informatics

MDPI

Article

Development of a Simulator for Prototyping Reinforcement Learning-Based Autonomous Cars

Martin Holen [1,*], Kristian Muri Knausgård [2] and Morten Goodwin [1]

[1] Centre for Artificial Intelligence Research, University of Agder, 4879 Grimstad, Norway; morten.goodwin@uia.no
[2] Top Research Centre Mechatronics, University of Agder, 4879 Grimstad, Norway; kristian.m.knausgard@uia.no
* Correspondence: martin.holen@uia.no

Abstract: Autonomous driving is a research field that has received attention in recent years, with increasing applications of reinforcement learning (RL) algorithms. It is impractical to train an autonomous vehicle thoroughly in the physical space, i.e., the so-called 'real world'; therefore, simulators are used in almost all training of autonomous driving algorithms. There are numerous autonomous driving simulators, very few of which are specifically targeted at RL. RL-based cars are challenging due to the variety of reward functions available. There is a lack of simulators addressing many central RL research tasks within autonomous driving, such as scene understanding, localization and mapping, planning and driving policies, and control, which have diverse requirements and goals. It is, therefore, challenging to prototype new RL projects with different simulators, especially when there is a need to examine several reward functions at once. This paper introduces a modified simulator based on the Udacity simulator, made for autonomous cars using RL. It creates reward functions, along with sensors to create a baseline implementation for RL-based vehicles. The modified simulator also resets the vehicle when it gets stuck or is in a non-terminating loop, making it more reliable. Overall, the paper seeks to make the prototyping of new systems simple, with the testing of different RL-based systems.

Keywords: autonomous driving; simulators; reinforcement learning

Citation: Holen, M.; Knausgård, K.M.; Goodwin, M. Development of a Simulator for Prototyping Reinforcement Learning-Based Autonomous Cars. *Informatics* **2022**, *9*, 33. https://doi.org/10.3390/informatics9020033

Academic Editor: Antony Bryant

Received: 1 March 2022
Accepted: 1 April 2022
Published: 15 April 2022

1. Introduction

Many simulators have been developed for autonomous cars [1–7], and some of the newer ones include support for reinforcement learning (RL) [2,4,6]. There is a focus on using standardized methods for testing, such as using code interactions in OpenAI Gyms [6]. Some simulators focus on the realism of their environment, creating highly realistic sensors, allowing the control of other actors in the simulation. Several of these simulators are available for free, while others charge a fee.

One of the major issues with existing autonomous car simulators is the ease of prototyping. Most simulators are large software systems and can often have less modularity while each part is decoupled, making them difficult to edit. Moreover, some do not use a standard editing engine, such as the game engines Godot, Unity, or Unreal, or a robotics engine such as MATLAB's Automated Driving Toolbox, Gazebo, or MuJoCo. RL has proven to be successful in tasks many consider more difficult than driving, such as cooperative games [8].

Reward hacking is a term used in RL algorithms when the agent finds an exploit, obtaining a high reward while not carrying out the intended task. Examples include picking up repair kits and damaging the boat in racing games, to obtain a reward equal or higher than from driving the boat. Each environment must consider that an RL algorithm will try to exploit the reward functions. Therefore, the reward functions must be created to stop the algorithm from reward hacking. Testing is required to determine which reward functions

results in the correct behavior and how to train the algorithms faster. For this, there is a lack of exhaustively tested simulators available.

We introduce a modified simulator to aid the training of RL-based autonomous vehicles, which focuses on the prototyping of systems. More specifically, the simulator allows for testing open questions in the research of RL autonomous vehicles, such as the role of continuously resetting the vehicle, explicit road tile rewards, and explicit road detection to guide the vehicle. We compare the developed simulator with the existing state-of-the-art CARLA system and show that our novel method is more suitable for RL research than the established CARLA system.

This simulator would be a step towards testing new and innovative reward functions, starting with those transferable to the real world, for instance, a road-detection algorithm.

This paper is organized as follows. Section 2 will focus on background information, including information on other simulators, some implementations of RL in different fields, and specific information on autonomous cars using RL. Section 3 explains the methods used, the inner workings of the simulator, its states, actions, and rewards, how the implemented sensors work, the reward functions, and how the vehicle avoids ending up in a non-terminating loop. The subsequent section discusses the results of our work, and the final section concludes the work and discusses future improvements.

1.1. Research Questions and Hypothesis

By developing a new simulator with RL research in mind, there are several open questions on how a simulator can best support research, which leads us to the research questions surrounding the environment.

As previously mentioned, testing the reward functions can verify whether the reward functions in the simulator obtains the desired behavior.

If an RL algorithm does not reward the hack, it should behave as the programmer intended. Currently, most simulators have somewhat different reward functions, and the influence of each reward function is not yet known.

We aim to create a lightweight simulator that supports RL with minimal effort for researchers. We would further like to investigate to what extent typical RL-based research is possible with such a lightweight simulator, which is not possible with current simulators such as CARLA.

1. Research Question (RQ)1: Will creating a reset system for the vehicle that stops the car after it is stuck improve the learning speed?
2. Research Question (RQ)2: Will creating a reward for each road tile be sufficient to inform the vehicle to stay on the road?
3. Research Question (RQ)3: Will a penalty for driving off the road decrease the number of time steps required for learning to stay on the road?

RQ 1 is a setup in which the vehicle is reset if it gets stuck or is taken off the road for too long. The vehicle is reset to see if it improves the speed at which the algorithm learns, in contrast to a scenario where the vehicle is allowed to carry out several actions before being reset. When the vehicle is stuck for a while, it is reset to not fill its memory with actions leading to nothing and which do not inform which actions to perform.

RQ2 refers to giving a reward for each road tile that the vehicle crosses. This reward aims to inform the vehicle of the distance it has traveled. We compare this method to one which uses tasks, i.e., first driving straight and, after completing this task, training the vehicle to drive straight and then turn, and so on.

RQ3 tests the effects of giving the actor a penalty for driving off the road, meaning giving it a reward of, for example, -0.01. We then see if its performance increases over time compared to the same implementation without this penalty. This reward function can easily be implemented in the real world. This allows for some feedback on a real car and includes the ability to stop an action that may be dangerous.

1.2. Intervention

Self-driving cars using RL is a relatively new field, which has received much interest in recent years. Training self-driving cars with simulators is a cheaper method than using physical vehicles. These simulators generally have different built-in reward functions [2,6]. They are often less modular and more decoupled, making changing the simulator to give the proper feedback to the vehicle difficult. Prototyping new reward functions could result in better performance and faster training. As we have seen RL performing better than humans in games [8–10], there is evidence that RL may benefit self-driving cars.

Our intervention is to modify a simulator by focusing on making a standardized system that gives feedback to the vehicle, testing these out, and finding which ones inform its actions best. This includes creating systems for resetting the vehicle when it gets into a non-terminating loop, giving it rewards based on how far it has driven, telling it when an episode finished, and more. Creating these systems, and being able to prototype such systems, will open opportunities for more research in this field and create innovative solutions for self-driving cars.

2. Background

Reinforcement learning (RL) is a rapidly growing field with some notable innovations in the last few years [8,9,11,12]. Among the most innovative discoveries includes playing games such as Atari [9] and Go while exceeding human performance [11] and performing on an expert level in Dota 2 [8]. RL methods have shown more remarkable performance compared to offline supervised learning-based methods [12]. This progress has led to an interest in the field and an ever-growing appetite for research to make RL-based autonomous vehicles.

An algorithm for RL self-driving cars needs a reward signal from its environment to guide the vehicle; training in a physical environment means continuously crashing to learn from its mistakes. Naturally, this is problematic and leads to most RL vehicle researchers using computer-based simulations. These allow the RL algorithm to perform many actions in a controlled environment. There are many simulation environments; each has its advantages and disadvantages depending on the problem area. Simulators that give a real number, representing a reward, include DeepDrive [4], PGDrive [6], and AWS DeepRacer [2].

2.1. Reinforcement Learning-Based Autonomous Vehicles

RL works with rewards given from the environment by performing actions. There are two main training methods: using data from human drivers and attempting to imitate them, or training based on the algorithm's actions. More recent implementations of self-driving cars include RL-based implementations, with some using imitation learning [13,14], and others performing RL from the ground up [15,16]. Imitation learning uses data from human drivers and attempts to imitate the same actions as those performed.

When implementing artificial intelligence (AI) in autonomous vehicles, it is common to divide the tasks into categories, namely scene understanding, localization and mapping, planning and driving policy, and control [17]. Scene understanding on its lowest level takes in the sensors and then tries to perceive the environment around it. An example of such an algorithm is a semantic segmentation algorithm that divides visible objects to distinguish between them [17]. Localization and mapping find the location of the vehicle and maps its surroundings. An example of this is the simultaneous localization and mapping (SLAM) algorithm, which finds the location of the vehicle in the environment and maps it [18]. The planning and driving policy revolve around multiple processes, such as lane planning, i.e., finding the fastest route to a location, and motion planning algorithms. It performs this without taking into account the constraints of the vehicle [17]. The controller defines the speed and angle of the steering wheel and translates the actions given by the driving policy.

2.2. Simulators

Different simulators focus on different topics; some focus on RL vehicles, others on supervised learning-based algorithms, and others focus on realism. Finally, we provide a brief overview of some simulators widely used in the industry and academia.

rFpro is a simulator designed to be highly realistic both in terms of its physics and visual fidelity, at the cost of computation, storage space, and monetary cost. It is made for supervised learning-based methods. rFpro includes a camera sensor and can purchase realistic sensor models from other companies and allows for the realistic simulation of sensors, with camera features such as a lens flare if the light is too bright [19]. Though the fidelity of these sensors is very high, and their maps are the most detailed amongst the other simulators, there is, however, no RL support for this simulator.

CARLA is a simulator made by the Computer Vision Center of Barcelona in cooperation with Intel and Toyota [1]. This simulator focuses on supervised learning, with a multitude of sensors and a variety of maps [20,21]. It includes autonomous agents, among which are vehicles, pedestrians, traffic lights, and more, giving the ability to control each agent or just a singular agent for a different degree of complexity [22]. CARLA is mainly focused on supervised learning (SL), not RL, with their RL-based version not being publicly available as of when this paper was written. The maps are focused on urban areas, and the Microsoft Windows support requires manual installation, with the recommendation being to create a virtual Python version in the CARLA folder. CARLA includes a wide variety of sensors, actors, and maps. Despite the description of RL [1], CARLA is missing an established and public baseline for RL research, indicating that it is cumbersome to implement and test RL-based vehicles. Instead, they use hard-coded agents to control vehicles, traffic lights, and other game actors.

DYNA4 is a simulator software created by Vector, which aims to aid prototyping and develop ADAS systems, testing, and modeling of vehicle dynamics. It includes a few sensors, including LiDAR, RADAR, Ultrasonic, and a camera [5]. The simulations are highly realistic, with good visual fidelity. This allows for training various ADAS algorithms, such as a lane-departure model. DYNA4 lacks support for RL-based autonomous vehicles with a focus on the industry and its needs.

MATLAB's Automated Driving Toolbox includes a variety of tools for the creation of an autonomous vehicle simulator. It includes a tool for creating a map for the vehicle to interact with and a scenario for it to drive. The tool also includes a few sensors, such as a camera sensor, LiDAR, and specifically a Velodyne LiDAR model [23].

Because it is a toolbox from which each programmer can create their simulator, reproducibility is more complicated here, as the maps must be shared by every programmer.

PGDrive is an open-ended car simulator based on RL. It is compatible with OpenAI Gym, making it easy to test with procedurally generated maps and a few premade ones [6]. The sensors include LiDAR, RGB, minimap, NaviMark, increment steering, and tire tension [24]. PGDrive is a simulator with rewards depending on the speed and if the vehicle is crossing a checkpoint, if it reached the goal, or broke any traffic rules, including collisions. The game engine used is Panda3D; the environments include a large variety of variability, and a significantly higher number of different maps than other environments, with a sound reward system. Due to its procedural nature, any additional systems would have to consider this procedural generation in creating and testing their systems, which adds complexity to the programming.

AWS DeepRacer is a racing simulator for training RL-based autonomous cars. Amazon developed this simulator, which can go from simulation to physical robots [2]. To train an algorithm in this environment, we need to use Amazon Web Services. The reward function is based on the location in the lane, with the highest reward being given for being in the middle of the road. If the vehicle is further from the middle, it is given less of a reward; if the vehicle is off the road, the vehicle receives no reward. The simulator has a simplistic reward system and does not run on the users' computers, instead running on the Amazon servers.

DeepDrive is a simulator for creating autonomous vehicles using SL or RL; it is made in Unreal Engine, with docker environments. It has a reward function that gives rewards and penalties based on the speed of the vehicle, the distance traveled from the last action, the vehicle's deviation from the center of the lane, and the G-forces felt by the vehicle [4]. It also has a reset system for the vehicle if it is not moving a lot. At the same time, the accelerator is being pressed down, implying the vehicle is attempting to move but cannot for multiple actions in a row. Actors can only use a camera as their sensors, so their sensor repertoire is limited. However, they can use this data through a script that saves the sensor data to files. Overall, the interface is standardized, and the reward system is comprehensive, but there is a lack in its sensors, and its code is decoupled.

Our paper aims to create a simulator that improves the training speed and rewards of an RL-based self-driving car with the ability to prototype new features quickly.

3. Method

Current simulators have large and complex codebases, which makes modifying them difficult. Creating a simulator for the aid of RL-based vehicles implies that researchers can test new reward functions, which may be possible to move from simulation to the real world. This prototyping allows for a better understanding of the effects of different reward functions, which aids training in a real-world scenario.

In order to create an RL-compatible environment, we modified and used Udacity's Self-Driving Car Simulator. The original simulator is made for one of Udacity's courses and uses the Unity game engine. Unity uses a deterministic physics engine, making replicating results easy. The modifications include giving feedback, such as rewards for driving over checkpoints and penalties for getting stuck or driving off the road. Other modifications are resetting the vehicle and creating a segmented view to train segmentation algorithms. An overview of the working of the simulator is presented in Figure 1.

Our sample follows the standard reinforcement learning experiment with a state (S_t), an action (A_t), a reward (R_t), and a next state (S_{t+1}) per time step t. Each of the states includes a list of images from time step $t - n$ up to the image from the current time step t, where n is a hyperparameter that limits the memory usage. The actions are the angles the steering wheel moves to control the vehicle, so action 0 is -0.77, which means an angle of -25.6 degrees. A reward is a real number, and the higher the reward, the further the vehicle drives.

Comparing the simulation experiments to those used in the CARLA papers [1], we create some tasks, split along the road as outlined below:

- Task 1 is to drive straight, leading to a hill.
- Task 2 requires the vehicle to drive straight and then drive up a short twisting hill and turn the corner.
- Task 3 is to clear a steep right turn up a hill, then turn left when on top of the hill. The road continues with twists and turns after this point.

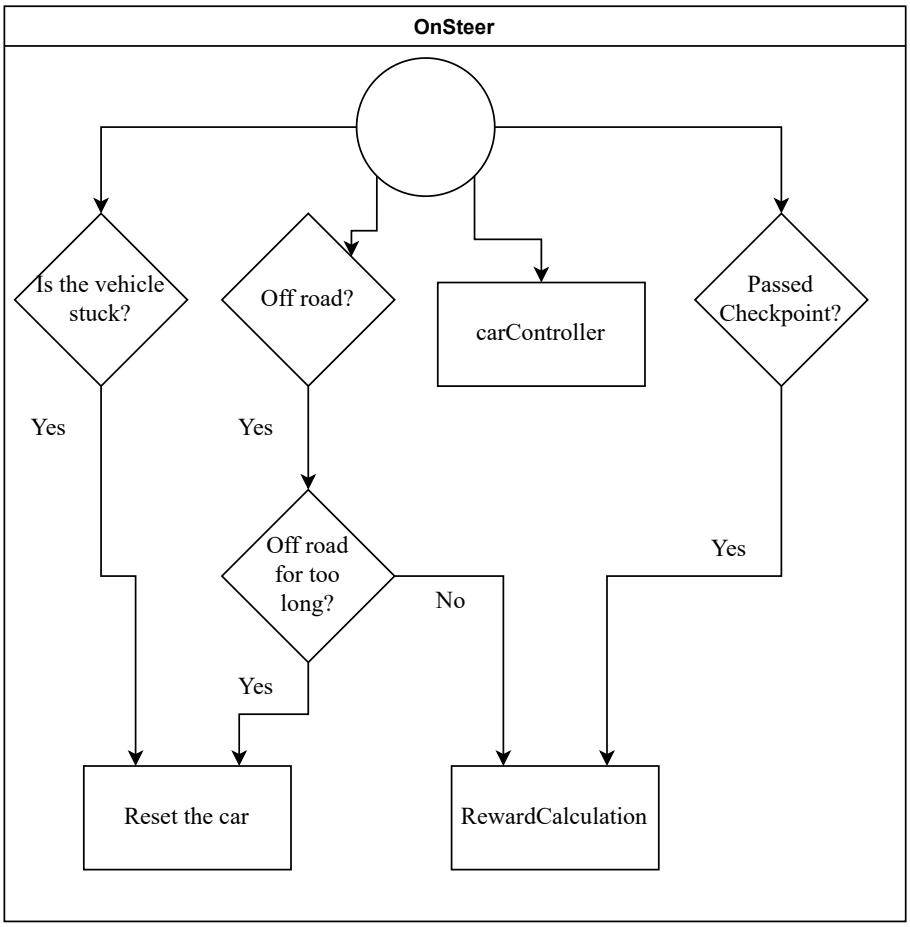


Figure 1. Interaction when given a steering command. The container is the function used, namely onSteer. The rectangles represent actions, e.g., resetting the vehicle or steering the vehicle. The rhombuses represent if statements, e.g., if the vehicle passed a checkpoint, give it a reward.

3.1. Sensors

Sensors help the AI sense its environment, locate objects, or determine what the vehicle is doing. Without proper sensors, the vehicle is not able to maneuver in its environment.

The creation of five virtual sensors adds an ability to sense the environment, and three of them were developed during this study. In addition, the study adopts two sensors developed by Unity and Udacity to develop the system. The five sensors are defined below:

- Sensor 1 detects when the car drives off the road, giving a penalty for going off the road. It was developed by the authors and was validated by manually driving in the environment, checking when any of the tires were off the road, and confirming that the sensor was triggered.
- Sensor 2 is the location sensor, detecting if the car is in the exact location for a more extended period. The location sensor is similar to a GPS, in that it gives the position relative to the world's origin. This sensor is created by Unity and is part of the game engine itself.
- Sensor 3 is a camera sensor made to capture RGB images of the environment the vehicle is driving in. This sensor comes with Unity, giving an idealized view. The version of the sensor included does not include ray tracing or any such feature, instead relying on an overhead light, with more simplified lighting calculations.
- Sensor 4 checks if the vehicle is stuck and resets it. The sensor is also an addition from the authors, and uses a ray cast from the tires, aims it downwards, and checks if there is a road tile below it. This is carried out by checking the render mask of the objects

below it, and if any of the tires are not on the road, the vehicle is reset. Combining this with a small script developed by the authors, we check if the vehicle has been stuck, has been by the divider, or is off-road for 300 actions. If the vehicle has carried out any of these for 300 actions, it is reset to a checkpoint and gives feedback back to the vehicle, telling it that the episode finished.

- Sensor 5 is a segmented camera, which is also an idealized sensor. We created it by making changes to shaders in a duplicated world, and these shaders show the respective color of an object, such as when the road would be a different color than the trees. The passive data collector for the segmented camera is shown in Figure 2. To obtain the image in the same location in both environments, we shadow the vehicle driven by the AI algorithm. This vehicle can also be driven using a navigation mesh, which allows for the passive collection of segmented images. An RGB and segmented image can be found in Figure 3a.

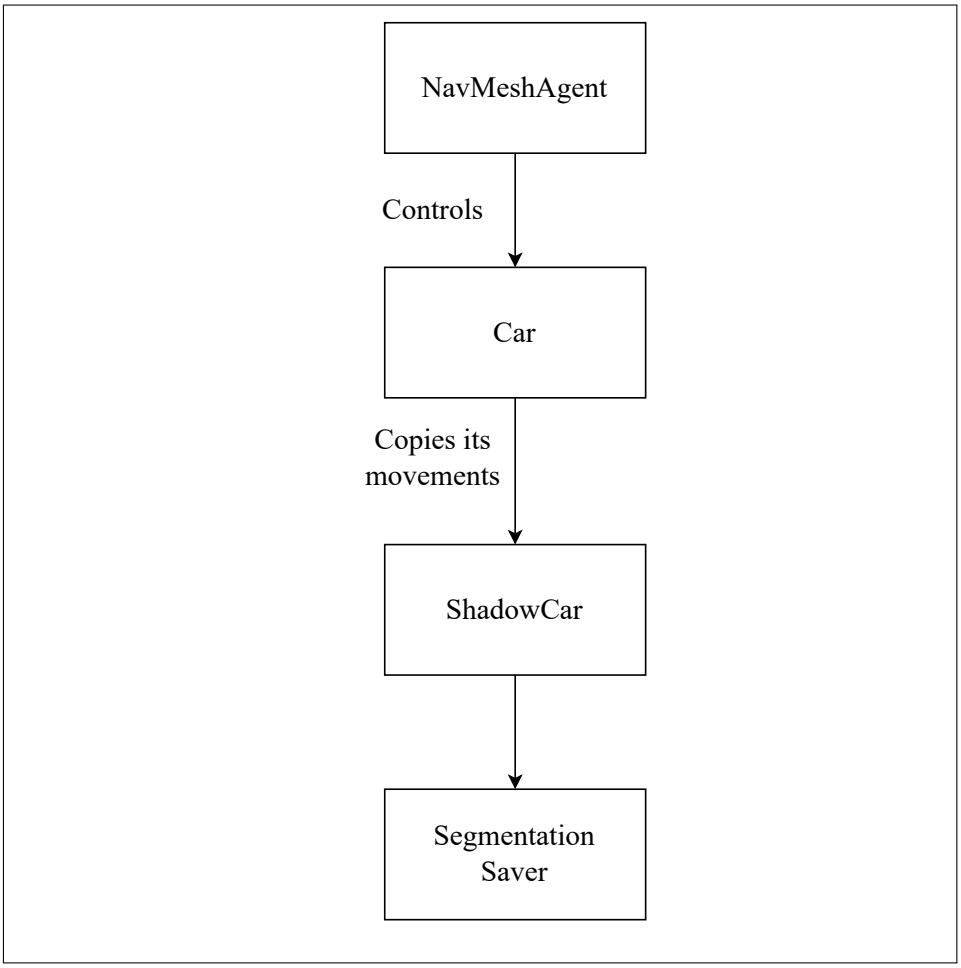
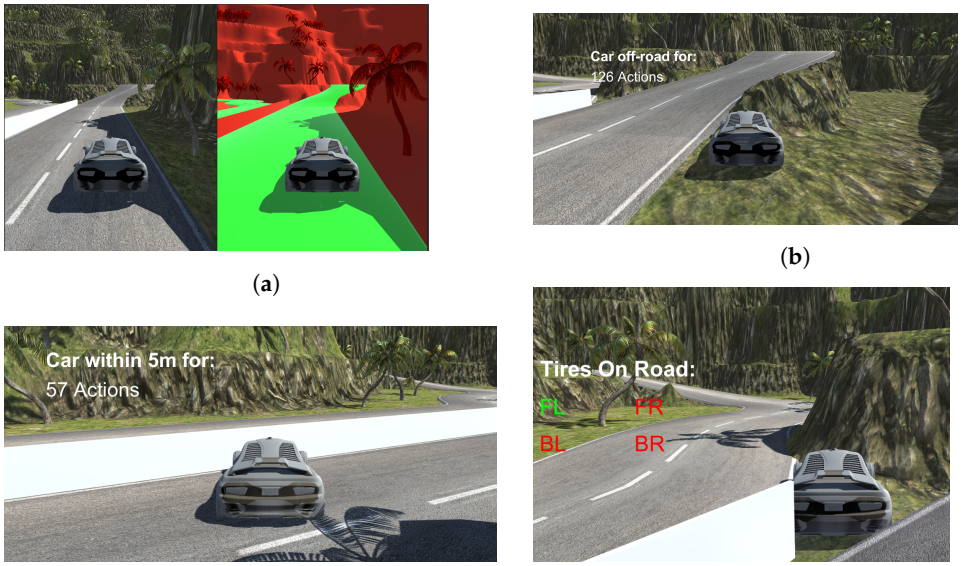


Figure 2. Overview of the modifications for performing segmentation.

Figure 3 shows the segmentation photos, the tires which are off the road and the number of times any of them have been off the road, as well as the number of times the vehicle has been in the area. Figure 3a shows the comparison between the RGB and the segmented image while Figure 3b shows the number of actions where the vehicle had any of its tires off the road. The number of actions during which the vehicle was within 5 m is shown in Figure 3c. Figure 3d shows which of the tires are on the road, with the green letters representing tires on the road, and red representing tires off the road. The letters FL refer to the front left tire, while the letters BR represent the back right tires.



(**a**) (**b**) (**c**) (**d**)

Figure 3. Shows the segmentation photos, the tires which are off the road and the number of times any of them have been off the road, as well as the number of times the vehicle has been in the area.

3.2. RL Feedback

The testing and validation of RL algorithms require a form of feedback to be given to the actor. This feedback aims to tell the actor how well it performs in the environment. The more representative a reward function is to the performance of the actor, the better. There is also the addition of resetting the vehicle when it gets into a non-terminating loop. Stopping a non-terminating loop is essential, as it means the vehicle can move in the environment and not simply try to act while it is stuck. This leads us to three phases, where we experiment with resetting the vehicle when it is in a non-terminating loop, creating a reward function that includes checkpoints and road detection.

Experiment 1 is to explore the effects of an experiment that resets the vehicle when it reaches an undesirable state, to speed up training. This is carried out by implementing an RL algorithm-based experiment that resets the vehicle when it is stuck in the same area over time or if the vehicle is consistently off the road. If the vehicle is either off the road or in the same area for more than 300 actions, it is reset to keep training. This experiment does not have any road tile reward. This experiment uses the RL system described in the study presented by CARLA [1].

Experiment 2 is meant to test the effects of adding a checkpoint experiment that includes each road tile, i.e., road tile rewards, to see if getting a reward from driving over a road tile is better than receiving a reward for tasks such as driving straight. The vehicle is tested on driving straight, driving around a corner, up a twisting hill, and down a twisting slope.

Experiment 3 is meant to test the effect of giving the vehicle a penalty for driving off the road. This penalty is a negative reward of -0.01 to inform the vehicle that the action it is carrying out is not desired. In this experiment, we also include the road tile reward. An overview of each of these experiments' systems can be found in Table 1.

Table 1. Experiment overview with road tile rewards (RTR), off-road penalties (ORP) and CARLA's built-in experiments (CBIE).

Experiments	RTR	ORP	CBIE
Experiment 1	No	No	Yes
Experiment 2	Yes	No	No
Experiment 3	Yes	Yes	No

3.3. RL Algorithm

During the testing for each of the experiments, we created a Deep Q-Network (DQN) algorithm [9]. This algorithm used a three-layer neural network, with an input layer of 200 by 66 by 3, a hidden layer of 64, and an output layer of 15 actions. These actions determine the amount the vehicle should steer to either side, with the accelerator being set at maximum, and the speed limit being set to 20 km/h. The accelerator is set to the maximum because it is necessary to get up the hill, and the speed limit is slightly above the minimum required speed.

The input state was the image and was repeated 16 times in a queue, to provide a perspective of time.

3.4. Modifications

The simulator required quite a few modifications to start working for its purpose. We created rewards based on checkpoints through the location of the car, as well as checking if it has driven over a road tile. We made a couple of sensors, namely a segmented camera and a road detection sensor. Changes could be quickly carried out by running the new scripts in the CommandServer script, easily adding new features. This makes the simulator easy to modify, as one only needs to call the scripts from the CommandServer, with every other script not affecting any of the changes made.

Segmented camera The segmented camera is created by duplicating the mesh of the world and all of the objects. Once they are duplicated, the objects have a material added to them, which creates their color. Said material does not have any reflectiveness, making the color closer to its RGB value no matter the shine thrown at the object. We then create an agent who can move around the world. This agent is a simple object, and a script that copies the main vehicle's movements is added to it. The script copies the main vehicle's x, y, and z positions with an offset. This offset means that the vehicle is placed in the same spot, but instead of being in the RGB world, it is placed inside the segmented world.

A group of cameras is added to the RGB and segmented vehicles for passive data collection. These cameras are located at the same position in local relation to the vehicle. When the cameras are added to the vehicle, they are used to collect segmented and RGB images from the same position. This is carried out by taking the cameras and saving them to a list every timestep the vehicle moves. We use a coroutine to save all the images close to the same time (generally within one frame). Once a large number of images has been stored, the time is stopped, and the images are saved to be filed in order. Stopping the time can be carried out simply by setting the timescale to 0, then back to 1 after the actions are performed. The RGB images and segmented images are saved in two separate folders. The only difference in their naming is that one includes segImages, and the other includes images in their path.

Off-road penalty Creating the road detection sensor requires a few steps. We take the tires, each of which has a Boolean corresponding to either on or off the road. We then use a raycast below the tires and see if we can find the road tiles. This is carried out by looking for the object's layer mask: "Road". If the mask is there, the vehicle is on the road, and the Boolean stays at false; otherwise, the Boolean is set to true. As this is carried out for all four tires, we obtain an overview of which tire is off-road.

Road tile checkpoints When creating the checkpoint rewards, we add the road tiles to a list, starting with the tile where the vehicle spawns. This list is ordered so that the following road tile is the connected road tile in the direction the vehicle is spawned. When the vehicle is less than 6m away from the next road tile (meaning it can be 1m off the road), we increase the reward and set the current goal to the next road tile in the list. The reward differs depending on whether the road tile is straight, with straight tiles giving a lower reward.

Reset system The reset system is a simple one, with two associated scripts. These scripts see how long the vehicle has been off the road and how long it has been within 5 m. If the vehicle has been off the road for n number of actions, or if the vehicle has been within

5 m of its current location within n actions, with n being a parameter set by the programmer. To check if the vehicle is off the road, we take the Boolean from each tire, seeing if any of them is true. Checking how long the vehicle has not moved is carried out via adding the vehicle's position every time it acts and comparing the location at index a and index a+parameter. The reset function is called if the vehicle's location is right next to it. This reset function resets the vehicle's position to the starting position (and slightly above the ground), and it resets the vehicle's velocity and acceleration, setting them to 0. The position can be changed or even parameterized to place the vehicle at a specific location randomly. However, this also requires that the road tile list be changed accordingly.

Passive data collector Along with the other systems, we also created a passive data collector. The data collector consists of a few things including an agent and a script for saving images (which can be RGB or both RGB and segmented images). The agent is a NavMesh agent, which drives around the map collecting images for each frame. The NavMesh agent is created by adding a NavMesh to the road surface and a value representing the weight for the vehicle to drive on the surface. The weight for the road is less than the weight for driving off-road, which generally causes the vehicle to stay on the road. We then need to create its targets, which are manually placed down and moved through some randomness to increase the variation in the vehicle's locations. The randomness is an added ± 2.5 m in the x and z position (can be 2.5 in both positions or only one). The vehicle then goes through its list of goals and moves from one goal to the other. We use the same script we did for saving RGB and segmented images for storing them in different folders (this can also be carried out when the vehicle is off or on the road).

4. Results

In this study, we have shown that introducing a lightweight simulator can support RL research better than more complex simulators, such as CARLA. The simulator, based on Unity, required several modifications to improve the performance of the applied RL algorithm. We trained an RL algorithm, specifically a Deep Q-Network (DQN), to verify this. The modifications were compared to the methodology introduced in the paper from CARLA [1].

Figure 4 shows the number of backpropagations for experiments 1, 2, and 3 and the respective rewards for each backpropagation. In this figure, an increase in reward indicates that the experiment has learned the actions correctly.

Research Questions

The experiments, as previously described in Section 3, focus on the different systems as compared to the research questions. Namely, the first experiment uses the CARLA systems for resetting and using tasks to give feedback. The second experiment focuses on changing this to giving feedback whenever the vehicle crosses one of the road tiles, which make up the road, and reset the vehicle when it gets stuck. Lastly, we have an experiment that builds on experiment 2 by adding a penalty for driving off the road.

For each research question, obtained the following results:

Research Question 1: "Will the creation of a reset system for the vehicle that stops the car after it is stuck improve the learning speed?" Figure 4 shows that experiment 3 which, along with experiment 2, includes the system that resets the vehicle after being stuck, performed the best. We see this in the accumulated reward for experiment 3 compared to experiments 1 and 2. This implies that resetting the vehicle improves the training time.

Research Question 2: "Will creating a reward for each road tile be sufficient to inform the vehicle to stay on the road?" This tests whether it is sufficient to only have road tile rewards to guide the vehicle properly. Experiments 2 and 3, which have rewards on the tiles, outperform system 1, indicating that the reward for driving to the next road tile improves training. As seen in Figure 4, even though experiments 3 and 2 are significantly faster at training than experiment 1, experiment 2 finds local optima. This discrepancy

indicates that reward tiles by themselves are not sufficient feedback. Hence, if a road tile reward alone turned out to be sufficient, experiments 2 and 3 would perform equally well.

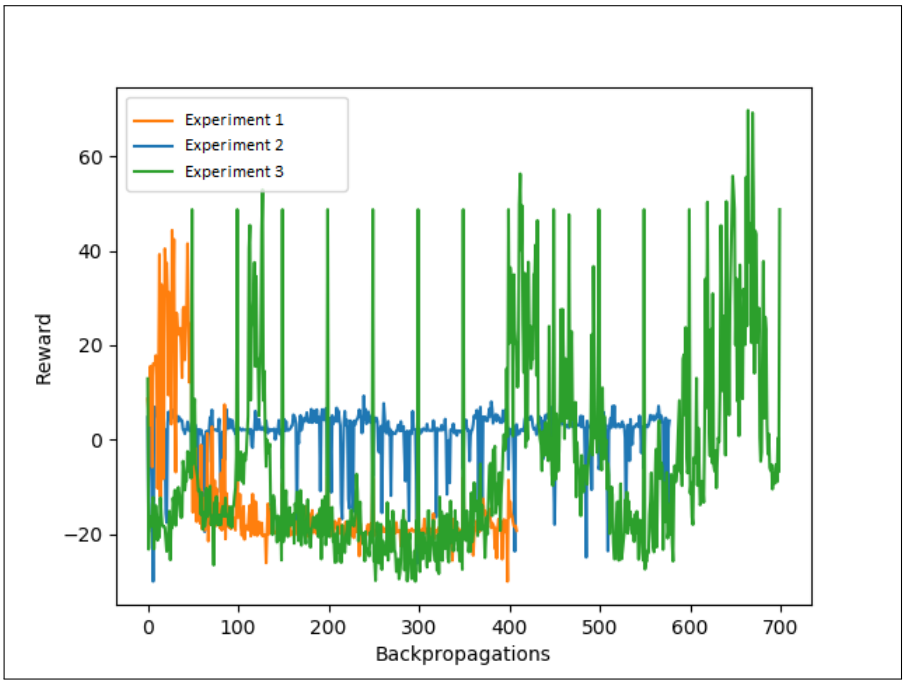


Figure 4. Normalized rewards.

Research Question 3: "Will a penalty for driving off the road decrease the number of time steps required for learning to stay on the road?". As seen in Figure 4, experiment 3 was the best performing experiment. The experiment shows that the penalty for driving off the road decreases the number of time steps required for learning to stay on the road. Hence, this indicates that the penalty for driving off the road improves the training time. Note that the vehicle in experiment 3 should obtain fewer rewards than the other experiments since it also receives penalties for driving off the course. This is, however, not observable in the results (see Figure 4). The opposite is observed; the vehicle in experiment 3 receives more rewards, further indicating that road detection and the penalty for driving off the road improves the training time and, in a collaborative manner, improves the system.

Looking at Figure 4, we find that experiments 1 and 2 do not seem to be improving a lot over time, which is likely caused by the systems finding local optima. The variations we see in their rewards decrease as the number of random actions the DQN algorithm performs decreases.

5. Discussion and Conclusions

The results are in line with our initial hypothesis. We tested three experiments: an experiment similar to that of the CARLA paper (experiment 1); rewards based on checkpoints with a reset experiment (experiment 2); and lastly, experiment 2, with an added penalty based on road detection (experiment 3).

The scope of the study limits us from running multiple times over different seeds, and as such, the same random seed was used for the different experiments. Using the same random seed could decrease the variation between the experiments. This means that given another random seed, some difference in the rewards for each experiment is likely, though the current experiments show some benefit to the systems as the reward is higher, given the same random seed. Experiment 1 uses the systems of CARLA, but the reward received for completing one task is equivalent to the reward received for driving the same distance, given the road tile reward function.

The main expectation was that the average amount of rewards would increase from experiment 1 to experiments 2 and 3, which is observed in the results. The average reward increases the penalty for driving off the road or driving into the divider.

In the first two experiments, we saw that the vehicle got into a local optimum when testing experiments 1 and 2. Experiment 2's local optima were to drive past two of the first checkpoints and then reset when driving off-road. For experiment 1, the vehicle managed to get to the base of the hill only a few times in the beginning, though eventually, when it performed fewer random actions, it drove directly into a wall or the divider on either the right or the left side of the road, and was stuck until the reset.

Local optima may be due to randomness; as at the beginning, they perform actions at random, according to the DQN algorithm.

Half of them steer to the right and the other half to the left when performing random actions. The rewards in experiment 2 are higher than in 1; this may be due to experiment 2 receiving a reward for each road tile it drove past. For experiment 1, the rewards were less frequent, with a minor penalty of -0.01 for each action, which was the main form of feedback. When looking at experiment 3, we see significant variations in the rewards, meaning that the experiment is exploring, ending up with a reward larger than the other experiments.

The vehicle in experiment 3 should receive lower rewards than the other experiments since it also receives penalties for driving off the course. Experiment 3 receiving lower rewards is, however, not observable in the results (see Figure 4). Instead, we noticed that the vehicle in experiment 3 receives more rewards, indicating that road detection and the penalty for driving off the road improve the training time and collaboratively improve the system.

We conclude that the augmented Udacity simulator has the necessary experiments for training RL-based autonomous cars. The modified simulator developed in this study has experiments for training the RL algorithm, giving a standardized reward function, and resetting the vehicle when it goes off the road or ends up in a non-terminating loop. These experiments aid the training of RL algorithms to train autonomous vehicles while giving a standard set of reward functions for training. Evidence also shows that the checkpoint and road detection algorithm works better than the CARLA paper's methodology; resetting the vehicle when it gets stuck or stays off the road for too long seems to be better than resetting the vehicle every n number of actions. More testing and prototyping of rewards similar to other simulators may aid the performance of algorithms trained on the developed simulator.

Additional Insight and Future Work

Several improvements can improve the simulator, such as improving the experiments and reward mechanisms using more realistic sensors, including virtual machine containers or OpenAI Gym compatibility. The improved reward mechanisms would give the vehicle more information on which actions were good and bad. Realistic sensors would allow for an easier transition from simulation to real-world environments or other more realistic simulators. The container compatibility and gym compatibility would allow for easier testing on servers. In addition, with gym compatibility specifically, it would allow for standardized algorithms such as the baseline algorithms.

Author Contributions: Software, M.H.; investigation, M.H., K.M.K. and M.G.; resources, M.H.; writing—original draft preparation, M.H.; writing—review and editing, K.M.K. and M.G.; visualization, M.H., K.M.K. and M.G.; supervision, M.G. and K.M.K. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: We would like to thank DYNA4 and rFpro for their interviews in which they informed us about their simulators.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Dosovitskiy, A.; Ros, G.; Codevilla, F.; López, A.; Koltun, V. CARLA: An Open Urban Driving Simulator. In Proceedings of the 1st Annual Conference on Robot Learning, Mountain View, CA, USA, 13–15 November 2017; pp. 1–16.
2. Balaji, B.; Mallya, S.; Genc, S.; Gupta, S.; Dirac, L.; Khare, V.; Roy, G.; Sun, T.; Tao, Y.; Townsend, B.; et al. DeepRacer: Educational Autonomous Racing Platform for Experimentation with Sim2Real Reinforcement Learning. *IEEE Int. Conf. Robot. Autom.* **2020**, 2746–2754.
3. Naumann, M.; Poggenhans, F.; Lauer, M.; Stiller, C. CoInCar-Sim: An Open-Source Simulation Framework for Cooperatively Interacting Automobiles. *IEEE Intell. Veh. Symp. Proc.* **2018**, *2018*, 1879–1884. [CrossRef]
4. Quiter, C.; Rehn, A. Deepdrive/Deepdrive: Deepdrive is a Simulator That Allows Anyone with a PC to Push the State-of-the-Art in Self-Driving. 2021. Available online: https://github.com/deepdrive/deepdrive (accessed on 1 February 2022).
5. Vector Informatik GmbH. DYNA4 | Virtual Test Driving | Vector. 2021. Available online: https://www.vector.com/int/en/products/products-a-z/software/dyna4/ (accessed on 1 February 2022).
6. Li, Q.; Peng, Z.; Zhang, Q.; Liu, C.; Zhou, B. Improving the Generalization of End-to-End Driving through Procedural Generation *arXiv* **2020**, arXiv:2012.13681.
7. Cai, P.; Lee, Y.; Luo, Y.; Hsu, D. SUMMIT: A Simulator for Urban Driving in Massive Mixed Traffic. In Proceedings of the IEEE International Conference on Robotics and Automation, Paris, France, 31 May–31 August 2020; pp. 4023–4029.
8. Berner, C.; Brockman, G.; Chan, B.; Cheung, V.; Dębiak, P.; Dennison, C.; Farhi, D.; Fischer, Q.; Hashme, S.; Hesse, C.; et al. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv* **2019**, arXiv:1912.06680.
9. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-level control through deep reinforcement learning. *Nature* **2015**, *518*, 529–533. [CrossRef] [PubMed]
10. Vinyals, O.; Babuschkin, I.; Czarnecki, W.M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D.H.; Powell, R.; Ewalds, T.; Georgiev, P.; et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* **2019**, *575*, 350–354. [CrossRef] [PubMed]
11. Silver, D.; Huang, A.; Maddison, C.J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. Mastering the game of Go with deep neural networks and tree search. *Nature* **2016**, *529*, 484–489. [CrossRef] [PubMed]
12. Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. Mastering the game of Go without human knowledge. *Nature* **2017**, *550*, 354–359. [CrossRef] [PubMed]
13. Sharifzadeh, S.; Chiotellis, I.; Triebel, R.; Cremers, D. Learning to Drive using Inverse Reinforcement Learning and Deep Q-Networks. NeurIPS Workshop on Deep Learning for Action and Interaction. *arXiv* **2016**, arXiv:1612.03653.
14. Liang, X.; Wang, T.; Yang, L.; Xing, E. CIRL: Controllable Imitative Reinforcement Learning for Vision-based Self-driving. In Proceedings of the European Conference on Computer Vision (ECCV), Munich, Germany, 8–14 September 2018.
15. Kendall, A.; Hawke, J.; Janz, D.; Mazur, P.; Reda, D.; Allen, J.M.; Lam, V.D.; Bewley, A.; Shah, A. Learning to drive in a day. In Proceedings of the IEEE International Conference on Robotics and Automation, Montreal, QC, Canada, 20–24 May 2019; pp. 8248–8254. [CrossRef]
16. Duan, J.; Li, S.E.; Guan, Y.; Sun, Q.; Cheng, B. Hierarchical Reinforcement Learning for Self-Driving Decision-Making without Reliance on Labeled Driving Data. *IET Intell. Transp. Syst.* **2020**, *14*, 297–305. [CrossRef]
17. Kiran, B.R.; Sobh, I.; Talpaert, V.; Mannion, P.; Sallab, A.A.A.; Yogamani, S.; Pérez, P. Deep Reinforcement Learning for Autonomous Driving: A Survey. *IEEE Trans. Intell. Transp. Syst.* **2020**, 1–18. [CrossRef]
18. Bailey, T.; Durrant-Whyte, H. Simultaneous localization and mapping (SLAM): Part II. *IEEE Robot. Autom. Mag.* **2006**, *13*, 108–117. [CrossRef]
19. rFpro. Driving Simulation for Autonomous Driving, ADAS, Vehicle Dynamics and Motorsport. 2021. Available online: https://www.rfpro.com/ (accessed on 1 February 2022).
20. CARLA Community. CARLA Simulator . 2017. Available online: https://carla.readthedocs.io/en/latest/ (accessed on 1 February 2022).
21. CARLA Community. Sensors Reference-CARLA Simulator. 2017. Available online: https://carla.readthedocs.io/en/latest/core_sensors/ (accessed on 1 February 2022).
22. CARLA Community. 2nd-Actors and Blueprints-CARLA Simulator. 2017. Available online: https://carla.readthedocs.io/en/latest/core_actors/ (accessed on 1 February 2022).
23. The MathWorks, I. Automated Driving Toolbox-MATLAB. 2021. Available online: https://se.mathworks.com/products/automated-driving.html (accessed on 1 February 2022).
24. Li, Q.; Peng, Z.; Zhang, Q.; Liu, C.; Zhou, B. Vehicle Configuration—PGDrive 0.1.1 Documentation. 2020. Available online: https://github.com/decisionforce/pgdrive (accessed on 1 February 2022).