Safe Reinforcement Learning in Duckietown

by

Jan Steinmüller

Bachelor Thesis in Robotics and Intelligent Systems

Submission: May 16, 2023 Supervisor: Dr. Amr Alanwar Abdelhafez

Constructor University | School of Computer Science and Engineering

Statutory Declaration

Family Name, Given/First Name	Steinmüller, Jan
Matriculation number	30004474
Kind of thesis submitted	Bachelor Thesis

English: Declaration of Authorship

I hereby declare that the thesis submitted was created and written solely by myself without any external support. Any sources, direct or indirect, are marked as such. I am aware of the fact that the contents of the thesis in digital form may be revised with regard to usage of unauthorized aid as well as whether the whole or parts of it may be identified as plagiarism. I do agree my work to be entered into a database for it to be compared with existing sources, where it will remain in order to enable further comparisons with future theses. This does not grant any rights of reproduction and usage, however.

This document was neither presented to any other examination board nor has it been published.

German: Erklärung der Autorenschaft (Urheberschaft)

Ich erkläre hiermit, dass die vorliegende Arbeit ohne fremde Hilfe ausschließlich von mir erstellt und geschrieben worden ist. Jedwede verwendeten Quellen, direkter oder indirekter Art, sind als solche kenntlich gemacht worden. Mir ist die Tatsache bewusst, dass der Inhalt der Thesis in digitaler Form geprüft werden kann im Hinblick darauf, ob es sich ganz oder in Teilen um ein Plagiat handelt. Ich bin damit einverstanden, dass meine Arbeit in einer Datenbank eingegeben werden kann, um mit bereits bestehenden Quellen verglichen zu werden und dort auch verbleibt, um mit zukünftigen Arbeiten verglichen werden zu können. Dies berechtigt jedoch nicht zur Verwendung oder Vervielfältigung.

Diese Arbeit wurde noch keiner anderen Prüfungsbehörde vorgelegt noch wurde sie bisher veröffentlicht.

16.05.2023 Jan Steinmüller

Date, Signature

Abstract

In recent years, reinforcement learning has proven to be very good at learning autonomous driving by randomly exploring the environment and learning from those explorations. This makes it very powerful for autonomous vehicles. However, because of the inherent random exploration, the autonomous vehicle might end up in unsafe or dangerous situations. Therefore this research proposes and compares a safety layer that will keep the vehicle safe, to the same reinforcement learning baseline. From this, a lot of things can be learned to further improve the safety of such exploration-based learning approaches for future implementations that could keep the vehicle even safer and potentially even learn faster during training.

Contents

1 Introduction

Deep reinforcement learning has been very successful at teaching autonomous vehicles how to drive autonomously. They have been proven to be better at adapting to new environments and can handle large amounts of data easier than standard control systems approaches. Another main advantage of reinforcement learning and deep reinforcement learning is that, unlike a lot of other popular machine learning approaches like imitation learning[1] and other regression algorithms, they do not need any training data. The agent will learn by itself by exploring the environment that it is in. This greatly reduces the time needed for data collection and labeling. It also removes the need to train large models which require large datasets and also large amounts of computing power.

However, these deep reinforcement learning agents like Deep-Q-Networks start their training by doing random actions and they learn from their resulting reward. While this random exploration of the environment is very good in order to learn and adapt to different environments, these random actions might be unsafe for the autonomous vehicle and possibly its passengers. Because of this, this research is introducing a safety layer that prevents unsafe actions and recovers to safe states if the vehicle already is in an unsafe state. Additionally, the reinforcement learning agent also learns from the actions taken to improve its safety in future situations. This should also speed up the overall training time of the reinforcement learning agent compared to the baseline agent without a safety layer since the agent will be able to learn if an action resulted in an unsafe state and disregard that action in similar future situations. The goal of this research is to test such a safety layer on a physical system that is close to real-life scenarios without being in danger of damage to the system or the environment. There is a need for a physical system because reinforcement learning algorithms are often trained and evaluated in simulation and then transferred to a physical system. However, these simulations are not perfect representations of the real environment and also lack external factors like sensor noise and the physical responses from the system not being fully predictable.

For these reasons, Duckiebots from MITs open source Duckietown [2] project were chosen. These Duckiebots simulate a simplified city-like environment on a smaller scale. These robots are also inexpensive and unlike a fully autonomous car are also cheaper to repair or replace.

2 Statement and Motivation of Research

The overall goal of this research is to compare a standard reinforcement learning approach like DQN with the same implementation with an added safety layer that will keep the autonomous system safe. The goal for the reinforcement learning agent was to learn to do lane following in a safe manner without leaving its own lane since this could potentially put the vehicle itself, other vehicles, humans, and the environment at risk. An ideal system should also recognize if it already is in a state that is considered unsafe and then recover by itself without human intervention.

To achieve these goals a safety layer was proposed in order to predict and evaluate the future state of an action selected by the reinforcement learning agent in the current state. This safety layer was based on a data-driven model approach in order to get a prediction that is as close as possible to the actual resulting state. During the training of the reinforcement learning agent, the safety of a selected action is checked, and then depending on how the action was selected, the action will either be optimized or another action will be selected. If no safe option was found, the robot and agent try to recover to a state where safe actions can be found again.

Therefore, the main questions that were investigated in this research are if a safety layer system like the one proposed here will keep the vehicle safe, whether it impacts the underlying reinforcement learning agent, and how reliable such a system is on a physical system that is influenced by sensor noise, unpredictability, and other factors. Additionally, it was investigated if there were any advantages that could stem from having a system that keeps the robot safe and closer to the intended position than an unbounded system whose only indicator of performance is the reward gained from executing an action.

One of the main decisions made during the planning stage of this research was to implement the reinforcement learning agent on the selected robotic system from scratch in order to be able to be in full control of where and when the safety layer will be executed and how the agent reacts to the information given by the safety layer. Because of this, Deep-Q-Networks was selected as the reinforcement learning agent because of their ease of implementation and also their ability to handle continuous state input, and their robustness against noisy input data.

To answer the selected questions, a low-cost open-source robotics project was selected for the physical robot and environment. The project of choice was Duckietown as these robots are a very close but simplified representation of real autonomous vehicle applications in the automotive industry. The robots have a small collection of cheap and easily available sensors on board. The main sensors are a camera, a time-of-flight sensor, wheel encoders, and an IMU. Additionally, these robots have been used in a lot of other research projects and therefore a lot of very useful packages already exist that can be used for this project's purpose. Furthermore, this project can also be shared, used, and improved by other institutions that are using Duckietown. For this purpose, the code for this research can be found on github[3].

There also already exists a lane following demo [4] within the open-source code of the Duckietown project. This demo was planned to be used as another comparison to the proposed approach since it is a pure control systems approach to lane following. Therefore it would be interesting how a machine-learning approach compares to a set of predefined mathematical rules that are fixed at execution time. Unfortunately, the lane following demo provided in the Duckietown repository seemed to not function correctly as the robot got stuck in turns and was not able to stay in lane on straight lanes. This made it impossible to compare with the reinforcement learning approaches proposed in this project.

3 Description of the Investigation

The main purpose of this investigation is to compare a reinforcement learning algorithm with a safety layer to the same reinforcement learning algorithm without a safety layer. Therefore the overall goal that the reinforcement learning algorithms were trying to achieve was selected to be lane following. This was mostly due to the sensors and packages that were already available for the Duckiebots. One of the used packages returns the pose within the lane that the robot is in. This was selected to be the foundation for the reinforcement learning agents since their reward functions could be based on this lane pose.

In the early stages of development, Q-learning [5] was the selected reinforcement learning method for its simple and model-free nature. The model-free nature was especially helpful as we have a physical robot that interacts with the physical world. In Q-learning there is a Q-table which has a Q-value at every possible state and action. Therefore the state space and action space are discrete and finite. Q-learning works by assigning Q-values to all possible actions in all possible states and then updating those Q-values iteratively with the reward from selecting the given action. These Q-values are calculated with the following formula:

$$Q\_values(s) = reward + discount\_rate * max(Q\_values(s')) \tag{3.1}$$

Since the Q-learning works with discrete finite state space, the early implementation was approximating the lane pose by rounding the lane pose to reduce the size of the state space and also reduce the size of the Q-table. This was still very sensitive to noise and not easily upgradeable to accept more input data. The reinforcement learning algorithm was later changed to Deep-Q-Networks [6] to work with continuous state space and also be more resistant to noise. This removes the need for having a huge Q-table with all possible states. It also speeded up learning for the agent as before, two states that were very similar might have had completely different Q-values. With Deep-Q-Networks this gets improved by giving the state to a neural network instead of using it as an index for a Q-table.

Deep-Q-Networks, which are also known as DQN, are an extension of Q-learning that introduces deep learning with neural networks into the process [6]. DQN works by calculating the Q-Values for a given state. The Q-values then dictate which action should be taken to gain the highest long-term reward. During training the Q-values are being trained similarly to how the Q-values are being calculated in Q-learning 3.1

Additionally a lot of already existing packages from the open-source Duckietown repositories are being used in the implementation. Most notably the lane pose node for input data to the agent and the kinematics nodes to make the robot move. Because of this, the project also uses the Duckietown messages found in the dt-ros-commons repository[7].

The states for the DQN agents are two-dimensional. They consist of the distance to the center of the lane and the angle towards the centerline of the lane. This can be seen in the diagram shown below:

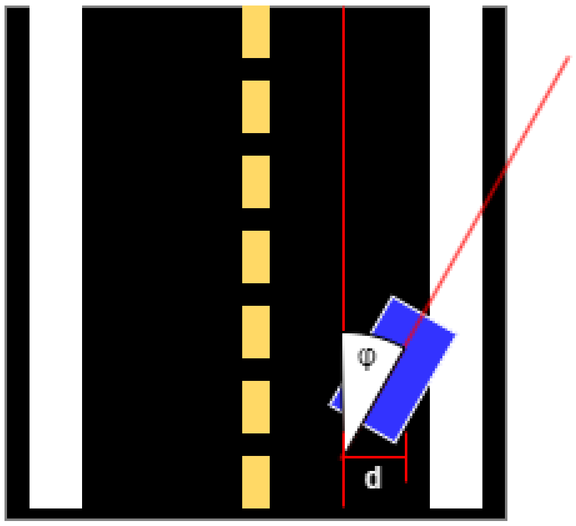

Figure 1: Lane Pose with distance d and angle φ

These values are published by the lane filter node[8] which can be found in the dt-core repository[9]. This package is also used for the provided lane-following demo which is based on the classical control systems approach. Unfortunately, this provided lane following demo [4] does not work well and therefore it cannot be compared to the reinforcement learning approaches in this investigation. The lane filter node estimates the position within the lane using the lines detected in the image and the encoder values from both motors. Unfortunately, this data is very noisy and the node can get confused when not enough lines are in view of the camera. In the implementation of the reinforcement learning agent, the lane pose values were buffered and averaged over the 5 past values. This reduced the noise by a lot without introducing too much delay into the system since the lane filter node publishes the lane pose 30 times per second and our DQN agent tries to run and execute actions at 5 Hz. Another improvement was done by introducing bright lights into the test environment. Adding proper lighting increased the detection rate of the yellow center lines in the lane. When using the ground projection node, which can also be found in the dt-core repository, it was apparent that the detection distance of the yellow lines went from about 5-15 centimeters to approximately 30-40 centimeters in front of the robot. This vastly increased the accuracy of the lane pose estimation as well since more line segments were used to estimate the lane pose.

In addition to the lane pose, the time step delta t (dt), which shows how long each chosen action is executed, is added to the state. This time step value is only being used by the linear regression model, which estimates the future pose of a selected action in a given state, and not by the DQN agent. This time step also stays fixed throughout the experiments but was included in the state as it was changed during development and can be adjusted for different situations. Our state-space, therefore, is continuous as the distance d and angle phi are continuous values.

As mentioned before, in early development, Q-learning was used instead of DQN but this was later changed to DQN as it improves the learning for the robot a lot. Since DQN works with a discrete action space, 100 random actions are generated at the start of training so that the robot has more options to choose from before either all actions are unsafe or the selected action has to be optimized because it would result in an unsafe state. This made the selection and learning process better as the robot was able to find safe options faster and did not have to recover from unsafe states with actions that could not be used to train the network as they were not in the action list that was generated at the start of training.

The actions in the implementation are two-dimensional. They consist of the linear velocity and the angular velocity in the robot's reference frame. These actions are generated with a modifiable range. The main reason for that is that in Duckietown Duckiebots can end up having different gain values for the maximum speed that the robot will execute. These gain values are being set during the calibration of the robot and can vary in the range from 0 to 1. One problem during development was that the robot cannot execute very small velocities as the motors are not receiving enough power to spin in these cases. Another problem is that large velocities made the robot inconsistent and overshoot especially if the rotational velocities were too high.

With the adjusted and lane pose the reward for the reinforcement learning agent is being evaluated with the following reward function:

$$reward = 1 - 3 * lane\_d^2 - lane\_\varphi^2 \tag{3.2}$$

The robot, therefore, is being rewarded more for being closer to the center and parallel to the lane. The distance d and the angle phi are multiplied by different amounts as their range of safety is different. Since the lanes in Duckietown are 22 centimeters wide, the range of safety is from -11cm to 11cm. The actual values in the implementation are in meters. Therefore the range looks as follows $range\_d = [-0.11, 0.11]$. Since these values are measured from the center of the robot and the robot has a wheel span of 12.5 cm, The adjusted safety range is $range\_d = [-0.05, 0.05]$. The range of the angle phi is in radians and therefore the selected safety range is $\varphi = [-\frac{\pi}{2}, \frac{\pi}{2}]$. Therefore the safety range for the angle was defined as $\varphi = [-0.8, 0.8]$

To ensure the safety of the robot a safety layer was implemented between the selection of an action and the execution of the action. This safety layer predicts the future lane pose of a selected action in the current state and with the current time step dt and then checks whether the predicted lane pose is within the safety range mentioned earlier. The prediction of the future state is done with a data-driven linear regression model. This model was trained by generating random actions in a similar way to the DQN agent. However when collecting data the robot continuously generates a new action for the next movement and also changes the time step to random duration within an adjustable range. During this, the robot will drive around and will save the lane pose before the action was taken, the action velocities itself, the duration of the action, and the actual resulting lane pose. For this part, it is important to run this data collection on the robot itself to remove any network delay from when the command is being sent and when the robot receives and executes the commands. It is also important to note that the robot stops after the execution of each action since this removes the influence of inertia from the previous action of the robot onto the current action.

Algorithm 3.1 Collect Data

```
procedure COLLECT_DATA
    while not data_collector_node.is_shutdown do
        original_state ← state_callback()
        new_v ← np.random.uniform(min_v, max_v)
        new_omega ← np.random.uniform(-max_omega, max_omega)
        velocities ← [new_v, new_omega]
        execute_action(velocities)
        time.sleep(time_step)
        execute_action([0, 0])
        current_state ← state_callback()
        action ← [original_state, velocities, time_step, current_state]
        // Saving data to file
        file.save(action)
```

After the data was saved to a file, the linear regression model was trained. The linear regression implementation used here is the standard scikit-learn implementation[10]. After training the model was then moved into the package of the DQN agent so that it is included in the Docker environment when the code is built.

During the development, a control systems approach was also tested. This approach aimed to estimate the future state by calculating a forward kinematics model. However, because of multiple unknown multiplication factors and the non-perfect execution of velocities due to inertia and unknown and unpredictable acceleration, the data-driven linear regression model seemed to have better accuracy at predicting the future state.

As mentioned before the safety layer sits in between the action selection and action execution. For this, it follows a similar approach to the safety layer in the paper "Safe Reinforcement Learning Using Black-Box Reachability Analysis"[11]. The agent has different approaches to unsafe action depending on whether they were selected randomly or by a learned decision. The pseudo-code looks as follows:

Algorithm 3.2 act

```
function ACT(self, state)
    if np.random.rand() $\leq$ self.epsilon then
        $safe$ ← False
        $action\_list$ ← self.actions.copy()
        while not $safe$ do
            $action$ ← random.randrange(len(action_list))
            $predicted\_state$ ← self.predict_state_lr($state$, self.actions[action])
            $safe$ ← self.safety_layer.check_safety($predicted\_state$)
            $action\_list$ ← action_list.remove(action_list[action])
            if $safe$ then
                Output "Random action"
                return $action$
            if action_list = [] or action_list = None then
                return self.get_back_to_safety($state$)
    $q\_values$ ← self.model.predict($state$, verbose = 0)
    $action\_list$ ← self.actions.copy()
    // sorting actions by q_values from high to low
    $action\_list$ ← [$x$ for _, $x$ in sorted(zip($q\_values$[0], $action\_list$), reverse = True)]
    $action\_list$ ← action_list[:5]
    for action in action_list do
        $predicted\_state$ ← self.predict_state_lr($state$, action)
        $safe$ ← self.safety_layer.check_safety($predicted\_state$)
        if $safe$ then
            Output "Learned Action"
            return self.actions.index($action$)
        else
            self.iter ← 0
            $action$ ← self.optimize($state$, action)
            if action is not None then
                Output "Optimized Action"
                return $action$
    // If no safe action was found, we try to recover to a safe state
    Output "Using inverse model to get back to safety"
    return self.get_back_to_safety($state$)
```

Which results in the following process diagram:

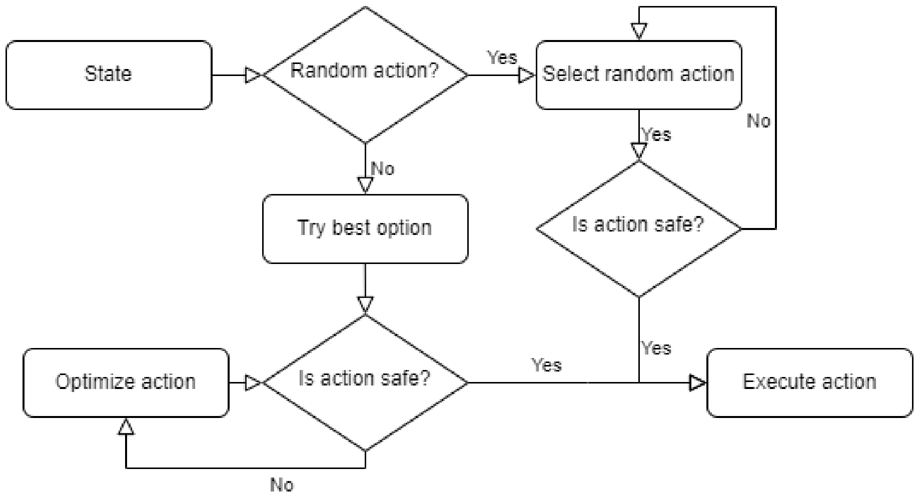

Figure 2: Process diagram of action selection and safety layer

First, the DQN agent decides if it is selecting to do a random action or select an action based on its learned knowledge. This randomness factor ε gets smaller over time each time the neural network is trained on the memory of executed actions. However, it is important to note that this randomness factor will never reach zero as it would prevent the agent from any further learning by exploring new policies. In that case, the DQN agent would not be able to adapt and learn from new environments which would result in just executing learned policies from previous training.

After the agent selected an action, the safety layer tests whether the selected action will result in a safe state. If it results in a safe state, the action will be executed as planned. The following diagram illustrates how the robot considers which actions are safe and which actions are unsafe.

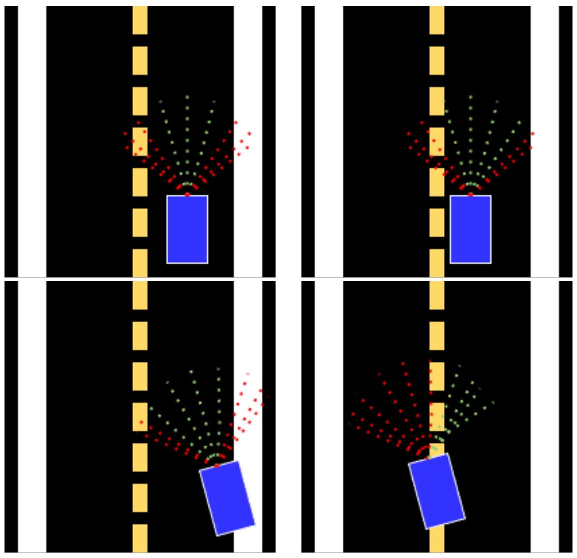
Figure 3: Duckiebot in different lane positions

In figure 3, red lines are the actions that are considered unsafe as they do not result in within the safe range that was defined earlier. Green lines represent safe actions that the robot can safely execute. The behavior of the agent is different depending on whether a random action was selected or an action based on the learned policies. If the agent selects to choose an action based on its learned model, the five actions with the highest Q-values are returned. They are then sorted from highest to lowest Q-value. Then the highest action will be selected and the safety layer checks if the action will result in a safe state. If it does not result in a safe state, the agent tries to optimize the action using the following recursive algorithm:

Algorithm 3.3 Optimize Action

```
function OPTIMIZE(self, state, action)
    if self.iter $>$ 3 then
        return None
    $new\_v \leftarrow$ action[0]$/2$
    $new\_omega \leftarrow$ action[1]$/2$
    $new\_action \leftarrow [new\_v, new\_omega]$
    $new\_state \leftarrow$ self.predict_state_lr$(state, new\_action)$
    Output "Predicted state: ", new_state, " with action: ", new_action, ""
    // checking if new state is safe
    $safe \leftarrow$ self.safety_layer.check_safety$(new\_state)$
    if $safe$ then
        return $new\_action$
    else
        Output "New action  is unsafe, trying again".format($new\_action$)
        self.iter $\leftarrow$ self.iter $+ 1$
        return self.optimize$(state, new\_action)$
```

If the option cannot be optimized after four iterations, the next best action of the five sorted options will be tested for safety. In case no safe optimization can be found for the five actions with the highest Q-Values, the robot assumes that it currently is in an unsafe state already. In this case, the get_back_to_safety(state) is being executed and the robot tries to recover to a safe state so that training can continue normally. It is important to note here that the recovery function is not included in diagram 2.

If the robot selects a random action instead, all actions are put into an action list. Then a random action will be selected from the list. If that action is safe, the robot executes the action normally and receives the reward for it. If the selected action is unsafe, it will be removed from the action list and another random action will be drawn. If all 100 actions are unsafe, the agent assumes it is in an unsafe state and needs to recover to a safe state. For this, the get_back_to_safety function will be executed. This recovery procedure looks like this:

Algorithm 3.4 Get Back To Safety

```
 1: function GET_BACK_TO_SAFETY(state)
 2:     dt ← self.sleep_time
 3:     // Getting back to a safe state
 4:     // If we are angled away from the lane
 5:     if (state[1] > 0 and state[0] > 0) or (state[1] < 0 and state[0] < 0) then
 6:         // Turning towards lane
 7:         theta ← -1.5 * state[1] / dt
 8:         v ← 0
 9:     else
10:         theta ← state[1] / dt
11:         v ← abs(state[0]) * np.sin(state[1]) * dt
12:     // Getting new action
13:     new_action ← [v, theta]
14:     OUTPUT "Using inverse model to get back to safety"
15:     return new_action
```

The function considers the robot to be in two different states. It is either outside of the lane and turned away from the center of the lane or outside of the lane and turned towards the center of the lane. The following diagrams [4][5] illustrate these two states and their way of recovery

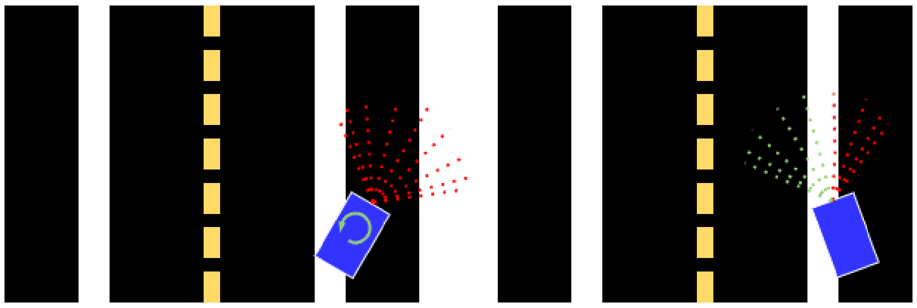
Figure 4: First possible recovery state

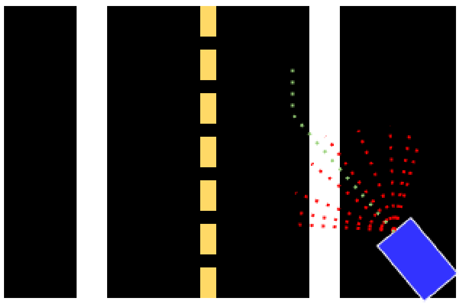
Figure 5: Second possible recovery state

If the robot is outside of the lane and turned away from the lane's center as can be seen in diagram 4, The robot turns towards the center of the lane and either a safe action can be found as before or the robot will result in the second recovery state in which it is outside of the lane and turned towards the center of the lane. This can be seen in diagram 5. In this case, the linear velocity will be adjusted with the time step length in order to return to a safe location. Additionally, the rotational velocity will be adjusted so that the robot will result in a parallel angle towards the centerline of the lane. Since the math might not be fully accurate for this because of unknown internal gains and multiplication factors, multiple iterations might be needed.

For the DQN agent's training, the robot comes to a full stop and starts training on a subset of the saved actions that the robot has experienced and saved. For this, the robot keeps track of some statistics. The first statistic is the total reward that the agent earned from executing actions in the current episode. If a total of 50 reward points are reached, the episode is ended. Additionally, the episode also ends if the total reward goes below 0. The other main statistic that the robot keeps track of is the step number. This is the number of actions that have been executed since the last time the robot was trained. Here it is important to note that recovery actions are also considered in the steps but they will not be added to the robot’s memory as they contain actions that are not in the randomly pre-generated list of actions. The next statistic is the minibatch size. The minibatch size determines the number of actions the agent randomly selects from its memory to train the neural network. This number grows over time as the robot has more saved actions to learn from. Additionally, this number also is influenced by the number of steps that have been taken within the episode. This is especially important for the beginning of training as the agent has to make sure that it only tries to train on as many actions have been saved to the agent’s memory yet.

Training happens if any of the three conditions are met. The first condition is that the agent received a total reward of 50 in the current episode which ends the episode and starts training. The robot is not moved during the training so it starts in the same location that it left off but the total reward and the number of steps are reset to 0. The minibatch size is not reset in that case. The second condition that could trigger the robot to train its neural network is that the total reward got below 0. This may happen if the robot continuously takes action that resulted in very far distances or large angles towards the lane’s center. Here it can be assumed that these actions were either unsafe or can be considered bad or less desirable. In this case, the episode is also ended as the robot will either continue to lose more total reward or take a long time to get back to the desired total reward of 50. While it is not intended for the robot to be moved during training after such an episode, in the case of the reinforcement learning agent without the safety layer, it is common for the robot to be placed back inside the lane if it went into unsafe locations and stopped there to train its model. The third condition for the robot to start training is after a certain number of steps is reached. This value grows through the experiment and is mainly there to stop indefinitely executing random actions at the beginning of training as there the total reward may be growing very slowly. Additionally, if the training was triggered by this condition instead of reaching the total reward goal, the amount of action taken before ending the episode can be compared. If fewer actions are needed to end an episode with the 50 reward goal, the actions that the agent selected were conceptually better compared to the actions taken by a robot that took more actions to complete an episode as each separate action got a higher reward on average compared to the actions that took longer to complete the episode.

4 Evaluation of the Investigation

To evaluate the performance of the proposed approach to other approaches, multiple indicators were selected. The main factor is the safety rate itself which describes how many of the chosen actions resulted in an unsafe state after execution. This will give us a clear indication of whether the proposed approach with a safety layer is safer than just a regular reinforcement learning agent. A higher safety rate means that the robot stayed within the safe range more than a lower safety rate.

Additionally to the safety rate, the recovery rate was also considered. The recovery rate describes how well the robot managed to recover to a safe state by itself. This rate can unfortunately not be tracked quantitatively by the robot itself as it cannot detect whether or not it had help from outside factors during execution time. However, this can very easily be evaluated by the user by observing the robot during its training. Quantitatively this means that if the robot is in an unsafe state but recovers from that state by itself, the recovery rate for that action would be 100%. If it would not be able to recover by itself the next time it would be in an unsafe state, the recovery rate would be changed to 50%.

It is also important to note that the environment also plays a big role in this factor as the robot might not physically be able to recover to a safe state if any of the wheels or the ball bearing in the back of the robot get caught on the outside part of the environment. During testing, it was observed that the robot with the safety layer would go outside of the lane and would then start the recovery process by turning towards the lane. However, by turning, the ball bearing got caught on the edge of the foam tiles that the Duckietown roads consist of. If there would have been another foam tile connected to the current lane foam tile, the robot would most like have recovered to within the lane by itself. During other tests with a different environment which consisted of a single straight road with foam tiles attached on both sides, the robot did not run into these problems.

To evaluate the safety performance, the environment was also kept the same during the training of both agents. A simple square of nine tiles was selected as the testing ground for both agents. The road was going around in a circle on the outside edges of the three-by-three square. The center tile was kept empty but was filled with a foam tile in order to increase the detection rate for the robot's visual pipeline and also to enable easier recovery as an empty square without a foam tile makes it hard to recover as mentioned above. It also is important to mention that the intersection tiles from the Duckietown road standard were not used in the testing of this project as they do not have any lane markings on them and this implementation relies on the constant data input of the lane pose which depends on both the yellow and white lines.

During testing, it was also noticed that the developed method for collecting quantitative data for the safety rate did not work out to be useful as the robot would leave the lane and the lane markings would end up out of the robot's camera view which would result in a wrong lane pose estimation. These wrong lane poses were often within the safety range of the robot. This was mostly a problem appearing with the reinforcement learning agent without the safety layer. The implementation with the safety layer would recover to a safe state or a state where it would see the lane markings again in most cases before losing track of its actual position. It should however be noted that this was not always the case and there were situations where this behavior appeared in the testing of the agent with the safety layer enabled.

Another performance factor that can be evaluated when observing the robot during training is the number of actions taken for each episode. Unfortunately, this factor is tied to the robot's training itself by design. This number should decrease over time as a smaller amount of random actions will be selected on average over time compared to the beginning of training. Therefore a direct number comparison between the two different approaches could not be done. This factor could be reworked in the future however to make it better comparable with automatically collected data from the robot itself. For example, it could be extended to show the average number of actions taken per episode at episode milestones like episodes 0, 5, 10, etc. This would make it a useful indicator to observe if the agent with the safety layer actually is learning faster since it learns to avoid unsafe actions. In the current form, however, it can already be used to compare whether training was triggered by finishing the episode by reaching the total reward goal or by reaching a maximum step limit. Unfortunately, the current implementation of the project does not include any data collection for this purpose, however, it can be observed by watching the robot's output closely before training is started.

To compare the different approaches both agent's training was started in episode 0 without any previous knowledge. They were both put into a straight-lane tile to start their training. Both of the robots were recorded during their training for one full lap of the road circle. A lot of things were observed during training. The videos of the robot were compared against each other to draw conclusions from them.

The most prominent observation was that there is a clear safety rate difference between the two approaches in this investigation. The agent without the safety layer reached the corners and continued to go straight or outside the lane. This resulted in the robot having to be manually returned to a safe location. This happened on all four corners of the track. The agent with the safety layer enabled on the other hand might have gone slightly outside of the lane but it mostly recovered by itself. It also should be noted that the robot recognized the unsafe state very fast and recovered immediately. Therefore the severity of taking the unsafe action was not as severe as with the agent without the safety layer. Therefore, during visual observation of the robot during training, it was only apparent to leave the lane once. There might have been other times when the robot left the safe range but it was not clearly apparent and with an adjusted and smaller safety range, the robot would have recovered before entering the unsafe area. This means that in future tests the agent could be calibrated to stay in a smaller space and therefore stay even safer than in this test.

Additionally, the recovery rate of the robot without the safety layer was 0% as it did not recover by itself a single time. The robot with the safety layer on the other hand managed to recover by itself every time except for one time when it went outside of the lane in a corner. After further evaluation, it was found that this was most likely due to the road markings being outside of the camera's view. From this, it was concluded that the safety layer may not be keeping the robot perfectly safe in every situation but the safety was greatly improved and the number of manual recoveries being done by the user were greatly reduced.

During the experiment, it was also apparent that in later episodes the robot with the safety layer enabled was able to finish full episodes in one run without being interrupted by training due to reaching the step limit. This was not the case for the robot without the safety layer. Here the robot would start training in the middle of the episode as the overall reward per action was lower.

Another observation was that having the safety layer enabled did not seem to impact the execution time of each timestep too much. This essentially means that there is no runtime performance disadvantage compared to running without the safety layer enabled.

Additionally, to these factors, other factors were observed that could be useful for further improvement of this project. The first observation was that the agent with the safety layer mostly had trouble recovering when the lane markings went out of the camera's field of view. To improve this, the safety range can be adjusted in future tests to limit the robot's safety range from the full lane to only the area where the robot can still see enough lane markings to still be able to accurately know its location and recover to a safe state. Additionally, it was also observed that the robot was having trouble with executing small velocities. This was most likely due to the motors not having enough torque at those speeds to accelerate the robot from a full stop. Similarly, too-high velocities appeared to be inconsistent and the robot tended to overshoot especially on rotations. Furthermore, these fast velocities made the camera image blurry which resulted in problems in the image pipeline trying to detect the lines in order to estimate the lane pose. Another observation was that the linear regression model used to predict the future lane pose from the current state and the selected action was not perfect either. This could be due to multiple factors. In further exploration of this topic, this model could be exchanged with other approaches like another neural network trained on similar random data. Another extension could be to continuously improve this trained model during the training of the reinforcement learning agent. Since the agent is already remembering the previous state, action, and reward, it could be extended to save the resulting lane pose as well and to train the safety layer model on this data during training time. Another possible improvement for the agent could also be to give more input data to the DQN agent. This is easily adjustable since the DQN agent can handle more input data without resulting in a large performance hit or large memory increase.

5 Conclusions

Based on the results of this project, it can be concluded that there is no disadvantage to using a safety layer when doing reinforcement learning since execution time is very similar. Moreover, the dramatically improved safety of the vehicle is helpful for the robot's training as fewer actions with lower or even negative rewards will be executed. Because of this, reinforcement learning agents with safety layers learn faster and reduce the number of unsafe actions that are being executed. Unfortunately, manual observation and intervention by the user were still necessary, however, the frequency was clearly reduced which further improved learning as the robots in testing did not know if an outside intervention was done which could result in an action being rewarded incorrectly. It was also concluded that this project did not reach perfect safety with the implementation. Therefore a fully autonomous reinforcement learning training without any human intervention has not yet been achieved. A lot of improvement factors have been found that can further improve the safety and recovery rate. Additionally, some major problems which are not direct results of the reinforcement learning or safety layer have been identified. These problems could be attempted to be fixed in different ways like improving the open source implementations of lane filter nodes[8] or adding more sensors or cameras to the robot in order to extend the input data to the agent. Another area that was untouched during the research of this project was other vehicles inside the current lane. The safety layer could potentially be extended to also include safety features that should keep the robot safe from hitting other vehicles.

References

[1] Luc Le Mero et al. “A Survey on Imitation Learning Techniques for End-to-End Autonomous Vehicles”. In: *IEEE Transactions on Intelligent Transportation Systems* 23.9 (2022), pp. 14128–14147. DOI: 10.1109/TITS.2022.3144867.

[2] Duckietown. *Front Page*. https://www.duckietown.org/. Nov. 2021.

[3] Janst1000. *Safe-RL-Duckietown*. https://github.com/Janst1000/Safe-RL-Duckietown. Accessed: May 2023. 2021.

[4] Duckietown. *dt-core (Version daffy) [Source code]*. https://github.com/duckietown/dt-core. Apr. 2023.

[5] C. J. C. H. Watkins. “Learning from delayed rewards”. Doctoral dissertation. University of Cambridge, 1989.

[6] V. Mnih et al. “Playing Atari with Deep Reinforcement Learning”. In: *arXiv preprint arXiv:1312.5602* (2013).

[7] Duckietown. *dt-ros-commons*. https://github.com/duckietown/dt-ros-commons. Accessed: April 2023. 2021.

[8] Duckietown. *Lane filter*. Duckietown dt-core. Retrieved May 12, 2023, from https://github.com/duckietown/dt-core/tree/daffy/packages/lane_filter.

[9] Duckietown. *dt-core: Code that runs the core stack on the Duckiebot in ROS*. https://github.com/duckietown/dt-core. 2023.

[10] scikit-learn developers. *LinearRegression - scikit-learn 1.2.2 documentation*. https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LinearRegression.html. [Accessed 12 May 2023]. 2022.

[11] M. Selim et al. “Safe Reinforcement Learning Using Black-Box Reachability Analysis”. In: *IEEE Robotics and Automation Letters* 7.4 (2022), pp. 10665–10672. DOI: 10.1109/LRA.2022.3192205.