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

Auto Assistant

– MIRPR report –

Team members
Lung Andreea, 258/2, lais2064@scs.ubbcluj.ro
Katona Ildiko-Noemi, Software Engineering, 258/1, kat.ildiko.noemi@gmail.com
Nagy Barnabás, Software Engineering, 258/2, n.barnabas@yahoo.com
Popa Cătălin, Software Engineering, 258/2, castle2145@gmail.com

2020-2021

Abstract

In the context of autonomous driving, personal driving assistants have gained enormous momentum in recent days, making the driving experience altogether a much more efficient, safe and easy task. A core problem in this field and the particular problem we are trying to solve in this paper is tracking surrounding vehicles and participants to the traffic, detecting road signs and offering in-app information that would help the driver make certain decisions in a more efficient way. In this study, we make use of a pre-trained model (TinyYoloV3) in order to run several experiments on data from the INRIA and Caltech datasets. We measure these results against state of the art and draw a conclusion after analyzing the different possible optimizations. We also discuss the philosophical aspects pertaining to autonomous driving and give some future directions of research.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 What? Why? How?

Many times while we are driving, we are faced with bad lightning or poor weather conditions or simply tiredness that makes us less alert and more prone to making mistakes in traffic. A single moment of lessened attention could easily lead to a car accident and loss of human lives. In an area like this, where the human factor plays such a tremendous role, it is important that we try to increase the confidence in our vehicle's safety and provide a tool for drivers to assist them in making certain decisions.

- ***What?*** - The problem of an Object Detection System, tailored to pedestrians and vehicle / road signs detection, which would further give us the possibility to warn the user of an imminent collision or automatically emergency break the car. The system would be a useful personal assistant, offering in-app information also about weather conditions, speed and traffic restrictions etc.
- ***Why?*** - Avoid car crash situations and minimise the chances of collision in traffic, provide useful support in tracking surrounding vehicles and participants to the traffic, anticipate other cars' actions and react accordingly-> improved safety factor and driver's comfort.
- ***How?*** - An intelligent algorithm based on AI classification systems. The classifier will be able to distinguish pedestrians, vehicles, street signs and provide relevant feedback to the user based on this.

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.

The second chapter describes the scientific problem in more detail.

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

In chapter 4 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 5 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 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 the points made throughout this paper and the results obtained and presents some guidelines towards future work in this area.

Chapter 2

Scientific Problem

2.1 Problem definition

The problem of road obstacle detection has been addressed by many major car manufacturers like Audi, BMW, Mercedes-Benz etc., integrating them in new car models in order to improve crash avoidance technology and increase confidence in their cars. However, most of these systems report poor results in badly lit environments and are often faulty in their detection processes.

The aim of this paper is to present a means of detecting pedestrians, vehicles and road signs in any kind of conditions and tie this to the current weather state to suggest to the user the most appropriate actions to be taken in certain traffic situations or to react automatically to them (further improvement).

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.

Why an intelligent algorithm is more suitable for this? - The shape of the pedestrian, for example, is rarely predictable, it may be moving or standing, may differ in color or position, creating endless possibilities of representation. Thus, we need a more complex system of identifying such obstacles. The two main options are:

- Feature Classifier approach - a classifier based on predefined features
- Deep Learning approach - a classifier with self learned features

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.

2.2 Challenges

The challenges we have faced in putting together an algorithm that would help us achieve this goal are related to the specific use case of working with pre-trained models. The models themselves are pretty large in size and take up a great deal of resources to run.

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. 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.

Lack of sufficient resources to run the algorithm was an imminent problem that we have come across, as well as the deployment part, where we have found it is difficult to deploy our app in the cloud due to lack of free options for dedicated GPU servers.

We have also placed a lot of focus on developing a "Diversity and inclusion" part, in which we try to compare the results the algorithm yields on people from different races compared to white people and how this could make our algorithm biased in terms of inclusion and multi-cultural coverage. This proved to be quite a challenging task, especially for arab women, where the algorithm would often fail to perform well at all.

Chapter 3

State of the art/Related work

With the growth of the vehicle technology, grew the incorporation of intelligent devices in driving. In order to provide safety and to reduce traffic accidents driving assistance systems(DASs) have been developed with the help of artificial intelligence(AI). Pedestrian detectors and driving assistants are ones of the features of DASs. The following articles provide solutions for these topics.

1. The first selected paper is "Driving situation-based real-time interaction with intelligent driving assistance agent" [9] by Young-Hoon Nho.

   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).

   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.

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

2. The second paper is the "Traffic signs recognition for driving assistance" [10] by Yatham Sai

Sangram Reddy, Devareddy Karthik, Nikunj Rana, M Jasmine Pemeena Priyadarsini, G K Rajini and Shaik Naseera.

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

Datasets: Each of the Traffic signs is recognised using a database of images of numbers and symbols used to train the KNN classifier using open CV libraries. (we are not given an exact size of this database or information about where the images come from) 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.

Obtained performance: 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.

Chapter 4

Investigated approach

4.1 Overall flow

For solving our problem, we have decided to make use of a pre-trained model instead of training our own classifier from scratch. A pre-trained model is one that was trained on a large benchmark dataset and solves a similar problem to the one we want to solve. Many times, the computational cost makes it a much better decision to import a pre-trained model (e.g. VGG, Inception, ResNet) and use this for our own problem.

Figure 4.1: High level diagram of the system architecture [2]

Usually, when using pre-trained models, we will want to perform transfer learning - that is, using previous information that comes from training the model on a large benchmark dataset and re-training on our own data to use the same model for a second task. This requires a repurposing of the model for our own needs, which usually implies removing the original classifier, adding one that fits our purposes, and finally fine-tuning our model.

When fine-tuning the model, we can either re-train the entire model, train some layers and leave the others frozen or freeze the convolutional base. For our case, we will employ the third strategy. This means, we will use the pre-trained model as a fixed feature extraction mechanism. This is useful in cases where, either we don't have much computational power at our disposal or we have a small dataset or the problem we are solving is very similar to the original one solved by the pre-trained model.

Our case falls into the third situation. 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. We provide a high level view of the whole architecture of the system below:

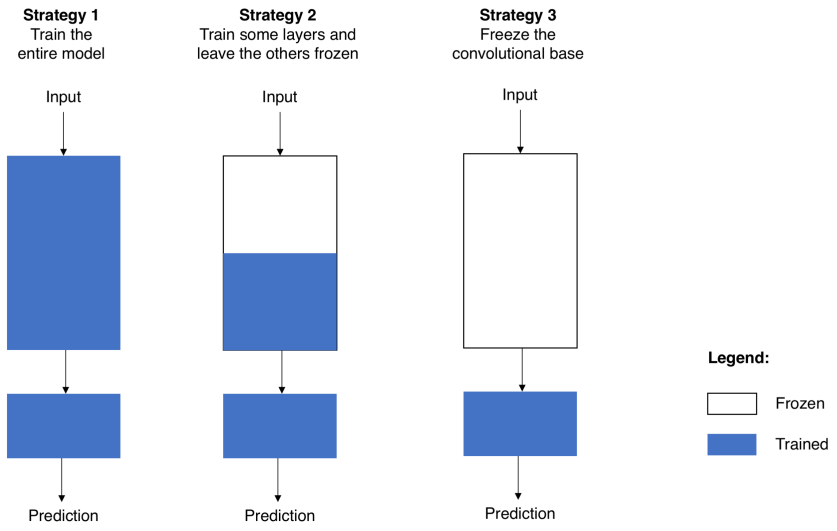

Figure 4.2: Fine tuning options of pre-trained models. [7]

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 The algorithm

For our image detection algorithm we decided to use an existing library to perform the image detection process and integrate the results into an accessible, easy-to-use application tailored to the user's needs.

ImageAI [6] is a powerful tool offering state-of-the-art machine learning algorithms capable of detecting and recognizing 80 different kind of common everyday objects, by use of pre-trained models like RetinaNet, YOLOv3 and TinyYOLOv3. These models were trained on the COCO dataset. In our project, we use the TinyYOLOv3 [11] model. This model is a simplified version of the YOLOv3. The YOLOv3 architecture is based on a variant of DARKNET, with a signficant number of 1x1 and 3x3

convolution kernels used for feature extraction.

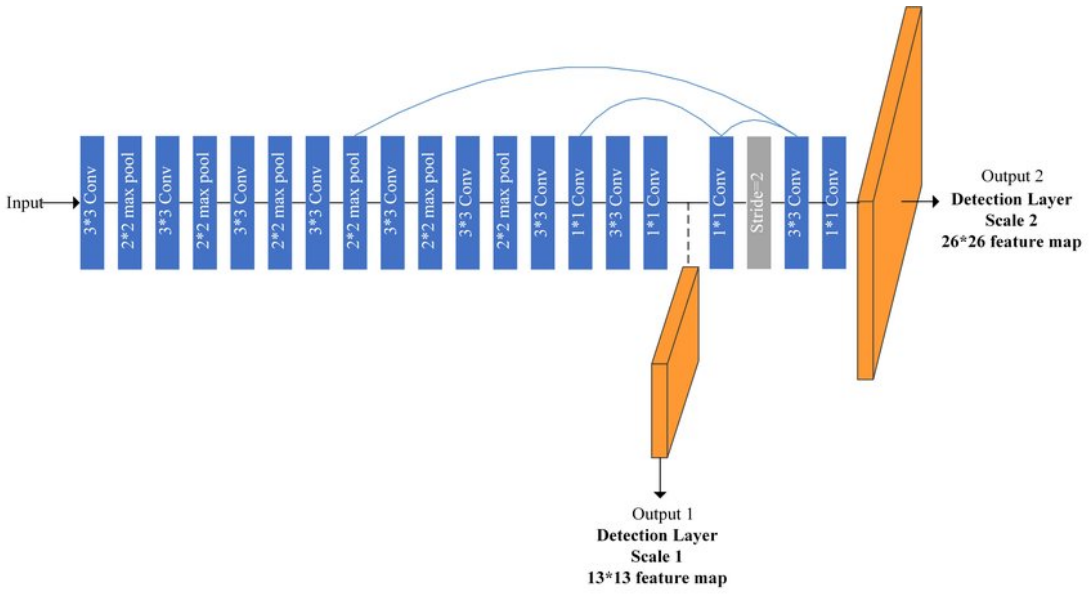

Figure 4.3: The architecture structure of TinyYOLOv3. [5]

In TinyYOLOv3, we see a reduction in the number of convolutional layers, its basic structure having only 7 convolutional layers, following for features to be extracted by only using a small number of 1x1 and 3x3 convolutional layers. TinyYOLOv3 uses the pooling layer instead of YOLOv3's convolutional layer with a step size of 2 to achieve dimensionality reduction. However, its convolutional layer structure still uses the same structure of Convolution2D + BatchNormalization + LeakyRelu as YOLOv3. The structure of TinyYOLOv3 is shown in Figure 4.3.

For the training process, the loss function used by TinyYOLOv3 is the same as that of YOLOv3, which is mainly composed of the position of the prediction frame (x, y), the prediction frame size (w, h), the prediction class (class), and the prediction confidence (confidence). The expression is given below:

$$loss = \frac{1}{n}\sum_{i=1}^{n} loss_{xy} + \frac{1}{n}\sum_{i=1}^{n} loss_{wh} + \frac{1}{n}\sum_{i=1}^{n} loss_{class} + \frac{1}{n}\sum_{i=1}^{n} loss_{confidence} \tag{4.1}$$

where n is the total number of trained targets, and functions represent the loss functions for each factor in equation.

4.3 How we use it for our problem

Getting started using this library is as simple as the next few lines of code:

```
detector = ObjectDetection()
```

```
detector.setModelTypeAsYOLOv3()
detector.setModelPath( os.path.join(execution_path , "yolo.h5"))
detector.loadModel()
```

where the first line is the creation of a new **ObjectDetection** class instance, the second line is setting the model type to YOLOv3, setting the model path and loading the model on the last line.

Employing the model for our specific problem is done by customizing the type of object we want to be detected in our input images, then passing it in to the **detectCustomObjectsFromImage** function:

```
custom_objects = detector.CustomObjects(car=True, motorcycle=True)
detections = detector.detectCustomObjectsFromImage(custom_objects=custom_objects,
            input_image=os.path.join(execution_path , "image3.jpg"),
            output_image_path=os.path.join(execution_path , "image3custom.jpg"),
            minimum_percentage_probability=30)
```

If we wish to improve the speed with which the algorithm processes the images we have the possibility to fine-tune the **minimum_percentage_probability** parameter in the **detectCustomObjectsFromImage** function. A lower value will result in showing more objects and improving speed, while increasing the value will ensure that objects with the highest accuracy are detected and it will be slower. The default value is 50. We can also couple this with the **detection_speed** parameter of the **loadModel** method, which can take one of five values: **"normal"**(default), **"fast"**, **"faster"** , **"fastest"** and **"flash"**.

4.4 Example of the algorithm in practice

In Figure. 4.4 example image we can observe the bounding boxes of the detected objects, as well as the probability for each of them. These are also available in code, so we can make use of these numbers and data.

4.5 Useful tools

Pedestrian Detection with OpenCV

OpenCV [13] is an open source computer vision and machine learning software library. It was created to provide a common infrastructure for computer vision application. It comes with a pre-trained

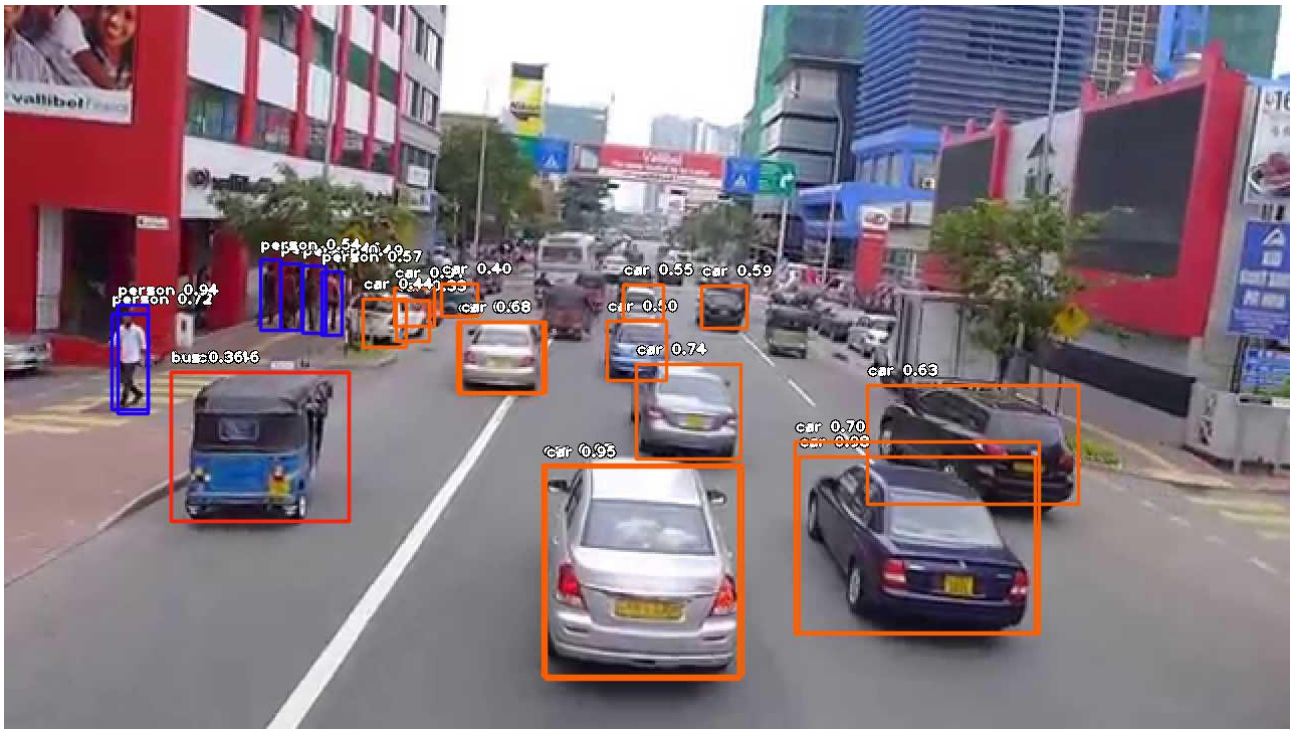

Figure 4.4: Example output image of TinyYOLOv3 algorithm.

histogram of oriented gradients and linear support vector machine which can be used to detect and identify pedestrians.[12]

Object Detection with ImageAI in Python

ImageAI [6] is a Python library built to help developers to create applications and systems with self-contained deep learning and Computer Vision capabilities using a few lines of code.

Using pre-trained models the ObjectDetection class of the ImageAI library contains functions to perform object detection on any image or set of images.[8]

Chapter 5

Application (numerical validation)

In this chapter we are going to explain the experimental methodology and present the results obtained by employing our model as well as a comparison with state of art approaches.

5.1 Requirements

The main list of functionalities required for the application are:

- The user should be able to take a picture or to upload one from the gallery and send it to the application.
- The application should be able to recognize cars, persons and traffic lights and display a rectangle around the seen cars in the picture.
- The application should display the speed of the car the user is in.
- The system should display the weather of the current day on the interface.
- The application should show a warning message for speed limit reached or bad weather conditions.

5.2 Methodology

- ***Evaluation criteria*** - For the purpose of evaluating our model performance, we place our focus on a few metrics that we consider more relevant, namely:
  - accuracy
  - runtime (using dedicated graphics)
  - feasibility of our application in the context of driving assistance field

- ***Hypotheses and experimental methodology*** - Due to the YOLO model being highly generalizable, 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 models (TinyYoloV3 and YOLOv3) to a known objects detection problem in order to integrate in an application that provides driving assistance support.

- ***Dependent and independent variables***

Independent variables	**Dependent variables**
System environment	Test dataset
Pre-trained model	Program output

Table 5.1: The dependent and independent variables

- ***Particularity of dataset*** - 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.

5.3 Data

5.3.1 The INRIA dataset

As an initial stage, we have tested our algorithm on a small dataset comprised of 288 pictures. The dataset is called INRIA [4] and is a popular pedestrian detection dataset, offering both data for training and testing.

The initial stage testing was carried out focusing on pedestrians, with vehicle detection being tested on 10 images only, following that in later increments we increase this number as well.

5.3.2 Caltech Pedestrian dataset

The second iteration of our algorithm testing was carried out on the CALTECH Pedestrian dataset [1], a big scale dataset consisting of approximately 10 hours of 640x480 30Hz video footage taken from a vehicle driving through regular traffic in an urban environment. The set totals about 250,000 frames (in 137 approximately minute long segments) with a total of 350,000 bounding boxes and 2300 unique pedestrians having been annotated.

The way we used the dataset for our application was by taking one of the subsets, **set01**, and splitting the video files into individual frames in order to obtain the input images for our algorithm. We then ran the algorithm on each of the 6 individual subsets of set01 (totalling about **10,800 images**) and comparing the results obtained for each of them. We expect the results obtained on such a large dataset to be less accurate, especially due to the real life nature of the images and the lower quality.

5.4 Results

5.4.1 Results on INRIA

We present the results for the INRIA dataset in the following figure:

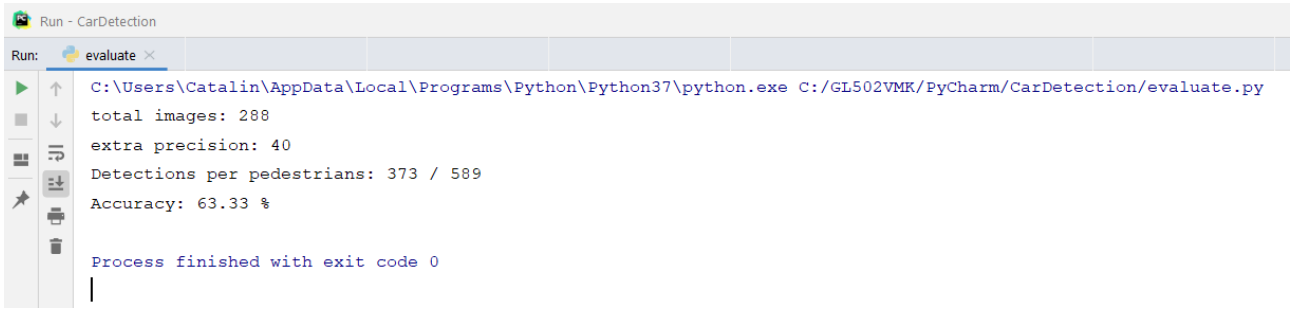

Figure 5.1: Algorithm output for the small size dataset.

Here, the extra precision argument is an indicator of the level of additional pedestrians detected by our tool in comparison to the number expected (this is due to the fact that the images are not completely annotated, as specified in the dataset docs).

Output images samples produced with INRIA:

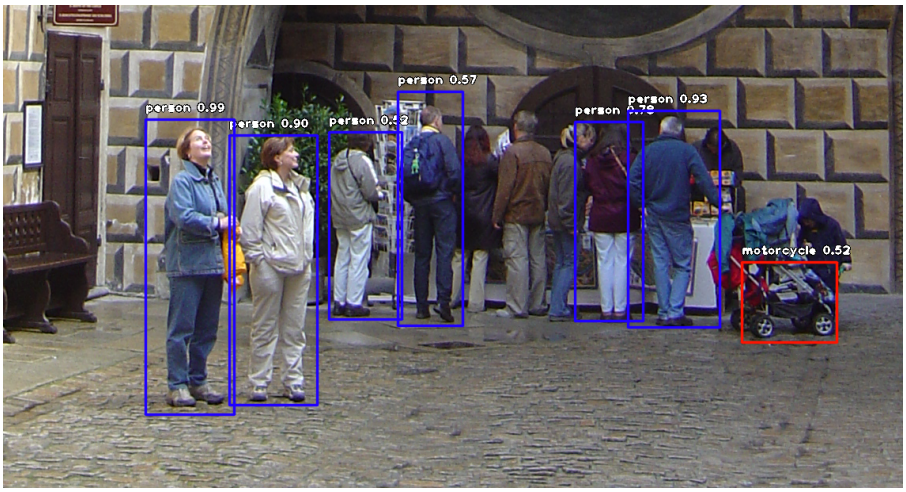

Figure 5.2: Output image sample for INRIA 1

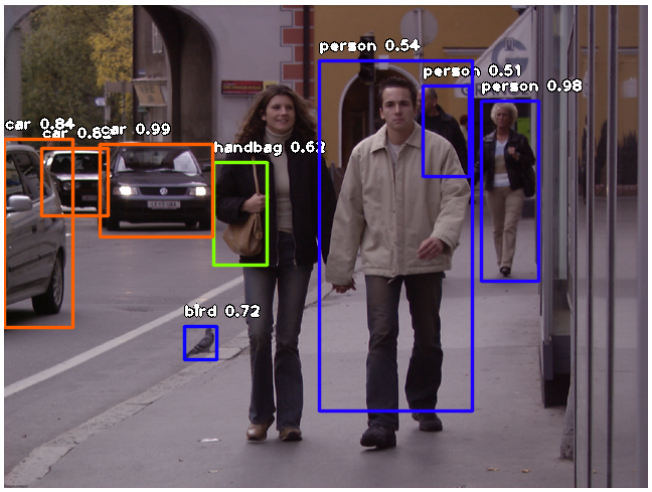

Figure 5.3: Output image sample for INRIA 2

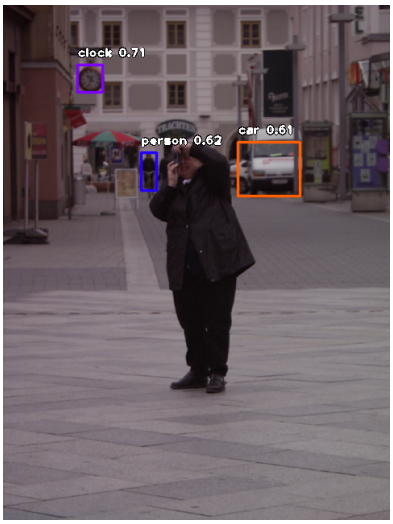

Figure 5.4: Output image sample for INRIA 3

Interpretation - We observe that in general, the algorithm produces very good results and detects most of the pedestrians/vehicles/obstacles. However, in the third sample output image (Figure 5.4), we notice an interesting phenomenon. The clock, the car and the person in the far away distance are accurately detected, but the person who is in the fore-front and is definitely larger than the objects in the back is missed by the algorithm. We attribute this fact to the rather dark stance of the figure, as well as the fact the face is not visible, making up for a spot of color rather than an individual person.

5.4.1.1 Alternative method results

We also performed a test using OpenCV's Scalable Vector Machine detector initialited using the included 'default people detector' preset. The result was as follows:

```
dataset: INRIA
total images: 288
extra precision: 36
Detections per pedestrians: 374 / 589
Accuracy: 63.5 %
```

The runtime for the whole dataset was only 30 seconds.

5.4.2 Results on Caltech

For the extended dataset we will present the results for each individual subset of the set01 subset (10,800 images in total, each subset of about 1800 photos, 6 subsets).

The experiment which produced the results that follow was ran using a 10% offset. That is, the obtained bounding boxes were considered valid if they were within a 10% margin of error when compared to the expected ones. Thus, ten percent becomes our chosen margin of error.

```
subset: V000
total images: 1711
extra precision: 10
Detections per pedestrians: 984 / 4488
Accuracy: 21.93 %
```

```
subset: V001
total images: 1842
extra precision: 2
Detections per pedestrians: 1950 / 7511
Accuracy: 25.96 %
```

```
subset: V002
total images: 1842
extra precision: 0
Detections per pedestrians: 2855 / 9224
Accuracy: 30.95 %
```

```
subset: V003
total images: 1841
extra precision: 22
Detections per pedestrians: 2459 / 5982
Accuracy: 41.11 %
```

```
subset: V004
total images: 1814
extra precision: 114
Detections per pedestrians: 1002 / 4067
Accuracy: 24.64 %
```

```
subset: V005
total images: 1814
extra precision: 0
Detections per pedestrians: 1249 / 5297
Accuracy: 23.58 %
```

On the large dataset, we obtain a total accuracy of:

- *Accuracy:* ***28.71%*** (10,499 out of 36,569 pedestrians detected)

which could be deemed as relatively low compared to the ***63.33%*** accuracy obtained for the smaller dataset, but this could be attributed to the size of the dataset, as well as the significantly reduced quality of the pictures.

Another interesting experiment we tried was running the full YOLOv3 model on one of the sub-subsets (V003 more specifically) and comparing the results. We obtained the following:

```
total images: 1841
extra precision: 430
Detections per pedestrians: 2526 / 4009
Accuracy: 63.01 %
```

which is significantly larger, so we can assume that the final accuracy would be somewhere around at least 40%, even up to a 45%.

Some sample output images produced by the algorithm run on the Caltech dataset can be seen in Figure 5.5, Figure 5.6 and Figure 5.7.

5.4.2.1 Results using alternative method

The results using OpenCV's HOG SVM using the Default People Detector preset were as follows:

```
subset:V000
total images: 1711
extra precision: 139
Detections per pedestrians: 436 / 3504
Accuracy: 12.44 %
```

```
subset:V005
total images: 1814
```

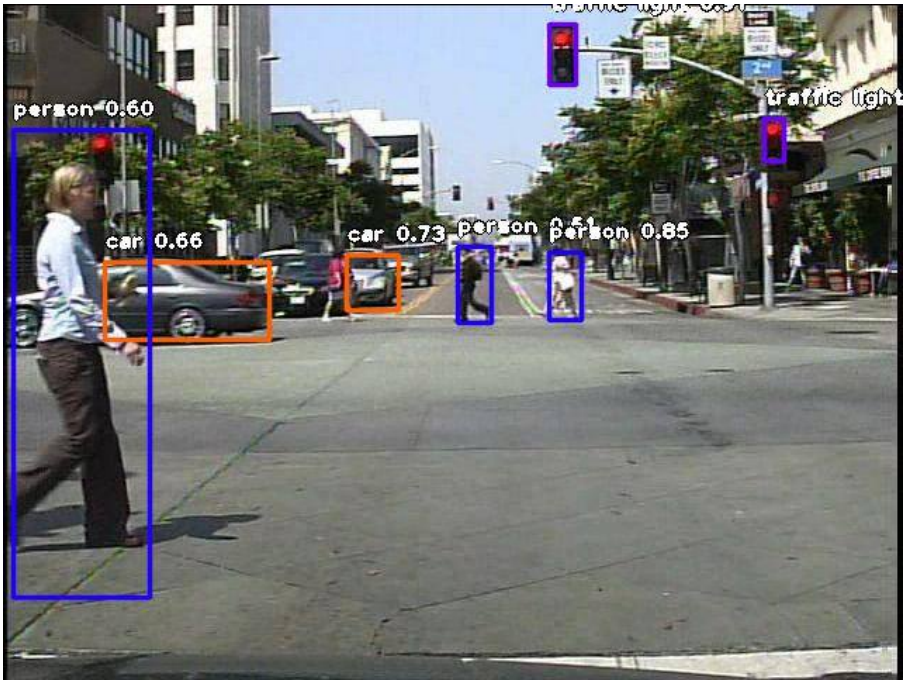

Figure 5.5: Output image sample for Caltech 1

Figure 5.6: Output image sample for Caltech 2

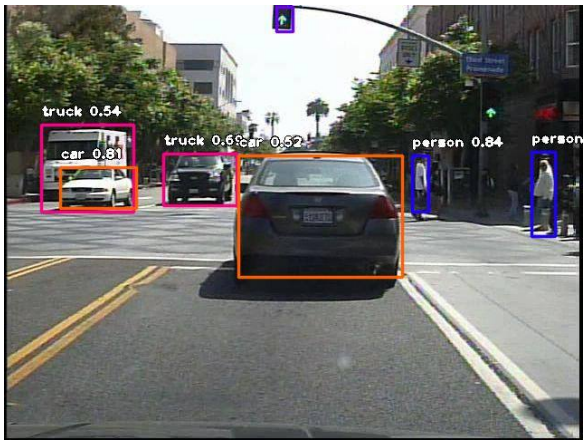

Figure 5.7: Output image sample for Caltech 3

```
extra precision: 79
Detections per pedestrians: 488 / 3230
Accuracy: 15.11 %
```

```
subset:V003
total images: 1841
extra precision: 113
Detections per pedestrians: 362 / 4009
Accuracy: 9.03 %
```

The runtimes were approx. 120 seconds for each presented subset (the subsets are all comparable in size).

We note that the above results were obtained using a much more generous error margin, namely 30%. We tried using the error margin (10%) we used when evaluatig YOLO, but unfortunately, the accuracy plummeted dramatically (at most 3%). Thus, we figured that a more permissive error margin would be suitable in this case.

Interpretation - In these images (Figure 5.5, Figure 5.6, Figure 5.7) it is clear that the algorithm performs well and manages to detect most vehicles, pedestrians and traffic objects like semaphores. However, just like in the case of the smaller dataset, we notice the darker images will have a significantly lower precision than bright images (notice how in the second image, Figure 5.6, none of the car or bus is detected).

One thing to be mentioned here is the way in which the Caltech dataset is annotated that considers a group of more indistinguishable humans in a distance or in a darker area as a group of "people", rather then trying to label each person. This means that our algorithm will not be able to detect such groups of pedestrians, since it is only programmed to detect "persons" in the first place, and that is one thing we need to note.

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.

5.4.2.2 Comparison between the two methods

The HOG SVM method was ran using the following parameters:

- winStride (4, 4)

- padding (8, 8) scale 1.0

Overall, the performance of the HOG SVM method was worse than YOLO (the most dramatic decrease was Caltech's "V003" subset, where YOLO got over 40% accuracy, while HOG SVM got only 9%). Even though the algorithm finished 20% faster for the Caltech's dataset subsets, the overall performance is comparable to YOLO when using the CPU for computations (at least if our results on INRIA are anything to go by).

The biggest advantage to our choice of YOLO implementation is the fact that it has a TensorFlow backend. Thus, it can be easily run both on CPU and GPU. On the other hand, the tested implemetation of HOG SVM is CPU bound, and thus not suitable for our intent.

5.4.3 Runtime analysis and comparison

We begin by listing out our system specifications for better benchmarking:

- ASUS ROG GL502VMK, i7 7700HQ, 16GB DDR4, GTX 1060 3GB (Mobile), Samsung NVMe SSD 500GB, 64−bit

Testing environement:

- Windows 10 version 2004, Python 3.7.9

INRIA (288 images)	**Caltech (1800 images)**
115.08 seconds	150.00 seconds

Table 5.2: INRIA and Caltech datasets runtime comparison

Total runtime for all 6 sub-subsets (set01): ***960.00 seconds***

–> which is just a bit over half an hour.

The reason for the small difference in processing time between the two, despite the large difference in number of images is that the bigger the picture is, the more time it will take for the algorithm to process it. Caltech offers standard resolution images (under 100kB), whilst INRIA has different dimension pictures (between 0,5 MB and a bit under 2,0 MB), hence it explains the processing times.

5.5 Supported platforms

The frontend application was tested on a device running the latest version of Android (Android 11, API Level 30), but it was designed to work with any version of Android from Android 8.0 (API Level 26) above.

According to the Figure. 5.8 we can state that our application will work on 60.8% of all the devices active right now, which we believe to be a great feat. Furthermore, since Android 8 appeared on the market in 2017, it is very unlikely that there might be a notable amount of people not having at least Android 8 on their phone.

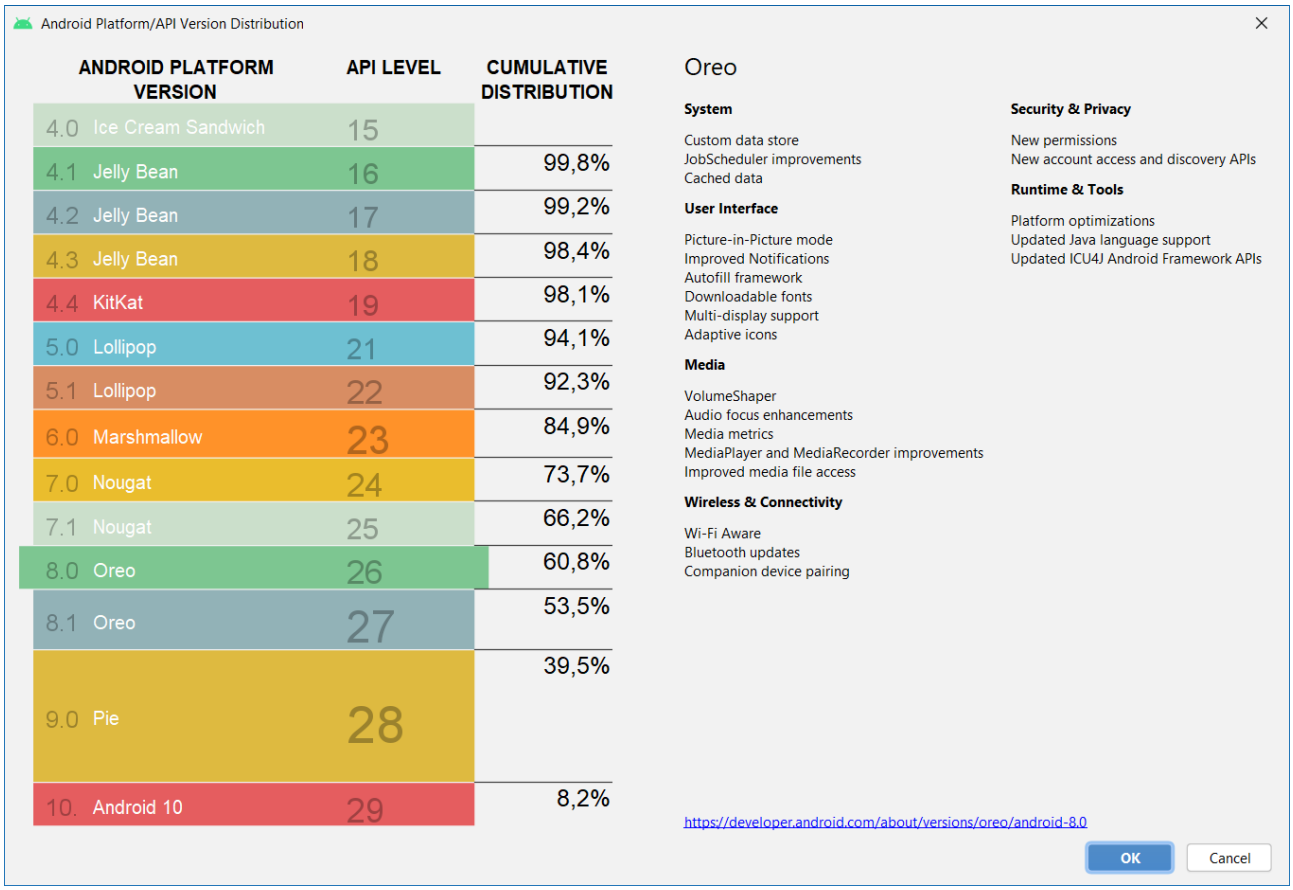

Figure 5.8: Android platform support [3]

5.6 Improvements and optimizations

5.6.1 Algorithm

We consider four important possible optimizations for our approach that we think could yield better results:

- ***resizing the images if they are too large*** - *implemented* - provides increased speed performance, any image sent to the server that is over 480 pixels height gets resized so it fits a 480p size ratio, making the algorithm run much faster. This is only done if the image is not already resized. Thus, it's possible to implement a client which performs the resizing before sending it to

the server. This saves network bandwidth and makes the overall process faster. Also, by checking if the picture hasn't been resized already and only resizing it if it wasn't, the server can support both clients which perform the resizing themselves and clients which don't.

- ***manually increase brightness on our test images*** - by doing so we hope to get better accuracy in detecting pedestrians, since we have noticed reduced performance of our algorithm on low resolution, darker images
- ***only consider the accuracy for detections with large confidence (>= 70%)*** - this way we can gain a speed boost by overlooking less probable detections or false positives
- ***Use the extended YOLOv3 algorithm instead of the tiny version*** - *tested* - although we notice the runtime increases with this, we believe the results are satisfactory from the point of view of the number of pedestrians detected accurately (accuracy jumped up to a 63.01%)

5.6.2 SE/Application

- ***show a warning for bad weather conditions/speed limit reached*** - *implemented* - the application warns the driver when the temperature falls below 0°C and when the speed limit goes over 50 km/h (we consider the application a city driving assistant)
- ***store the received images on the server for further use*** - *implemented* - we can use the images received from the client for different statistics, studies of accuracy etc. - for the moment we have implemented this feature to store the respective images, following that we use these for analytical operations in some future work
- ***send the bounding boxes back from the server instead of the whole image*** - in this case the client would draw the bounding boxes themselves directly on the phone, this would in turn save some bandwidth and make the app speedier
- ***have TinyYolo included in the mobile application)*** - this would mean we would run the detections directly on the phone and the app would become pretty much serverless (however, this is not an easy task to achieve)

5.6.3 UI improvements

With respect to the UI, we present below two screenshots with the UI's evolution.

On the left side, we have the (very basic) initial UI, with just 3 elements: the photo that is being sent for analysis, the button with allows us to load a new photo, and the button which shows us the current wheather.

The new UI, on the other hand, while having a (very) similar design to the first one, has more elements:

- a button which allows us to select the image from the gallery (and also to take a photo on the spot!)
- a label with the current vehicle speed

5.7 Discussion

- ***Is your hypothesis supported?*** - We believe our initially stated hypothesis holds, since we demonstrated the algorithm behaves well on new, unseen, diverse sets of data.
- ***Strengths and weaknesses of our method compared to state of the art***

  Strong points - YOLO learns generalizable representations of objects. When trained on natural images and tested on artwork, YOLO outperforms top detection methods like DPM and R-CNN by a wide margin. The ability to work well with generalized inputs makes it suitable to apply onto new domains or unexpected inputs, which covers very well the field of autonomous driving.

  Another strong point is that YOLO is very fast, compared to other methods, which makes it extremely suitable for real time detection, which is ultimately the goal of this paper.

  What we thought to be also an advantage of the algorithm over others is that it is far less likely to predict false detections where nothing exists.

  Weak points - However, despite all these positive aspects, we still obtained poor results in terms of accuracy (especially on the larger dataset). Some reasons for there has been a loss in accuracy could be that:

  - We used the Tiny model of YOLOv3 and not the full YOLOv3 (we notice that when we tried using the YOLOv3, the accuracy drastically increased)
  - system and environment limitations
  - low quality of the images as a result of being taken in motion (VGA i.e. 640 x 480) –> motion noise (lit areas show up fine in these images, but darker areas are almost indistinguishable and blended out)

- ***Explanation of the results in relation to the underlying algorithm*** - YOLO imposes strong spatial constraints on bounding box predictions since each grid cell only predicts two boxes and can only have one class. This spatial constraint limits the number of nearby objects that our model can predict. Our model struggles with small objects that appear in groups, such as groups of people.

5.8 User interface

For the user interface, we have created an Android mobile app that will communicate with the server by means of a websocket communication. Since we are doing an Android application, it is very easy to integrate it with GPS, weather services and other such features that are very useful for a driving assistant app.

Below is a screenshot of our app's main screen, including an option to get the current weather and speed and how the result of the processed image will be displayed:

In the second screenshot we observe how we can use the camera phone to take photos and send these directly to our server for processing by pressing the "Start identifying objects" button.

5.9 Deployment and CD

5.9.1 Implementation considerations

The back-end of the application is written in Python. The AI part uses ImageAI, which, in turn, makes use of TensorFlow and OpenCV. The communication between the frontend and the backend is handled using SocketIO. Thus, there is a significant number of dependencies, especially when considering the sub-dependencies of the aforementioned libraries.

An important caveat is that the current versions (as of November 2020) of the ImageAI and OpenCV libraries do not work out of the box with the latest version of TensorFlow (2.3). In order to work, our application requires TensorFlow version 1 to be installed (we have tested with version 1.15).

In order to obtain an acceptable level of performance, using a GPU-accelerated version of TensorFlow (and, of course, having a dedicated GPU available) is essential. For future improvements, such as attempting real-time object detection, using a GPU-accelerated version of TensorFlow would be a no-brainer. This becomes a problem when trying to deploy our app in the cloud, because servers with dedicated GPU resources are often only available using paid hosting plans, something which is out of our reach.

Another hurdle is the disk space required: free hosting platforms (such as Heroku) have limitations regarding the maximum size of the (compressed) application image. From our dependencies, TensorFlow alone takes a few hundred megabytes of (uncompressed) disk space (for instance, TensorFlow 1.15 takes a little over 400 MB). What's more, the YOLO models themselves also take a significant amount of disk space, with the exception of the appropriately-named 'tiny' model. One example is the YoloV3 model, which takes around 200 MB.

Last but not least, since the apps sends significant amounts of data over the network and performance is a concern, the server should be geographically located close to the client. A powerful GPU-enabled server in the US would be of little use to clients in Europe.

5.9.2 Back-end

5.9.2.1 CI

Since Python is a weakly typed language, we cannot check that the project 'compiles'. However, it is possible to check the dependency graph (to ensure all the dependencies are compatible with each other) and even perform some static analysis on the code (in this case, some code linting using the established PyLint). For this, we use GitLab's built-in CI/CD pipeline tool, which performs the aforementioned checks. Just like BitStream, it also sends us e-mail notifications for all jobs. A new job is run everytime we push new code.

A possible future improvement would be the inclusion of unit tests into the pipeline.

5.9.2.2 Deployment

As discussed above, cloud deployment would not be a trivial matter (especially due to the GPU power and disk space considerations), so the backend is deployed locally for now. The system requirements would be along the lines of:

- CPU : a recent (2015 or newer), dual-core CPU or better (quad-core recommended)
- RAM : 8GB (or more) of DDR3 memory or better (16GB of DDR4 recommended)
- GPU : a reasonably recent (2013 or newer) dedicated GPU with 2GB VRAM (or more) and CUDA support (a nVidia GeForce GTX 900 series or newer with 3GB of VRAM or more is recomended)
- Storage : SATA SSD (NVMe SSD recommended)
- Python : 3.7

- OS : Windows 10 (64-bit)

We have tested the performance of the AI part on an ASUS ROG GL502VMK laptop which meets the recommended requirements listed above with positive results. We also have a Python Virtual Environment that we have used for testing purposes to speed up the time needed to receive a response from the server, so the application runs much faster.

5.9.3 Front-end

5.9.3.1 CI

We have implemented a simple CI pipeline using BitRise. Using the 14-day free trial, it is possible to perform CI operations on code fetched from a pre-authorized Git repository. As a started point, we check whether the build passes or not.

We have also configured BitRise to automatically perform all CI operations every time new code is pushed to the Git repository. Another nice built-in feature are live e-mail notifications: everytime a build (automated or manual) is performed, an e-mail will be sent to the account owner with the build's result.

Test running is also available out of the box. A future improvement would be writing test cases for the application.

5.9.3.2 Deployment

Being a client app, it is supposed to be installed directly on the end users' devices. In order to keep the app up to date in the long term, it would be possible to list the app on the Google Play Store and provide updates to the end users that way.

5.9.4 Screenshots

Below are screenshots of the tools used for CI.

5.10 Testing

Only manual (empirical) testing was performed. Due to the nature of our application (especially the AI part), there is little room for using established automated testing methods (such as unit tests).

5.10.1 Frontend-Backend communication

This section posed some issues in the sense that sometimes, sending pictures from the backend to the frontend failed. This meant that it was no longer possible to use the AI component of our application. We have tried solving this issue by rewriting the application server to use a different library.

The results were marginally better, but still improved the rate at which network problems occur. Again this was also tested empirically.

We suspect that the fact that we had been using an Android emulator instead of an actual physical device might have also influenced the network performance.

Last but not least, future testing could be done in an objective manner by checking the quality of the network connection between the client and the server by means of measuring the ping, jitter and, most importantly, the packet loss.

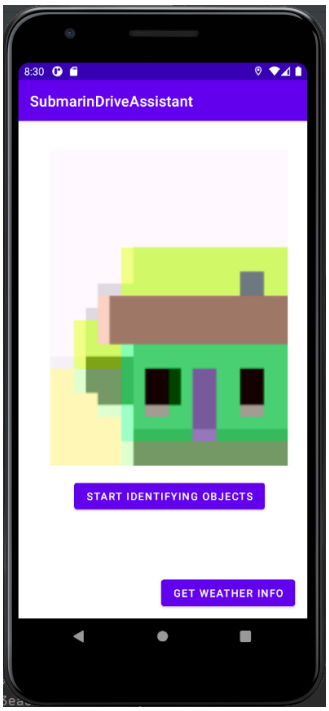

Figure 5.9: Old UI

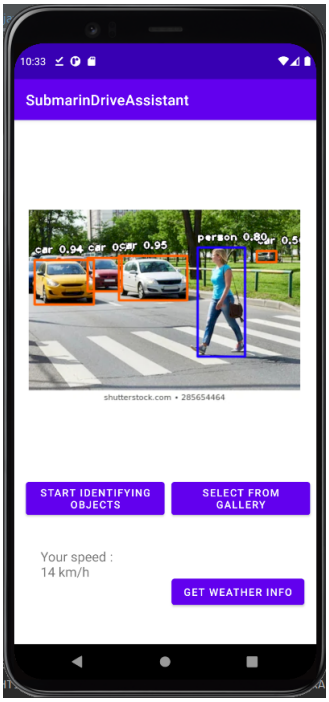

Figure 5.10: New UI

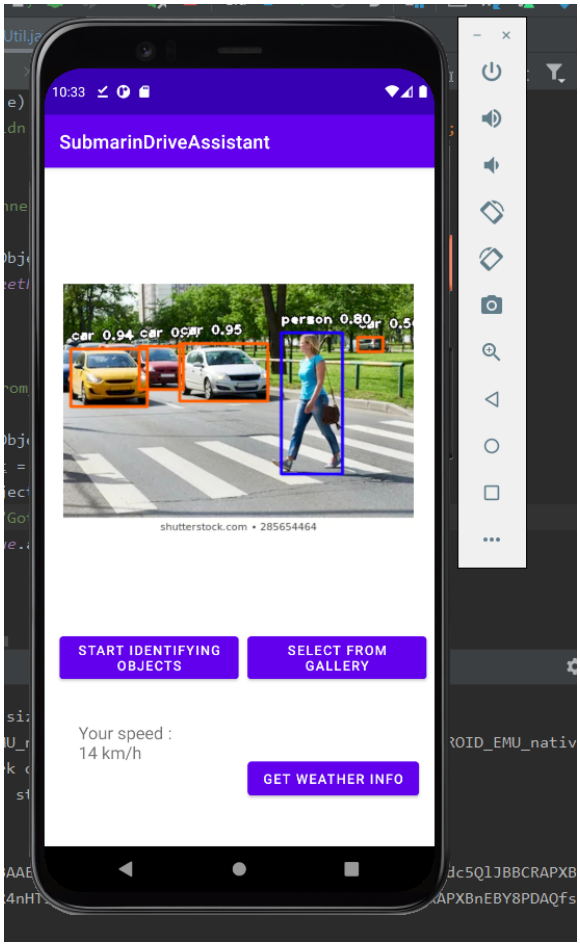

Figure 5.11: Screenshot 1

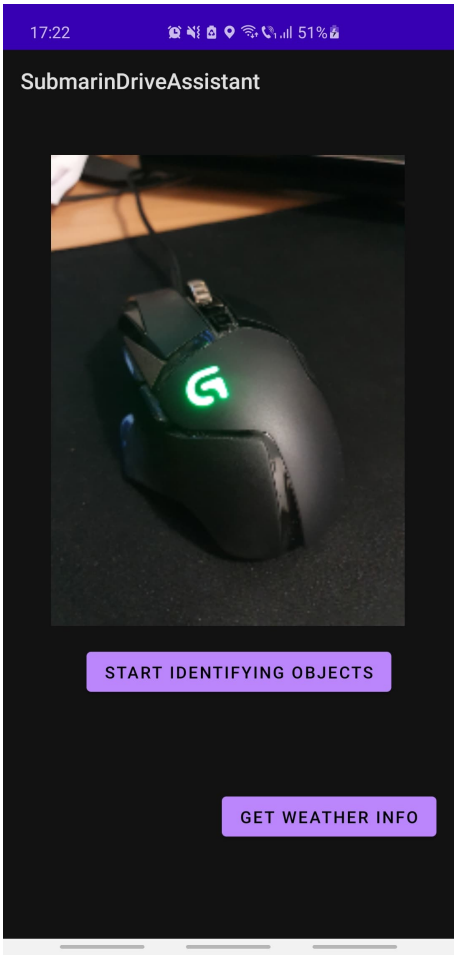

Figure 5.12: Screenshot 2

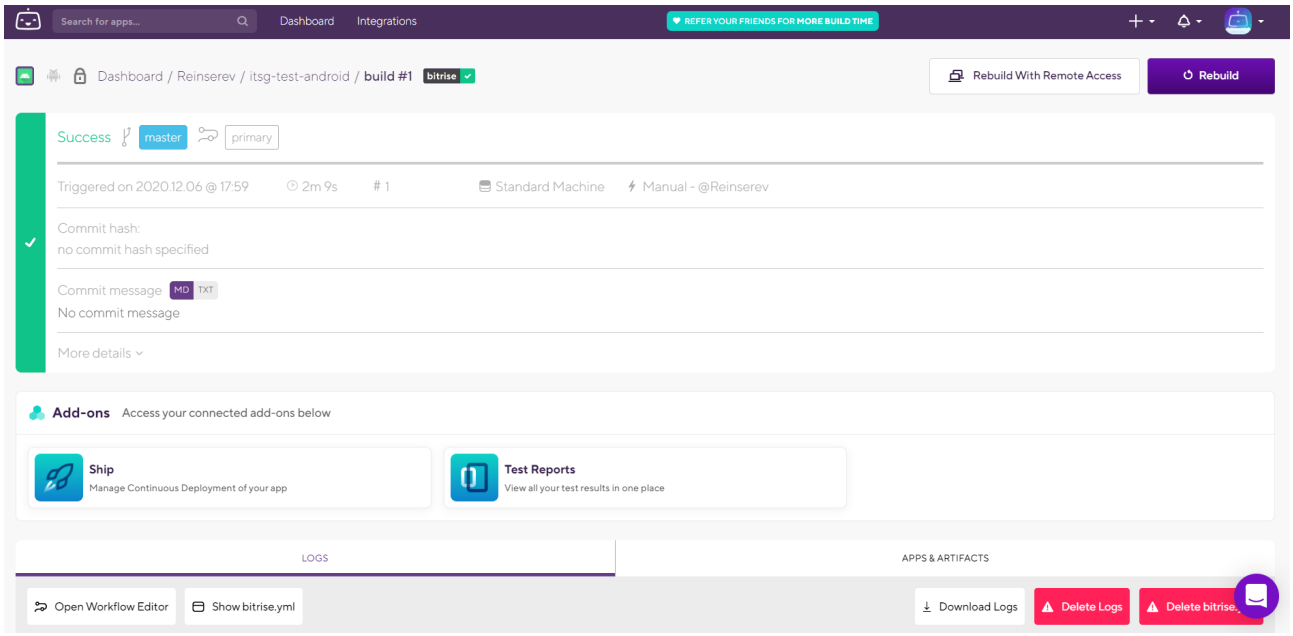

Figure 5.13: BitRise

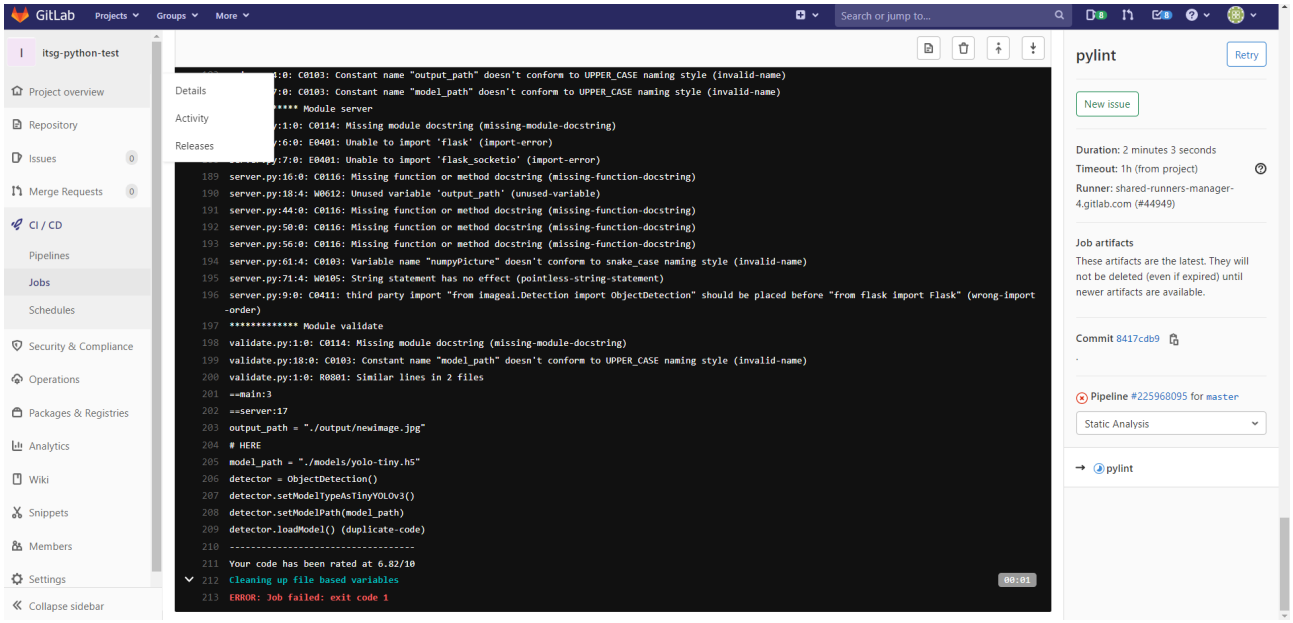

Figure 5.14: GitLab

Chapter 6

Philosophical aspects

6.1 Social impact

The social impact of driving assistants is expected to take off in the next couple of years, with an ever increasing tendency to fully automate this process and give drivers a seamless experience when behind the steering wheel. The ultimate goal is to automate most of these cumbersome tasks that drivers are faced with everyday and decrease the effort that needs to be put into the action of driving almost to a steady zero.

With this in mind, the introduction of AI into the autonomous driving field seems particularly useful, as it enables for many applications and is very suitable for this task altogether. We believe two of the most important factors that need to be discussed in regard to the social impact are:

- Improved safety factor and driver's comfort
- Minimised chances of traffic collisions

As of today, we can say the task of driving is still highly reliant on the human factor and depends largely on variable factors, like distributive attention of the driver and agility in traffic, as well as ability to stay focused. It is easy to see why these could turn driving into a rather risky and dangerous task, that could even result in loss of human lives, thus the need for a way to automate the various aspects of the driving experience arises, making an AI assistant the perfect candidate.

6.2 Ethics and solution objectivity

Ethics surrounding driving assistants and automotive systems that help the driver with the decision-making process all eventually gravitate back to whether these decisions can be made in a sensible

manner, given some unexpected extreme situations or not.

Given, for example, a situation in which the assistant detects the imminent collision with a group of pedestrians, but their lives can be saved if the passengers' one is put to risk, what decision is made in this situation? This is where the ethical aspects come into place and questions start to arise.

However, specialists say, these kind of situations are rather extreme and very unlikely to happen in a real life setting. Ultimately, it must be kept in mind that the design of such driving-aid tools needs to balance the safety of others â pedestrians or cyclists â with the interests of cars' passengers.

It also needs to be noted that the accuracy of such automated systems is never going to be 100% correct. That means, we cannot base our entire confidence in the correctness of a driving assistant and we should always apply our own judgement as well.

We believe our solution to be quite objective from a perspective of algorithm fairness, there is little room for subjectivity in what classification problems are concerned. The only part that could involve subjectivity to a certain degree is the decision making step or the predictions, which could be based off on a limited set of drivers as starting point for the learning process of the algorithm.

6.3 Diversity and Inclusion

In this section, we wish to discuss the different aspects related to race, ethnicity and other such differentiations that could yield different results, based on the range of people the algorithm is applied onto.

To test the performance of our algorithm on these different groups of people, we have run a number of tests on people from different races as well as people dressed in religious clothing, and compared these results with the ones obtained on white people.

6.3.1 Discussion

To start, we'd like to restate the main purpose(s) of the AI component, which is to detect and label pedestrians and cars (other vehicles are a 'nice to have', and are not mandatory).

For the task of pedestrian detection, the application needs to label the whole pedestrian in the form of a bounding box which surrounds the whole person. This is very different from the task of facial recognition, where only the faces of people in the picture need to be detected and highlighted (and potentially identified). From this point of view, pedestrian detection might be easier to do in a diverse and inclusive manner, since there are fewer potential hurdles in the way when compared to facial recognition. People have the same basic body shape, no matter their race. That some races are

Figure 6.1: A street in Africa (1)

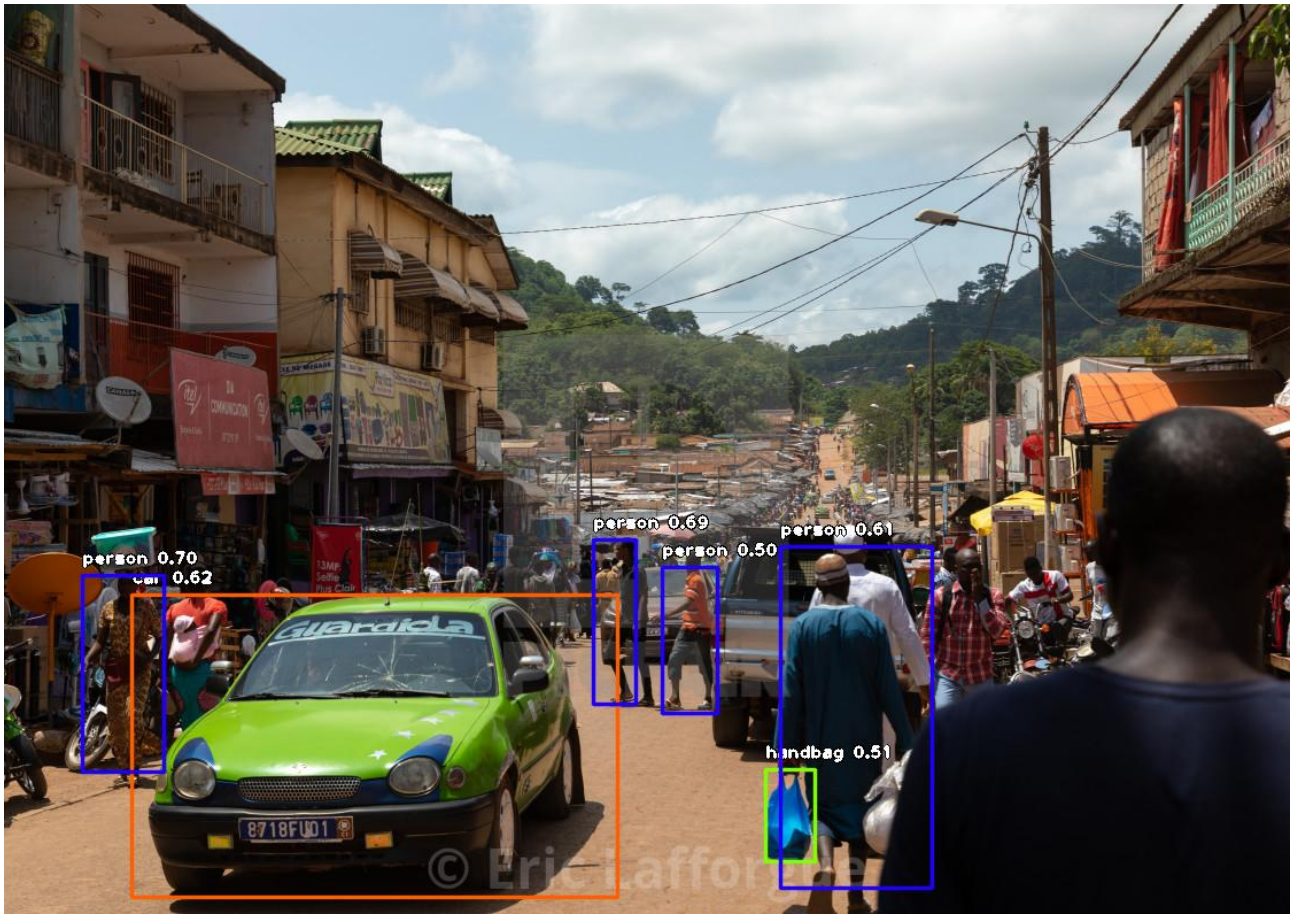

Figure 6.2: A street in Africa (2)

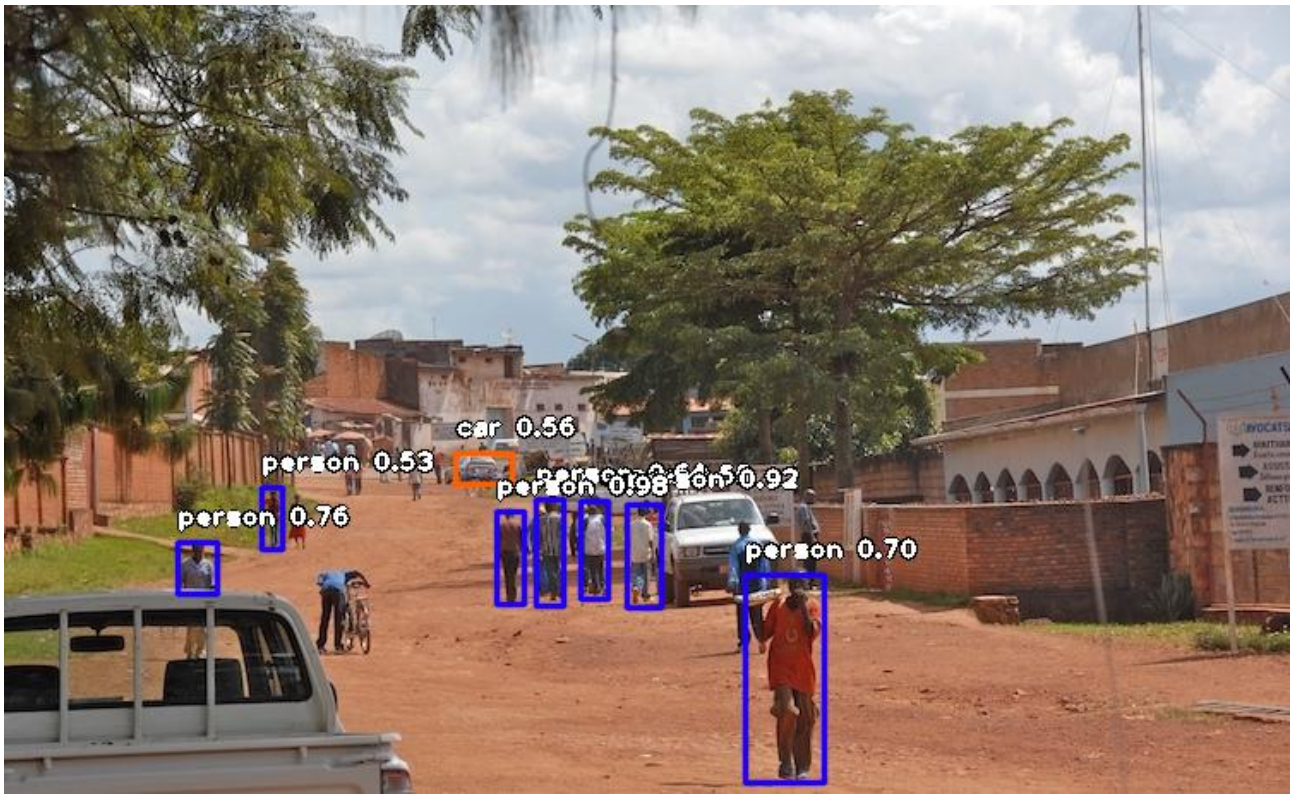

Figure 6.3: A street in Africa (3)

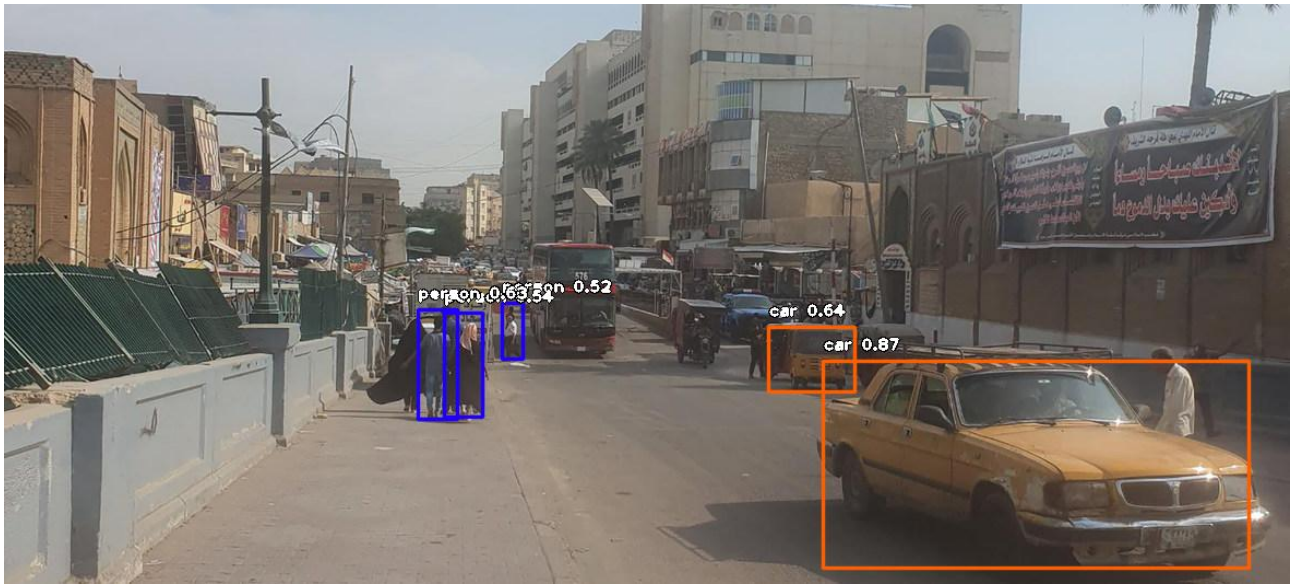

Figure 6.4: A street in the Arab world (1)

Figure 6.5: A street in the Arab world (2)

shorter/taller than others and/or their constitution may (slightly) vary, are just details.

Facial features, on the other hand, tend to vary from one race to another, which means that a facial recognition algorithm needs additional input data to train on as to ensure that it works for people of all races and, more importantly, to avoid any kind of racial bias, especially if its purpose is to identify people. This is especially important in the context of potential use of such systems by law enforcement organizations, a subject which has drawn a significant amount of public scrutiny and criticism and has become (very) controversial.

6.3.1.1 Africa

We chose to analyze a few pictures from some streets in Africa in order to test whether the used AI method (YOLO) works for people of color (African people, in this case) or not.

In the first picture, Fig. 6.1, the method is almost a complete failure. Only one person out of the four in plain sight are identified. Interestingly enough, the motorcycle is identified, but not its rider. Not even the woman carrying a basket above her head (to the right of the motorcycle), which is arguably the easiest to spot in the whole picture, is identified.

Whether to blame this failure on racial bias or on the method itself, we do not know. On one hand, it is clear that there are at least two (it can be argued that there are three!) people in plain sight which were missed. On the other hand, this wouldn't be the first time that the YOLO implementation we're using has missed people 'hidden' in plain sight. Until further research, we may assume that there is no racial biased involved, and the model simply lacked the appropriate training data to properly identify African people. Thus, it can be stated that the training dataset could use more diversity.

The situation is remedied in the second (Fig. 6.2) and third (Fig. 6.3) pictures though. In the second picture, just one person in plain sight is missed, while 4 others are successfully identified (note that the person standing right in front of the camera is not taken into consideration since he is out of focus).

Lastly, there are 7 detentions in the last picture, which is a good result. Even one of the more distant persons has been correctly identified. However, one person hidden in plain sight was missed.

To conclude, we can only speculate that detecting people located at a reasonable distance (that is, not too close to the camera, but not too far away) works best, regardless of the person's race.

6.3.1.2 The Arab World

We chose some pictures from the Arab world to test whether or not our solution labels people wearing religious clothing (hijab), especially people wearing burqas, are detected as pedestrians or not.

As we may see in all pictures, but especially in the second (Fig. 6.5) and the third (Fig. 6.6), women wearing burqas are not identified at all. This is not a singular occurrence, as people wearing them are consistently not identified in all 3 pictures (only in the last one, a single person is detected, leaving a total of 6 burqa-wearing women undetected). Thus, we may say that more training data from the Arab world would most likely help with this issue, in order to ensure that people wearing this kind of religious garment are detected properly.

6.3.1.3 China

We chose some Chinese streets to test whether Asian people are detected correctly or not. The results are good; we were unable to find any issue on our sample of 3 pictures.

6.3.1.4 Germany

We chose some German streets to evaluate the algorithm on white people. As expected, everything went flawlessly.

6.3.1.5 Mexico

Finally, we chose Mexico as a way of evaluating the used method on some different people of colour, specifically, Mestizo people. As with Asian people, the results are good.

6.3.2 Conclusion

As we observe from the experiments undertaken, there may be a difference in accuracy when using Tiny YOLO V3 to detect people of different races or people wearing religious clothing. While we cannot wholeheartedly say that our approach is race-inclusive and makes no differences based on the ethnicity of the subjects (our tests on African people were inconclusive), we can certainly state that people wearing burqas have a significant chance of not being detected.

Thus, it is advisable for the training data to be large enough to contain a healthy amount of samples from people of different races (especially African people) and cultural backgrounds (especially Muslim people, whose female believers wear different types of religious garments). However, the latter might prove challenging, especially in the context of highly-developed secular states (like those in Western Europe) outlawing face-masking garments (including burqas!) in public. Thus, the data would have to be taken directly from a country where such practices are still legal.

Figure 6.6: A street in the Arab world (3)

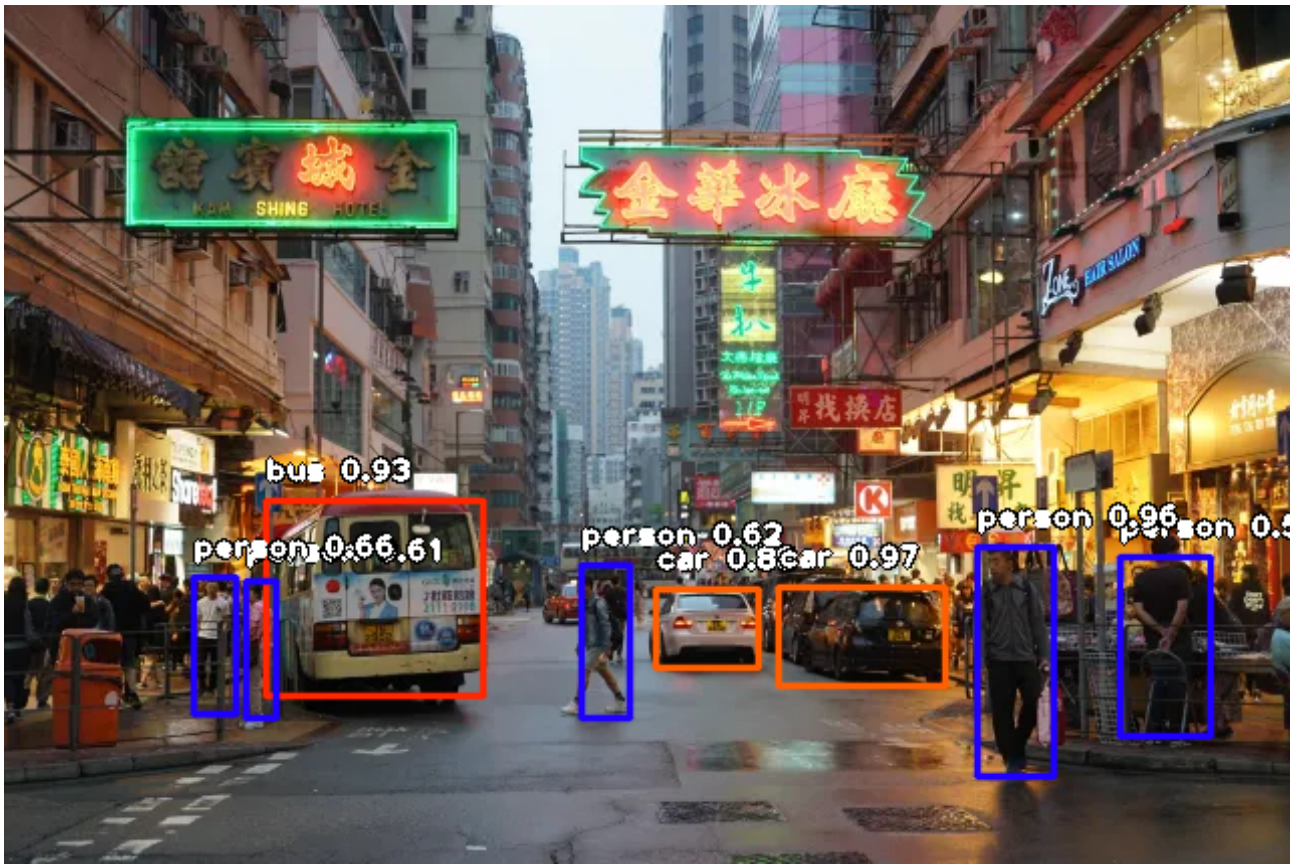

Figure 6.7: A street in China (1)

Figure 6.8: A street in China (2)

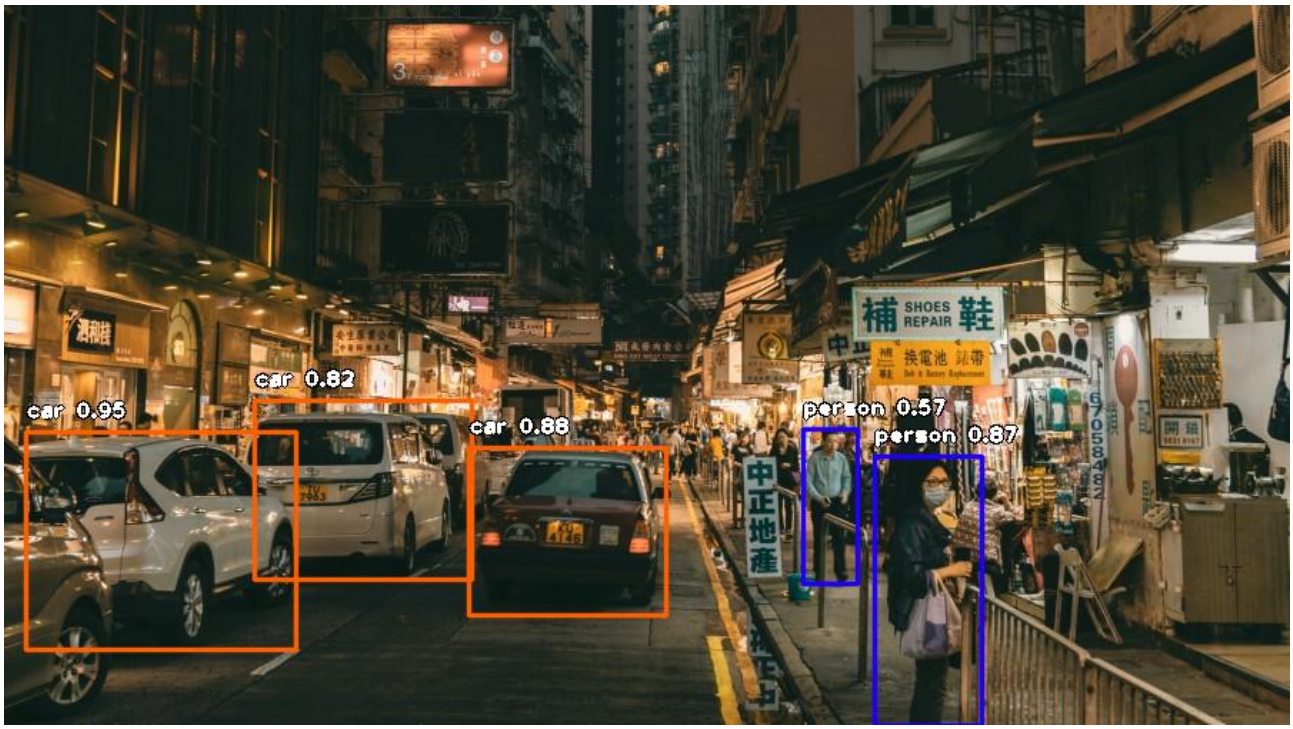

Figure 6.9: A street in China (3)

Figure 6.10: A street in Germany (1)

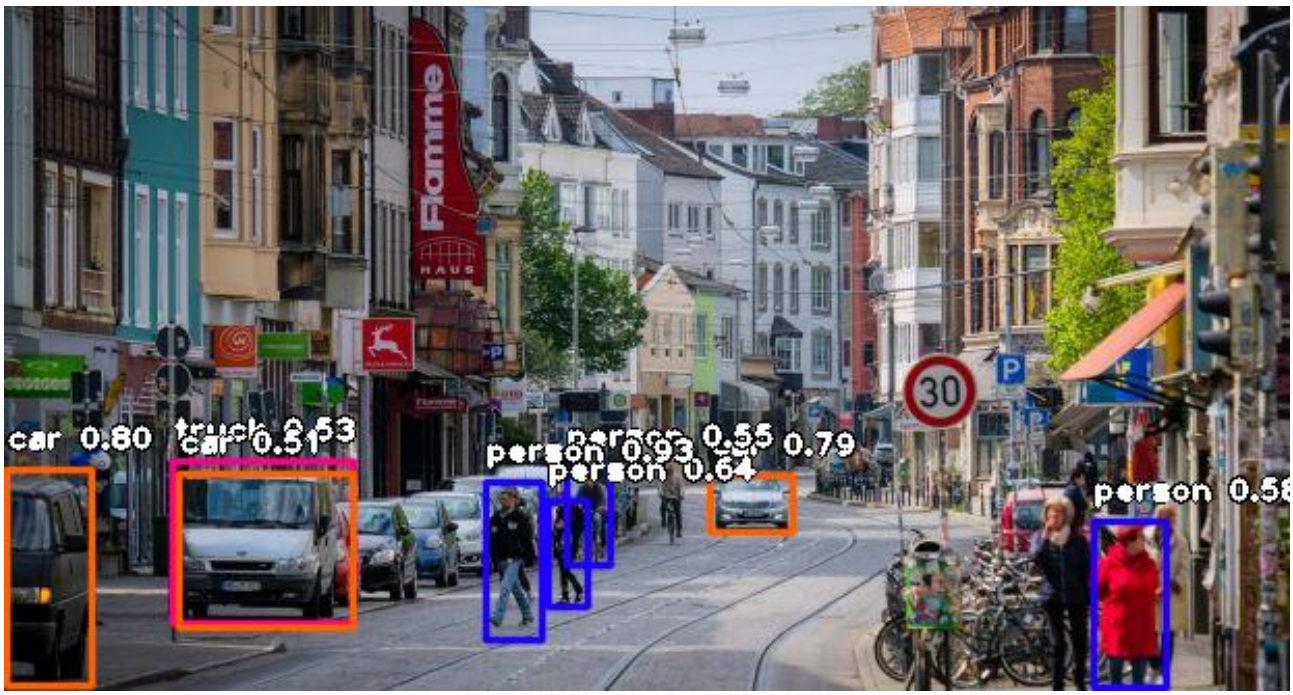

Figure 6.11: A street in Germany (2)

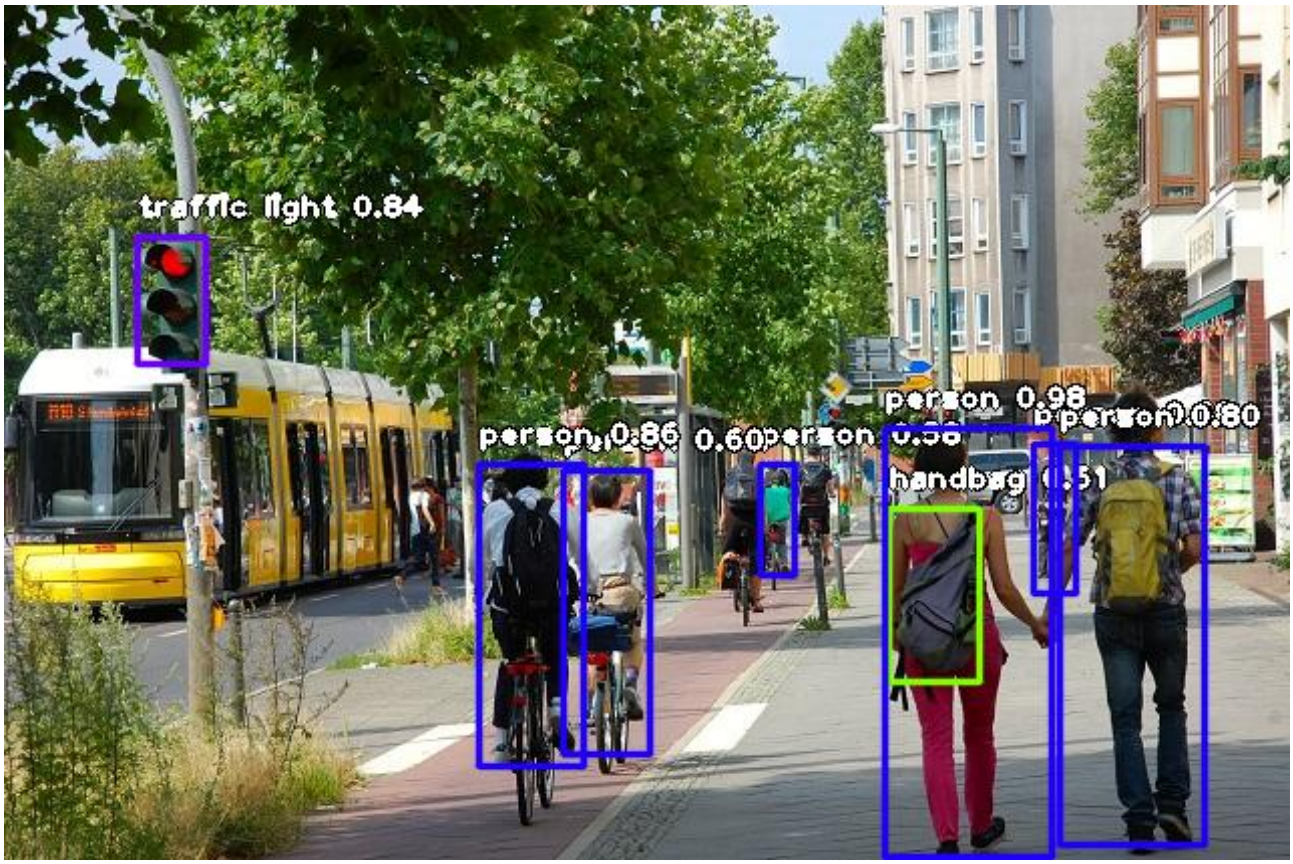

Figure 6.12: A street in Germany (3)

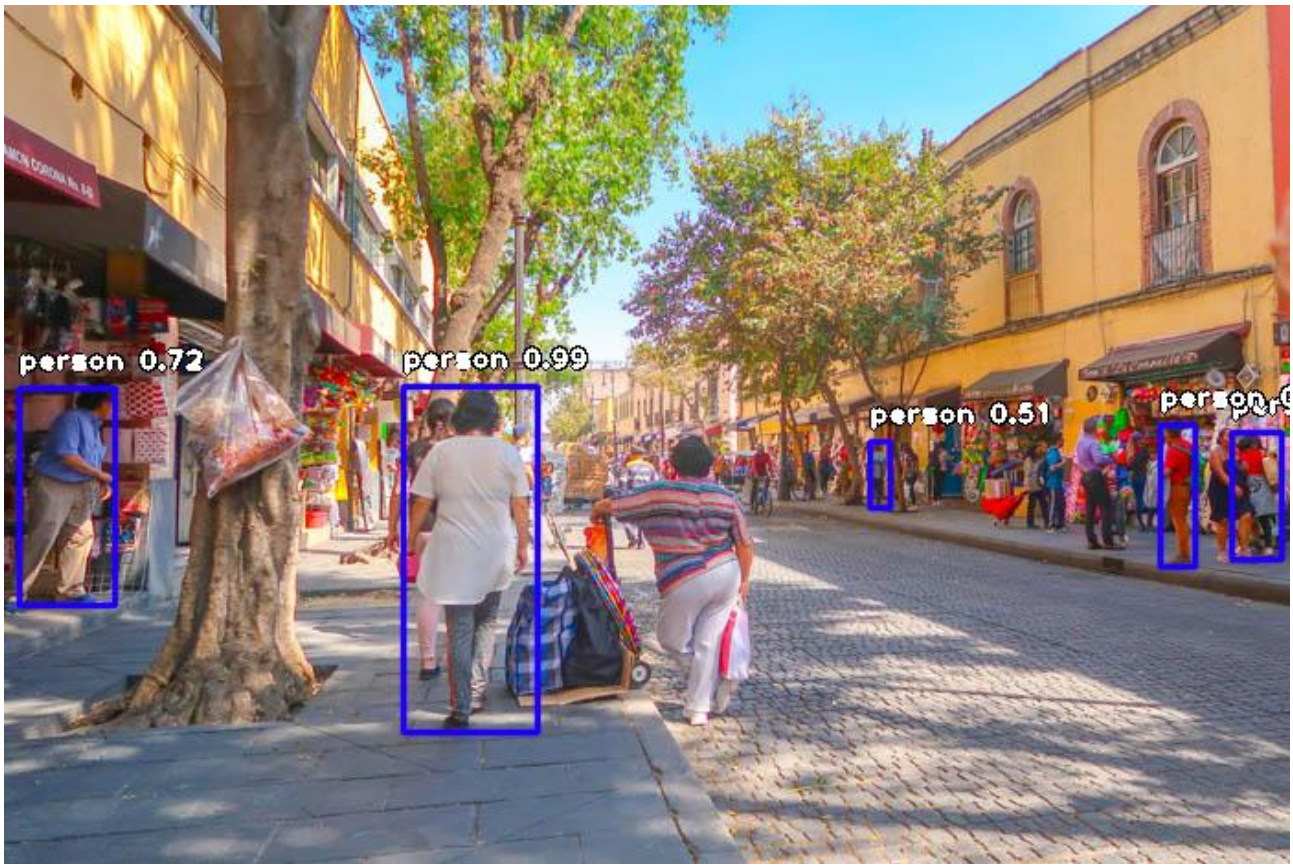

Figure 6.13: A street in Mexico (1)

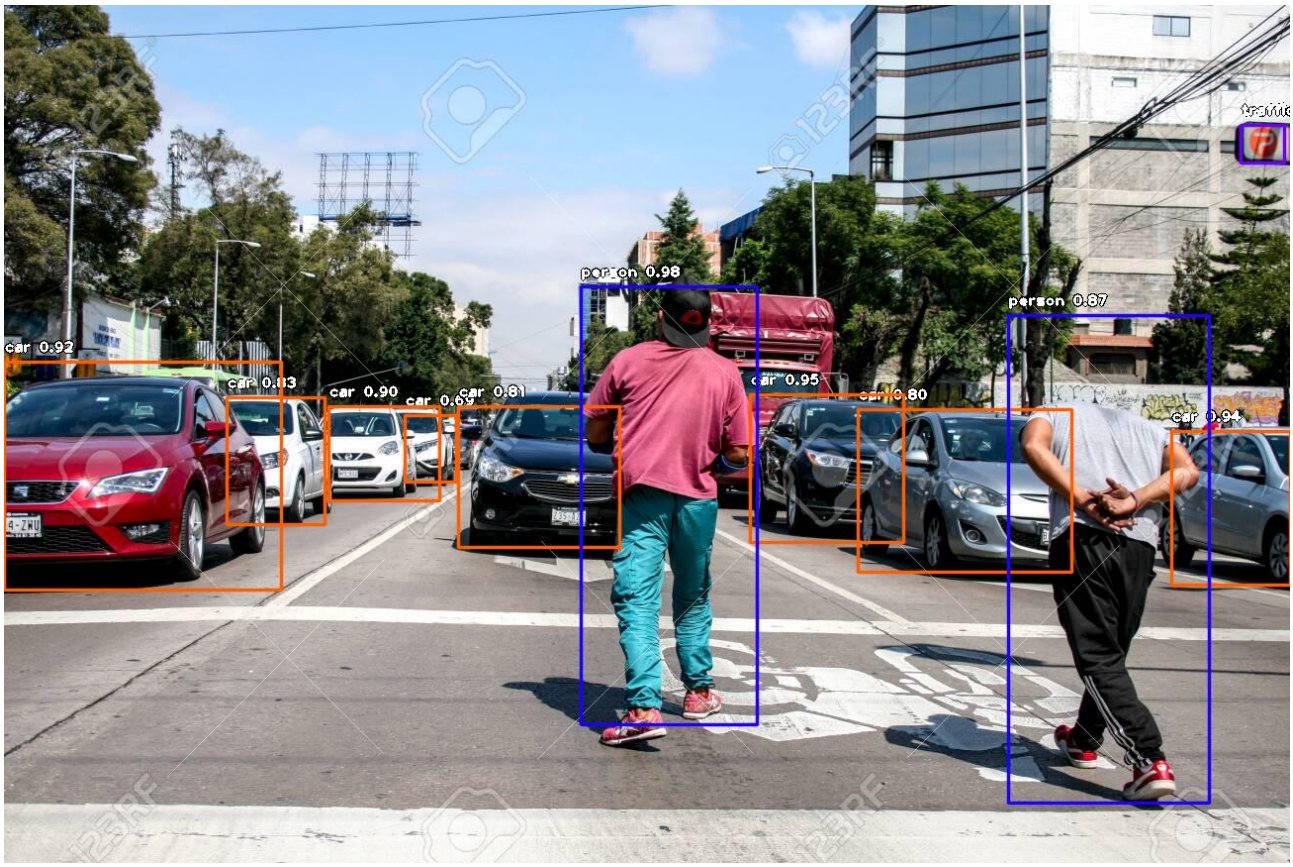

Figure 6.14: A street in Mexico (2)

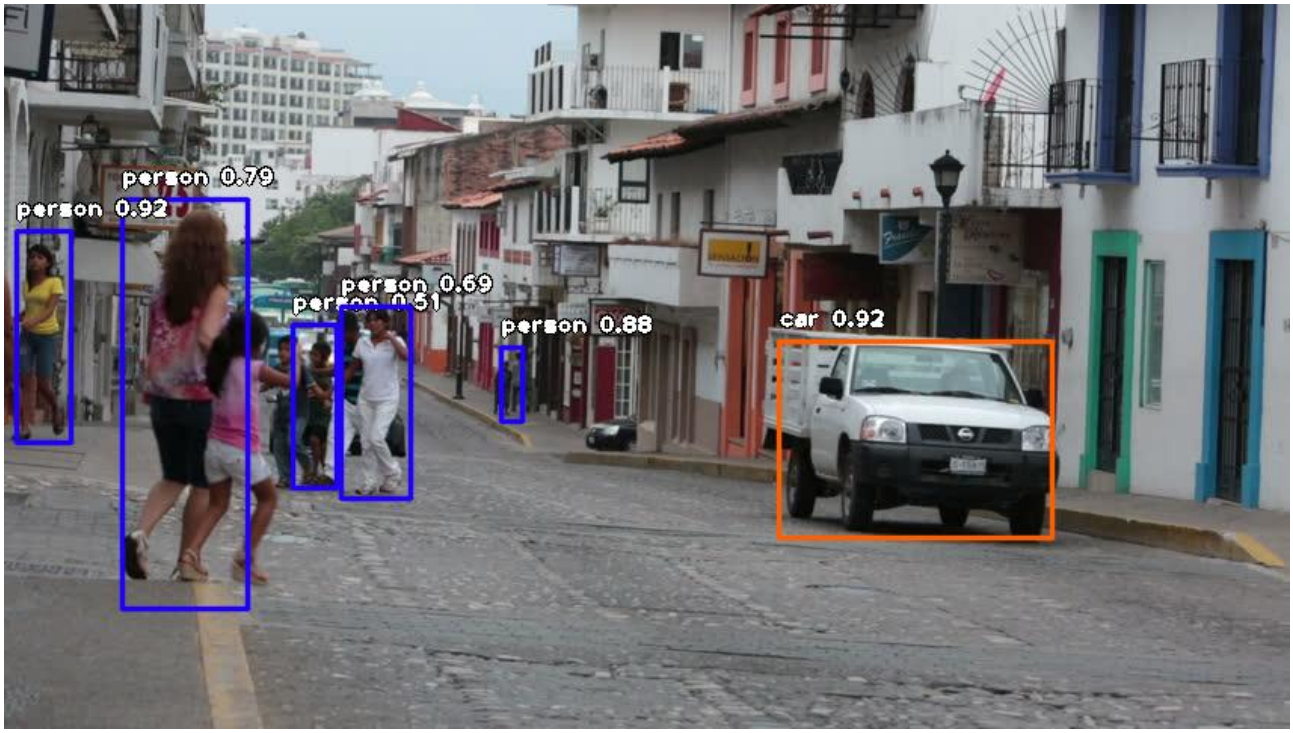

Figure 6.15: A street in Mexico (3)

Chapter 7

Conclusion and future work

To summarize our work, we believe we have demonstrated an accurate representation of the Object detection method in autonomous driving field, yielding quite satisfactory results and raising some interesting points of discussion for future enhancements. Although we did not reach state of the art, we are confident our research is still solid enough to be taken into consideration, especially by the means of the original contributions we bring.

The main point that was observed throughout our experiments is that the Tiny model of YOLOv3 falls short of the full version of YOLOv3 in terms of performance, but the tiny is considerably faster, which we believe to be a great asset for delivering real time results. We notice on darker images we obtain poorer results and it's hard to tell apart individual persons in a larger group of people.

One future direction we think we could take is to move 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 could further bring us closer to a real-time experience.

We have also shown that minor improvements can be made if we increase the brightness of the images used, producing slightly better results in qualitative terms.

One other point that we consider will be important towards future research in the area is 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. This is the first step towards a more diverse approach in the field of pedestrian detection and we believe it would be interesting to be analyzed further.

Appendices

Appendix A

Individual Contributions

List of authors (in alphabetical order) and their contributions:

- Katona Ildiko-Noemi : Android Developer (frontend) + backend to frontend interface + presentation script
- Lung Andreea Cristina : all of the report + slides for presentation
- Nagy Barnabás : Lead Android Developer (frontend) + demo for presentation
- Popa Cătălin : AI backend, benchmarking the chosen method (YOLO) using two well-established datasets (INRIA and Caltech), Diversity and Inclusion sub-chapter + presentation

Bibliography

[1] Caltech pedestrian detection benchmark. http://www.vision.caltech.edu/Image_Datasets/CaltechPedestrians/#:~:text=The%20Caltech%20Pedestrian%20Dataset%20consists,2300%20unique%20pedestrians%20were%20annotated. (Accessed on 01/04/2021).

[2] Dive really deep into yolo v3: A beginnerâs guide | by ethan yanjia li | towards data science. https://towardsdatascience.com/dive-really-deep-into-yolo-v3-a-beginners-guide-9e3d2666280e. (Accessed on 29/12/2020).

[3] How to find the android version distribution statistics in android studio. https://www.xda-developers.com/android-version-distribution-statistics-android-studio/. (Accessed on 01/04/2021).

[4] Inria person dataset. http://pascal.inrialpes.fr/data/human/. (Accessed on 01/04/2021).

[5] The network structure of tiny-yolo-v3. | download scientific diagram. https://www.researchgate.net/figure/The-network-structure-of-Tiny-YOLO-V3_fig1_338162578. (Accessed on 01/04/2021).

[6] Official english documentation for imageai! â imageai 2.1.5 documentation. https://imageai.readthedocs.io/en/latest/. (Accessed on 01/04/2021).

[7] Transfer learning from pre-trained models | by pedro marcelino | towards data science. https://towardsdatascience.com/transfer-learning-from-pre-trained-models-f2393f124751. (Accessed on 29/12/2020).

[8] Guest Contributor. Object detection with imageai in python. link.

[9] Young-Hoon Nho. Driving situation-based real-time interaction with intelligent driving assistance agent. 08 2015.

[10] Yatham Sai Sangram Reddy, Devareddy Karthik, Nikunj Rana, M Jasmine Pemeena Priyadarsini, G K Rajini, and Shaik Naseera. Traffic signs recognition for driving assistance. *IOP Conference Series: Materials Science and Engineering*, 263:052046, nov 2017.

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

[12] Adrian Rosebrock. Pedestrian detection opencv. link, 2015.

[13] OpenCV team. About opencv. link, 2020.