Using V2X and reinforcement learning to improve autonomous vehicles algorithms with CARLA

by

Mahmoud Abdalkarim

A thesis submitted to the Graduate Faculty of
Auburn University
in partial fulfillment of the
requirements for the Degree of
Master of Science - Computer Science and Software Engineering

Auburn, Alabama
May 7, 2022

V2X, Reinforcement Learning, Machine Learning, Autonomous Vehicles, Algorithms

Approved by

Xiao Qin, Chair, Alumni Professor, Computer Science and Software Engineering
Ashish Gupta, Co-chair, Professor of Business Analytics, Department of Systems and Technology
Bo Liu, Assistant Professor, Computer Science and Software Engineering

Abstract

Autonomous vehicles (AV) or cars of future are only growing in popularity. However, there is a reported lack of trust in these AV. According to a recent survey conducted by the AAA automotive group on understanding people's attitudes towards self-driving cars they found out that only 14% of drivers would trust and feel safe riding in an autonomous vehicle. Current autonomous vehicles rely on sensors such as RGB cameras, LiDAR, RADAR, and more. These sensors have limited perception and prediction capabilities in certain ambient conditions. This research aims to study the impact of connecting self-driving cars with their surrounding through Vehicle-to-Everything (V2X) data. V2X is a communication system where data from sensors, traffic lights and many other sources travel through a high-bandwidth and low latency network and can be used as input for autonomous cars. We expand on this by introducing a reinforcement learning (RL) algorithm that benefits from the use of V2X and trains a car in a simulated testbed on various maneuvering scenarios to emphasize the impact of using V2X compared to the use of traditional sensors alone. Furthermore, we compare our solution with a simple algorithm that relies on the use of a camera (RGB) sensor in various lighting and weather conditions. We use the open-source simulation software CARLA to validate the improvement of the autonomous vehicle algorithm coupled with V2X and RL.

To my mom, dad, and
Palestine, ***all of Palestine****.*

Acknowledgments

This master's thesis would not have been completed without the contributions of many people. I would like to thank my advisors Dr. Xiao Qin and Dr. Ashish Gupta for their endless support and feedback throughout the work on this thesis. Dr. Qin was my international advisor when I first came to Auburn as a Fulbright scholar, he helped me settle in a new environment and make the most out of my experience at Auburn. I would also like to thank Dr. Gupta for welcoming my research idea in self-driving cars in addition to the enormous support throughout the project. I want to thank Dr. Bo Liu for meeting with me multiple times and for providing his valuable feedback that helped me elevate my research project. I also had the pleasure of taking graduate courses with all the committee members, all of which I thoroughly enjoyed.

None of this would be possible without all the people who helped me get where I am right now, my family members back home in Palestine. My Mom and Dad and my siblings: Maysam, Dana, Ihab, and Issam. I would like to quote a part of my father's PhD acknowledgments at Michigan State University: **"**To my family, I know how you were patiently accepting my commitments and assignments as a graduate student. I am proud that in spite of the many challenges and difficulties that we faced in these two years, we together got the mission done. Thank you so much and I dedicate this thesis for you. From the bottom of my heart, I thank my family in Palestine for their unlimited support and encouragement through my graduate school study. I feel so lucky to have such a supportive mother, father, sisters, and brothers.**"**

Last but not least, I would like to thank all my friends back home in Palestine as well as my friends here in the States for their never-ending encouragement and support throughout my degree.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AV	Autonomous Vehicle
BSM	Basic Safety Messages
CNN	Convolutional Neural Network
C-V2X	Cellular V2X
DQN	Deep Q-Network
DSRC	Dedicated Short Range Communications
eTIS	Electronic-Tire Information System
GM	General Motors
GPS	Global Positioning System
KPI	Key Performance Indicator
LTE	Long-Term Evolution
MDP	Markov Decision Process
ML	Machine Learning
NHTSA	National Highway Traffic Safety Administration
POMDP	Partially Observable Markov Decision Process
ReLU	Rectified Linear Unit
RGB	Red Green Blue
RL	Reinforcement Learning
V2X	Vehicle to Everything

Chapter 1 Introduction

This work highlights the importance of using Vehicle-to-Everything (V2X) communication in autonomous vehicles and how it can improve AV algorithms. We will show this significant importance using reinforcement learning (RL) to teach an agent the simple maneuvering scenario of lane changing / car passing. This scenario will rely on V2X data being transmitted between various vehicles running in the scenario.

Although autonomous vehicles appear to be a new technology and only gained great attention in recent years, they are not a new creation. In GM's 1939 exhibit, Norman Bel Geddes created the first self-driving car. The car was electric, and it was guided by radio-controlled electromagnetic fields that were generated by metal spikes embedded in the roadway. This was only a concept; it was not made a reality until 1958 by General Motors [3]. The car had sensors that could detect the current that was flowing through the wires on the road and that was used to tell if the car should steer left or right. Later in 1977, the Japanese made this idea better by integrating a camera system into their car [4]. This camera sent images to a computer which would process the info in the image and help make decisions. The problem? The car could only travel below 20 mph. The breakthrough came a decade later, when the Germans were able to fit a Mercedes-Benz van with camera sensors that could drive itself up speeds of up to 56 mph [4].

In the mid 2000's, there was a rapid acceleration in the development of autonomous vehicles. One such example is the Defense Advanced Research Projects Agency (DARPA) challenge which first took place on March 13, 2004 [5]. The goal of the challenge was to autonomously navigate

142 miles through the desert. A year later, Stanford's robot "Stanley" was the first ever fully autonomous vehicle to complete the DARPA course [6].

A decade later, many big tech companies began investing in the idea of self-driving cars. One such company is Google. In 2009, Google started the self-driving car project with a simple goal of driving ten uninterrupted 100-mile routes using sensors fitted on a Toyota Prius. Years later, Waymo became an independent autonomous driving company under Alphabet. Since then, Waymo and Google have been improving their self-driving technology to a point where they are able to offer fully autonomous rides in many parts of the United States [7]. Another giant leading the way in self driving cars is Tesla. The main two areas of focus for Tesla are safety and autonomous driving. Tesla claims that their cars feature a safety-first design and are made to be the safest in the world [8]. In addition, new Tesla cars are often coupled with an Autopilot system which Tesla claims that it will give the driver behind the wheel more confidence and make the driving experience more enjoyable. This system functions with eight surround cameras that provide 360 degrees of visibility as well as twelve ultrasonic sensors to complement this vision [9].

According to a research note provided by the U.S. Department of Transportations, there were 37,461 people killed in crashes on U.S. roadways in 2016 [10]. In the same report, they highlighted that of all accidents around 94% to 96% are caused by human error. These statistics, among others, are primarily used by self-driving cars advocates like Google and Tesla to argue that the human element must be removed from the driving process, thus making our roads much safer. Another great aspect to push for self-driving cars is regarding traffic congestion. The main cause of congestion is the varying reaction time needed for acceleration and deceleration exhibited by

different drivers. Removing this human factor and replacing it with an automated and equivalent process among cars on the road can help alleviate traffic congestion.

Autonomous cars nowadays used a wide range of sensors such as RGB cameras, LIDAR, RADAR and more. These sensors are used for many objectives like lane keeping, obstacle detection or even full autonomous driving as in Tesla's autopilot. They also rely heavily on GPS for navigation and route planning. Moreover, they are equipped with complex machine learning and computer vision algorithms to help in the perception and prediction of their surroundings.

By far, the biggest promise made by the autonomous vehicle industry is safety and reducing the human error factor. However, this is **only true** if autonomous vehicles have **complete and sound perception and prediction** of their environment. Not only that, but autonomous vehicles need to understand the context of the surroundings as well. An example of this is how an autonomous vehicle would struggle to predict a child running after a ball that comes rolling across the road. Currently, autonomous cars have limited sensing capabilities, leading to a very cautious and conservative behavior on the road. One way to improve this is by using forms of communication between different autonomous cars on the road to compensate for the deficiency in perception.

In the rest of this chapter, we will go over the problem we are trying to solve, the justification for this research and a detailed outline for the reset of this thesis.

1.1 What we are Trying to Solve

In our work, we create a **system that utilizes V2X to enhance the perception and prediction** of autonomous vehicles on the road. Autonomous vehicles still struggle with perception and contextual understanding as discussed previously. To solve this, we introduce a system where

autonomous vehicles can exchange information between each other about their local perceptions to help other vehicles better understand the surrounding environment. **To demonstrate our proposed system, we take the scenario of lane changing / car passing that relies on V2X information being passed between multiple vehicles on the road.** We teach a reinforcement learning agent to take actions based on the V2X information being passed between vehicles. An illustration of this scenario can be seen in figure 1.1.

Deploying and testing such a system in real life would be very expensive and unsafe. For these reasons, we will use the open-source simulator for autonomous driving research CARLA. To evaluate the effects of having V2X communication between autonomous vehicles, thousands of simulations can be run, and metrics can be extracted for performance measurements. Some key factors to be measured could be the number of collisions that happened in the scenario runs (when using regular sensors and when using V2X), and the time it took for the lane change action.

1.2 Research Justification

On March 18, 2018, a self-driving Uber SUV did not recognize a woman who was jaywalking on the street with her bicycle and that lead to the self-driving car fatally hitting the woman [11]. According to an investigation, the reason behind the accident was that this car was not designed to avoid an imminent collision. Once the car's sensors and cameras realized that there was someone on the street, it was too late to brake. The car failed to predict her path, because she was a jaywalking pedestrian, and it could not tell if the woman was a bike, pedestrian, or another vehicle. This could be an example where V2X sensors could have seen the path of the pedestrian much earlier than regular sensors could have and could have altered the outcome.

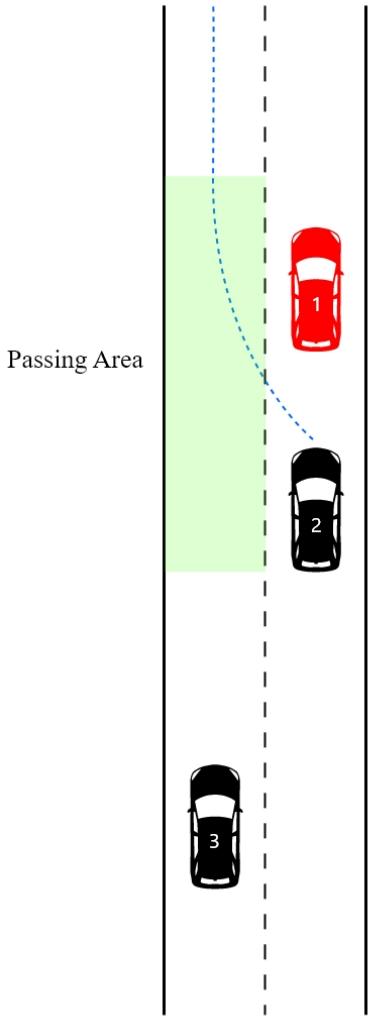


Figure 1.1: Lane changing scenario used to illustrate the importance of V2X communication. In this scenario, car 2 needs to pass the emergency stopped vehicle car 1. Car 2 needs to be aware of adjacent lanes to make sure that it does not hit car 3 when committing to the pass. In clear conditions, this might seem trivial, but in harsh conditions such as rain or fog, it becomes more complicated and harder to detect using ordinary sensors.

The importance of connectivity lies in giving autonomous cars situational awareness that cannot be seen using regular sensors. Another clear example that comes to mind, is aquaplaning. In a regular non-connected situation where cars follow each other and use regular sensors, it is hard for a following car to predict that a car in front of it aquaplaned. But with V2X technology,

the story is completely different. Cars that go into aquaplaning can inform cars behind them of this situation, thus causing them to slow down and be informed of such incidents. Aquaplaning is just one case that can be mitigated and solved with the extension of autonomous cars to use V2X technology. Many more cases, like fog and rainy conditions can also benefit from this technology. Another important case that comes to mind is in the decision-making process of lane changing and car passing. Figure 1 shows more uses cases where V2X could be helpful and potentially save lives.

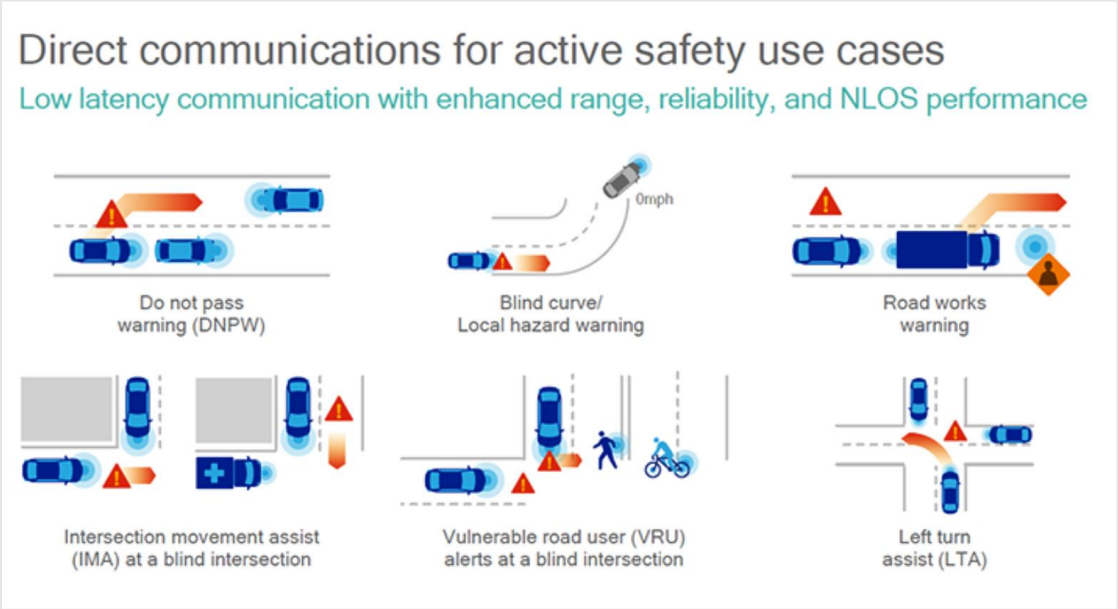


Figure 0.2: Use cases in which V2X can be applied. Source: Qualcomm.

1.3 Thesis Outline

This thesis has six remaining chapters:

Chapter 2 provides a brief literature review on different research and solutions that study the use of V2X in autonomous cars.

Chapter 3 provides a technical background for the proposed system, including discussing V2X technology further, reinforcement learning, and the open-source simulator for autonomous driving research CARLA.

Chapter 4 talks about the methodology used and proposed system design, how V2X information was integrated into the reinforcement learning algorithm, and how different components were implemented.

Chapter 5 discusses the experiments that were run and how each of them was setup. In addition, it discusses the different KPIs that were used in the evaluation process.

Chapter 6 we report our results and observations, the implications for research and practice, and the limitations of this study.

Chapter 7 mainly draws conclusions and describe our future plans with this research.

Chapter 2 Literature Review

This chapter will provide a brief literature review of current and relevant research done on the use of Vehicle-to-Everything (V2X) data in self-driving cars as well as research regarding the current challenges regular sensors, such as cameras and radar, face in self driving cars.

2.1 Seeing the Unseen

Self-driving cars use a lot of sensors for precepting and predicting their surrounding environments. Most notably cameras, radar, and LiDAR are used for this purpose. These sensors typically work in an efficient and effective manner when there is a direct line-of-sight to the objects they are trying to see and when the weather conditions are optimal for this use. These sensors can be effective to figure out the speed, path, acceleration, and even shape of the object they are trying to detect. Although some of the sensors can used on self-driving cars such as radar and lidar have advantages over regular cameras, because they are not affected by how dark or bright the scene is, some of them struggle when it comes to factors like fog or rain or snow and object detection in this kind of weather is considered the one of the most critical issues facing self-driving cars [12]. Moreover, sensors like LiDAR can process billions of data points per second in a manner that is both more accurate and more efficient than a regular 2-D image which requires complex computations. But what happens when these sensors are used in adverse weather environments or when used around obstacles and curves? In the first case, sometimes it can be considered a computer vision problem that can be solved by using complex algorithms and adding more hardware. In the second case, there is no clear or guaranteed way to see objects that are considered non-observable.

In the case of adverse and harsh weather, researchers are working on complex algorithms that can improve object detection in self-driving cars. There have been many efforts to create and understand systems that can function in such weather conditions. In [12], researchers at the joint college of engineering of Florida A&M and Florida State University studied the impact of foggy weather on state-of-the-art Convolutional Neural Networks (CNN) object-detectors. In their work, they investigate how defogging images can improve the performance of CNN-based object detectors. They employed a Cycle consistent Generative Adversarial Network (CycleGAN)-based fog removal technique to improve the quality of foggy images. After training the model using the YOLOv3 network, object detection was performed on the foggy images to see how well the defogging algorithm works. In moderate fog, there was around a 5% increase in in the recall and precision of the object detector after defogging the images. With heavy fog, there was a negligible improvement after defogging the images. This shows the significant impact of adverse weather and how it can, to a big degree, hinder the accuracy of object detectors in self-driving cars.

Research has also tried to tackle the second problem, which deals with sensing and seeing vehicles and other objects around curves and obstacles and outside the sight of regular sensors. Whether using image sensors or other types of sensors such as radar, scene detection depends on light or lasers travelling directly and in a non-obscured fashion between the sensor and the object. Non-line-of-sight (NLOS) methods try to recover objects that are not directly visible to the sensing system. Using multiple cameras has shown to be somewhat effective to overcome poor illumination and allow for the recovery of 2D images of objects which are partially illuminated and hidden from the camera [13]. Other successful NLOS methods rely on cameras use an ultrafast laser and a streak camera to shine short laser pulses and calculate the reflection interval of these pulses to create both a spatial and temporal dimension of hidden objects [14]. This method however

suffers since it requires special and expensive hardware and that it can only function for single small objects. In addition, it relies on prior knowledge of the scene such as the distance to the reflective wall. These drawbacks make such systems ineffective for use cases such as self-driving cars. Other methods specific to self-driving cars use Doppler radar sensors to construct real-time live tracking of objects around corners [15]. This is done by diffusing reflections from a relay wall that are scattered and transmitted back after hitting an invisible object. But in open areas such as highways or places without reflective relay areas, this method could struggle to predict hidden objects.

2.2 V2X for Safety Warnings

Current network hardware and infrastructures cannot guarantee the latency required for V2X to be used in safety-critical tasks. This is why most of the research and application deployment related to V2X has been scaled down to mostly providing safety warnings to vehicles on the roads. Continental is currently working on an aqua planning assistance concept that can predict the risk of aquaplaning before it happens. This is done by detecting when the front wheels are in a floating situation, and it will trigger a warning to the driver [16]. This concept system uses camera's mounted on the side of the car as well as the tire-mounted eTIS (electronic-Tire Information System) sensors to detect early aquaplaning situations. Moreover, the system utilizes V2X technology to give a warning to cars in the near vicinity about a potential aquaplaning scenario on the road ahead, effectively acting as a sensor that supports multiple cars. Other research has implemented applications to use V2X to send warning notifications to autonomous vehicles in certain conditions. In [17], researchers modified the simulation platform CARLA to enable On-Board Unit in the loop testing and build a lateral controller for the purpose of this research. They

studied the possibility of using V2X to send out Basic Safety Messages (BSM) in the case of Emergency Vehicle Alert (EVA) and High Beam Assist (HBA).

Other works such as [18], created a complete V2X data set using the CARLA simulator. This data set includes an ego vehicle sensing information such as cameras, GPS, speed, and orientation as well as sensory information from vehicles in the near vicinity using V2X communication. A data fusion technique is used to combine data from a weather module, map module, sensing module, and the V2X communication module for the data set to be created.

2.3 Simulation Platforms for Autonomous Research

Even though autonomous vehicles are ramping up in popularity and production numbers, it remains somewhat difficult to enter such a market. Car productions as well algorithms development and deployment remain very high in cost and impractically when thinking about the underlying safety of these cars on the road. To overcome this, companies as well as researchers rely on simulation and modeling software to develop their vehicles and test their algorithms and performances, this way, the costs are kept at a minimum and no human life is at risk of any danger of underperforming autonomous algorithms. Not only is real-world testing expensive and unsafe, sometimes it can be awfully hard to reproduce the same scenario or situation twice or identically, whereas in a simulated world, thousands of runs can be execute for exactly the same scenario [19].

According to the Alphabet owned company, Waymo, they state that they have driven over 15 billion miles of simulated AV driving [20]. This is something that would require an extreme number of resources and money to complete. The simulators used by these companies tend to be private software, but other open-source software that is available to the public exists and

is used by thousands of researchers in the field. Most notably, CARLA, which is used in this research, as well as CarSim, PreScan, and many others.

[19] is an extensive survey done by researchers at Wayne State University that studies different simulation platforms for autonomous vehicles and how they perform. The study focuses on various categories which are considered requirements for these simulation platforms. Most importantly, they focus on the perception capabilities, vehicle control, and vehicle path planning of the simulators. In their findings, they highlight the flexibility and powerful performance of CARLA and the accompanied user configurable API. In addition, they state that CARLA is very well suited for custom end-to-end testing of unique functionalities.

Chapter 3 Background

In our work, we created a reinforcement learning agent that benefited from V2X data exchange between different vehicles. In the first section of this chapter, we give a background of V2X and how it works. In section 3.2 we discuss the idea of reinforcement learning. Finally, in section 3.3 we give an overview of CARLA and how it was leveraged to support our autonomous driving research.

3.1 Vehicle to Everything (V2X)

Intelligent Transportation Systems (ITS) provide knowledge to vehicles through forms of communication. ITS can lead to a significant improvement in decision making, congestion, and in reducing accident risks [21]. One way to leverage ITS is using V2X communication. Vehicle to Everything (V2X) technology refers to combining vehicle-to-vehicle (V2V), vehicle-to-pedestrian (V2P), vehicle-to-infrastructure (V2I), and vehicle-to-network (V2N). V2X has great ability to improve the safety of roads and the efficiency of traffic. A new communication standard called Dedicated Short-Range Communications (DSRC) has been designed to support V2X communications and enable the wide range of safety applications that come along with it. DSRC and V2X applications depend on the real-time live transfer of data among vehicles and their surrounding infrastructure. These data messages usually encapsulate information about the vehicle state such as location, speed, acceleration, and even sensory information. The Department of Transportation (DOT) estimates that up 82% of all crashes in the United States can be addressed by the use of V2X technology [22].

The possible use cases of V2X are endless. By far the biggest use case and promise of V2X is safety. Increasing the perception and prediction of self-driving cars to be aware of their

surrounding vehicles and obstacles no matter the weather or the road conditions will have incredible impact on how safe self-driving cars will become. Another use case is the reduction of traffic congestion and thus pollution. The ability for cars to communicate with each other efficiently will allow these cars to make "green" and environment friendly decisions. Moreover, V2X will open the doors for new applications and modes to help the communication between drivers and pedestrians that do not exist yet.

Currently, the main types of technology used for V2X are the Dedicated Short Range Communication (DSRC) system and Cellular-V2X (C-V2X). DSRC uses a wireless standard called WAVE and it consists of a set of IEEE and SAE standards [22]. On the other hand, C-V2X uses Long Term Evolution (LTE). Both perform somewhat similarly, but DSRC is considered to have better range and reliability especially in applications where safety is the main concern. Hardware wise, C-V2X use 4G cellular radio and will eventually use 5G when the technology is mature enough. Most vehicles have 4G cellular radio for infotainment systems and such, so this helps a lot in the ramp up and expansion of V2X in general. The delay for mass deployment right now is that both 4G and 5G do not yet have the guaranteed latency for DSRC and C-V2X.

V2X is currently not considered a complete replacement for sensing technology on self-driving cars, rather it complements the sensors that already exist. This is because not all cars have or will have V2X installed on them, thus, the need for regular sensors such as lidar and radar to detect non-communicating cars is necessary. In addition, V2X along with regular sensors will be useful even when all cars have some forms of communication between them. The reason for this is V2X will allow different cars to transmit different sensing information from different environments, leading to more accurate perception in all cars

3.2 Reinforcement Learning (RL)

Reinforcement learning is a type of machine learning that allows an agent to learn by interacting with its environment to achieve a goal through a sequence of decisions. Trial and error in reinforcement learning is done using a feedback loop where an agent receives a signal based on its observation and actions. This feedback loop is shown in figure 3.1. Machine learning can be classified into supervised learning, unsupervised learning, and reinforcement learning. In supervised learning, we care about correctly predicting the next output based on a labeled training data set. On the other hand, in reinforcement learning, our task is to take the correct action based on what we observe in our environment. Reinforcement learning also differs from unsupervised learning in terms of the goal. In unsupervised learning, the goal is to find similarities and differences between data points. The goal in reinforcement learning, is to maximize the total reward.

The nature of RL is very intuitive, think of the process of training a dog to do a new trick. If the dog accomplishes the trick, they receive a delicious reward, and if they do not, they could receive a negative reward. In the same way, to get the agent to the goal, they either get rewards or penalties for their actions. The main components that make up a basic reinforcement learning problem are:

1. **The environment**: This is the place where the problem lives and where the agent operates.
2. **State**: Represents the current situation of the RL agent.
3. **Reward Signal**: Feedback signal from the environment based on an agent's actions.
4. **Policy**: A definition of what actions can be taken in each state.
5. **Value function**: A mapping of taking a certain action in a certain state to a future reward signal.

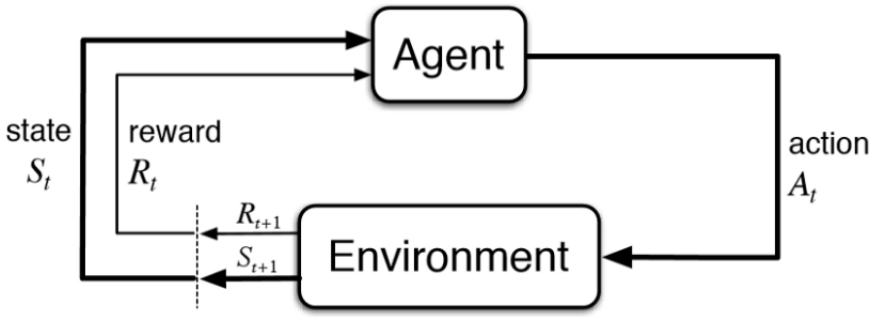


Figure 3.1: The action-reward feedback loop of a basic RL problem [23]

Figure 3.1 Shows the action-reward feedback loop for a single step in any RL problem. What can be noticed is that at any time point (t) when an agent applies an action (A_t) on the environment, it leads to being in a new state (S_{t+1}) and obtaining a new reward for taking that action (R_{t+1}). The new state and reward signals are fed into the agent and a new step is taken. The agent keeps on taking steps until the terminal state is reached. The terminal state is considered the last state in the episode where the agent achieves their desired goal. The agent tries to build what is called an optimal policy, where it has enough information to take the right action in each state in which it can land. Although, the reward policy (the value of rewards or penalties at each step) is designed by the problem developer, **no hints or suggestions are given to the model in regard to what it should do**. These actions need to be learned on its own so that it can maximize the total reward in each episode. Therefore, the model starts with random actions and trials until it can understand the problem effectively and efficiently, sometimes ultimately leading to behavior that can only be described as extraordinary.

Despite sometimes needing many trials and powerful computing machines to reach an optimal policy, a reinforcement learning agent can learn things that humans cannot. Not only can reinforcement learning learn things that humans cannot, but through complicated search

algorithms and extended trials, it can even learn sequences of actions that were previously thought to be wrong by humans.

3.2.1 Q-Learning

Q-learning is a simple value-based reinforcement learning algorithm that has the simple task of finding the **best action** take, given a **particular state** that the agent is in. Value-based algorithms use and update the value using the Bellman equation. Q-learning seeks to achieve a policy that maximizes the total reward in the problem. It does this while being an off-policy algorithm, meaning that it can learn the optimal policy without having a set of required actions to take, thus learning independently of the agent's actions.

The Q-learning agent maintains a table of values called the Q-Table. $\boldsymbol{Q}^*(\boldsymbol{s}, \boldsymbol{a})$ is an element in the table that represents expected cumulative reward of taking action a while being in state s. This is done through trial and error over many episodes without any prior knowledge of the environment or what actions are right and what are wrong. This is called **Temporal Differences (TD)**. The initial setup of the Q-table can be seen in table 1. The table consists of x columns, where x is the number of different actions that can be taken and y rows, where y is the number of different states the agent can end up. The table starts off with initial values for all pairings $Q[s, a]$ set to 0.

Table 3.1: Initial setup for a Q-Table with three states and 4 actions.

Q-Table		Actions			
		Action 1	Action 2	Action 3	Action 4
States	State one	**0**	**0**	**0**	**0**
	State two	**0**	**0**	**0**	**0**
	State three	**0**	**0**	**0**	**0**

After the table is initialized, the algorithm is run for a certain number of iterations for training. In each iteration, an action is selected and performed, and the reward is measured. After that, the values of the Q-table are updated accordingly (using the equation shown below). Note that here α represents the learning rate and governs to which extent the new information overrides the old information. The immediate reward of taking action a_t is represented by the reward r_t. The discount factor represented by γ represents the importance of future rewards. A discount factor of zero means that future rewards have new effect, and this would be considered a short-sighted model. A discount factor approaching one means that the model will aim for long-term high rewards.

$$Q^{new}(s_t, a_t) = Q(s_t, a_t) + \alpha \,.\, [\, r_t + \gamma \,.\, max_a Q(s_{t+1}, a_t) - Q(s_t, a_t) \,]$$

Table 2 shows the values after the algorithm is done. What can be noticed is that for each state, the algorithm provides a quality (Q) value of taking different actions. This means that for example, when the agent is in state one, the best action to take is action 3.

Table 3.2: Q-table after running the algorithm for a certain number of steps.

Q-Table		Actions			
		Action 1	Action 2	Action 3	Action 4
States	State one	**0.12**	**0.99**	**5.21**	**0.0**
	State two	**1.32**	**5.78**	**0.0**	**1.11**
	State three	**1.16**	**10.32**	**2.22**	**3.59**

3.2.2 Exploration vs Exploitation

One of the key problems encountered in reinforcement learning and in Q-learning is the exploration-exploitation dilemma. The goal of the reinforcement learning agent is to ultimately take the optimal action at every state, but what if the agent does not know what this action is? At the initial stages of the agent's interaction with the environment, they are not very aware of what

actions are best. This leads the agent to **explore** and gather as much information as possible so they can be more informed in the future. After gathering information and becoming more confident with the environment, the agent must **exploit** the knowledge obtained in order to select the best action possible. When to explore and when to exploit is what we mean by the explore-exploit dilemma. There are multiple approaches to deal with this problem. **Greedy** approaches tend to exploit all the time. They lock-in on the best action they know gives satisfactory results and it does not take into account other actions that actually might be better. On the other hand, **epsilon greedy** approaches try to balance between exploiting and exploring. In this approach the agent sets aside a portion of its actions with probability ϵ to explore new and interesting actions to take. The other $(1-\epsilon)$ % of the time, the agent selects the action that it considers to be the best based on experience. Another modified approach is the decayed **epsilon greedy approach**. The intuition behind this approach is that the agent starts off not knowing a lot about the environment, so the possibility for exploration ϵ is set to a high constant. As time goes on and the agent starts learning from the environment, ϵ is decayed so that the agent can exploit the best actions and reduce the number of times it tries to explore.

3.2.3 Deep Reinforcement Learning

Deep reinforcement learning is at the intersection of deep learning and reinforcement learning. It combines the complexity of deep artificial neural networks with the framework of reinforcement learning to help an agent learn optimally. In deep reinforcement learning, neural networks are used as function approximators. This is very useful when the state space is too large or ambiguous, just like in autonomous vehicles. Rather than using an incredibly large Q-table, which is impossible for large problems, neural networks can be used to approximate the value

function. This means that neural nets can be used to map states to values or state-action pairs to Q values. Deep nets can be trained on a wide range of states and actions to learn how impactful they are on the way to reaching the ultimate goal of the problem.

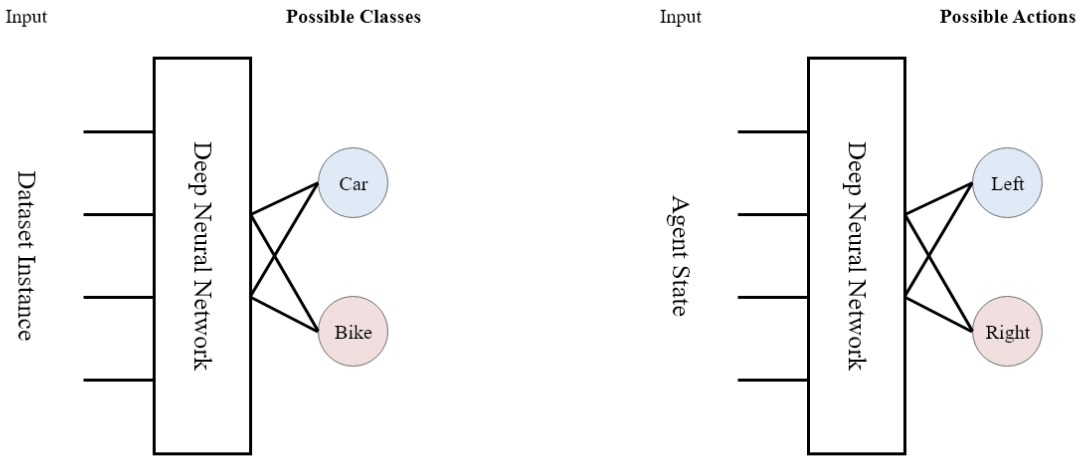


Figure 3.2: (a) shows the basic structure of a regular neural network and how it is mainly used for classification. (b) shows how neural networks in RL are used to output the best action for a given state.

One key difference between regular neural networks and the ones used in reinforcement learning is in the goal they try to achieve. Neural networks are mainly used to be able to classify data instances into their correct class. On the other hand, neural network agents are used to provide the right action to take given a certain state. This is shown in the figure 3.2.

3.3 CARLA

CARLA is an open-source simulator tool for autonomous driving research created by researchers in Intel Labs. CARLA was developed from the ground up to support the development, training, and validation of autonomous driving systems. Many digital assets such as vehicles, cars,

maps, buildings, and much more are provided by CARLA and can be used freely. In addition, the simulation tool provides many sensors of various specifications to be used in the simulation process. Full control of all actors, maps, weather, and overall environment is also included.

The simulator consists of a client-server architecture. The server is built on Unreal Engine and is responsible for handling everything related to the simulation environment. This includes the physics of the simulation, rendering different actors, and updating the world state of the environment. On the other hand, the CARLA client leverages the CARLA API to control the initialization of the scene, the logic of the actors (sensors, vehicles, pedestrians, etc.), the settings of the world, receiving sensory feedback from the server and much more. Using the provided Python application programming interface API, a developer can access all specific parts of the simulation. This includes behavior of the environment, movement of the actors, setting up scenarios, and much more. A developer can also install different types of sensors on many actors such as vehicles and bicycles. These sensors range from RGB camera sensors, GPS sensors, LIDAR sensors and much more. Through the API, the developer is able to program different settings for the sensors. This includes but is not limited to, the field of view of cameras and lidars, the response time of sensors, quality, and size of imaging sensors. In addition to the existing sensors, CARLA also allows the developer to create new sensors from the ground up. Users can build their sensors to different specifications and control them to achieve their specific objectives.

Simulators like CARLA play a key part in self-driving cars research. According to Waymo, researchers run approximately 20 million miles of simulated driving per day [20]. Waymo stats that running the simulation for one day is equivalent to 100 years of real-world driving. Simulators also allow for setting up different scenarios and running them repeatedly for training and validation purposes. This comes in handy when these scenarios are risky and unsafe to run in

the real world. Besides the repetitive nature of simulating scenarios, using simulators is also much more cost-effective especially when multiple vehicles with multiple sensors are involved. Finally, using simulators can decrease the risk of crashes that can happen at the preliminary stages of self-driving algorithms before they become mature enough for the real world.

Chapter 4 Methodology

This chapter will talk about how the system was designed. This includes the creation of the environment, creating the reinforcement learning agent, representing the V2X state in the environment, and an overview of the complete system design.

4.1 The Environment

The main goal of this project was to teach an agent (self-driving vehicle) how to act using V2X data from its surrounding environment, mainly cars in the near vicinity of this agent. The most generic form of V2X is C-V2X, which relies on cellular connectivity. Since both 4G and 5G connectivity cannot provide the latency that is required for V2X to be used in real-time and safety-critical algorithms, our focus in this project will be solely on the software aspect of integrating V2X data into a self-driving learning agent. This means that we will not go into the hardware challenges, including the networking challenges, which might arise from such a system. Moreover, for simplicity reasons, we consider an optimal channel of dataflow between all the actors in the environment. This means that packet loss or introduction of noise into data exchange will not be considered in this project.

The core of the system and the environment is the **OpenAI Gym toolkit**. The gym library provides an easy-to-use suite of reinforcement learning tasks. It also allows the development and comparison of several types of reinforcement learning algorithms. It includes many environments which are already pre-made and can be used to teach agents different objectives like playing ping pong or even the simple task of walking. **Moreover, there is included support the creation of custom environments that can be engineered to achieve specific goals and objectives.** The

main components of the Gym environment are the action space, observation space, reward signal, and the step function. The action space represents the set of actions an agent in this environment can take. The observation space represents the logical and numerical representation for the states that the agent can end up in. Both the action space and observation space can be set to various data types such as **Discrete** values, where the action space or state space represents a single number, or they can be of a **Box** data type, where the action space or state space can take a vector representation as their values. The reward signal is a measurement of how well the agent is performing at the current time step. If the reward is positive, this means that the agent took a step in the right direction towards the optimal goal or reached the end goal. If the reward is negative, this means that the agent took an action that negatively affected the progress towards the goal. An example of a positive reward in a self-driving car environment is reaching the destination, while an example of a negative reward could be causing an accident. The step function brings all the previous components together. This is because at any time "step", the reinforcement learning agent selects an action from the action space which it considers to be the best action to take. Taking this action leads to receiving either a positive or negative reward as well as causing the agent to end up in a new state from the total observation space. This means that the step function takes an action as input and returns our agent's new state and the reward for taking the action. The figure on the following page shows a basic flow diagram of the step function in the Gym reinforcement learning environment. We can see that after initializing the agent with a state from the observation space, the agent takes actions on the environment selected from the action space. If the agent is done and the goal is reached, the environment is reset so a new episode can begin. Otherwise, the agent calculates the reward for the current action and the new state and proceeds to take another action in the step function.

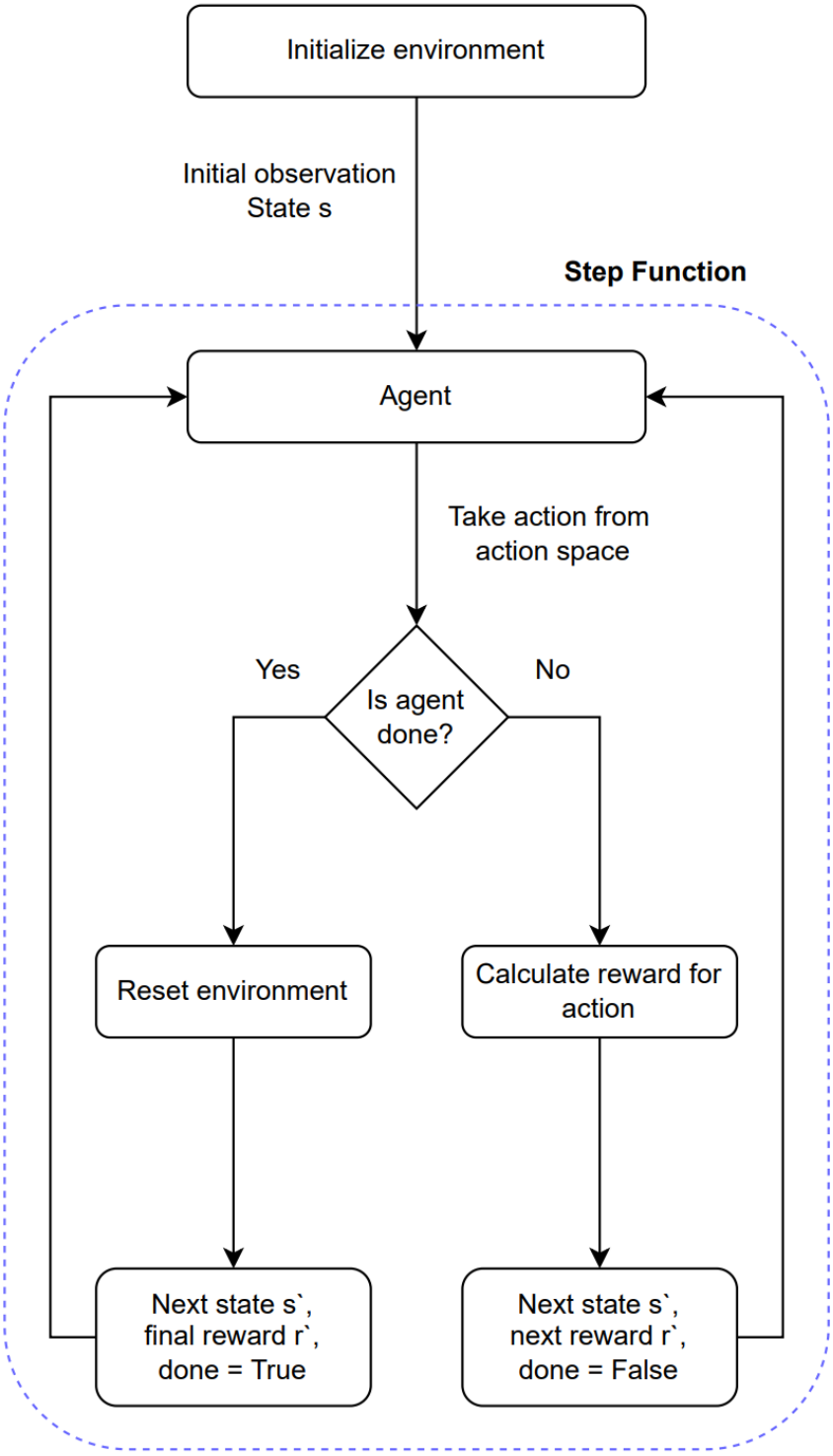


Figure 04.1: Flow diagram of the step function in a Gym environment.

The OpenAI Gym environment can encapsulate other environments inside of it for the creation of custom user environments. In our case, The CARLA World is encapsulated as an environment by the Gym environment. This means that the system provides all the manipulations needed for the gym environment while having exposure to the CARLA environment and all the actors in the CARLA World object. This architecture allows the interaction between all the Gym environment components and the CARLA world. The step function can take actions on different actors (vehicles, pedestrians, etc.) in the world and as a result of these actions produce new states that the agent can end up. While the behavior of these different actors can control the reward signal the agent receives and ultimately control the learning process.

4.2 The Agent

In reinforcement learning, the agent is the entity responsible for taking the correct action depending on the current state the agent is in. This agent receives the observation of the current state as well as the reward and must decide which action is the correct one to take to get closer to the final goal. The agent is made up of two components: the policy and the RL algorithm, this can be seen in figure 4.2. Simply put, the policy is a mapping between observations and actions. This can be done using a simple lookup table it can be as complex as a deep neural network with tunable parameters that approximates the correct action. The learning algorithm oversees trying to find the optimal policy that always leads to the best action in any case. It keeps track of previous experiences related to states, actions, and rewards and updates the policy in a manner that achieves the highest long-term reward for the agent.

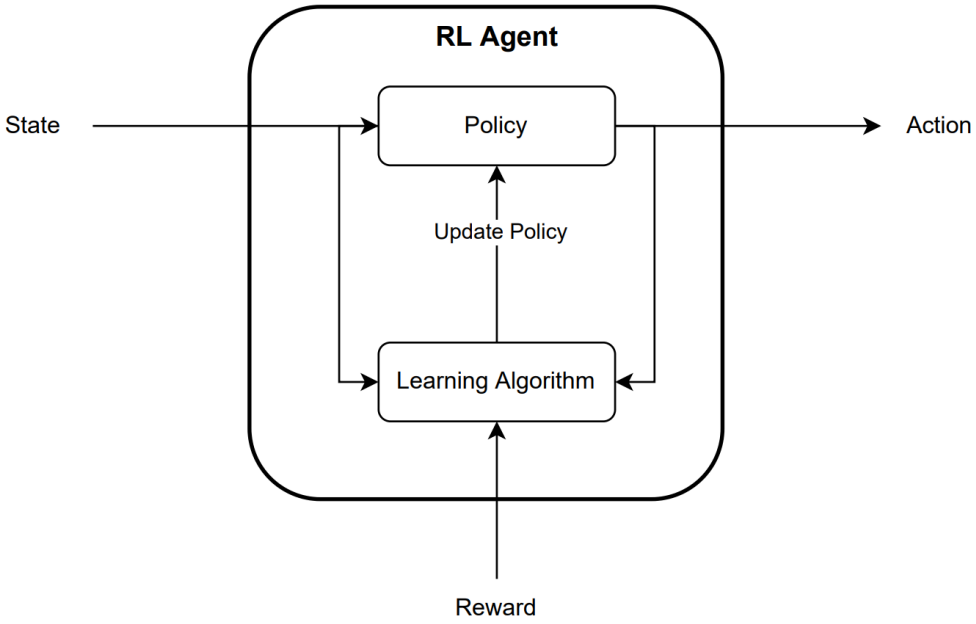


Figure 04.2: Main components of a reinforcement learning agent

The selection of which type of agent to use usually depends on the problem being solved. The simplest agent is a Q-Learning agent that is represented by a table of state-action pairs. This is used to select the action with the highest expected future reward. This agent is somewhat effective when the observation is space is known and not very huge. When the state space is bigger, as in our example of a self-driving car environment, a Q-Learning lookup table would become impossibly huge. The best practice for selecting an agent type is usually starting with the simpler, and faster to train, algorithm that works with the required action space and observation space. As will be discussed in the following chapter, the action space in our problem is discrete and the observation space is a continuous float vector. The simplest algorithm and the easiest to implement in this case is a DQN agent. The DQN agent consists of three main components which are the policy, the memory, and the deep neural network. In our agent, the selected policy is an epsilon greedy policy as discussed in chapter 3. The main goal of an epsilon greedy policy is to try to solve

the exploitation-exploration problem. To help with this task, epsilon is set to be linearly decaying so that it starts off with a high value, when the agent does not have much experience, and slowly decays upon gaining further experience about the environment. The memory stores the last N experiences that the agent went through. Where N is a finite limit defined by the user. At time t, the agent's experience e_t is stored in this memory as the following tuple:

$$e_t = (s_t, a_t, r_{t+1}, s_{t+1})$$

This memory entry keeps track of the current state s_t, the action a_t and the corresponding reward and next state. This tuple represents a summary of the agent's experience at time t. All the different experiences over all the different episodes that the agent goes through, are stored in this memory. The final component of DQN architecture is the deep neural network. The deep neural network is passed a state from the environment and in return the network outputs the expected q-value for each action from the action space.

4.3 V2X State Representation and State Reduction

One of the most important aspects of having an effective learning agent is representing the state space in an efficient way. If the state space representation is ridiculously small and most observations lead to similar states than our learner will not be able to differentiate between them. If the state space is huge and every time step produces a unique and never-before-seen state, then it becomes nearly impossible for the learner to learn all these unique states. The state space represents the input for our DQN agent, and it is necessary for the agent to learn how to behave in the environment.

To have effective V2X communication between different vehicles in the simulator, we allow the cars to exchange information about their precise GPS location as well as their velocity. Since we are using CARLA simulator, all the required data for the communication between vehicles is already exposed by the environment. This allows us to represent the vehicles state in an effective way. CARLA allows the creation of custom sensors such as V2X sensors. This requires the program to be built from source and complicates things a lot. For simplicity reasons, we take advantage of having access to the CARLA world object. Using this object allows us to access any data property of any actor in the simulation. We can easily use this data to create our very own representation of V2X data exchange.

In this research project, we consider a single vehicle to be the "ego vehicle." Although we assume parallel communication between all the cars in the simulation, we focus on the ego vehicle receiving V2X data from all the other cars in the simulation. Using the Carla environment, the ego vehicle has access to the locations and velocities of other actors in the simulation. The challenge then becomes how do we represent the data received from other vehicles into a state which is fed into the DQN agent? The simplest way to represent the V2X data is by using matrix representation. Each row represents an instance of V2X data about a single vehicle. We can clearly see this in the matrix below where each vehicle's GPS location as well as velocity is shown. If the ego vehicle does not receive any V2X data from any surrounding vehicles, the matrix is zero-padded to signify an empty state of V2X communication.

car	$X-Location$	$Y-Location$	$Velocity$
car 1	120.33	5.12	75.22
car 2	150.12	17.13	54.31
⋮	⋮	⋮	⋮

The problem with taking the absolute values of GPS coordinates, is that these values can have a significantly extensive range and it becomes almost impossible to include all the different GPS locations that the cars can have. Another problem with this approach is repetitiveness. This is because we can have remarkably similar road scenarios that produce different GPS coordinates and complicates the learning process for the agent. This is illustrated in the figure below. We can see that all three scenarios are somewhat similar, and the relationship between the ego vehicle (in blue) and trailing vehicle is identical. Using absolute GPS coordinates will produce three different states and will only make the state space larger. This will significantly increase the time needed for the agent to learn.

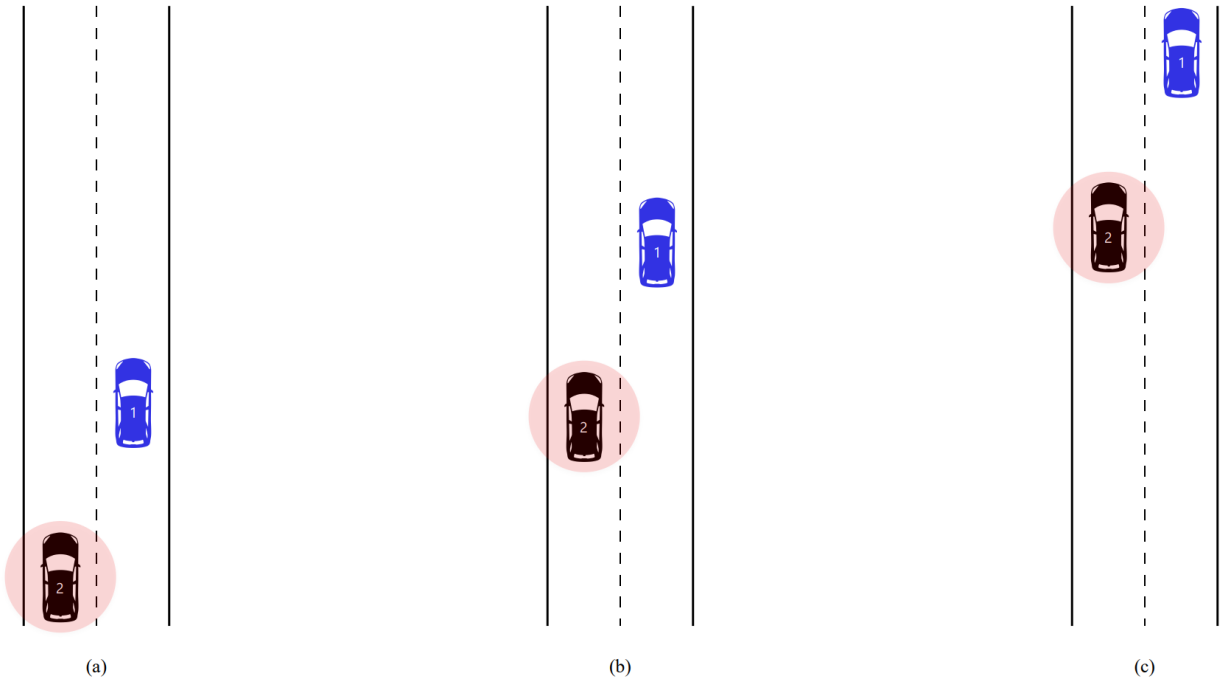


Figure 4.3: Using absolute GPS coordinates produces different V2X state representations in all the scenarios above. Even though the three scenarios look similar, the GPS exchange (represented by the red circle) between the ego vehicle (in blue) and the trailing vehicle (vehicle number 2) produces different states.

To overcome this, we propose using a relative vector representation between the location of the ego vehicle and all the vehicles in the near vicinity. Let us assume that the state of the ego vehicle is as shown below:

car	$X-Location$	$Y-Location$	$Velocity$
$ego\ vehicle$	170.66	55.32	36.25

We can then represent the V2X state by taking the difference of the X-location and Y-location of the GPS coordinates between the go vehicle and all the cars in the near vicinity. The final state representation becomes as follows:

car	$X-Location$	$Y-Location$	$Velocity$
$ego\ -\ car\ 1$	50.33	50.20	75.22
$ego\ -\ car\ 2$	20.54	38.19	54.31
⋮	⋮	⋮	⋮

Using relative state representation instead of absolute representation, means that scenarios and cases that look alike on a map, will have the same observation state fed into the DQN agent. This makes it much simpler and quicker for the DQN agent to learn different behaviors. Variations and locations or rotations of the vehicles relative to the main vehicle become irrelevant and do not contribute nor make a difference in the learning process. We will see how this helps significantly with our scenario of car passing or lane-changing in the next chapter. The figure on the following page shows how multiple scenarios that look similar will have the same vector representation for their states.

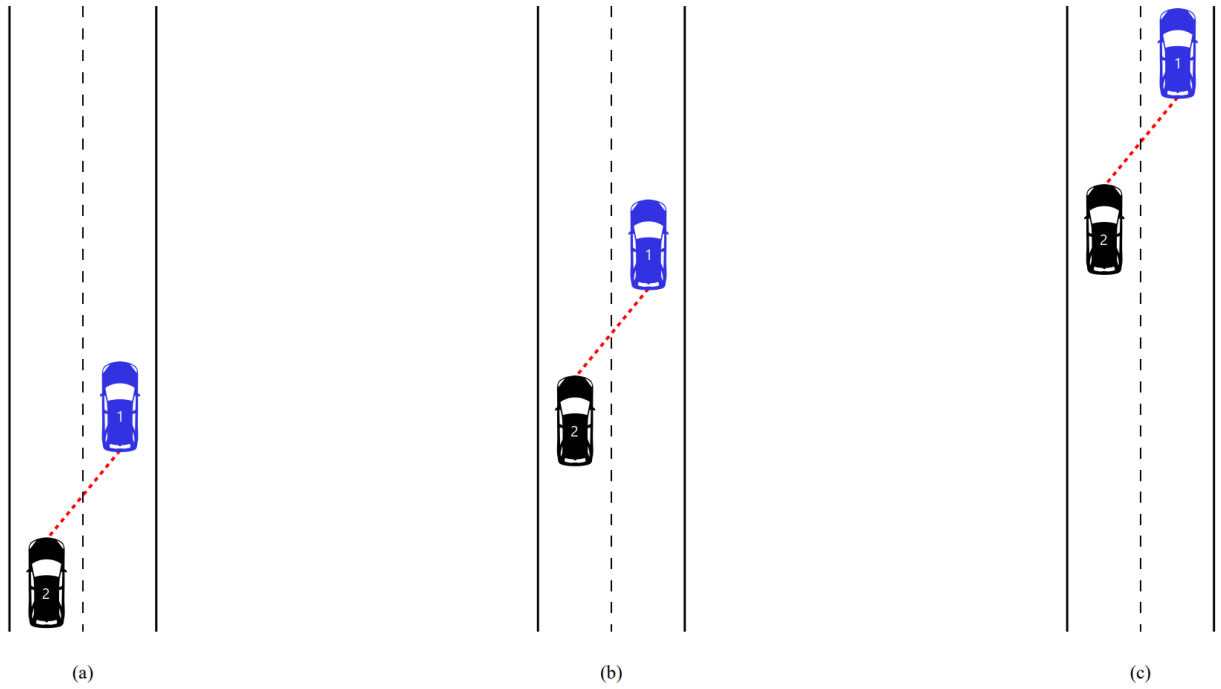


Figure 04.4: Using relative GPS coordinates produces the same exact V2X state representations in all the scenarios above. Despite the cars being in different locations, the state vector between the ego vehicle (in blue) and the trailing vehicle (vehicle number 2) produces the same vector which is shown in the dashed red line.

4.4 System Design

In our purpose system, we encapsulate the Carla environment with the open AI gym environment and all its functionality. The DQN agent, along with its newly learned policy and deep neural network, takes control of which actions to take. The V2X communication data is exposed from the Carla environment, and then it is converted into vector form for efficient state reduction. Note that the extraction of V2X data and which data to prioritize and which data to discard depends on the goal of the system. This can also be said regarding the reward signal. If the goal of the system is to teach an agent to stop at a red light, then there will be a high reward for completing this objective. On the other hand, the reward should penalize an agent for causing a collision in a system that has an objective of car passing. Figure 4.5 pieces all the components

together and shows a representation of the proposed system. Specifics that have to do with our experiment of car passing will be discussed in the coming chapter.

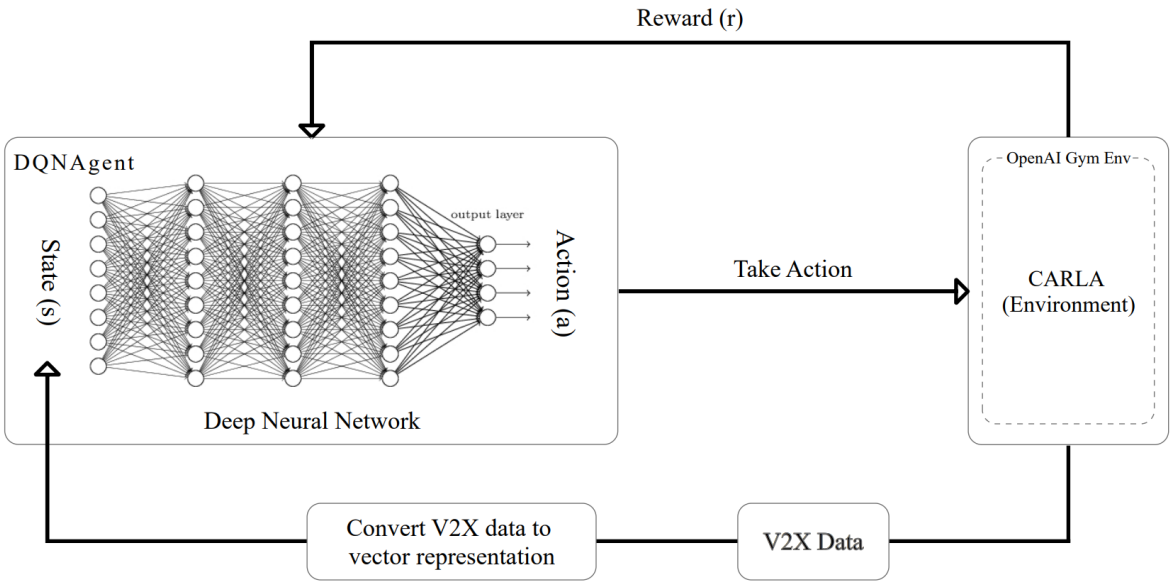


Figure 04.5: System design that shows the relationship between the OpenAI Gym environment, the CARLA environment, and the reinforcement learning agent.

For the training of the learning agent, a four-layer neural network was used. The input layer contains the state representation of the V2X communication, while the output layer contains, which action to take. Each layer of the neural network contains 64 units, all of which use a Rectified Linear Unit (ReLU) activation function. The number of training steps varies based on the scenario considered. For all training runs, a decaying epsilon-greedy approach with a max value of 1.0 and a min value of 0.1 was used.

Chapter 5 Experiments and Results

Our proposed system can be generalized for many self-driving car scenarios and experiments. This can be done by altering the reward signal accordingly as well as changing which V2X data to consider. Regardless of which scenario the agent is observing, main goal is to show that the agent will be able to learn how to behave correctly and safely. In this research, focus on the scenario of lane changing to pass a stopped emergency vehicle. This is the same scenario discussed previously in chapter 1 and can be seen in figure 1.1. Moreover, scenarios that have to do with lane changing or passing other cars can show how regular sensors such as cameras struggle especially in severe weather or around curbs whereas V2X communication is invariant to these challenges.

5.1 The Scenario

For this scenario, The CARLA map asset called “Town04” was used. The scenario contains three main types of actors. The first is the ego vehicle. This is the vehicle that will act as our learning agent and try to use all the V2X communication data from surrounding vehicles. The second actor is the stopped emergency vehicle which our agent will try to pass. The third and final type of actors are the passing vehicles which will be transmitting their V2X data to our learning agent. The learning agent must bypass the stopped emergency vehicle without hitting any of the passing actors in a timely manner. This scenario takes place on a multi-lane highway to demonstrate that the agent can learn with the existence of multiple cars and not just one other car. An illustration as well as a live image from the simulation environment is shown in figure 5.1.

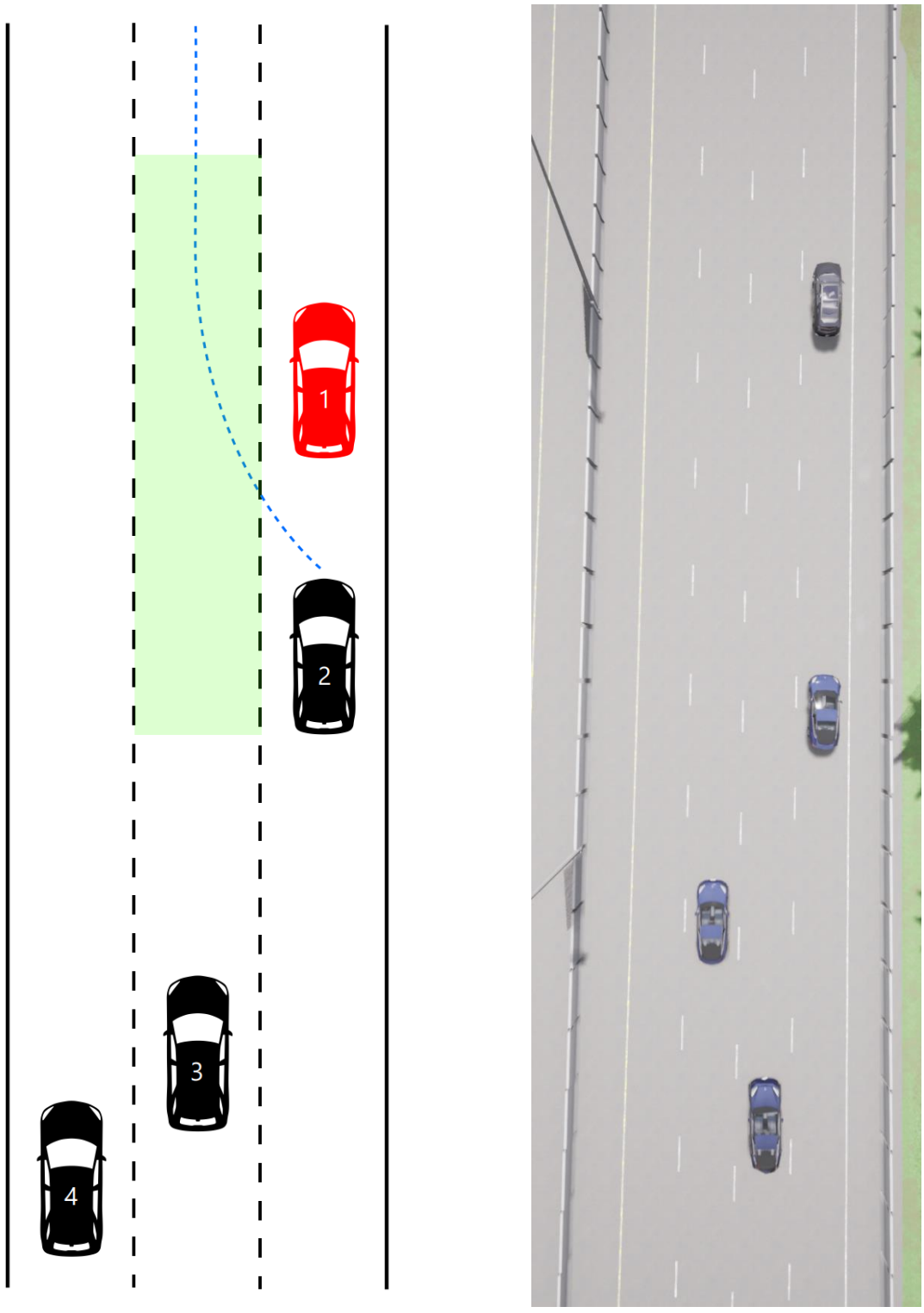


Figure 5.1: (left) Shows an illustration of the car passing scenario taught to the learning agent. (right) Shows the live view of the scenario being run in the CARLA simulator.

The scenario is considered done and successful if the ego vehicle commits to the passing action and reaches the target destination (the destination for the ego vehicle is in front of the stopped emergency vehicle). The scenario fails if the ego vehicle crashes into any of the vehicles on the road or gets stuck without taking any actions. To effectively teach the learning agent this behavior, the following rewards were set for the corresponding actions:

Table 5.1: Action / Reward table that was used to teach the agent the scenario

Action	Each time step	Causing a collision	Reaching destination
Reward	- 1,000	- 1,000,000	1,000,000

The negative reward for each time step is to encourage the agent to proceed with the passing action as fast as possible while also taking safe actions and causing a collision, hence the exceptionally large negative reward for doing so. In the case that there are no vehicles behind the ego vehicle or the cars behind are terribly slow, then the ego vehicle must proceed with the car passing actions without any stoppage at all.

5.2 Setting Up the Scenario

The scenario was set up using the open source simulation platform CARLA. The entry point to the scenario is the configuration file that controls all aspects of the simulation. It is responsible for the weather conditions in the scenario, spawn locations and destinations of all the vehicles, training and testing episode counts, the location of the spectator camera, and much more. After the configuration file has been loaded and applied to the CARLA environment, a new Gym environment is created along with the DQN agent responsible for teaching the ego vehicle the correct behavior for the scenario. The configuration file controls the complete path which the

passing vehicles will follow as well as the ego vehicle. In addition, a collision sensor is attached to the ego vehicle and keeps track of any collisions that happen. If during any episode a collision does happen, that episode is considered terminated with a large negative reward. CARLA provides a longitudinal and lateral controller which can be used to set the starting and ending location for any actor in the simulation. These controllers are used for both route planning and control. Both were utilized to allow the passing vehicles to follow a straight path along the stopped emergency vehicle and the ego vehicle. They were also used to control the route that the ego vehicle will take. This makes it simple for the agent and narrows down the actions it can take to either follow the path provided by the controllers or apply the brakes and stop for other vehicles passing by.

At any timestep of any episode during the simulation, the agent creates the V2X state representation. This is the same representation which was discussed in chapter 5 of this thesis. This state along with the reward achieved is used by the DQN agent to create a replay memory of state-action value pairs. To help reduce the state space and only allow for V2X communication between affecting vehicles only, the ego vehicle only considers vehicles in the near vicinity. For the scenario of car passing, This near vicinity is defined as all cars 100 meters to the back of the ego vehicle and 40 meters front of the ego vehicle. Vehicles outside this scope have no major effect on the decision making and cannot contribute negatively to things such as collisions. The number of vehicles that are accounted for in the state representation is a constant set in the configuration file. If the number of cars in the near vicinity is less than this constant, then the vector is zero-padded. If no vehicles are in the near vicinity of the ego vehicle, then the whole vector is zero-padded. Considering only the near-vicinity cars reduces the observation space significantly. Our learning agent does not need to learn how to pass while accounting for a car on a completely different

highway far away. Figure 5.2 illustrates and compares the range of V2X communication that was considered.

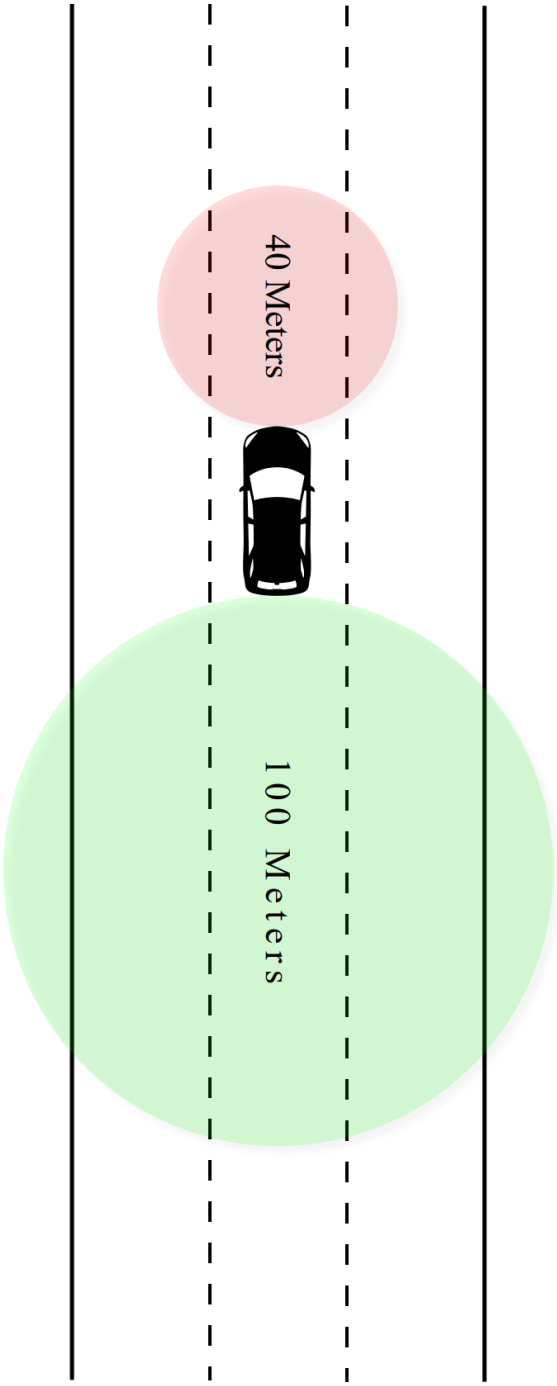


Figure 5.2: This figure shows the range of the V2X communication which is accounted for in the car passing scenario.

During the initialization of any episode, the passing vehicles starting locations are randomized and selected from a pool of distinct locations. Moreover, the speeds of these vehicles are also changed every episode. More than **4,000 different combinations** of passing vehicle locations and their speeds exist. This makes for a very realistic car passing scenario. Not only does the learning agent need to learn how to behave in reference to other cars various locations, but it also needs to learn how to maneuver while thinking about other cars speeds.

Another piece of the scenario setup is controlling the environment weather. For this project, we consider the cases of clear weather, rainy / foggy weather, and harsh weather which is a combination of rain and fog during very dim night conditions. V2X communication is invariant to weather, but these weather parameters will come in handy when comparing this solution with another that relies on the use of cameras or other sensors that might suffer in harsh weather.

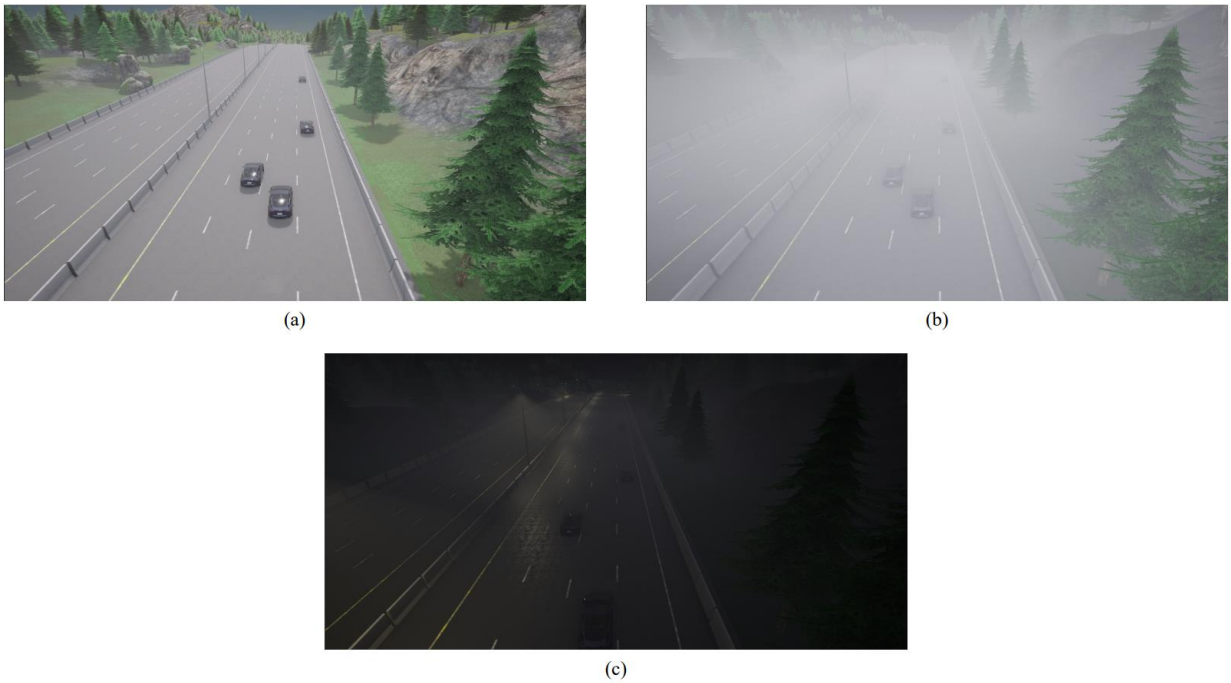


Figure 05.3: Different weather settings used throughout the simulation. (a) shows clear and sunny weather. (b) shows foggy and rainy weather. (c) shows harsh weather (night and rain)

5.3 Modifying CARLA Simulator

As previously mentioned, The controllers already provided by Carla were used to control route planning and movement of the passing vehicles. For the ego vehicle, a custom passing agent was created that relies on The existing controller. This modification was made to account and apply the different actions our DQN agent could take. The main responsibility of this custom agent is to identify the location of the stopped emergency vehicle and decide when the passing behavior should begin. Once the emergency vehicle is identified, the environment is notified that the passing behavior can start. The passing behavior starts by setting the nearest left lane as a destination for the ego vehicle, in order to pass the stopped emergency vehicle. Other than that, custom agent functions in the same manner as the original controllers. The only difference is that the input provided is dependent on the DQN agent.

5.4 Key Performance Indicators Considered

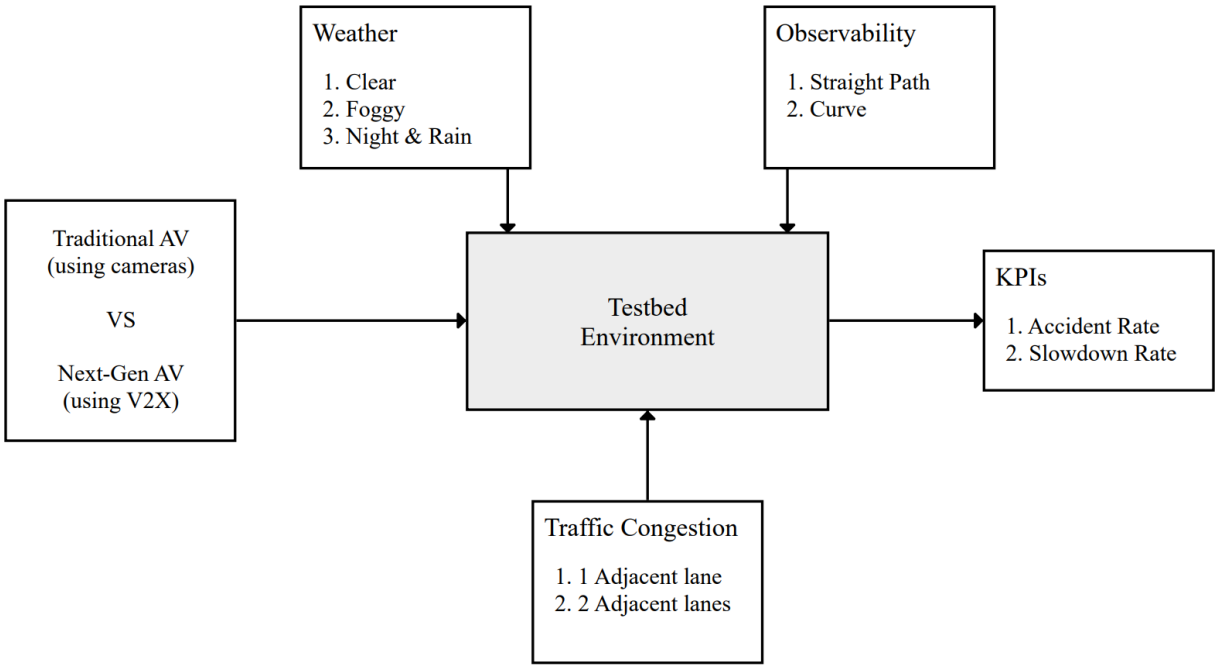
Key Performance Indicators (KPIs) are quantifiable measures that tell us how good the performance was over time for a specific objective. In our case, the objective was safely passing a stopped vehicle and changing lanes without causing any collisions. The first and most important KPI that was selected and focused on was the number of collisions over a certain number of tests. After all, the goal of this project is to demonstrate that communication between vehicles in the case of lane changing will improve safety, this means that the number of collisions needs to be as **minimal** as possible. The second KPI considered was the time needed to complete the passing maneuver. Even though we want a very safe system with zero collisions, we also want a system that performs efficiently. Self-driving cars always have the option to wait infinitely until all the

cars on the road are gone and then perform their intended maneuver. The problem with this approach is it contradicts one of the fundamental core goals of autonomous driving - reduction of traffic congestion. Think of a passing car that is traveling very slowly, in this case the learning agent needs to step on the gas and perform the passing maneuver without waiting for the slow car. An optimal learning agent is one that performs the maneuver in a safe and quick manner.

To calculate the time efficiency of our passer, we first calculate the time needed to perform the passing maneuver without any oncoming traffic at all. This is basically the time needed for the agent to move from the starting point until the end point of the scenario. After that, we calculate the average time per episode over all the episodes that are run. We then find the **slow-down rate** that occurred while performing the passing maneuver. For example, if the time needed to perform the scenario without any other cars is 10 seconds, and after performing 10,000 episodes we found out that the average time to perform an episode is 15 seconds, we can say that a slowdown of 50% has occurred.

The reason we do not just calculate the average time per episode and see how it changed in different weather conditions as well as in the V2X case and the camera case is because the simulation platform performs differently in these conditions on our single GPU machine. When running the simulation in fog or rainy weather, the time needed per episode increases, this is because CARLA has to compute and allocate much more GPU resources for these scenarios. The same thing goes for when using a camera as opposed to V2X. The more sensors in the simulation, the more time and delay CARLA needs to finish an episode. Using a rate that compares between the same scenario setup when cars are present and when cars are not, is the most accurate way to calculate how the passer was affected and how much slow down occurred because of other cars.

The complete testbed environment that takes into account all conditions and factors is shown below. Our testbed takes these different configurations as input for each run and produces the KPIs associated with that specific run. We have a total of 12 configurations when a traditional AV that uses cameras is used and another 12 when V2X is used. Making our testbed support a total of 24 different scenarios.



5.5 Comparing with RGB Camera

To compare our project with a system that relies on regular sensors, we created a simple system that houses backward facing camera and uses that to decide on whether to pass or not. This is a basic system that uses object detection to identify traffic in the adjacent lanes. The object detector uses a pre-trained YOLO-V3 object detection model that can accurately detect cars. This system was created to be an overly cautious system that immediately stops when it senses traffic behind it that might cause a collision in the passing action.

We are well aware of the fact that self-driving cars with regular sensors rely on multiple sensors together. In addition, multiple types of sensors could be used along with a complex sensor fusion algorithm for object detection and obstacle avoidance. It is overly complicated and time-consuming to create such a system, therefore we used a simple, cautious system that uses a single camera with a very wide field of view that can see all the cars behind it. Even though cars with complex fusion systems perform better than this, the sensors they use still struggle and conditions like fog, snow, and rain. Furthermore, the use of these sensors, even with complex systems, will greatly struggle in scenarios such as experiment number two. This is because there is no direct physical line-of-sight between these sensors and the cars or objects they are trying to detect.

5.6 Experiment 1: Straight Path Passing

The first experiment that was run is the simple experiment illustrated in figure 5.1. This experiment our learning agent had to pass the stopped emergency vehicle while accounting for oncoming traffic on a straight path highway where all the cars were always visible to each other. This seems like a trivial scenario where all systems should perform well, but it was very surprising how hard it could be for some systems to perform this simple task and conditions that were not ideal. This will be shown later on when comparing our V2X learning agent with another learning agent that relies on a backward facing camera for detecting oncoming traffic. The experiment was run multiple times in different conditions. In the first run, we considered only the adjacent lane to the ego vehicle having cars passing through it. Secondly, we considered multiple lanes (the two adjacent lanes) having cars pass through them. Finally, the same previous two runs were conducted in the different weather conditions which were discussed previously. Each run was done for a total of 2,000 different episodes and the total number of collisions and average number of steps for each

episode were reported. Table 5.2 summarizes the results of running this experiment in the different weather conditions as well as the different lane setups. Table 5.3 summarizes the results of running this experiment in the different weather conditions using the camera-only algorithm.

Table 5.2: Results of running the V2X algorithm on the straight path experiment

Adjacent Lanes	**Weather Condition**	**Successful passing rate**	**Slow-down Rate**
1 car in the adjacent lane	Clear	99.70 %	38.50 %
1 car in the adjacent lane	Fog + Rain	99.65 %	43.70 %
1 car in the adjacent lane	Harsh (Night + Rain)	99.70 %	46.93 %
2 cars in the two adjacent lanes	Clear	97.80 %	55.52 %
2 cars in the two adjacent lanes	Fog + Rain	97.65 %	62.58 %
2 cars in the two adjacent lanes	Harsh (Night + Rain)	97.70 %	58.88 %

Table 3.3: Results of running the simple cautious camera algorithm on the straight line as well as the curve (no direct line-of-sight) experiments.

Adjacent Lanes	**Weather Condition**	**Successful passing rate**	**Slow-down Rate**
1 car in the adjacent lane	Clear	97.80 %	61.56 %
1 car in the adjacent lane	Fog + Rain	73.10 %	66.63 %
1 car in the adjacent lane	Harsh (Night + Rain)	61.00 %	58.48 %
2 cars in the two adjacent lanes	Clear	97.40 %	77.21 %
2 cars in the two adjacent lanes	Fog + Rain	97.30 %	72.31 %
2 cars in the two adjacent lanes	Harsh (Night + Rain)	62.50 %	59.66 %

5.7 Experiment 2: Passing With Cars Around a Curve

Since the first case might seem trivial for most sensing systems (even though cameras struggle in harsh conditions as will be shown later on), we wanted to challenge the learning agent with a tougher task. In this experiment, most of the parameters and configurations are the same as the first one, the main difference is the location of the cars. Here, the ego vehicle tries to pass the stopped emergency vehicle with traffic in the adjacent lanes coming from around a partially hidden road curve. This means that the ego vehicle physically cannot see nor sense the cars in the adjacent lanes. There is no viable way for sensors such as lidar and cameras to detect these oncoming objects. On the other hand, using V2X communication with precise GPS location transmission ensures that the ego vehicle is aware of the other cars even though it cannot physically see them. This is considered a partially observable Markov decision process (POMDP) and it is a generalization of a regular Markov decision process (MDP). In a POMDP The learning agent cannot directly observe the state of the environment. An abstract representation of this scenario is shown in figure 5.4. Table 5.4 shows how our V2X algorithm performed in this experiment, and table 5.5 shows how the camera-only cautious algorithm performed.

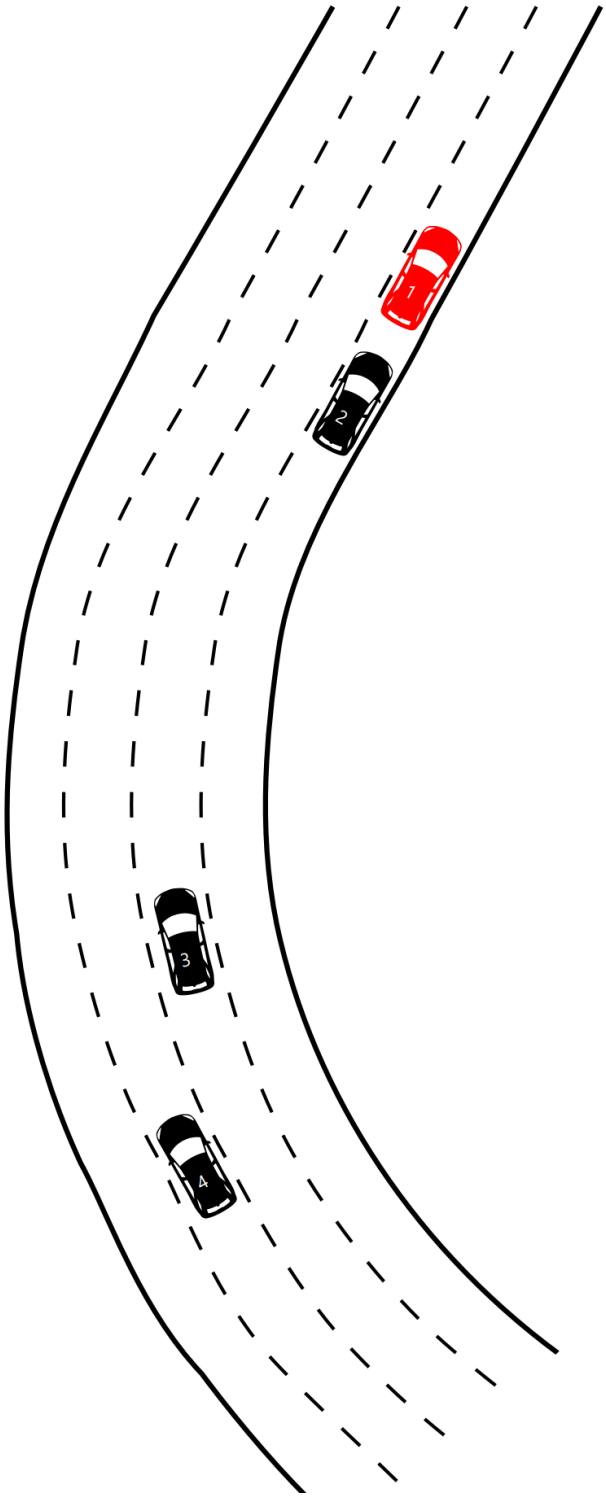


Figure 05.4: In the curb scenario, car 2 needs to pass the emergency stopped vehicle car 1. Car 2 needs to be aware of adjacent lanes to make sure that it does not hit car 3 or 4. In this case, cars 3 and 4 can become unobservable when they are hidden around a curve and cannot be detected by regular sensors such as cameras.

Table 5.4: Results of running the V2X algorithm on the curve (no direct line-of-sight) experiment

Adjacent Lanes	Weather Condition	Successful passing rate	Slow-down Rate
1 car in the adjacent lane	Clear	93.00 %	31.40 %
1 car in the adjacent lane	Fog + Rain	93.35 %	27.93 %
1 car in the adjacent lane	Harsh (Night + Rain)	92.25 %	31.80 %
2 cars in the two adjacent lanes	Clear	94.00 %	81.90 %
2 cars in the two adjacent lanes	Fog + Rain	93.90 %	76.46 %
2 cars in the two adjacent lanes	Harsh (Night + Rain)	94.15 %	64.58 %

Table 4.5: Results of running the simple cautious camera algorithm on the straight line as well as the curve (no direct line-of-sight) experiments.

Adjacent Lanes	Weather Condition	Successful passing rate	Slow-down Rate
1 car in the adjacent lane	Clear	87.65 %	39.97 %
1 car in the adjacent lane	Fog + Rain	83.95 %	39.26 %
1 car in the adjacent lane	Harsh (Night + Rain)	60.15 %	35.36 %
2 cars in the two adjacent lanes	Clear	92.50 %	82.76 %
2 cars in the two adjacent lanes	Fog + Rain	80.70 %	61.37 %
2 cars in the two adjacent lanes	Harsh (Night + Rain)	69.60 %	58.62 %

Chapter 6 Discussion of Results

In both of our experiments, we have shown how effective the use of V2X technology can be. In the straight path experiment, the V2X solution was very impressive regardless of the weather or number of cars in the adjacent lane. In the case of having one car in the adjacent lanes, the V2X solution was able to achieve a passing rate of approximately 99%, and when the number of cars increased to two, the passing rate was well above 97%. Although there was a 3% decrease in passing rate when two cars are in the adjacent lane, due to the state space being much larger, the solution is still very impressive. Another aspect to look at is the slowdown rate. In the straight path experiment, the slowdown rate was kept at a minimum (between 38% - 47%) and the car only slowed down when it needed to avoid a collision. The same thing can be said about having two cars in the adjacent lane when it comes to the slowdown rate, the car becomes more cautious and thus slows down a little bit more (between 55% - 63%) due to having more cars on the road. When looking at how the cautious camera only algorithm performed in the straight path experiment, one can tell that it performs quite well when weather conditions are ideal or close to ideal. But when the weather conditions ramp up or trend towards being harsh, it significantly struggles to keep up with the V2X algorithm. This is evident from the passing rate going all the way down to around 60% in harsh conditions. Moreover, since this is a cautious system, it tends to slow down more than the V2X algorithm. When comparing the V2X solution to the camera only solution in experiment one, it is clear that the V2X solution not only reduces the number of collisions, but it will also help in reducing the overall traffic congestion in a scenario such as our experiment.

Experiment two shows the real potential for using V2X sensors. This is because even in ideal conditions, regular sensors will struggle when dealing with an unobservable environment

such as a road curve. Even though this scenario is harder for the V2X algorithm to learn, it still achieves better results than the use of a camera-only algorithm. One thing that can be noticed in this experiment is that even in ideal or close to ideal conditions, the camera-only algorithm is far from perfect and still causes a lot of collisions. In harsh conditions, the camera-only algorithm caused a collision almost half of the times the experiment was run, which is a drastic and dangerous number when it comes to autonomous vehicles safety.

We believe that this experiment has great implications on self-driving cars research. V2X has still not been explored to the fullest, nor has it been used in many real-life algorithms, but we have shown that the potential for using such systems is significant.

Chapter 7 Conclusion

Summary

We have shown the significance of using V2X in self-driving algorithms and how it can easily outperform simple camera algorithms. Using our reinforcement learning agent, we were able to create a robust car passing and lane changing algorithm that can safely maneuver our car and in a timely manner. In addition, we have shown that V2X algorithms are immune to weather changes and road conditions by demonstrating that they perform similarly in all weather conditions, something that regular sensors struggle in.

Despite demonstrating the benefits of V2X in just two experiments over multiple scenarios, we believe that the use of V2X can benefit other examples on the road. V2X can significantly improve congestion, help with intersection crossing, send live updates to cars about road conditions, and much more.

Limitation and Future Work

Research never ends, and there is always room for improvement. An aspect that limited our project was the use of a single GPU machine. Intense machine learning projects require a lot of resources, especially in the learning and training process. Using a multi-GPU server and allowing the project to be trained for millions of more steps, would significantly improve the model that was achieved. For our future work, we plan to explore different experiments other than the car passing scenario to see how effective V2X can be in other aspects of self-driving cars. In addition, we plan to create custom V2X sensors that can act like real V2X sensors with the addition of packet drops and delays.

References

[1] E. Edmonds, "AAA: Today's Vehicle Technology Must Walk So Self-Driving Cars Can Run," American Automobile Association (AAA), 25 February 2021. [Online]. Available: https://newsroom.aaa.com/2021/02/aaa-todays-vehicle-technology-must-walk-so-self-driving-cars-can-run/. [Accessed 2 December 2021].

[2] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez and V. Koltun, "CARLA: An open urban driving simulator," in *Proceedings of the 1st Annual Conference on Robot Learning*, 2017.

[3] B. Gringer, "History of the Autonomous Car," TITLEMAX, [Online]. Available: https://www.titlemax.com/resources/history-of-the-autonomous-car/#:~:text=In%20GM's%201939%20exhibit%2C%20Norman,made%20this%20concept%20a%20reality.. [Accessed 03 December 2021].

[4] K. GAMMON, "Future Past: Self-Driving Cars Have Actually Been Around for a While," Car And Driver, 16 November 2016. [Online]. Available: https://www.caranddriver.com/news/a15343941/future-past-self-driving-cars-have-actually-been-around-for-a-while/. [Accessed 10 January 2022].

[5] DARPA OUTREACH, "The DARPA Grand Challenge: Ten Years Later," DARPA, 13 March 2014. [Online]. Available: https://www.darpa.mil/news-events/2014-03-13. [Accessed 23 January 2022].

[6] S. Thrun, M. Mike, D. Hendrik, S. David, A. Andrei, D. James and F. Philip, "Stanley: The robot that won the DARPA Grand Challenge," *Journal of field Robotics*, vol. 23, no. 9, pp. 661-692, 2006.

[7] "Seeing the road ahead," Waymo, [Online]. Available: https://waymo.com. [Accessed 21 January 2022].

[8] Tesla, "Built for Safety," Tesla, [Online]. Available: https://www.tesla.com/safety. [Accessed 19 January 2022].

[9] Tesla, "Tesla Autopilot," Tesla, [Online]. Available: https://www.tesla.com/autopilot. [Accessed 24 January 2022].

[10] U.S. Department of Transportation, "Traffic Safety Facts," NHTSA's National Center for Statistics and Analysis, Washington, DC, 2017.

[11] R. Gonzales, "Feds Say Self-Driving Uber SUV Did Not Recognize Jaywalking Pedestrian In Fatal Crash," NPR, 7 November 2019. [Online]. Available: https://www.npr.org/2019/11/07/777438412/feds-say-self-driving-uber-suv-did-not-recognize-jaywalking-pedestrian-in-fatal-.

[12] I. Ogunrinde and S. Bernadin, "A Review of the Impacts of Defogging on Deep Learning-Based Object Detectors in Self-Driving Cars," in *SoutheastCon 2021*, 2021.

[13] P. Sen, B. Chen, G. Garg, S. R. Marschner, M. Horowitz, M. Levoy and H. P. A. Lensch, "Dual Photography," in *ACM SIGGRAPH 2005 Papers*, 2005.

[14] A. Velten, T. Willwacher, O. Gupta, A. Veeraraghavan, M. G. Bawendi and R. Raskar, "Recovering three-dimensional shape around a corner using ultrafast time-of-flight imaging," *Nature communications*, vol. 3, no. 1, pp. 1-8, 2012.

[15] N. Scheiner, F. Kraus, F. Wei, B. Phan, F. Mannan, N. Appenrodt, W. Ritter, J. Dickmann, K. Dietmayer, B. Sick and F. Heide, "Seeing Around Street Corners: Non-Line-of-Sight Detection and Tracking In-the-Wild Using Doppler Radar," in *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2020.

[16] "Aquaplaning assistance concepts," Continental, [Online]. Available: https://www.continental-automotive.com/en-gl/Passenger-Cars/Technology-Trends/Road-Condition-Observer/Assistenzkonzepte-fur-Aquaplaning-(1). [Accessed 4 September 2021].

[17] S. Rajab and R. Miucic, "Assessment of Novel V2X Applications Using a Simulation Platform," *SAE Technical Paper,* 2021.

[18] T.-K. Lee, T.-W. Wang, W.-X. Wu, Y.-C. Kuo, S.-H. Huang, G.-S. Wang, C.-Y. Lin, J.-J. Chen and Y.-C. Tseng, "Building a V2X Simulation Framework for Future Autonomous Driving," in *2019 20th Asia-Pacific Network Operations and Management Symposium (APNOMS)*, 2019.

[19] P. Kaur, S. Taghavi, Z. Tian and W. Shi, "A survey on simulators for testing self-driving cars," in *2021 Fourth International Conference on Connected and Autonomous Driving (MetroCAD)*, 2021.

[20] Waymo, "Off road, but not offline: How simulation helps advance our Waymo Driver," Waymo, 28 April 2020. [Online]. Available: https://blog.waymo.com/2020/04/off-road-but-not-offline--simulation27.html#:~:text=At%20Waymo%2C%20one%20day%20in,and%20complexity%20of%20our%20experience.. [Accessed 30 January 2022].

[21] G. Dimitrakopoulos and P. Demestichas, "Intelligent Transportation Systems," *IEEE Vehicular Technology Magazine,* vol. 5, no. 1, pp. 77-84, March 2010.

[22] J. Kenney, "Dedicated short-range communications (DSRC) standards in the United States," *Proceedings of the IEEE,* vol. 99, no. 7, pp. 1162-1182, 2011.

[23] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction (2nd Edition), MIT press, 2018.

Appendices

config.json File


```
{
  "project_mode": "model_load",
  "model_load_name": "v2x_str8_4_64_900K_Fin1-V268-2022-02-25",
  "map":"Town04",
  "weather_options": ["clear", "fog_rain", "night"],
  "selected_weather": "night",
  "clear_weather":{
    "cloudiness": 10.000000,
    "precipitation": 0.000000,
    "precipitation_deposits": 10.000000,
    "sun_altitude_angle": 60.000000,
    "sun_azimuth_angle": 150.000000,
    "fog_density": 40.000000,
    "wetness": 30.000000,
    "fog_distance": 60.000000,
    "fog_falloff": 2.000000
  },
  "fog_rain_weather":{
    "cloudiness": 100.000000,
    "precipitation": 100.000000,
    "precipitation_deposits": 10.000000,
    "sun_altitude_angle": 60.000000,
    "sun_azimuth_angle": 150.000000,
    "fog_density": 100.000000,
    "wetness": 100.000000,
    "fog_distance": 1.000000,
    "fog_falloff": 2.000000
  },
  "night_weather":{
    "cloudiness": 100.000000,
    "precipitation": 100.000000,
    "precipitation_deposits": 10.000000,
    "sun_altitude_angle": -90.000000,
    "sun_azimuth_angle": 150.000000,
    "fog_density": 100.000000,
    "wetness": 100.000000,
```

```
    "fog_distance": 1.000000,
    "fog_falloff": 2.000000
  },
  "train_steps": 1500000,
  "test_episodes": 2000,
  "train_delta": 0.035,
  "test_delta": 0.065,
  "distance_changer": [0, 5, 8, 11, 13],
  "throttle_selector" : [0.3, 0.4, 0.5, 0.6, 0.75, 0.8, 0.85, 0.9, 0.9, 1.0],
  "straight_car_count": 2,
  "spectator_location" : [310.018829, 10.253301, 8.635016],
  "front_proximity": 40,
  "back_proximity": 100,
  "location_shuffle": true,
  "agent" : {
    "spawn_location": [290.018829, 10, 3],
    "destination" : [210.018829, 9.2, 3]
  },
  "straight_cars": [
    {
      "spawn_location": [305.018829, 13.253301, 3],
      "destination" : [180.018829, 12.253301, 2]
    },
    {
      "spawn_location": [305.018829, 16.253301, 3],
      "destination" : [180.018829, 15.253301, 2]
    }
  ],
  "stopped_car":
  {
    "spawn_location": [250.018829, 9.2, 5]
  }
}
```

config_curb.json File


```
{

  "project_mode": "model_load",

  "model_load_name": "v2x_curb_5_128_1.5M_March-V438-2022-03-06",
  "map":"Town04",
  "weather_options": ["clear", "fog_rain", "night"],
  "selected_weather": "clear",
  "clear_weather":{
    "cloudiness": 10.000000,
    "precipitation": 0.000000,
    "precipitation_deposits": 10.000000,
    "sun_altitude_angle": 60.000000,
    "sun_azimuth_angle": 150.000000,
    "fog_density": 40.000000,
    "wetness": 30.000000,
    "fog_distance": 60.000000,
    "fog_falloff": 2.000000
  },
  "fog_rain_weather":{
    "cloudiness": 100.000000,
    "precipitation": 100.000000,
    "precipitation_deposits": 10.000000,
    "sun_altitude_angle": 60.000000,
    "sun_azimuth_angle": 150.000000,
    "fog_density": 100.000000,
    "wetness": 100.000000,
    "fog_distance": 1.000000,
    "fog_falloff": 2.000000
  },
  "night_weather":{
    "cloudiness": 100.000000,
    "precipitation": 100.000000,
    "precipitation_deposits": 10.000000,
    "sun_altitude_angle": -90.000000,
    "sun_azimuth_angle": 150.000000,
    "fog_density": 100.000000,
    "wetness": 100.000000,
    "fog_distance": 1.000000,
    "fog_falloff": 2.000000
  },
```

```
"train_steps": 1500000,
"test_episodes": 2000,
"train_delta": 0.04,
"test_delta": 0.045,
"throttle_selector" : [0.3, 0.4, 0.5, 0.6, 0.75, 0.8, 0.85, 0.9, 0.9, 1.0],
"straight_car_count": 2,
"spectator_location" : [391.606750, 32.900692, 17.390894],
"spectator_rotation" : [-18.276028, -133.833313, 0.000057],
"front_proximity": 40,
"back_proximity": 100,
"location_shuffle": true,
"agent" : {
  "spawn_location": [350.018829, 10, 3],
  "spawn_rotation": [350.018829, 10, 3],
  "destination" : [270.018829, 9.2, 3]
},
"straight_cars": [
  {
    "spawn_transforms": [
      {
        "location" : [384.018829, -23, 3],
        "rotation" : [0.000000, -260, 0.000000]
      },
      {
        "location" : [374.018829, 0, 3],
        "rotation" : [0.000000, -230, 0.000000]
      },
      {
        "location" : [360.018829, 10, 3],
        "rotation" : [0.000000, -210, 0.000000]
      },
      {
        "location" : [360.018829, 10, 3],
        "rotation" : [0.000000, -210, 0.000000]
      },
      {
        "location" : [360.018829, 10, 3],
        "rotation" : [0.000000, -210, 0.000000]
      },
      {
        "location" : [360.018829, 10, 3],
        "rotation" : [0.000000, -210, 0.000000]
```

```
    },
    {
      "location" : [384.018829, -30, 3],
      "rotation" : [0.000000, -270, 0.000000]
    },
    {
      "location" : [383.018829, -19, 3],
      "rotation" : [0.000000, -260, 0.000000]
    },
    {
      "location" : [383.718829, -20.19, 3],
      "rotation" : [0.000000, -260, 0.000000]
    },
    {
      "location" : [383.718829, -35.44, 3],
      "rotation" : [0.000000, -270, 0.000000]
    },
    {
      "location" : [366.758829, 5.72, 3],
      "rotation" : [0.000000, -215, 0.000000]
    },
    {
      "location" : [366.758829, 5.72, 3],
      "rotation" : [0.000000, -215, 0.000000]
    },
    {
      "location" : [366.758829, 5.72, 3],
      "rotation" : [0.000000, -215, 0.000000]
    },
    {
      "location" : [366.758829, 5.72, 3],
      "rotation" : [0.000000, -215, 0.000000]
    },
    {
      "location" : [366.758829, 5.72, 3],
      "rotation" : [0.000000, -215, 0.000000]
    }
  ],
  "destination" : [180.018829, 12.253301, 2]
},
{
  "spawn_transforms": [
    {
      "location" : [387.518829, -23, 3],
      "rotation" : [0.000000, -265, 0.000000]
    },
```

```
        {
          "location" : [377.518829, 1, 3],
          "rotation" : [0.000000, -225, 0.000000]
        },
        {
          "location" : [354.918829, 16, 3],
          "rotation" : [0.000000, -195, 0.000000]
        },
        {
          "location" : [387.115936, -22.2, 3],
          "rotation" : [0.000000, -260, 0.000000]
        },
        {
          "location" : [375.815936, 2.9, 3],
          "rotation" : [0.000000, -230, 0.000000]
        },
        {
          "location" : [382.415936, -7.1, 3],
          "rotation" : [0.000000, -240, 0.000000]
        },
        {
          "location" : [387.815936, -36.12, 3],
          "rotation" : [0.000000, -270, 0.000000]
        },
        {
          "location" : [387.815936, -49.23, 3],
          "rotation" : [0.000000, -270, 0.000000]
        }
      ],
      "spawn_location": [305.018829, 16.253301, 3],
      "spawn_rotation": [305.018829, 16.253301, 3],
      "destination" : [180.018829, 15.253301, 2]
    }
  ],
  "stopped_car":
  {
    "spawn_location": [310.018829, 10, 2],
    "spawn_rotation": [310.018829, 10, 2]
  }
}
```

V2X_passer.py

```
'''
This is an RL learning agent that teaches a single car
to pass a stopped vehicle in the least number of steps possible
there is only 2 cars here.
'''

import argparse
import collections
import datetime
import glob
import logging
import math
import os
import numpy.random as random
import re
import sys
import weakref
import datetime

try:
    import numpy as np
except ImportError:
    raise RuntimeError(
        'cannot import numpy, make sure numpy package is installed')

# ==============================================================================
# -- Find CARLA module ---------------------------------------------------------
# ==============================================================================
try:
    sys.path.append(glob.glob('../../carla/dist/carla-*%d.%d-%s.egg' % (
        sys.version_info.major,
        sys.version_info.minor,
        'win-amd64' if os.name == 'nt' else 'linux-x86_64'))[0])
except IndexError:
    pass

# ==============================================================================
# -- Add PythonAPI for release mode --------------------------------------------
# ==============================================================================
try:
```

```
    sys.path.append(os.path.dirname(os.path.dirname(os.path.abspath(__file__))) +
'/carla')
except IndexError:
    pass

import carla
from carla import ColorConverter as cc

from passAgents.navigation.behavior_agent import BehaviorAgent  # pylint:
disable=import-error
from passAgents.navigation.basic_agent import BasicAgent  # pylint:
disable=import-error
from passAgents.navigation.passing_agent import PassingAgent  # pylint:
disable=import-error
from passAgents.tools.misc import get_speed, positive, is_within_distance,
compute_distance


from random import randrange
import time
from gym import Env
from gym.spaces import Discrete, Box

from tensorflow.keras.models import Sequential, Model
from tensorflow.keras.layers import Dense, Flatten, Input
from tensorflow.keras.optimizers import Adam
import random
import json

STEPS_LIMIT = 20000
STRAIGHT_CAR_COUNT = 1
BOUNDING_BOXES_ENABLED = True
TRACKING_LINES_ENABLED = True
TRACK_CAMERA = True
PATH_ARROWS_ENABLED = False

class WorldEnv(Env):
    def __init__(self, world, old_settings, args, config):
        self.world = world
        self.config = config
        self.old_settings = old_settings
        self.pass_agent = None
```

```
        self.args = args
        self.camera = None
        self.collision_sensor = None
        self.collision_history = 0


        self.stats = {'fit': {'collisions' : 0}, 'test': {'collisions' : 0}}
        self.max_car_count = config["straight_car_count"]
        # Actions we can take, brake or take the regular agent step control
        self.action_space = Discrete(2)
        self.observation_space = Box(low=-200, high=200, shape=(1,
self.max_car_count,3))


        self.state = None
        self.not_passed = True
        self.completed_counter = 0
        self.zero_actions = 0
        self.one_actions = 1
        self.crash_counter = 0
        self.step_counter = 0
        self.spectator = self.world.get_spectator()

        # Cars
        self.final_destination = None
        self.stopped_vehicle = None
        self.main_vehicle = None
        self.distance_changer = config["distance_changer"]
        self.throttle_selector = config["throttle_selector"]
        self.throttle_values = []
        self.straight_car_count = 0
        self.straight_cars = []
        self.straight_agents = []
        self.straight_paths = []
        self.stage = 'fit'
        self.total_passing_time_steps = 0
        self.eps_start_time = datetime.datetime.now()
        self.eps_end_time = datetime.datetime.now()
        self.eps_times_total = 0
        self.did_crash = False
        self.init_env()
```

```
    def render(self):
        # Implement visualization (no need for this here since it is already
done)
        pass



    '''
    Function to get the cars behind the agent
    Note that this discards the stopped vehicle
    Takes two angels which are the low and up angle
    As shown below.
    Low-------------- AGENT CAR
                           |
                           |
                           |
                           |
                          High
    '''
    def get_cars_behind(self, back_proximity = 100):
        vehicle_list = self.world.get_actors().filter("*vehicle*")
        cars_behind = []
        ego_transform = self.main_vehicle.get_transform()
        proximity_th = back_proximity
        low_angle_th = 80
        up_angle_th = 180
        main_vehicle_id = self.main_vehicle.id
        stopped_vehicle_id = self.stopped_vehicle.id
        for vehicle in vehicle_list:
            if main_vehicle_id == vehicle.id or stopped_vehicle_id == vehicle.id:
                continue
            target_transform = vehicle.get_transform()
            if is_within_distance(target_transform, ego_transform, proximity_th,
[low_angle_th, up_angle_th]):
                # print('vehicle is behind us: ', vehicle)
                cars_behind.append(vehicle)
        return cars_behind

    def get_cars_infront(self, front_proximity = 20):
        vehicle_list = self.world.get_actors().filter("*vehicle*")
        cars_behind = []
        ego_transform = self.main_vehicle.get_transform()
```

```
        proximity_th = front_proximity
        low_angle_th = 0
        up_angle_th = 50
        main_vehicle_id = self.main_vehicle.id
        stopped_vehicle_id = self.stopped_vehicle.id
        for vehicle in vehicle_list:
            if main_vehicle_id == vehicle.id or stopped_vehicle_id == vehicle.id:
                continue
            target_transform = vehicle.get_transform()
            if is_within_distance(target_transform, ego_transform, proximity_th,
[low_angle_th, up_angle_th]):
                # print('vehicle is behind us: ', vehicle)
                cars_behind.append(vehicle)
        return cars_behind

    '''
    Function to get the cars affecting our ego vehicle.
    '''
    def get_effecting_cars(self, back_proximity = 100, front_proximity = 20):
        effecting_cars = []
        cars_behind = self.get_cars_behind(back_proximity)
        for car in cars_behind:
            effecting_cars.append(car)
        cars_infront = self.get_cars_infront(front_proximity)
        for car in cars_infront:
            effecting_cars.append(car)
        return effecting_cars

    '''
    Function to generate the number of cars in the scenario.
    '''
    def init_car_counts_in_scenario(self):
        random_gen = random.uniform(0, 1)
        if random_gen > 0.9:
            self.straight_car_count = 0
        elif random_gen > 0.45:
            self.straight_car_count = 1
        else:
            self.straight_car_count = 2

    '''
    Function to offset the cars current location to generate scenario
```

```
    randomization.
    '''
    def get_location_offset(self):
        distance_changer = random.choice(self.distance_changer)
        if random.random() > 0.5:
            return distance_changer
        return distance_changer * -1

    '''
    Function to initalize the path of straight path passing vehicles.
    '''
    def get_straight_cars_path(self):
        standard_rotation = carla.Rotation(pitch=0.000000, yaw=-180,
roll=0.000000)
        location_offset = self.get_location_offset()
        location1 = self.config['straight_cars'][0]['spawn_location']
        location2 = self.config['straight_cars'][1]['spawn_location']
        straight_location1 = carla.Transform(carla.Location(x=location1[0] +
location_offset, y=location1[1], z=location1[2]), standard_rotation)
        straight_location2 = carla.Transform(carla.Location(x=location2[0],
y=location2[1], z=location2[2]), standard_rotation)
        destination1 = self.config['straight_cars'][0]['destination']
        destination2 = self.config['straight_cars'][1]['destination']
        straight_destination1 = carla.Transform(carla.Location(x=destination1[0],
y=destination1[1], z=destination1[2])).location
        straight_destination2 = carla.Transform(carla.Location(x=destination2[0],
y=destination2[1], z=destination2[2])).location

        path1 = {'spawn': straight_location1, 'destination':
straight_destination1}
        path2 = {'spawn': straight_location2, 'destination':
straight_destination2}
        arr = [path1, path2]

        # # CRITICAL: Change this
        # Remove shuffling to make the car always be adjacent
        # if self.config['location_shuffle']:
        #     np.random.shuffle(arr)
        return arr

    '''
    Function to itialize the environment.
```

```
    '''
    def init_env(self):
        #
================================================================================
        # -- initalizing how many straight cars there is ------------------------
-------
        #
================================================================================
        self.init_car_counts_in_scenario()
        self.straight_paths = self.get_straight_cars_path()
        for i in range(0, self.straight_car_count):
            throttle = random.choice(self.throttle_selector)
            self.throttle_values.append(throttle)
            print('throttle is: ', throttle)
        # Set the spectator loaction
        spectator_location = self.config['spectator_location']
        spectator_location_transform =
carla.Transform(carla.Location(x=spectator_location[0], y=spectator_location[1],
z=spectator_location[2]),
                                                        carla.Rotation(yaw=-170,
pitch=-15))
        if TRACK_CAMERA:
            self.spectator.set_transform(spectator_location_transform)

        #
================================================================================
        # -- Getting important variables ----------------------------------------
----
        #
================================================================================
        map = self.world.get_map()
        spawn_points = map.get_spawn_points()
        blueprint_library = self.world.get_blueprint_library()

        #
================================================================================
        # -- Spawning stopped vehicles / Emergency Stopped ----------------------
-------
        #
================================================================================
        stopped_vehicle_bp =
blueprint_library.filter("vehicle.lincoln.mkz_2017")[0]
```

```
        stopped_location = self.config['stopped_car']['spawn_location']
        self.stopped_spawn_location =
carla.Transform(carla.Location(x=stopped_location[0], y=stopped_location[1],
z=stopped_location[2]), carla.Rotation(pitch=0.000000, yaw=-180, roll=0.000000))
        self.stopped_vehicle = self.world.spawn_actor(stopped_vehicle_bp,
self.stopped_spawn_location)
        braking_control_full = carla.VehicleControl(throttle=0.0, brake=1.0)
        self.stopped_vehicle.apply_control(braking_control_full)

        #
================================================================================
        # -- Spawn locations and destinations  ---------------------------------
-------
        #
================================================================================
        spawn_location = self.config['agent']['spawn_location']
        self.main_vehicle_spawn_location =
carla.Transform(carla.Location(x=spawn_location[0], y=spawn_location[1],
z=spawn_location[2]), carla.Rotation(pitch=0.000000, yaw=-180, roll=0.000000))
        final_location = self.config['agent']['destination']
        self.final_destination =
carla.Transform(carla.Location(x=final_location[0], y=final_location[1],
z=final_location[2])).location


        #
================================================================================
        # -- Spawning straight line vehicle/s  ---------------------------------
----------
        #
================================================================================
        for i in range(0, self.straight_car_count):
            vehicle_bp = blueprint_library.filter("vehicle.tesla.model3")[0]
            spawn_location = self.straight_paths[i]['spawn']
            straight_car = self.world.spawn_actor(vehicle_bp, spawn_location)
            straight_car.apply_control(braking_control_full)
            # straight_car.set_simulate_physics(False)
            self.straight_cars.append(straight_car)

        #
================================================================================
```

```
        # -- Spawning lane change agent  ------------------------------------
----
        #
==============================================================================
        vehicle_bp = blueprint_library.filter("vehicle.tesla.model3")[0]
        self.main_vehicle = self.world.spawn_actor(vehicle_bp,
self.main_vehicle_spawn_location)
        self.main_vehicle.apply_control(braking_control_full)

        #
==============================================================================
        # -- Spawning And attaching collision sensor  ---------------------------
----------
        #
==============================================================================
        collision_blueprint =
self.world.get_blueprint_library().find('sensor.other.collision')
        self.collision_sensor = self.world.spawn_actor(collision_blueprint,
carla.Transform(), attach_to=self.main_vehicle)
        weak_self = weakref.ref(self)
        self.collision_sensor.listen(lambda event:
WorldEnv._on_collision(weak_self, event))


        #
==============================================================================
        # -- Adding camera to regular vehicle for spectator to follow -----------
--------------------------------
        #
==============================================================================
        # Not used now
        camera_bp = self.world.get_blueprint_library().find('sensor.camera.rgb')
        camera_transform = carla.Transform(carla.Location(x=-20,z=10),
carla.Rotation(-20,0,0))
        self.camera = self.world.spawn_actor(camera_bp, camera_transform,
attach_to=self.main_vehicle)


        #
==============================================================================
        # -- initalizing agents -------------------------------------------
```

```
        #
================================================================================
        for i in range(0, self.straight_car_count):
            straigt_agent = BehaviorAgent(self.straight_cars[i],
behavior='normal')
            route_trace =
straigt_agent.set_destination(self.straight_paths[i]['destination'])
            self.straight_agents.append(straigt_agent)

        self.pass_agent = PassingAgent(self.main_vehicle, behavior='normal')
        route_trace2 = self.pass_agent.set_destination(self.final_destination)


    '''
    Step function that takes the action from the DQNAgent and applies it
    to our ego vehicle, as well as stepping through the other vehicles.
    '''
    def step(self, action):
        self.step_counter += 1
        done = False
        info = {}

        pass_control = carla.VehicleControl(throttle=0.0, brake=0.0)

        # Calculate distance from the passing car to the obstacle car
        dist_to_onstacle =
self.main_vehicle.get_transform().location.distance(self.stopped_spawn_location.l
ocation)
        if dist_to_onstacle <= 20 and self.not_passed:
             self.pass_agent.start_pass()
             self.not_passed = False

        if action == 0:
            self.zero_actions += 1
            pass_control = self.pass_agent.run_step(debug = PATH_ARROWS_ENABLED)
        if action == 1:
            self.one_actions += 1
            pass_control.throttle = 0.0
            pass_control.brake = 0.5

        self.main_vehicle.apply_control(pass_control)
```

```
        dist_to_dest =
self.main_vehicle.get_transform().location.distance(self.final_destination)



        for i in range(0, self.straight_car_count):
            straigt_dist_to_dest =
self.straight_cars[i].get_transform().location.distance(self.straight_paths[i]['d
estination'])
            straigt_control = self.straight_agents[i].run_step(debug =
PATH_ARROWS_ENABLED)
            straigt_control.manual_gear_shift = False
            straigt_control.throttle = self.throttle_values[i]
            straigt_control.brake = 0.0
            if straigt_dist_to_dest < 10:
                straigt_control.throttle = 0
                straigt_control.brake = 1.0
            self.straight_cars[i].apply_control(straigt_control)

        reward = -1000
        if dist_to_dest < 5:
            self.eps_end_time = datetime.datetime.now()
            reward = 1000000
            print('reward is: ', reward)
            print('Number of steps was: ', self.step_counter)
            print('self.one_actions: ', self.one_actions)
            print('self.zero_actions: ', self.zero_actions)
            self.total_passing_time_steps += self.step_counter
            done = True
        if self.collision_history > 0:
            reward = -1000000
            print('self.one_actions: ', self.one_actions)
            print('self.zero_actions: ', self.zero_actions)
            done = True
            self.did_crash = True


        if self.step_counter > STEPS_LIMIT:
            print('self.one_actions: ', self.one_actions)
            print('self.zero_actions: ', self.zero_actions)
            done = True
```

```
        # state is the main car location X,Y and the distance to the oncoming car
        main_location = self.main_vehicle.get_transform().location
        # distance_to_straight =
self.main_vehicle.get_transform().location.distance(self.straight_vehicle.get_tra
nsform().location)
        state, cars_behind = self.get_current_state(main_location)
        self.visualize_debug(cars_behind)
        return state, reward, done, info


    '''
    Function to get the current state of the ego vehicle according to our set
    state-space.
    '''
    def get_current_state(self, main_location):
        cars_behind = self.get_effecting_cars(back_proximity =
self.config['back_proximity'], front_proximity = self.config['front_proximity'])
        if len(cars_behind) > self.max_car_count:
            print('-----------------------------------------------')
            print('len of cars is: ', len(cars_behind))
            print('-----------------------------------------------')
        relative_x = 0
        relative_y = 0
        cars_state_space = []

        for i in range(0, self.max_car_count):
            cars_state_space.append([0, 0, 0])

        for i in range(0, len(cars_behind)):
            speed_straight = 0
            t = cars_behind[i].get_transform()
            v = cars_behind[i].get_velocity()
            c = cars_behind[i].get_control()
            speed_straight = (3.6 * math.sqrt(v.x**2 + v.y**2 + v.z**2))

            other_loation = cars_behind[i].get_transform().location
            relative_x = round(main_location.x, 4) - round(other_loation.x, 4)
            relative_y = round(main_location.y, 4) - round(other_loation.y, 4)
            cars_state_space[i] = [round(relative_x,4), round(relative_y,4),
speed_straight]

        return cars_state_space, cars_behind
```

```
    '''
    Function to visualize the vectors as well
    as the boxes around the cars.
    '''
    def visualize_debug(self, cars_behind):
        # ....
        debug = self.world.debug
        world_snapshot = self.world.get_snapshot()
        if BOUNDING_BOXES_ENABLED:
            for vehicle in self.world.get_actors().filter('vehicle.*'):
                transform = vehicle.get_transform()
                bounding_box = vehicle.bounding_box
                bounding_box.location += transform.location
                self.world.debug.draw_box(bounding_box,
transform.rotation,thickness = 0.05, life_time=0.1)
            # for actor_snapshot in world_snapshot:
            #     actual_actor = self.world.get_actor(actor_snapshot.id)
            #     # print('actual_actor.type_id: ', actual_actor.type_id)
            #     if 'vehicle' in actual_actor.type_id:
            #         print('drawing')
            #
debug.draw_box(carla.BoundingBox(actor_snapshot.get_transform().location,carla.Ve
ctor3D(0.5,0.5,2)),actor_snapshot.get_transform().rotation, 0.05,
carla.Color(255,0,0,0), 0.1)
            # # ...
        if TRACKING_LINES_ENABLED:
            for vehicle in cars_behind:
                debug.draw_line(self.main_vehicle.get_transform().location,
vehicle.get_transform().location, 0.1, carla.Color(255,0,0,0), 0.1)


    '''
    Function to handle when a collision happens.
    '''
    @staticmethod
    def _on_collision(weak_self, event):
        """On collision method"""
        self = weak_self()
        if not self:
            return
```

```
        if self.stage == 'fit':
            self.stats['fit']['collisions'] += 1
        else:
            self.stats['test']['collisions'] += 1

        self.crash_counter += 1
        self.collision_history = 1
        time.sleep(1)

    def check_for_obstacle(self):
        # Calculate distance from the passing car to the obstacle car
        dist_to_onstacle =
self.main_vehicle.get_transform().location.distance(self.stopped_spawn_location.l
ocation)
        # print('dist_to_onstacle: is ', dist_to_onstacle)
        if dist_to_onstacle > 30:
             return False
        return True

    #
================================================================================
    # -- Function to get the simulation case rolling -------------------------
---
    #
================================================================================
    def run_case_setup(self):
        '''
        Basic idea here is to get the case rolling and the cars moving
        at the beginning of each Simulation run, because we dont want
        the cars to start off stopped.

        Another way to think of it is [IMPORTANT]:
        Our agent is running normally always, the stopped car that
        is an obstacle to us, sends a V2X signal that it is stopped and
        in emegency mode. Our agent can check if that car is on the same road
        and in the same lane as us. ONCE we get within let's say 50 meters
        of that car, our model can then start running
        '''
        obstacle_detected = self.check_for_obstacle()
        while not obstacle_detected:
            # run the cars normally straigt
            pass_control = self.pass_agent.run_step(debug = PATH_ARROWS_ENABLED)
```

```
            self.main_vehicle.apply_control(pass_control)
            for i in range(0, self.straight_car_count):
                straigt_dist_to_dest =
self.straight_cars[i].get_transform().location.distance(self.straight_paths[i]['d
estination'])
                straigt_control = self.straight_agents[i].run_step(debug =
PATH_ARROWS_ENABLED)
                straigt_control.manual_gear_shift = False
                straigt_control.throttle = self.throttle_values[i]
                straigt_control.brake = 0.0
                if straigt_dist_to_dest < 10:
                    straigt_control.throttle = 0
                    straigt_control.brake = 1.0
                self.straight_cars[i].apply_control(straigt_control)

            obstacle_detected = self.check_for_obstacle()

        # state is the main car location X,Y and the distance to the oncoming car
        main_location = self.main_vehicle.get_transform().location
        # distance_to_straight =
self.main_vehicle.get_transform().location.distance(self.straight_vehicle.get_tra
nsform().location)
        state, cars_behind = self.get_current_state(main_location)
        self.one_actions = 0
        self.zero_actions = 0
        return state


    #
================================================================================
    # -- Set the env stage to either testing or fitting ------------------------
---
    #
================================================================================
    def set_environment_stage(self, stage):
        self.stage = stage

    def print_stats(self, num_episodes):
        completed_episodes = num_episodes - (self.crash_counter) - 1
        print('Fitting Stats')
        print('Number of collisions: ', self.stats['fit']['collisions'])
        print('Testing Stats')
```

```
        print('Number of collisions: ', self.stats['test']['collisions'])
        print('Total number of steps: ', self.total_passing_time_steps)
        print('completed_episodes: ', completed_episodes)
        print('Average # steps / eps: ', self.total_passing_time_steps /
completed_episodes)
        print('self.eps_times_total: ', self.eps_times_total)
        print('average time per episode: ', self.eps_times_total /
completed_episodes)


    # ==================================,
==========================================
    # -- Function to reset the env variables ----------------------------------
---
    #
================================================================================
    def reset(self):
        self.straight_car_count = 0
        self.straight_cars = []
        self.straight_agents = []
        self.straight_paths = []
        self.throttle_values = []

        if self.collision_sensor and self.collision_sensor.is_alive:
            self.collision_sensor.destroy()

        actor_list = self.world.get_actors()
        for vehicle in actor_list.filter('vehicle.*.*'):
            vehicle.destroy()
        for sensor in actor_list.filter('sensor.camera.*'):
            if sensor and sensor.is_alive == True:
                sensor.destroy()

        self.collision_history = 0
        self.step_counter = 0
        self.not_passed = True
        self.completed_counter += 1
        print('self.completed_counter: ', self.completed_counter)
        print('self.crash_counter: ', self.crash_counter)
        if not self.did_crash:
            delta = self.eps_end_time - self.eps_start_time
            print('delta in ms: ', int(delta.total_seconds() * 1000))
```

```
            self.eps_times_total += int(delta.total_seconds() * 1000)
        self.did_crash = False
        self.eps_start_time = datetime.datetime.now()
        self.init_env()
        return self.run_case_setup()


from keras.models import load_model
EPOCHS = 10000
EPISODES = 1
model_name_root = 'saved_models/v2x_str8_4_64_2_5M_Fin1'

from datetime import date


# ================================================================================
# -- Function to setup the path for the model saving process -------------------
# ================================================================================
def get_model_path_name():
    version_name = model_name_root
    version_file_name = 'model_version_number.txt'
    with open(version_file_name) as f:
        lines = f.read()
        version_number = int(lines.strip())
        today = date.today()
        version_name += '-V' + str(version_number) + '-' + str(today)
        print("version_name:", version_name)
    with open(version_file_name, "w") as myfile:
        myfile.write(str(version_number+1))
    return version_name

# ================================================================================
# -- Loading the config JSON file ----------------------------------------------
# ================================================================================
def get_config_json(file_path):
    f = open(file_path)
    data = json.load(f)
    return data

# ================================================================================
# -- Main Game loop ------------------------------------------
# ================================================================================
```

```
def game_loop(args):
    try:
        #
================================================================================
        # -- Setting different settings  ------------------------------------
----
        #
================================================================================
        MODEL_PATH = get_model_path_name()
        config = get_config_json("config.json")

        if args.seed:
            random.seed(args.seed)

        client = carla.Client(args.host, 3000)
        client.set_timeout(10.0)
        world = client.get_world()
        map = world.get_map()
        if not config['map'] in map.name:
            world = client.load_world(config['map'])
        old_settings = world.get_settings()
        settings = world.get_settings()

        settings.fixed_delta_seconds = config['train_delta']
        world.apply_settings(settings)
        original_weather = world.get_weather()
        print('original_weather is: ', original_weather)
        selected_weather = config['selected_weather']
        weather = None
        weather_from_config = None

        if selected_weather == "clear":
            weather_from_config = config['clear_weather']
        elif selected_weather == "fog_rain":
            weather_from_config = config['fog_rain_weather']
        elif selected_weather == "night":
            weather_from_config = config['night_weather']

        weather = carla.WeatherParameters(sun_altitude_angle=
weather_from_config['sun_altitude_angle'],
                                          cloudiness=
weather_from_config['cloudiness'],
```

```
                                            precipitation =
weather_from_config['precipitation'],
                                            wetness=
weather_from_config['wetness'],
                                            fog_density=
weather_from_config['fog_density'],
                                            fog_distance=
weather_from_config['fog_distance'],
                                            fog_falloff=
weather_from_config['fog_falloff'])
        world.set_weather(weather)


        env = WorldEnv(world, old_settings, args, config)
        states = env.observation_space.shape
        actions = env.action_space.n


        project_mode = config['project_mode']
        model = None
        if project_mode == 'model_load':
            # In the case we want to load an already saved model
            settings.fixed_delta_seconds = config['test_delta']
            world.apply_settings(settings)
            model_path = 'saved_models/' + config['model_load_name']
            model = load_model(model_path)
            dqn = build_agent(model, actions)
            dqn.compile(Adam(lr=1e-3), metrics=['mae'])
            env.set_environment_stage('test')
            settings.fixed_delta_seconds = config['test_delta']
            world.apply_settings(settings)
            scores = dqn.test(env, nb_episodes= config['test_episodes'],
visualize=False)
            print(np.mean(scores.history['episode_reward']))
            env.print_stats(config['test_episodes'])
            # print('Total number of steps: ', total_passing_time_steps)
            # print('Average # steps / eps: ', total_passing_time_steps /
config['test_episodes'])

        else:
            # In we want to create and train a new model
            model = build_model(states, actions)
```

```
            model.summary()
            # model = load_model(MODEL_PATH)
            dqn = build_agent(model, actions)
            dqn.compile(Adam(lr=1e-3), metrics=['mae'])
            env.set_environment_stage('fit')
            dqn.fit(env, nb_steps=config['train_steps'], visualize=False,
verbose=1)
            settings.fixed_delta_seconds = config['test_delta']
            # settings.no_rendering_mode = False
            world.apply_settings(settings)
            dqn.model.save(MODEL_PATH)
            env.set_environment_stage('test')
            scores = dqn.test(env, nb_episodes= config['test_episodes'],
visualize=False)
            print(np.mean(scores.history['episode_reward']))
            env.print_stats()


    finally:
        world.apply_settings(old_settings)
        world.set_weather(original_weather)
        actor_list = world.get_actors()
        for vehicle in actor_list.filter('vehicle.*.*'):
            vehicle.destroy()
        for vehicle in actor_list.filter('sensor.*.*'):
            vehicle.destroy()


from rl.agents import DQNAgent
from rl.policy import BoltzmannQPolicy
from rl.policy import EpsGreedyQPolicy
from rl.policy import LinearAnnealedPolicy
from rl.memory import SequentialMemory


def build_model(states, actions):
    print('states are: ', states)
    model = Sequential()
    model.add(Dense(64, activation='relu', input_shape=states))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(64, activation='relu'))
```

```
    model.add(Flatten())
    model.add(Dense(actions, activation='linear'))
    return model

def build_agent(model, actions):
    policy = LinearAnnealedPolicy(EpsGreedyQPolicy(), attr='eps', value_max=1.,
value_min=.1, value_test=.05, nb_steps=600000)
    # policy = BoltzmannQPolicy()
    memory = SequentialMemory(limit=1000000, window_length=1)
    dqn = DQNAgent(model=model, memory=memory, policy=policy, nb_actions=actions,
nb_steps_warmup=100, target_model_update=1e-2)
    return dqn

# ================================================================================
# -- Run Simulation Function -----------------------------------------
# ================================================================================

def run_simulation():
    """Run Simulation Method"""

    argparser = argparse.ArgumentParser(
        description='CARLA Automatic Control Client')
    argparser.add_argument(
        '-v', '--verbose',
        action='store_true',
        dest='debug',
        help='Print debug information')
    argparser.add_argument(
        '--host',
        metavar='H',
        default='127.0.0.1',
        help='IP of the host server (default: 127.0.0.1)')
    argparser.add_argument(
        '-p', '--port',
        metavar='P',
        default=2000,
        type=int,
        help='TCP port to listen to (default: 2000)')
    argparser.add_argument(
        '--res',
        metavar='WIDTHxHEIGHT',
        default='1280x720',
```

```
        help='Window resolution (default: 1280x720)')
    argparser.add_argument(
        '--sync',
        action='store_true',
        help='Synchronous mode execution')
    argparser.add_argument(
        '--filter',
        metavar='PATTERN',
        default='vehicle.*',
        help='Actor filter (default: "vehicle.*")')
    argparser.add_argument(
        '-l', '--loop',
        action='store_true',
        dest='loop',
        help='Sets a new random destination upon reaching the previous one
(default: False)')
    argparser.add_argument(
        "-a", "--agent", type=str,
        choices=["Behavior", "Basic"],
        help="select which agent to run",
        default="Behavior")
    argparser.add_argument(
        '-b', '--behavior', type=str,
        choices=["cautious", "normal", "aggressive"],
        help='Choose one of the possible agent behaviors (default: normal) ',
        default='normal')
    argparser.add_argument(
        '-s', '--seed',
        help='Set seed for repeating executions (default: None)',
        default=None,
        type=int)

    args = argparser.parse_args()

    args.width, args.height = [int(x) for x in args.res.split('x')]

    log_level = logging.DEBUG if args.debug else logging.INFO
    logging.basicConfig(format='%(levelname)s: %(message)s', level=log_level)

    logging.info('listening to server %s:%s', args.host, args.port)
    print(__doc__)
    try:
```

```
        game_loop(args)

    except KeyboardInterrupt:
        print('\nCancelled by user. Bye!')


if __name__ == '__main__':
    run_simulation()
```