Safety Layer for Autonomous Driving with Donkeycar

Yiğit İlk, Roman Mishchuk, Liam Paul Brandt,
Patrik Valach, Michał Chrzanowski, Paul Hagner
Technical University of Munich, Heilbronn, Germany
{yigit.ilk, roman.mishchuk, liam.brandt, patrik.valach, michal.chrzanowski, paul.hagner}@tum.de

Abstract

Our group created this project along with this report for the Cyber-Physical Systems and Robotics SS2024 course at TUM Heilbronn. The goal was to create an autonomously driving vehicle whose priority focus is precision and safety. Therefore, we procured a robotic car equipped with advanced machine-learning capabilities.

On top of this existing framework, we implemented an additional safety layer, utilizing internal car sensors to ensure the project's focus is achieved. For that, a virtual environment was created to keep track and ensure that the car drives only where it is allowed to (legal zones). This virtual environment is a replica of a real-world road the car drives on, allowing it to alter the path dictated by the algorithm in case it is going to end up in the non-legal zone. Such an approach ensures that the machine learning algorithm does not pose a risk to itself or anything else on its track during autonomous driving.

Contents

List of Figures

Listings

1. Introduction

Machine Learning is a rapidly developing and ever more popular way to solve the autonomous driving problem. Yet, due to the nature of this approach, the resulting driving plan is never perfect and may often be dangerous to execute. This poses a significant challenge when deploying such systems.

That made us curious to test out the methods of making the machine-learning approach safer for use in real-world complex environments. Hence, we set the following goal for the project: *given a trained machine-learning (later ML) model capable of navigating the environment with some mistakes (entering illegal zones), construct an algorithm that would predict and correct the mistakes of the model before they are actually executed, ensuring the prevention of any and all blunders from happening.*

For the reasons explained later, we decided to work on autonomous car driving in a static environment. So, our main focus is not on the smoothness of driving, nor do we set a defined path for the car to follow or to race in any manner. But, the goal is for us to put a car in its starting position and then have it navigate the static environment safely on its own without colliding with any object or crossing into any illegal areas on the track.

Overall, we aim to provide a framework for other systems that use static environments (or even non-static ones, with the possible addition of LIDAR, which is not a part of our initial approach) and need safety in automated moving cars/robots. For example,

in the case of a warehouse, the layouts of the floor are always the same regardless of any other changes (except accidental drops of carried cargo, etc.). As we can have predefined places where we put products/items/shelves, this is exactly the safety problem our framework is capable of tackling.

2. Literature Review

This project draws upon the methodologies described in the paper "Safe Reinforcement Learning Using Black-Box Reachability Analysis" (Alanwar A. et al., 2022) [1]. In it, the following method of solving the safety problem is described: during each step of the receding-horizon, the Reinforcement Learning agent perceives the environment via a camera and/or LIDAR input and then generates a sequence of actions through trajectory rollout. This sequence is then validated for safety using reachability analysis by checking if the predicted trajectory destination intersects with an illegal zone. If that is the case, the trajectory is adjusted using gradient descent, and failsafe maneuvers are executed if necessary.

It should also be mentioned that while working on Implementation, we have extensively used guides and recommendations provided on the Wiki of the JetRacer platform [2], Wiki of the Robotic Operating System (later ROS) that the JetRacer uses [3]. Apart from that, we have used the DonkeyCar [4] project as a base for our ML model.

3. Methodology

– Safety Layer Framework –

We have adapted the approach described in [1] by taking into account the features our setting has:

- As the car is not using any kind of manipulator and only drives on a flat 2D map, our state space has a dimensionality of 2 and can be viewed as $\mathbb{R}^2$. This would allow us to simplify the Reachability Analysis during the Implementation.
- As it is not a part of our focus, we have used the Machine Learning approach instead of the Reinforcement Learning described in the paper. Due to the Safety Layer being merely a wrapper over the driving model, these two approaches end up being interchangeable, allowing us to simplify Implementation without affecting our objective.

Due to those features, our framework involves collecting data before a model is deployed for data-driven credibility analysis and using this data while the model is running to guarantee safety.

In the following subsections, we describe the methodologies of individual parts of the project as well as the requirements for them.

3.1. Track Requirements

The purpose of our project is to work in a complex and diverse environment. The track should contain multiple obstacles on the road of different sizes. The road should have sharp turns to introduce a more complex environment in which the ML model is more likely to be pushed to the limits in means of safety. This is all done with the purpose of visualizing as many approaches a car can make in our limited track in order to make sure that in a larger environment, there would be smaller possibilities of harder-to-navigate areas.

3.2. Hardware Platform Requirements

As we want other people to easily deploy and extend upon our implementation while being able to easily modify it for their own use cases, we decided to use a commercially available hardware platform to implement our project. Also, using a platform based on open-source software goes a long way in making our research available to as many different parties as possible and promotes further advancements within this codebase and research.

The hardware should be capable of driving and steering. It should possess visual input, such as a camera, to have input for an ML model. We also require it to have sensors that are capable of precisely keeping the position of the hardware within the predefined track.

3.3. Machine Learning Model

We decided to train our own ML model that would be tailored to be able to drive around a track autonomously. The model should be able to drive on any normal track without following any preset destination routes. It does not need to be perfect in means of safely navigating around the track and we expect it to make mistakes to better mimic other possible models out there that lack safety on their own.

3.4. Observing model performance and finding weak spots

We are interested in finding the areas in which the model struggles the most. The key indicators are hitting an obstacle and/or driving over the track lines, which can all be changed while deploying the codebase to the system. For the growth of this project, we want to be able to find any patterns between those hard-to-maneuver areas and check for possible ways to overcome the issue either with extra training over time or with our safety layer providing new moves that the model itself cannot do.

3.5. Developing safety layer logic

We want to create a virtual environment that represents the real world. The car will check against this environment by using its internal sensors to determine if it is about to collide with something.

In case the model is leading the car into a collision, it should intercept this command and override it with recovery logic. This recovery logic can be implemented such that it reverses the prior outputs of the ML model

with added randomness to the movement in order for the model to try again from a different position, which is a simple logical chain of movements that can be modified in different ways suiting the needs of different environments.

Eventually we expect the safety layer to guide the car by helping it skip this obstacle the model has failed to avoid. During this recovery logic, it is paramount that we keep checking for collisions in order to not create another incident in the process of recovery.

3.6. Evaluating safety layer performance improvement

The gathered data can then be used to compare the driving performance of the car with and without the safety layer by observing the behavior of the car in critical locations of our track where the model has failed before. It will be clearly observable when our recovery logic gets triggered in practice and if it needs more fine-tuning to recover the car from these critical zones safely, allowing us to test different recovery logics.

4. Implementation

4.1. Building the Track

We have created our track (Figure 1) according to the requirements mentioned in Methodology (Track Requirements). We have used boxes of various sizes as well as another car to create our desired environment. The layout requires the model to perform several sharp corners as well as gives it flexibility on how to drive (multiple roads to follow up). For example the model is able to pick a direction without stopping around the biggest box as a roundabout.

The usage of the taped illegal zones was important, as it allowed us to introduce the sharp turns to the track. The boxes also serve another purpose: making sure that the ML model does not overfit on the tape.

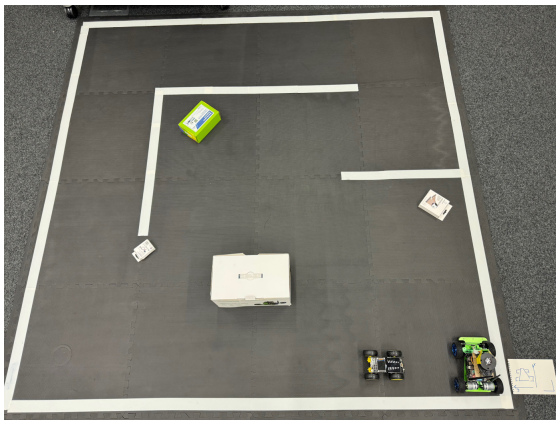

Figure 1: The track that is used

4.2. Hardware Platform

For our project, we have decided to use the "JetRacer ROS kit" from the brand Waveshare [1]. This platform is used to develop and run our implementation of the safety layer.

It has the following components:

- `Raspberry Pi RP2040 Chip` for handling all external sensors' connection with the main Nvidia Jetson Nano Board.
- `37-520 Metal Encoder Motor` for handling movement and tracking tire rotations.
- `IMU sensor MPU9250` for tracking physical motion around the track. Here IMU stands for Inertial measurement unit.
- `LIDAR RPLIDAR A1` for 3D mapping of tall enough objects, which is omitted for our purposes.

We have chosen this kit because it matches our criteria for developing the safety layer, as it contains all the necessary parts to create and run a machine-learning model whilst also containing an IMU unit that will allow us to feed our safety layer with sufficient data to keep track the position of the agent in the state space (location of the car on the track).

Out of the box, it comes with a camera that we have determined to be mounted too close to the ground and without enough vantage points. Despite having a fish-eye lens, its low resolution did not allow it to see corners very well. Therefore, we have decided to mount the camera on the top of the kit on the LIDAR sensor, as could be seen on Figure 2. Due to the nature of our track, LIDAR could not detect the obstacles, therefore, using it as a mount did not hurt our implementation. We decided to exclude LIDAR data in our model and went with only camera inputs.

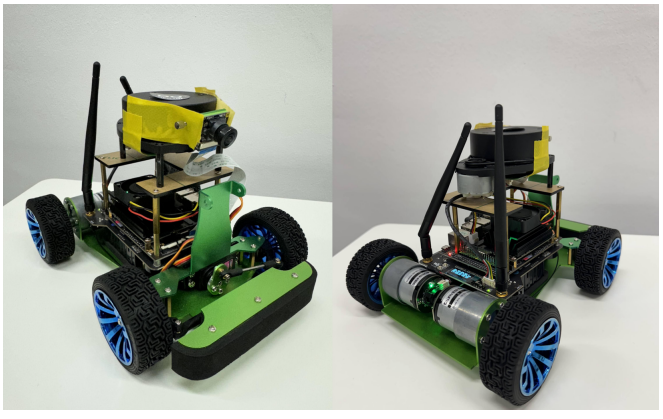

Figure 2: The model car platform: "JetRacer ROS kit"

4.3. Machine Learning Model

The development of our robotic car involved a multi-faced approach, integrating both hardware and

1. https://www.waveshare.com/product/jetracer-ros-ai-kit.htm

software components to achieve autonomous navigation. The very first step in our project was to select and configure the robotic car kit, which included various pre-designed libraries for machine-learning applications.

To train our model, we used the DonkeyCar platform [4] in order to collect the camera input and the corresponding moves the car should do, which were provided using joystick inputs. Using the chosen library, we manually generated training data by driving the car on the track, ensuring accuracy and correcting any deviations by backing up when necessary. This process resulted in a dataset of approximately 11,000 records. We found that further increasing the number of records would lead to impractically long training times without improvements on the model driving capabilities, overfitting the model at 15,000 records. It is further simply impossible to conduct the training when a history of >3 previous states is used for recurrent models, as our RAM limit of 32GB is not enough on an x86 machine.

After extensive testing, we determined that the RNN (Recurrent Neural Network) LSTM (Long short-term memory) model provided the best performance for our needs as it has a memory of 3 previous frames, allowing it to navigate quite well in most cases. Its input/output structure is provided in Listing 1.

Listing 1: ML model structure

```
Model: "lstm"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 img_in (InputLayer)         [(None, 3, 120, 160, 3)]  0

 time_distributed            (None, 3, 58, 78, 24)     1824
 (TimeDistributed)

 ...

 dense_3 (Dense)             (None, 10)                650

 model_outputs (Dense)       (None, 2)                 22

=================================================================
Total params: 441,924
Trainable params: 441,924
Non-trainable params: 0
_________________________________________________________________
```

Our ML model controls both the thrust and turning of the car. The latter is designed to navigate autonomously by leveraging a camera to perceive its environment and make decisions on future movements.

4.3.1. Raw Model Performance. The model can drive the car around the track but it does make mistakes. It struggles with sharp corners the most but has issues avoiding some obstacles as well. This is caused by several factors, mainly the quality of the images from the camera, insufficient training data, and the difficulty of the track itself. Fortunately, this model is perfect for our use case since we can test the safety layer more thoroughly, as it will be utilized often in our testing.

4.4. Safety Layer

In this section, we will discuss the steps taken in order to integrate our safety layer on top of the DonkeyCar platform [4] as a wrapper that can easily be reused or modified, a decision motivated by our commitment to enhance the car's operational safety. This bespoke safety layer is designed to substantially mitigate collision risks and furnish the car with the ability to recover from potential collision scenarios, thereby elevating overall safety.

4.4.1. Simplified Algorithm. Our initial approach to implementing this wrapper was to reuse the code provided in the original paper [1]. Yet, given the hardware limitations of the autonomous driving platform we were using, it was impossible to execute the suggested algorithm as is. For that reason, we decided to simplify it to the form that our platform could handle.

Firstly, as the only sensor that the car can use to determine its state in $\mathbb{R}^2$ is IMU, which has only finite precision, not only we can store the state on a hardware level as a pair of integers, but also encode the set of illegal states as a binary matrix. In this matrix, the pair of integers would act as a pair of indices, allowing us to check the legality of a given state. The resulting matrix (Figure 3, illegal zones colored in white) has 0.5cm precision, with a dimensionality of 446x446.

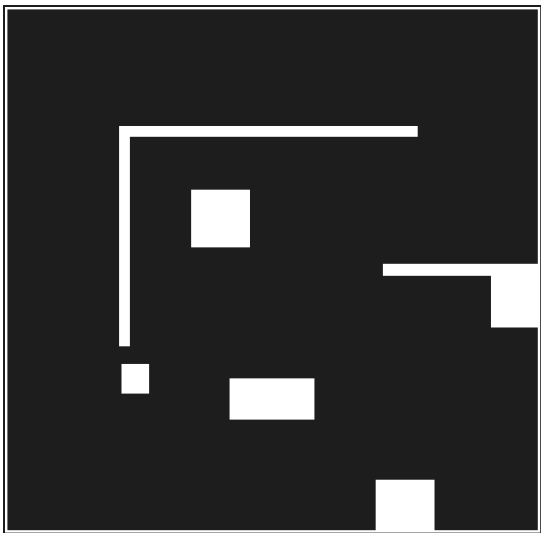

Figure 3: The visualization of illegal states

Secondly, given the slow speed of the car, we were able to reduce the number of receding-horizon steps up to 1 without any problems for safety, further decreasing the complexity of the algorithm.

4.4.2. Reading IMU Data. The car position within this map is mapped by subscribing to the updates of the IMU node exposed by the JetRacer image. In order to start the said node, we were using the command provided in Listing 2.

Listing 2: IMU Node Start

```
roslaunch jetracer jetracer.launch
```

4.4.3. Debugging. In order to quicker find mistakes in our code and to provide a visualization of the technical part of this project, we have created an application

capable of rendering the state of the car in real-time (Figure 4, illegal zones colored in white, the position of the car is indicated by a red circle). It does so by exposing a UDP port on a running device, to which the car could send the updates on its state as well as additional debug data in an async manner.

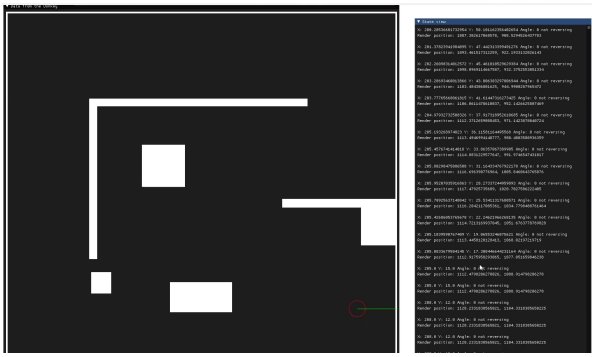

Figure 4: The digital visualization of movement

4.4.4. Summary. With that, we have the following algorithm. Firstly, the current position is obtained using IMU. Then by checking for collisions, we can determine whether or not to engage recovery mode. If recovery should be engaged, the car reverses (as long as there is no new collision detected) for a few seconds and then the RNN model takes the control back over. Figure 5 also provides a view of how this algorithm is implemented in the form of a wrapper in the code.

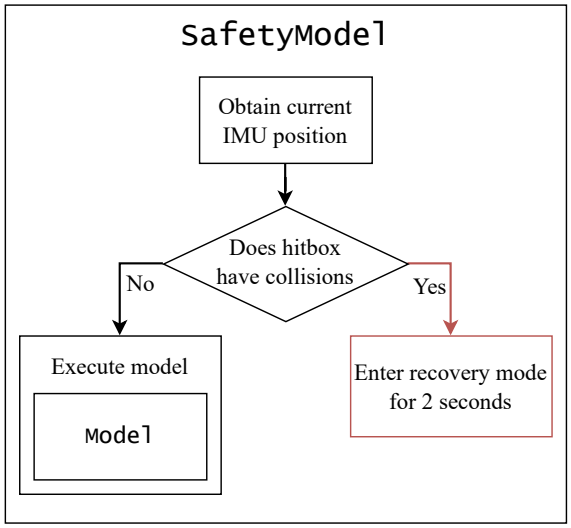

Figure 5: The safety model implementation

5. Results

The final phase of our project involved implementing the trained model and safety features on the physical robotic car. We conducted numerous real-world tests to confirm the system's functionality and reliability. The results demonstrated that our approach effectively combined machine learning with real-time simulation, enabling the car to navigate autonomously while avoiding collisions.

In summary, our methodology encompassed the following key steps:

- 🚗 Selection and configuration of the robotic car kit;
- 📖 Training the RNN LSTM model;
- 💥 Development of a custom safety layer for collision detection and recovery;
- 👷 Extensive simulation and testing to validate the model and safety features;
- ✅ Implementation and real-world testing to ensure reliable autonomous navigation.

Videos showcasing the functionality can be found in a Google Drive folder[2]. YouTube video of the sample safety layer running can be found here[3]. The GitHub repository with our code is also available online[4].

6. Conclusion

The safety layer has proven to be a viable option for avoiding obstacles. It successfully prevents the car from hitting any obstacles or lines. It recovers the car successfully, and then the model proceeds to take back over and successfully navigates problematic corners and obstacles.

One limitation that we encountered was the hardware platform running out of RAM and CPU since the freeware that we were using was single-threaded and not multi-threaded. Thus, without extra steps taken on startup, the car drives slowly and jumpy as it tries to keep up with the slow single core. This also leads to inaccuracies with the IMU and, thus, our safety layer. We have observed that if we let the program run for a few seconds to finish up all the initial tasks until the end, we get rid of this problem, but it takes a couple of minutes.

References

[1] M. Selim, A. Alanwar, S. Kousik, G. Gao, M. Pavone, and K. H. Johansson, "Safe reinforcement learning using black-box reachability analysis," *IEEE Robotics and Automation Letters*, vol. 7, no. 4, pp. 10 665–10 672, 2022. [Online]. Available: https://ieeexplore.ieee.org/document/9833266

[2] The JetRacer is produced by the brand WaveShare. [Online]. Available: https://www.waveshare.com/wiki/JetRacer_ROS_AI_Kit

[3] The JetRacer uses Robot Operating System to control the movement and allow for data flow in between sensors. [Online]. Available: https://wiki.ros.org/

[4] The JetRacer uses a Python self-driving library, DonkeyCar release 4.4.0. [Online]. Available: https://github.com/autorope/donkeycar

2. Google Drive link: https://drive.google.com/drive/u/4/folders/1LBQNHNGYqd2sx3zTi2BTtfxd9h3JRHgH

3. YouTube link: https://www.youtube.com/playlist?list=PLv1CQOJ_BloPb-eMFPb94HP8QZfsvYy9g

4. GitHub link: https://github.com/ValachPatrik/Safety-Layer-for-autonomous-driving-with-Donkeycar