BABEŞ BOLYAI UNIVERSITY, CLUJ NAPOCA, ROMÂNIA
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE

PEDESTRIANS DETECTOR

– MIRPR report –

Team members
Cazacu Arina Ioana, Software Engineering, 258/1, arina.cazacu@gmail.com
Cilean Liliana, Software Engineering, 258/1, lilianacilean@yahoo.com

2021-2022

Abstract

Self-driving vehicles are one of the most exciting and impactful applications of AI, and providing a car with the ability to see stands at the very beginning of it all. This paper illustrates three experiments implementing pedestrian detection on a React Native mobile app using different models, respectively: Tiny YOLOv3 with Flask REST server, COCO SSD and BodyPix, both provided by TensorFlowJS, hence the final app being client only. The models were pretrained on COCO dataset. Our intention was to get closer to real time prediction, so COCO SSD with a Lite MobileNetv2 backbone turned out to be the best in terms of accuracy and also, by far, the fastest possible option.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 What? Why? How?

There are a lot of people out there dreaming about how the future will look like, and there is a very common answer to this question: self-driving cars. Currently, the automotive industry is trying to step out of the ordinary and offer an autonomous experience to the driver. A car is equipped with lots of sensors, just like humans - it can see things, or react to them. The idea is to teach the car to make decisions based on what these sensors intercept. This paper is trying to focus on one of the many branches of autonomous driving: endowing the car with the ability of seeing.

- What is the (scientific) problem?

  An autonomous car should be able to "see" and make its own decisions based on the input. This paper aims to provide a fast and reliable computer vision solution for pedestrians detection, which is one of the most crucial aspects when it comes to self-driving cars. Using the input from the camera, any pedestrian should be detected in less than the blink of an eye, and from a considerable distance, so that the moving car gets the possibility to react smoothly.

- Why is it important?

  The goal is to get to a higher level of autonomy, but the hardest thing to do is keeping humans as safe as possible. The number of car crashes is huge every year, mostly because of lack of attention or driver's drowsiness. A machine will never get tired and this is why the automotive industry is trying to design the car to take over most of the driver's responsibilities. The thing here is that when it comes to human safety, the machine is not allowed to make mistakes, so the purpose is first of all to make these detection algorithms reach perfection.

- What is our basic approach?

  The idea is to create an intelligent algorithm that gets an image as input and outputs it with an emphasis on where the pedestrian has been detected. The algorithm should be able to provide really fast and accurate responses, therefore transfer learning techniques will also be used.

1.2 Paper structure and original contribution(s)

The research presented in this paper is focused on outlining the theory behind the TinyYoloV3 model and employing it for the particular problem of pedestrian detection in different contexts.

The main contribution of this report is to present a solution based on an intelligent classifier consisting of a pre-trained model which is run against multiple sets of data in the aim of solving the problem of pedestrian, vehicle and road sign detection.

The second contribution of this report is the development of a simple and intuitive mobile application that will present a practical user interface through which the user can easily test the algorithm results on input of their own.

The third contribution of this thesis consists of the employment of a number of optimizations with a view to increasing the overall accuracy of the algorithm and testing its performance in different scenarios.

The work is structured in seven chapters as follows:

The first chapter is a short introduction in the subject of object detection in the driving assistance field, what it is about and why it is important and our reasons that were behind choosing this topic.

The second chapter describes the scientific problem in more detail and considers the advantages and disadvantages of our approach.

The third chapter treats some other related work in the field and gives a brief description of their results.

In chapter four we provide the investigate approach, together with the tools and technologies that were used in order to implement it. We describe the underlying architecture of the TinyYoloV3 model and the algorithm employed by it, stating how we will use this for our problem and how it is suited for the driving assistance object of study. We show how the algorithm works in practice and provide a short list of the tools we will be using for our study.

Chapter five comprises the main part of this report and consists of the description of our application requirements, the methodology by which we plan to solve the problem, the datasets we will be conducting our experiments on and the results obtained in the end. At the end of the chapter, we also

provide some discussion around the results and potential optimizations to the algorithm, comparing the results obtained with the initial ones. The chapter ends with a small presentation of the user interface.

Chapter 6 explains the experimental methodology and the numerical results obtained with our approach and the state of the art approaches. Our focus in this chapter is on the interpretation and the statistical validation of the results. Also, this chapter is a dive into the philosophical aspects of autonomous driving and how this is likely to affect the way in which we report ourselves to the task of driving in general. We analyze the objectivity of the solution proposed and raise some interesting questions relating to the ethics of the smart driving assistants in general. We also provide some interesting data about the way our algorithm performs on individuals of different races and ethnicities, by this trying to advance the idea of diversity and inclusion in the way we use such technology.

The last chapter offers a summarization of our conclusions and future work and also try to analyze the strenghts and weaknesses of our application with the focus on what we can improve both in the algorithm and the application.

Chapter 2

Scientific Problem

2.1 Problem definition

Advanced driver-assistance systems (ADAS) are groups of electronic technologies that assist drivers in driving and parking functions. Through a safe human-machine interface, ADAS increase car and road safety. ADAS use automated technology, such as sensors and cameras, to detect nearby obstacles or driver errors, and respond accordingly.

As most road accidents occur due to human error, ADAS are developed to automate, adapt, and enhance vehicle technology for safety and better driving. ADAS are proven to reduce road fatalities by minimizing human error. Safety features are designed to avoid accidents and collisions by offering technologies that alert the driver to problems, implementing safeguards, and taking control of the vehicle if necessary. Adaptive features may automate lighting, provide adaptive cruise control, assist in avoiding collisions, incorporate satellite navigation and traffic warnings, alert drivers to possible obstacles, assist in lane departure and lane centering, provide navigational assistance through smartphones, and provide other features.

The point of this paper is to show a implies of identifying people on foot in any kind of conditions and tie this to the current climate state to propose to the client the foremost suitable activities to be taken in certain activity circumstances or to respond naturally to them.

The main advantage of an artificial intelligent algorithm is its ability to analyse and employ an enormous quantity of data, much more efficiently than possible for humans through classical statistical analyses. Moreover, the more data received, the more accurate the result will be.

The personal driving assistant would be built off an intelligent classification algorithm based on neural networks. Other methods used for obstacle detection include:

- template matching approach - the real image is compared against a sufficient number of templates

of the object-of-interest to identify the presence of the object in the sample image.

We will be using the Deep Learning approach. Our classifier will receive as input the image or video sequence and output the same image, with the obstacles marked and delimited accordingly. This should provide a good start for further improvements and additional features that would contribute to a better navigation experience in autonomous driving.

On the other hand, processing image with people is a challenge even for an intelligent algorithm. Most of the images may be difficult to read an interpret, which means that large sets of data with labeled images are required for the algorithm to work properly. Machines are still having a hard time understanding images with people and using certain unclear images can be misleading for the AI, that can provide erroneous or implausible detections.

2.2 Challenges

Much of effort was concentrated on trying to make the algorithm run faster and get a better accuracy with the amount of resources that we have had at our disposal. In trying to work with a model that is pre-trained, we have had little control over the time it takes the algorithm to do a detection. The models themselves are pretty large in size and take up a great deal of resources to run.

Lack of sufficient resources to run the algorithm was an imminent problem that we have come across.

Chapter 3

State of the art/Related work

In this chapter we present a few methods that were used for pedestrian detection in different contexts. In order to provide safety and to reduce traffic accidents driving assistance systems have been developed with the help of artificial intelligence and ones of the features are pedestrian detectors and driving assistants.

Firstly, we will take a look at "Traffic signs recognition for driving assistance" (add link to bibliography) by Yatham Sai Sangram Reddy, Devareddy Karthik, Nikunj Rana, M Jasmine Pemeena Priyadarsini, G K Rajini and Shaik Naseera.

- What is their problem and method?

  Problem: The paper proposes a method to detect the traffic sign board in a frame and to identify the sign on it.

  Used algorithm: HAAR Cascade Training - Viola Jones Algorithm for detecting differences in pixel intensities.

- How is our problem and method different?

  Our problem is a little bit different than the one from the mentioned paper, because we try to detect people that are present in certain picture. At the same time, they mention using real-time camera system. The implementation is also done differently, since we are using COCO-SSD for object detection, while their implementation was done by training a cascade classifier for head detection from the scene, by using HAAR features through OpenCV.

- DataSets: Each of the Traffic signs is recognised using a database of images of numbers and symbols used to train the KNN classifier using OpenCV libraries. The images are split into

positives (containing the target road sign) and negatives (containing background). For each positive image an annotation file is created, and then a vector (.vec) file for all of them.

- Results: For the traffic signs with single contours, the nearest neighbour is found and the output is the response returned by the classifier. For the traffic signs with multiple contours, the responses returned by the classifier are sorted based on their x-positions after the nearest neighbours are found. The authors do not offer an aggregated result in terms of accuracy or detection rate of their algorithm. Some sample outputs are presented, though, and they look like: "Sign Board: School Zone", "Speed limit: 50 km/h" etc. The algorithm works directly on frames captured from the camera and presents satisfactory results.

Next, we will take a look at "Driving situation-based real-time interaction with intelligent driving assistance agent" (add link to bibliography) by Young-Hoon Nho.

- What is their problem and method?

  Problem: The article presents an intelligent driving assistance agent that interacts with the driver in real time. Different driving situations are recognized by the algorithm, such as speed bump, corner, crowded area, uphill, downhill, straight, parking space. The algorithm combines driving intention recognition with driving situation information, that is automatically tagged as the vehicle is driven, to build a long-term interaction model.

  Used algorithm: They use an algorithm based on hidden Markov models (HMMs).

- How is our problem and method different?

  Our problem is a little bit different than the one from the mentioned paper, because we try to detect people that are present in certain picture. At the same time, they mention using real-time camera system. The implementation is also done differently, since we are using COCO-SSD for object detection, while their implementation was done by using an algorithm based on hidden Markov models.

- DataSets: The acquisition of data was done in intervals, while driving a Kia Morning vehicle, using on-board diagnostics 2 (OBD2). The data collected included: steering wheel data, velocity and accelerator pedal data, latitude and longitude as serial data. 430 data inputs were obtained, each of them being labeled by one of the 7 driving situations considered.

- Results: The results they obtained with this approach is 94.9 percent accuracy, more specifically - 408 our of 430 correctly recognized inputs.

Chapter 4

Investigated approach

There are a lot of documented ways of implementing object detection nowadays, and most of them are written in Python. Why is this the most popular language for AI and ML based projects? Because it offers a great library ecosystem. Libraries provide base level items so developers don't have to code them from the beginning every time.

TensorFlow is a free and open-source library for machine learning and artificial intelligence. It can be used across a range of tasks but has a particular focus on training and inference of deep neural networks. It uses Python to provide a convenient front-end API for building applications, while executing those applications in high-performance C++. This is the core library that is used in this project, with a twist that will be revealed in the second experiment.

4.1 First Experiment

We first started by implementing a mobile application with a regular client-server model. The server was written in Python with a Flask RESTful API. The client side is implemented in React Native and has to send the photo to the server and then wait for the detection.

The YOLOv3 pre-trained model we have decided to use comes with vehicle, persons and road sign detection out of the box, so we only need to fine-tune it to fit the specific needs of our application and optimize its performance for our problem.

The way we will measure the quality of the detection is by tracking the overall accuracy and trying to maximize the number of successful detections, at the same time reducing the number of false positives. Another aspect that we considered of tremendous importance is the speed of the algorithm, since we are striving for a pleasant user experience and are driving towards a real-time experience, we believe an almost instant response would be desirable, so reducing the response time as much as

possible was another factor that we looked at in our work.

4.2 Second Experiment

As fast detection is a vital aspect in automotive, and since we are driving towards a real-time experience, an almost instant response would be desirable.

Reducing the response time as much as possible is the main reason for switching to another perspective: TensorFlow for JavaScript. It is designed as a breakthrough for running entirely client-sided machine learning models. And it is exactly what we needed to get rid of the client-server communication overhead, bringing us a few steps closer to real-time detection.

TensorFlow.js offers a set of pre-trained models that have been ported from the "pythonic" version of Tensorflow Object Detection API - an open source framework built on top of TensorFlow that makes it easy to construct, train and deploy object detection models. The documentation from Tfjs' GitHub provides a very clear overview for the utility of each model, by splitting them up in four great categories:

- Images
- Audio
- Text
- General Utilities

Each model also has a short and concise description next to it, along with the installation command. The area of interest for this particular project is related to Images and object detection. After browsing through the models' descriptions we decided that we will be using the object detection model: COCO-SSD, which is a perfect fit for our problem's hypothesis.

4.2.1 The Model - overview

COCO-SSD is an object detection model that aims to localize and identify multiple objects in a single image. This model detects objects defined in the COCO dataset, which is a large-scale object detection, segmentation, and captioning dataset. More detailed information about the data can be found in Chapter 5, the Data section.

SSD stands for Single Shot MultiBox Detection, as the model is capable of detecting 80 classes of objects (from person, vehicles, traffic light, stop signs to elephants and bananas). It can take as input

any browser-based image elements (<img>, <video>, <canvas> elements, for example), returning an array of bounding boxes with class name and confidence level. SSD is a neural network architecture made of a single feed-forward convolutional neural network that predicts the image's objects labels and their position during the same action. The counterpart of this "single-shot" characteristic is an architecture that uses a "proposal generator," a component whose purpose is to search for regions of interest within an image.

Once the regions of interests have been identified, the second step is to extract the visual features of these regions and determine which objects are present in them, a process known as "feature extraction." COCO-SSD default's feature extractor is lite_mobilenet_v2, an extractor based on the MobileNet architecture. In general, MobileNet is designed for low resources devices, such as mobile, single-board computers, e.g., Raspberry Pi, and even drones.

4.2.2 Mobile Integration

The React Native mobile app has to be created with Expo because the React Native CLI method raises a lot of compatibility issues and version constraints between TensorFlow.js and other packages. Moreover, the linking has to be done manually without Expo, which can lead to more complicated environmental errors.

After setting up the Expo project so that it has TensorFlow.js properly installed, the model can be "installed" to the mobile app by running the following command:

```
$ npm install @tensorflow-models/coco-ssd
```

There are three main steps for making a prediction:

1. **import** model

```
import * as cocossd from "@tensorflow-models/coco-ssd";
```

2. **load** model

```
const model = await cocossd.load();
```

3. **detect** image

```
const predictions = await model.detect(img);
```

At **load**, the developer is able to specify certain configurations for the model. The documentation provides the following ModelConfig interface:

```
export interface ModelConfig {
  base?: ObjectDetectionBaseModel;
  modelUrl?: string;
}
```

The **base** attribute controls the base CNN model, which can be one of the following:

- 'mobilenet_v1'
- 'mobilenet_v2' - has the highest classification accuracy
- 'lite_mobilenet_v2' - *default*, is the smallest in size and the fastest in inference speed. Since we are keen on achieving the fastest possible results, we will be using this as the base model.

For the **detection** part, we are passing as a parameter the image converted to a Tensor. A Tensor is a container which can hold data in multiple dimensions. This method can also receive the image in different shapes, such as html elements: ImageData, HTMLImageElement, HTMLCanvasElement, HTMLVideoElement. It provides two more optional parameters: maxNumBoxes and minScore. Thus, the **detect** function can have the following arguments:

- **img**: either Tensor or html element to perform the detection on.
- **maxNumBoxes**: the maximum number of bounding boxes of detected objects. There can be multiple detections of the same class, but at different locations. Default: 20.
- **minScore**: The minimum level of confidence for returning detections. It can be a value between 0 and 1, default: 0.5.

The **output** is an array of objects. An object contains the bounding box parameters, the class and the score:

```
[{
  bbox: [x, y, width, height],
  class: "person",
  score: 0.8380282521247864
}, {
  bbox: [x, y, width, height],
  class: "kite",
  score: 0.74644153267145157
}]
```

4.3 Third Experiment

As also mentioned in the previous section, TensorFlow.js provides a set of pre-trained models. These models are hosted on NPM and unpkg so they can easily be used in any project after an installation step. This is why for our third experiment, we tried to identify other models that could be appropriate for our pedestrians detection use case. As we studied again the models in the Images category, another one seemed to be a good fit for: BodyPix.

4.3.1 The Model - overview

BodyPix is a person and body part segmentation model built for in browser real-time detection (we will also talk about the drawbacks of using it on mobile). This model can be used to segment an image into pixels that are and are not part of a person, and into pixels that belong to each of twenty-four body parts. It works for multiple people in an input image or video.

4.3.2 Mobile Integration

Similarly, the model can be installed by typing the following command:

```
$ npm install @tensorflow-models/body-pix
```

It can be used in the project after following the same 3 main steps:

1. **import** model

```
import * as bodyPix from "@tensorflow-models/body-pix";
```

2. **load** model

```
const model = await bodyPix.load(/**optional arguments**/);
```

3. **segment** image

```
const segmentation = await model.segmentPerson(image,
                                    {flipHorizontal: false,
                                     internalResolution: 'medium',
                                     segmentationThreshold: 0.7});
// other options:
//   - model.segmentPersonParts
//   - model.segmentMultiPerson
//   - model.segmentMultiPersonParts
```

Architecture	quantBytes=4	quantBytes=2	quantBytes=1
ResNet50	~90MB	~45MB	~22MB
MobileNetV1 (1.00)	~13MB	~6MB	~3MB
MobileNetV1 (0.75)	~5MB	~2MB	~1MB
MobileNetV1 (0.50)	~2MB	~1MB	~0.6MB

Figure 4.1: Different model sizes depending on architecture

By default, BodyPix **loads** a MobileNetV1 architecture with a 0.75 multiplier. This is recommended for computers with mid-range/lower-end GPUs. A model with a 0.50 multiplier is recommended for mobile. The ResNet architecture is recommended for computers with even more powerful GPUs.

The developer can also load other versions of the model, by specifying the architecture explicitly in bodyPix.load() using a ModelConfig dictionary with the following possible parameters:

- **architecture**: the base model 'ResNet50' or 'MobileNetV1'
- **outputStride**: 8, 16, 32. Stride values 16, 32 are supported for the ResNet architecture and stride 8, and 16 are supported for the MobileNetV1 architecture. The smaller the value, the larger the output resolution, and more accurate the model at the cost of speed. A larger value results in a smaller model and faster prediction time but lower accuracy.
- **multiplier**: the float multiplier for the depth (number of channels) for all convolution ops, and can be 1.0, 0.75 or 0.50. Used only by by the MobileNetV1 architecture. The larger the value, the larger the size of the layers, and more accurate the model at the cost of speed. A smaller value results in a smaller model and faster prediction time but lower accuracy.
- **quantBytes**: the number of bytes per float used for weight quantization, possible options:
  - 4: (no quantization) highest accuracy and original model size.
  - 2: slightly lower accuracy and 2x model size reduction.
  - 1: lower accuracy and 4x model size reduction.
- **modelUrl**: optional

The following summary contains the sizes of the models when using different quantization bytes and multipliers:

However, the documentation specifies two recommended configurations for loading the model:

- ResNet (larger, slower, more accurate):

```
const model = await bodyPix.load({architecture: 'ResNet50',
                                   outputStride: 32,
                                   quantBytes: 2});
```

- MobileNet (smaller, faster, less accurate):

```
const model = await bodyPix.load({architecture: 'MobileNetV1',
                                   outputStride: 16,
                                   multiplier: 0.75,
                                   quantBytes: 2});
```

The **second model** managed to have the fastest prediction in our application: 44 s. This is more than 40 times slower than our second experiment (approx. 1 second per prediction). Based on the table regarding the architecture we have also tried implementing the other configurations without having any success in obtaining a faster prediction (only values with a minimum of 50 seconds).

Undoubtedly, this is the point where our third experiment ended, because of the bad performance in terms of time. A reaction that takes more than 44 seconds for a speeding car could be even more dangerous than a lack of reaction. In other words, we must keep in mind that the faster the answer, the less threatened human lives are, and we want to minimize the risk as much as possible.

Chapter 5

Application (numerical validation)

5.1 Methodology

- What criteria are we using to evaluate our method?

  Since our approach is based on the pre-trained version of the COCO-SSD model, we performed our experiments having in mind three aspects:

  - accuracy

  - runtime (time for processing)

  - feasibility of our application in the context of driving assistance field

  The reason behind this was to balance these three.

- What specific hypotheses does our experiments test?

  Due to the COCO-SSD model being very general,it is less likely to break down when applied to new domains or unexpected inputs. As a result, we can assume that the model will perform well on our task of detecting pedestrians, vehicles and other participants to the traffic. Starting with this assumption, our aim is to identify how well we can apply these existing algorithms of pre-trained model to a known objects detection problem in order to integrate in an application that provides driving assistance support.

- What are the dependent and independent variables?

  Independent variables: Pre-trained model, System environment Dependent variables: DataSets, program output

- What is the training/test data that was used, and why is it realistic or interesting?

Our dataset contains real time images taken from in-motion video footage, making up for a very realistic experience, perfectly tailored to a real time driving assistant application.

5.2 Data

For the dataset, we selected some subsets from the COCO datasets. We will dive in details below.

COCO refers to the "Common Objects in Context" dataset, the data on which the model was trained on. This collection of images is mostly used for object detection, segmentation, and captioning, and it consists of over 200k labeled images belonging to one of 90 different categories, such as "person," "bus," "zebra," and "tennis racket."

Features of the COCO dataset

- Object segmentation with detailed instance annotations

- Recognition in context

- Superpixel stuff segmentation

- Over 200â000 images of the total 330â000 images are labeled

- 1.5 Mio object instances

- 80 object categories, the âCOCO classesâ, which include âthingsâ for which individual instances may be easily labeled (person, car, chair, etc.)

- 91 stuff categories, where âCOCO stuffâ includes materials and objects with no clear boundaries (sky, street, grass, etc.) that provide significant contextual information.

- 5 captions per image

- 250â000 people with 17 different keypoints, popularly used for Pose Estimation

5.3 Results

Each algorithmically generated result, such as an object bounding box or segment, is stored separately in its own result struct. This singleton result struct must contains the id of the image from which the result was generated (a single image will typically have multiple associated results). Results for the whole dataset are aggregated in a single array. Finally, this entire result struct array is stored to disk as a single JSON file.

Our initial intuition was to take a look at the original benchmark of the author of COCO-SSD algorithm which was completed on the COCO dataset and was run on a Pascal Titan X. The COCO-SSD algorithm itself has multiple versions that are suited for specific usecases.

```
               Class     Images     Labels          P          R     mAP@.5 mAP@.5:.95: 100% 157/157 [05:54<00:00,  2.26s/it]
                 all       5000      36335      0.711      0.602      0.649      0.453
Speed: 0.2ms pre-process, 60.9ms inference, 2.1ms NMS per image at shape (32, 3, 640, 640)
```

Figure 5.1: Algorithm output for COCO dataset

```
Accumulating evaluation results...
DONE (t=17.19s).
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.465
 Average Precision  (AP) @[ IoU=0.50      | area=   all | maxDets=100 ] = 0.655
 Average Precision  (AP) @[ IoU=0.75      | area=   all | maxDets=100 ] = 0.508
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.297
 Average Precision  (AP) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.515
 Average Precision  (AP) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.585
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=  1 ] = 0.359
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets= 10 ] = 0.594
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=   all | maxDets=100 ] = 0.649
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= small | maxDets=100 ] = 0.476
 Average Recall     (AR) @[ IoU=0.50:0.95 | area=medium | maxDets=100 ] = 0.699
 Average Recall     (AR) @[ IoU=0.50:0.95 | area= large | maxDets=100 ] = 0.779
Results saved to runs/val/exp13
```

Figure 5.2: Average precision for YOLOv3

For the metrics used in comparison, we focused on AP (Average precision) which is a popular metric in measuring the accuracy of object detectors like Faster R-CNN, SSD, etc. Average precision computes the average precision value for recall value over 0 to 1.

5.3.1 Results for YOLOv3

We present the results for the YOLOv3 using COCO dataset in the following figure:

Output images samples produced with YOLOv3:

We observe that in general, the algorithm produces very good results and detects most of the pedestrians, vehicles and obstacles. However, just like in the case of the smaller dataset, we notice the

Figure 5.3: Output image sample for YOLOv3 1

Figure 5.4: Output image sample for YOLOv3 2

Figure 5.5: Output image sample for YOLOv3 3

Figure 5.6: Output image sample for YOLOv3 4

Model name	Speed (ms)	COCO mAP	Outputs
SSD MobileNet V2 FPNLite 320x320	22	22.2	Boxes

Table 5.1: Results on COCO dataset

Model summary	minival mAP	test-dev mAP
(Fastest) SSD w/MobileNet (Low Resolution)	19.3	18.8
(Fastest) SSD w/Inception V2 (Low Resolution)	22	21.6
(Sweet Spot) Faster R-CNN w/Resnet 101, 100 Proposals	32	31.9
(Sweet Spot) R-FCN w/Resnet 101, 300 Proposals	30.4	30.3
(Most Accurate) Faster R-CNN w/Inception Resnet V2, 300 Proposals	35.7	35.6

Test-dev performance of the "critical" points along our optimality frontier.

Figure 5.7: mAP for Faster R-CNN, R-FCN, and SSD

darker images will have a significantly lower precision than bright images (notice how in figure 5.6, the car is not detected and also some persons are not detected).

As s strong point for the algorithm, we could say that it manages to successfully tell apart trucks and larger vehicles of normal sized cars, and that means we can easily warn the driver of some imminent collision with a large vehicle to avoid an accident. (Figure 5.5)

5.3.2 Results for COCO SSD and comparison with other models

TensorFlow 2 provides a collection of detection models pre-trained on the COCO 2017 dataset. The one we are using for mobile object detection has the results detailed in table 5.1.

Why have we chosen this backbone for our object detector? We believe that the crucial question is not the classical one regarding which detector is the best, as this might not be possible to answer. Instead, we are looking for the best balance between speed and accuracy that our problem needs.

A report conducted by Google Research offers a survey paper to study the tradeoff between **speed and accuracy** for Faster R-CNN, R-FCN, and SSD. The results show that SSD on MobileNet has the highest mAP among the models targeted for real-time processing, as Figure 5.7 shows.

On the other hand, since we are giving the model images as input, we should also refer to the **object size** when measuring the performance. For large objects, SSD performs pretty well even with a simple extractor. SSD can even match other detectors' accuracy using better extractor. But SSD performs much worse on small objects comparing to other methods. The following example is a screenshot from our application showing that no objects were detected by COCO SSD in a blurry picture with smaller objects, while, in our previous experiment, the objects were successfully detected by YOLO (see figures 5.8, 5.9).

At the same time, COCO SSD outperforms YOLOv3 in terms of **time**. Our application is designed to run on mobile, therefore it is not possible to store the model on the device, it needs to be accessed

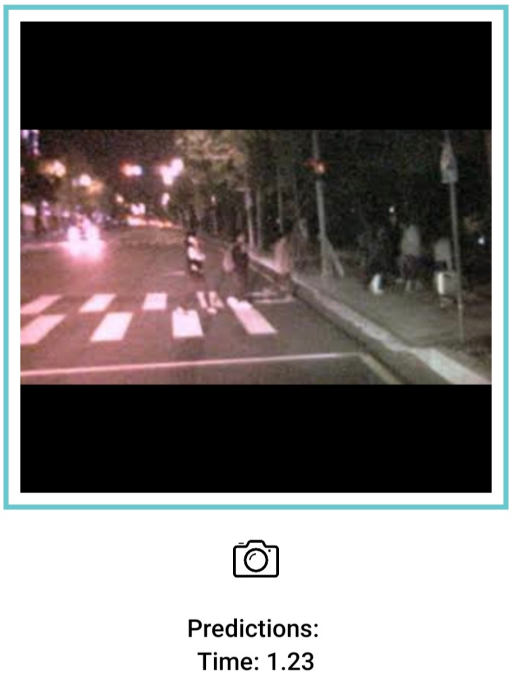

Figure 5.8: Output from COCO SSD for blurry, small objects

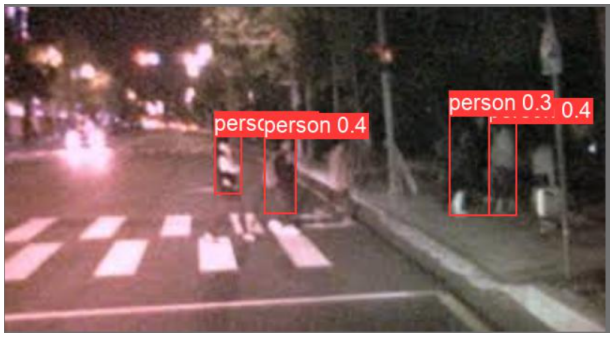

Figure 5.9: Output from YOLOv3 for blurry, small objects

through an endpoint (REST server in our first experiment). For a clearer understanding of the matter, we propose a step by step list describing the process for running each of the two methods.
In order to make a prediction using YOLOv3, the following steps have to be pursued:

1. rescale the image
2. send it to the server
3. decode the image
4. run prediction
5. encode the image
6. send prediction response

All these steps turned out to be very time consuming, as they all add a lot of overhead to the prediction task. Not to mention the cold start issue - when no request has been made to the server and the model is not loaded yet, loading it will take a few more seconds before running the actual prediction. In order to make the prediction using COCO SSD, TensorFlowJS is the fastest choice for mobile. The model is loaded once the app is first accessed, with no need for making a request for getting the predictions. Thus, the steps are:

1. convert the image to string
2. convert to tensor
3. get prediction

The pipeline is run on the client side entirely, with no other exhaustive and time consuming side tasks. On the next page there are a couple of examples from after running COCO SSD on the mobile application.

In the picture 5.11 at page 22, a very good result and also great performance from COCO SSD can easily be observed. The objects that are smaller have also successfully been detected here, as they are not blurry anymore (as in Figure 5.8). Moreover, the time difference between COCO SSD prediction using TensorFlowJS and YOLOv3 using a Python Flask REST server is huge - 4x the number of seconds that the SSD needs for the second option to get to the results. More details regarding the actual value of a millisecond in the context of driving will be described in the next section.

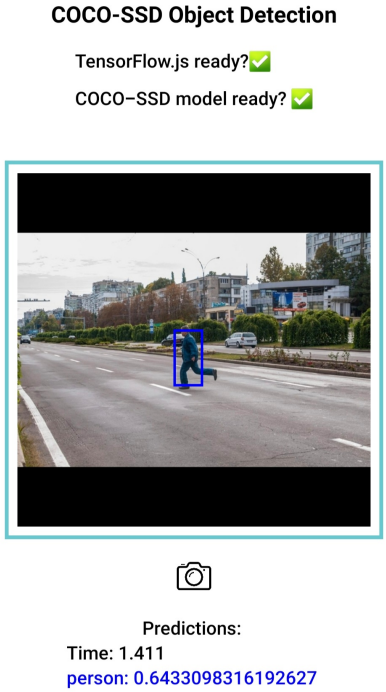

Figure 5.10: COCO SSD prediction time example 1 - **COCO SSD: 1.4 (s) | YOLOv3: 5.32 (s)**

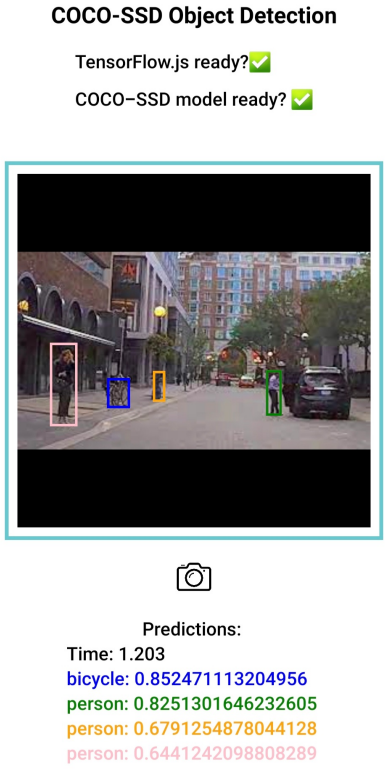

Figure 5.11: COCO SSD prediction time example 2 - **COCO SSD: 1.2 (s) | YOLOv3: 4.7 (s)**

5.4 Discussion

5.4.1 Hypothesis, comparison and interpretability

Our hypothesis is pedestrians detection using a mobile device, with the remark that our strategy should also comply as much as possible with the driving process (be fast).

It is very difficult to get high accuracy from a model that was designed to run on mobile phones. Therefore, as we also mentioned in the previous section, the best balance between speed and accuracy had to be a primary focus point in all our choices. We believe that our hypothesis was accomplished by getting quite good predictions in the fastest possible time (approx. 1 second).

There are plenty of other models that have very good performance in terms of accuracy on object detection. While accurate, these approaches have been too computationally intensive for embedded systems and, even with high-end hardware, too slow for real-time applications. Taking into consideration that it takes more than 5 seconds to detect the pedestrian, we moved on from the first tiny YOLOv3 experiment to a more faster, mobile suited version, COCO SSD and TensorFlowJS. If we were to refer to the other configurations that were also available for our Mobilenet backbone, Figure 5.12 shows that our SSD model is the best available option from the current TensorFlowJS Object Detection Model Zoo: By using SSD, we only need to take one single shot to detect multiple objects within the image, while regional proposal network (RPN) based approaches such as R-CNN series that need two shots, one for generating region proposals, one for detecting the object of each proposal. Thus, SSD is much faster compared with two-shot RPN-based approaches.

Apart from its standalone utility, we believe that our monolithic and relatively simple SSD model provides a useful building block for larger systems that employ an object detection component.

Self-driving vehicles are one of the most exciting and impactful applications of AI.

5.4.2 Other aspects

More than 35,000 people die every year in motor vehicle crashes in the US alone. Since self-driving vehicles can theoretically react faster than human drivers and don't drive under the influence, text while driving, or get tired, they should be able to dramatically improve vehicle safety. They also promise to increase the independence and mobility of seniors and others who cannot easily drive.

On the other hand, people use commonsense reasoning to handle unexpected phenomena while driving: if we see a ball roll onto the street, we know to look out for children chasing the ball. People do not learn about all these possible corner cases in driving school. Instead, we use our everyday commonsense reasoning skills to predict actions and outcomes.

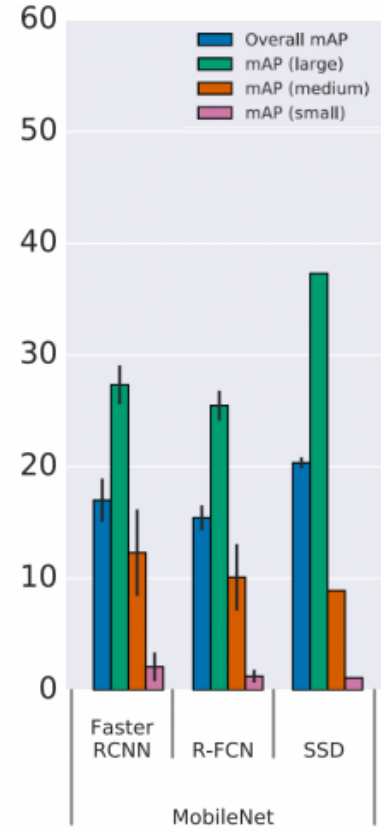

Figure 5.12: mAP comparison between different Mobilenet models

Unfortunately, no one knows how to build commonsense reasoning into cars, or into computers in general. Instead of commonsense reasoning capabilities, ADAS (Advanced Driver Assistance Systems) developers must anticipate and integrate every possible situation. Machine learning can only help to extend the knowledge about any kind of situations, and define better and safer behaviours in case of the unknown.

Chapter 6

Conclusion and future work

The problem of pedestrians detection is still a problem nowadays. Even though a lot of progress has been made in this field, there still remain some aspects that are yet to be improved. As a closing word, we can conclude that our system for pedestrians detection represents a real benefit nowadays, in which the safety of people in traffic it is really important.

The use of COCO SSD can be seen as a strength, being easy to integrate and maintain in a system like this, while performing good the given task. As seen during this paper, we performed more experiments in order to choose COCO SSD as our final approach. We considered the trade-off between accuracy and performance which is an important aspect for real-time computer vision tasks.

We believe we have demonstrated an accurate representation of the pedestrians detection method in autonomous driving field, yielding quite satisfactory results and raising some interesting points of discussion for future enhancements.

We moved the processing of the image and the detection part on the client in order to decrease the overall processing time and deliver faster results, which bring us closer to a real-time experience.

We notice on darker images we obtain poorer results and it is hard to tell apart individual persons in a larger group of people and minor improvements can be made if we increase the brightness of the images used, producing slightly better results in qualitative terms.

In addition to the presented implementation of the solution, we would like to include several future improvements such as: the possibility to take a picture in real time, an interactive 3D representation of the results and the inclusion in the study of different races and ethnicities in the pedestrian detection part and analyzing how the accuracy of the results obtained on such groups of people differ from our initial results.

We consider that this paper would offer support and encouragement for future endeavors in the pedestrians detection.