Masterpraktikum - Simulation-Based Autonomous Driving in Crowded City

Philipp Kutschmann, Nikola Nincic

***Abstract*—Machine learning, particularly deep learning using Convolutional Neural Networks (CNNs), has gained prominence in various industries, including the automotive sector. One compelling application of machine learning in this context is autonomous driving. This paper explores an end-to-end approach powered by a CNN for autonomous driving. The method is tested in two simulator environments—a simple track with no traffic and a complex city simulation with traffic lights, other vehicles, and intersections. In the presented implementation, the end-to-end approach is combined with YOLOv8 for object detection. This is done to address challenges related to traffic lights and vehicle recognition. Further, the achieved performance results in the two simulators are presented. In the first simulator, which involves a straightforward circular track, the end-to-end approach successfully keeps the vehicle on the road. However, in the more complex second simulator, with city-like scenarios and traffic, challenges arise due to issues in the simulator. Lastly, the simulator-related issues that hampered the overall performance of the autonomous driving algorithm, including delays, image resolution, resizing problems, unrealistic traffic behaviour, and high computational demands are discussed.**

I. Introduction

Over the past years, machine learning has gained more and more traction. It has found its way into many industries, including the automotive sector. A natural application of machine learning in that context is given by autonomous driving. However, as it is a safety-critical task, it is essential that the model controls the car without interference throughout diverse scenarios that occur while driving. Therefore, this task requires a solution that can deal with its inherent complexity. Deep learning approaches have shown impressive results in many different areas ranging from object detection to playing chess. These results give us reason to believe that it also might be a viable tool to deal with the task of driving a car. As humans, we mostly use visual data to steer the car through traffic. As a common network type for processing images, CNNs represent a network type to extract information of visual data. Based on their training, these networks can detect relevant features in an image, e.g. road signs or streets, and use these features to predict a reasonable action. This method is an end-to-end approach as it directly infers the controls based on the camera input. In our work, we focused on this end-to-end approach powered by a CNN. The large advantage of this method in the context of autonomous driving is that the training data is not required to be labelled by humans. After establishing the groundwork by implementing a ResNet50 architecture and testing it on the Cifar-10 dataset, we tested our method in two types of environments. The first one is a simple simulator with a single track and no traffic. The second one is a more complex simulator of a small city with traffic lights, other cars and intersections. The details of our method and results will be presented in the following sections.

II. Related Work

The technical side of autonomous driving has been a long focus of research. From the first radio-controlled and remotely operated car in 1925 and the autonomous car of Prof. Dickmanns 1994 [6] to today's approaches multiple other methods have been tried. Currently, the most common method is a modular software stack that divides the task into multiple subsystems as seen in figure 1. A main advantage of this system is the explainable decision-making. Therefore, each modular component can be improved on its own and it is easy to find out the bottleneck, should the car not reach the desired performance.

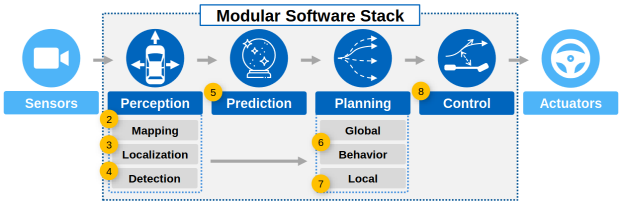

Fig. 1: Modular Software Stack currently used for autonomous driving.

Other approaches try to directly predict the car controls send to the actuators based on the camera data with help of a neural network. The most popular paper on the topic of this end-to-end autonomous driving by Bojarski et al. [2] was the baseline for many research papers to come and still achieves impressive results. In their experiments, they used a car with a front-facing and two side-facing cameras. The only predicted output of their model is the steering angle. With the help of carefully chosen training data and data augmentation, the trained model was able to steer the car with only a few human interceptions on real roads.

Since the publication of the ground-breaking paper "Attention is all you need" [9] by Vaswani et al. that introduced the concept of attention layers, transformers have found their way in many application areas of machine learning. Autonomous driving is no exception to this.

In their work from 2021 [7], Prakash et al. present a multi-modal fusion transformer for end-to-end autonomous driving. They argue that neither image-only nor LiDAR-only is enough to safely navigate through real traffic, as global context can only be acquired using both. Using a transformer, they fuse

Fig. 2: Screenshot from the CARLA simulator [1]

the information of the sensors and predict an action based on it (see figure 3). An interesting detail here is that only a single-view RGB image is used for the visual information. For 3D information, they use LiDAR bird's eye view projections. According to their paper, this projection includes 32m in front and 16m to each side of the car. For the task of navigation, the authors use sparse GPS locations. This approach to end-to-end autonomous driving led them to impressive results in the CARLA simulator [5] (see figure 2) that provides complex settings with traffic lights, pedestrians and other traffic participants. However, detecting red lights and handling them accordingly represents an issue for this approach.

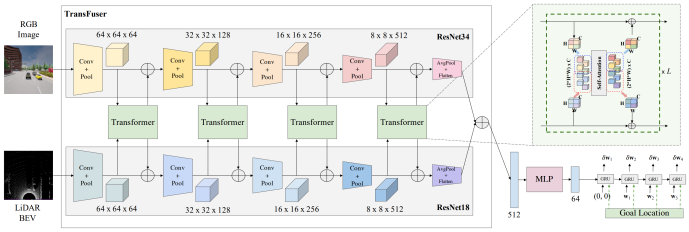

Fig. 3: Examples for the traffic light classes

A similar method to this was described by Xiao et al. in their paper [10]. In their paper, they describe a multi-modal approach using convolutional and fully connected layers, instead of transformers (see figure 4).

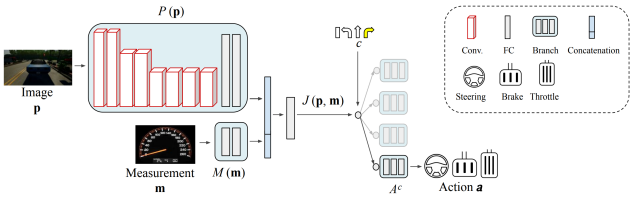

Fig. 4: Examples for the traffic light classes

III. Method

As a method that showed promise in various works [7][10][2][3], we chose to explore end-to-end autonomous driving on two simulators for this work. In particular, we are using a CNN to process images provided by front and side-facing cameras attached to the simulated car. We recorded the images and corresponding vehicle commands for the training by driving the simulated car ourselves. In the first simulator, the expected output is only the steering angle. In the second one, the throttle and brakes are additionally predicted. In both settings, the recorded vehicle commands are used to supervise the training process. Further, we chose a simplified version of the ResNet50 architecture and MSE-loss to train the model on our custom dataset. This was already enough to obtain decent performance on the first simulator with reduced complexity. However, for the second task, this approach delivered unsatisfying results. For this reason, we chose to incorporate the object detection provided by the YOLO-model [8], specifically YOLOv8. Using YOLO we can improve the performance of our model while keeping the advantage of an end-to-end network. Detecting other cars and traffic lights, we have more information to decide on an action. However, since we use a CNN and not a transformer, we did not feed this information into the neural network but decided to combine the end-to-end approach with a rule-based algorithm. This decision required us to train a second, lightweight CNN that takes an image of a traffic light as input and predicts its state. We chose to divide the states into three classes (figure 5):

- Red (including yellow)
- Green
- Ignore (unidentifiable state + pedestrian traffic lights)

Fig. 5: Examples for the traffic light classes

This lightweight model is trained separately and not used when training the end-to-end model, but only during testing. Since irrelevant traffic lights might be captured by the cameras, we chose to use the majority of red/green detected traffic lights to make a decision. In case red traffic lights dominate, the car is forced to stop, otherwise, the CNN takes control of the car.

Another rule we added to increase safety, is stopping the car when YOLO detects another vehicle close in front of ours. This is determined by checking for a large vehicle bounding box with its centre within an area in the middle image, as shown in figure 7. How we incorporated these modules into our AI is demonstrated in figure 6 and listing 1 in the appendix.

IV. Results

A. First Simulator

Setting: The virtual environment in this simulator consisted of two selectable tracks, one of which was a very simple circular road with clearly marked road edges. The other one was a more complex road through a mountain landscape with less clear edges. Neither traffic nor intersections are included in the environment.

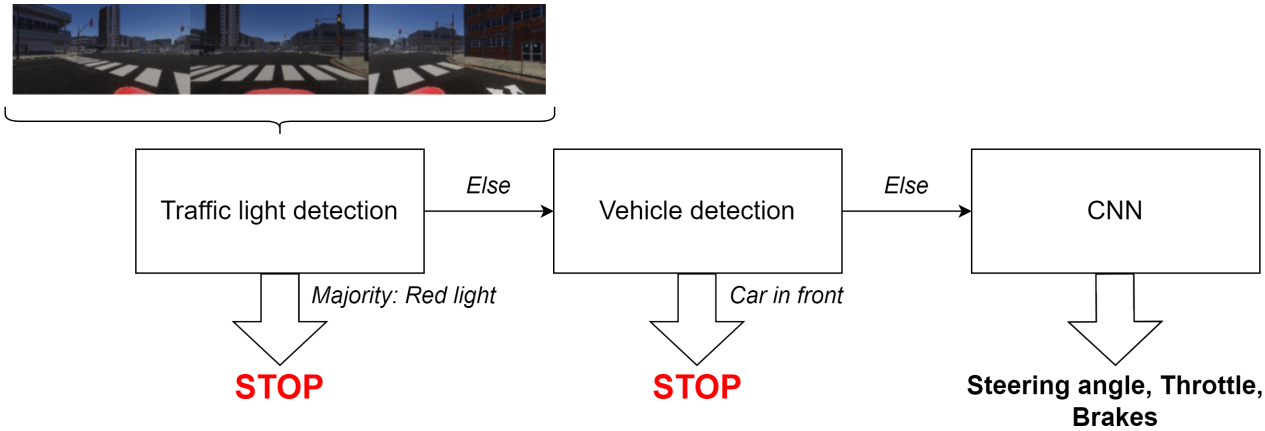

Fig. 6: Our Model architecture

Training: The training dataset is recorded using an included tool in the program. We drove the car ourselves and the tool created data samples from the drive. Each sample consists of a combined image and the according steering angle.

Testing: In the testing phase, the simulator runs in parallel to the model and the images from the simulator are fed to the model in real-time. The model then returns the predicted steering angle to the simulator. With the received input taken into account, the simulator now renders the next frame and the loop repeats.

Achieved performance: After fine-tuning our model, it was capable of driving confidently on the given roads. It manages to drive safely through all sections, with sometimes cutting corners if the racing line was driven during recording of the training data. Additionally, we believe that the pure end-to-end approach is definitely capable enough to achieve perfect driving in the two scenarios.

B. Second Simulator

Setting and obstacles In this simulator the virtual world consists of a small city with intersections, traffic lights and diverse buildings. Additionally, other cars and buses are driving on the streets. These vehicles are computer-controlled and follow the traffic rules most of the time.

Training: As in the first simulator, we obtain our training data by recording ourselves manoeuvring the car through the map. The functionality for that is provided by the simulation software itself. Each sample of the recorded data contains a combined low-resolution image from the left-, right- and front-facing cameras, as well as the velocity, steering angle, throttle and brake. The low quality of the training data made recognizing the state of traffic lights from a distance a difficult task, as seen in figure 8. Furthermore, in order to avoid ambiguity in the route planning, we trained the model on a single route through the city and always took the same lane during recording.

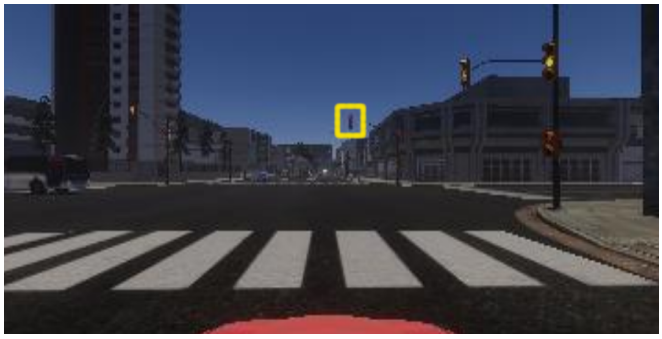

Fig. 8: Image from the middle camera in the second simulator

Testing: For testing a trained model, the simulator includes an autonomous mode. Similar to the first simulator, the images recorded by the three virtual cameras are sent to the model in real-time and the model output is sent back to the simulator. The data flow in this loop can be observed in figure 9. However, as the simulator is flawed, the images were fed to the model with a few seconds delay. Additionally, the simulator was computationally demanding causing a low frame rate, even on stronger machines. These two issues combined, but especially the first one, made the development of a decently performing autonomous driving model extremely challenging.

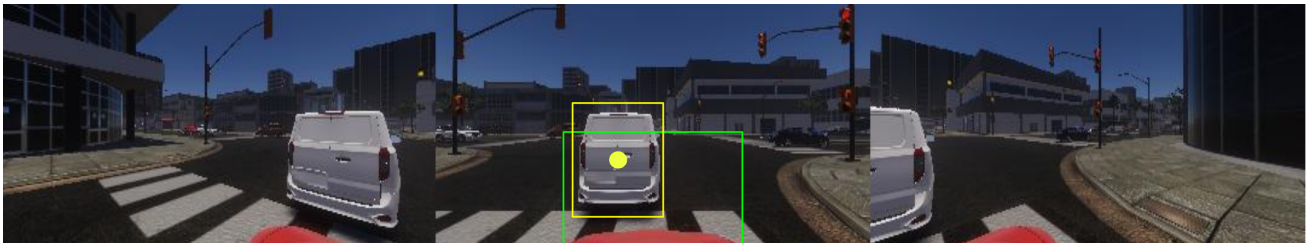

Fig. 7: Examples for the traffic light classes

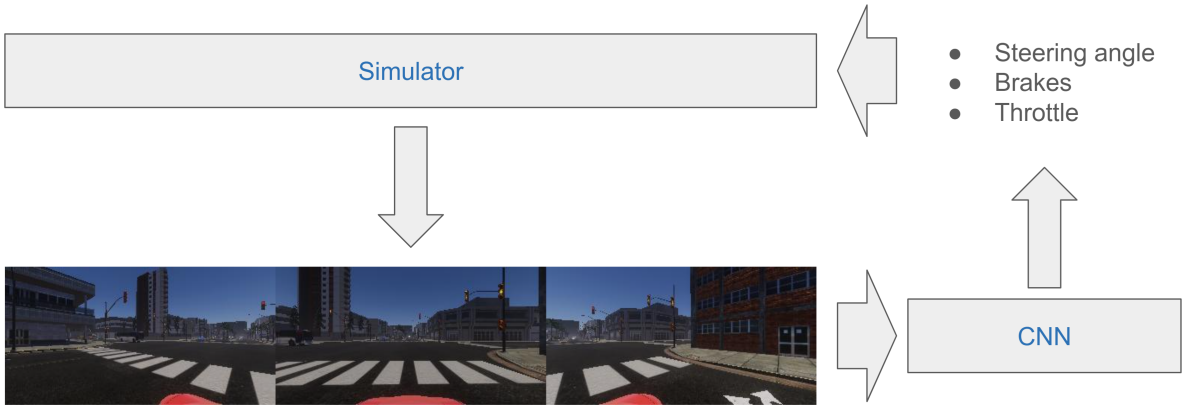

Fig. 9

With the delay between the simulator and the model, the AI was forced to predict an action for a future, unknown state. This problem is visualized in figure 11.

Achieved performance: In this scenario, our model's performance was not satisfactory, due to the aforementioned reasons. The high delay caused the AI to react too late to critical situations like traffic lights, other vehicles or turns. All in all, we were not able to create a model that could efficiently handle the problems of the simulator.

V. Discussion

The autonomous driving algorithm for the first simulator manages to keep the car on the road and avoids crashing even after multiple rounds. This decent result is due to the limited complexity of the presented scenarios. This shows that the pure end-to-end approach is a viable solution for the task of lane-keeping. Therefore, predicting steering angles from input images handled in succession provides a method that can control the car in the desired way.

However, if the the traffic gets more complex, this method has some unresolved issues. One open question is how to handle intersections. As the pure end-to-end method has no global navigation, it is ambiguous to the network where to go. One possible solution would be to add an action command from a navigation system as an additional input at intersections [4] but as there was no map available this option was not an option. The additional lack of lane markings at the intersection combined with underrepresented training data of this scenario oftentimes causes the algorithm to break. Another challenge is the prediction of other traffic participants. A common case present in the input images is a car in front of the ego vehicle in front of a green traffic light (see figure 10). Just from this one image, it is impossible to predict the speed of the other car and therefore our own action (e.g. is the car standing still as the light just turned green and we need to break OR have we been following the car down the road and we can keep our speed?).

Additionally, our results in the second simulator were massively impeded by issues of the simulator itself. We identified five major problems:

Fig. 10: Demonstration of speed estimation issue

1) There is a large delay (multiple seconds) between what is happening in the simulator and the images that get sent to the network for evaluating the car controls. The delay is visible in figure 11 as the upper window displays the image sent to the network while in the lower window, the current position of the vehicle can be seen. In the upper image, the crossroad is still far away while the car is actually already about to cross the intersection. This infers that the network needs to predict the steering angle, throttle and brake multiple seconds into the future based on the images it gets. This issue is also present if the network is turned off, so no other computationally complex tasks are being executed causing the delay. As driving a car is a highly real-time dependent task, a few seconds of delay is a major hurdle. Secondly, as the recorded images are matched to the corresponding steering inputs during the recording of the training data, this temporal prediction into the future is not learned during training. This delay is therefore one of the biggest reasons for the mediocre performance of the algorithm for the second simulator.
2) Another issue is the low image resolution of the images

Fig. 11: Delay between simulator and the model input images

sent to the network. In these images, it is oftentimes hard to distinguish a red from a green traffic light even as a human (see figure 8).

3) As the images recorded for training have a different resolution and aspect ratio from the images during inference, resizing the images posed another obstacle. Since the images of the front- and two side-facing cameras are merged horizontally into one image. Simply cropping the images to fit the aspect ratio would result in cutting away half of the side-facing cameras while the middle image remains the same shape. This resulted in further loss in performance since the network needed to generalize to the different-sized images during inference.
4) To enhance the realism of the simulator, we also propose to correct the other traffic participants to follow the traffic rules. The most noticeable issue was that other cars always used the left lane if the street had multiple lanes going in the same direction. Oftentimes, on the left lane turning right is prohibited with lane markings on the ground at intersections. As the cars turned right anyhow, this caused crashes if we wanted to go straight on the right lane (obeying the traffic rules, see figure 12).
5) Lastly, the simulator lacks general performance as a fairly capable PC is required to solely run the simulator. This can be seen and felt by a low frame rate while driving manually and the graphics and texture quality do not justify this poor performance. This reduces the computational resources available for the network at inference.

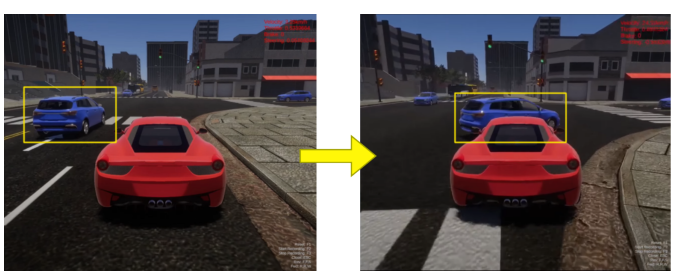

Fig. 12: Illegal right turn by other vehicles in the second simulator

VI. Outlook

The issues present with the end-to-end method require further work to be viable for real-world applications. One of the hurdles was the lack of speed information for other traffic participants. This could be mitigated by either providing other sensor data as additional input to the network or by using a network which can handle sequential data (e.g. LSTMs or Transformers). Both approaches but especially the latter one would require much more training data, longer training times and more compute power than we had available.

Furthermore, it is doubtful that the method of predicting a steering angle from a set of consecutive input images is even an acceptable solution for just lane-keeping assistance for two reasons: Firstly, already existing lane-keeping algorithms with pattern recognition are more computationally efficient. Second and more importantly, ensuring safety and explaining the decision-making of the end-to-end approach remains an open issue.

References

[1] Carla docs. https://carla.readthedocs.io/en/stable/running_simulator_standalone/. Accessed: 2023-09-08.
[2] Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.
[3] Kashyap Chitta, Aditya Prakash, and Andreas Geiger. Neat: Neural attention fields for end-to-end autonomous driving. In *Proceedings of the IEEE/CVF International Conference on Computer Vision (ICCV)*, pages 15793–15803, October 2021.
[4] Felipe Condevilla, Matthias Miller, Antonio M. López, Vladlen Koltun, and Alexey Dosovitskiy. End-to-end driving via conditional imitation learning. In *Proceedings of IEEE International Conference on Robotics and Automation*, 2018.
[5] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An open urban driving simulator. In *Proceedings of the 1st Annual Conference on Robot Learning*, pages 1–16, 2017.
[6] Marco Lalli. *Autonomes Fahren und die Zukunft der Mobilität*. Springer Berlin, Heidelberg, 2020.
[7] Aditya Prakash, Kashyap Chitta, and Andreas Geiger. Multi-modal fusion transformer for end-to-end autonomous driving. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 7077–7087, 2021.
[8] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016.
[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.
[10] Yi Xiao, Felipe Codevilla, Akhil Gurram, Onay Urfalioglu, and Antonio M López. Multimodal end-to-end autonomous driving. *IEEE Transactions on Intelligent Transportation Systems*, 23(1):537–547, 2020.

Appendix

Listing 1: Code sample of our drive pipeline

```
# Make forced stops (e.g. red lights) last for two seconds,
# before continuing the drive
global last_forced_stop
if time.time() - last_forced_stop < 2:
    send_control(0, 0, 1) # (steering angle,throttle,brakes)
    return
global throttle, brakes, steering_angle
# Recognize obstacles and traffic lights using YOLO-net
yolo_out = yolo.predict(source=img, conf=0.4, verbose=False)[0].boxes
yolo_bboxes = yolo_out.xywh # detected bboxes in format X,Y,width,height
traffic_lights = yolo_bboxes[yolo_out.cls == 9]
obstacles = yolo_bboxes[yolo_out.cls <= 8]
img_center_w, img_center_h = img.shape[3]/2, img.shape[2]/2
# Check for traffic lights in front of the car and predict their current state
tl_total, tl_green = 0,0
for t in traffic_lights:
    if t[0] > 200 and t[1] < 850: # ignore traffic lights on the image edges
        cut_img = TLnet.prepare_input(img[0], t)
        cut_img = np.array(cut_img.permute(1, 2, 0).cpu())
        cut_img = torch.from_numpy(cv2.resize(cut_img, (25, 45))) \
        .permute(2, 0, 1).cuda()
        state = torch.argmax(tl_net(cut_img)) # -> 0=ignore, 1=green, 2=red
        if state != 0:
            tl_total += 1
            if state == 1:
                tl_green += 1
if tl_total>0 and tl_green/tl_total < 0.5:
    throttle, brakes, steering_angle = 0,1,0
    send_control(steering_angle, throttle, brakes)
    last_forced_stop = time.time()
    print(f"Red light detected!")
    return
# Check for obstacles in front of the car
for o in obstacles:
    o_center = (o[0]+o[2]/2, o[1]+o[3]/2)
    # checks if the center of a vehicle bbox is in front of our car
    if o_center[1] > img_center_h \
        and o_center[0] > img_center_w - img.shape[3]/18 \
        and o_center[0] < img_center_w + img.shape[3]/18 \
        and o[2] > 50:
            throttle, brakes, steering_angle = 0,1,0
            send_control(steering_angle, throttle, brakes)
            last_forced_stop = time.time()
            print("Car in front detected!")
            return

# No red lights or obstacles detected --> give the control to the model
t, b, s = model(img)[0]
throttle, brakes, steering_angle = t.item(), b.item(), s.item()
print(time.time() - last_forced_stop)
send_control(steering_angle, throttle, brakes)
```