Motorvehicle University of Emilia-Romagna

Advanced Automotive Electronic Engineering

Neural Network Computing and AI for automotive

# Vehicle detection and tracking

**Author:**
889711 Ravagli Giacomo

**Professors:**
Cucchiara Rita
Baraldi Lorenzo

## Abstract

Vehicle detection and tracking is an interesting theme in computer vision, most common applications being surveillance and autonomous driving. This paper explores two of the most used network architectures for object detection, namely Faster-RCNN and Yolo, and explains how vehicle tracking has been performed on the output of the two networks using SORT.

*Keywords: detection, tracking, Python, Pytorch, FasterRCNN, Yolo, SORT*

## 1 Introduction

Vehicle detection and tracking is an interesting theme in computer vision, most common applications being surveillance and autonomous driving. This paper explores two of the most used network architectures for object detection, namely FasterRCNN and Yolo, and explains how vehicle tracking has been performed on the output of the two networks using SORT. SORT stands for "Simple Online and Realtime Tracking" and is an algorithm that is used to solve the many-to-many association problem over multiple frames, exploiting, under the hood, the Hungarian (Munkres) algorithm. This solution is more robust than using a nearest-neighbor approach and it has a negligible overhead in terms of computation time.

## 2 Handling a video stream input

Loading a video and working on it is not much different than with images. A video is, in fact, a series of images (frames). OpenCV [1] is exploited to capture the video, split it in an array of images and to modify them in order to suit our needs. The most common operation performed on images is resizing: for example, the release of Yolo chosen needs images to be resized to 416x416 before feeding the network. Also, OpenCV can be used to draw bounding boxes around detected objects. The use of this library is recommended since it is user-friendly and those operations have negligible impact on the overall execution time.

## 3 Faster RCNN

It has been decided to start from something already known to some extent by the author i.e., as the title suggest, Faster RCNN [2]. The Torchvision package offer a list of convolutional neural network ready to be used; some of them have also pre-trained models. It has been decided to use a Faster RCNN with ResNet 50 backbone, pre-trained over COCO dataset. This is quasi state-of-the-art, so it is not surprising that, using it, we obtained very good performances.

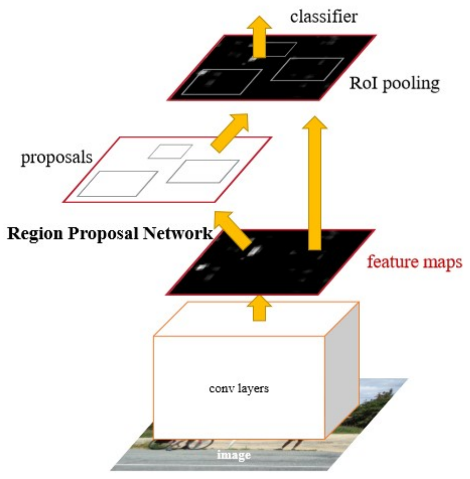

Figure 1: Conceptual scheme for Faster RCNN

### Tracking

As introduced in the previous section, tracking was performed using SORT [7]. SORT is an algorithm for multiple object tracking that can be applied even to real-time applications. It is efficient and quite precise. It is worth to underline that SORT works "on top" of a given detection algorithm. That said, we can point out that object tracking with SORT is also robust

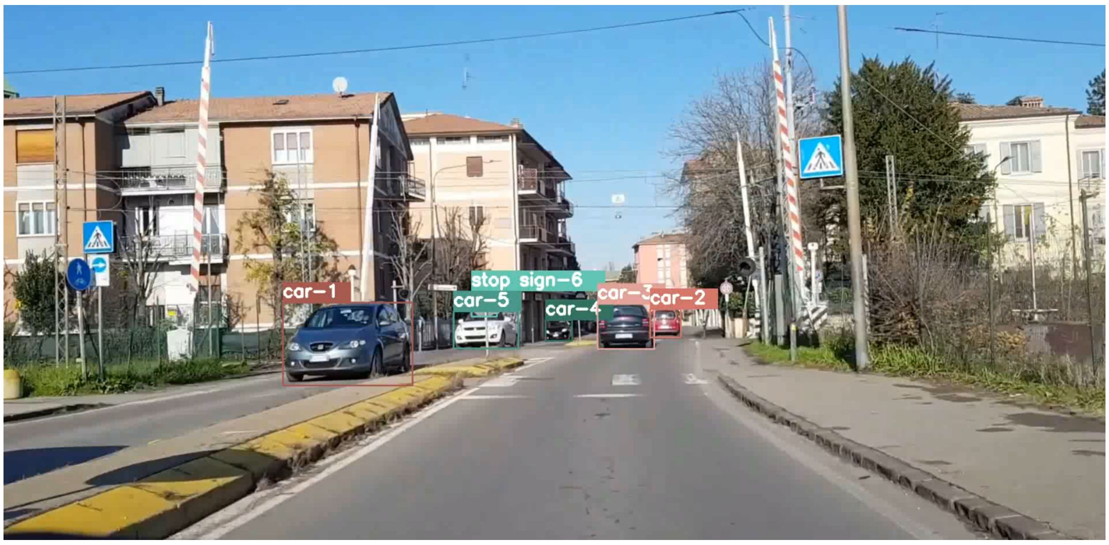

Figure 2: Results with Faster RCNN, pre-trained on COCO

with respect to missed detection: it may happen that object A is detected in frame $t$, is not sensed at all in frame $t+1$ and it is detected again in frame $t+2$; using SORT we are able to associate the two detection at frame $t$ and $t+2$ to the same object A. In this situation other algorithms would tend to indicate that in the two frames there are different objects.

Of course, in order to use SORT with Faster RCNN, some manipulation was needed. In fact, SORT expect a tensor input, and the version of Faster RCNN used returns a list of dictionaries.

### Weaknesses

The main weakness in this application is that despite what the name suggest, Faster RCNN is no fast at all. Running the Python script on a GTX950M GPU we were able to achieve about 1 fps video output. It is suggested that changing the Faster RCNN backbone the speed could be improved [3], however we are still far from real time performances. This is inherently related to the working scheme of Faster RCNN (see Figure 1).

## 4 Yolo v3

Yolo is another algorithm for object detection [4]. The architecture is fairly simple (Figure 3) and the working principle is based on the fact that the image is only looked once, i.e. it pass only one time through the convolutional neural network: during this passage bounding boxes are generated and object classification is performed. Because of this fact, Yolo is much faster than other detection strategies: in our case it was possible to reach an output video stream of 15 fps, which is quasi-real-time. On the other hand, Yolo is less precise than e.g. Faster RCNN, in particular it struggles when two object overlaps of a significant degree. In the first versions of Yolo also detecting small object was an issue, but the author claim it has been solved with Yolov3 [5].

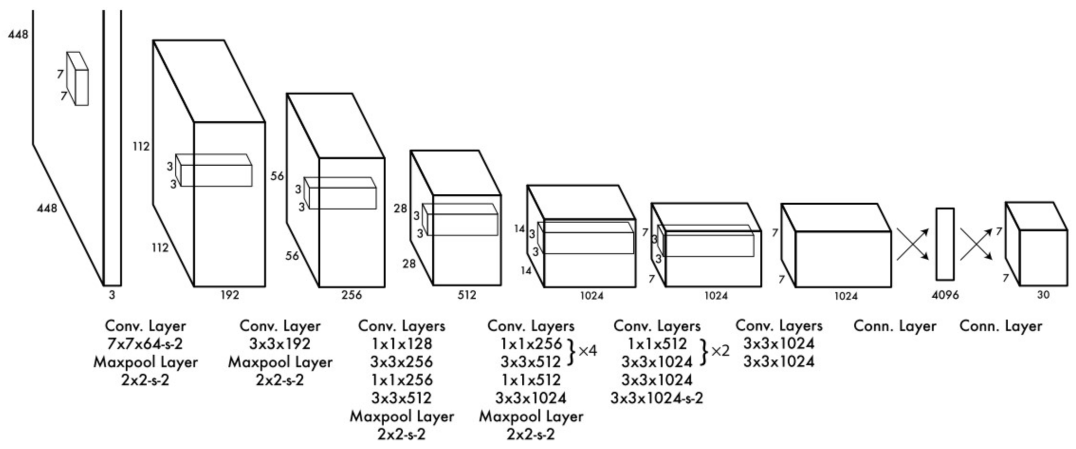

Figure 3: Yolo v1 architecture

## Detection

Conceptually, object detection with Yolo works in the following way: input image is split in an $S$x$S$ grid. Each grid cell predicts $B$ bounding boxes and confidence scores for them. Then, at the end, non-maximum suppression is employed in order to avoid that the same object is detected multiple time: in fact often happens that many of the predicted bounding boxes recognize that there is an object (or a part of it), and so we have to extract the one having higher confidence, hopefully being the one that best "wraps around" the image.

## Tracking

Again, SORT was employed for vehicle tracking. In this case the adaptation work was very little since the data type output from Yolo detection is ready to feed SORT.

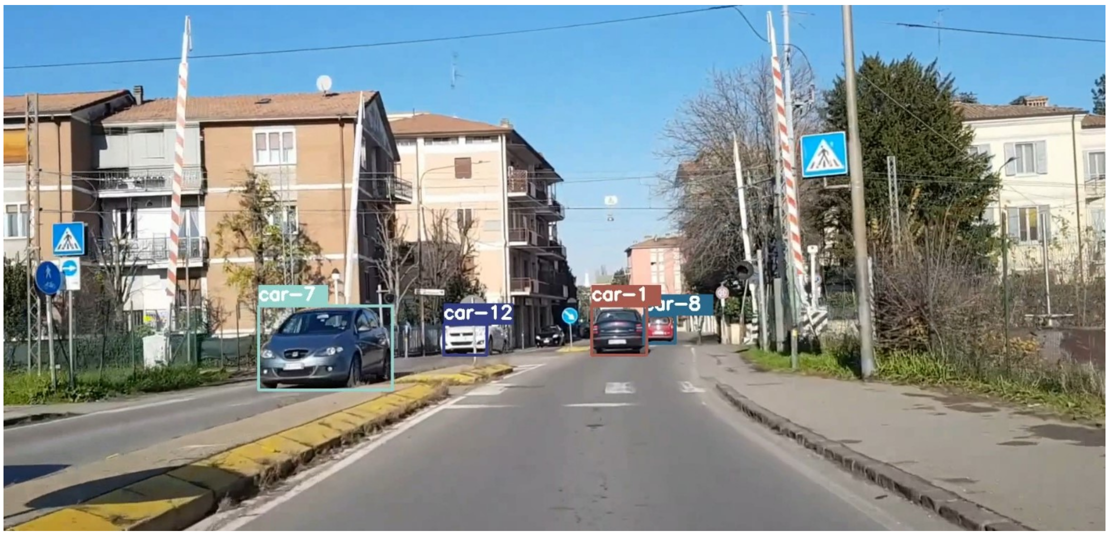

Figure 4: Results with Yolov3, pre-trained on COCO

It is worth to note the following: when we display the bounding boxes, we are displaying the result after tracking with SORT. Debugging the code we noticed that sometimes the objects (vehicles, truck, pedestrians...) highlighted by SORT are less than those indicated by Yolo. Thus, if the code is to be used for autonomous driving applications such as forward collision avoidance it would be more profitable to consider the detections Yolo gives out.

## 5 Train Yolov3 over Kitti dataset

Since the target application is vehicle tracking, it was decided to try to see if retraining the Yolo network over a dataset containing a lot of images of car, trucks and pedestrian could lead to better results.

The dataset chosen was Kitti. It provides 7800 images taken from a stereocamera mounted on a car, and each image is labelled.

### Image resizing

First operation to be performed is to resize the training/validation images. In fact Yolo works on a square image, so if no "smart" resizing is performed we obtain wrong results. It is necessary, in fact, to maintain the aspect ratio. The issue with this dataset is that the images have an aspect ratio 3.3 : 1, thus large gray bands will appear in the output image. Moreover the image is resized to 416x416 before entering the network, so there is an huge reduction in dimension: original file are around 1225x370.

OpenCV and PIL packages have been used in this step.

### Adjustment to labels

Inspecting a label file, 9 classes are found: *car, van, truck, pedestrian, person sitting, tram, misc, don't care.*

Initially, it was thought only to remove *misc* and *don't care* entries; later on it was also decided to remove *person sitting* and *cyclist* since looking at the resulting images it was almost impossible, even for a human, to understand the presence of one of those objects due to the resizing; we believed that this could lead to a bad training. A dedicated Python script was realized to iterate over labels file and remove unwanted entries.

After that, the last step consisted in converting the labels file from Kitti format to the format expected by Yolo. Another Python script was used to perform this task.

### Training strategy

The training was performed on a GTX950M. Due to limitations in GPU memory, maximum batch size during training was limited to 3 images; thus, an epoch took almost 1 hour to complete. After each epoch, the updated weight file is saved.

Training strategy can be seen in Figure 5. We started from pre-trained weight over the Kitti dataset, that were not satisfying in term of detections and since often "misc" label was present in place of the correct one, see Figure 6. So we questioned if it were possible to refine the result by continuing the training with label adjustment we presented before. First choice was the choice of the optimizer to be used; we started from Adam, which should be the most promising one. We let the

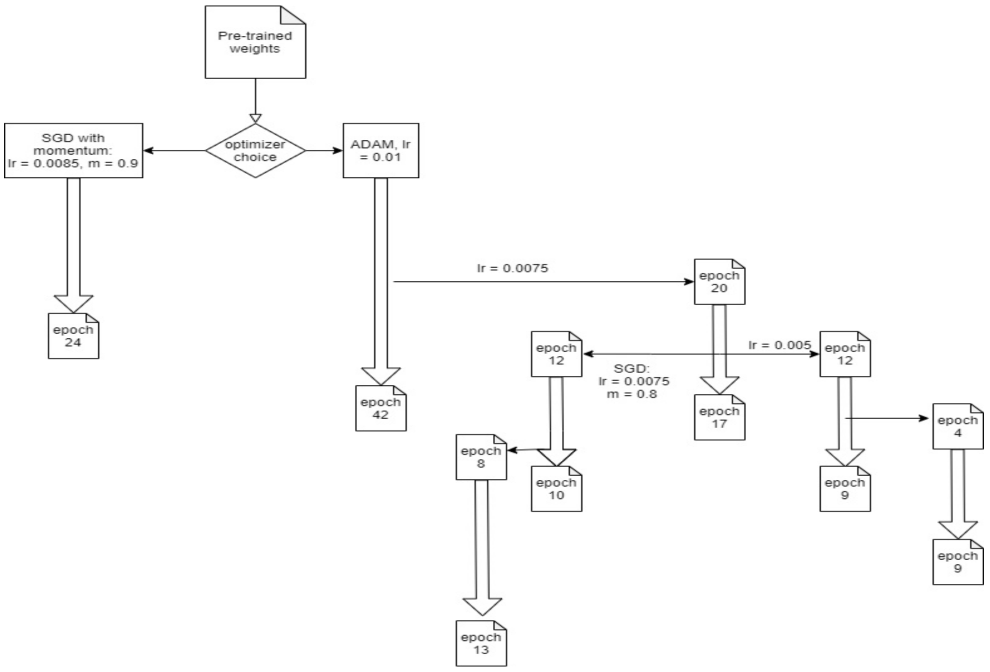

Figure 5: Training strategy

training run for 42 epochs then, analyzing the result, it was noticed a deterioration of detection performances after epoch 20. So we restarted from that point reducing the learning rate. The same problem was found out again, so at first we tried to reduce the learning rate furthermore, then we decided to continue the training with SGD. The last tentative done was to restart from the pre-trained weight file and

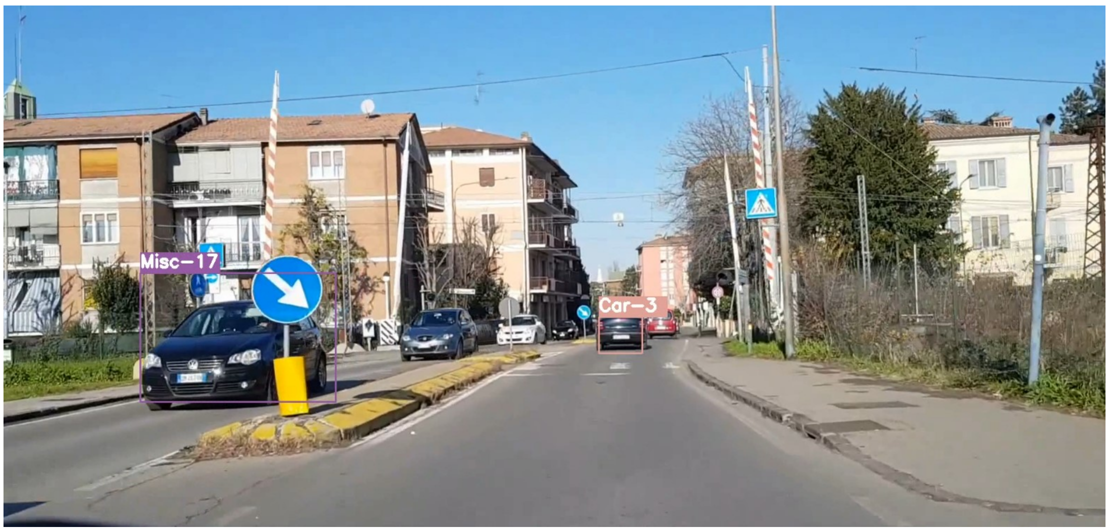

Figure 6: Results with Yolov3, pre-trained on Kitti

this time use SGD with momentum as optimizer instead of Adam.

## Results

To be honest, before starting the training we expected much better results than those we obtained. Actually, sometimes the final result after several epochs are even worse than the point we started from.

After all, we achieve some refinements, but we are still far from the expected result, namely reaching comparable performances to the network pre-trained over COCO dataset. A possible explanation is that the images used for training are very small due to the 3.3 : 1 aspect ratio.

Results, as well as source code for this project, can be found at [9].

## References

[1] OpenCV; https://docs.opencv.org/4.2.0/d6/d00/tutorial_py_root.html

[2] *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*, by Ren, He, Girshick and Sun

[3] Making Faster RCNN faster; https://jkjung-avt.github.io/making-frcn-faster/

[4] *You Only Look Once: Unified, Real-Time Object Detection*, by Redmon, Divvala, Girshick and Farhadi

[5] *YOLOv3: An Incremental Improvement*, by Redmon and Farhadi; https://pjreddie.com/darknet/yolo/

[6] Object detection and tracking in Pytorch; https://towardsdatascience.com/object-detection-and-tracking-in-pytorch-b3cf1a696a98

[7] *Simple Online and Real-time Tracking*, by Bewley, Ge, Ott, Ramos and Upcroft

[8] The Kitti dataset; http://www.cvlibs.net/datasets/kitti/eval_object.php?obj_benchmark=2d

[9] GitHub repository for this project; https://github.com/giacomoravagli/Car-detection-and-tracking