Deployment of a Safe Reinforcement Learning Agent through Environmental Robustness Testing

Daniel Smetankin*,Mehrdad Moradi*
and Joachim Denil*
*Faculty of Applied Engineering - Department of Electronics-ICT
University of Antwerp, Antwerp, Belgium

***Abstract*—Reinforcement learning (RL) has gained interest as a artificial intelligence method in recent years. Its potential to solve complex classification and control problems has sparked increasing interest, particularly in the field of autonomous driving. Despite its growing popularity, RL adoption in real-world applications remains limited. This is largely due to the fact that existing RL algorithms are incapable of ensuring safety in new or untested scenarios. This study presents a method for maintaining safety constraints that entails subjecting the agent to a series of randomly generated scenarios throughout the testing phase. In each scenario, the agent's robustness is tested until it reaches its failure point, which is defined as a violation of the safety constraints. In the event of failure, the agent undergoes a process of retraining, wherein we utilize hyperparameter tuning and reward shaping. The ultimate objective is to enable the agent to learn a new policy that effectively handles the scenario in which it previously failed. Through a trial and error approach, our aim is to discover an optimal reward function and determine the appropriate hyperparameter settings that facilitate the learning of a safe policy by an RL agent. This safe policy should enable the agent to effectively navigate and succeed in multiple randomly generated scenarios. Our case study revolves around the implementation of a Deep Deterministic Policy Gradient (DDPG) RL agent with the purpose of managing the acceleration control in an Adaptive Cruise Control (ACC) system. The primary function of an ACC system is to automatically adapt the vehicle's speed in order to uphold a safe distance from vehicles ahead. Once we have validated the development of a safe agent using our proposed method through testing in a simulated environment, the next step will be to assess whether this safe agent also performs reliably in real-world conditions. To accomplish this, we will convert the learned policy into C++ code and deploy it onto a TRAXXAS 4-TEC 2.0 VXL RC car. After analyzing the results, we found that the robustness-tested agent consistently adhered to safety constraints in challenging situations, while the non-tested agent occasionally failed to meet safety requirements during both simulation and deployment tests.**

I. Introduction

Artificial Intelligence and machine learning often get used as synonyms. But these are two different fields. Artificial Intelligence (AI) refers to the capacity of a machine to exhibit human-like capabilities such as reasoning, learning, planning, and creativity [1]. Machine learning is a sub-field of Artificial Intelligence. Which refers to the technology and algorithms that enable systems to recognize patterns, make judgments and improve through experience [2]. One interesting machine learning approach is reinforcement learning, which is used to solve an array of problems one of which is the development of self-driving cars. Essentially, in reinforcement learning, the agent interacts with the environment by taking actions, which subsequently lead to consequences. These consequences are observed and represented through observations (See Fig. 1). To determine the desirability of these consequences, a reward function is employed, where positive rewards signify favorable outcomes, and negative rewards indicate unfavorable ones. In RL, the agent's objective is to learn a policy that maximizes the net long-term reward by making appropriate actions in different states [3]. An RL agent consists of a policy and a RL algorithm. The policy represents the mapping from possible states to possible actions. The RL algorithm plays a crucial role in shaping the learning process by defining the method through which the agent learns. It determines how the agent updates and refines its policy based on the observed rewards and the consequences of its actions. RL algorithms learn by trial and error, similarly to how children explore their surroundings and learn to perform actions that lead to a desired outcome [4].

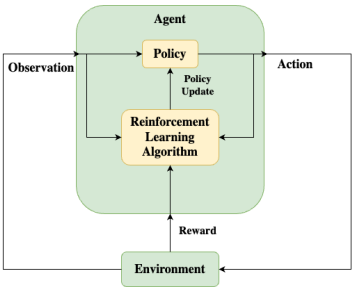

Fig. 1: Reinforcement learning diagram

RL has demonstrated remarkable success in tackling complex tasks in simulated environments [5]. However, when transferring these algorithms to the real world, challenges such as partial observability, non-stationarity, real-time inference, system delays, and satisfying safety constraints can significantly reduce their performance and efficiency. In this study, we will focus specifically on satisfying safety constraints to enable safe and reliable operation of the RL agent in real-world deployment. Safe RL can be defined as the process of learning policies that maximize the expected return in problem domains where ensuring adequate system performance and/or complying with safety constraints is of utmost importance throughout the learning and deployment stages

[6]. Ensuring the safety of physical systems is paramount to their effective control since these systems can potentially cause damage or harm to themselves or their environment. This safety consideration is critical not only during system operation but also during exploratory learning phases. Such safety considerations can be made with respect to the system (limiting system temperatures, contact forces, or maintaining minimum battery levels) or with respect to the environment (avoiding obstacles, limiting end effector velocities) [7]. The research question we aim to answer is twofold: Can the safety of a reinforcement learning (RL) agent be ensured through environmental robustness testing? Additionally, once a safe RL agent has been successfully developed, how effectively will it adhere to safety constraints in real world environments? Important to note is that the environmental robustness testing method chosen for this study is a random scenario generator. Nevertheless, it should be acknowledged that there exist alternative, more advanced approaches that have the potential to produce improved results.

To ensure the preservation of safety constraints across various scenarios, we present a method that continuously subjects the RL algorithm to robust testing. This involves exposing the algorithm to randomly generated scenarios until it reaches a point of failure. Failure is defined as a violation of the safety constraints. When such failure occurs, the RL agent will be retrained with the help of reward shaping and hyperparameter tuning. Through continuous retraining, testing different reward functions, and exploring various hyperparameter settings, our goal is to develop an agent capable of learning a policy that can perform effectively not just in the specific training scenario, but also in diverse and unfamiliar scenarios. The proposed method involves several steps. Initially, we will train the agent using Matlab Simulink. Once the agent is trained, we will assess its performance by testing it across various randomly generated scenarios. Our objective is to identify an agent that can successfully navigate through 100 randomly generated scenarios. Once such an agent is found, we will extract its greedy policy and convert it to C++ code. The resulting C++ code will be deployed on a RC-car with the help of a NVDIA-jetson controller, which has the Robot Operating System (ROS) installed. ROS is utilized to handle sensor data input and motor control. The case study chosen for this research focuses on an Adaptive Cruise Control (ACC) system. Which goes beyond conventional cruise control by dynamically adjusting the vehicle's speed based on the movements of the preceding vehicle. The agent will assume the crucial role of managing the acceleration control component within the ACC system. The contribution of this research is to provide an approach that is both straightforward and effective for evaluating the safety of a trained RL agent and gaining an understanding of its safety performance during deployment. Additionally, we present a comprehensive method that outlines the process of training an RL agent in Matlab Simulink, testing it within the Simulink environment, and subsequently transforming the Simulink-based agent into C++ code for deployment on a physical RC car.

II. Related Work

Respecting safety constraints in the development and deployment of autonomous systems, such as self-driving cars, is closely related to the concept of Safety of the Intended Functionality (SOTIF). SOTIF focuses on ensuring that the intended functionality of the system does not lead to unreasonable risks or hazards due to functional limitations or foreseeable misuse [8]. SOTIF serves as a guiding framework for automotive engineering teams, providing recommendations for design, verification, and validation practices. Unlike traditional functional safety, which primarily addresses risks stemming from system failures, SOTIF focuses on assessing the ability to ensure necessary safety functionalities in unfamiliar conditions without any failures occurring. This involves considering various factors, such as the performance limitations of car components like sensors and systems, as well as unexpected variations in the road environment. To adhere to SOTIF requirements, automakers must conduct extensive simulations and employ machine learning and AI techniques to analyze vast amounts of data, enabling them to anticipate how vehicles will respond to intricate real-world scenarios [9]. In essence, our approach aligns with the objectives of SOTIF by emphasizing the importance of testing and retraining to improve the agent's ability to respect safety constraints in a variety of real-world scenarios.

Extensive research has been conducted on safe reinforcement learning. For instance, one notable example includes the development of safe exploration. The objective of safe exploration is to uphold zero-constraint violations throughout the entire learning process, accomplished by incorporating a safety layer into the policy. Studies have demonstrated that the employment of a safety layer approach has shown to improve performance in terms of reward, facilitate more efficient exploration by guiding the exploratory actions towards feasible policies, and most importantly, maintain safety within continuous action spaces [10]. Shielding is a method commonly associated with safe exploration. This approach introduces a shield, which acts as a reactive system that monitors the actions taken by the learning agent. The shield intervenes and corrects the agent's actions only when they result in a violation of the specified safety constraints. By employing this shield, the learning agent can operate safely within the given environment while still learning and adapting to optimize its performance (See Fig. 2) [11].

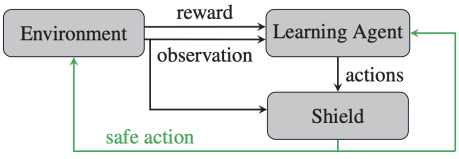

Fig. 2: Shielded reinforcement learning

Another method employed to maintain safety within Reinforcement Learning is Parallel Constrained Policy Optimiza-

tion (PCPO). PCPO extends the traditional actor-critic architecture to a three-component learning framework, incorporating neural networks to approximate the policy function, value function, and a newly added risk function. PCPO algorithms have been shown to guarantee safety constraints during learning for various autonomous driving tasks, offering benefits such as higher learning speed, improved data efficiency, and prevention of the agent from getting stuck at sub-optimal policies or, at the very least, a safe sub-optimal policy [12]. These methods align with our goal of developing a safe RL agent, as we share the ambition of prioritizing safety. One way we aim to achieve this is by terminating a learning episode if a safety constraint is violated. While it might be a simple approach, incorporating safety considerations into the RL framework can be highly valuable. By taking such precautions, we can enhance the safety of the learning process and promote the development of a reliable and secure RL agent. A key difference between our method and the safe exploration methods, is that we don't focus on ensuring safety during training but rather validate safety after the RL agent has been trained.

Efforts have also been made to overcome the 'reality-gap' between simulations and the real world in RL. One such method is domain randomization, which is a simple but promising technique for addressing this gap. Instead of training a model on a single simulated environment, the simulator is randomized to expose the model to a diverse range of environments during training. The objective of domain randomization is to introduce sufficient simulated variability during training time such that at deployment the model is able to generalize to real-world data. This approach has not only proven effective in simulations but also in real-world scenarios, as successful deployment tests have demonstrated the model's ability to generalize to real-world data during testing [13]. Our approach involves robustly testing our agent in various randomly generated simulated scenarios until failure is encountered. Subsequently, we retrain our agent to handle each scenario it previously struggled with. This approach bears similarities to domain randomization, but with a key distinction. While domain randomization randomizes scenarios during training, we introduce randomization during the test phase, aiming to assess and improve the agent's performance in diverse scenarios.

III. Case Study

This research will utilize the case study of Adaptive Cruise Control (ACC). The primary objective of an ACC system is to regulate the speed of the controlled vehicle based on the velocity of the leading vehicle, while maintaining a safe distance between them. To model an ACC system for a car, we employed the Simulink example of 'ACC with sensor fusion' (See Fig. 3). This example model incorporates sensor fusion between radar and camera technologies to detect objects ahead. It combines the information from both sensors to determine the relative distance between the vehicles. To determine the acceleration output, this model utilizes the relative velocity, relative distance, and longitudinal velocity as inputs as can been seen in (See Fig. 4). The acceleration output itself is a continuous value ranging from -3 to 2 $\mathrm{m/s^2}$. The safe distance is also incorporated into to ACC controller, but it is calculated rather than being implemented as an input. Relative velocity is the difference in velocity between two vehicles. Longitudinal velocity specifically refers to the velocity or speed of an object in the direction of its motion. In the ACC with sensor fusion example, a bicycle model is also integrated. The bicycle model is a simplified mathematical representation utilized to depict the motion and behavior of a car. In order to accurately model the acceleration and braking of an RC car, we incorporate parameters such as the total vehicle mass (3kg) and the longitudinal distance from the center of gravity to the front (16cm) and rear tires (15cm) into the Bicycle car model. By including these parameters, we can create a more precise depiction of the RC car's movement. Due to the relatively low mass of an RC car, it can accelerate and brake more quickly compared to a regular car. Accounting for these factors in our simulation helps us achieve a higher level of accuracy, aligning our model with the real-world behavior of an RC car.

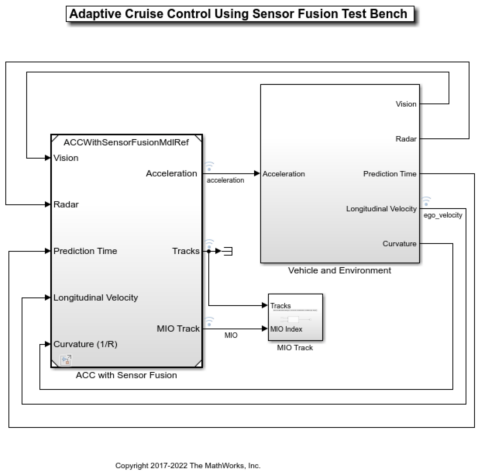

Fig. 3: ACC with sensor fusion Simulink Model

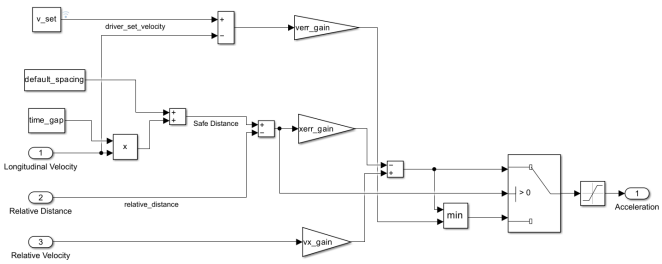

Fig. 4: Classic ACC controller

v_{set} : initial set velocity (m/s)
d_{default} : the distance spacing between 2 cars (m)
t_{gap} : the time gap between 2 cars (s)
K_{verr} : ACC velocity error gain (N/A)
K_{xerr} : ACC spacing error gain (N/A)
K_{vx} : ACC relative velocity gain (N/A)

IV. Method

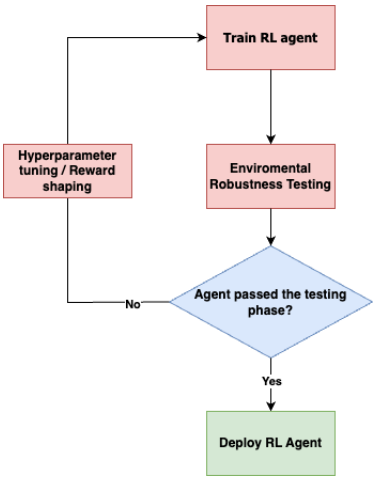

Fig. 5: Method Flowchart

To ensure the creation of a safe RL agent, we employ a specific method (See Fig. 5). The process begins by training an RL agent to perform a specific task. Once the agent is successfully trained, it undergoes a series of environmental robustness tests, exposing it to scenarios or environments it hasn't encountered during training. If the agent passes these tests, we proceed to the deployment phase, where we generate a greedy policy based on the trained RL agent. The purpose of using a greedy policy is to prevent the agent from exploring further and instead focus on taking actions that maximize immediate returns. However, if the agent fails the robustness testing phase, it signifies that it struggled in a particular scenario or environment. In such cases, we save that scenario or environment and retrain the agent by employing techniques such as hyperparameter tuning and reward shaping, aiming to equip it to handle that specific situation. Through this iterative process, our goal is to discover an appropriate reward function and effective hyperparameters that enable the agent to learn a safe policy. This policy should possess the capability to generalize across multiple diverse scenarios that the agent has never encountered or been explicitly trained on. In the following subsections it will be illustrated how this method is implemented in our particular case study, which focuses on training an RL agent to manage the acceleration of an ACC system.

A. Reinforcement Learning

The first step in creating an RL (Reinforcement Learning) agent is to choose an RL algorithm from a variety of options available, such as SAC (Soft Actor-Critic), PPO (Proximal Policy Optimization) and DDPG (Deep Deterministic Policy Gradient) among others. The DDPG (Deep Deterministic Policy Gradient) algorithm was selected as the implementation for reinforcement learning due to its suitability for continuous action spaces, such as acceleration. DDPG has commonly been chosen as the preferred algorithm in autonomous driving tasks [14]. It combines the benefits of the deterministic policy gradient algorithm, actor-critic methods, and deep Q-networks. The DDPG algorithm is a model-free, off-policy reinforcement learning method that use the actor-critic architecture. Model-free RL algorithms do not explicitly focus on constructing or relying on an explicit model of the environment. Instead, they learn directly from experience and observations without modeling the underlying dynamics. In contrast, model-based RL algorithms specifically aim to construct an explicit model of the environment, capturing its transition dynamics and other properties. Off-policy RL algorithms learn from experiences generated by different policies, while on-policy RL algorithms learn from experiences generated by the same policy being updated. As mentioned earlier the RL DDPG agent assumes control over the acceleration of the RC-car. It relies on 4 observations signals, the safe distance, relative distance, relative velocity, and its own velocity, to make informed decisions regarding acceleration.

B. Generating Scenario's

To customize the scenario, the Matlab Driving Scenario Designer app was used. This app allows users to create and modify driving scenarios, such as changing the road geometry, adding other vehicles and adjusting vehicles trajectories. In our driving scenario, the main elements remain consistent: our car and one lead car. However, the specific dynamics of the lead car, such as its accelerations, velocities, and path, are randomly generated. To automate the generation process, a MATLAB script is employed that modifies the dynamics of the lead vehicle within the scenario file created using the Driving Scenario Designer. This script generates 10 points along a road, with each point having randomly generated x and y coordinates within the road's boundaries. Additionally, a velocity between 0 and 20 m/s is generated for each point. The points are then sorted based on their x positions, ensuring that the trajectory remains straight ahead without our car reversing towards us. By incorporating this randomization process, we introduce variability into the driving scenarios. It is worth emphasizing that while we opted for random scenario generation, there are alternative techniques available to generate challenging scenarios more effectively.

C. Deployment

Once our reinforcement learning (RL) agent successfully completes the environmental robustness testing, meaning it passed 100 randomly generated sceanrios in a row, we generate

a greedy policy based on that trained RL agent. This greedy policy is generated as a single Simulink block. To obtain the implementation in C++, we utilize the MATLAB Embedded Coder app, which is capable of generating C++ code from Simulink blocks or MATLAB code. The resulting C++ code incorporates the logic of the greedy policy, defining how specific observation signal values are mapped to corresponding action values. The resulting C++ code will be deployed on an NVIDIA Jetson, a high-performance embedded controller.

V. Experimental Setup

A. Simulation

In the simulation experimental setup, the "ACC with sensor fusion" example was utilized as a foundation. The traditional ACC system was decoupled from the car model, specifically disconnecting the acceleration connection between the two models. Instead, the RL agent was introduced between the ACC model and the "vehicle model and environment model". The RL agent block accepts an observation vector, a multiplexer (MUX) was used to merge four observation signals: relative distance, relative velocity, safe distance, and longitudinal velocity into such a vector. Additionally, we design a reward function that guides the learning process of our agent. This reward function is connected to the reward input of the RL agent block , enabling the agent to learn corrective behaviors based on the provided feedback (See Fig. 6). Lastly, we incorporate an "IsDone" block that determines the criteria for ending an episode of learning. In our case, the requirements to conclude an episode are if the difference between the relative distance and longitudinal difference is less than -5. or if the longitudinal velocity becomes negative. These conditions serve as indicators for the termination of the learning episode. By implementing these modifications and configurations, we establish an experimental setup, that facilitates the training and learning process of the RL agent in the context of the ACC system and car model.

Once the RL agent is integrated into the ACC Simulink model, the next step involves configuring the RL agent itself, which entails selecting suitable hyperparameters. Initially, neutral values were chosen for the hyperparameters based on their relevance to a DDPG RL agent. As the agent learns, it becomes evident whether these values are optimal. To identify the most effective hyperparameters, the Matlab Experiment Manager is employed for a process called hyperparameter tuning. This involves experimenting with various hyperparameters within a defined range and selecting the ones that result in the highest reward as the optimal choices.

As previously mentioned, the DDPG agent utilizes an actor-critic network. Setting up a well-performing actor-critic network is also crucial in this context. The actor-critic architecture in reinforcement learning consists of two networks: the actor(1) and the critic(2). The actor network determines which action to take, while the critic network provides feedback to the actor regarding the quality of the action and suggests adjustments. The learning process of the actor is guided by the policy gradient approach. On the other hand, the critic evaluates the actions generated by the actor by calculating the value function [15].

$$a = \pi(s;\theta) \tag{1}$$

$$Q(s,a) = Q(s,a;\theta) \tag{2}$$

a : action taken by the action network

s : current state of the environment.

θ : denotes the parameters of the actor network.

$Q(s,a)$: estimated action/Q-value for a given state-action pair.

a : is the action taken in that state.

θ : denotes the parameters of the critic network.

The actor and critic networks in DDPG are implemented as neural networks. These neural networks consist of several key components, including an input layer at the top, an output layer at the bottom, and one or more hidden layers in between. The specific structure and types of hidden layers can vary depending on the implementation. In a neural network, each layer can consist of a specific number of nodes or neurons. The number of nodes in the input layer are determined by the dimensionality of the input data. The number of nodes in the output layer depends on the requirements of the task, such as the number of possible actions or the desired output dimension. The number of nodes in the hidden layers is a decision made by the engineer and can vary based on the complexity of the problem and available computational resources, in this study 48 neurons are chosen for each hidden layer. Increasing the number of neurons in the hidden layers enables the network to capture more complex patterns in the data. However, it is important to avoid overfitting, where the network becomes too complex and starts to memorize the training data rather than learning generalizable patterns [16]. Therefore, the choice of the number of nodes in each layer involves striking a balance between model complexity and generalization ability.

The actor network employed in this study [17] comprises 4 observation signals that serve as inputs to the network. These signals are processed through a sequence of 3 iterations, consisting of a fully connected layer followed by a ReLU (Rectified Linear Unit) layer followed by a fourth fully connected layer. In a fully connected layer, each neuron is connected to every neuron in the previous layer, establishing a dense connectivity pattern. Fully connected layers are commonly employed to capture intricate non-linear relationships between input and output data. A ReLU layer applies the ReLU activation function element-wise to the input. ReLU, short for Rectified Linear Unit, introduces non-linearity into the neural network. It is often employed between fully connected layers or convolutional layers to introduce non-linearities and improve the network's generalization ability [18]. The ReLU activation function replaces negative input values with zero, while maintaining positive values unchanged, thereby enhancing the network's ability to capture non-linear patterns in the data. After the fourth fully connected layer, a hyperbolic

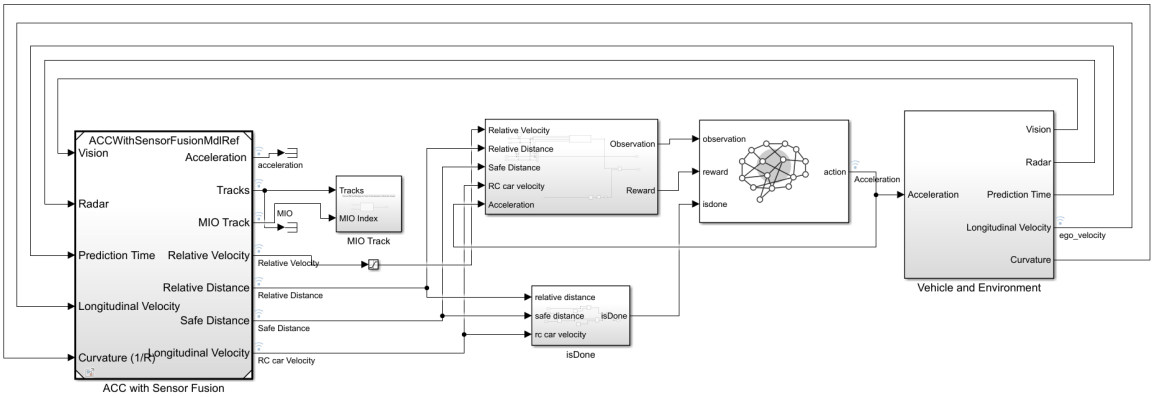

Fig. 6: Experimental setup ACC model with Reinforcement Learning Implemented

tangent (tanh) layer is employed to map the output values between -1 and 1. Subsequently, a scaling function is utilized to further map the values from -1 to 1 to the desired range of -3 to 2, which corresponds to the dimensions of our action. This actor network is visualized in Fig. 7. The critic network [17] depicted in Fig. 8 utilizes fully connected and ReLU layers. It is important to note that the critic network takes both the action signal and the observation signals as inputs. This is necessary because the critic network's role is to evaluate and determine the Q-value for the given state-action pair.

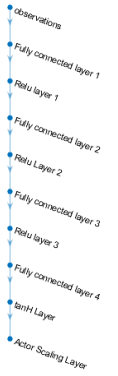

Fig. 7: Actor Network

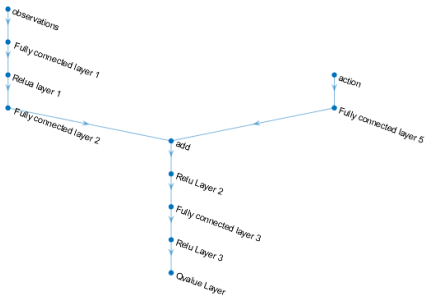

Fig. 8: Critic Network

B. Deployment

Once the RL agent has completed its training and we have obtained the C++ code representing its policy, we can proceed with deployment. The C++ code, containing the learned policy, was deployed on a NVIDIA Jetson Xavier NX, an embedded computing platform specifically designed for AI applications. Compared to other embedded controllers like the Raspberry Pi or Arduino, the NVIDIA Jetson offers significantly greater computing power. In order to establish communication between the hardware and the C++ code, the Robot Operating System (ROS) was utilized. ROS provides a collection of software libraries and tools that facilitate the development of robot applications. ROS encompasses the necessary infrastructure for running code and managing communication between processes. In the ROS framework, a system is built upon loosely coupled processes called nodes, each responsible for a specific task. Nodes communicate with one another by passing messages through logical channels known as topics. Using the publish/subscribe model, nodes can send and receive data from other nodes. In this study, ROS topics were utilized to establish communication with the lidar sensor and the electric motor. Directly communicating with an electric motor can be challenging. To simplify the process, an open-source electronic speed controller called the Vedder Electronic Speed Controller (VESC) was used. This controller enables the user to read and set the motor speed effectively, streamlining the interaction with the RC car's Velineon 3500 brushless electric motor. To incorporate ROS functionalities, the generated C++ code containing the RL policy was expanded with the ROS C++ Library which allows the user to listen and publish on ROS topics. The complete deployment setup is depicted in Fig. 9. Fig. 10 presents a visualization of the topic network that was established within the ROS framework. It showcases all the ROS topics that were used for either listening or writing purposes. During the deployment phase of the RL agent, it is essential to generate the observation signals, which were previously calculated for us in the simulation setup. While in the simulation, these signals were readily available, during deployment we needed to develop them ourselves. The relative distance signal is accurately computed using the LIDAR sensor, which has a range from 0.15 cm to 20m. Any objects detected closer than 0.15 cm are classified as being at a distance of 0cm. On the other hand, the longitudinal velocity, relative velocity, and safe distance signals are calculated within the C++ code using formulas 3,4,5 and 6. Only formula 5 which is used to calculate the safe distance, remains consistent between simulation and deployment. The formulas used for the rest of the observations signals differs for simulation and deployment. It is worth highlighting that in the calculation of the relative velocity, we opted to compute the lead car's velocity every second. This deliberate choice introduces a slight system delay, adding an extra challenge for the RL agent during deployment. A notable distinction between deployment and simulation is the use of different sensors for determining

relative distance. In simulation camera and radar was used while during deployment a LIDAR sensor was chosen. This difference is not problematic as we abstract the data from both sensor systems into a relative distance parameter. During deployment, an additional mapping process was required to convert the acceleration values generated by the RL agent into RPM values that could be sent to the electric motor. To achieve this, we utilized linear interpolation formula(7) to map the acceleration values ranging from -3 to 2 $\mathrm{m/s^2}$ to the RPM range of -2000 to 7400 RPM.

$$v_{\text{rccar}} = \left(\frac{\omega_{\text{electric motor}}}{\text{gear ratio} \times 60} \right) \times (D_{\text{wheel}} \times \pi) \ \text{[m/s]} \quad (3)$$

$$v_{\text{relative}} = \frac{D2 - D1}{t2 - t1} - v_{\text{rccar}} \ \text{[m/s]} \quad (4)$$

$$d_{\text{safe}} = (t_{\text{gap}} \times v_{\text{rccar}}) + d_{\text{default}} \text{[m]} \quad (5)$$

$$\omega_{\text{electric motor}} = \frac{rpm}{polepairs} \ \text{[rad/s]} \quad (6)$$

$$y = y_1 + \frac{(x - x_1) \cdot (y_2 - y_1)}{x_2 - x_1} \quad (7)$$

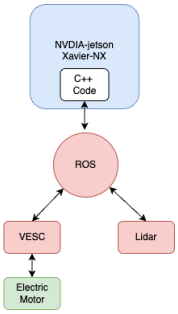

Fig. 9: Deployment Experimental Setup

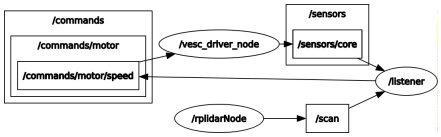

Fig. 10: ROS topics graph

VI. Results

Two agents are introduced to showcase the results: Agent S and Agent A. Agent S, a safe RL agent, demonstrated safe performance by flawlessly navigating 100 consecutive randomized scenarios without violating safety constraints, thereby passing the environmental robustness testing phase. In contrast, Agent A, an alternate RL agent, performed well but could only handle a maximum of 12 random scenarios consecutively, thereby failing the environmental robustness test. Agent S and Agent A were trained using different reward functions, as shown in formulas (8) and (9), and were trained with slightly different hyperparameters, as presented in Table 1 and Table 2. To assess their performance, we will evaluate the behavior of the RL agents in three distinct scenarios. Both agents were trained on first scenario where the RL agent has to follow a car that accelerates moderately and gradually brakes towards the end. In the second scenario, of intermediate difficulty, the lead car accelerates to a speed of 20 m/s and rapidly decelerates to 0 m/s within a short distance of just 5 meters. The third scenario poses a high level of difficulty, as the lead car behaves erratically, swerving from left to right and frequently moving in and out of the field of view of the camera and radar sensors. Additionally, in this challenging scenario, the lead car abruptly brakes from 20 m/s to 0 m/s within a distance of 5 meters. When comparing the behavior of agent S (Fig. 11) and agent A (Fig. 12) in the easy scenario, we observe a distinction in their behavior. Agent A demonstrates a remarkable ability to adapt to the exact velocity of the lead car, as evidenced by the majority of the test showing a relative velocity close to zero, indicating a small velocity difference between the two cars. On the other hand, agent S also attempts to match the velocity of the lead car, but does so in a slower and more cautious manner, with no significant changes in relative velocity occurring within a short period of time. By examining the graph depicting safe distance and relative velocity, we can observe that Agent A tends to drive closer to the safe distance compared to agent S, who adopts a more cautious approach and strives to maintain a greater distance. Additionally, when comparing the acceleration data of both agents, it is evident that agent A occasionally exhibits abrupt changes in acceleration, while agent S displays smoother acceleration behavior. Thus far, both agents appear to be functioning adequately. When examining the behavior of the two RL agents in the second scenario (Fig. 13 and Fig. 14), where the lead car abruptly decelerates from a speed of 20 m/s to 0 m/s within a distance of 5 meters, we can once again observe that agent A attempts to maintain a closer distance to the safe limit compared to agent S. However, this strategy led to a safety violation as agent A exceeded the safe distance and ultimately collided with the lead car at the 33 second timestamp. In this particular scenario, it becomes evident that the cautious approach adopted by agent S proves to be more effective in terms of safety. Throughout the entire scenario, agent S managed to maintain a relative distance larger than the safe distance, even during the moment when the lead car applied aggressive braking. Agent S demonstrates an acceleration behavior that appears to be more comfortable for the passengers, while agent A tends to accelerate and brake at maximum capacity, potentially causing discomfort. In the final and most challenging scenario, we encounter a lead car that swerves from left to right, intermittently disappearing and reappearing within the field of view of the object detection sensor. It is crucial to note that if the car fails to detect any lead cars or objects in front of it, the relative distance measurement becomes meaningless. Since the calculation of relative velocity relies on the relative distance, we may encounter a NaN error during the velocity calculation. To address this issue, a saturation block is implemented on the relative velocity signals. This saturation block ensures that the signal remains within predefined bounds, specifically ranging between 20 m/s

and -20 m/s. Consequently, in situations where no relative distance is detected, resulting in a NaN value for relative velocity, the saturation block will assign a value of 20 m/s. It is important to highlight this behavior because it may appear that the relative velocity abruptly jumps to 20 m/s at times. This occurrence signifies that no lead car is detected, and the saturation block is activated to provide a consistent value for the relative velocity. In Fig. 16 and Fig. 15, it is evident that both agents encounter situations where there is no available data for the relative distance due to the lead car swerving in and out of the sensor's field of view. As a result, there are noticeable jumps in the relative velocity signals to 20 m/s. Despite the absence of relative distance data and occasional inaccuracies in the relative velocity measurements, agent S manages to successfully navigate the scenario without colliding with the car, even when the lead car applies aggressive braking at the end. On the other hand, agent A avoids crashing into the car while it is swerving but eventually collides with it when the lead car begins braking. This demonstrates that agent S is the safer Reinforcement Learning agent for controlling the acceleration of an Adaptive Cruise Control system in a simulated environment.

During the deployment tests, both agents successfully avoided collisions with the lead car. However, this outcome can be attributed to the fact that the RL agents were tested at maximum speeds of 8 km/h, which did not generate highly risky situations in terms of Adaptive Cruise Control (ACC). Despite the simplicity of these scenarios, we were able to observe some differences in safety between the agents. For instance, we conducted a test where another RC-car accelerated to 8 km/h and then applied the brakes as quickly as possible. In Fig. 17, it is noticeable that agent A crossed the safe distance at two points. However, it's important to note that in reality, the crossing was minimal. On the graph, the relative distance appears as 0 because the LIDAR sensor registers distances below 15 cm as 0 cm. In contrast, as shown in Fig. 18, agent S consistently maintained a safe distance without crossing it. Examining the velocity readings of agent A in Fig. 19, we observed that towards the end of the graph, when the lead car came to a standstill, agent A moved backwards. This backward movement can be considered a safety violation when the lead car is stationary. On the other hand, agent S remained stationary along with the lead car, as depicted in Fig. 20. This observation is further confirmed by the acceleration graph in Fig. 22, where agent S exhibits an acceleration of 0 when the lead car is stationary, while agent S shows a negative acceleration, as illustrated in Fig. 21.

$$r_{\text{agent S}} = D_{\text{t}} + V_{\text{t}} - (\triangle v_{\text{t}})^2 \tag{8}$$

$$r_{\text{agent A}} = (10 \times D_{\text{t}}) + V_{\text{t}} - \frac{(\triangle v_{\text{t}})^2}{10} - (a_{\text{t-1}})^2 \tag{9}$$

$\triangle v_{\text{t}}$: relative velocity - longitudinal velocity at time t

$a_{\text{t-1}}$: acceleration action taken at the previous time step

TABLE I: Dt calculation A.A

$d_{\textbf{relative}} - d_{\textbf{safe}}$	$D_{\textbf{t}}$
$>= 0$	$\tan(d_{\text{relative}} - d_{\text{safe}})$
<0	-10

TABLE II: Vt calculation A.A

$v_{\textbf{relative}}$	$V_{\textbf{t}}$
$>= 0$	1
<0	-100

TABLE III: Dt calculation A.S

$d_{\textbf{relative}} - d_{\textbf{safe}}$	$D_{\textbf{t}}$
$>= 0$	$d_{\text{relative}} - d_{\text{safe}}$
<0	-15

TABLE IV: Vt calculation A.S

$v_{\textbf{relative}}$	$V_{\textbf{t}}$
$>= 0$	0
<0	-1000

VII. Conclusion

Based on our findings, we can draw the conclusion that environmental robustness testing offers a straightforward yet effective approach for evaluating the safety of a trained RL agent. When the RL agent fails during testing, this method provides valuable insights to developers regarding the challenging scenarios that the agent struggles to handle. These insights enable them to make necessary adjustments in subsequent training iterations. Furthermore, it can be inferred that the safety observed during testing can be translated to deployment scenarios. However, it is essential to acknowledge that numerous other challenges must be addressed to ensure safe deployment of RL agents. These challenges include addressing issues related to partial observability, non-stationarity, real-time inference, and system delays. Safely deploying RL agents requires careful consideration and resolution of these challenges in addition to respecting safety constrains.

VIII. Future Work

In our future work we would like to investigate whether increasing the required number of successful scenarios enhances safety and, if so, by what extent. This is likely dependent on the complexity of the task the agent needs to learn and the intricacy of the environment in which it operates. Conducting research in this area will provide insights into the correlation between the number of passed scenarios and safety enhancement. By delving into alternative methods of environmental robustness testing, such as adversary learning, intriguing opportunities arise that go beyond the scope of random scenario generation. In adversary learning, a second RL agent endeavors to minimize the reward obtained by the trained agent by generating challenging scenarios. Additionally, investigating the impact of utilizing CUDA code instead of C++ for translating the greedy policy block to deployable code would be intriguing. CUDA enables faster computation on GPUs, and the Matlab Coder app supports CUDA code generation. Unfortunately, limited access to suitable facilities prevented us from replicating intermediate and difficult simulation scenarios during deployment tests. Conducting tests at higher speeds (72 km/h or 20 m/s) would truly challenge the agents safety. It is worth noting that both agents experienced less smooth acceleration during the actual deployment compared to the simulation. This issue, especially prominent for agent S, could potentially be resolved by implementing a PID controller for acceleration.

REFERENCES

[1] TechTarget. What is artificial intelligence (ai)? [Online]. Available: https://www.techtarget.com/searchenterpriseai/definition/AI-Artificial-Intelligence

[2] U. of Columbia. Artificial intelligence (ai) vs. machine learning. [Online]. Available: https://ai.engineering.columbia.edu/ai-vs-machine-learning/

[3] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[4] Synopsys. What is reinforcement learning? [Online]. Available: https://www.synopsys.com/ai/what-is-reinforcement-learning.html

[5] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[6] J. Garcıa and F. Fernández, "A comprehensive survey on safe reinforcement learning," *Journal of Machine Learning Research*, vol. 16, no. 1, pp. 1437–1480, 2015.

[7] G. Dulac-Arnold, N. Levine, D. J. Mankowitz, J. Li, C. Paduraru, S. Gowal, and T. Hester, "Challenges of real-world reinforcement learning: definitions, benchmarks and analysis," *Machine Learning*, vol. 110, no. 9, pp. 2419–2468, 2021.

[8] M. Khatun, M. Glaß, and R. Jung, "Scenario-based extended hara incorporating functional safety and sotif for autonomous driving," in *ESREL-30th European Safety and Reliability Conference*, 2020.

[9] L. Narodetsky. Iso 26262 vs. sotif (iso/pas 21448): What's the difference? [Online]. Available: https://content.intland.com/blog/iso-26262-vs-sotif-iso-pas-21448-whats-the-difference

[10] G. Dalal, K. Dvijotham, M. Vecerik, T. Hester, C. Paduraru, and Y. Tassa, "Safe exploration in continuous action spaces," *arXiv preprint arXiv:1801.08757*, 2018.

[11] M. Alshiekh, R. Bloem, R. Ehlers, B. Könighofer, S. Niekum, and U. Topcu, "Safe reinforcement learning via shielding," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[12] L. Wen, J. Duan, S. E. Li, S. Xu, and H. Peng, "Safe reinforcement learning for autonomous vehicles through parallel constrained policy optimization," in *2020 IEEE 23rd International Conference on Intelligent Transportation Systems (ITSC)*, 2020, pp. 1–7.

[13] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 23–30.

[14] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[15] V. Konda and J. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems*, S. Solla, T. Leen, and K. Müller, Eds., vol. 12. MIT Press, 1999. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/1999/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf

[16] B. Sabiri, B. El Asri, and M. Rhanoui, "Mechanism of overfitting avoidance techniques for training deep neural networks." in *ICEIS (1)*, 2022, pp. 418–427.

[17] Mathworks. Train ddpg agent for adaptive cruise control. [Online]. Available: https://nl.mathworks.com/help/reinforcement-learning/ug/train-ddpg-agent-for-adaptive-cruise-control.html

[18] H. Ide and T. Kurita, "Improvement of learning for cnn with relu activation by sparse regularization," in *2017 International Joint Conference on Neural Networks (IJCNN)*, 2017, pp. 2684–2691.

IX. APPENDIX

TABLE V: Hyperparameters Agent S

Hyperparameters	**Value**
Target Smooth Factor	1e-3
Discount Factor	0.99
Mini-batch Size	128
σ	0.58
σ decay rate	1e-5
Actor learning rate e	1e-4
Critic learning rate	1e-3

TABLE VI: Hyperparameters Agent A

Hyperparameters	**Value**
Target Smooth Factor	1e-3
Discount Factor	0.97
Mini-batch Size	64
σ	0.61
σ decay rate	1e-5
Actor learning rate e	1e-4
Critic learning rate	1e-3

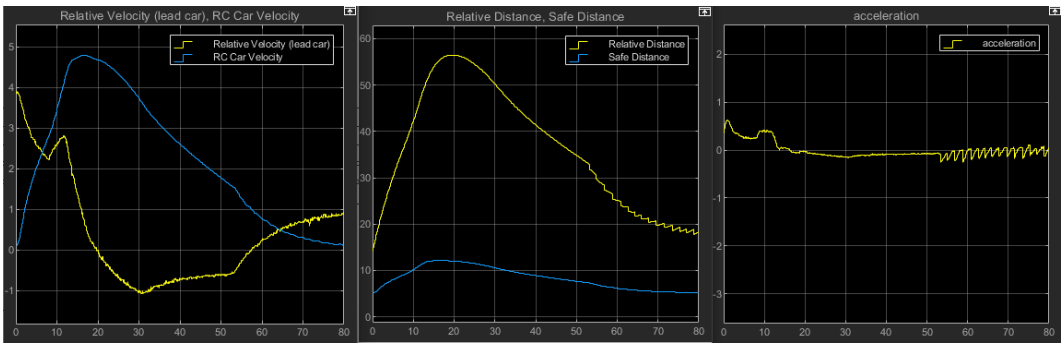

Fig. 11: RL Agent S in an easy scenario (Simulation)

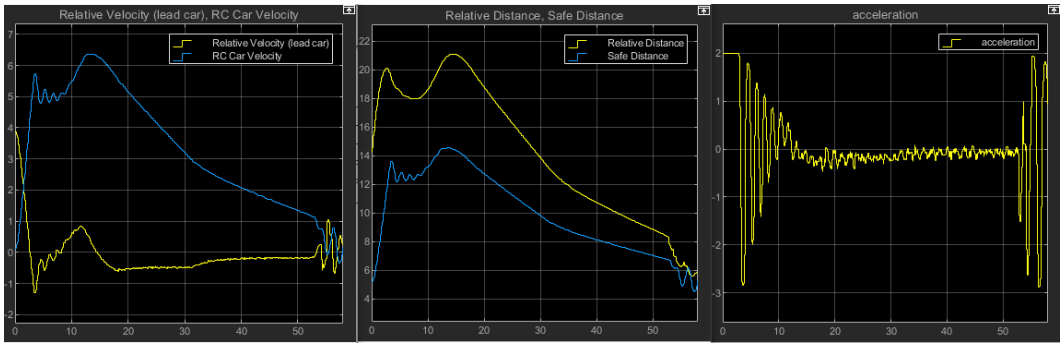

Fig. 12: RL Agent A in an easy scenario (Simulation)

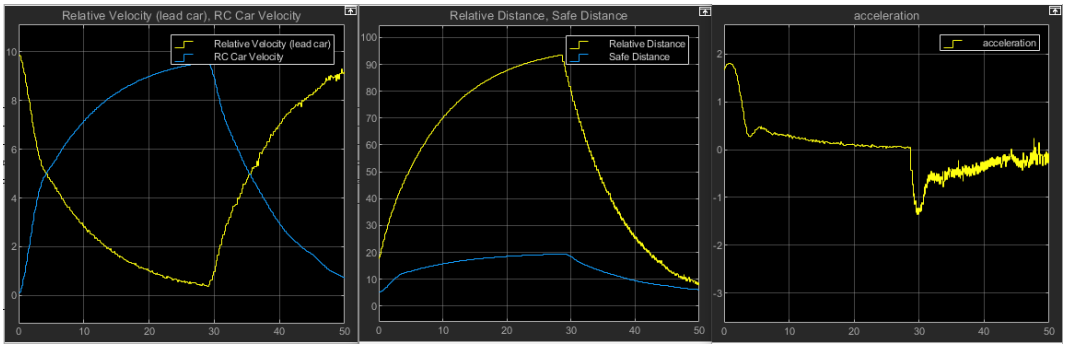

Fig. 13: RL Agent S in an intermediate scenario (Simulation)

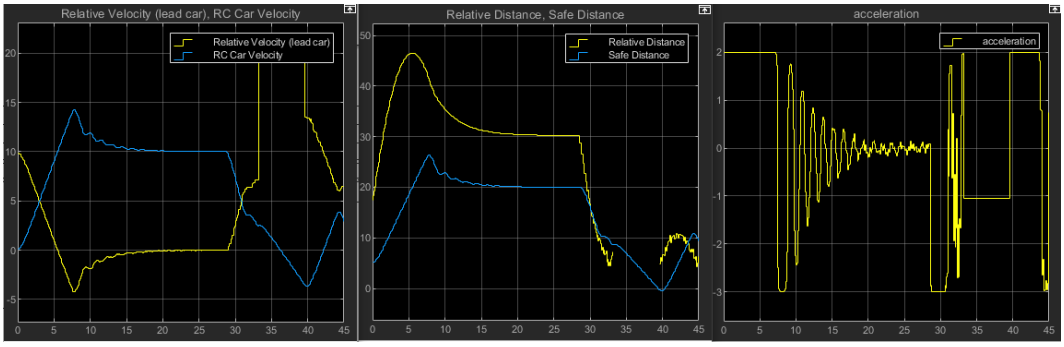

Fig. 14: RL Agent A in an intermediate scenario (Simulation)

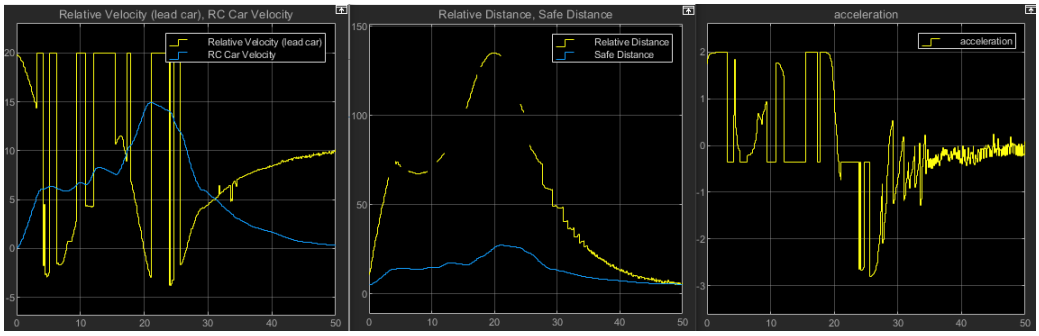

Fig. 15: RL Agent S in a difficult scenario (Simulation)

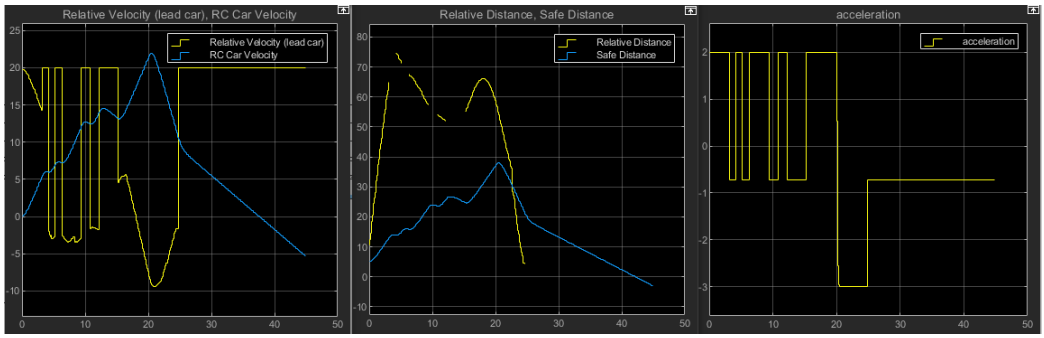

Fig. 16: RL Agent A in a difficult scenario (Simulation)

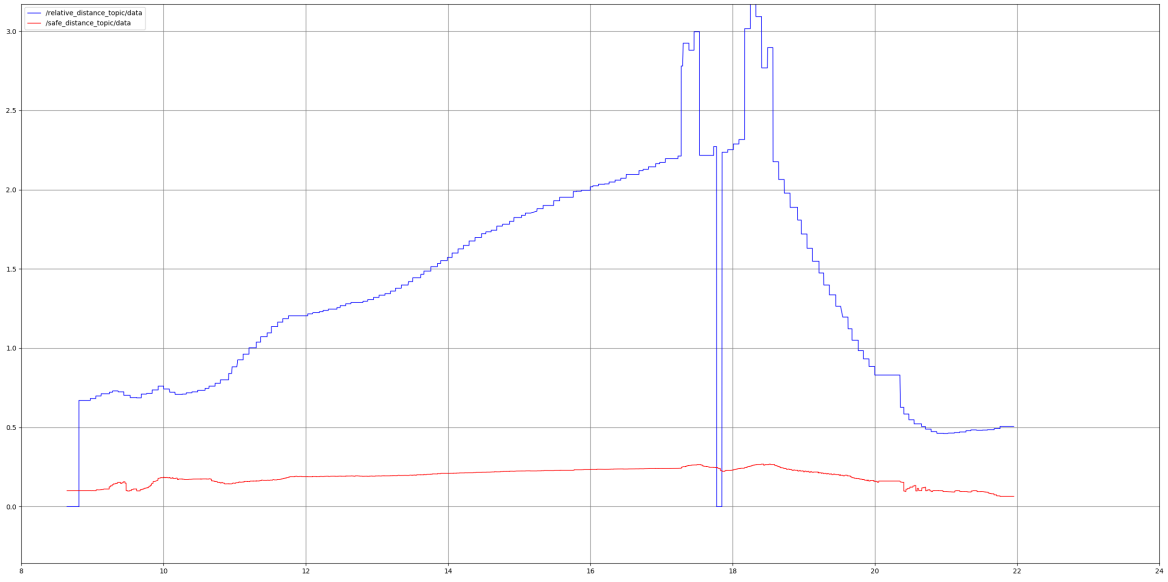

Fig. 17: RL Agent A Relative Distance and Safe Distance (Deployment)

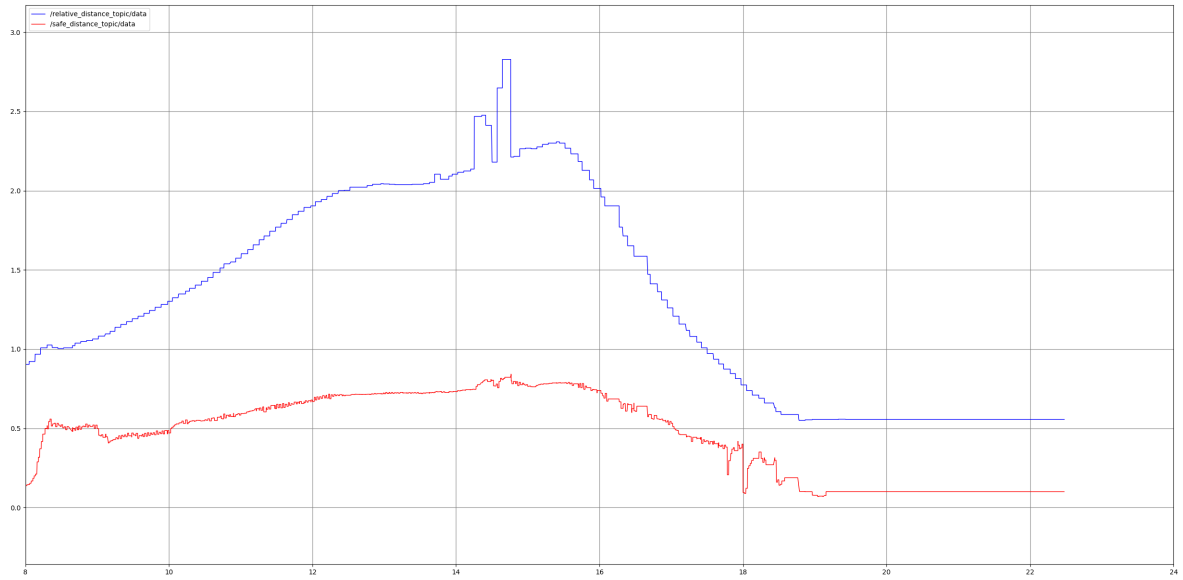

Fig. 18: RL Agent S Relative Distance and Safe Distance (Deployment)

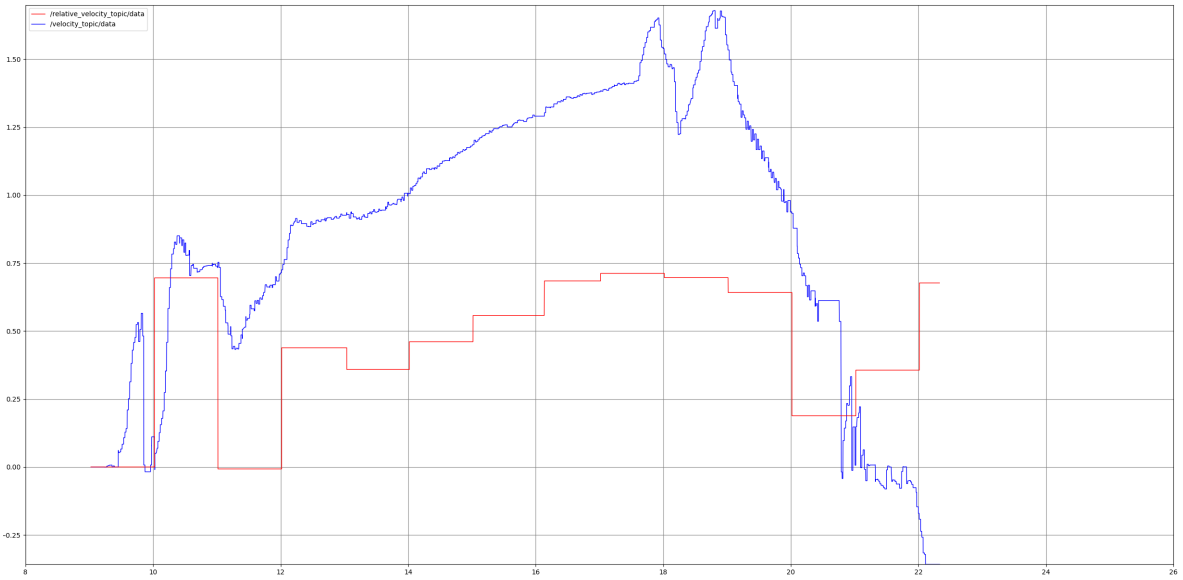

Fig. 19: RL Agent A Relative Velocity and RC car Velocity (Deployment)

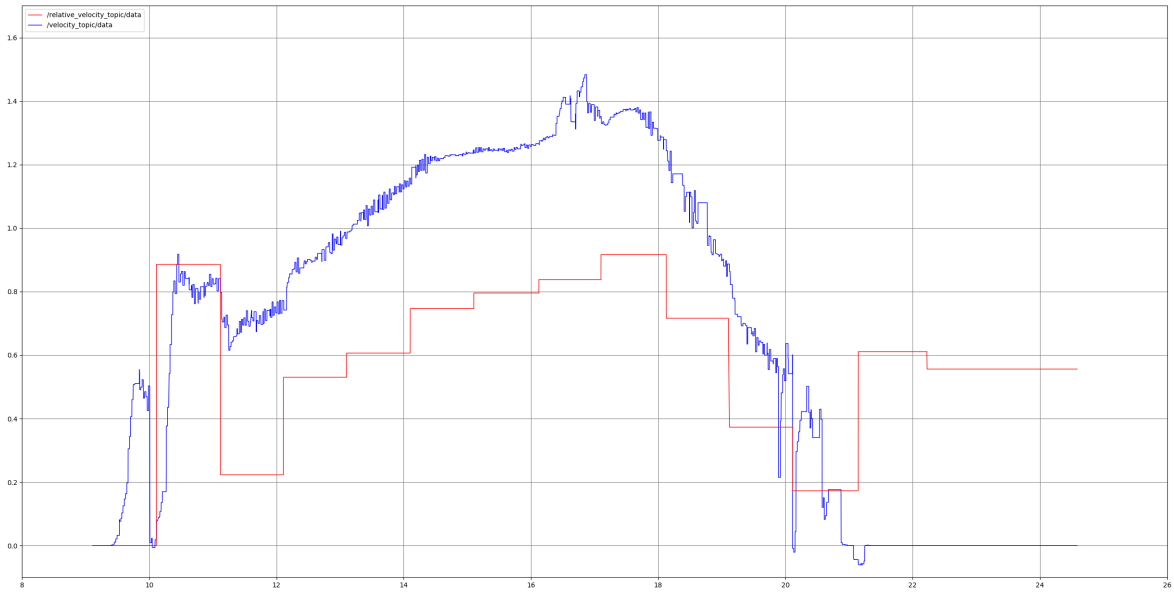

Fig. 20: RL Agent S Relative Velocity and RC car Velocity (Deployment)

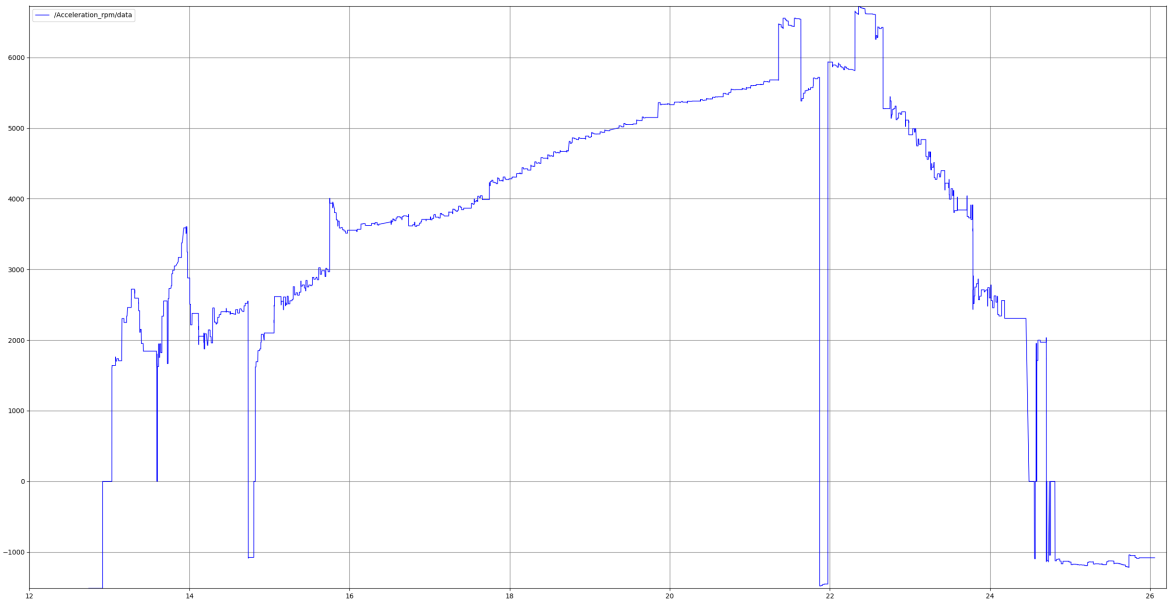

Fig. 21: RL Agent A Acceleration (Deployment)

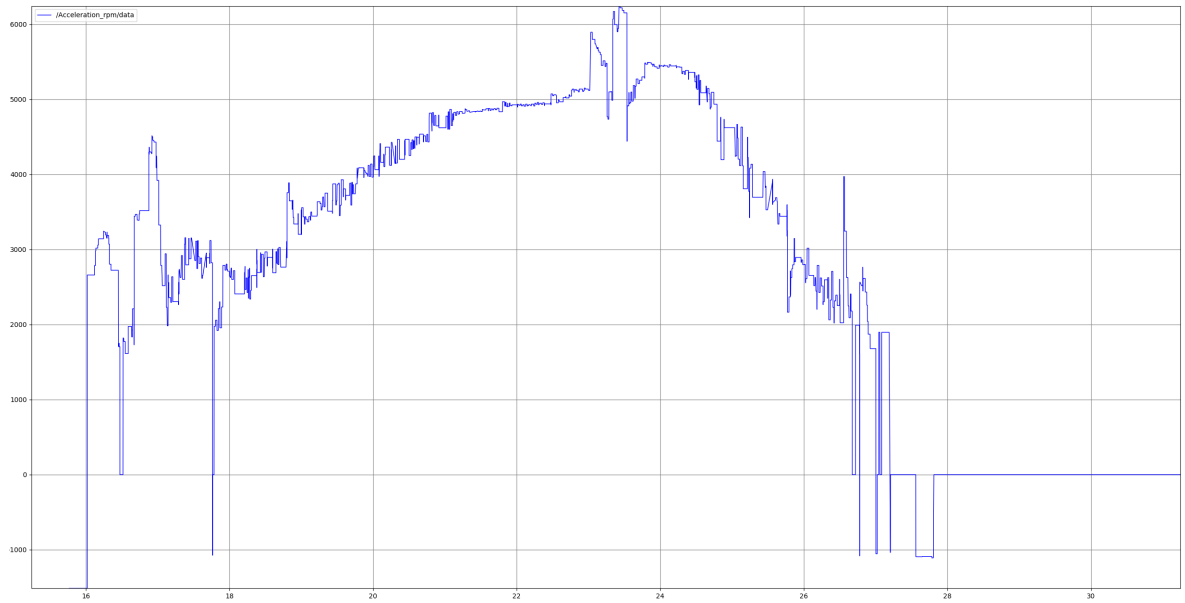

Fig. 22: RL Agent S Acceleration (Deployment)