Final Report - Team 12
Simulation-Based Autonomous Driving in Crowded Cities

Ferdinand Duetsch
MS Computer Science
Technical University Munich
ferdinand.duetsch@tum.de

Michael Sodamin
MS Robotic Cognition Intelligence
Technical University Munich
michael.sodamin@tum.de

***Abstract*—**

This report reflects on the development of a control algorithm for the master practical "Simulation-Based Autonomous Driving in Crowded Cities".

It summarizes the current state of research and best practice approaches for self-driving models. It applies them to solve a simulated city scenario within an in-house built unity driving simulator.

The authors implemented a modular control system for lateral and longitudinal control. It consists of an end-to-end steering module, a traffic light, and a crash-avoidance system. The RBG input images are automatically created and augmented before training the imitation learning-based model. Modern visual detection models like YOLO8 and CSP-SG are used and the results are shared across multiple modules.

The control system achieves a decent reliability (86% per intersection) while driving through the simulated city following turn commands.

The project demonstrates how much performance can be reached with little previous knowledge and consumer hardware.

***Index terms*—self-driving, end-to-end, machine-learning, ai, driving simulator**

I. Introduction

In the context of the master practical "Simulation-Based Autonomous Driving in Crowded Cities" (course code: 950691549), we present an algorithm designed to enable autonomous driving of a simulated car through complex urban environments. This involves navigating heavy traffic, crossing intersections, responding to traffic lights, and avoiding collisions with other cars.

The simulator provides real-time RGB images from forward-facing cameras. Information about the position was not available in the simulator version we used (WindowsV25-2). The algorithm's core objective is to predict the appropriate lateral and longitudinal motion of the vehicle, accomplished through steering angle, throttle, and brake control.

The development of our final simulator took place in Unity, in parallel with the algorithm's evolution. Our team supported the development of the simulator through feature implementation, testing, bug reporting, and the building of Windows-compatible versions [1].

The algorithms were designed, trained, and executed on our local machines which are both running Windows. The first system consists of an AMD Ryzen 5800x CPU paired with a Nvidia RTX 3070 GPU. The other system is an AMD Ryzen 7700 processor and an RTX 3090 GPU.

This work adds to the growing body of research on autonomous driving and demonstrates how the latest technologies can be effectively implemented in a functional setting.

II. State of Current Research

A. End-to-end vs Modular

To create autonomous robots in driving situations or other, there are two main approaches. The end-to-end approach as described by Tampuu et al [2] or modular approach, for example, used by the winners of the 2005 DARPA Grand Challenge [3].

B. Sensors Used

In end-to-end approaches like [2] or [4], there are usually multiple input sensors used like stereo camera, LiDAR, and GPS/IMU. This can help in training end-to-end or modular models. However, multiple projects achieved decent results with just single RGB cameras [5], [6]. This also represents the way humans are capable of driving cars (with one eye closed for monocular vision).

C. Learning Models

Models can be trained via imitation learning [7] or reinforcement learning [8]. Both models have their advantages and disadvantages.

1) *Imitation Learning:* Within imitation learning, an "expert" (for example a human driver) is recorded navigating

through various training scenarios. The recorded data is then used for supervised learning. The model tries to predict based on the input what the expert would do in this moment. A downside is that experts rarely make mistakes which would bring the car in bad positions where it has to recover from. The algorithm though could come in those bad positions and it is therefore often hard to learn the recovery from mistakes.

Therefore, an additional method that can improve model performance is expert intervention on noise. During training, a noise input is given so the car deviates from the perfect position on the road. The expert can intervene by steering the car back on the road. With this technique recovery from wrong estimations can be trained. [5]

2) *Reinforcement Learning:* During reinforcement learning, the model learns actively while acting within the simulation. Desired behaviour is rewarded, and undesired behavior is punished. This way of learning is best done in a simulation as doing it in the real world would be very dangerous. [8] Reinforcement learning requires a constantly ongoing loop of action and reaction. It does therefore not allow for batch speed-up like imitation learning does.

D. Training Data

Training data is one of the most important parts of creating robust neural network models and is vital for its success. Training data is collected through real driving [6], model cars [5], or simulators. Successful existing simulators are CARLA and LGSVL. They can provide a multitude of sensors and calibrations, traffic, scalability via multi-client architecture and are open source. [9]

The university chair of our practical course created their own simulator. Together with the supervisor and other students, we assisted in the development and used it to collect data. This made it easy to adapt and customize the simulator to our needs.

To virtually create more training data, images are often augmented [10]. This furthermore helps to improve generalizability and stability. Augmentation can be flipping the image, changing its clarity, contrast, or brightness, blurring or cropping it.

E. Possible model improvements

1) *Adding additional channels:* Models can train faster, achieve higher task performance, and generalize better when adding information like semantic segmentation to model input as a new channel as shown by [11]. Other information to add can be depth information, albedo or optical flow. Whilst simulators can provide this information with perfect precision, during runtime specialized models can try to create this additional information. [2]

2) *HD Maps:* For the implementation of route planning, HD maps are imperative for driving in dense urban environments [2]. Obtaining those HD maps is often costly.

III. Methodology

In this section, we explain our approach to solving the task of creating an autonomous driving controller for the provided simulator. We explain the pre-tasks, our training data collection, and improvement, the data pre-processing, model architecture, and training. We also mention improvements we tried but discarded for the final model.

A. Pre-Task 1: Building a Resnet

To learn useful model structures we first implemented a Resnet50 from scratch using the original paper [12]. The net was trained and tested on the CIFAR10 database [13].

B. Pre-Task 2: Solving the Udacity Challenges

To learn the basics of steering and driving in simulations we started on solving the Udacity Challenge 2017 [14]. The challenge was solved by fully training a Resnet-18 with manual throttle adjustment depending on the steering angle. The input images were pre-processed with greyscale and training images were augmented through flip and blur.

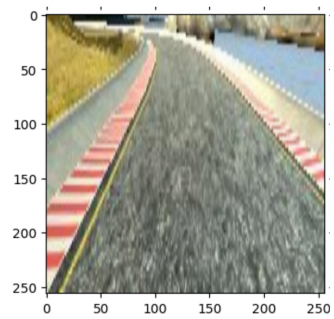

Figure 1: Example Input Task 2

In this phase, we furthermore tested road segmentation, edge detection, and line detection models. As steering prediction we built and tested the Nvidia E2E, Hybridnet, deeplab and Yolop [15] [6] [16] [17][18]. For data augmentation, we also used color, noise, and blocks. [10]

C. Collecting Training Data

We chose to use a supervised learning approach via imitation learning [7], as it is the dominant paradigm [2]. To collect the training data we followed two main approaches, manual and automatic steering.

1) *Manual Data Collection:* The standard way as proposed to collect data is to steer the model manually with keyboard, mouse, or gamepad controls. The researcher drives the car through the simulation and the inputs plus respective video data are recorded. This approach is time-consuming and prone to errors as the steering is often not perfect and de-

layed or includes delays in the start and stopping of the recording. Since we had gamepads available, we collected the manual data using those to have fast-reacting and gradual, non-binary, controls.

2) *Automatic Data Collection:* After all participants gained access to the Unity simulator, we built our own training data collection process. This means we used the internal autopilot system of the simulator to automatically generate the training data we needed. Similar cameras like on the main car were attached to a random traffic car which can steer automatically through waypoints and anti-collision systems. New scripts were created to be able to collect the steering angle and throttle and the images were separately saved. Additionally, we could increase the time of the simulation, to collect even more data in a shorter time frame. In this way, we were able to collect the basis for our main dataset, which consists of about 75,000 images. It roughly represents an hour of driving.

3) *Dataset and Dataset Balancing:*

Figure 2 and Figure 3 show the distribution of steering angles in our data. All steering angles were distributed among 100 bins, which are shown in the histograms. The steering angles are in the range [-1, 1]. The x-axis shows the steering angle range and the y-axis the frequency of the steering angles. Note, that the scale of the y-axis differs.

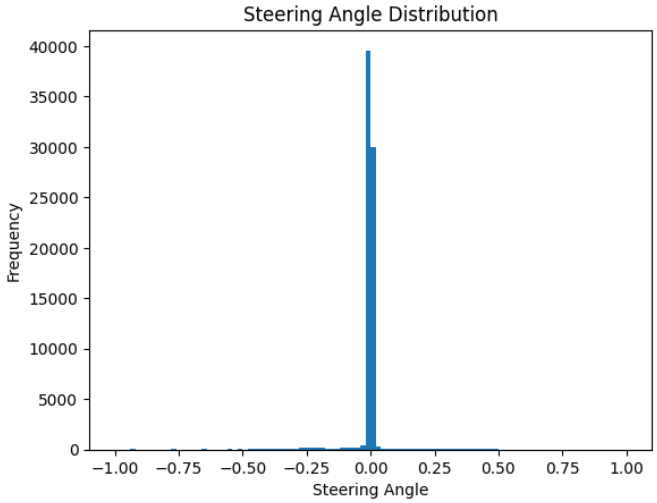

Figure 2: Distribution of steering angles in complete data

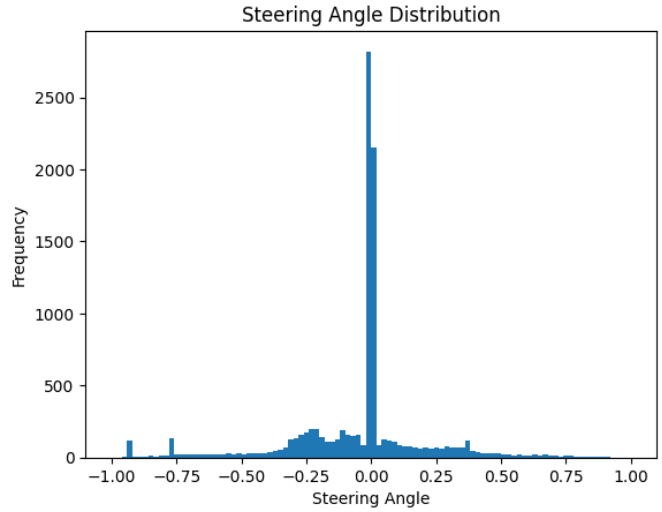

Figure 3: Distribution of steering angles in balanced data

Figure 2 shows the steering angle distribution of the complete dataset. It also shows that the number of steering angles with a value near 0 is dominating the dataset. Therefore, models trained on the complete data struggle to make turns and often output a very low steering angle. To balance the dataset and make the trained model output larger steering angles, we randomly removed data points with a steering angle near 0. In total, our dataset was reduced to about 10,000 entries.

Figure 2 shows the steering angle distribution after balancing. In Figure 3, it may seem that steering angles very close to 0 are still dominating the dataset. However, they now only make up about 50% of the data. Balancing the dataset highly improved the performance of models trained on it and higher steering angle outputs were now much more common.

4) *Hybrid Data Collection:* The models trained on the balanced dataset still struggled in some traffic scenarios. Often they were not able to recover to the middle of the lane when the car was driving on the very edge of the lane or the pavement. Also, the models still struggled to make turns in some cases and output a too-small steering angle. To solve this problem, we extended the set of balanced automated training data with manually recorded data. To teach the model how to recover to the middle of the lane, we collected data while driving straight on the pavement. After collecting this "bad" data, we added a steering angle bias in the direction of the middle of the lane to the data entry. To make the model more robust for making turns, we manually recorded more natural data of us taking turns with a gamepad.

5) *Three-image Training Approach:* To be able to create a robust steering model, we used three images as suggested by [5] and [6]. The three possible image variants can be seen in Figure 4. When loading an image during training, it is randomly selected, which image of the three possible cam-

era setups is used. The left and right images then receive a steering bias of 7.5 and -7.5 degrees. This is a very important parameter and different values were tried.

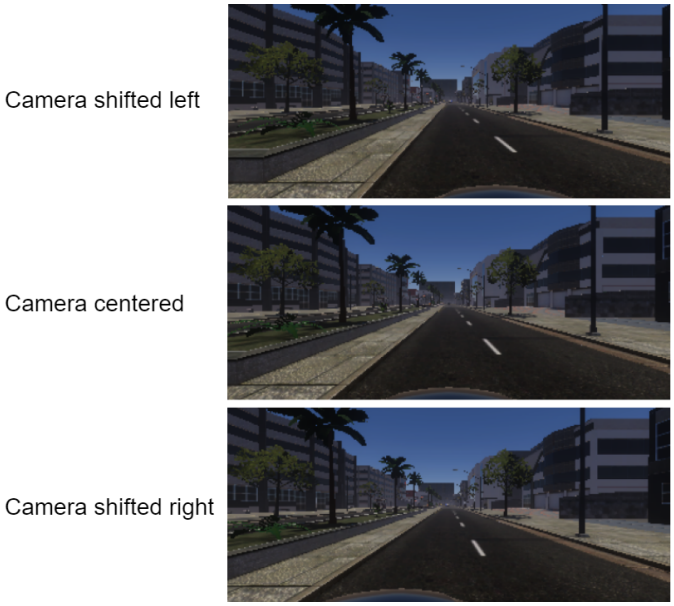

Figure 4: Possible Camera Setups

6) *Data Preprocessing:* All images are preprocessed before inputting them into the model. This is not the data augmentation. The preprocessing pipeline is applied to all images during training, validation, or testing time. It consists of the following steps:

1. Convert to grayscale
2. Crop images to relevant content
3. Resize images to 256x256

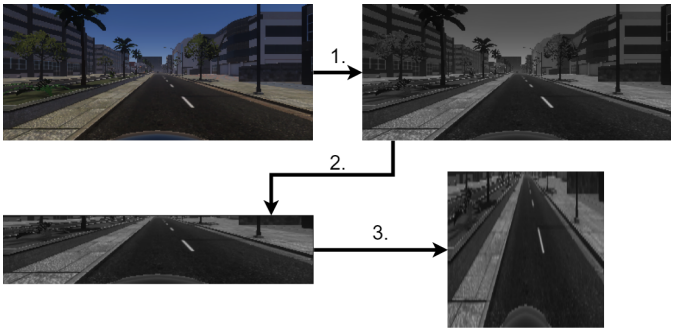

Figure 5: Preprocessing Pipeline

7) *Data augmentation:* The performance of large neural models relies heavily on their training data. Specifically, the problem of overfitting needs to be countered. Good data augmentation is an important measure to prevent overfitting by creating, additional viable data that can be used to generalize the understanding of a situation for the model. [10]

We applied various data augmentation techniques, to decrease the amount of overfitting. In contrast to our data processing step, the data augmentations were only applied during training time. The data augmentation pipeline consists of the following steps. For each step, the probability is given at the end.

1. Random *horizontal flip* + *sign flip* of steering angle (*p=0.5*)
2. Add *gaussian noise* (*p=0.25*)
3. Add *gaussian blur* (*p=0.25*)

Figure 6 shows two example results of the same image after being input to the augmentation pipeline. The *horizontal flip* (1.) and *Gaussian blur* (3.) steps were applied to the left image. To the right image, only the *Gaussian noise* stage (2.) was executed.

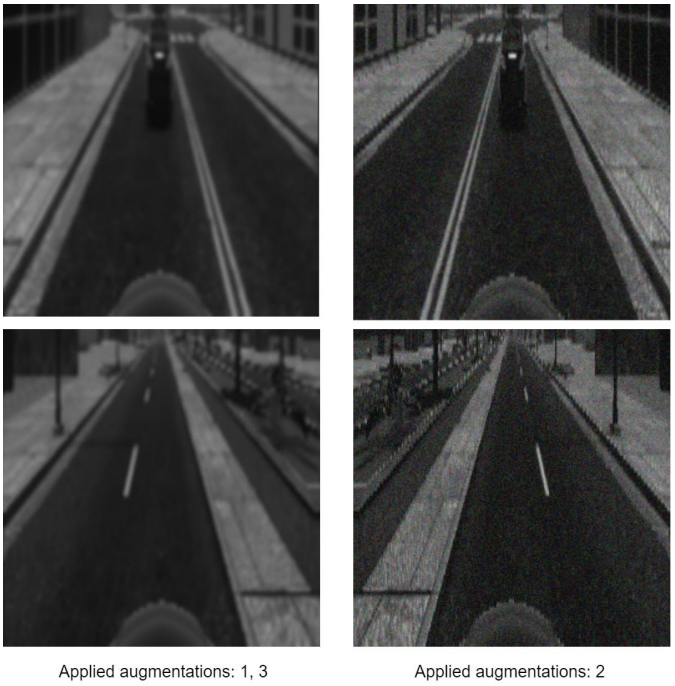

Figure 6: Augmented Image Examples

We empirically tested multiple variations of the data augmentation pipeline. This included the addition of augmentation steps such as *random crop*, *color jitter*, or *random erasing*. The configuration above seemed to work the best for us. More information on these data augmentation methods can be found in the PyTorch documentation [19].

D. Simulator Control

Figure 7 shows an overview of our simulator control software architecture. It shows, that our software is divided into two parts, the control output computation and the detection computation. Each part is handled asynchronously by its own thread. The detection thread is responsible for traffic light, crash / distance to the next vehicle, and intersection detection. The control output thread is responsible for computing the control output which consists of the steering angle, throttle, and brake. It uses the preprocessed image

and the model to compute the steering angle. The steering angle and the results from the detection thread are then used to compute the final control output.

Additionally, we receive an HTTP request containing the current image multiple times a second. This is not shown in Figure 7. We immediately respond with the previously computed control output containing the throttle, brake, and steering angle values that should be applied. Also, the current image is updated. The advantage of this approach is, that we can minimize the delay in our response and the simulator can run smoothly and reliably. Our hardware allows us to recompute the control output and traffic light detection results with a high frequency. Therefore, even though the computed control output in the response is most likely based on a previous image, it is recent enough to not decrease our performance.

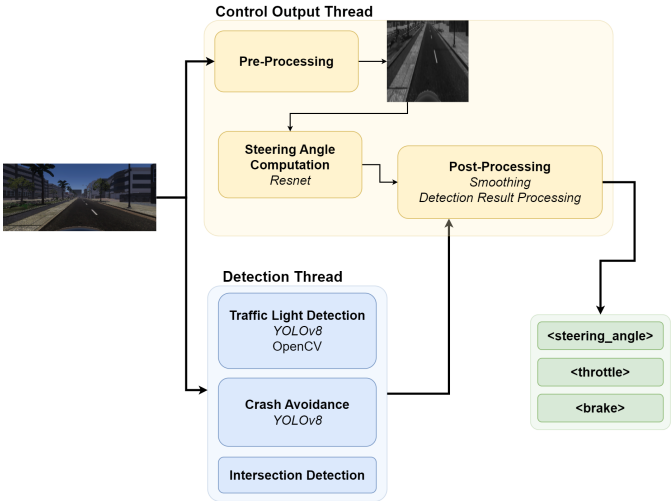

Figure 7: Final Software Architecture

1) *Steering Angle Prediction:*

Model: Whilst the overall controller is a modular approach, for the steering angle prediction itself, we use an end-to-end approach. We input the live camera image into the model, which then outputs a steering angle.

We used a similar model which was very successful in the pre-work of task 2 (Section III.B). As the model architecture in Figure 8 shows, we created a model with 3 main parts, where a Resnet-18 (without the fully connected layers) represented the core. To match the required input size we used an upsampling layer first (torch.nn.upsample, with default configuration). Three fully connected layers were then custom-trained by us to predict the steering angle. Between the three fully connected layers, a ReLu activation function was used (not displayed in Figure 8).

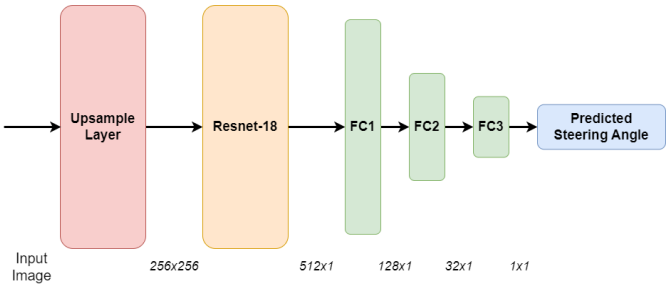

Figure 8: Steering Model Architecture

Training: The model was trained on our GPUs and training was supervised using Tensorboard. For the reading and augmentation of our input data, we customized the torch dataloader. Our final model used batch size 64 and a steering bias of 7.5 degrees.

2) *Traffic light classification:* For the traffic light detection, we use a multi-stage approach. The low resolution of 320x160 pixels makes reliable traffic light detection hard and irrelevant traffic lights with wrong color indications for the current way to drive are in view.

Figure 9: Traffic light scenario

Step 1: Detection: We used the detection information of the YOLO8 model [20], this is also reused in the car distance calculator. YOLO8 is a fast and reliable best-practice model for object recognition. We used the *YOLO8m-seg* model unrefined from the official website.

Figure 10: Detected traffic lights

Step 2: Selection and color detection: When approaching the intersection most of the traffic lights are detected. However, only the middle, frontal-facing ones (and not pedestrian ones) are relevant for the car to decide whether it should stop or continue to drive. We saw two ways to tackle this problem: Training a model that can identify the direction and color of traffic lights [21], or a more straightforward approach by only using traffic lights that are roughly in the horizontal middle of the image and using classical computer vision to extract the dominant color. We went with the second approach and implemented everything from scratch.

Figure 11: Cropped Traffic Light

After selecting the traffic lights that are within the inner third of the image, an HSV channel separation mask is created through OpenCV [22] using experimentally selected values for green, yellow, and red. These values are not expected to change within the simulation.

For all masks, the amount of in-masked pixels is compared and the most common one is chosen as predicted output. If the amount of most common pixels is too low, we do not detect the traffic light as one. This is done to prevent the detection of traffic lights, which are far away.

Step 3: Stopping command: If a red light is detected in front of the car it stops immediately with full brake inputs until the light becomes green again. This is an overwriting signal.

3) *Sub-Module: Traffic Distance Detection:* In order to not crash into the other traffic participants, a traffic distance detection was implemented, which computes the distance to the next car. If the distance to the next car is too low, our car should stop.

Distance Computation: The detection information of the YOLO8 model [20] which is used for the traffic light detection, can also be used for the computation of the distance. The YOLO8 model detects multiple classes at the same time, such as cars, trucks, buses, or traffic lights. For the distance computation, we only consider the detections of other vehicles which are approximately in the center of the image. The higher the y-coordinates of the detected bounding box (edges) of a vehicle, the closer the vehicle is to our own car. We compute the distance to the closest car, by computing the maximum y-coordinate of the bounding boxes.

Braking Thresholds: We empirically selected two distance thresholds. The higher one determines, at which distance we should set our throttle value to 0 and slightly start braking. The lower one determines the emergency braking threshold. If the distance to the next car is below this threshold, full braking is applied. The following figure illustrates the detection and thresholds.

Figure 12: Distance Computation Illustration

4) *Control Output Computation:* As stated above, we obtain the steering angle by our trained steering model. Since we do not receive the current velocity in the simulator version we used, we initially always use a throttle value of 80%. We then run a smoothing across our past throttle and braking values. We always use the lowest of our past 5 throttle values and the highest of our past 3 braking values. This is done to enable more careful driving. If we stand still in front of a red traffic light, we don't want a single wrong prediction to affect our result. We do not smooth the steering angle values, since the model should be able to quickly react. After having our control values smoothed, we brake according to the distance to the next vehicle as described above. For very high steering values, however, we do brake in case of a close vehicle. This is because only vehicles in front of us are detected. Therefore, if we make sharp turns, we will most likely not hit them. As a last step, we run the traffic light detection. If we detect a red light, we stop. There is only one specific situation in which we don't want to stop in front of a red traffic

light. If we want to turn right or left at an intersection, our program may see the traffic light of the other lane after making part of the turn. This light is most likely red since we just had a green light. This situation will lead to our car stopping in the middle of the intersection. This situation is tackled by checking whether there was a yellow light in between the transition from the green traffic light to the red one. If not, then we conclude that this situation occurred and we continue driving. The transition from green to red must have happened within a second to reduce the false positive detections of this situation.

5) *Navigation:* Our approach at the beginning of the project was, to train a single end-to-end model based on a high amount of driving data. We managed to train such a model which is able to keep the lane on the straight and in curves. Situations in which the model was performing weak, were reduced by manually collecting additional training data (described in Section III.C.4). However, there was one major flaw with this approach. The training data did include three different behaviors at intersections: Driving *left, straight,* and *right*. During the training process, the ground truth steering angle at an intersection could point in any of the three directions. This led to an undetermined behavior at testing time. The car chose any of the three directions at random and was struggling to handle intersections at all. It was possible that steering angle predictions of subsequent frames point in opposite directions since there was no input to the model to encode in which direction it should drive.

We tackle this problem, by training four different models. One *general model*, for all situations that are not at an intersection and three models for each possible direction of an intersection. The idea is to switch between the models to predict a steering angle, depending on the intended behavior as described as more beneficial than inputting the turn command into one model by Codevilla et al [5].

For example, if it is desired to turn left at an intersection, the *left turn* model is selected until the intersection is finished. The behavior should be given prior to the execution of the code.

Training Specific Intentions: The *general model* is trained using our balanced dataset as described at the beginning of this chapter. For the others, we created three new training datasets by filtering the balanced dataset. The model which should simply drive *straight* at an intersection (*no-turn model*) was only trained on data, that had relatively small absolute steering angle values. Therefore, the model only learns to make small corrections and to not make any sharp turns. The *left* and *right turn models* are trained on left and right turn data mixed with some *straight* line data such that the models also learn to drive *straight* (e.g. after the intersection). In addition to filtering out steering angles in the wrong direction, a small random subset of *straight* line data elements was kept in the respective datasets. This was done for two reasons.

1. The models did only output a relatively small steering angle, due to the imbalance of the dataset (much more training data elements with a steering angle of 0).
2. Reduce the occurrence of situations in which the car drove *straight* instead of turning at an intersection.

In the following, the training data of the *right turn model* is used as an example, of how the training data differs between the different models. Figure 3 shows the distribution of steering angles in the balanced dataset. The *general model* is trained on the balanced dataset. Figure 13 shows the distribution of steering angles in the training data of the *right turn model*. As introduced in Section III.C.3, all steering angles were distributed among 100 bins, which are shown in the histograms. The steering angles are in the range [-1, 1]. The x-axis shows the steering angle range and the y-axis the frequency of the steering angles. Also here, the scale of the y-axis differs. When comparing the two histograms, it shows, that for the *right-turn model* all data turning left was filtered out. As stated before, many data entries with a steering angle of 0 are filtered out. Due to the strict filtering, the length of the dataset was reduced from about 10,000 entries to about 2,000.

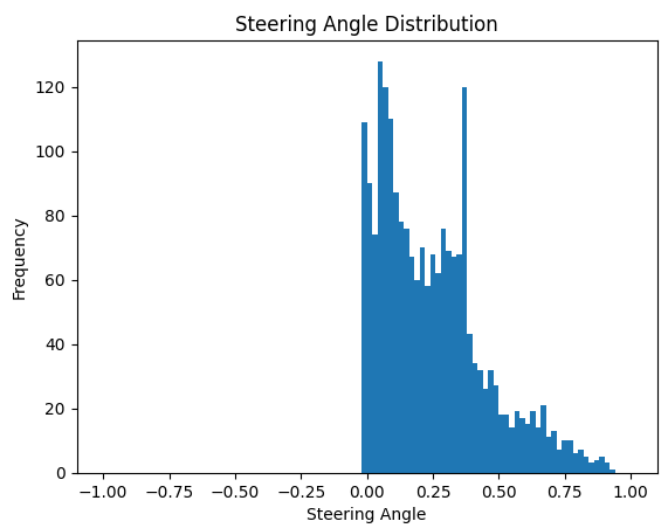

Figure 13: Right turn model: Distribution of steering angles in training data

Detecting Intersections: For the navigation approach to work, it is essential to reliably detect when a new intersection starts and finishes. Otherwise, the wrong model may be selected at an intersection and the navigation goal won't be reached. To detect an intersection, the traffic light detection output can be used. As soon as a traffic light is detected, it is assumed that the car is currently at an intersection. Afterwards, if

there is no detected traffic light for two seconds then it is assumed that the intersection has ended. This method is used to count the intersections that were already encountered.

Navigation Implementation: The direction in which the car should be driven can be given by a navigation configuration. This navigation configuration consists of a list of directions that is given prior to the execution of the program. The possible values are *left*, *straight*, and *right*. An example is the list *left, left, straight*. At the first two intersections, the *left turn model* is selected. At the third turn, the *no-turn model* is selected. The models are selected by counting the encountered intersections, as previously described. After all navigation commands are executed, it is assumed that the car has arrived at its goal and the program stops the car.

E. Discarded Approaches

1) *Adding Segmentation Model Output as Channel for Steering Model Input:*

As shown by [11], adding segmentation information in addition to an input image can greatly improve model performance. We, therefore, experimented with adding the CSP-SG model by Dr Zhou, which we trained as intended by the author but with a reduced time of 50 epochs. [23]

The result of this model was concatenated as an additional channel to the input image. The recognized segments were flattened to one channel by choosing the most likely predicted output.

Whilst this technique helped the controller to stay on the road, it did not help in differentiating between lanes (as lines on the road lost significance to to model). Therefore, the car ended up driving in the middle of the road instead of the middle of a lane which led to more crashes with other traffic participants. This, paired with the significant additional computational overhead led us to not include the technique in the final model.

2) *Using Time Series Models:* As detecting a save path to drive is a continuous, timebound action we experimented with LSTM layers for our steering prediction model. For this, we used the CNN-LSTM architecture from [24]. An LSTM layer with input size 512 and hidden size 128 was used in combination with a pre-trained encoding part of a ResNet. In a forward step, the 4 most recent images were used as input. For each image, a 128x1 vector was computed using the decoding part of the ResNet. These encodings were used as input for the LSTM. The output of the LSTM was then fed into two fully-connected layers with a ReLU activation function in between. The LSTM approach did only perform the same or slightly better than the direct approach with simple output smoothing. However, there would be a lot of work necessary to make it usable with our hybrid data collection approach. This is because in our dataset it is not guaranteed, that consecutive images are temporal dependent (since datasets were merged and datapoints were filtered). Due to the added complexity and the low-performance increase, we discarded this approach.

IV. Experimental Results

A. Traffic light detection

The detection of traffic lights worked just about well enough from the beginning on. The resolution only allows for stopping just before or right on the crosswalk.

Figure 14: Stopped at Traffic Light

No major adjustments were needed.

B. Evaluating the Complete Pipeline

Our system demonstrates good performance on a high level when executing autonomous mode with just turn signals as input. Figure 15 shows an exemplary execution of a right turn.

Figure 15: Example Turn Performance

In order to evaluate the developed software, the program was executed with all possible routes with a length of two intersections. Since the routes *left, straight* and *right, straight* do not exist (*left* and *right* are ending in T-junctions), these were not possible to test. Three runs were executed for each of the seven possible routes, which led to a total number of attempts of 21. If the car fails to complete the first intersection the attempt is considered as *FAILED*. If it only managed to complete the first intersection, the result of the attempt is still considered as *FAILED*. To distinguish between the first and the second case, the color of the second result is set to orange. If both intersections were completed successfully, the attempt is considered as *PASSED*. A *PASSED* attempt does not mean, that there were no traffic violations. The metric

for this test was only whether the goal position was reached and no wrong turns were made at intersections.

Navigations	Run 1	Run 2	Run 3
left, left	*PASSED*	*PASSED*	*FAILED*
left, right	*PASSED*	*PASSED*	*PASSED*
straight, left	*PASSED*	*PASSED*	*FAILED*
straight, straight	*FAILED*	*FAILED*	*FAILED*
straight, right	*PASSED*	*FAILED*	*PASSED*
right, left	*PASSED*	*PASSED*	*PASSED*
right, right	*PASSED*	*PASSED*	*PASSED*

As seen in Table 1, 15 out of the 21 attempts were successful and the goal region was successfully reached. In five attempts, only the first intersection was completed successfully and in one attempt, the program failed to complete the first intersection. The reasons for failed attempts were the following:

- Crashed into cars in front (1x)
- Drove of Lane (2x)
- Wrong intersection detection → wrong navigation (2x)

As for a per-intersection analysis, this means that out of 42 intersections, 6 were missed. This is an 86% success rate on intersections including the need to be able to follow a lane, brake, and not crash into cars.

C. Computational Performance

The system was timed on test performance running parallel with the simulation on the AMD Ryzen 7700 processor and an RTX 3090 GPU in Windows.

The computation of one cycle of the control output thread (see Figure 7) takes under 10ms and one cycle of the detection loop (see Figure 7) including traffic light control and traffic distance control takes under 50ms.

V. Discussion

In this report, we demonstrated the creation of a moderately well-working self-driving algorithm in a short time and with little prior knowledge. The created control system can handle driving on roads, taking defined turns at intersections, obeying traffic lights, and avoiding simple traffic collisions.

This work shows that even with consumer hardware and little knowledge, decent results can be achieved. The report can function as a positive reason for hope and guidance for anyone who wants to step into the field of autonomous driving.

We showed that existing algorithms like YOLO8 or the CSP-SG model can be used to accelerate the model creation. Creative approaches for data collection were shown and best practices for data augmentation and cleaning were carried together and applied.

The solved scenario only includes basic driving scenarios in one city with rules and best practices obeying traffic participants. The system is not trained or ready for deviations from this ideal scenario. See Section VI for possible extensions.

A. Findings and Learnings

Some findings and learning we want to highlight for fellow self-driving car engineers are the following:

1. Recent advancements in consumer graphics cards and compute power and easy access to online tools like Google Collab Notebooks allow anyone to build basic but decently working self-driving car algorithms.
2. Working within a simulation allows for great flexibility and development speed. Even better if one can build and expand on the capabilities. Using our code to automatically record great training data saved us a lot of time.
3. There is a huge amount of research available, even though there is a lot in the form of YouTube videos and blog posts. Also reading the original papers can help a lot in understanding the existing research better.
4. Monitoring and documenting training progress with Tensorboard can help strategically improve settings and hyperparameters. One should add all possible hyperparameters (also e.g. steering bias or created, augmented images) in this logging system
5. Modular approaches allow to reuse of existing algorithms, especially for detection and segmentation, easily.
6. Working in a team can help motivate and get unstuck in challenging questions and situations.

VI. Outlook

A. Navigation

One missing element of our navigation implementation is, that the user has to specify the goal by providing turns which should be made at an intersection. Ideally, the user only has to specify a region on a map, and the program automatically drives there. In practice, this could be imple-

mented by a global planner which finds a route from start to finish by using the Dijkstra or A*-algorithm. Its output would be the sequence of directions which is then input as the navigation commands to our current program.

To be able to work on this we would need to get additional information from the simulator.

- The current position
- The desired possible positions (e.g. street names, intersection names)
- A map of the environment.

B. Robustness Improvements

Whilst 86% of correctly taken turns is a good achievement for the baseline, it does not even come close to the required accuracy of what a self-driving system should provide.

Reliability must be improved for sure.

Some possible approaches could be:

- More training data, also more diverse with different cities and scenarios
- Improving on including the CSP-SG model
- Fine-tuning of learning parameters
- Attention mechanisms and leveraging time series again after tweaking other parameters

C. More difficult scenarios

Currently, the simulation only represents the most basic driving scenario of following one lane without lane changes, obeying traffic lights and not crashing into other vehicles.

Possible important additions to these scenarios and tasks could be

- Pedestrians
- Lane changes
- Additional traffic signs (stop, yield, ...)
- Bikes
- Other cars behaving non-conformant (sudden lane changes or stops)
- Different weather conditions
- Roadworks
- Different city layouts and designs

References

[1] Y. G. Z. Y. M. S. H. L. L. M. L. L. H. L. Y. L. H. L. G. C. A. K. Liguo Zhou* Yinglei Song*, "Garchingsim: an autonomous driving simulator with photorealistic scenes and minimalist workflow," *Unreleased*, 2023.

[2] A. Tampuu, T. Matiisen, M. Semikin, D. Fishman, and N. Muhammad, "A survey of end-to-end driving: architectures and training methods," *IEEE Trans. Neural Networks Learn. Syst.*, vol. 33, no. 4, pp. 1364–1384, 2022, doi: 10.1109/TNNLS.2020.3043505.

[3] S. Thrun, M. Montemerlo, et al., "Stanley: the robot that won the darpa grand challenge," *J. Field Robot.*, vol. 23, no. 9, pp. 661–692, 2006, doi: 10.1002/rob.20147.

[4] D. Coelho, and M. Oliveira, "A review of end-to-end autonomous driving in urban environments," doi: 10.1109/ACCESS.2022.3192019.

[5] F. Codevilla, M. Müller, A. López, V. Koltun, and A. Dosovitskiy, "End-to-end driving via conditional imitation learning," 2017. [Online]. Available: http://arxiv.org/abs/1710.02410

[6] M. Bojarski, D. D. Testa, et al., "End to end learning for self-driving cars," 2016. [Online]. Available: http://arxiv.org/abs/1604.07316

[7] B. D. Argall, S. Chernova, M. Veloso, and B. Browning, "A survey of robot learning from demonstration," *Robot. Auton. Syst.*, vol. 57, no. 5, pp. 469–483, 2009, doi: 10.1016/J.ROBOT.2008.10.024.

[8] R. Sutton, "Introduction to reinforcement learning r a i l &."

[9] P. Kaur, S. Taghavi, Z. Tian, and W. Shi, "A survey on simulators for testing self-driving cars," 2021, pp. 62–70, doi: 10.1109/MetroCAD51599.2021.00018.

[10] C. Shorten, and T. M. Khoshgoftaar, "A survey on image data augmentation for deep learning," 2019, doi: 10.1186/s40537-019-0197-0. [Online]. Available: https://doi.org/10.1186/s40537-019-0197-0

[11] B. Zhou, P. Krähenbühl, and V. Koltun, "Does computer vision matter for action?," *Sci. Robot.*, vol. 4, no. 30, 2019, doi: 10.1126/scirobotics.aaw6661.

[12] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," 2015. [Online]. Available: http://arxiv.org/abs/1512.03385

[13] A. Krizhevsky, "Learning multiple layers of features from tiny images," 2009.

[14] Udacity, "Udacity's self-driving car simulator," 2017.

[15] X. Dai, "Hybridnet: a fast vehicle detection system for autonomous driving," *Signal Processing: Image Communication*, vol. 70, pp. 79–88, 2019, doi: https://doi.org/10.1016/j.image.2018.09.002. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0923596518303771

[16] L.-C. Chen, G. Papandreou, I. Kokkinos, K. Murphy, and A. L. Yuille, "Deeplab: semantic image segmentation with deep convolutional nets, atrous convolution, and fully connected crfs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 40, no. 4, pp. 834–848, 2018, doi: 10.1109/TPAMI.2017.2699184.

[17] V. Badrinarayanan, A. Kendall, and R. Cipolla, "Segnet: a deep convolutional encoder-decoder architecture for image segmentation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 12, pp. 2481–2495, 2017, doi: 10.1109/TPAMI.2016.2644615.

[18] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: unified, real-time object detection." [Online]. Available: http://pjreddie.com/yolo/

[19] Pytorch, "Transforming and augmenting images," 2017. [Online]. Available: https://pytorch.org/vision/stable/transforms.html

[20] J. R. Terven, and D. M. Cordova-Esparaza, "A comprehensive review of yolo: from yolov1 and beyond under review in acm computing surveys," 2023.

[21] automaticaddison, "How to detect and classify traffic lights – automatic addison." [Online]. Available: https://automaticaddison.com/how-to-detect-and-classify-traffic-lights/

[22] G. Bradski, "Opencv," 2010.

[23] L. Liu, L. Zhou, Z. Bing, R. Wang, and A. Knoll, "Real-time semantic segmentation in traffic scene based on cross stage partial block," 2022. [Online]. Available: https://github.com/zhouliguo/cspsg.

[24] P. Radhakrishnan, "Cnn-lstm," 2022. [Online]. Available: https://github.com/pranoyr/cnn-lstm