

BUDAPESTI MŰSZAKI ÉS GAZDASÁGTUDOMÁNYI EGYETEM

Villamosmérnöki és Informatikai Kar

Computer Vision Systems Project Summary

Kürti Ádám HZUCDC

Pucsok Sándor PP22N9

2023

**Content**

[**1.** **Project Description** 3](#_Toc137245902)

[**2.** **Vehicle detection algorithm** 4](#_Toc137245903)

[**2.1** **Object detection** 4](#_Toc137245904)

[**2.2** **Our vehicle detection model** 5](#_Toc137245905)

[**3.** **License plate detection and reading with OCR** 7](#_Toc137245906)

[**4.** **Estimating vehicle velocity** 10](#_Toc137245907)

[**5.** **Testing** 12](#_Toc137245908)

[**6.** **Summary** 14](#_Toc137245909)

[**Literature** 15](#_Toc137245910)

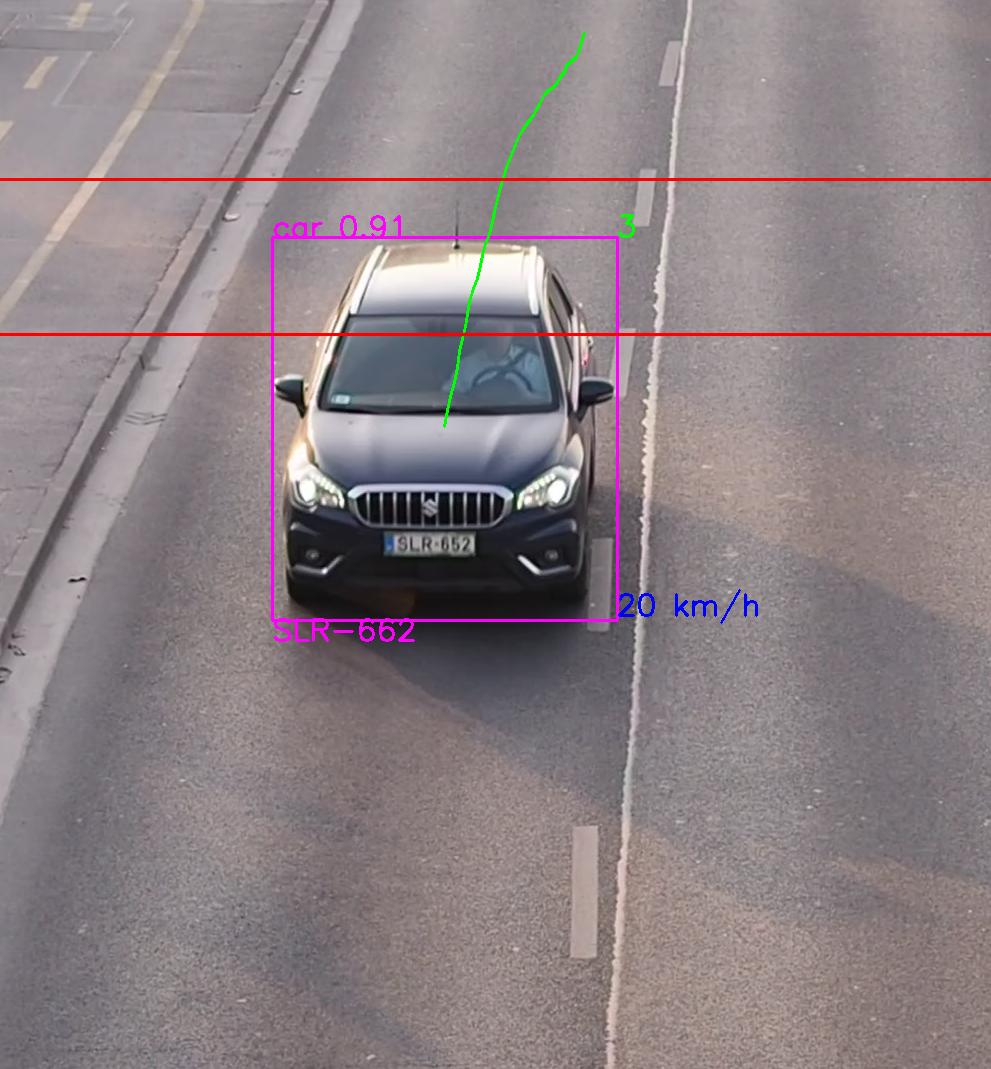
1. **Project Description**

**The aim of this project is to develop an algorithm that can efficiently recognize and detect cars, their license plates and read it and approximate the velocity of the vehicles in video recordings. These will be combined to form a speed camera.**

**Our main idea was to use the following techniques to achieve the speed camera functionality:**

1. **Vehicle detection and cropping**
2. **Vehicle indexing and tracking**
3. **License plate detection and cropping**
4. **License plate reading with optical character recognition**
5. **Vehicle velocity estimation using optical flow**

**In the following will we go through these techniques and present our solutions.**



1. Figure: Our speed camera algorithm in action

1. **Vehicle detection algorithm**
   1. **Object detection**

In this chapter we introduce the algorithm what we use for vehicle detection, which is basically an object detection problem.

For object detection there are several widely used neural network architectures each with different speed and accuracy. The R-CNN (Region-based Convolutional Neural Network) family and the YOLO (You Only Look Once) family are two popular approaches for object detection in computer vision.

The R-CNN family consists of several variants, including R-CNN, Fast R-CNN, and Faster R-CNN. These models use a two-stage approach. First, they generate region proposals using selective search or similar algorithms. Then, each proposed region is classified and refined using a CNN. The R-CNN family is slower compared to YOLO models. It involves multiple passes through the CNN for each region proposal, which can be computationally expensive. The R-CNN family tends to provide higher accuracy in object detection tasks, especially for smaller objects. The two-stage approach allows for more precise localization and classification. They perform well in scenarios with a wide range of object sizes and densities. The two-stage approach helps in handling diverse objects effectively.

The YOLO family on the other hand, follows a one-stage architecture. YOLO divides the input image into a grid and predicts bounding boxes and class probabilities directly using a single pass through the network. YOLO models are known for their real-time object detection capabilities. They process the entire image in a single forward pass, resulting in faster inference times. YOLO models sacrifice some accuracy for speed. They prioritize detecting larger objects and can struggle with small objects or objects close together. However, newer versions of YOLO, such as YOLOv4, have improved accuracy by incorporating various techniques. YOLO models excel in detecting larger objects and objects with moderate density. They can struggle with small objects and densely packed scenes.

In summary, if accuracy is the primary concern and real-time performance is not critical, the R-CNN family is a good choice. On the other hand, if real-time performance is crucial, and some loss in accuracy is acceptable, the YOLO family is a more suitable option, that’s why we chose the YOLO family.

YOLO algorithm aims to predict a class of an object and the bounding box that defines the object location on the input image. It recognizes each bounding box using four numbers:

* Center of the bounding box ((bx, by)
* Width of the box (bw)
* Height of the box (bh)

In addition to that, YOLO predicts the corresponding number c for the predicted class as well as the probability of the prediction (Pc). YOLO divides the image first into a grid. With the existence of a grid, it’s possible to detect one object per grid cell instead of one object per image. It generates the predictions in the following format:

where:

* (Pc) is the probability of the object class
* Bx and By are coordinates of the center of the bounding box, relative to the cell
* Bw and Bh are width and height of the bounding box relative to the whole image
* Ci are the confidence values of each class where i is the number of classes.

For training a YOLO network the following annotation format is used:

* One row per object
* Each row is class x\_center y\_center width height format.
* Box coordinates must be in normalized xywh format (from 0 - 1). If the boxes are in pixels, division of x\_center and width by image width, and y\_center and height by image height is needed
* Class numbers are zero-indexed (start from 0).

The training files must be organized in the following way:

* ../datasets/coco128/images/im0.jpg # image
* ../datasets/coco128/labels/im0.txt # label

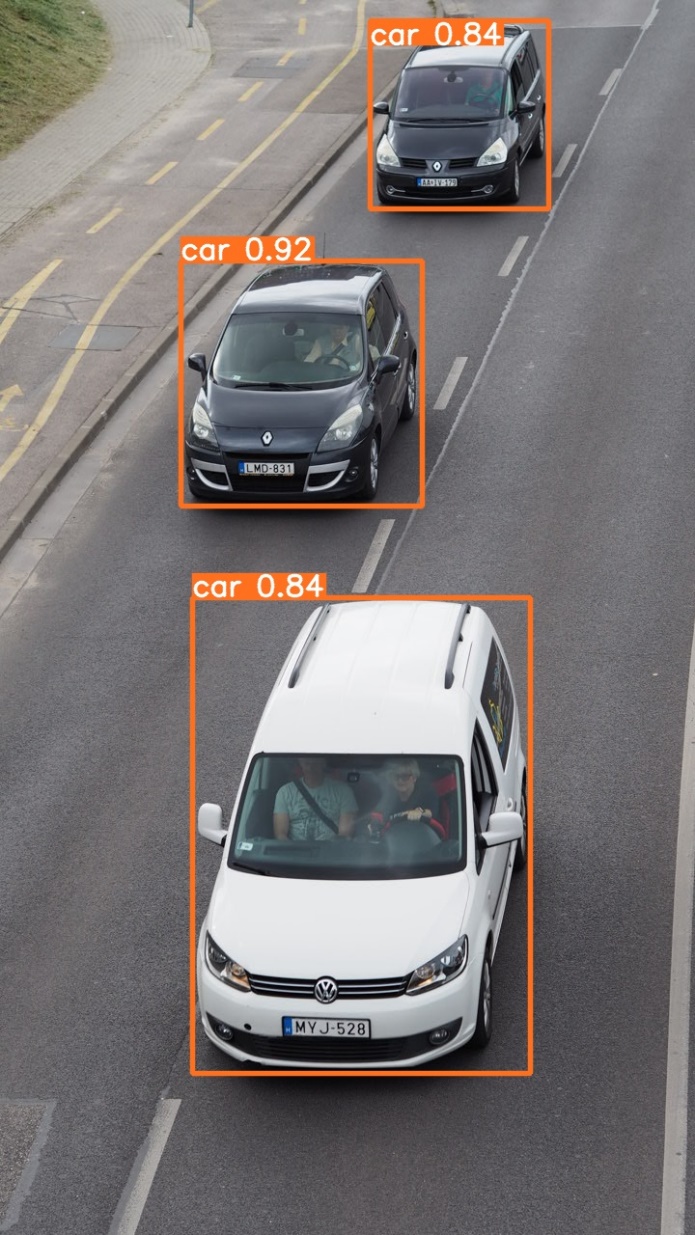
Also a dataset.yaml file needs to be created that contains the dataset root directory and the relative paths to the train/val/test datasets, and also the indexing of the class names. [3]

* 1. **Our vehicle detection model**

As mentioned in the previous chapter, the detection was performed using the YOLO algorithm, which was able to achieve adequate accuracy to use it in our case.

First, we tried to train our own YOLO model for detecting various vehicle types. For this we used the [MIO-TCD](https://tcd.miovision.com/challenge/dataset.html) localization dataset, containing 137,743 images at different times of the day and different periods of the year by thousands of traffic cameras deployed all over Canada and the United States. It has 11 different labels for different vehicle types. The training code can be seen [here](https://github.com/Kurtiadam/speedcam_cvs/blob/main/YOLO_vehicle_detection/code/YOLO_train.ipynb). The format wasn’t ready for YOLO training, so we had to convert it for which we wrote a [script](https://github.com/Kurtiadam/speedcam_cvs/blob/main/YOLO_vehicle_detection/code/yolo_format_converter_vd.py). During training over 50 epochs, we achieved a 0.762 mAP50 and 0.564 mAP50-95 score. However, after trying the trained network with our own made and real images from vehicles, the model performed disappointingly. The reason for this probably was, that the trained images were low resolution and contained even lower resolution vehicles with very little detail, but our images very clear, high-resolution images with a lot of details.

Knowing this, we went for an easier approach. We used the pre-trained YOLOv8 model that can detect different types of vehicles out of the box, e.g., cars, trucks, busses, or motorcycles. From the pre-trained weights, we tested the nano, small and medium versions and we concluded that the small version achieves adequately accurate and quick results for our vehicle detection algorithm.



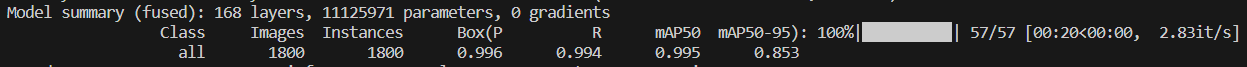
2. Figure: Vehicle detection with the YOLOv8s model

Most of the vehicle detections were above 80% confidence and the inference time was around 30 ms, not slowing our video pipeline down too much. As the detection worked well after several tests, we could move on to the next task, which was to detect and read the number plates.

1. **License plate detection and reading with OCR**
   1. **License plate detection**

In this chapter we introduce the algorithm what we use for license plate detection and reading.

As for the license plate detection, we again used the YOLOv8 algorithm. Since the YOLO can’t recognize license plates out of the box, we had to look for a dataset to train it on. We have found a good quality dataset called [RodoSol-ALPR Dataset](https://github.com/raysonlaroca/rodosol-alpr-dataset). It contains 20,000 images captured by static cameras located at pay tolls owned by the Rodovia do Sol (RodoSol) concessionaire, which operates 67.5 kilometers of a highway (ES-060) in the Brazilian state of Espírito Santo. There are images of different types of vehicles (e.g., cars, motorcycles, buses, and trucks), captured during the day and night, from distinct lanes, on clear and rainy days, and the distance from the vehicle to the camera varies slightly. All images have a resolution of 1,280 × 720 pixels. We had to ask for permission to use the dataset for our academical use case. The format again wasn’t ready to be used for YOLO training, so we created another [script](https://github.com/Kurtiadam/speedcam_cvs/blob/main/YOLO_license_plate_localization/code/yolo_format_converter_lp.py) to convert it to an appropriate format. The training was done locally on our PC, with an RTX 3070 GPU. With the pretrained small weight size and with only 5 epochs a quite good results could be achieved with only around 10 ms inference time:



3. Figure: The results of the license plate detection training



4. Figure: Our license plate detection in action

* 1. **Reading of license plates**

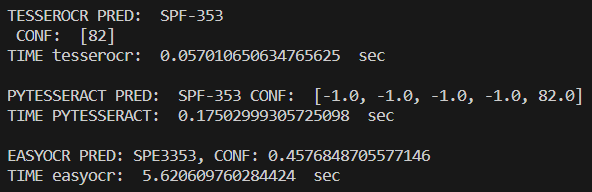
To read the license plates firstly we used the *pytesseract* OCR engine. This solution was good in terms of results but was very slow, it only could do inference in a few hundred milliseconds. The next one we tried was EasyOCR, but this solution was worse than the previous one in every aspect, so we discarded it quickly. After the above-mentioned solutions didn’t offer adequate results, we found the API version of pytesseract: *tesserocr*. This did not slow down the program that much and worked with seemingly the same accuracy.

Tesserocr is a Python wrapper around the Tesseract C++ API. Whereas Pytesseract is a wrapper for the tesseract-ocr CLI. Therefore, with Tesserocr we can load the model at the beginning of our program and run the model separately (for example in loops to process videos). With pytesseract, each time we call image\_to\_string function, it loads the model and processes the image, therefore being slower for video processing [2].

The comparison for the different OCR techniques can be seen at 3. Figure for the input seen at 5. Figure.



5. Figure: An example license plate



6. Figure: The comparison of different OCR techniques.

We can see that the EasyOCR library didn’t achieve as good as a result in terms of both accuracy and speed as the other two. TesserOCR was just as accurate as Pytesseract but achieved a three-fold speed increase over its rival, so we chose this technique. Sadly, the two best performing techniques don’t always report a confidence score for the prediction of each character as we can see on the 6. Figure with the -1.0 values. This meant, that we couldn’t reliably use this value for updating the prediction when a higher confidence score was found, since it could be that the correct license plate text was found but with -1.0 confidence values.

We also tried different preprocessing techniques to help the OCR recognize the text on the license plates. We tried Gaussian and bilinear filtering to remove noise, Blackhat conversion into gray image, thresholding, eroding/dilating the binary images and angle correcting the slanted images. The smoothing filtering and thresholding into binary images seldom produced better results, so we discarded them. The Blackhat grayscale conversion produced a little bit better result with angle correction, so we kept those solutions. At the 7. Figure the results of the different techniques can be seen from top to bottom: the original image, Blackhat grayscale conversion, thresholding and opening.

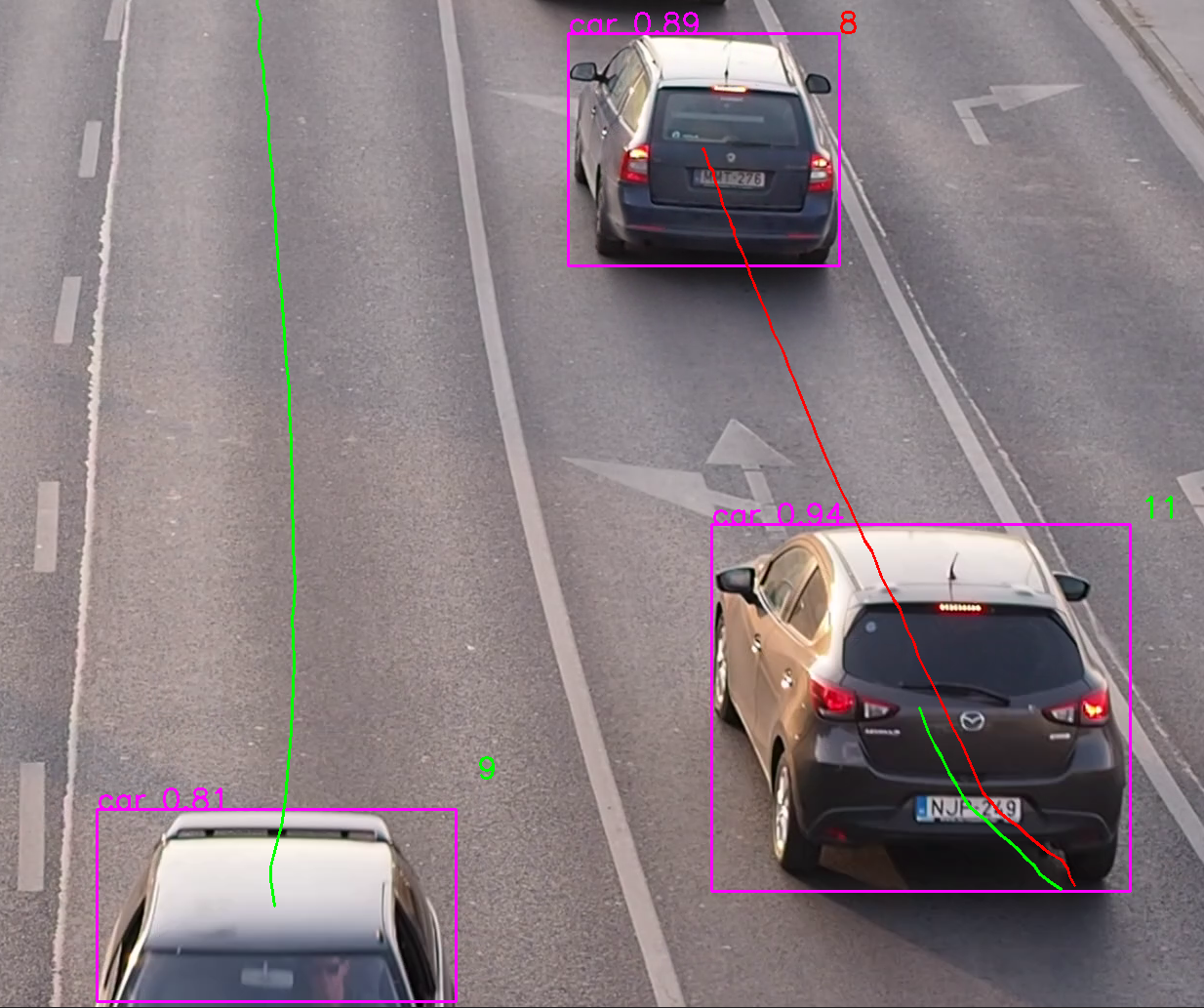
A képen szöveg, Betűtípus, képernyőkép, szám látható

Automatikusan generált leírás

7. Figure: The preprocessing techniques results for the license plates

1. **Estimating vehicle velocity**

For estimating a vehicles velocity, we also need to track it across the video. Otherwise only independent detections would be accessible for us for each frame. For the tracking we used the [SORT](https://github.com/abewley/sort) tracking algorithm. It is a simple online and real-time tracking algorithm for tracking multiple 2D objects in video sequences. It is designed for online tracking applications where only past and current frames are available and the method produces object identities on the fly. While this minimalistic tracker doesn't handle occlusion or re-entering objects its purpose is to serve as a baseline and testbed for the development of future trackers. The algorithm expects the bounding box coordinates and the confidence of the detection for each vehicle and gives back the detection bounding box and an index for each tracked vehicle. The indexing behavior is influenced by the three input parameters: *max\_age, min\_hits* and *iou\_threshold*. The parameter *max\_age* defines the maximum frames that an index will be kept for, we set it to 60. The *min\_hits* parameter specifies how many detections a vehicle must get within the *iou\_threshold* to consider it one vehicle and to index it. We set these values for 5 and 0.4. The assigned indices can be seen at the top right corner of the bounding boxes. The user can also see the driven path of the tracked vehicle with red and green lines on the output video.



8. Figure: The tracked and indexed vehicles

Now that we could track individual vehicles and index them, it was time for the velocity estimation. For that, we tried 3 different techniques.

At first, we measured a region of the image (between to dashed lane pieces) calculated a pixel/meter ratio as we knew how distant the two lane pieces in reality are. This solution wasn’t working properly however, because we assumed a constant ratio across the image and our test video footage was shot from above. This resulted in smaller velocities as the vehicles were approaching the camera.

The second technique we tried to use was the pinhole camera model. The pinhole camera model is a simplified representation of how light travels through a camera and forms an image on the image sensor or film. The basic idea is to compare the size of the object in the image to its known size or a reference object with a known size.

where:

* Distance to object is the distance from the camera to the object we're trying to measure (in the same units as the focal length).
* Actual height of the object is the real height of the object we want to measure (in the same units as the focal length).
* Focal length is the distance between the camera lens and the image sensor (usually measured in millimeters).
* Height of the object in the image is the height of the object as it appears in the image (measured in pixels or any other consistent unit).

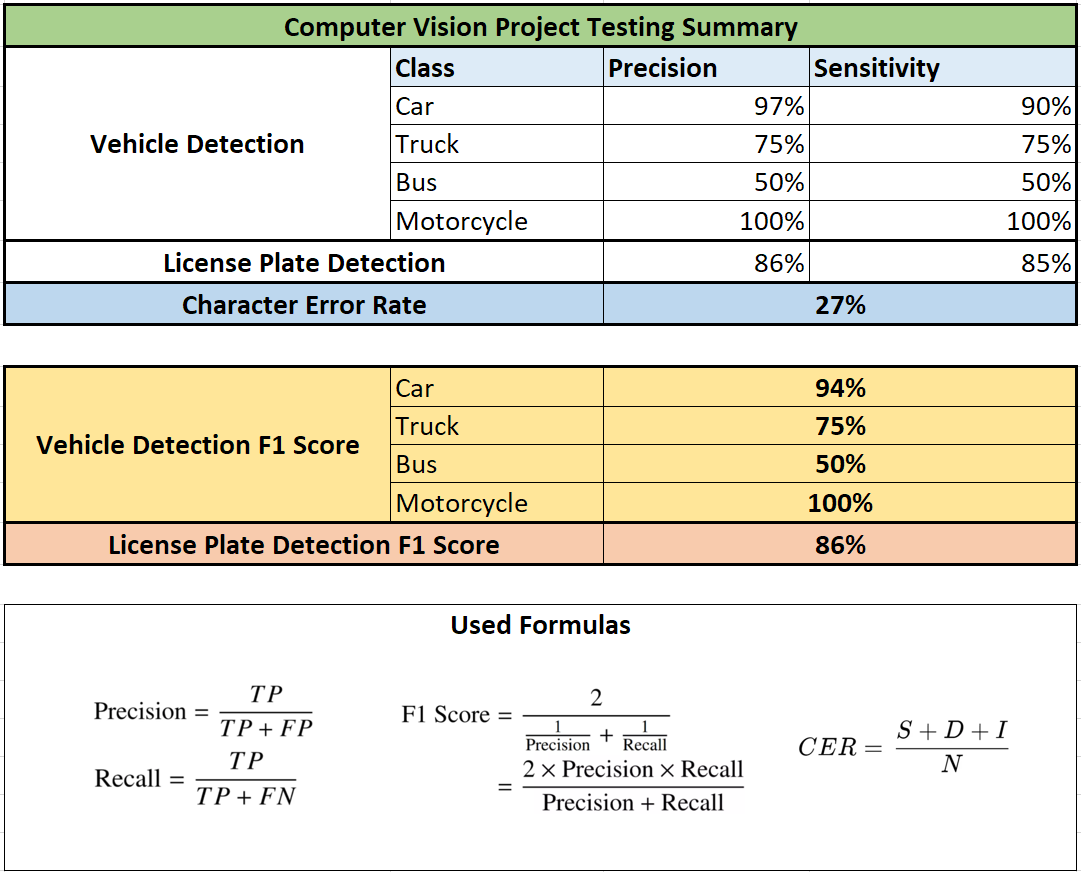
This also wasn’t working for us, because the approaching vehicle’s size wasn’t changing dynamically enough resulting in very slow velocities with vehicles close to the camera. The reason behind it was again, that we took the footage from above.

The third technique solved our problems. It consisted of drawing two horizontal lines on the image, between which the distance is known from the real world and watching when a vehicle crosses these lines. At the start of the program a window pops up for the user that shows the first frame of the video. On this image the user can select two points with two clicks and can give an input in the terminal to the program about the points distance in the real world in meters. The vehicles have their bounding box centers appended in their dictionaries, subtracting the last two from each other can give us the direction where the vehicle is heading. If it comes closer to the camera, we watch where it first crosses the upper line defined by the user’s click and then the second crossing across the bottom line, resulting in two frame indices one for each crossing. Subtracting these gives us a delta, that tells us during how many frames the vehicle passed the area which sizes we know in reality. Upon dividing this real distance by the delta in frames and multiplying it by the frames per second of the video we can get the velocity of the vehicles. The estimated speeds are indicated on the bottom right corner of each vehicles bounding box that crossed the measuring lines.

1. **Testing**

When the algorithm was fully ready, we shot several test videos and burst photos in the city with an Olympus OMD E-M10 camera and a 45 mm f1.8 fixed zoom lens. The most suitable location was at the Petőfi-Bridge, on the Buda side, as we could film cars driving along the quay from the bridge from above, similarly as a fixed speed camera would.

We made 96 test images for vehicle detection, 104 license plate detection images and 58 optical character reading images for license plate text recognition. The results of the evaluation can be found [here](https://github.com/Kurtiadam/speedcam_cvs/blob/main/documentation/Summary%20Calculations.xlsx). We calculated different performance evaluation metrics; their results can be found in the 9. Figure.

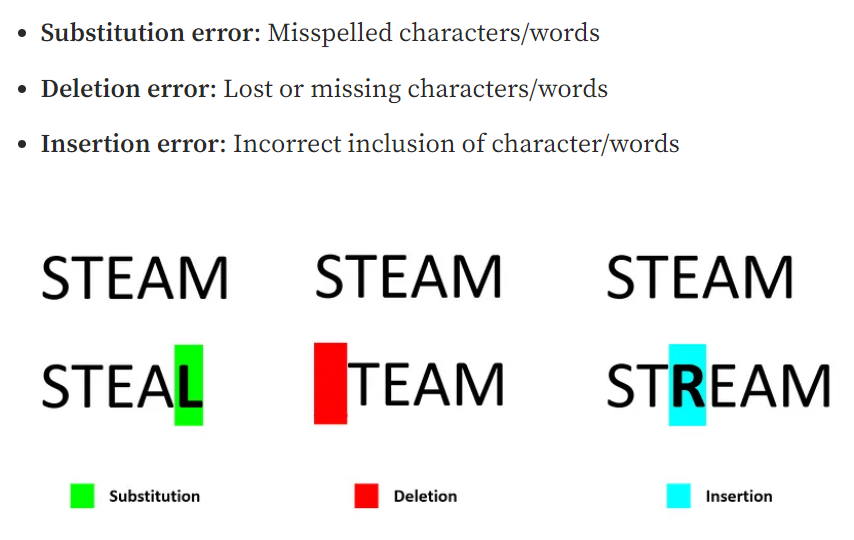


9. Figure: The results of the testing

For the vehicle detection we assumed an 80% confidence threshold, above which the prediction was considered as valid. We can see that we achieved a high precision, sensitivity, and a resulting F1-score for car detection, which category was overrepresented in the testing images. The other classes have varying metrics, because they were underrepresented, e.g., the bus class had only 2 occurrences. Because of this, they don’t affect the overall goodness of the model.

For the license plate detection, we assumed a 60% minimum threshold. The precision and sensitivity and thus also the F1-score is around 85%.

Our optical character recognition algorithm achieved a 27% character error rate (CER), meaning that almost ¾ of its predictions were correct. This was calculated from the error components seen at 10. Figure.



10. Figure: The error components of the character error rate [4]

1. **Summary**

This project, in which we developed an algorithm for car detection, license plate recognition, and speed estimation, has been an incredible experience for us. Over the past period, we have put in intensive work to understand the principles of computer vision, the latest methods in object detection and character recognition, as well as techniques for speed estimation.

Working on this project not only enhanced our theoretical knowledge but also provided us with practical experience. We encountered challenges and difficulties along the way, which we successfully overcame. Preparing the data, debugging errors, and fine-tuning the algorithm were all tasks in which our team collaborated effectively.

In conclusion, this project has been a rewarding journey that has allowed us to delve into the fascinating field of computer vision and machine learning. We have gained valuable knowledge and skills that will undoubtedly be beneficial in our future endeavors.

# **Literature**

[1] Tesserocr: <https://python.plainenglish.io/tesserocr-vs-pytesseract-d6720207bb54>

[2] Picture of OCR Process Flow: <https://nanonets.com/blog/ocr-with-tesseract/>

[3] YOLO training annotation: <https://github.com/ultralytics/yolov5/wiki/Train-Custom-Data>

[4] Character error components: <https://towardsdatascience.com/evaluating-ocr-output-quality-with-character-error-rate-cer-and-word-error-rate-wer-853175297510>