Creative Technology
Module 7 “Hands-on AI” Project Report

Self-Driving Car

Ronin de Vreeze, Daan Meinardi

Supervisors: Selin Hülagü

Date: 15/04/2026

Department of Interaction Technology,
Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

1 Introduction

Welcome to our project report of module 7, in this report we document the technical details of the project and show why we make certain decisions in the creation of our self-driving car.

In our installation we focus on object recognition by camera, by using convolutional neural networks. We use live camera footage to track which road signs are present in the image and another model to find possible paths in the road. The car is running and internal state machine that controls the wheels based on the output of the two models, creating a car that can drive itself around a small track and obey the traffic signs around it.

In this report we will take you through all the steps to reproduce the system, from data collection to live prediction. How noise in the camera data challenged us to apply smart filtering to increase accuracy and the setbacks we faced regarding certain models.

2 Hardware

- **Frame with 4 wheels and 4 TT-motors** [1]:
  We chose this frame because of its size and simplicity, it has more then enough space to attach components. The whole kit was easily assembled and we added a

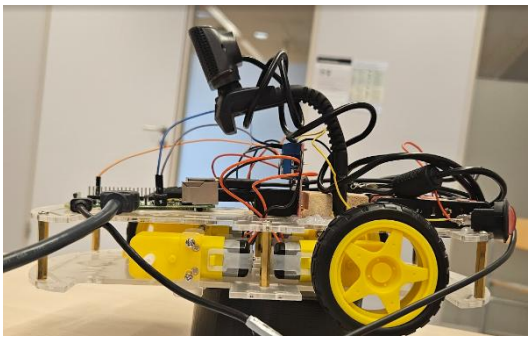

Figure 1

- **Lasercut camera mount**: Used to get the camera into a good position where it can see both the road and the signs.

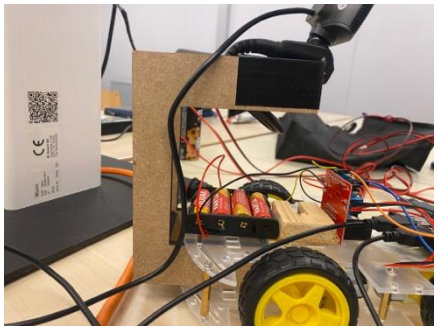

Figure 2

- **Raspberry PI**: We chose this because it is compact and has enough processing power to handle the intersection CNN model locally.

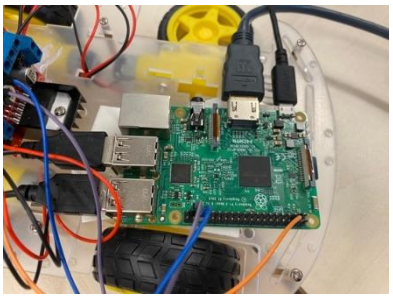

Figure 3

- **Webcam**: Used to gather the data and for the live categorisation.

Figure 4

- **L298N Motor driver:** Needed for controlling the motors on the car

Figure 5

- **Battery pack**: The motors need external power supply, that's why we use 2x4 1.5v AA batteries.

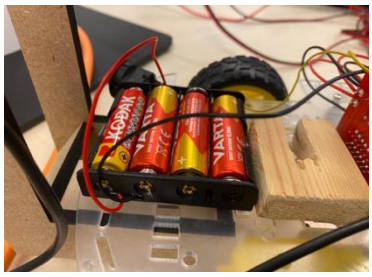

Figure 6

- **Road signs**: 3 different 3D printed road signs that we train the YOLO model on

Figure 7

- **Tablecloth**: A white background where the road is put on. We chose white so that there is a high contrast with the black tape and signs.
- **Duct tape**: Used to make the road on the tablecloth and hold things together.
- **Cables**: to power the Raspberry PI and connect all the electronics

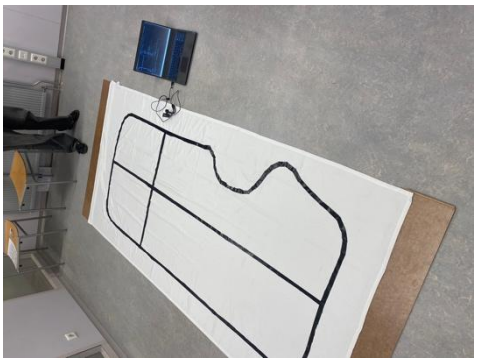

Figure 8

- **Laptop:** not to forget that the laptops were used to write, research, and run programs. The yolo model was run on a laptop and code on the raspberryPi was changed remotely from a laptop. A crucial part of all the steps and could not be replaced. We truly used the power and mobility of these computers.

System architecture:
The webcam is connected to the Raspberry Pi, which controls the motors via the DC motor controller. On the Raspberry Pi we have a local TFLite model to classify the current road paths ahead (also called the intersection-model from now on). Furthermore, the video is also streamed on a hotspot using web-sockets to an external laptop to run the road-sign model, since it is much larger. This laptop is also used to display essential information while running the models.

Setup assembly:
This chapter quickly describes the setup process to get the car running. Firstly we assembled the kit and wired all the cables so that the Raspberry Pi can control the wheels and the speed via the motor controller. The wooden cutout was glued together and later glued to the car construction.

Next, the Raspberry Pi was connected to an external display, keyboard and mouse in order to program it, making sure it was also connected to the local hotspot running from one of our phones. After all the code was written the display could be disconnected and everything was setup on the device itself.
To run the scripts we decided to use SSH, since it works very quickly and allowed us to disconnect the HDMI cable to the display. From one of our laptops, we used SSH to get into the Raspberry Pi and run the controller script from there. The script would print debugging information and model statuses to the console so that the models could be evaluated quickly.

Lastly, the motor controller was connected to a separate power supply to give the 12 volts needed to give the car enough power. We added a switch between them to easily turn on and off the wheels.

The script also uses websockets to stream the video feed to an local endpoint, which can be accessed as long as both devices are on the same network. The script with the sign classification uses this feed and can be run from anywhere.

3 Data collection

3.1 Process of data collection

In our project we did not need any people for data collection. All our data is collected with the webcam mounted on the frame and by simulating the route where the car encounters signs and follows the road. We developed a custom script using python and OpenCV to show, record label and save all the images to the right folder at the same time.

For the intersection CNN:
The first model needed to have images of the different intersections, the classes we decided to distinguish were the following: Straight/curved road, 4-way crossing, T split, split to left and split to right.
The most efficient approach we found was to have the camera running via a python script and continuously save the image to a folder for the right class. The class could be changed by pressing keys on the keyboard. After a couple minutes of data collection, we had a dataset of around 300

image per class, all 160x140 in size. With this raw labeled data of the intersections, it is ready to be processed.

We made sure the dataset was diverse by capturing the road lines from different angles and perspectives, sometimes on the line and sometime slightly off of the line. This made sure that the model would still be able to recognize the signs, even when it was not directly in front of them. This would be aided by pre-training augmentation as well, as discussed later on in the report.

Sign Detection (YOLOv8)
The YOLO model doesn't' need much data to start training, so we settled for about 50 images of the signs, from different angles and different distances. These images were taken with the same webcam and resolution as we would use on the demo day, to avoid losing accuracy because of changes out of our control.

3.2 Preprocessing of data

To make it easier for the intersection-model we made a filtering pipeline, this pipeline was abstracted into its own function in a helpers.py file, so that we could easily apply the same filter when we are predicting live images. The filter started by applying a grayscale filter and cutting of the top half of the images (it prevents it from looking to far ahead). Next, we applied a median blur and threshold, this creates smooth rounded edges around the line the car has to follow. The smoothing and thresholding was applied one more time to increase the effect with the hope of reducing noise and increasing performance.

The motivation behind the filter is to reduce data. We thought about what the model would need to recognize the right class and found that it only needs to know where the lines is and where the line is not. That is why we minimalized the input by only including black and white pixels for the line.

Figure 9: Unprocessed image

Figure 10: Processed Image

Then, before the training started, we applied a simple 4 step augmentation to the dataset, changing the perspective, zoom, rotation and translation slightly. The reason for this was that the model

should also recognize the lines when it is not directly in front of them, it also increases the amount different images the model will encounter.
Another advantage of filtering is that the models will remain accurate even when they operate in a different environment, since most of the difference between environments can be eliminated with some noise filtering. This was very important for the demoday.

As far as the labels are concerned, there was not much data preprocessing to do, except for one-hot encoding the classes to remove any useless meaning the model might find in their order.

Sign Detection (YOLOv8)
We trained our own YoloV8 model by using labelImg in python to label all the road signs manually and then training the "yolo26m.pt" with a training set. We set this up using a YouTube video [2] that guided us through the labeling and the training. As you can see in figure 11 the labeling of the data was done manually. In every image we drew a box around the feature and labeled it. The label information is stored in a file and matches the image name. From the image and labels file a sample is taken and put in a validation images and validation labels file.

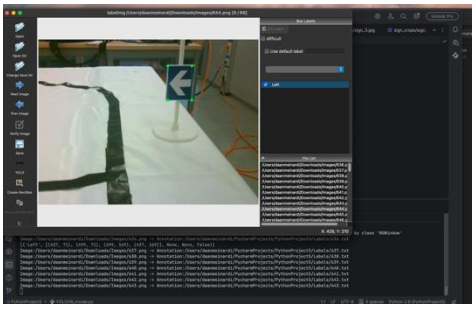

Figure 11

For the Yolov8 model to train in PyCharm it needs a setup .yaml file that has the path to the data and classes info
as in figure 12.
Then now its fully ready to be trained with Ultralytics.

```
train: /Users/daanmeinardi/PycharmProjects/PythonProject5
val: /Users/daanmeinardi/PycharmProjects/PythonProject5

nc: 5

names: ["Stop","Left","Straight","Right","Turn"]
```

Figure 12

3.3 Description of the data set

The dataset consists of images taken from a fixed camera mounted on the car. Because the camera angle never changes, it has a consistent view of every important feature on the track. The track itself is a controlled environment made up of tape intersections and 3D-printed signs, which keeps the visual data focused and easy for the models to process.

Translation to real world:
Since the data was collected by moving the car manually it might not actually translate well to the real world, since the movement of the car might be different, faster, slower, earlier, snappier, etc. This is a risk of using this approach, but we tried to minimize it by moving the car like how it would drive when running, staying low to the ground and not tilting the camera too much.

4 AI model

4.1 AI algorithms applied

In our project we planned to use 2 different CNN models: One model to recognize intersections and one custom trained Yolov8 model to crop and classify the road signs out of the full sized images.

The intersection model is a Convolutional Neural network with the following architecture:

- 3 Convolutional layers with 3x3 kernels, range from 32 up to 128 filters, with max pooling in between
- A global average pooling layer to flatten
- Dense layer with 128 neurons
- Dropout layer
- Final 6 neuron output layer

From the start it was clear that we would use a CNN for this problem, since it is exactly what they are made for. CNN's are the best choice then data is spatially (and semantically) connected in a grid, like in a image. Furthermore, we argue that the edge detection and line identification strengths of CNN's would work well to classify lines and intersections of lines.

Since the classes were not perfectly equally distributed (ranging from 1932 to 697), but also didn't want to remove too many images, we used a solution we came across during our research: class weights. This easy-to-implement solution adds different weights to the cost function for different classes, in order to compensate for the imbalanced dataset.

The final step was to compile the model using the Adam optimizer and using categorical cross entropy as the loss function, since it is the best choice for one-hot encoded data. The training data was split into 80% training and 20% testing.

After tuning hyperparameters like the kernel size and number of convolutional layers, we trained with a batch size of 64 we noticed that the accuracy kept increasing until around about 30 epoch, so we trained a final model with 40 epochs and saved it.

Sign recognition
In the beginning we focused on developing a custom CNN for feature recognition. However, we noticed that even with large amounts of labeled data, the model struggled to consistently identify signs in the frame. Through different iterations of CNN structures, the same issue kept coming back: the model was overly sensitive to the positioning of the object. In many cases, the location

of the feature in the frame seemed to influence the classification more than the characteristics of the feature itself.

To address this, we implemented extra pooling layers to improve translational invariance and replaced the traditional Flattening layer with Global Average Pooling (GAP) before the dense layers. This change was intended to reduce overfitting. While earlier models showed high accuracy on the training set, they performed poorly on the validation set. GAP helped to reduce this by forcing the model to identify the presence of a feature regardless of its specific coordinates.

Even after these architectural improvements, the CNN remained too inconsistent for real-time decision-making. While a CNN is theoretically the perfect choice for image classification; ignoring position and edge detection, the complexity of our track environment required a more fail proof solution. This led us to use YOLOv8, which is specifically designed to handle both localization and classification in a single efficient pass, providing the stability that our custom CNN lacked.

The Yolov8 model is also a CNN at its core, it uses convolutional layers to compute matrices and filter out different parts of an image. It is just so well defined that it is able to do the locating and classifying of the sign in one go and this made it the perfect choice for us to implement into our project. We needed to be able to recognize some signs to make it any sort of interactive at the demo day and since the CNN for the intersections was a success, we could take a different route for the sign recognition. The Yolo model was easy to train and required relatively little preprocessing, just a bit of labeling, as said in part 3.2 We used a medium sized model, since we were able to run it on a laptop and weren’t limited by any microcomputers power, but still wanted the speed of a smaller model then the large ones.

4.2 Quality of the AI model

On the figure below you can see the confusion matrix output by the intersection-model after training, as visible it is nearly perfect on the whole testing set.

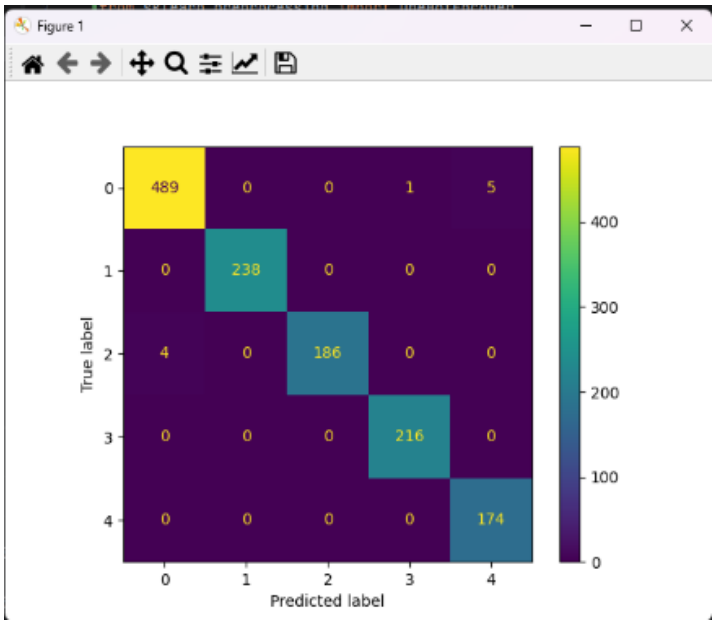

Figure 13: Confusion Matrix for intersection classification

However, since images are large and diverse, we also converted it to a TFLite model and ran it using the live input from the webcam. We verified that the model would correctly predict the intersection based on the filtered video input, see figure 14 and 15.

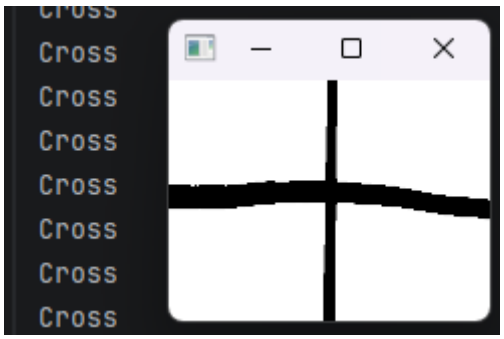

Figure 14

Figure 15

Immediately we noticed that the model was noise in its output, often getting it right 9 out of 10 times, but sometimes predicting something else. In the final script, one misclassification could mean that the car makes an unexpected turn, so we needed to fix it.
We found the simplest solution was also the best, a rolling window. Using some simple NumPy functions we were able to take the mean of the last 10 classifications, for each class. This creates a mode smooth output, where 1 or even more misclassifications would not disturb the output.

We used the Yolo26m model to train, which is a medium Yolo model from this year. Initially we trained the model on a laptop over 10 epochs and this alone took about an hour to train and the results were okay on the validation set. When we tested the model live on the webcam view, we noticed it was very bad at recognizing the sign in any movement or with any noise. Rethinking our strategy, we realized that we need to use images from various distances and then frame and label them correctly. After creating a larger and more diverse dataset we tried to retrain the model with a lot more epochs, but this came with new issues. It would take ages to finish the training this way, so when even training on the GPU didn't make it quicker, we learned that we could use google Colab's cloud-based GPU and this made the training go very quick.

From this training we eventually found our best model that we used on the demo day. This model can classify each sign as seen on figure 18 and performed with a high enough accuracy to also classify it live on a test run. Sadly, when mounting the camera on the car and testing in a real scenario we noticed that the left and right sign were basically never being detected and with this happening shortly before the demo day we decided to fix all the driving logic first instead of looking at more training or tuning. We do think that since the model was trained on 3 different blue signs, color plays an important role in why it has trouble distinguishing the arrow signs. In this case its then good for us that it chooses to identify all three as one instead of performing bad on all three.

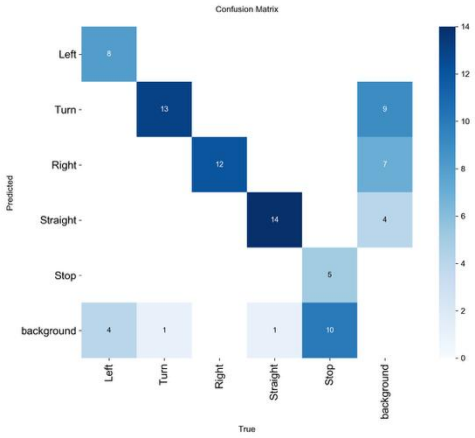

Figure 16: confusion matrix for the first Yolo model

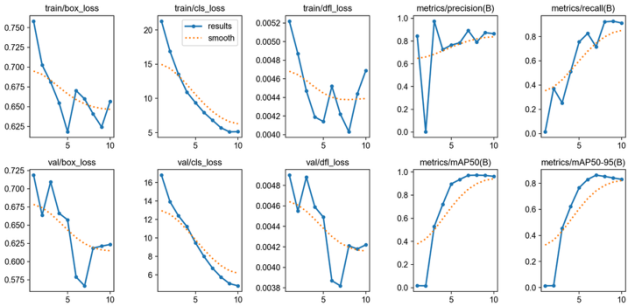

Figure 17: Different loss functions automatically generated by Ultralytics

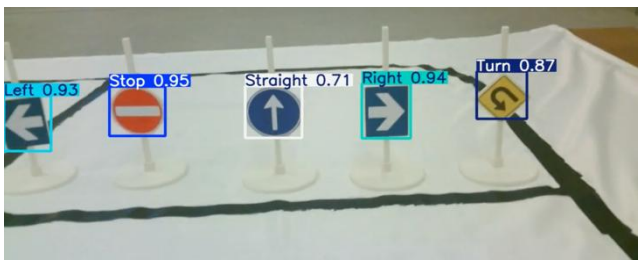

Figure 18: Sign detection and classification

5 Installation

5.1 Description of the Installation

The last component missing was a model to steer the car to stay on the line, luckily this was easy to implement since we already had a filtered image of the line. A very simple algorithm analyzing the horizontal offset of the line (relative from the center) was already good enough to keep the car on the line.

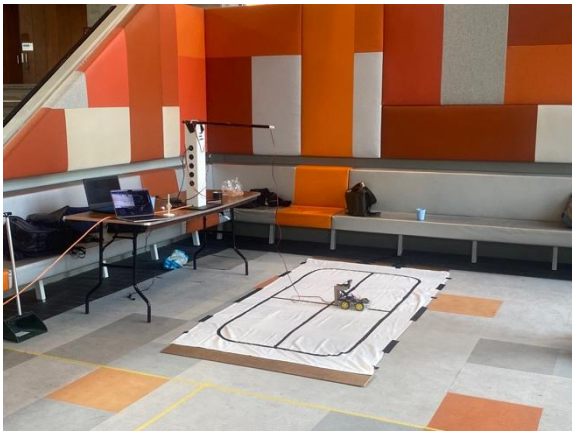

Figure 19

The installation consists of the track, As you can see in figure 19 we set up the installation in an open space in the foyer of HG building at VU. The installation has the purpose of showing people that the car can follow lines, identify and act upon certain intersection types and identify and act upon certain signs. We chose to keep a simple road structure so that the CNN model can show off the recognition of the intersections and little noise would be in the camera view, of course this has been explained also in the data collection section.
We made a DIY cable guide to keep the RaspberryPI
power cable off the ground and in most cases this worked great. One laptop was used to show what the car was seeing trough the webcam and showed live sign classification and intersection recognition. Like this people could easily understand what was happening and why the car made certain decisions.

5.2 Evaluation

The installation was not as complete as we wanted when we proposed the plan in the beginning of the project, but still we had a good product to showcase and were able to show key insights into why certain parts didn't work. We were able to let the car drive over the lines and at any given intersection, choose a direction and let the car steer that way. We could let people decide which path to take and see if the car would stay on track. Also we still had the sign prediction model very accurate on three different signs: Stop, Straight and U-turn. With these signs we could show off how the model was detecting the signs on a laptop screen and let people place the sign in front of the car and watch it react. Because we ran the direction choosing on the laptop keyboard, our main part of user interaction was way worse then we planned. If users could have continuously placed signs in different locations on the road and watch how the car would react it would have been more interactive. We did manage to work around this by letting people choose a sign to put in front of the car or let them choose direction.

We did manage to avoid any breaks or bugs during the demo time. The installation was running for over two hours without issue, and we were able to show people within minutes how the

installation works and explain some of the logic behind it. People seem very interested in figuring out how the data is shared with the different devices and how the car knows where to go.

6 Conclusion

Looking back at the past two weeks, this project has been a significant learning experience in the practical application of AI and systems architecture. We successfully moved from the initial planning and data collection stages to a fully functioning installation that demonstrates real-time autonomous navigation.

Our technical journey highlighted the importance of choosing the right tool for the specific task. While we successfully implemented a custom CNN for intersection recognition using a filtered grayscale pipeline , we learned through iteration that sign detection required a much larger CNN model like YOLOv8. This turning point allowed us to overcome object location sensitivity issues that had previously hindered our manual CNN builds.

We are really happy with the way we showcased our project at the demo day, we hope it showed our optimistic starting goals, determination to make it work and skills to create a backup plan and execute this less than 24 hours later. We were able to explain and discuss possible improvements on the demo day itself and got nice feedback on how these could be implemented. In general it was a successful project in which our learning goals where achieved.

Reflecting on the project, we would change/improve the following:

- Create a larger and more diverse dataset for the sign model to train on and apply the full steering logic just with signs.
- Improve the line following so that even when it fails to keep the line in vision, it knows where it left the screen and by that is able to find its way back to the line. This would make the car feel way more independent/smart

We are most proud of:

- Switching our strategy the day before the demo day because we werent getting anywhere and still delivering a working project the next day
- The way we worked around the numerous issues each extra layer of depth the code gave. Learning about code logic along the way.

7 Links and references

[1]Frame and motor https://www.otronic.nl/nl/chassis-met-4-wielen-en-4-tt-motoren-robot-basis-d.html?source=googlebase&gad_source=1&gad_campaignid=19639985996&gbraid=0AAAADCZK5qvAgAm15ivD9coHU7JQxepMe&gclid=Cj0KCQjwv-LOBhCdARIsAM5hdKd5bERIrKD15NAlLVsKQYLNclbP_oWneTIzDZMKnG9IYaNQFsPsEeEaAtHzEALw_wcB
[2] https://www.youtube.com/watch?v=gRAyOPjQ9_s
[3] https://github.com/ultralytics/ultralytics?tab=readme-ov-file