 0.892 | **0.379** | 0.226 | 0.134 | 0.846 |

**BEST WEIGHTED mAPs on TURIN (CVAT) FOR THE 4 MODELS**

The Weighted mAP@0.50 IOU were calculated for all of the checkpoints evaluated on the Turin Castelfidardo dataset, and those checkpoints were finally ordered by the WmAP@0.50 in order to give the best candidates for the tracking purpose (remembering here that 93% of the traffic are cars, it seemed to be the best ordering).

The following tables presents the Top 5 checkpoints based on the WmAP@0.50 on the Turin sequence.

SSD + Mobilenet v2 - TURIN C.so CASTELF.

| Step | WmAP.5 | mAP.5$_{car}$ | mAP.5$_{bus}$ | mAP.5$_{van}$ | mAP.5$_{oth}$ | mAP.5$_{all}$ |
|---|---|---|---|---|---|---|
| 27412 | **0.551** | 0.909 | 0.550 | 0.235 | **0.510** | 0.869 |
| 183045 | 0.532 | 0.901 | 0.464 | **0.298** | 0.467 | 0.863 |
| 169771 | 0.531 | 0.921 | 0.495 | 0.265 | 0.442 | 0.880 |
| 167118 | 0.527 | **0.927** | **0.663** | 0.290 | 0.229 | **0.884** |
| 72355 | 0.512 | 0.911 | 0.609 | 0.248 | 0.282 | 0.868 |

SSD + Inception v2 - TURIN C.so CASTELF.

| Step | mAP.5$_{all}$ | mAP.5$_{car}$ | mAP.5$_{bus}$ | mAP.5$_{van}$ | mAP.5$_{oth}$ | WmAP.5 |
|---|---|---|---|---|---|---|
| 112344 | **0.444** | 0.926 | 0.209 | **0.332** | 0.307 | 0.885 |
| 131145 | 0.432 | 0.926 | 0.209 | 0.247 | **0.345** | 0.882 |
| 26467 | 0.422 | **0.945** | 0.336 | 0.316 | 0.093 | **0.898** |
| 18335 | 0.418 | 0.892 | 0.225 | 0.281 | 0.276 | 0.850 |
| 13317 | 0.408 | 0.892 | **0.379** | 0.226 | 0.134 | 0.846 |

## 6.4 Tracking analysis

The preceding detection analysis was conducted on the Top 50 available checkpoints, based on the training evaluation mAPs, for each model, even if only the Top 5 for each of the analysis characteristics were actually showed. Carrying on such an exhaustive testing strategy for the tracking evaluation would be prohibitive in terms of resources and time. Indeed, the tests that will be presented in the next sections will focus only on the Top and Worst 5 found in the previous detection performance analysis.

### 6.4.1 Testing strategy

The tracking tests evaluate the framework's ability to count the vehicles on the C.so Castelfidardo sequence, based on the selected detection model checkpoint and on the detector speed performance.

Each individual test loads the checkpoint in a TensorFlow detector, produces detections, counts them on both directions based on the IOU tracker and a middle counter line, then compares the produced vehicles counts, both total and by class, with the ground truth value for the 15 minutes Turin annotated sequence.

The *ground truth countings* for the vehicles on both directions of the Turin sequence are:

- Vehicles count north direction: $191_{car} + 1_{bus} + 15_{van} + 5_{oth} = 212_{all}$
- Vechiles count south direction: $243_{car} + 2_{bus} + 31_{van} + 3_{oth} = 279_{all}$

The $\delta_i$s in the next tables are the average counting errors, defined as the average of the two error components of the two directions. Each error component, one for north direction and one for south direction, are the difference between the ground truth counting and the checkpoint's counting. The definitions are the following:

$$\delta_{i,north} = C_{groundtruth,i,north} - C_{ckpt,i,north}$$

$$\delta_{i,south} = C_{groundtruth,i,south} - C_{ckpt,i,south}$$

$$\delta_i = Average(\delta_{i,north}, \delta_{i,south})$$

$$\delta_{all} = \sum_i \delta_i, i \in \{car, van, bus, other\}$$

### 6.4.2 Detection performance impact on counting

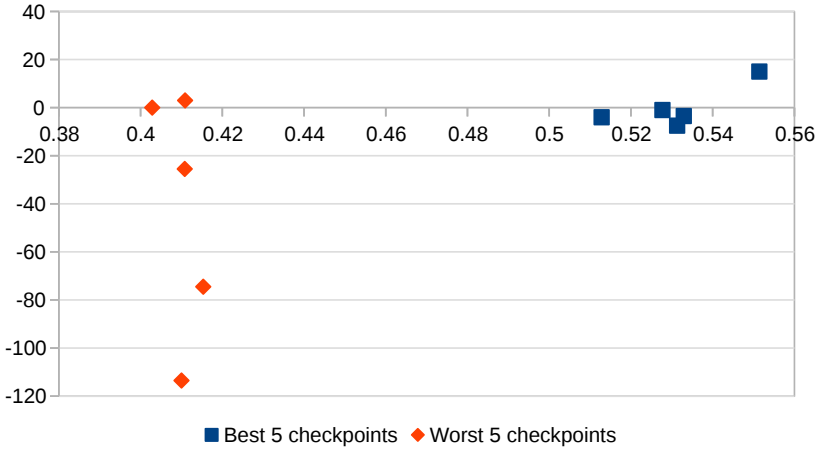
The previous section presented analysis on the detection performances of the various considered checkpoints, but the correlation between those precision metrics and the counting ability of the tracker are yet to be assessed, Thus, the first set of tests were executed on the best and worst 5 checkpoints of the SSD + MobileNet v2 architectures, ranked over the mAP@0.50 IOU metric.

SSD + MobileNet v2 - Best 5 on mAP 0.50

| Step | $mAP@.5$ | $\delta_{all}$ | $\delta_{car}$ | $\delta_{bus}$ | $\delta_{van}$ | $\delta_{oth}$ |
|---|---|---|---|---|---|---|
| 27412 | 0.551 | 15 | 7 | 0.5 | 9 | -1.5 |
| 183045 | 0.532 | -3.5 | -25.5 | -0.5 | 24 | -1.5 |
| 169771 | 0.531 | -7.5 | 2 | -1 | -7 | -1.5 |
| 167118 | 0.527 | -1 | 5 | -0.5 | -3.5 | -2 |
| 72355 | 0.512 | -4 | -21.5 | 0.5 | 17 | 0 |
| Avg Best 5 | 0.531 | -0.2 | -6.6 | -0.2 | 7.9 | -1.3 |

SSD + MobileNet v2 - Worst 5 on mAP 0.50

| Step | $mAP@.5$ | $\delta_{all}$ | $\delta_{car}$ | $\delta_{bus}$ | $\delta_{van}$ | $\delta_{oth}$ |
|---|---|---|---|---|---|---|
| 83719 | 0.402 | -3.5 | 0 | -1 | 0.5 | -3 |
| 164074 | 0.410 | -36.5 | -113.5 | 0 | 79.5 | -2.5 |
| 58021 | 0.410 | -16.5 | -25.5 | 0 | 11 | -2 |
| 73114 | 0.410 | -3 | 3 | -0.5 | -4 | -1.5 |
| 187606 | 0.415 | -44.5 | -74.5 | 0 | 34 | -4 |
| Avg Worst 5 | 0.409 | -20.8 | -42.1 | -0.3 | 24.2 | -2.6 |




As it can be seen from the tables and from the scatter plot, even though two of the worst checkpoint obtain a very low error score, it's easy to notice that the lower is the mAP value, the more is the tracking of the vehicles detection likely to fail. Indeed, the average counting error of the worst 5 checkpoints, that is -20.8 vehicles, is far from the clearly lower value of the best 5 ones (-0.2 vehicles).

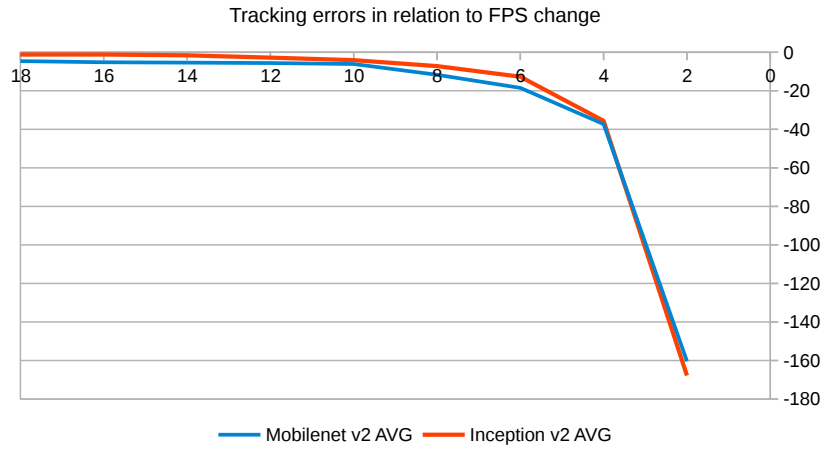
### 6.4.3 FPS resistance

The aim of the second set of tests, with a running time of more than 10 hours, is to asses the resistance of the tracking capabilities at lower detection framerates. In fact, slower models like Inception v2, which runs on the Jetson board at a limited framerate, miss detected frames when run on a live stream, and produce distant detections, increasing the car displacement between two consecutive frames. The tracker, as explained in the relevant section, uses a metric that suffers from this situation.

These tests are run by simulating the missed frames due to lower fps detections, ranging from 18 fps (same as the annotated source) down to 2

fps, with a step of 2fps, on both SSD + MobileNet v2 detector and Inception v2 detector. At each fps step and for each model, the Top 5 detectors, ranked this time by the Weighted mAP@0.50, were tested and the average error was used for each fps step.

| FPS | Mobilenet v2 $\delta_{all,top5}$ | Inception v2 $\delta_{all,top5}$ |
|---|---|---|
| 18 | -4.6 | -1.2 |
| 16 | -5.2 | -1.2 |
| 14 | -5.4 | -1.6 |
| 12 | -5.6 | -2.8 |
| 10 | -6.1 | -4.1 |
| 8 | -11.7 | -7.2 |
| 6 | -18.5 | -12.7 |
| 4 | -37.3 | -35.6 |
| 2 | -160.4 | -167.8 |




Both the SSD + MobileNet v2 and the SSD + Inception v2 Top 5 checkpoints appear to resist well all the way down to 10 FPS, with an error of missed vehicles that rises exponentially passed that point.

It is worth noting that having used the WmAP@0.50 for the ranking of the Top 5 checkpoints led to an average error for the Top 5 SSD + MobileNet v2 checkpoints at 18 FPS (same as source so no frames are dropped) which is higher than the one obtained by the Top 5 according to the standard mAP@0.50 (-4.6 vs. 0.2), suggesting that the latter may finally be a better metric for assessing the performance.

### 6.4.4 Inception v2 tests

To complete the tracking testing section, a table with the results of the best 5 SSD + Inception v2 checkpoints at 18 and 5 FPS, ranked on the WmAP@0.50 IOU will be presented here. The reason why, among all the available steps, the 5 FPS is chosen is that it mimics the running speed of the Inception v2 model on the Nvidia Jetson board, according to the Benchmark section, and giving results that are pretty close to the ones that would be obtained by using it instead of the MobileNet v2.

SSD + Inception v2 - Best 5 on WmAP@0.50 at 18 fps

| Step | $WmAP@.5$ | $\delta_{all}$ | $\delta_{car}$ | $\delta_{bus}$ | $\delta_{van}$ | $\delta_{oth}$ |
|---|---|---|---|---|---|---|
| 26467 | 0.422 | -9.5 | -5 | 4 | -10.5 | 2 |
| 112344 | 0.444 | -0.5 | 7.5 | 3 | -9.5 | -1.5 |
| 131145 | 0.432 | 1 | 3.5 | 4.5 | -8.5 | 1.5 |
| 137177 | 0.361 | -3 | 4.5 | 1.5 | -8 | -2.5 |
| 153566 | 0.407 | 6 | 17 | 2.5 | -11 | -2.5 |
| Avg Best 5 | 0.413 | -1.2 | 5.5 | 3.1 | -9.5 | -0.3 |

SSD + Inception v2 - Best 5 on WmAP@0.50 at 5 fps

| Step | $WmAP@.5$ | $\delta_{all}$ | $\delta_{car}$ | $\delta_{bus}$ | $\delta_{van}$ | $\delta_{oth}$ |
|---|---|---|---|---|---|---|
| 26467 | 0.422 | -33.5 | -25 | 3.5 | -11 | -1 |
| 112344 | 0.444 | -9.5 | -1 | 2 | -8 | -2.5 |
| 131145 | 0.432 | -22 | -15 | 2.5 | -8.5 | -1 |
| 137177 | 0.361 | -25.5 | -16.5 | 2 | -8.5 | -2.5 |
| 153566 | 0.407 | -11 | -1.5 | 3 | -11 | -1.5 |
| Avg Best 5 | 0.413 | -20.3 | -11.8 | 2.6 | -9.4 | -1.7 |

It is worth noting that, since Inception v2 at

## 6.5 Final considerations

The results highlighted in this final chapter on the two apparently most promising models combinations trained on the DETRAC dataset, the SSD + MobileNet v2 and SSD + Inception v2, showed that they share the same detection performance on both the DETRAC-Eval and the Turin datasets. It's been also confirmed, as expected, that the mAP metrics are linked to the counting performance of the tracker used in this work.

Both the models have a very low counting error at full speed but suffer at decreased speeds and since Inception v2 runs at only 5 fps on the Nvidia Jetson Board, it's definitely discarded.

The best choice is in fact the SSD + MobileNet v2, which runs at 19 FPS on the board and provides a very low average counting error on the test data. The checkpoint currently loaded on the board at the company's branch is 167118.

# Bibliography


[1] E. Bochinski, V. Eiselein, and T. Sikora, *High-Speed tracking-by-detection without using image information*, in 2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS), IEEE, 8 2017, pp. 1–6.

[2] K. Fukushima and S. Miyake, *Neocognitron: A Self-Organizing Neural Network Model for a Mechanism of Visual Pattern Recognition*, Springer, Berlin, Heidelberg, 1982, pp. 267–285.

[3] R. Girshick, *Fast R-CNN*, in 2015 IEEE International Conference on Computer Vision (ICCV), IEEE, 12 2015, pp. 1440–1448.

[4] ——, *Python implementation of Faster-RCNN on Caffe*. https://github.com/rbgirshick/py-faster-rcnn, 2015.

[5] R. Girshick, J. Donahue, T. Darrell, and J. Malik, *Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation*, in 2014 IEEE Conference on Computer Vision and Pattern Recognition, IEEE, 6 2014, pp. 580–587.

[6] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, and K. Murphy, *Speed/accuracy trade-offs for modern convolutional object detectors*, (2016).

[7] D. H. Hubel and T. N. Wiesel, *Receptive fields, binocular interaction and functional architecture in the cat's visual cortex*, The Journal of Physiology, 160 (1962), pp. 106–154.

[8] Joseph Redmon, *Darknet: Open Source Neural Networks in C*, 2013.

[9] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, in Advances in Neural Information Processing Systems 25, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, eds., Curran Associates, Inc., 2012, pp. 1097–1105.

[10] Y. LeCun, Y. Bengio, and G. Hinton, *Deep learning*, Nature, 521

(2015), pp. 436–444.

[11] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, *SSD: Single Shot MultiBox Detector*, Springer, Cham, 2016, pp. 21–37.

[12] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, *You Only Look Once: Unified, Real-Time Object Detection*, in 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), IEEE, 6 2016, pp. 779–788.

[13] S. Ren, K. He, R. Girshick, and J. Sun, *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, IEEE Transactions on Pattern Analysis and Machine Intelligence, 39 (2017), pp. 1137–1149.

[14] V. Sze, Y.-H. Chen, T.-J. Yang, and J. S. Emer, *Efficient Processing of Deep Neural Networks: A Tutorial and Survey*, Proceedings of the IEEE, 105 (2017), pp. 2295–2329.

[15] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, *Going Deeper with Convolutions*, (2014).

[16] J. R. R. Uijlings, K. E. A. van de Sande, T. Gevers, and A. W. M. Smeulders, *Selective Search for Object Recognition*, International Journal of Computer Vision, 104 (2013), pp. 154–171.

[17] P. Viola and M. J. Jones, *Robust Real-Time Face Detection*, International Journal of Computer Vision, 57 (2004), pp. 137–154.

[18] L. Wen, D. Du, Z. Cai, Z. Lei, M.-C. Chang, H. Qi, J. Lim, M.-H. Yang, and S. Lyu, *UA-DETRAC: A New Benchmark and Protocol for Multi-Object Detection and Tracking*, (2015).

[19] B. Widrow, D. E. Rumelhart, and M. A. Lehr, *Neural networks: applications in industry, business and science*, Communications of the ACM, 37 (1994), pp. 93–106.

[20] 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, 9 2017, pp. 3645–3649.