2022/23

RL based planning of a self-driving car in CARLA

DEPARTMENT OF
COMPUTER SCIENCE

University of Liverpool
Liverpool L69 3BX

Abstract

The dissertation shows the implementation of the project to train a self-driving car in the Carla simulator using a reinforcement learning based algorithm. The project uses Carla to create a simple urban traffic scenario and a car for training and applies DQN (Deep Q-Network) algorithm to train the car's planner. The goal of the project is to implement the process of training the car with existing algorithm and to evaluate the results.

The project trains a CNN and a Xception neural network model separately, and after comparing the data obtained through Tensorboard, the CNN is selected for further training. The result is obtaining a self-driving car that can drive normally on the road with a certain probability of avoiding obstacles. Although the planner of this car is not a good driver, it can continuously learn from the environment to improve its performance, which proves the effectiveness and potential of the DQN algorithm. By continuing to optimise the algorithm and solve some problems of the project, it may be possible to train a better self-driving car.

Content

1. Introduction 4

1.1 Project Description 4

1.2 Theoretical introduction 5

2. Design (the code is included in the appendices) 6

2.1 Connecting to Carla's World & Creating a scenario 6

2.2 DQN algorithms 9

2.3 Main method 16

3. Implementation (Realisation) 17

3.1 Training process 18

3.2 Problems 20

4. Testing & Evaluation 21

4.1 Tensorboard data of two models 22

4.2 Continuing Testing 26

5. Conclusion 30

6. BCS Project Criteria & Self-Reflection 31

7. References 33

8. Appendices 33

8.1 Class "VehicleEvn" [1] [2] 33

8.2 Class "DQN" [2] 37

8.3 Class "ModifiedTensorBoard" [2] 40

8.4 Main method [2] 41

1. Introduction

This section provides an overall project description and a theoretical introduction of the algorithms used in the project. Its purpose is to provide some background information for the main part of the Dissertation, which includes Design, Implementation, Evaluation, Conclusion and BCS Project Criteria & Self-Reflection.

1.1 Project Description

The title of the project is *RL based planning of a self-driving car in CARLA*, which means using reinforcement learning based algorithms to train a planner for a self-driving car in a simulator. The project will use Carla, an open-source simulator based on a virtual engine, to create a simple urban traffic scenario, and train the planner of a self-driving car by DQN (Deep Q-Network) reinforcement learning algorithms to learn how to drive properly in the urban environment, such as following the correct road and avoiding some obstacles. According to the performance of the vehicle, evaluate and improve the algorithms.

The object of this project is to learn reinforcement learning algorithms and apply an existing algorithm to train a self-driving car model. At the same time, change the algorithm to enhance performance of the car according to evaluations. Moreover, the implementation process of the project needs participants to get familiar with the basics of Carla, such as the basic components of Carla and the syntax. Participants can choose an existing map in Carla and add any number of vehicle and pedestrian NPCs to it to create an urban traffic environment, and then add a single car as the object that will be trained and install some sensors on the car.

The focus part of the project is the use of reinforcement learning algorithms, which can help participants understand and learn existing algorithms, and then apply them to train self-driving car models. The goal of this project is not to train a self-driving car that meets a certain standard, so the final model trained may not be a competent driver. The whole project is a process of continuous exploration and refinement. Participants will observe how the car performs in the map, evaluate the algorithm and model based on the data obtained from the training, and modify it to improve the vehicle's performance so that the self-driving car model constantly approaches a normal vehicle. This will also allow participants to improve their data analysis skills, compare different training models and draw out their strengths and weaknesses. In summary, the result of the project includes the best car model that the participants can train and a series of conclusions from the performance evaluation.

1.2 Theoretical introduction

Carla is an open-source simulator for the study of autonomous driving that simulates an extremely realistic environment, such as a modern city with buildings, roads, vegetation and almost everything else that a real city has [1]. Users can add bule prints like vehicles, pedestrians and sensors to suit their needs. Sensors are connected to the user-controlled vehicle and different sensors can be used to collect data through a simulated scenario [1]. Carla has two main components: World, which refers to the simulated scene and is the equivalent of a server, and Client, which is where the user controls the simulated scene using Python. The client needs to connect to the server via a terminal to enable the user to perform a series of operations on the simulator.

Reinforcement learning is a subset of the machine learning field whose goal is to learn through the interaction of an intelligence with its environment so that it can make decisions autonomously to achieve optimal goals [3]. “State” and “Action” are two key concepts in reinforcement learning. By performing different actions in different states, an intelligence can be rewarded or punished differently, and it will adapt its learning strategy with the aim of increasing the cumulative reward value, so that the intelligence will gradually choose the action with a high reward value, thus enabling it to achieve the desired optimal action [3].

DQN (Deep Q-Network) is a classical algorithm in reinforcement learning, and the project uses DQN reinforcement learning algorithm to train the model. DQN is a deep learning-based RL algorithm which is based on an improvement of the Q-learning algorithm. In DQN there are two neural networks: one is the evaluation network (Q-network) and the other is the target network. The evaluation network is used to evaluate the Q-value of the current state and action, while the target network is used to calculate the Q-value of the next state and action, which is used to update the evaluation network [2].

The Q-value represents the long-term cumulative reward for taking an action in each state. In DQN algorithm, the Q-value function outputs the Q-values of all feasible actions at a given state and chooses the action with the highest Q-value to perform [2]. Through constant updating of the Q-value function, the neural network model learns how to choose the optimal action to obtain the maximum reward, so that the model gradually acts with the desired action. Q-values are therefore the goal of the DQN algorithm optimisation.

The core of the DQN algorithm is experience replay and fixed Q-goals. Experience replay stores an intelligence's experience in a replay buffer (memory), from which it is randomly selected for training. It makes better use of the relevance of the data and reduces the relationship between samples. Fixed Q-Target uses the target network to calculate Q-values rather than the evaluation network. This approach reduces jitter in Q-values and thus improves learning efficiency [2].

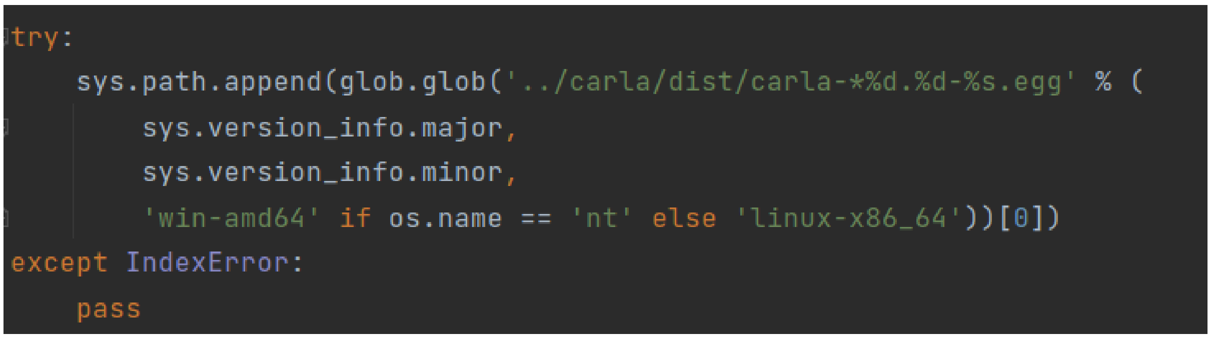
2. Design (the code is included in the appendices)

This section describes the code implementation of the project. It explains how each part of the entire program is constructed and connected and specifies the construction of the Carla environment and the implementation of the DQN algorithm. The program contains 3 classes, one to create the Carla environment, one to define a DQN algorithm, one to modify the Tensorboard, and a main method to run the whole program. The whole program implements a self-driving car model trained in a city map, with the training process divided into several episodes, each lasting a few seconds (the time can be modified at any time).

Note: Some parts of the code reference [1] or [2].

2.1 Connecting to Carla's World & Creating a scenario

This project uses the Anaconda virtual environment to configure Carla 0.9.11, and use the corresponding Anaconda environment in Pycharm to programme, using a python version of 3.7. The first step is to add the Carla Python API to the search path of the Python module as a way of enabling the client to communicate with the Carla server [2]. The code adds the path to sys.path and uses the glob module to find the path to Carla's egg file. If it finds a file that matches, it adds it to sys.path, allowing the compiler to import the Carla module at runtime, otherwise it executes pass and does nothing.



```
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
```

Figure 1: Add the Carla Python API to the search path of the Python module

Then, a class called "VehicleEvn" is created to represent a map environment (the code is included in the appendices), and in this class operations such as creating clients, loading maps, adding NPCs, adding training vehicles and sensors are performed. The attributes of the class include the display and the size of the camera sensor's image. The constructor for this class is shown below, where a Carla client (self.client) is created and connected to the local host ('localhost') and the default port number 2000. Set the client's time-out to 10 seconds using "set_timeout" method. Next, load an existing city map in the Carla simulator using "load_world" method. Then, obtain a blueprint of the "Dodge Charger" using "filter" method and assigned to the "self.vehicle_main" variable as the blueprint of the vehicle being trained. Finally, NPCs are initialised by "init_npc" method.

The "init_npc" is a method of class "VehicleEvn", which can add any number of NPC to the map. The algorithm structure of this method is as follows:

• Obtain spawn points from the current map.

• Select some models of vehicle from the blueprint library (users can choose the model of vehicle that they like).

• Set a max number of vehicles and prepare a list for them.

• Randomly select spawn points and spawn the chosen number of vehicles (use "try_spawn_actor" method). Put the vehicles in the list and set them to autopilot mode, allowing them to move on their own.

• The same applies to the algorithm for spawning pedestrians. As an extra for each walker, spawn a controller and assign it to him, and set up pedestrian navigation by having them move to a random location in the map.

The next step is to define a "set" method to spawn the vehicle used for training and add some sensors:

• Spawn the training vehicle at a random spawn point in the map and add it to an array of actor objects.

• Set the spectator's view to the side of the training vehicle to observe it.

• Set up a RGB camera sensor with a specified image size (320×240) and field of view, attach the RGB camera sensor to the training vehicle and add it to the array of actor objects. In addition, define a call-back function to process the image data of the RGB camera. It is triggered every time the sensor receives new data, which contain the raw images captured by the camera. The call-back function takes this raw data, converts it to a NumPy array, reshapes it to the desired image size, and extracts only the first three color channels (RGB). Then, it updates the "front_camera" attribute with the processed image data. If "SHOW_CAMERA" is set to True, the processed image is also displayed in a window using OpenCV library.

• Initialize a collision sensor with respect to the training vehicle and adds it to the list of actor objects. There is also a call-back function for it.

• Wait until the "front_camera" attribute is not none, indicating that the RGB camera sensor has started capturing images, start the episode.

• Return the captured "front_camera" image data.

Finally, define a “step” method to control the movements of the training vehicle and assign different reward values depending on the state of it:

- The parameter action indicates the action to be taken by the vehicle, 0 means turn left, 1 means go straight, 2 means turn right. Use “apply_control” method to set the throttle, steer and brake of the car.
- After applying the control, the code determines whether brake is needed by checking if the absolute steering value is greater than 0.5. If it is, the code applies both throttle and brake with appropriate values to slow down the car. If the steering value is less than 0.5, the code applies only throttle and no brakes.
- Get the velocity of vehicle and convert the speed of the vehicle from vector to scalar form.
- Set the reward value. If a vehicle collision is detected, the value of “done” (indicating whether the episode is over) is true and reward is a negative value (indicating punishment); If the speed of the vehicle is less than 50km/h, the value of “done” is false and reward is a negative value; If the speed is greater than or equal to 50km/h, then the value of “done” is false and reward is a positive value.
- If the time of an episode exceeds a limit (SECOND_PER_EPISODE), the episode is end.

2.2 DQN algorithms

The DQN algorithm in this project is based on the implementation of the Tensorflow framework, an open-source framework widely used for machine learning that provides a rich library of methods and functions to efficiently build and train neural network models. The version of Tensorflow is 1.14, which is paired with Keras 2.2, corresponding CUDA (Compute Unified Device Architecture) and cuDNN (CUDA Deep Neural Network library).

The first step is to create a class called “DQN” to include algorithms (the code is included in the appendices). The constructor for this class is shown below.

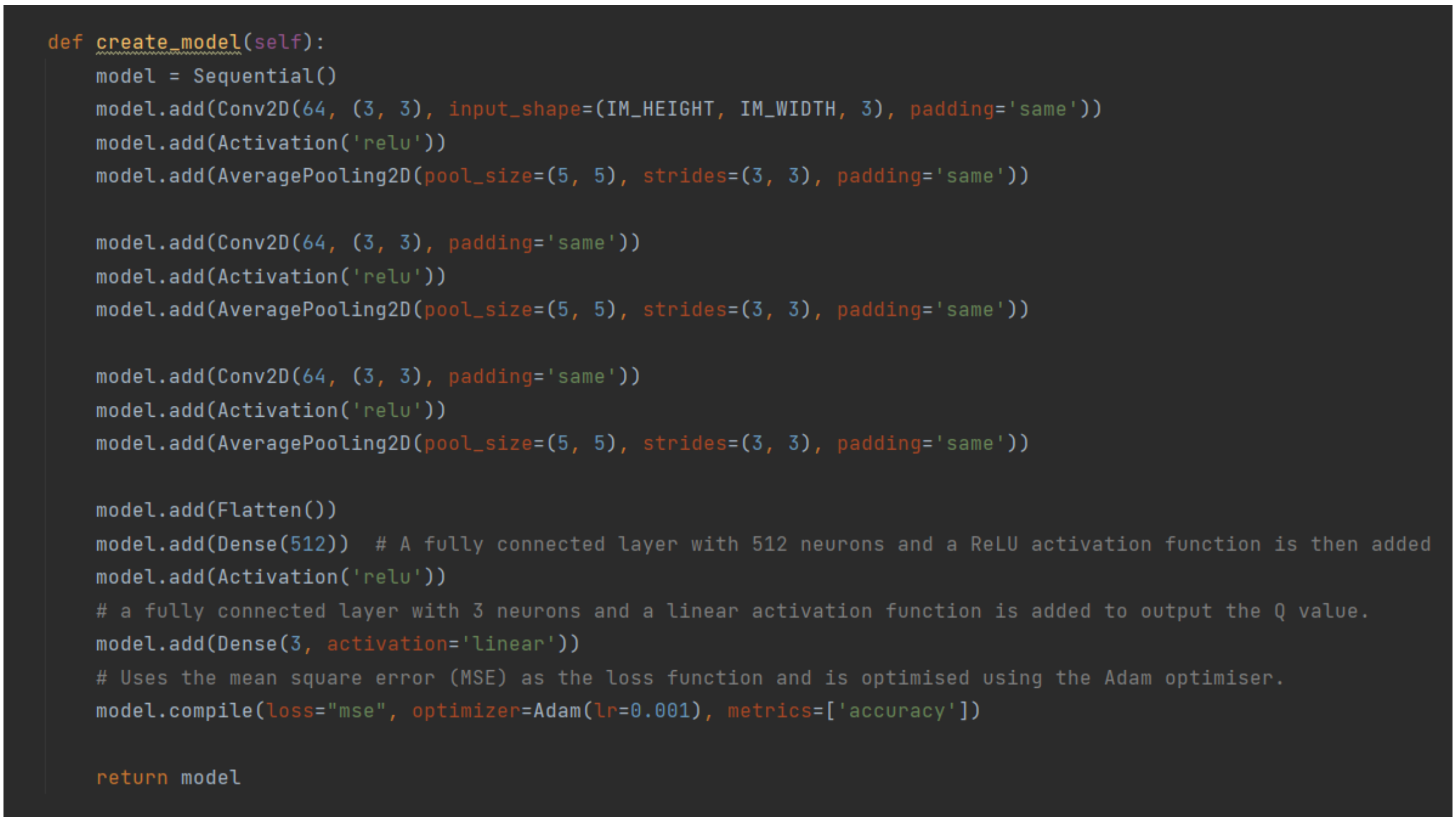
- Create an evaluation network and a target network model, and set target network's weights using evaluation network's weights, which initializes the target network with the same parameters as the evaluation network.

- Create a double-ended queue to store the replay memory of the agent that includes the agent's experience during training, and the replay memory can be sampled.
- Create a tensorboard object for logging the agent's training progress. Class "ModifiedTensorBoard" will be introduced later.
- Get the default TensorFlow graph, which is used to manage the TensorFlow operations and variables within the agent.
- Initialize a boolean value that indicates whether the agent should terminate or continue running and a value indicating whether the agent's training has been initialized. Initialize a variable to keep track of the last logged episode and a variable to keep track of the number of times the target network has been updated.

Then, define the neural network model for training. This project trains two models, one is a simple CNN (Convolutional Neural Network) model with 3 convolutional layers, each containing 64 convolutional kernels (3×3) [2]. The architecture of it:

- Convolutional Layers: Three convolutional layers are added sequentially, each followed by a ReLU (Rectified Linear Unit) activation function and average pooling. These layers are responsible for extracting features from the input image.
- Flatten Layer: The output from the convolutional layers is flattened into a 1-dimensional vector to be fed into the fully connected layers.
- Fully Connected Layers: A fully connected layer with 512 neurons and a ReLU activation function is added to learn high-level representations of the flattened features.
- Output Layer: A fully connected layer with 3 neurons (corresponding to the Q-values of the three possible actions) and a linear activation function is added to output the Q-values.

- Compilation: The model is compiled using mean squared error (MSE) as the loss function and the Adam optimizer with a learning rate of 0.001.

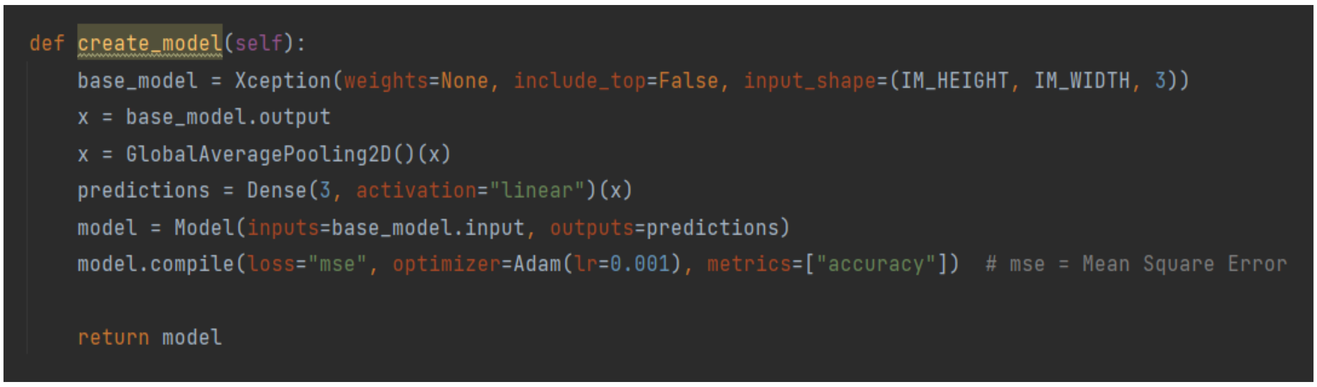


```python
def create_model(self):
    model = Sequential()
    model.add(Conv2D(64, (3, 3), input_shape=(IM_HEIGHT, IM_WIDTH, 3), padding='same'))
    model.add(Activation('relu'))
    model.add(AveragePooling2D(pool_size=(5, 5), strides=(3, 3), padding='same'))

    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(Activation('relu'))
    model.add(AveragePooling2D(pool_size=(5, 5), strides=(3, 3), padding='same'))

    model.add(Conv2D(64, (3, 3), padding='same'))
    model.add(Activation('relu'))
    model.add(AveragePooling2D(pool_size=(5, 5), strides=(3, 3), padding='same'))

    model.add(Flatten())
    model.add(Dense(512))  # A fully connected layer with 512 neurons and a ReLU activation function is then added
    model.add(Activation('relu'))
    # a fully connected layer with 3 neurons and a linear activation function is added to output the Q value.
    model.add(Dense(3, activation='linear'))
    # Uses the mean square error (MSE) as the loss function and is optimised using the Adam optimiser.
    model.compile(loss="mse", optimizer=Adam(lr=0.001), metrics=['accuracy'])

    return model
```

Figure 2: CNN model

The other one is a Xception model [2]:

- The Xception model is used as the base model. This model is pre-trained on ImageNet and can extract high-level features from images.
- Global Average Pooling: The output from the base model is passed through a global average pooling layer, which reduces the spatial dimensions of the features while retaining the important information.
- Dense Layer: A fully connected dense layer with 3 neurons and a linear activation function is added to produce the Q-values for the given state.
- Model Compilation: The model is compiled using mean squared error (MSE) as the loss function and the Adam optimizer with a learning rate of 0.001.



```
def create_model(self):
    base_model = Xception(weights=None, include_top=False, input_shape=(IM_HEIGHT, IM_WIDTH, 3))
    x = base_model.output
    x = GlobalAveragePooling2D()(x)
    predictions = Dense(3, activation="linear")(x)
    model = Model(inputs=base_model.input, outputs=predictions)
    model.compile(loss="mse", optimizer=Adam(lr=0.001), metrics=["accuracy"])  # mse = Mean Square Error

    return model
```

Figure 3: Xception model

Define a method to update the replay memory with a tuple “transition” containing the experience, such as current state, action, reward, new state and “done”. The purpose of this method is to add a transition to the replay memory that is a buffer that stores data about transitions between an intelligence and its environment. A transition is the process by which an intelligence receives a reward for taking an action from the current state and moves to a new state. A transition usually also includes a flag (done) to indicate whether the end state has been reached. By adding transitions to the experience replay memory, the intelligence can learn from past experiences and use this data to improve its strategies and behaviour.

The key method in class “DQN” is “train” method, which uses deep RL algorithms to optimise the weights of the neural network model to better predict the Q-value of each action, enabling the model to learn and make optimal action choices in the environment. The architecture of the method is shown below:

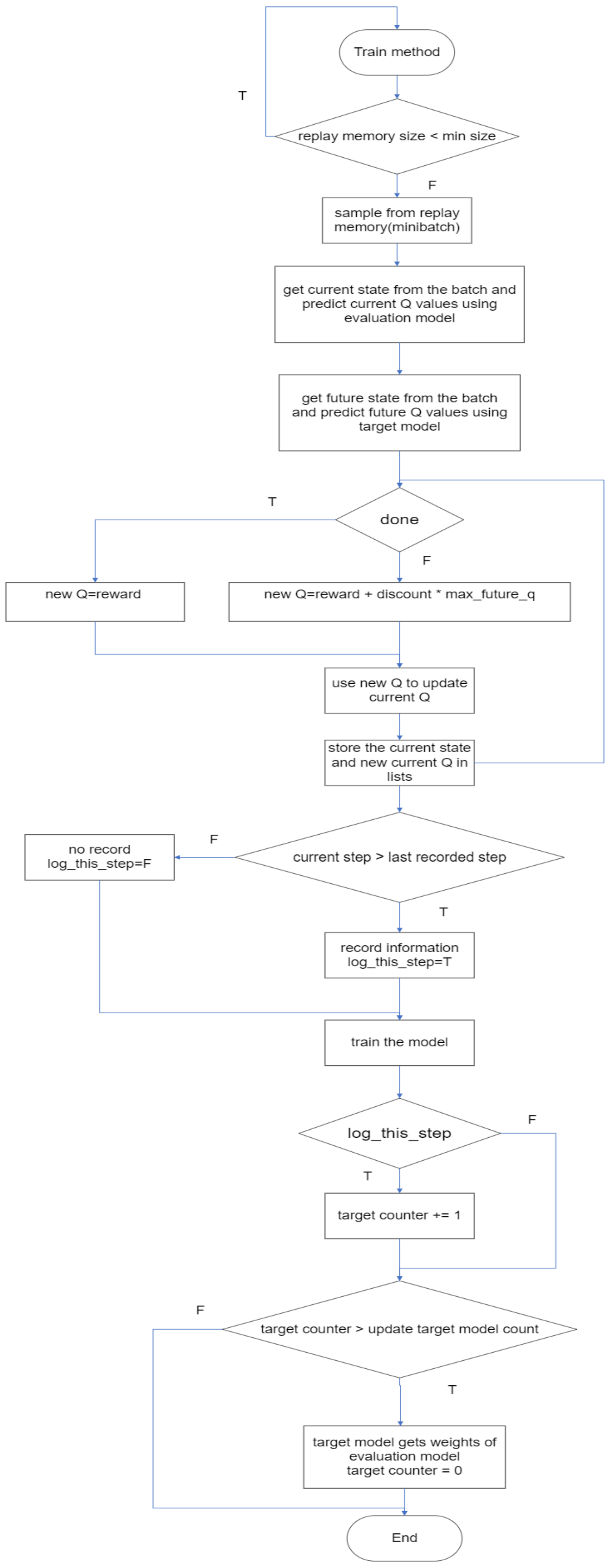

Train method
T
replay memory size < min size
F
sample from replay memory(minibatch)
get current state from the batch and predict current Q values using evaluation model
get future state from the batch and predict future Q values using target model
T
done
F
new Q=reward
new Q=reward + discount * max_future_q
use new Q to update current Q
store the current state and new current Q in lists
F
no record log_this_step=F
current step > last recorded step
T
record information log_this_step=T
train the model
F
log_this_step
T
target counter += 1
F
target counter > update target model count
T
target model gets weights of evaluation model target counter = 0
End

Figure 4: The flowchart of “train” method

- Use a “if” statement to check if the length of the replay memory is less than the minimum replay memory size (MIN_REPLAY_MEMORY_SIZE). If the memory size is smaller than the minimum requirement, the training process is skipped, and the method returns without performing any further computations. The purpose of it is to delay the start of training until there is enough experience stored in the replay memory, which can improve the stability and effectiveness of the training process.
- Define a variable “minibatch” to randomly sample a batch of data from the memory.
- Convert the data into inputs for the current state and the new state and use the model to make predictions. The algorithm is extracting the element of current state in the batch and converting it to a NumPy array. Normalise by dividing by 255 to scale the range of pixel values to between 0 and 1. Then, the normalised current state is predicted using the current model (evaluation model) to obtain a list of Q-values for the current state. It is the same for the prediction of new Q-values list (using target model).
- Use a for loop to calculate the new Q-value and use the new Q-value to update the current Q-value. Store the updated current Q-value and current state in two lists respectively.
- Determine if the current training information needs to be recorded by checking if the current training step count is larger than the last recorded step count.
- Train the model: the model is trained using the “model.fit()” method. DQN algorithm uses the current state as input and the future Q-value as output to train the model for iterative optimization with cumulative rewards. This method accepts the input data and the target(output) data and trains the model according to the given parameters. Specifically, “np.array(Status)/255” indicates that the input data is normalised by dividing by 255 to scale the pixel values to a range of 0 to 1, and “np.array(q_value)” indicates the target data, i.e. the updated result of the Q-value. “TRAINING_BATCH_SIZE” specifies the size of each training batch, set verbose to 0 to turn off the log output of the training process, set shuffle to false to not randomly rearrange the training samples. Next, perform call-backs at the end of each training step, “self.tensorboard” is used as the call-back function to

record the logging information of the training process, if “log_this_step” is false, the call-back operation is not executed. By using this method, the model can be trained, and its parameters can be updated to optimise its performance and improve the accurate estimation of Q-values for state-action pairs.

- Ensure that the target model is updated regularly to track improvements to the current model, which enables an update loop of the two neural network models.

Moreover, define a method to obtain a model's estimate of the Q-value of the state by passing in state data for use in decision action selection or to assess the value of the state. Finally, define a method to train the model in a continuous loop. It generates a random input x and output y for initialising the model. The model is then trained once using this random pair of inputs and output, using “fit” method, and setting verbose to false to not print the training process and batch size to 1 to use it for each training session. Set training initialized flag to true to indicate that the model has been initialized. Then, train the model in an infinite loop by executing the “train” method. The loop terminates when the terminate flag is set to true. Such a loop training method can be used to continuously train the model to gradually improve the performance and accuracy of the model.

In addition, this code has a “ModifiedTensorBoard” class that inherits from the “TensorBoard” class in TensorFlow and overrides and extends some of its methods (the code is included in the appendices) [2].

- Override the initialization method for setting the initial number of step and log writer.
- Override the method to set the model, leaving it empty and not doing anything, so as to avoid creating a default log writer.
- Override the method “on_epoch_end” to save logs with the set step number.
- Override the method “on_batch_end” and “on_train_end” to make them become empty methods.

- Custom a method for saving custom metrics information. The method creates a writer and writes the custom metric information to the log, then closes the writer.

By using the ModifiedTensorBoard class, it is possible to save custom metric information during training and write it to the same log file for subsequent analysis and visualisation.

2.3 Main method

- Set TensorFlow's GPU configuration: specify the proportion of GPU memory that each process can use, set via the “MEMORY_FRACTION” parameter. Then, apply the GPU configuration to TensorFlow's session and set it as the default session. This means that each process can use up to the specified percentage of GPU memory, controlling the amount of memory TensorFlow can allocate to the GPU to avoid overloading the GPU memory and thus allocate resources appropriately between multiple tasks.
- Set the parameters and random seeds and create a file path to save the model.
- Create the environment class object “env” and the DQN class object “agent”, and a new thread as a background thread. The target function of the thread is the loop training method. By executing the training loop in a separate thread, the training process can be separated from the main thread, thus avoiding blocking the main thread. This allows training and other operations to be carried out simultaneously, increasing the concurrency and efficiency of the program.
- Apply the method (get_qs) to the “agent” object to obtain the predicted Q value. Define a for loop to start iterating over the episodes, using "tqdm" to display the progress bar. At the beginning of each episode, clear the collision record and set the step of the tensorboard to the current episode number. Then, the parameters are initialised, and the environment is initialised with the "set" method (creating everything). Define an embedded while loop to perform the process of action selection, environment interaction and experience replay at each step. In the loop, the model takes an action in the current state, if the random number is greater than epsilon (control the probability of exploration of the model), take the optimal action predicted by the model, otherwise choose the action randomly. Next, the model executes the selected action by interacting with the environment and obtaining the new state, reward and termination flag “done”. Also,

store the current experience (current state, action, reward, new state, done) into the experience replay buffer using the “update_replay_memory” method. At the same time, update the current state, total reward and step number. If “done” is true, break the loop and destroy the actors.

- In the for loop, the next step is to append episode rewards to a list and log stats (every given number of episodes), which means to calculate the average reward, maximum reward and minimum reward, and update data to Tensorboard.
- Then, decay epsilon to make the intelligence gradually explore less during training move to choose better actions. The aim of this is that in the early stages of training, the intelligence explores more to discover new movements and strategies. As training progresses, the value of epsilon decreases, and the intelligence is more inclined to select known superior movements to improve performance and stability.
- Finally, when all episodes have been executed, execute the termination procedure and save the model in the specified file path.

3. Implementation (Realisation)

This section describes the implementation of the code in the Carla simulator, including the training process of the self-driving car model, the data obtained by Tensorboard, and some of the problems encountered when running the code.

The evaluation and target neural network models are continuously updated through interaction with Carla's simulated environment. The desired outcome of the project is that the cumulative reward value should show an increasing trend with iterating of Q-values, the car model is biased towards selecting actions with high reward values and its behaviour should move closer to that of a competent driver.

During training period, the program uses TensorBoard to record data of the training, such as rewards, lose rates and accuracy, to monitor the performance of the model. This allows developers to visually observe the training progress of the model, as well as identify any potential problems or chances for improvement.

3.1 Training process

The figure below shows the performance of the self-driving car model in a city environment. The program trains the car through many episodes, each lasting for ten seconds. During each episode, the algorithms encourage the car to incrementally accelerate over a speed to avoid it spinning in place. In the early stages of training, the car cannot find the correct direction of travel accurately, it may exhibit disoriented behaviour, such as random movement or frequent changes in driving line. This occurs because the model has not yet acquired a complete understanding of the correct driving strategy, and it just keeps exploring the environment.

In addition, the current episode ends immediately when the car is involved in a collision. This helps the training process to quickly identify incorrect behaviour and correct it. By constantly interacting with the environment and obtaining feedback, the self-driving car model can gradually learn and optimise driving behaviour, gradually acquiring the correct driving skills and decision-making abilities. Overall, the initial phase of training demonstrates the exploratory behaviour of the self-driving car model in an urban environment, and the model needs to be trained over several episodes to gradually improve driving accuracy and effectiveness.

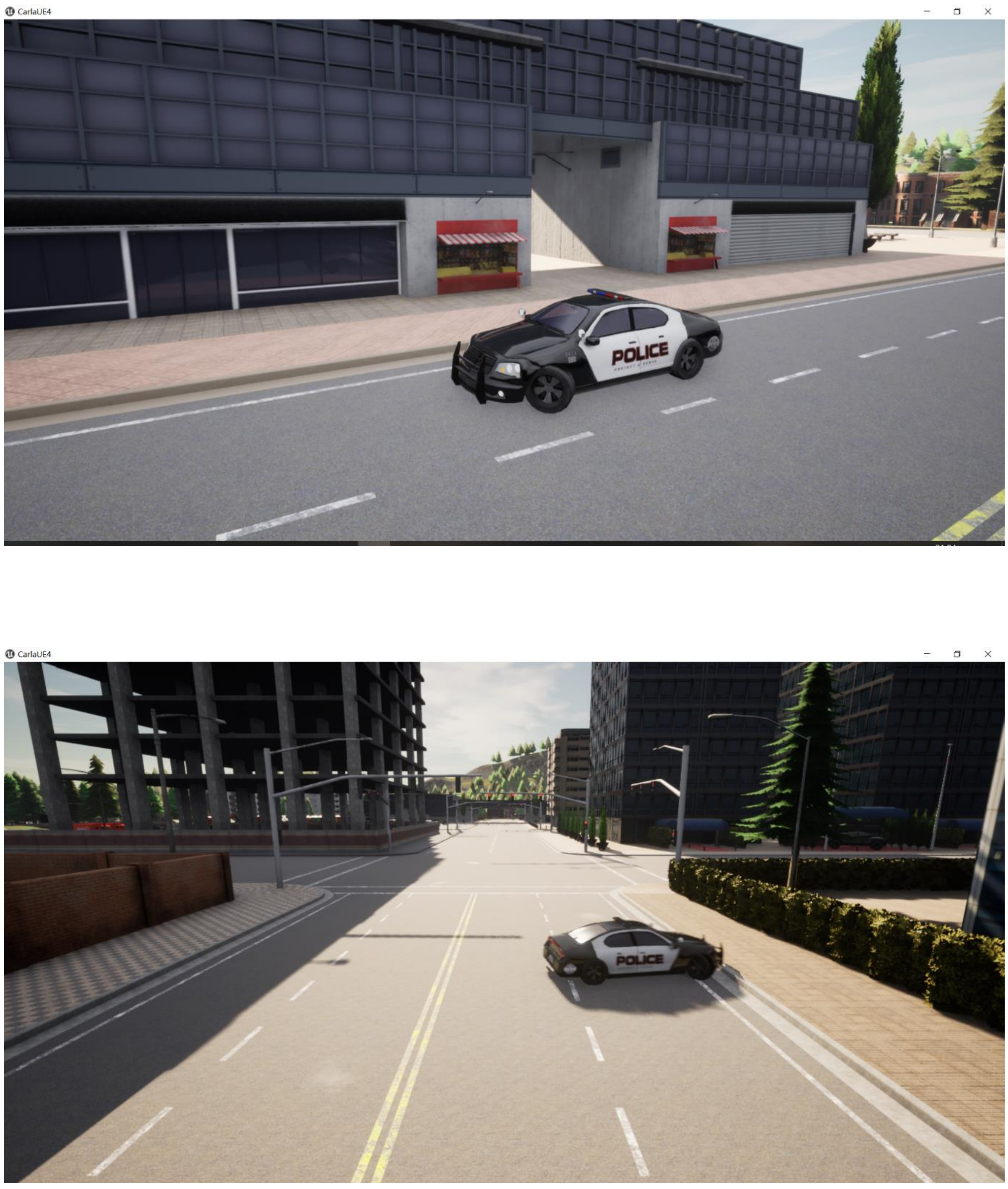


Figure 5: The vehicle in the early stages of training

After a period of training, the program sets the epsilon to decay continuously so that the model gradually moves from exploration mode to performing optimal actions. At this point it can be observed that the self-driving car is able to drive normally on the road and is able to stay in its lane without deviating. However, the model still has flaws, such as its poor performance in avoiding obstacles, which often leads to accidents such as collisions. Therefore, some of the parameters of the training process may need to be modified.

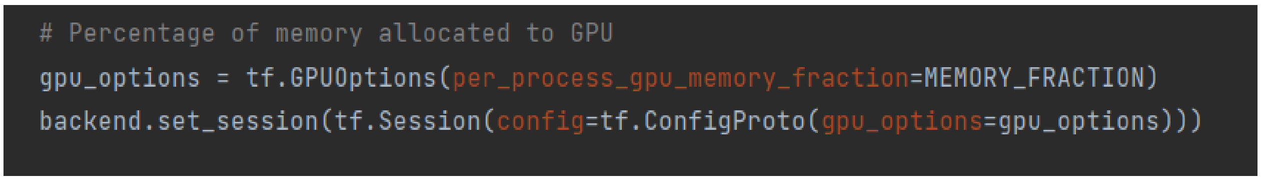
Figure 6: The vehicle after a period of training

This project trains a CNN model and a Xception model separately. After a period of training, the two models obtained similar results, both showing that they were good at driving in a straight line and not good at avoiding obstacles. However, there is a degree of difference in the data in the Tensorboard between the two models, which will be illustrated in the evaluation module below.

3.2 Problems

- When training the Xception model, this model requires more memory due to the large amount of computation. Therefore, during the training process, the problem of running out of memory may be encountered. To solve this problem, some parameters can be turned down to reduce memory usage, such as reducing the size of the images acquired by the camera, reducing the size of replay memory and the size of the batches sampled from memory. These adjustments may affect the effectiveness of the training, and therefore need to be traded off and adjusted on a case-by-case basis. CNN models generally do not have this problem, so the above parameters can be adjusted to normal to train the model.

- When allocating memory to the GPU, if too much memory is allocated to each process, it may cause the program to report an error, so the “Fraction” value needs to be reduced appropriately (set it to 0.4 or 0.6).



```
# Percentage of memory allocated to GPU
gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=MEMORY_FRACTION)
backend.set_session(tf.Session(config=tf.ConfigProto(gpu_options=gpu_options)))
```

Figure 7: The method of allocating memory to GPU

- Version problems with Tensorflow: Even if the versions of Tensorflow and CUDA correspond to each other, sometimes the system will report an error, such as the cuDNN file cannot be found or the cuDNN version is incorrect. The solution is to adjust the version of Tensorflow, for example, change version 1.15 to 1.14 (but only if it matches the CUDA version).

4. Testing & Evaluation

In this section, the Tensorboard data for both models are showcased, providing insights into various metrics such as accuracy, loss values, epsilon decay curves, and reward values. The data is analyzed and compared to gain valuable insights. Based on the conclusions drawn from the analysis, the model parameters are optimized and modified accordingly. This leads to further training iterations with the updated model, aiming to enhance the performance of the models. The iterative optimization process ensures continuous improvement and refinement of the models based on the insights gained from the Tensorboard data analysis.

4.1 Tensorboard data of two models

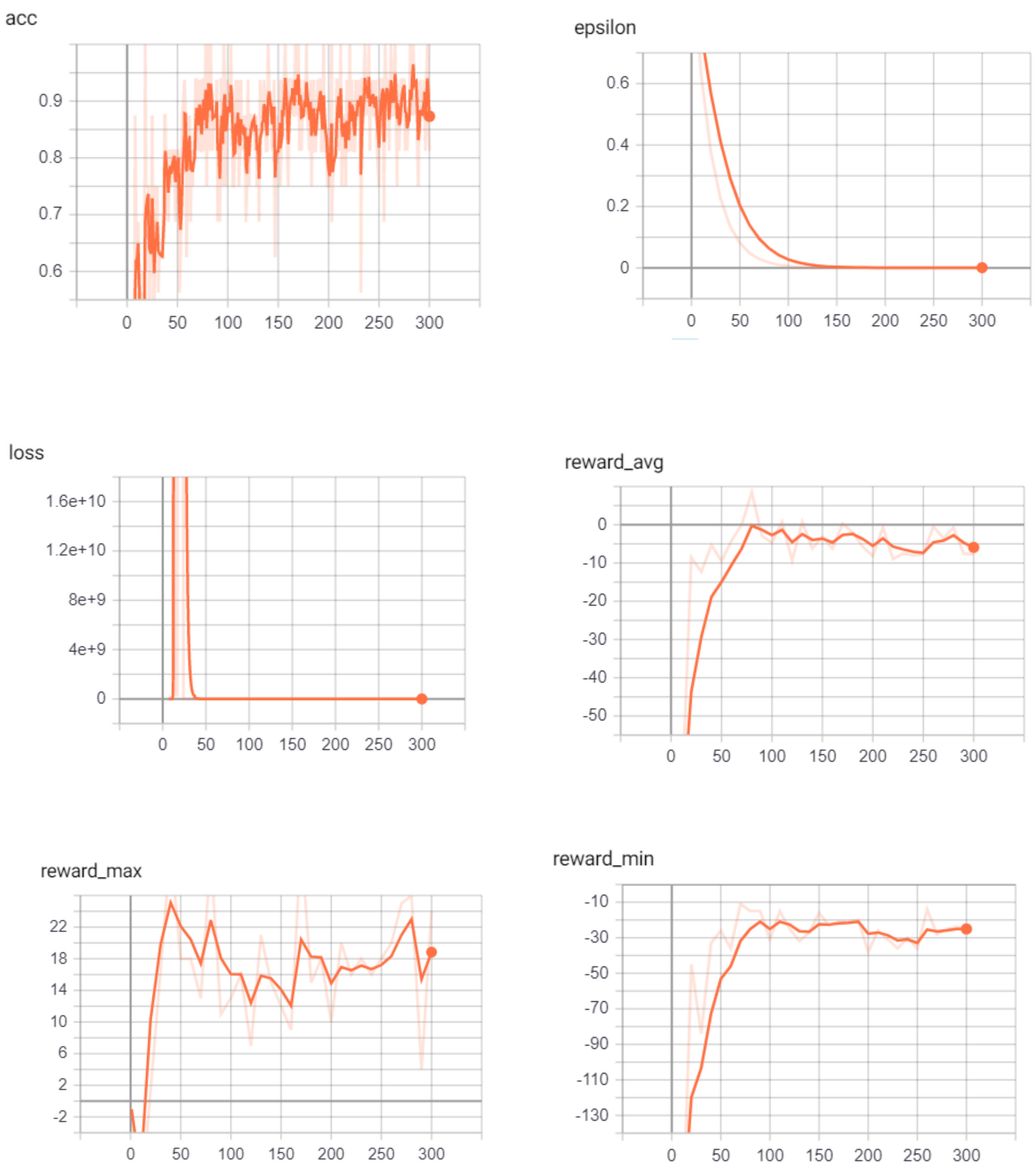


Figure 8: The data of Xception model

Analysis of the data on Tensorboard shows that the accuracy of the Xception model grows gradually during training and eventually reaches a relatively stable high level, usually around

0.9. The loss rate of the Xception model is explosive, with loss values reaching orders of magnitude in the thousands even when reaching a steady state. This may imply that the model encountered some difficulties during the training process, making it difficult to reduce the loss effectively. Possible causes include noise or imbalance in the training data, the complexity of the model architecture, etc.

In terms of reward values, it is observed that all three reward indicators (maximum, minimum and mean) of the Xception model showed a tendency to increase rapidly and then stabilise (with the maximum reward value fluctuating more). This indicates that the model has a strong learning ability during the learning process and can adapt quickly to the environment and make positive progress. However, the tendency for the reward values to stabilise may mean that the model has reached a certain saturation point in learning and is unlikely to continue to make significant improvements. This may be because the model has already mastered the main patterns and strategies in the environment (the decay of epsilon) and further learning becomes relatively difficult to make further progress.

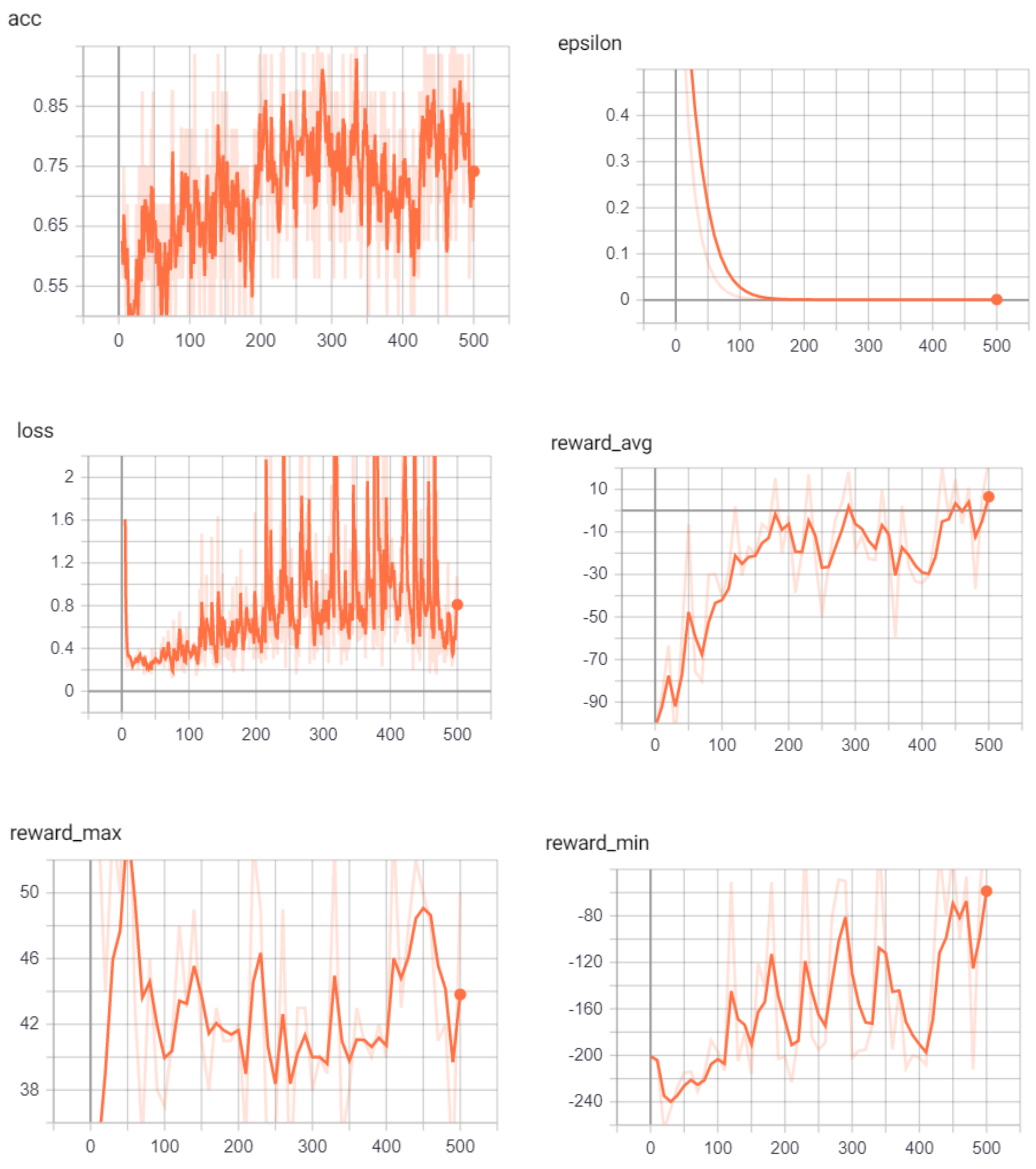


Figure 9: The data of CNN mode

The average accuracy of the CNN model is around 0.7 relative to the Xception model, indicating that its average learning ability may not be as good as that of the Xception model. However, the CNN model has a low loss rate, usually fluctuating between 0 and 2, with the occasional larger loss likely due to the model encountering a poor environment. For the reward values, although the maximum and minimum reward values of the model fluctuated considerably, the average reward value tended to increase overall and continued to grow, which is a positive sign that the model still has the potential for further learning and can continue to be trained.

In addition, epsilon decay is intended to allow the model to gradually explore less and select better actions during training. However, current observations show that attenuating epsilon to lower values results in models (both Xception and CNN) performing a single action and a decrease in learning rate. Therefore, adjusting the epsilon decay rate is necessary to avoid the model falling into over-exploration or being too conservative, which is to maintain some exploration capability and facilitate continuous learning and improvement of the model during the training process.

In summary, analysis of the Tensorboard data shows that the Xception model performs well in terms of accuracy and reward value but has a high loss rate and may face some training difficulties. In contrast, the CNN model has a lower accuracy rate, but a smaller loss rate and an overall upward trend in reward value, showing potential for continued learning.

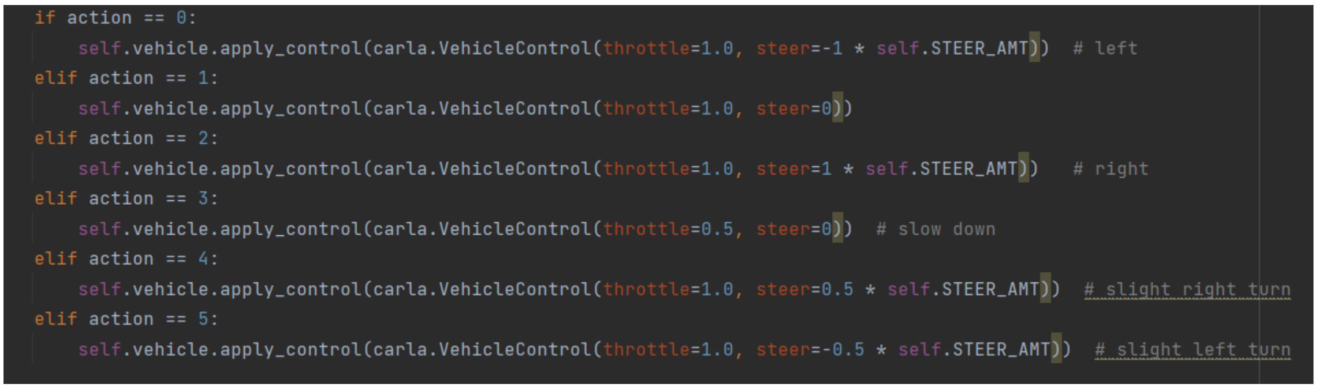
Therefore, the current conclusion is that both models are capable of learning in their environment. The Xception model is relatively complex and therefore has higher accuracy and greater learning capability. However, this can also lead to overfitting of the model to the training data, making it perform poorly against unseen data, with high loss values and overfitting problems [2]. In addition, training the Xception model requires more memory and the current limited GPU memory leads to the result to reduce parameters and simplify the algorithm to achieve training, which is the limitation of the Xception model in the project (it cannot be further optimised). In contrast, the CNN model is relatively simple and can be trained better by setting the pixels of the images acquired by the camera and the capacity of the replay memory to a higher level. Although the accuracy of this model is lower than that of Xception, it is sufficient to be used to improve the performance of the car (which may require longer training time). In addition, the smaller loss rate shows that the CNN model is likely to be more stable during training and has a lower risk of overfitting problems.

Based on analysis of the data, the project decides to select the CNN model to continue training the self-driving car model and to modify some parameters to achieve more optimised results. The expectation is that the vehicle’s rate of avoiding obstacles can increase to make better reactions to different driving environments. Such an optimisation process will help to further improve the performance and reliability of the self-driving vehicle.

4.2 Continuing Testing

The training of the CNN model is continued by modifying the following parameters:

- Adjust the reward value to -200 or lower on vehicle collisions to increase the punishment.
- Decreasing the decay rate of the epsilon and increasing the minimum value of the epsilon (e.g., 0.2). This allows the self-driving car to increase its exploration time and the minimum epsilon ensures that the car retains a certain rate of exploration during training, while also giving it sufficient opportunity to exploit what it has learnt.
- Increase the duration of each episode to give the model more time to explore.
- Define more movements for the car model, such as deceleration, slight left or right turns, to make it better suited to urban driving conditions.



```
if action == 0:
    self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=-1 * self.STEER_AMT))  # left
elif action == 1:
    self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=0))
elif action == 2:
    self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=1 * self.STEER_AMT))   # right
elif action == 3:
    self.vehicle.apply_control(carla.VehicleControl(throttle=0.5, steer=0))  # slow down
elif action == 4:
    self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=0.5 * self.STEER_AMT))  # slight right turn
elif action == 5:
    self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=-0.5 * self.STEER_AMT))  # slight left turn
```

Figure 10: The change of actions

As the training continued, the model's performance in avoiding collisions improved. Although the model cannot completely eliminate the risk of collision, its chances of avoiding obstacles improved. This means that the model can make better judgements and take actions when faced with different driving scenarios, minimising the probability of collision with an obstacle. In addition, with the addition of a minimum epsilon, the model not only performs the optimal action, but also continues to perform some exploration. This provides additional possibilities for improving the performance of the model. By maintaining a certain level of exploration behaviour, the model can discover new strategies and actions to cope with complex traffic environments and unexpected situations. Such an exploration-utilisation balance allows the model to improve performance while remaining adaptable to unknown situations. The picture below shows a scenario where a car dodges by swerving when it is about to collide with a fence.

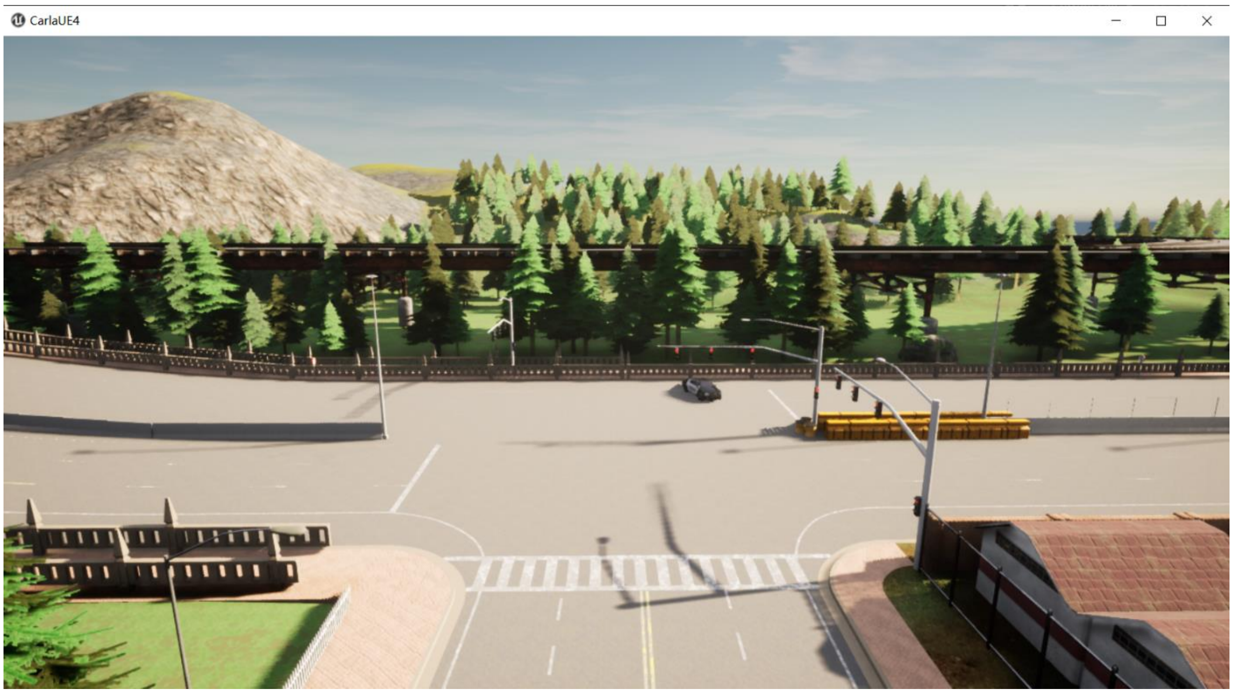


Figure 11: The vehicle that has avoided an obstacle

The figure below shows the data from this iteratively trained CNN model. Its average accuracy is around 0.6, with a fluctuation of roughly 0.2, which is generally passable, as too much accuracy is not necessarily a good thing. Because of the higher penalty values, the model loss values are slightly elevated compared to before, with sudden prominent losses from time to time, but this is not to an explosive degree and still falls within the normal range. In addition, the reward values, similar to before, continue to increase while fluctuating and have a tendency to continue to rise. Combined with the model's performance in the environment, it does appear to have been optimised.

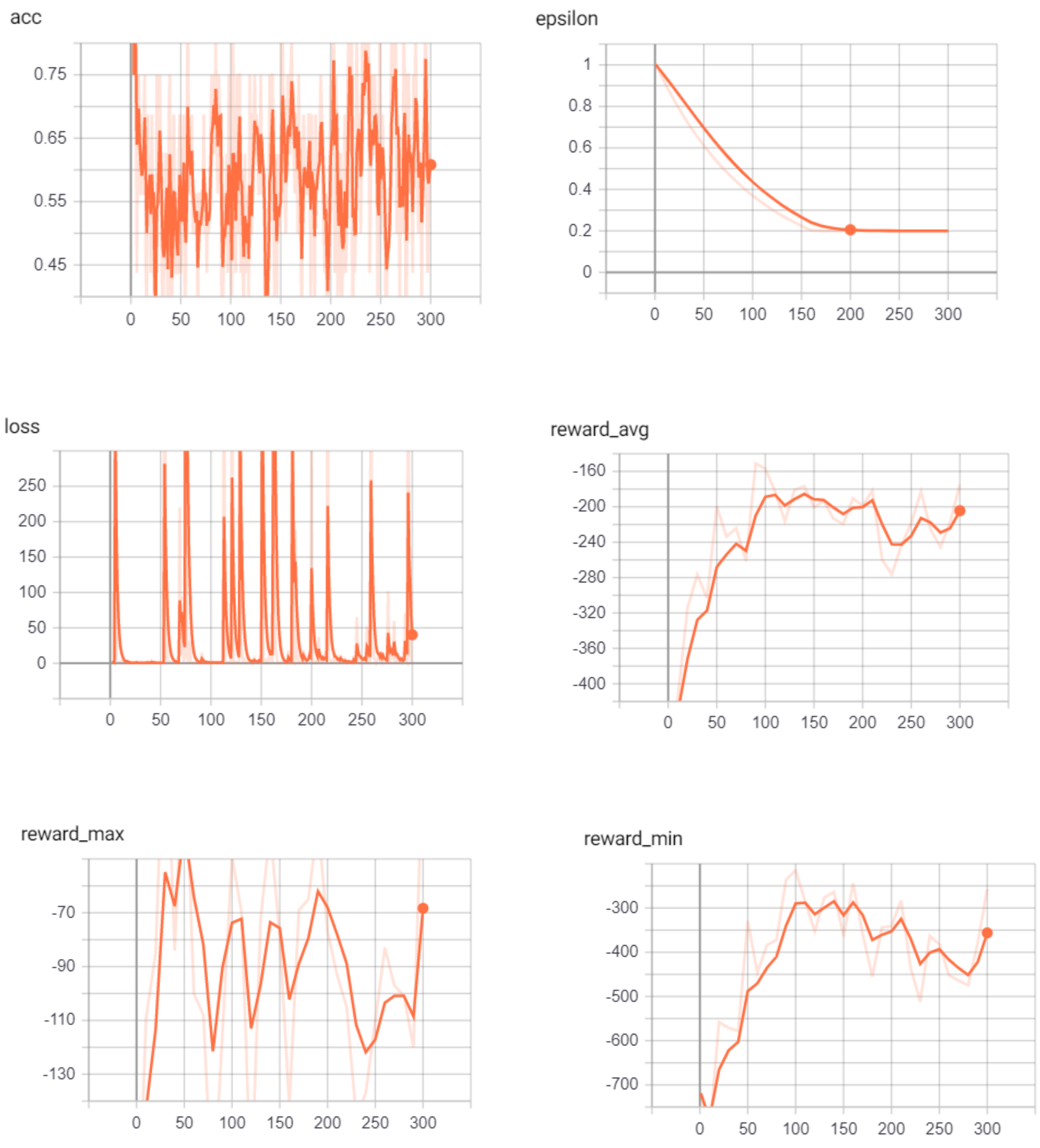


Figure 12: The new data of CNN model

Based on the training results of the project, an evaluation of the DQN algorithm can be made:

The DQN algorithm trains self-driving car models in Carla that can learn continuously in the environment. This means that the model can gain experience and gradually improve its driving strategy by interacting with the environment. Although the learning rate may not be fast, this is reasonable as self-driving cars need to deal with complex driving scenarios and decisions. DQN algorithms may require longer training times to converge to a better strategy in some complex tasks. This is due to the nature of the reinforcement learning problem, which requires a lot of interaction with the environment to gain experience and optimise the strategy. Therefore, this project iterates by saving the model after each training session and then loading this model in the next training round to continue training.

At the same time, the advantages of DQN are clear. It uses deep neural networks to process high-dimensional state spaces and can model and learn from complex environments, making them applicable to complex problems in the real world. It uses techniques such as experience replay and target networks to solve the problem of high correlation between samples in reinforcement learning, improving learning efficiency and stability. This is reflected in the fact that self-driving cars can be trained to accurately determine the optimal action as driving in a straight line along a road (similar to the desired outcome) in urban scenarios, and that the car model is doing its best to reduce the occurrence of collisions. The DQN reinforcement learning algorithm shows some potential and progress. In the future, new models and algorithms can be added to this project to perform more complex training to obtain better models.

5. Conclusion

This project successfully implemented the training of a self-driving vehicle model with a reinforcement learning based algorithm. Using Carla as an open-source simulator for the virtual engine, creating a simple urban traffic scenario, and training the car's planner with the DQN algorithm. The goal of the project is to observe the performance of the car in the map, evaluate the algorithms and models against the training data and modify them to improve the performance of the self-driving car.

During the project, participant learns the basics of Carla, including its components and syntax. The project selects existing maps in Carla and adds vehicle and pedestrian NPCs to simulate a realistic urban traffic environment. Additionally, it installs RGB and collision sensors for the training vehicle and train CNN and Xception models using the DQN algorithm respectively. In a comparison of the two models, it concludes that the CNN model is a better choice for training. Moreover, in the first training result, the car tends to perform one action and is difficult to avoid obstacles. Then, the project modifies some parameters in the algorithm, such as increasing the punishment value and raising the minimum epsilon of the model, continues the training using the CNN model. In the results of the second training session, the self-driving car has an increased chance of avoiding obstacles and can effectively balance the execution of high-reward actions with exploration, ensuring normal driving in most situations. Although this car is still far from matching a real driver, the overall progress has presented ongoing optimization efforts.

It should be clear that the goal of this project is not to train a perfect self-driving car, but to implement a process of training a model using the algorithm to improve its performance, and to obtain an understanding of the algorithm through continuous evaluation and exploration, so the current project has been somewhat successful. It is worth highlighting that the DQN algorithm is effective at training neural network models, and despite its relatively slow learning speed, its learning ability and stability are excellent. Therefore, the DQN algorithm can continue to be used for training in the future. In addition, a major limitation in this project is the limited memory of the GPU. Due to this limitation, the project is unable to use more complex algorithms to optimise the training of the model. However, this problem can be solved in the future by using a GPU with more memory, allowing for more complex algorithms to be employed. For example, an obstacle detector model can be added to define more complex actions for the car so that it can learn better in the environment. It is also possible to complicate the CNN model.

In summary, the DQN algorithm offers endless possibilities for self-driving cars to gradually approach the level of real driving. The success and experience of this project provides useful guidance for future research and development. Through continuous improvement and exploration, it is expected to further improve the performance of self-driving cars and make them better adapted to complex traffic environments and challenges.

6. BCS Project Criteria & Self-Reflection

- The project’s algorithm and development approach involve the application and evaluation of Python language and reinforcement learning algorithms, which will allow participants to apply the programming skills, algorithmic thinking, and data analysis skills gained from the degree programme.
- Creating a scenario in the simulator and training the self-driving car to learn and improve in this scenario requires constant modification and refinement, for example by adding new algorithms and trying out new parameters, which represents innovation to a certain extent.
- The project combines Carla simulator, DQN reinforcement learning algorithm and a self-driving car to successfully train the car's planner in a city map, allowing it to continuously learn and improve its performance in the environment, and to evaluate its performance.
- The DQN reinforcement learning algorithm shows some potential and progress with the self-driving car model trained in Carla. Although the learning rate may not be fast enough, the model is able to progressively improve its ability to avoid collisions and find a balance between exploration and exploitation to improve overall performance. This provides a promising basis for further optimisation and development of autonomous driving systems, and offers the prospect of safer, more efficient driving on the road. However, it

still needs to continue researching and improving algorithms to further improve learning speed and driving performance.

- **Critical Reflection**:

  During this project, I completed the work on time under the schedule. Firstly, I demonstrated a good learning ability, which was shown by how quickly I became familiar with the Carla simulator and the Carla python syntax. I could also learn how to install the software required to run Carla from the internet (e.g., configuring environment variables when installing CUDA). In addition, I gained knowledge of reinforcement learning, understanding the principles of the DQN algorithm and successfully applying it to training an intelligence. This once again gave me a shock of machine learning, which means although only some basic DQN algorithms were implemented, the Q-value iterations under the two networks allowed the model to show a good learning capability. Then, with the data obtained from Tensorboard, I compared the strengths and weaknesses of the two models, successfully identified the parts of the algorithm that needed to be modified and selected the better CNN model to continue training, which further improved my data analysis skills and deepened my understanding of the algorithm.

  At the same time, the project has some shortcomings. In relying on the use of existing reinforcement learning algorithms, all I have been able to achieve so far is to modify and improve on them, such as defining methods or modifying parameters, so there is a lack of creativity. Furthermore, training self-driving car models is a long-term process. This project has not spent enough time continuing to train the model after modifying the algorithm. Although the model has improved, the data show that there is still more room for improvement and more time should still be spent training it. Finally, the lack of hardware facilities also has a negative impact on the project. The GPU used in this project has a small amount of memory, when using more complex models there was a problem of insufficient memory due to the amount of computation. If the parameters were turned down, the learning efficiency of the model would be affected. In addition, the memory limitations made it difficult to increase the algorithm, and the car could only learn some basic actions, so the project requires some equipment support. Overall, however, the project achieves its initial goals and produces some good results.

7. References

[1] Carla, "First steps with CARLA", Carla official website. [Online]. Available: https://carla.readthedocs.io/en/latest/tuto_first_steps/

[2] Sentdex, "Machine Learning". [Online]. Available: https://pythonprogramming.net/machine-learning-tutorials/

[3] "What is reinforcement learning?", University of York. [Online]. Available: https://online.york.ac.uk/what-is-reinforcement-learning/

8. Appendices

8.1 Class "VehicleEvn" [1] [2]

```
class VehicleEvn:
    SHOW_CAMERA = SHOW_SCREEN  # Set whether to display the camera screen
    STEER_AMT = 1.0
    im_width = IM_WIDTH  # Setting the camera sensor image width
    im_height = IM_HEIGHT  # Setting the camera sensor image height
    front_camera = None
    actor_list = []
    collision_hist = []
    vehicles = []

    def __init__(self):
        self.client = carla.Client('localhost', 2000)
        self.client.set_timeout(10.0)
        self.world = self.client.load_world('Town05_Opt')
        # Add a vehicle for training
        self.vehicle_main =
self.world.get_blueprint_library().filter("vehicle.dodge_charger.police")[0]

        # Init NPC vehicles
        self.init_npc()

    # Add some NPC cars and pedestrians
    def init_npc(self):
        self.spawn_points = self.world.get_map().get_spawn_points()
        # Select some models from the blueprint library
        self.group = ['dodge', 'audi', 'tesla', 'mini', 'mustang', 'lincoln', 'prius', 'nissan', 'crown',
'impala',
                      'mercedes']
        self.blueprints = []
        for vehicle in self.world.get_blueprint_library().filter('*vehicle*'):
            if any(model in vehicle.id for model in self.group):
                self.blueprints.append(vehicle)

        # Set a max number of vehicles and prepare a list for those we spawn
        max_vehicles = 1
        max_vehicles = min([max_vehicles, len(self.spawn_points)])
        self.vehicles = []

        # Take a random sample of the spawn points and spawn some vehicles
        for i, spawn_point in enumerate(random.sample(self.spawn_points, max_vehicles)):
            temp = self.world.try_spawn_actor(random.choice(self.blueprints), spawn_point)
            if temp is not None:
                self.vehicles.append(temp)
        for vehicle in self.vehicles:
            vehicle.set_autopilot(True)

        # Set a max number of walkers and prepare a list for those we spawn
        max_walkers = 1
        max_walkers = min([max_walkers, len(self.spawn_points)])
        self.walkers = []
        self.controller = []
        # Spawn walkers
        for i, spawn_point in enumerate(random.sample(self.spawn_points, max_walkers)):
            pedestrian_bp =
random.choice(self.world.get_blueprint_library().filter('*walker.pedestrian*'))
            temp2 = self.world.try_spawn_actor(pedestrian_bp, spawn_point)
            if temp2 is not None:
                # Spawn walker controllers
                controller_bp = self.world.get_blueprint_library().find('controller.ai.walker')
                ai_controller = self.world.spawn_actor(controller_bp, carla.Transform(), temp2)
                self.controller.append(ai_controller)
                self.walkers.append(temp2)
        for people in self.controller:
            # Set up pedestrian navigation
            people.start()
            people.go_to_location(self.world.get_random_location_from_navigation())

    def set(self):
        self.actor_list = []
        self.collision_hist = []
        # Add the training vehicle
        self.spawn_points = self.world.get_map().get_spawn_points()
        self.vehicle = self.world.spawn_actor(self.vehicle_main,
random.choice(self.spawn_points))
        self.actor_list.append(self.vehicle)

        # Add spectator
        self.spectator = self.world.get_spectator()
        self.spectator_vehicle_offset = carla.Location(x=-3, z=3)  # offset behind the vehicle
        vehicle_transform = self.vehicle.get_transform()
        new_transform = carla.Transform(vehicle_transform.location +
self.spectator_vehicle_offset)
        self.spectator.set_transform(new_transform)

        # Defining the sensor - RGB camera
        self.rgb_cam = self.world.get_blueprint_library().find('sensor.camera.rgb')
        self.rgb_cam.set_attribute("image_size_x", f"{self.im_width}")
        self.rgb_cam.set_attribute("image_size_y", f"{self.im_height}")
        self.rgb_cam.set_attribute("fov", f"110")  # Setting up the front camera

        transform = carla.Transform(carla.Location(x=2.5, z=0.7))
        self.sensor = self.world.spawn_actor(self.rgb_cam, transform, attach_to=self.vehicle)
        self.actor_list.append(self.sensor)
        self.sensor.listen(lambda data: self.image_data(data))

        self.vehicle.apply_control(carla.VehicleControl(throttle=0.0, brake=0.0))
        time.sleep(4)

        # collision sensor
        collsensor = self.world.get_blueprint_library().find('sensor.other.collision')
        self.collsensor = self.world.spawn_actor(collsensor, transform, attach_to=self.vehicle)
        self.actor_list.append(self.collsensor)
        self.collsensor.listen(lambda event: self.collision_data(event))

        while self.front_camera is None:
            time.sleep(0.01)
        self.episode_start = time.time()
        self.vehicle.apply_control(carla.VehicleControl(throttle=0.0, brake=0.0))

        return self.front_camera

     # Callback functions
    def collision_data(self, event):
        self.collision_hist.append(event)

    # The function converts an image from a 1D array form to a 3D form
    def image_data(self, image):
        i = np.array(image.raw_data)
        i2 = i.reshape((self.im_height, self.im_width, 4))
        i3 = i2[:, :, : 3]
        if self.SHOW_CAMERA:
            cv2.imshow("", i3)
            cv2.waitKey(1)
        self.front_camera = i3

    def step(self, action):
        # Defining the movement of the car
        if action == 0:
            self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=-1 *
self.STEER_AMT))  # left
        elif action == 1:
            self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=0))
        elif action == 2:
            self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=1 *
self.STEER_AMT))   # right
        elif action == 3:
            self.vehicle.apply_control(carla.VehicleControl(throttle=0.5, steer=0))  # slow down
        elif action == 4:
            self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=0.5 *
self.STEER_AMT))  # slight right turn
        elif action == 5:
            self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=-0.5 *
self.STEER_AMT))  # slight left turn

        # Determining whether the brakes are needed
        if abs(self.vehicle.get_control().steer) > 0.5:
            self.vehicle.apply_control(carla.VehicleControl(throttle=0.8,
steer=self.vehicle.get_control().steer, brake=0.4))
        else:
            self.vehicle.apply_control(carla.VehicleControl(throttle=1.0,
steer=self.vehicle.get_control().steer, brake=0.0))



        v = self.vehicle.get_velocity()
        kmh = int(3.6 * math.sqrt(v.x**2 + v.y**2 + v.z**2)) # Converting the speed of a vehicle
from vector to scalar form
        if len(self.collision_hist) != 0:
            done = True
            reward = -200

        elif kmh < 50:
            done = False
            reward = -1
        else:
            done = False
            reward = 1

        if self.episode_start + SECOND_PER_EPISODE < time.time():
            done = True

        return self.front_camera, reward, done, None
```

8.2 Class "DQN" [2]

```
class DQN:
    def __init__(self):
        self.model = load_model('C:/Users/qsy/PycharmProjects/Carla/another/new.model')  #
create an evaluation network model
        self.target_model = self.create_model()  # create a target network model
        self.target_model.set_weights(self.model.get_weights())  # set target network's weights
using evaluation network's weights

        self.replay_memory = deque(maxlen=REPLAY_MEMORY_SIZE)  # double-ended queue
        self.tensorboard = ModifiedTensorBoard(log_dir=f"logs/{MODEL_NAME}-
{int(time.time())}")

        self.graph = tf.get_default_graph()
        self.terminate = False
        self.training_initialized = False
        self.last_logged_episode = 0
        self.target_counter = 0

   # This code defines a neural network model. It contains 3 convolutional layers, each
containing 64 filters, each of
    # size 3x3. A ReLU (Rectified Linear Unit) activation function is added after each
convolutional layer and an average pooling layer of
    # size 5x5 with a step size of 3x3.

    def create_model(self):
        model = Sequential()
        model.add(Conv2D(64, (3, 3), input_shape=(IM_HEIGHT, IM_WIDTH, 3),
padding='same'))
        model.add(Activation('relu'))
        model.add(AveragePooling2D(pool_size=(5, 5), strides=(3, 3), padding='same'))

        model.add(Conv2D(64, (3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(AveragePooling2D(pool_size=(5, 5), strides=(3, 3), padding='same'))

        model.add(Conv2D(64, (3, 3), padding='same'))
        model.add(Activation('relu'))
        model.add(AveragePooling2D(pool_size=(5, 5), strides=(3, 3), padding='same'))

        model.add(Flatten())
        model.add(Dense(512))  # A fully connected layer with 512 neurons and a ReLU
activation function is then added
        model.add(Activation('relu'))
        # a fully connected layer with 3 neurons and a linear activation function is added to
output the Q value.
        model.add(Dense(3, activation='linear'))
        # Uses the mean square error (MSE) as the loss function and is optimised using the
Adam optimiser.
        model.compile(loss="mse", optimizer=Adam(lr=0.001), metrics=['accuracy'])

        return model

    def update_replay_memory(self, transition):
        # transition = (current_state, action, reward, new_state, done)
        self.replay_memory.append(transition)


# Use deep RL algorithms to optimise the weights of the neural network to better predict
the Q
# value of each action, enabling the model to learn and make optimal action choices in
different environments

    def train(self):
        if len(self.replay_memory) < MIN_REPLAY_MEMORY_SIZE:  # Check if the memory
meets the minimum size requirement, if not then do not train
            return
        minibatch = random.sample(self.replay_memory, MINIBATCH_SIZE)  # Random
sampling of a batch of data from a memory

        # Use the current(evaluation) model and the target model to predict the Q value of the
current state and the next state respectively
        # The current state is converted to a numpy array and normalised. Then, the current
state is predicted using--
        # --the current model (self.model) and the predicted Q values are stored in the
current_qs_list
        current_states = np.array([transition[0] for transition in minibatch]) / 255
        with self.graph.as_default():
            current_qs_list = self.model.predict(current_states, PREDICTION_BATCH_SIZE)
        future_states = np.array([transition[3] for transition in minibatch]) / 255
        with self.graph.as_default():
            future_qs_list = self.target_model.predict(future_states, PREDICTION_BATCH_SIZE)

        Status = []
        q_value = []
        # Q-value update formula
        for index, (current_state, action, reward, new_state, done) in enumerate(minibatch):
            if not done:
                max_future_q = np.max(future_qs_list[index])
                new_q = reward + DISCOUNT * max_future_q  # Calculate new Q value
            else:
                new_q = reward

            current_qs = current_qs_list[index]
            current_qs[action] = new_q   # update Q value of the action of the current state
            Status.append(current_state)
            q_value.append(current_qs)

        # Determine if the current training information needs to be recorded by checking if the
current training step count is
        # greater than the last recorded step count
        log_this_step = False
        if self.tensorboard.step > self.last_logged_episode:
            log_this_step = True
            self.last_logged_episode = self.tensorboard.step
        # Train the model
        with self.graph.as_default():
                # input/output/the number of samples used in each training session/not output
the log of the training process/not disrupt the order of the training data
            self.model.fit(np.array(Status)/255, np.array(q_value),
batch_size=TRAINING_BATCH_SIZE, verbose=0, shuffle=False, callbacks=[self.tensorboard] if
log_this_step else None)

        if log_this_step:
            self.target_counter += 1

        # Ensure that the target model is updated regularly to track improvements to the
current model
        if self.target_counter > UPDATE_TARGET_EVERY:
            self.target_model.set_weights(self.model.get_weights())
            self.target_counter = 0

# Continuous training of DQN models
    def train_in_loop(self):
        # Generate a random input x and output y for initialising the model
        x = np.random.uniform(size=(1, IM_HEIGHT, IM_WIDTH, 3)).astype(np.float32)
        y = np.random.uniform(size=(1, 3)).astype(np.float32)
        with self.graph.as_default():
            self.model.fit(x, y, verbose=False, batch_size=1)

        self.training_initialized = True
        while True:
            if self.terminate:
                return
            self.train()
            time.sleep(0.01)

# Get the Q value of the current state (state)
    def get_qs(self, state):
        return self.model.predict(np.array(state).reshape(-1, *state.shape)/255)[0]
```

8.3 Class "ModifiedTensorBoard" [2]

```
class ModifiedTensorBoard(TensorBoard):

    # Overriding init to set initial step and writer (we want one log file for all .fit() calls)
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.step = 1
        self.writer = tf.summary.FileWriter(self.log_dir)

    # Overriding this method to stop creating default log writer
    def set_model(self, model):
        pass

    # Overrided, saves logs with our step number
    # (otherwise every .fit() will start writing from 0th step)
    def on_epoch_end(self, epoch, logs=None):
        self.update_stats(**logs)

    # Overrided
    # We train for one batch only, no need to save anything at epoch end
    def on_batch_end(self, batch, logs=None):
        pass

    # Overrided, so won't close writer
    def on_train_end(self, _):
        pass

    # Custom method for saving own metrics
    # Creates writer, writes custom metrics and closes writer
    def update_stats(self, **stats):
        self._write_logs(stats, self.step)
```

8.4 Main method [2]

```
if __name__ == '__main__':
   # Percentage of memory allocated to GPU
   gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=MEMORY_FRACTION)
   backend.set_session(tf.Session(config=tf.ConfigProto(gpu_options=gpu_options)))

   FPS = 60
   ep_rewards = [-200]
   random.seed(1)
   np.random.seed(1)
   tf.set_random_seed(1)
   if not os.path.isdir('another'):
      os.makedirs('another')
   agent = DQN()
   env = VehicleEvn()

   trainer_thread = Thread(target=agent.train_in_loop, daemon=True) # The training loop
for the model is started in a separate thread
   trainer_thread.start()
   while not agent.training_initialized:
      time.sleep(0.01)

   agent.get_qs(np.ones((env.im_height, env.im_width, 3)))

   # tqdm--Show progress bar--The program starts iterating over each episode
   for episode in tqdm(range(1, EPISODES + 1), ascii=True, unit="episodes"):
      env.collision_hist = []  # Clear the collision history before each episode starts
      agent.tensorboard.step = episode  # Set the number of steps in the tensorboard to the
current number of episodes
      episode_reward = 0
      step = 1
      current_state = env.set()
      done = False
      episode_start = time.time()

      while True:
# Take an action in the current state, if the random number is greater than epsilon, take the
optimal action predicted
# by the model, otherwise choose the action randomly
         if np.random.random() > epsilon:
            action = np.argmax(agent.get_qs(current_state))
         else:
            action = np.random.randint(0, 3)
            time.sleep(1/FPS)
            new_state, reward, done, _ = env.step(action)
            episode_reward += reward
            agent.update_replay_memory((current_state, action, reward, new_state, done)) #
Store the current experience in the experience replay
            current_state = new_state
            step += 1

            if done:
                break

        for actor in env.actor_list:
            actor.destroy()


        # Append episode reward to a list and log stats(every given number of episodes)
        ep_rewards.append(episode_reward)
        if not episode % AGGREGATE_STATS_EVERY or episode == 1:
            average_reward = sum(ep_rewards[-AGGREGATE_STATS_EVERY:]) / len(ep_rewards[-
AGGREGATE_STATS_EVERY:])
            min_reward = min(ep_rewards[-AGGREGATE_STATS_EVERY:])
            max_reward = max(ep_rewards[-AGGREGATE_STATS_EVERY:])
            agent.tensorboard.update_stats(reward_avg=average_reward,
reward_min=min_reward, reward_max=max_reward, epsilon=epsilon)

            # Save model, but only when min reward is greater or equal a set value
            if min_reward >= MIN_REWARD:
                agent.model.save(f'another/{MODEL_NAME}.model')

        # Decay epsilon makes the intelligences gradually explore less and less during training
and gradually move to better actions
        if epsilon > MIN_EPSILON:
            epsilon *= EPSILON_DECAY
            epsilon = max(MIN_EPSILON, epsilon)

    # Set termination
    agent.terminate = True
    trainer_thread.join()
    agent.model.save(f'another/{MODEL_NAME}.model')
```