REINFORCEMENT LEARNING FOR SELF DRIVING CARS IN CARLA SIMULATOR

A thesis submitted in partial fulfilment of the requirements for the award of the degree of

B.Tech.

in

Instrumentation and Control Engineering

By

Akkannagari Rakesh(110119012)

Grandhi Sri Sreya (110119043)

K.Deekshitha Hima Priya(110119048)

Kokilavayamatham Spoorthi(110119054)

Thummala Greeshma(110119120)

Completed at the

National Institute of Technology, Tiruchirappalli, India

INSTRUMENTATION AND CONTROL ENGINEERING

NATIONAL INSTITUTE OF TECHNOLOGY

TIRUCHIRAPALLI-620015

MAY 2023

BONAFIDE CERTIFICATE

This is to certify that the project titled **REINFORCEMENT LEARNING FOR SELF DRIVING CARS IN CARLA SIMULATOR** is a bonafide record of the work done by

Akkannagari Rakesh(110119012)
Grandhi Sri Sreya (110119043)
K.Deekshitha Hima Priya(110119048)
Kokilavayamatham Spoorthi(110119054)
Thummala Greeshma(110119120)

in partial fulfilment of the requirements for the award of the degree of **Bachelor of Technology in Instrumentation and Control Engineering of the NATIONAL INSTITUTE OF TECHNOLOGY, TIRUCHIRAPPALLI**, during the year **2022-2023.**

Dr. S.Narayanan	**Dr. K. Dhanalakshmi**
Guide	Head of Department

Project Viva-voice held on ___________________________

Internal Examiner	**External Examiner**

ABSTRACT

The aim of this project is to utilize reinforcement learning (RL) approach for self-driving cars in the Carla simulator. The RL algorithm is trained to optimize the decision-making process of the autonomous vehicle in various scenarios, such as lane following, intersection crossing, and pedestrian avoidance. The proposed RL agent is designed using a deep neural network which is DQN that takes the vehicle's sensor data as input and outputs the corresponding control commands. The training process is conducted using a reward function that encourages safe driving behaviours and penalizes any violations of traffic rules.

The performance of the RL agent is evaluated using various metrics such as distance travelled, time taken, and the number of collisions. The experimental results show that the proposed RL approach outperforms traditional rule-based approaches in terms of driving performance and generalization to new scenarios.

Overall, this project demonstrates the effectiveness of RL for self-driving cars in a simulated environment, contributing to the development of intelligent transportation systems. The results show that RL is a promising approach for autonomous driving, as it can learn from experience and adapt to new situations, leading to safer and more efficient driving.

Keywords: Reinforcement Learning; CARLA Simulator; Python; DQN; self-driving car

ACKNOWLEDGEMENTS

We would like to express our sincere gratitude to all those who have contributed to the successful completion of our final year project. First and foremost, we would like to thank our guide Dr. S Narayanan for his guidance and support. We are also grateful to the Project Evaluation Committee consisting of Dr. A. Ramakalyan, Dr. D. Ezhilarasi, and Dr. C. Geetha for providing us with the necessary constructive feedback which is invaluable throughout the entire project. Finally, we wish to thank the Head of Department, Dr. K. Dhanalakshmi for her support.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 – INTRODUCTION

1.1 Introduction

Artificial Intelligence has become extremely popular in today’s world. It is the reproduction of regular knowledge in machines that are customized to impersonate the activities of people. These machines can learn and execute human-like tasks. As these technologies continue to grow, they will significantly affect our quality of life. One of the technologies that have emerged with this growth is autonomous driving. An autonomous car is a vehicle which senses its environment and operates without any human involvement.

The main aim of this thesis is to set a base on the topic of autonomous driving with the help of the open-source software CARLA in which we will be able to set up an environment and simulate the learning of an intelligent car.

Terms and concepts

The key concepts of this work are defined in the next section as the reader must be familiarized with the following concepts and understand them correctly as well as the topics related to autonomous driving that can generate more confusion.

Machine learning

Machine learning is a method of data analysis that automates analytical model building. It is a branch of artificial intelligence based on the idea that systems can learn from data, identify patterns, and make decisions with minimal human intervention.

Reinforcement learning

Reinforcement learning is a machine learning training method based on rewarding desired behaviors and/or punishing undesired ones. In general, a reinforcement learning agent can perceive and interpret its environment, take actions and learn through trial and error.

DQN

The Deep Q-Network algorithm was developed by DeepMind in 2015. It could solve a wide range of Atari games (some to superhuman level) by combining reinforcement learning and deep neural networks at scale. The algorithm was developed by enhancing a classic RL algorithm called Q-Learning with deep neural networks and a technique called experience replay.

ANNs

Neural networks, also known as artificial neural networks (ANNs) are a subset of machine learning and are at the heart of deep learning algorithms. Their name and structure are inspired by the human brain, mimicking the way that biological neurons signal to one another.

Autonomous Car

A self-driving car, also known as an autonomous vehicle (AV) or driverless car, is a car incorporating vehicular automation, that is, a ground vehicle that can sense its environment and move safely with little or no human input.

CARLA

CARLA is the open-source simulator for autonomous driving research. It has been developed from the ground up to support development, training, and validation of autonomous driving systems. The simulation platform supports flexible specification of sensor suites, environmental conditions, full control of all static and dynamic actors, map generation and much more.

Problem to be resolved.

The project consists of using the CARLA Simulator to obtain autonomous driving. It has been divided into two sections to facilitate and ensure the correct completion of the work. The first part consists of setting up the environment that we will be using for the execution of the learning process. This includes the prerequisites, software installation, library dependencies and any other requisite. The second part of the project is the proper learning of the car. To accomplish this a circuit will be defined with an origin, a destination, and a series of waypoints where the car must pass through. Once the waypoint system has been developed, we can start with the learning process of the car.

Thanks to reinforcement learning, the car will perceive and interpret its environment, take actions, and learn through trial and error.

CHAPTER 2 – LITERATURE REVIEW

Vehicle control can correct errors coming from motion planning and decision-making tasks. In recent years, great efforts have been made to improve vehicle control for AVs, particularly using DL. Sharma et al. [1] collected data include images labelled by vehicle speed, steering angle, and throttle using the TORCS simulator (TORCS). This data is fed into a CNN to train their model to estimate vehicle speed and steering angle at the same time. However, their approach is limited only by CNN. Thus, TORCS simulator is not realistic to validate the performance of their model.

Mujica et al. [2] presented a model to steer AVs through several scenarios with lane changes and traffic lights. They trained their proposal with CNN and RNN to extract features from the data collected by the Udacity simulator. Their results are somewhat better compared to previous.

Xing et al. [3] suggested method to effectuate lane change on a freeway. The model uses both CNN and RNN to train their model. Their results are not tested in a simulator to validate their performance.

In [7], authors presented RL method to generate the steering angle of an AV to avoid sharp turns in the road. Their model showed significant stability during learning, but it excluded other road users in the environment.

The previous approaches, one of them being trained with DL techniques using data collected from simulators such as TORCS or Udacity, do not provide realistic scenarios like in the real world. Moreover, training a model with single data is inefficient because each environment has its own characteristics. For these reasons, we propose an RL model based on DQN method that allows AV to learn from the environment to reach its destination and also avoid collisions with other vehicles and pedestrians. Moreover, we test and validate our approach in the latest version of the CARLA simulator, which is more realistic and contains buildings, pedestrians, etc.

CHAPTER 3 – CARLA SETUP

Our first order of business is to get Carla. It's simple. Head over to CARLA official website and then scroll down a bit to the latest release.

From here, at least at the time of my writing this, the latest version to have windows support is CARLA 0.9.8. We are using Windows machine for this project; we need to grab CARLA 0.9.8.zip.

Later, you might be able to get more up-to-date versions. Feel free to do that, just know that things will change over time, including the code syntax. It might be best for you to learn on the same versions that I am using, then later upgrade. Do whatever you want though!

Once you have the compressed info, extract it, and you will have what you need to run. In the main directory of what you just extracted; you will have a CarlaUE4.exe on Windows.

To run this on Windows, just double click the .exe.

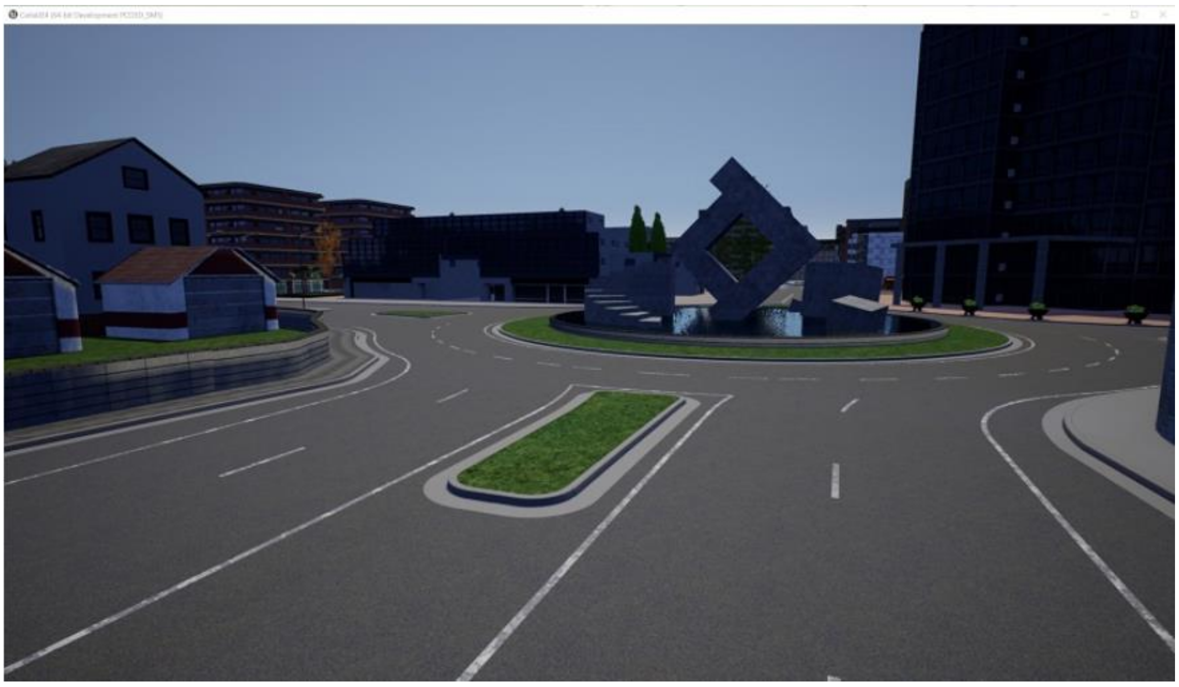


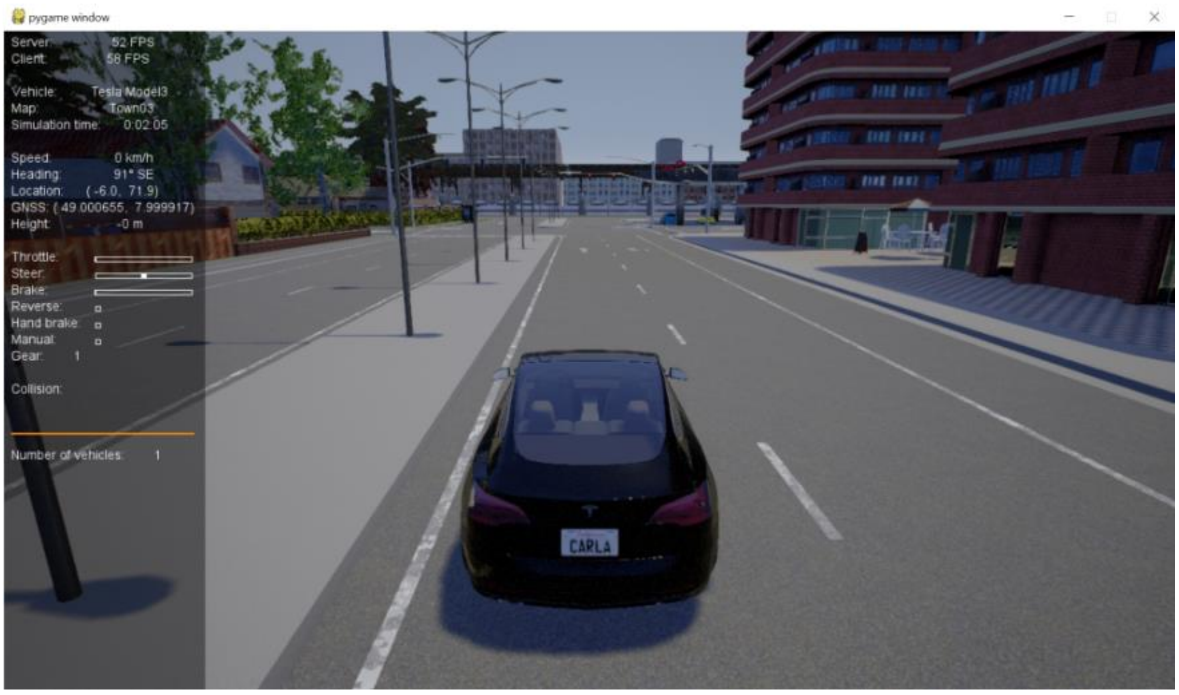
Fig 3.1 The Carla Simulator Environment

This will run the server. You should eventually see the map, which you can navigate with the WASD keys and your mouse. Of course, nothing is yet here. This is just our environment. The GET STARTED section shows some examples of what we can do, which make use of the examples that we can find in the PythonAPI directory.

Go ahead and navigate from the main Carla directory to the examples: PythonAPI/examples. Here, you will have some files like manual_control.py, dynamic_weather.py, and spawn_npc.py to name a few. We can check some of these out. Open terminal/cmd and run one like manual_control.py:

py -3.7 manual_control.py

This will create another window. What you're seeing is an example of what we can do with the Python API. Here, you can control the car with WASD keys, and Q will change into and out of reverse.



```
Server:          52 FPS
Client:          58 FPS

Vehicle:     Tesla Model3
Map:             Town03
Simulation time:    0:02:05

Speed:           0 km/h
Heading:          91° SE
Location:     ( -6.0,  71.9)
GNSS: ( 49.000655,  7.999917)
Height:           -0 m

Throttle:
Steer:
Brake:
Reverse:
Hand brake:
Manual:
Gear:      1

Collision:

Number of vehicles:     1
```

FIG 3.2 Car Environment In CARLA

There are python API's that allow users to control every aspect of the simulation. Our view is to have everything available from Python, loading maps, adding sensors, adding static obstacles, controlling vehicles and pedestrians, and what not.

Before moving forward here are the 3 important concepts:

- **Actor:** Actor is anything that plays a role in the simulation and can be moved around, examples of actors are vehicles, pedestrians, and sensors.

- **Blueprint:** Before spawning an actor, you need to specify its attributes, and that's what blueprints are for. We provide a blueprint library with the definitions of all the actors available for you to choose.
- **World:** The world represents the currently loaded map and contains the functions for converting a blueprint into a living actor moving around in the simulation.

__**Sensors Used**__

Cameras are the first main type of sensors that we can use in vehicles. They take a shot of the world from their point of view a can retrieve data in every simulation step.

- RGB camera provides clear vision of the surroundings. Looks like a normal photo ofthe scene.

The next group of sensors are detectors, and they retrieve data when the object they are attached to registers a specific event.

- Collision detector retrieves collisions between its parent and other actors.

CHAPTER 4 – REINFORCEMENT LEARNING (DQN)

Reinforcement Learning is all about learning the optimal behavior in an environment to obtain maximum reward with an agent.

4.1 Introduction

In RL we have two main actors: the agent and the environment, in this case CARLA, in which an agent is trained. To make this training possible we have three main interactions between them. These are the following:

- State: it describes the status of the agent. This could be the current position of the car, the speed or even the orientation of the vehicle.
- Action: it describes what can an agent do. Moving forward, pressing the brake, or rotating the car are different actions that can be performed.
- Reward: it is feedback on the action previously taken on a state. This could be a positive or negative reward.

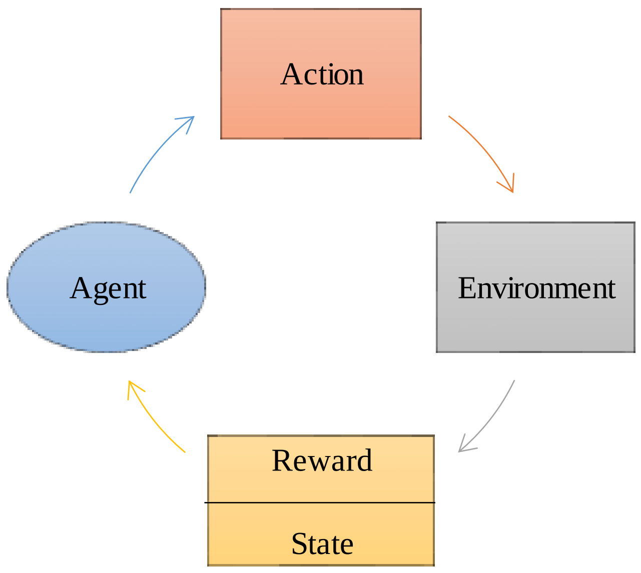


Fig 4.1 Reinforcement Learning

Maximizing the reward obtained by performing a series of actions is the main objective of the agent as we previously mentioned. Therefore, a proper reward system is very important as choosing one action or the other will be affected by it.

During the training process we will see two main stages: exploration and exploitation. The exploration stage has a specific purpose, to visit states that you have not yet visited or to take actions you have not yet taken. When you start the first episodes you want the agent to explore and do some trial and error to find the actions that will return a positive reward. On the other hand, the exploitation stage is when you decide to use the knowledge and memory from the previous episode to maximize the rewards. This stage usually comes at the end of the training session or when you have a trained agent.

The agent will require a policy to learn. The policy defines how the agent will behave at a given time and map from perceived states of the environment to actions to be taken when in those states. To store all this data the policy uses a huge table with all the information saved for each state. At the beginning, if the agent has not been trained before this table will be empty, in other words, it will have no experience. This in when the exploration stage becomes useful. After the agent has performed a series of steps, it will start leaning by adjusting its predicted rewards for specific pairs of state-actions towards the received rewards. These predicted values are known as Q-values and are used as feedback on how good a state is.

The parameter that determines the probability of the agent performing a random action is the epsilon (∈). It is a real value that exists in [0, 1]. This variable introduces randomness into our training algorithm forcing the agent to try different actions. A high epsilon, for example, ∈ = 1 will result in all actions are random which means we are in the exploration stage. On the other hand, a lower epsilon such as ∈ = 0 will result in no random actions which means we are in the exploitation stage. Implementing an epsilon decay formula is very important during the training process. This is because we want to travel from the exploration stage to the exploitation to maximize learning. The formula

$$\in_n = \in_{n-1} \times \in_{decay} \qquad (1)$$

we will be using will be the following:

We will need to set an $\in_{decay}$ that, when multiplied by the current epsilon, will give us the next epsilon value. To calculate the epsilon value in a certain number of episodes we will use the following formula:

$$\in = \in_{initial} \times \in_{decay}^{n} \qquad (2)$$

Where n is the number of episodes, $\in_{decay}$ the epsilon decay per episode and $\in_{initial}$ the epsilon used at the beginning.

Another important parameter is the discount factor (γ). It is a real value that exists in [0, 1] and determines how the agent should care about the rewards achieved in the past, present and future. If γ=0 the agent will focus on actions that produce an immediate reward whereas if γ=1 the agent cares on actions that will produce a high future reward due to a sequence of actions.

4.2. DQN Agent

Deep Q-Network is a reinforcement learning algorithm that combines Q-Learning with deep neural networks to let RL work for complex, high-dimensional environments, like video games, or robotics.

DQN uses the Bellman's equation to solve the problem of being unable to see future rewards. Otherwise, only immediate rewards would be shown. The formulation of the equations is as follows:

$$Q^{new}(s_t, a_t) = (1-\in) \times \underbrace{Q(s_t, a_t)}_{old\ value} + \in \times \underbrace{(r_t + \gamma * max(Q^{new}(s_{t+1}, a))}_{learned\ value} \tag{3}$$

Where:

- Q is the accumulated reward value
- t is the current state
- s is a specific state
- a is a specific action
- $\in$ is the epsilon, also known as learning rate.
- r is the reward associated to a specific state
- γ is the discount factor.

The agent can use this equation to link up all the information learned resulting in a batch of memory that can be used for improving the efficiency of its learning.

4.3 ANN

For this project we used a CNN.

It consists of:

- 3 convolutional layers with 64 neurons each + ReLU activation function which is used for filtering information.
- 3 pooling layers performed with Average pooling. They are used to reduce the dimensions of the feature maps, in this case 3x3.
- 3 dropouts layers following the convolutional networks to avoid overfitting.
- 1 flatten layer to convert the vector to 1D.
- 1 dense layer with 64 neurons.
- 1 dense layer with 3 neurons (3 actions) with linear activation function. It is used to determine the output of neural network like yes or no.

CHAPTER 5 – IMPLEMENTATION

Implementing DQN (Deep Q-Network) on Carla involves several steps, including setting up the environment, defining the neural network architecture, and implementing the DQN algorithm. Here is a detailed implementation description:

1. Setup Carla Environment: First, you need to install Carla and set up the environment for training the DQN model. This involves downloading and installing Carla, creating a Python virtual environment, and installing the required packages, such as TensorFlow and Keras.

2. Define the Neural Network Architecture: Next, you need to define the neural network architecture for the DQN model. The neural network should take as input the state of the environment (e.g., camera images, lidar data) and output the Q-values for each possible action. The architecture can be customized based on the specific requirements of the task, such as the number of hidden layers and nodes.

3. Implement the DQN Algorithm: The next step is to implement the DQN algorithm itself. This involves initializing the replay buffer, which stores past experiences, and the Q-network, which predicts the Q-values. The DQN algorithm then iteratively performs the following steps:

- Select an action using the epsilon-greedy policy (i.e., with a probability of epsilon, select a random action, otherwise select the action with the highest Q-value)
- Execute the selected action in the environment and observe the next state and reward.
- Store the experience in the replay buffer.
- Sample a batch of experiences from the replay buffer and update the Q-network using the loss function.

4. Train the DQN Model: After implementing the DQN algorithm, you can start training the DQN model. This involves repeatedly running the DQN algorithm in the Carla environment, collecting experiences and updating the Q-network, until the model converges.

The idea here is that your environment will have a step method, which returns observation, reward, done, extra_info, as well as a reset method that will restart the environment based on done flag.

Next, we're going to create environment's class, which we'll call CarEnv. It will be convenient to have some constants at the top of our script for the model, agent, and environment, so the first thing I'll do is create our first constant, which will be SHOW PREVIEW.

STEER_AMT is how much we want to apply to steering. Now, it's a full turn. Later we might find it's better to steer a little less forcefully, maybe doing something cumulative instead... etc. For now, full steer!

The collision_hist is going to be used because the collision sensor reports a history of incidents. Basically, if there's anything in this list, we're going to say we've crashed.

The algorithm:

```
1. Initialize replay memory capacity.
2. Initialize the policy network with random weights.
3. Clone the policy network and call it the target network.
4. For each episode:
    1. Initialize the starting state.
    2. For each time step:
        i. Select an action.
            1. Via exploration or exploitation
        ii. Execute selected action in an emulator.
        iii. Observe reward and next state.
        iv. Store experience in replay memory.
        v. Sample random batch from replay memory.
        vi. Preprocess states from batch.
```

vii. Pass batch of preprocessed states to policy network.

viii. Calculate loss between output Q-values and target Q-values.

    1. Requires a pass to the target network for the next state.

ix. Gradient descent updates weights in the policy network to minimize loss.

    1. After x time steps, weights in the target network are updated to the weights in the policy network.

CHAPTER 6 – MODEL RESULTS AND DISCUSSIONS

The main important parameters that are needed to get tuned or the whole process of Deep Q learning depends on Q values that depends on the rewards that agent is getting for a particular action. In our project we trained the model 3 times with varying reward function and ANN used.

6.1 Training 1:

The first training is done in "Town1" where the rewards and actions are described below. This test is the first approach with CARLA and all its environment setup. Obviously, good results are not going to be expected in the iteration.

The goal primarily here is to avoid collisions. The agent will be taking actions based on Q values associated with each action.

Training conditions:

Total Training Time	2 hrs
No.of Episodes	100
Actions	Action 0: Turn left Action 1: Accelerate the car Action 2: Turn right
Rewards	Reward 1: Collision: -200 points Reward 2: Velocity > 50 km/h: +1 point Reward 3: Velocity < 50 km/h: -1 point
Sensors Used	RGB camera Collision detection sensor
Model Used	Xception

Table 6.1 Training conditions for Xception Model

Results:

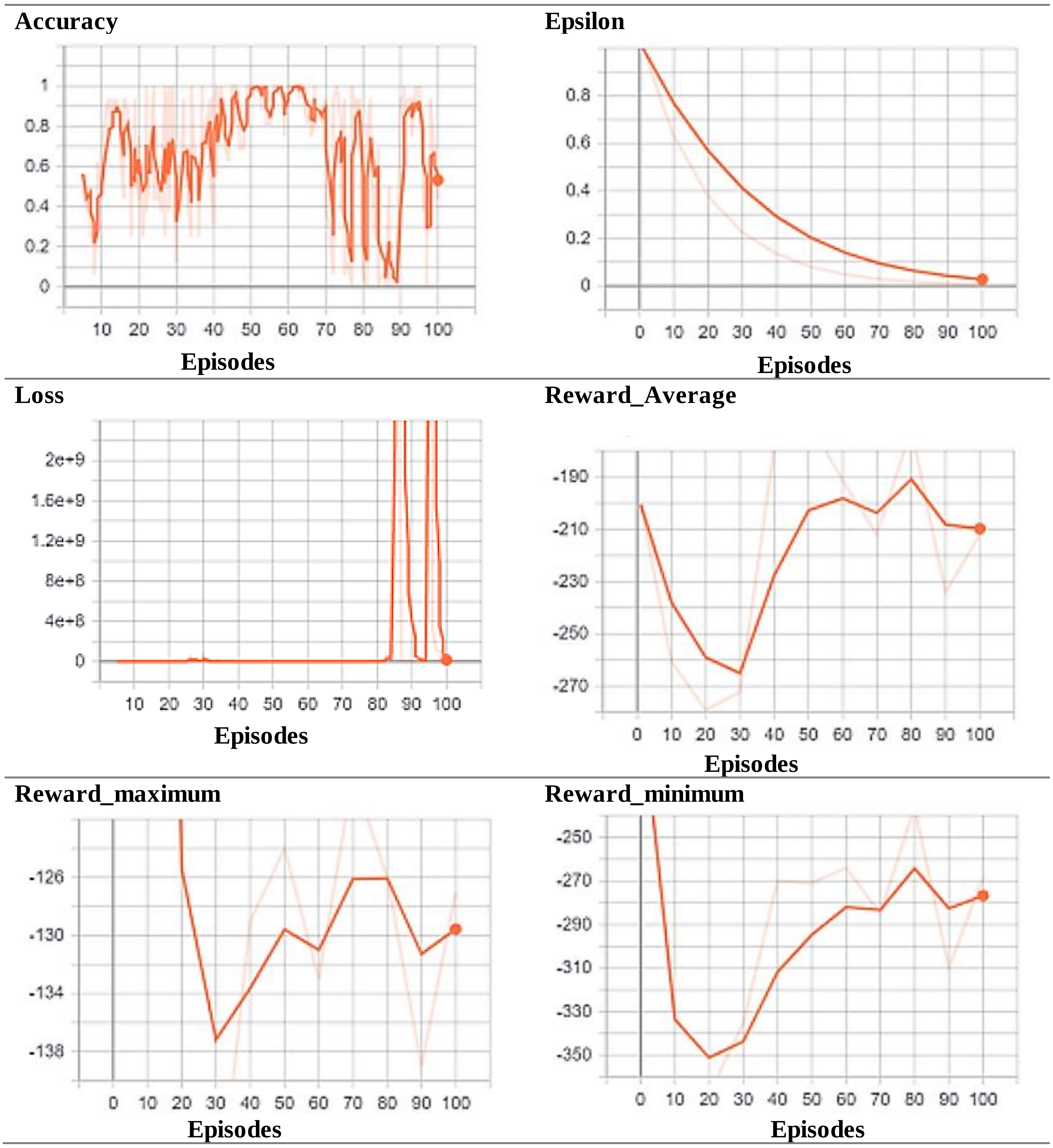


Fig. 6.1 Results of Training-1

The first thing we discovered was that both loss and q values were basically exploding. This was seemingly, and now obviously, due to the monumental size of the crash penalty in comparison to all else. Also possibly the bounds being out of range. For example, we might have had more success by doing something more like:

- +0.005 (1/200) for each frame driving > 50KMH
- -0.005 (-1/200) for each frame driving < 50KMH
- -1 (-200/200) for a collision and episode is over

But instead we just went with:

- +1 for each frame driving > 50KMH
- -1 for each frame driving < 50KMH
- -10 for a collision and episode is over

And also we were finding that our accuracy for the model was quite high (like 80-95%), so this tells me that we were almost certainly just overfitting every time.

6.2 Training 2:

To avoid above mentioned problems we went with other model the simple CNN and with the new reward function.

Training conditions:

Total Training Time	20 hrs
No.of Episodes	100,000
Actions	Action 0: Turn left Action 1: Accelerate the car Action 2: Turn right
Rewards	Reward 1: Collision: -10 points Reward 2: Velocity > 50 km/h: +1 point Reward 3: Velocity < 50 km/h: -1 point
Sensors Used	RGB camera Collision detection sensor
Model	CNN

Table 6.2. CNN Model Training Conditions

Results:

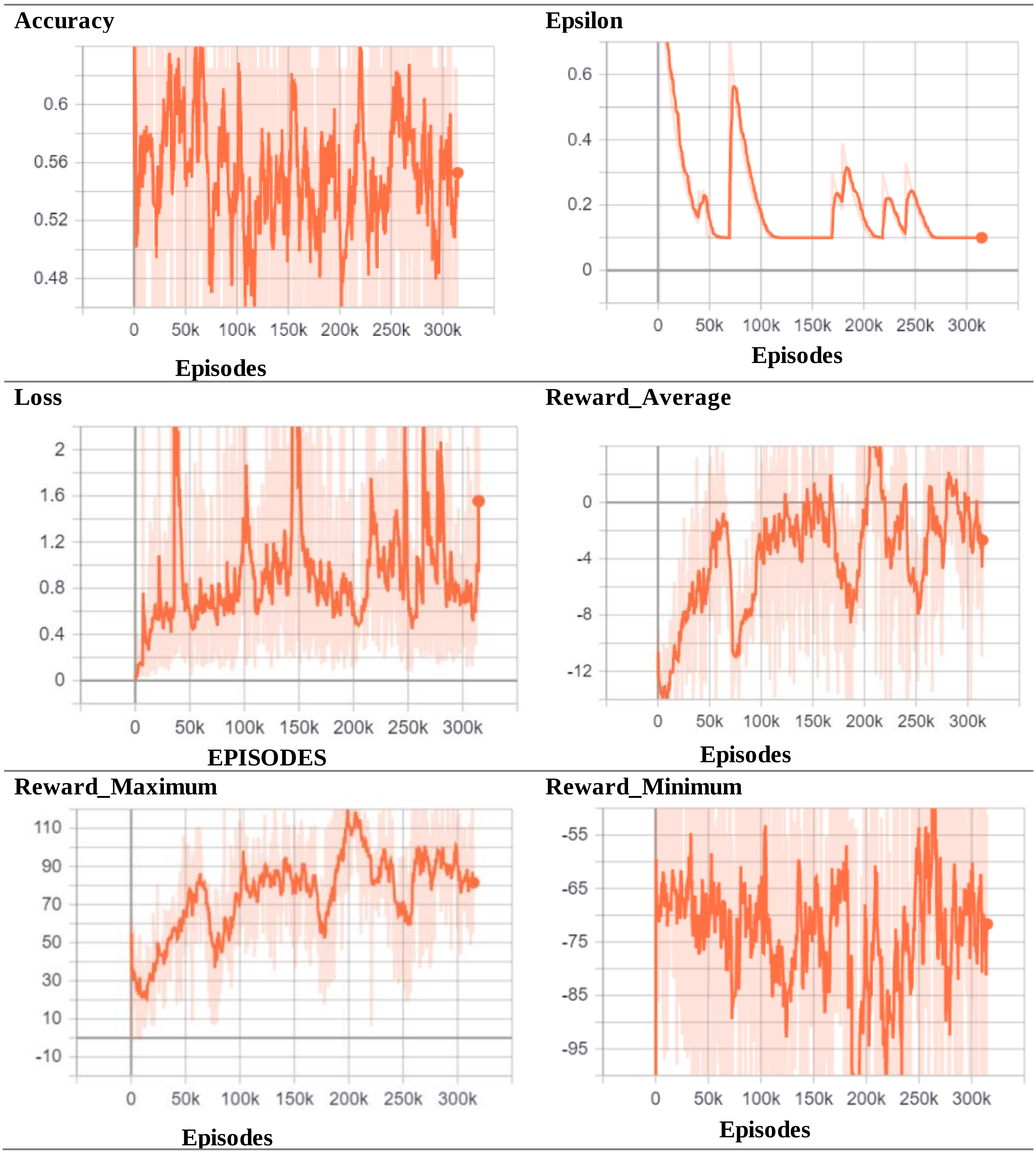


Fig. 6.2 Results of Training-2

The graphs show that during the training session, the maximum and average reward had an upward trend. This is a good sign as it shows that the car is learning something and trying to maximize this reward. The problem is that when we look at the learning episodes themselves, we realize that the car is driving but is not aware that it must get to a particular place. Moreover, it does not consider the directions of the lane, the car simply decides to stay on the road and avoids collisions.

6.3 Training 3:

For this training it was decided to implement a new reward system as the previous one did not consider anything about reaching a desired destination. In addition, a new waypoint system was created which consists of dividing a trip from point A to point B into small one-meter sections. The actions remained the same.

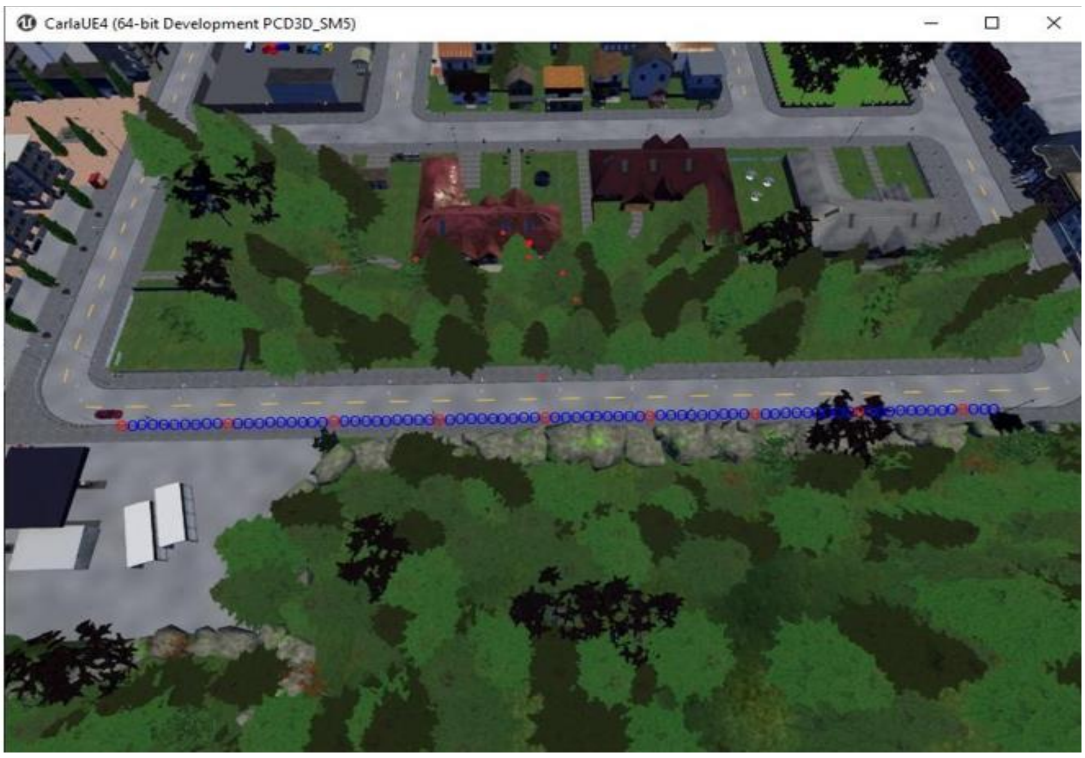


Fig. 6.3 Setting up of Way points between Source and Destination

Training conditions:

Total Training Time	4 hrs
No.of Episodes	1400
Actions	Action 0: Turn left Action 1: Accelerate the car Action 2: Turn right
Rewards	Reward 1: Collision: -10 points Reward 2: Arrived at destination: +20 points Reward 3: If distance to destination decrease: +5 Reward 4: If distance to destination increases: -5 Reward 5: short training episode: -10
Sensors Used	RGB camera Collision detection sensor Waypoint system
Model	CNN

Table 6.3 Training Conditions for CNN Model with extra parameters

Results:

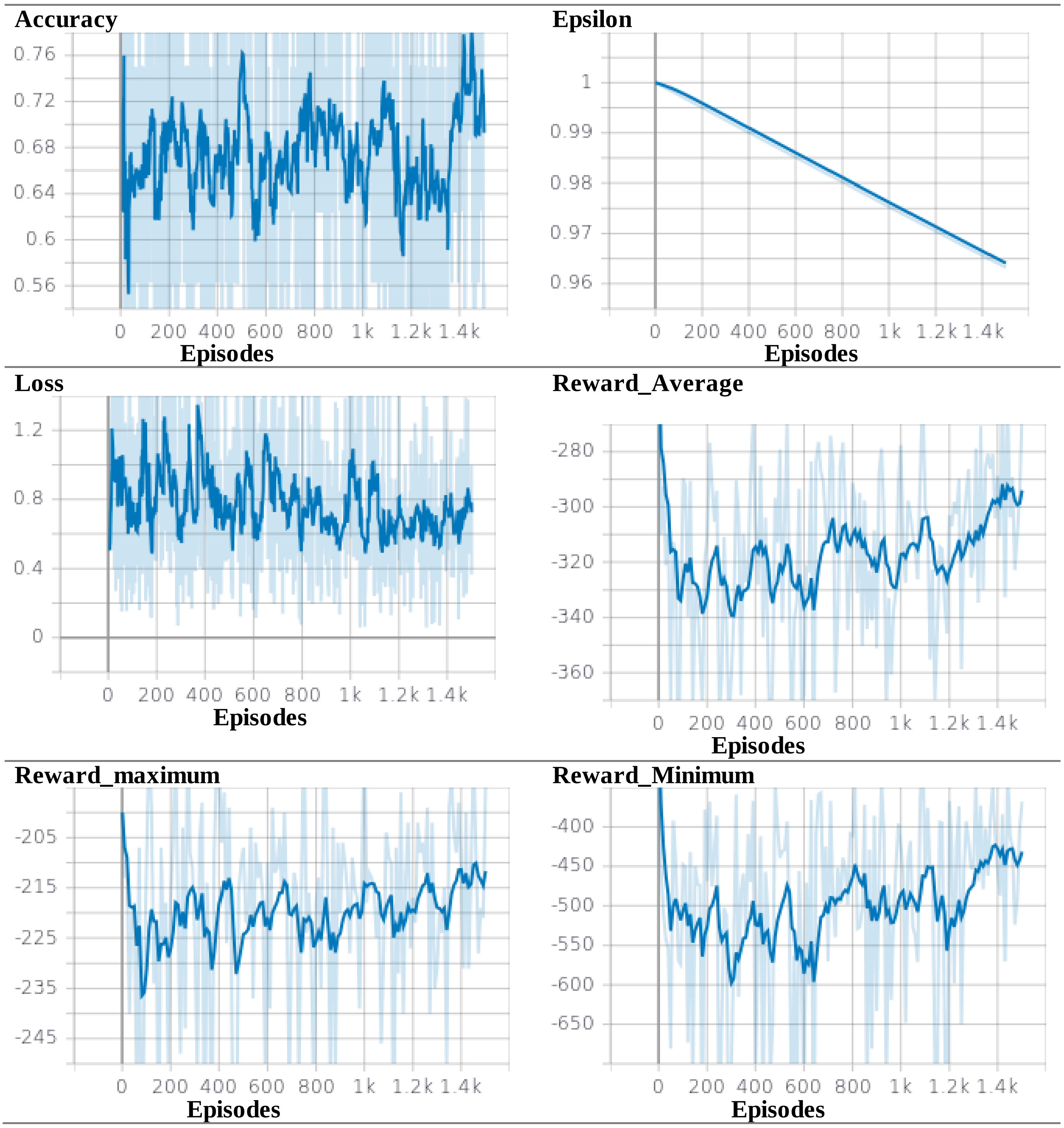


Fig. 6.4 Results of Traning-3

Results:

There is a slight upward trend that can be seen in average, maximum and minimum reward graphs that indicates the car is learning something, but the trend is not that high as we have limited its training to a smaller number of episodes. The agent is avoiding collisions and was able to move from source to destination, but the movement is not that smooth there is change in action for each frame. Also, it is not following the lane changes which need to handle.

CHAPTER 7 – CONCLUSIONS

Several studies have explored the use of DQN for self-driving cars using Carla, and some of the conclusions that can be drawn from these studies are:

1. DQN can effectively learn to drive in Carla: It has been shown that DQN can learn to navigate the Carla environment and complete driving tasks such as avoiding obstacles and reaching a target destination.

2. DQN can improve its performance with more training: Like most reinforcement learning algorithms, DQN's performance improves as it receives more training. This means that it can learn to drive more efficiently and avoid more obstacles with more training.

3. DQN is sensitive to the choice of hyperparameters: The performance of DQN can be sensitive to the choice of hyperparameters such as the learning rate, discount factor, and exploration rate. Optimal hyperparameters can vary depending on the specific task and environment.

4. DQN can struggle with complex scenarios: While DQN can perform well in simple driving scenarios, it can struggle with more complex scenarios that require long-term planning and decision-making. In such scenarios, more advanced reinforcement learning algorithms or other machine learning techniques may be necessary.

5. However, there are still several challenges and limitations to be addressed in the development and deployment of DQN-based self-driving cars. These include safety concerns, data requirements, computational resources, and legal and regulatory compliance.

6. Further research and development in this area will be necessary to address these challenges and improve the robustness and safety of DQN-based self-driving cars. Despite these challenges, DQN remains a promising approach for developing safe and efficient self-driving cars.

In summary, this project has demonstrated the effectiveness of DQN in training a self-driving car to navigate a simulated environment and has highlighted the challenges and limitations of this approach. It is hoped that this project will contribute to the development of safer and more autonomous self-driving cars in the future.

REFERENCES

[1] S. Sharma, G. Tewolde, and J. Kwon, “Lateral and longitudinal motion control of autonomous vehicles using deep learning,” in 2019 IEEE International Conference on Electro Information Technology (EIT). IEEE, 2019, pp. 1–5.

[2] D. Mújica-Vargas, A. Luna-Álvarez, J. de Jesús Rubio, and B. CarvajalGámez, “Noise gradient strategy for an enhanced hybrid convolutionalrecurrent deep network to control a self-driving vehicle,” Applied Soft Computing, vol. 92, p. 106258, 2020.

[3] Y. Xing, C. Lv, H. Wang, D. Cao, and E. Velenis, “An ensemble deep learning approach for driver lane change intention inference,” Transportation Research Part C: Emerging Technologies, vol. 115, p. 102615, 2020

[4] F. Hui, C. Wei, W. ShangGuan, R. Ando, and S. Fang, “Deep encoder– decoder-nn: A deep learning-based autonomous vehicle trajectory prediction and correction model,” Physica A: Statistical Mechanics and its Applications, vol. 593, p. 126869, 2022.

[4] X. Nie, C. Min, Y. Pan, K. Li, and Z. Li, “Deep-neural-network-based modelling of longitudinal-lateral dynamics to predict the vehicle states for autonomous driving,” Sensors, vol. 22, no. 5, p. 2013, 2022.

[5] L. Chen, X. Hu, B. Tang, and Y. Cheng, “Conditional dqn-based motion planning with fuzzy logic for autonomous driving,” IEEE Transactions on Intelligent Transportation Systems, 2020.

[6] J. Chen, C. Zhang, J. Luo, J. Xie, and Y. Wan, “Driving maneuvers prediction based autonomous driving control by deep monte carlo tree search,” IEEE Transactions on Vehicular Technology, 2020.

[7] Ó. Pérez-Gil, R. Barea, E. López-Guillén, L. M. Bergasa, C. GómezHuélamo, R. Gutiérrez, and A. Díaz-Díaz, “Deep reinforcement learning based control for autonomous vehicles in carla,” Multimedia Tools and Applications, vol. 81, no. 3, pp. 3553–3576, 2022.