Training a Reinforcement Learning Agent to Drive a Car Autonomously in CARLA Simulator

Ricardo Zambrano
University of Maryland
rzambrano@gmail.com

Abstract

This term project sought to develop a self-driving agent to operate in CARLA simulation by using deep q-learning. It implemented an approach to follow the self-driving car industry trends and simulate how humans perceive the environment when driving. Thereby, only sensor cameras were used and the input to the deep q-network agent (the state observation) were the image signal and velocity information. Because of memory and computation limitations of the system in which the agent was trained only grayscale images were used as an input to the neural network and pretrained weights were not used. It was found that it took just a few epochs to get the agent to reach its performance peak. After this point the agent's performance started to decline progressively. This might be an indication that the learning process went of track, in which case other learning algorithms may be better suited for the self-driving problem, for example PPO. It was observed that more cameras needed to be attached to the vehicle, as this would allow the vehicle to perceive objects approaching from behind and from the sides.

I. Introduction

The purpose of this project was applying deep reinforcement learning as well as convolutional neural networks architectures to the problem of self-driving vehicles.

In the exploratory phase of this project using a scaled vehicle was considered (i.e. AWS Deep Racer). However, this idea was discarded because of the following reasons:

- The cost of the hardware (above $399)
- The potential hidden costs of running the training in the AWS platform
- The challenge of debugging hardware while also debugging the software
- Most importantly, the AWS platform would provide off-the-shelf algorithms for the deep reinforcement learning algorithms. It was considered that using these algorithms as a black-box would not be conducive to learning about how to implement deep reinforcement learning algorithms.

After discarding the approach of working with hardware, the option taken was working with a simulation platform for experimenting with self-driving vehicles. The selected platform was CARLA. The versions of the simulation used were v0.9.14 and v0.9.15. The later was launched recently and proved more stable than the previous version.

CARLA's environment provided a python API as well as a ROS2 bridge, this made it an excellent candidate to run deep reinforcement learning algorithms. The system available for running experiments was able to run ROS2 through the Windows Linux Subsystem. However, CARLA's engine, Unreal Engine, was not running under the WLS. Thereby, the simulation had to be run in Windows, which meant access to the simulation only through the python API.

A. Goal of the Agent

The proposed goal was to develop a self-driving agent to safely drive from a set of starting points to target destinations within CARLA simulation. A set of sub-goals during the course of the drive was to train the agent to respect traffic rules, change lanes in areas with traffic, go through roundabouts, respond to sudden movements made by other vehicles, and avoid hitting pedestrians.

Given the hardware limitations the goal of the agent had to be re-stated. For instance, within CARLA all vehicles that are part of the environment are managed by the Traffic Manager. Meanwhile, each pedestrian needs an 'controller.ai.walker' to be spawn and attached to the walker in order to be animated. This generates an overhead that would be using resources needed for running the simulation while running the training loop. Because of this reason pedestrian were not included in the environment.

Likewise, observing traffic lights would require the use of pre-trained weights and full-color images. During the first trials of the training algorithm the system ran out of memory quickly and thus the algorithm never progressed far. Because of this limitation the approach was changed from using RGB images generated by the camera sensor to using a gray-scale images. This meant processing a stack of four images with one channel instead of four images with three channels each.

Because of the aforementioned limitations, it was decided to narrow the scope of the agent to just driving around without running over obstacles. The agent would ignore traffic signs, including sticking to the correct lane.

B. Sensor Selection

An early goal of this project was to use camera sensors only. It seems that this is the trend in the mainstream industry of self-driving car makers.

CARLA has advanced sensors available, such as: lane invasion sensors, LIDAR, obstacle detector, RADAR, and Global Navigation Satellite System (GNSS). It is hypothesized that using these sensors' signal would make training the agent easier. In some cases, some of these signals could be used in parallel with the agent to improve the agent's driving, instead of using the signals as an input to the agent's neural network.

Notwithstanding, at the time of this writing most of these advanced sensors are either expensive or cumbersome to implement in a real-world environment. Because of thee two reasons the use of advanced sensors was discarded from the beginning.

II. Literature Survey

The goal of this project was to learn to implement known machine learning algorithms and architectures to the problem of self-driving cars. This project was not attempting to generate new knowledge. Thereby, the literature review was oriented towards understanding the implementation details of four reinforcement learning algorithms:

- Deep Q-Learning Network (DQN)
- Deep Deterministic Policy Gradient (DDPG)
- Soft Actor Critic (SAC)
- Proximal Policy Optimization (PPO)

The expectation set in the proposal was to use at least one of the aforementioned reinforcement-learning algorithms to train the self-driving agent.

Because of the time limitations characteristic of a term project there was time to design, implement, and run only one of the proposed reinforcement learning algorithms. Among the proposed options, the sensible selection as a starting point was the Deep Q-Learning algorithm. This is an algorithm that has been tried and tested many times. Because of the lineage of this algorithm, there are a lot of references available about how to design and train agents using this algorithm.

III. Methodology

A. Deep Q-Learning (DQN)

Deep-Q Learning is an extension of Q-Learning. Like the classic Q-Learning, this is a value-based method. The latter means the goal is to learn a value function that maps a state to the expected value of being at that state. It is worthwhile emphasizing that in a value-based method we learn/approximate the optimal policy π^* <u>indirectly</u>, by training a value function that outputs the value of a state or a state-action pair. Given this value function, the policy will take an action.

In this family of reinforcement learning algorithms there is no policy function available after training. However, after training there is a function that outputs the value of every state action pair, if the goal were to choose the action leading to the biggest reward, it would be possible to obtain a policy by encoding a 'greedy' behavior. In this way, actions can be chosen and the agent would have -in practice- a policy.

Therefore, for value-based methods, there is no need to train the policy. In the case of the classic Q-Learning algorithms, the policy is a simple pre-specified function (i.e. a Greedy Policy, as state above), that uses the values given by the value-function to select its actions. The following equation presents formally the link between the value function and the optimal policy approximation in the context of classic Q-Learning:

$$\tilde{\pi}^{*} = argmax_{a}[Q^{*}{}_{\pi}\,(s,a)]$$

$Q_{\pi}(s,a)$ stands for the value-action function. For each state-action pair, this function outputs the expected return if the agent starts in that state, takes a given action, and then follows the policy forever after. Formally;

$$Q_{\pi}(s,a) = E_{\pi}[G_t | S_t = s, A_t = a]$$

Where, G_t is the total cumulative reward from timestep t.

B. Exploration Versus Exploitation

One of the challenges that arise in reinforcement learning, and not in other kinds of learning, is the trade-off between exploration and exploitation. To obtain a lot of reward, a reinforcement learning agent must prefer actions that it has tried in the past and found to be effective in producing reward. But to discover such actions, it has to try actions that it has not selected before. The agent has to exploit what it has already experienced in order to obtain reward, but it also has to explore in order to make better action selections in the future (Barto, 2018).

In order to address the exploration and exploitation problem, during training actions are selected using an epsilon-greedy approach. Using this approach, the select_action() function chooses an action using the current policy with probability (1 – Є) and a random action from the action space with probability Є.

At the beginning of the training loop the value of Є is high. Then as the training progresses the value of epsilon decays towards in final value at a rate of $e^{-(\text{training\_steps} / \text{decay\_factor})}$. In practice, at the beginning of the algorithm exploration is promoted by having a higher chance of selecting a random action. Towards the end of the training exploitation is promoted by selecting actions from the policy.

C. Policy and Target Networks

When implementing a deep q-network two neural networks need to be instantiated: a policy network and a target network. The target network is identical to the policy network at the beginning. During training the target network has either fixed weights for a fixed number of training steps - after which it gets its weights updated from the policy network, or, gets its weights updated using a soft update at every training step. The soft update is controlled by the value of tau (τ). A larger value of tau translated in a harder update of the target network.

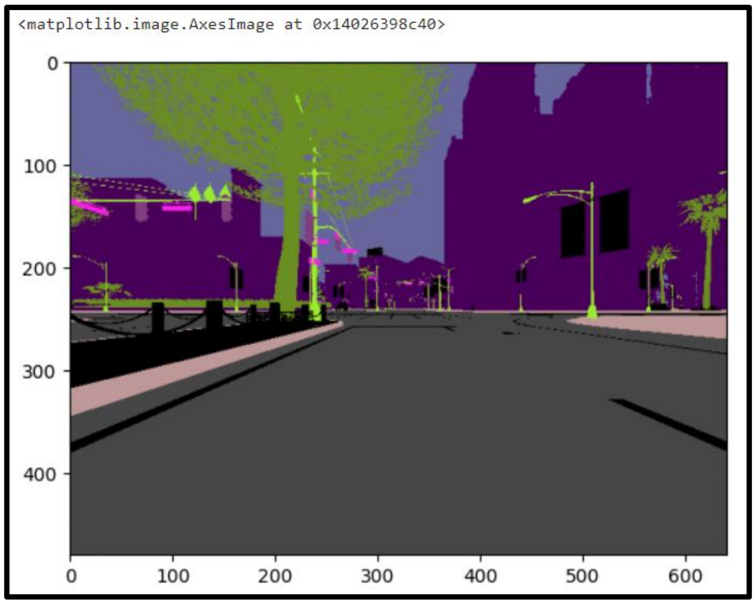
In this project a soft update of the target network was used.

D. Observations

The signals chosen from the environment to select actions were:

- Images from a camera sensor. Three camera sensors were setup: an RGB camera, a depth camera, and a semantic segmentation camera. Only the RGB and the semantic sensor camera were used in the training trials
- Magnitude of the speed of the vehicle
- Angular velocity of the vehicle

The rationale behind this selection was that humans rely of vision and speed perception to chose actions when driving a vehicle. Humas also have a model of the world that provides information about the objects in the surroundings of the vehicle, this is the reason the semantic segmentation camera was chosen as a signal candidate. CARLA's "semantic segmentation" camera classifies every object in the view by displaying it in a different color according to the object class. E.g., pedestrians appear in a different color than vehicles (see figure 1).



```
<matplotlib.image.AxesImage at 0x14026398c40>
```

Figure 1- View of the semantic segmentation camera

Another reason the magnitude of the vehicle's speed was chosen was that the reward function was proportional to the speed of the vehicle.

At this point it is important to note that given the hardware limitation only one camera sensor could be used at a time. This meant the agent had neither rear-view mirror nor side mirror. This situation put the agent into a clear disadvantage compared to a human driver or a commercial self-driving car (Tesla features nine cameras).

The computation limitation also made impractical to pass the image data through a convolutional base of a pre-trained network (such as the Xception). Using the pre-trained weights in combination with a fine-tuned head layer can provide a head start to a newly trained network.

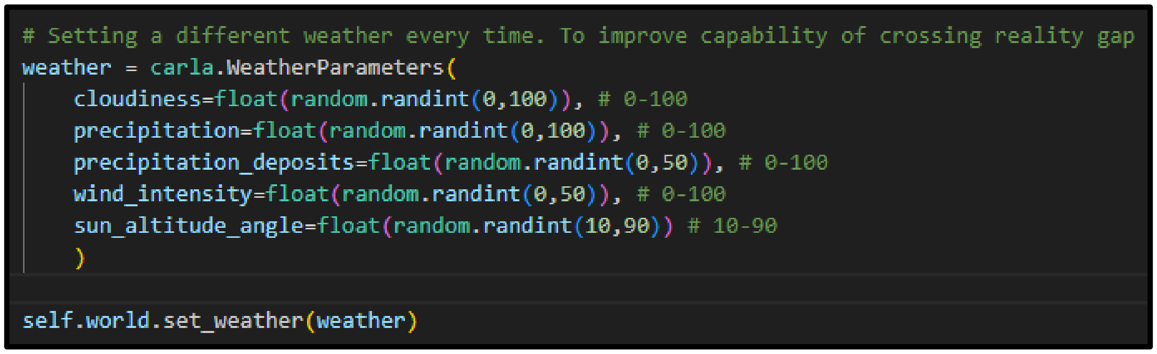
E. Temporal Limitation Problem

As mentioned above the agent processes images, as frames from a given camera sensor. Providing one frame at a time will lead to a problem known as “temporal limitation”, which is that one frame does not have enough information from the environment. In particular, from a single image the agent neither can infer the direction of movement of the vehicle nor assess the direction of movement of other moving objects present in the environment. Thereby, in order to get enough information from the environment a stack of frames may be passed to the reinforcement learning algorithm.

F. Crossing the Reality Gap Considerations

Although the goal of the agent was simplified in a way that the agent would work only in a race track and not in the streets of any city, some considerations were made to make the agent rely less on implementation aspects of the environment.

In particular, CARLA’s weather feature was leveraged to change the driving conditions the agent was facing. Figure 2 showcases the weather conditions that were chosen randomly at the beginning of every episode. Weather conditions also introduced noise in the camera sensors.



```
# Setting a different weather every time. To improve capability of crossing reality gap
weather = carla.WeatherParameters(
    cloudiness=float(random.randint(0,100)), # 0-100
    precipitation=float(random.randint(0,100)), # 0-100
    precipitation_deposits=float(random.randint(0,50)), # 0-100
    wind_intensity=float(random.randint(0,50)), # 0-100
    sun_altitude_angle=float(random.randint(10,90)) # 10-90
    )

self.world.set_weather(weather)
```

Figure 2- By changing the weather, driving conditions were different in each episode

G. System Requirements

The simulation environment used was CARLA. This environment has the following requirements:

- CARLA 0.9.15 built for Windows systems (version 0.9.14 was used but the engine crashed frequently).
- Python. Python is the main scripting language in CARLA. CARLA supports Python 2.7 and Python 3 on Linux, and Python 3 on Windows. This project used python 3.8.18
- Pip. Some installation methods of the CARLA client library require pip or pip3 (depending on your Python version) version 20.3 or higher.

- PyTorch 2.1.1 with access to CUDA 12.1

The simulation and learning was run in the following system:

- Operating System: Windows 11 Home
- Processor: 12th Gen Intel(R) Core(TM) i9-12900F 2.40 GHz
- RAM: 32.0 GB (31.8 GB usable)
- GPU: NVIDIA GeForce RTX 3080

IV. Simulation

A. Description of Simulation Environment

CARLA is an open-source autonomous driving simulator. It was built from scratch to serve as a modular and flexible API to address a range of tasks involved in the problem of autonomous driving. The simulator meets the requirements of different use cases within the general problem of driving (e.g. learning driving policies, training perception algorithms, etc.). CARLA is grounded on Unreal Engine to run the simulation and uses the OpenDRIVE standard (1.4 as today) to define roads and urban settings. Control over the simulation is granted through an API handled in Python and C++.

The CARLA simulator consists of a scalable client-server architecture. The server is responsible for everything related with the simulation itself: sensor rendering, computation of physics, updates on the world-state and its actors. CARLAS's documentations recommends running the server with a dedicated GPU, especially when dealing with machine learning. In this project, a GPU was used to both run the simulation as well as the training loop.

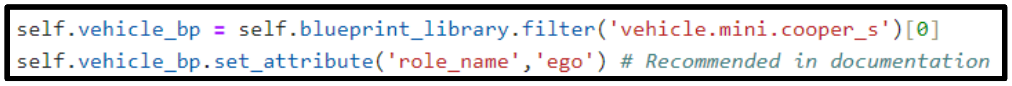
The client side consists of a sum of client modules controlling the logic of actors on scene and setting world conditions. This is achieved by leveraging the CARLA API (in Python or C++), a layer that mediates between server and client that is constantly evolving to provide new functionalities.

Within the context of CARLA's environment the project the focus of the agent is to control a special type of actor called carla.Vehicle, which incorporates special internal components that simulate the physics of wheeled vehicles.

B. Data, and Sensors

The agent will focus in controlling the special actor named 'ego vehicle'. This actor is the vehicle of focus in the simulation environment. Sensors are attached to de 'ego vehicle' to get an observation from the simulation.

The procedure to set the ‘ego vehicle’ consist is setting an attribute to the vehicle’s blue print. The procedure is depicted in the following sniped of code:

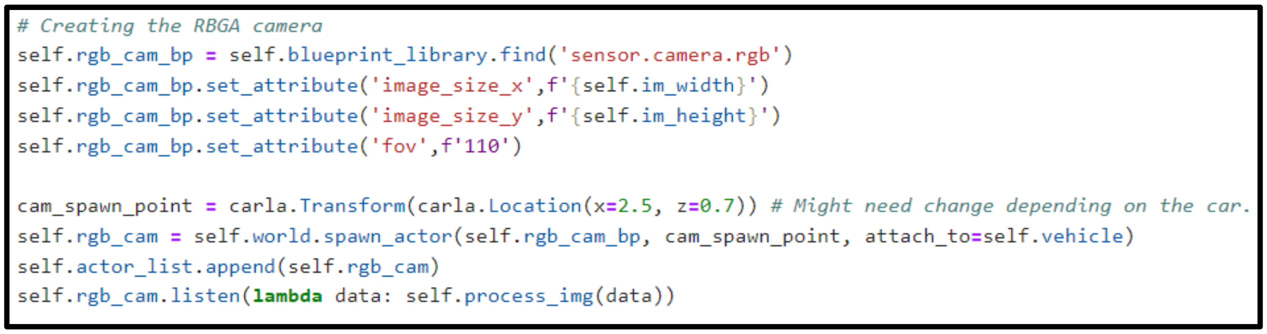


```
self.vehicle_bp = self.blueprint_library.filter('vehicle.mini.cooper_s')[0]
self.vehicle_bp.set_attribute('role_name','ego') # Recommended in documentation
```

Attaching sensors to the ego car requires:

1. Finding the sensor’s blueprint in the blueprint library
2. Set the size attributes for the images generated by the sensor (if the sensor produces imagery). For this project the generated images will have a size of 480x640 pixels
3. If the sensor generates images the next step is setting the ‘field of view’ (fov) attribute of the camera
4. Spawning the sensor and attaching it to the ego vehicle. The spawn position is relative to the center of the vehicle, thereby if the goal is to place the sensor at the front of the hood of the car it is required an additional parameter: carla.Location()
5. As with any other actor the sensors need to be appended to the ‘actor_list’. This step is required to keep track of all spawned actors. Tracking actors is useful for destroying then at the end of a simulation. This last point proven challenging as it will be described in subsequent sections
6. Finally, to start gathering the generated data the listen() method is passed to the sensor. Inside the listen() method a lambda function is passed to process the data into a desired format

The following snippet of code showcases the procedure to spawn an RGB camera:



```
# Creating the RBGA camera
self.rgb_cam_bp = self.blueprint_library.find('sensor.camera.rgb')
self.rgb_cam_bp.set_attribute('image_size_x',f'{self.im_width}')
self.rgb_cam_bp.set_attribute('image_size_y',f'{self.im_height}')
self.rgb_cam_bp.set_attribute('fov',f'110')

cam_spawn_point = carla.Transform(carla.Location(x=2.5, z=0.7)) # Might need change depending on the car.
self.rgb_cam = self.world.spawn_actor(self.rgb_cam_bp, cam_spawn_point, attach_to=self.vehicle)
self.actor_list.append(self.rgb_cam)
self.rgb_cam.listen(lambda data: self.process_img(data))
```

Thinking about the data required to make decisions while driving the following sensors were installed:

1. A collision sensor: this will be used for the reward function. The main goal of the ego car is driving around while avoiding hitting an object. This sensor is required to train the car in achieving its goal. In particular, when this sensor is activated, the episode ends. This sensor also triggers a penalty into the reward function
2. An RGB camera: this is an intuitive data source given that human drivers can see the road. The expectation is using this data as a surrogate for human’s capacity to see their surroundings

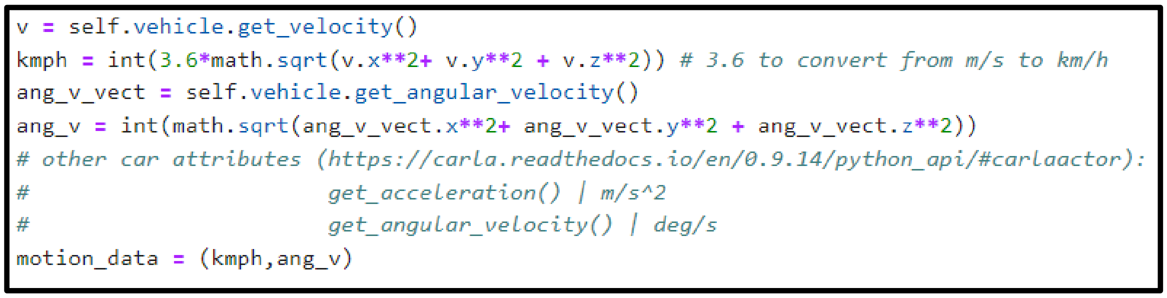
3. A depth camera: one capability humans have is to perceive depth. This signal might improve chances of the training agent to drive without hitting objects. However, due to time limitations this sensor was not using during these trials of training
4. A semantic segmentation camera: this camera returns an image of the objects in the field on view of the camera with each object colored with a distinct color. The idea is to imitate human's ability to know where each object begins and where it ends

The following processing functions were created to handle the generated data:

- def on_collision(): to extract the frame when the collision occurred, the intensity of the collision, and the actor that has hit by the ego vehicle
- def process_img(): to process CARLA's RGBA format (the Alpha values are discarded) and normalize the values. The normalization is recommended for input data used in deep learning
- process_depth(): to convert the depth camera data into a logarithmic depth map
- process_segmentation(): to convert data from the semantic segmentation camera into a segmentation map

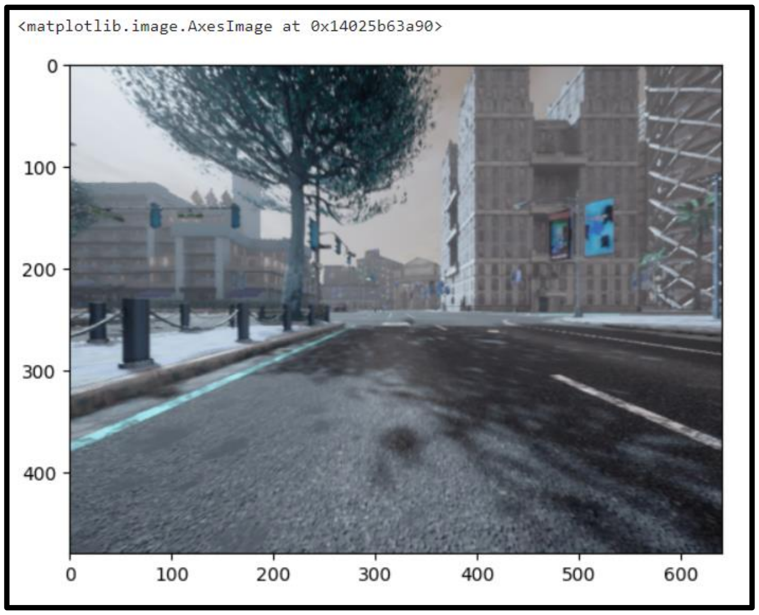
Some of the aforementioned processing functions rely on intermediate helper functions.

Finally, as mentioned before, it was hypothesized that the other two pieces of data humans use to make decisions while driving are the speed and angular velocity of the vehicle. These two measures were extracted from the vehicle. The following snipped of code showcases the methods used to extract this data:



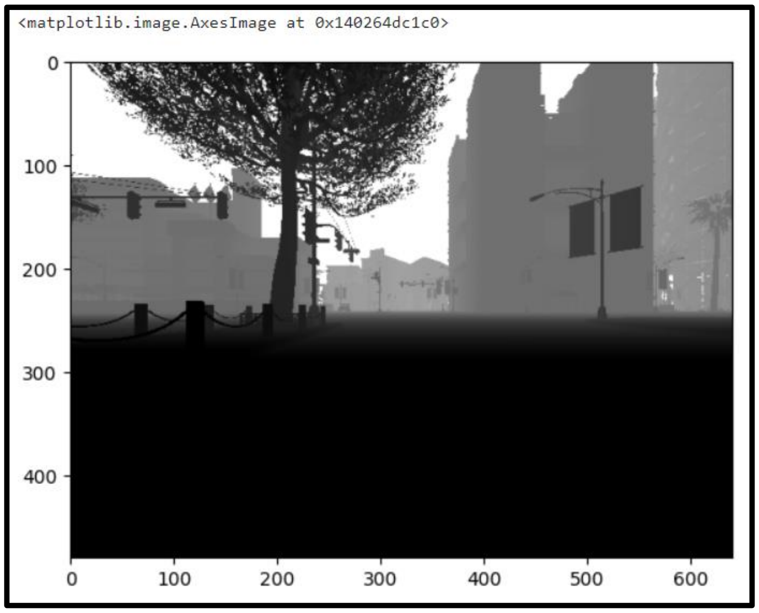
```
v = self.vehicle.get_velocity()
kmph = int(3.6*math.sqrt(v.x**2+ v.y**2 + v.z**2)) # 3.6 to convert from m/s to km/h
ang_v_vect = self.vehicle.get_angular_velocity()
ang_v = int(math.sqrt(ang_v_vect.x**2+ ang_v_vect.y**2 + ang_v_vect.z**2))
# other car attributes (https://carla.readthedocs.io/en/0.9.14/python_api/#carlaactor):
#                        get_acceleration() | m/s^2
#                        get_angular_velocity() | deg/s
motion_data = (kmph,ang_v)
```

The following images showcase the image data collected from the camera sensors (see Figure 1 for the semantic segmentation camera):



```
<matplotlib.image.AxesImage at 0x14025b63a90>
```

Figure 3- View of RGB camera



```
<matplotlib.image.AxesImage at 0x140264dc1c0>
```

Figure 4- View of depth camera

C. Environment Suitable for Reinforcement Learning on top of the Simulation

To perform reinforcement learning it is required an environment suitable for reinforcement learning. That is an environment aligned with the reinforcement learning process.

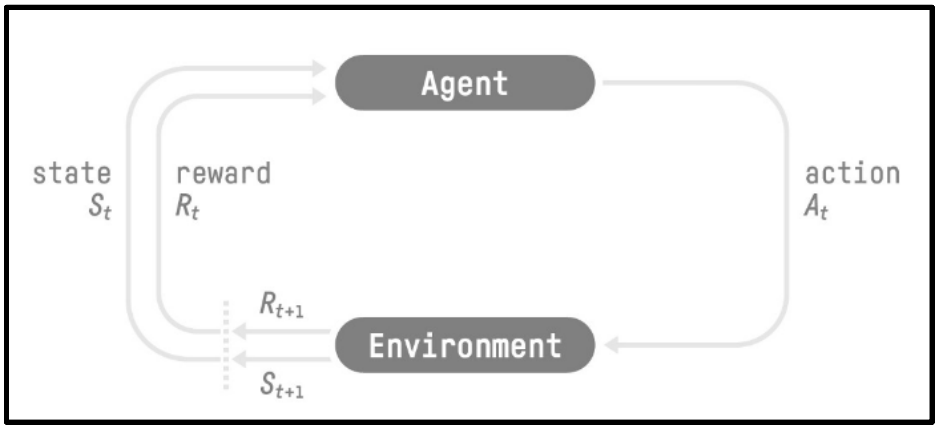


Figure 5- Reinforcement Learning Process

The reinforcement learning framework requires an environment that:

- Registers the beginning of an episode
- Receives an action at each step of the episode
- Advances the episode by a step given the action
- Returns a reward and an observation generated as a consequence of the action
- Terminates the episode when certain conditions are met

While programming the environment, the action space was generated as well. The simulation provides an infinite action space. However, to simplify the training process, a discrete set of actions was generated for the environment. Originally, it included the following legal actions:

0: "Full left - Full Throttle",
1: "Left - Full Throttle",
2: "Left - Half Throttle",
3: "Soft Left - Half Throttle",
4: "Straight - Full Throttle",
5: "Straight - Half Throttle",
6: "Soft Right - Half Throttle",
7: "Right - Half Throttle",
8: "Right - Full Throttle",
9: "Full Right - Full Throttle",
10: "Full Brake - Straight",
11: "Full Brake - Left",
12: "Full Brake - Right"

There was a reserved action 13: “Reverse – Full Throttle”. This action was not active in these set of trials. The reason being avoiding the vehicle to learn to go forward and backward in a long street to avoid hitting objects while accumulating rewards.

The environment generates ten vehicles that drive around the scenario under the directions of the traffic manager. This slightly approximated the simulation to race-track driving conditions. A portion of these vehicles ignored traffic signs (simulating bad drivers). Walkers were not generated at this time because each one would need an individual 'controller.ai.walker' actor attached to it to direct them around the scenario. It is believed that generating many actors will take too much resources making difficult the memory-consuming training process.

Each episode would terminate when either a collision occurs or after certain amount of time. For the moment each episode was limited to 60 seconds.

The following were methods and functions used to control the environment:

- CarlaEnv() a class to instantiate the environment. It loosely follows the template of Open AI’s gym environments
- manual_episode_end(): applies the destroy() method to the ego vehicle, the NPC vehicles, and the sensors
- reset() to start a new episode. It generates a new ego vehicle, the NPC vehicles, sets the weather, spawn the sensors, initiates the variables to store sensor data, and selects the sensor to be used by the training algorithm. This method returns the initial observation, the reward, and a Boolean variable indicating whether the episode ended
- Equally important, the CarlaEnv() class has all the methods to preprocess the sensor data. At the environment’s level it returns numpy data types
- step() contains the action space and the reward function. The method receives an action and, on each step, it returns an observation, the reward of the step, and a Boolean variable indicating whether the episode ended

Finally, the environment has the ability to display the RBG camera view using OpenCV. This feature was used only for the demo of the agent.

V. Implementation

A. Deep Q-Network Architecture

For this project a simple architecture was selected for the deep q-learning agent. If consisted in:

- 3 two-dimensional convolutional layers with Relu actiovation
- 1 layer to flatten the output o the convolutional layer block
- A step to concatenate the flattened output with the motion data vector
- 1 linear layer with Relu activation with an output of 128 neurons
- 1 linear layer with output size equal to number of actions

See Figure 6 for a graphical description of the agent's architecture. Figure 7 showcases the neural network implementation in PyTorch.

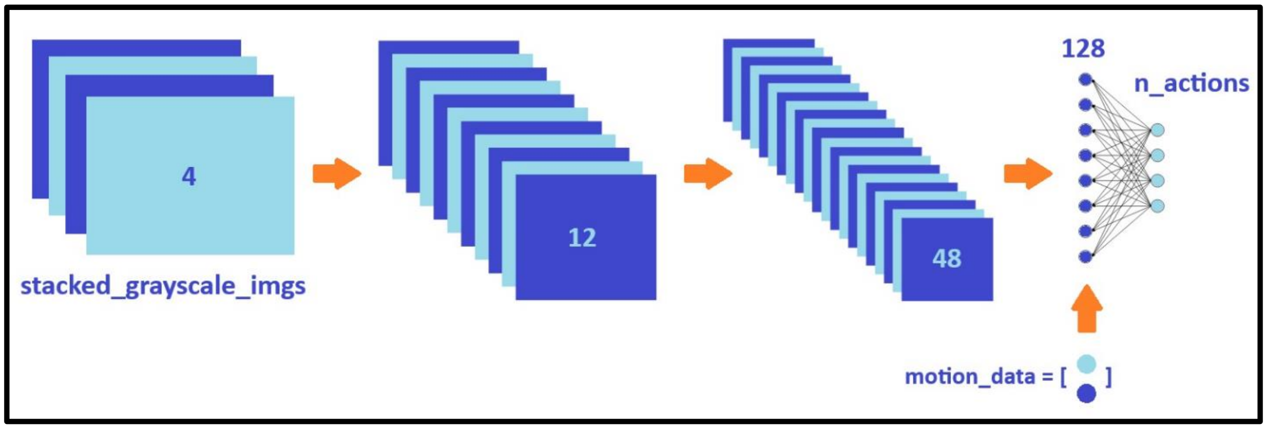
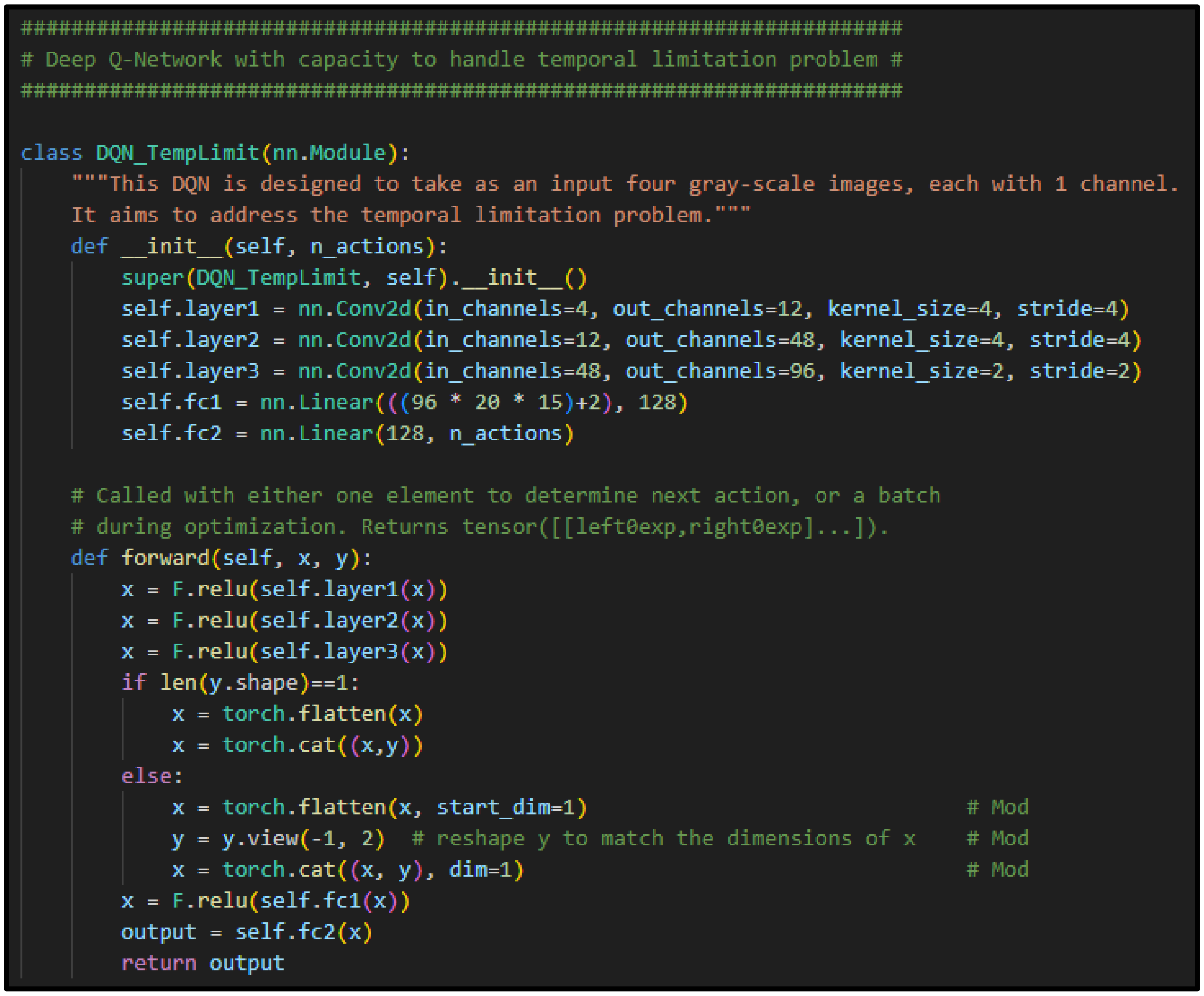


Figure 6- Deep Q-Learning Agent - Network Architecture



```
######################################################################
# Deep Q-Network with capacity to handle temporal limitation problem #
######################################################################

class DQN_TempLimit(nn.Module):
    """This DQN is designed to take as an input four gray-scale images, each with 1 channel.
    It aims to address the temporal limitation problem."""
    def __init__(self, n_actions):
        super(DQN_TempLimit, self).__init__()
        self.layer1 = nn.Conv2d(in_channels=4, out_channels=12, kernel_size=4, stride=4)
        self.layer2 = nn.Conv2d(in_channels=12, out_channels=48, kernel_size=4, stride=4)
        self.layer3 = nn.Conv2d(in_channels=48, out_channels=96, kernel_size=2, stride=2)
        self.fc1 = nn.Linear(((96 * 20 * 15)+2), 128)
        self.fc2 = nn.Linear(128, n_actions)

    # Called with either one element to determine next action, or a batch
    # during optimization. Returns tensor([[left0exp,right0exp]...]).
    def forward(self, x, y):
        x = F.relu(self.layer1(x))
        x = F.relu(self.layer2(x))
        x = F.relu(self.layer3(x))
        if len(y.shape)==1:
            x = torch.flatten(x)
            x = torch.cat((x,y))
        else:
            x = torch.flatten(x, start_dim=1)                          # Mod
            y = y.view(-1, 2)  # reshape y to match the dimensions of x     # Mod
            x = torch.cat((x, y), dim=1)                               # Mod
        x = F.relu(self.fc1(x))
        output = self.fc2(x)
        return output
```

Figure 7- Deep Q-Learning Agent - Network Implementation

B. Image Processing for PyTorch

CARLA's output data types are numpy arrays. It was decided to keep this datatype when developing the gym-like environment in order to have a final environment compatible with multiple machine learning frameworks. In other words, adapting the environment to PyTorch tensor data type would make the environment only compatible with PyTorch networks.

This implied that each observation had to be processed prior to pass it as an input to the PyTorch network. The state_gen() function placed on the main() loop receives the 'mode' of the sensor (the type of camera generating the signal), convert the numpy array signals into torch tensors, sends them through the stack_img() function that stacks the four three-channel frames, and it finally pass the stack through the preprocess_img_stack() function that generates the input to the agent, a stack of four one-channel frame. When state_gen() receives a depth map signal it skips the preprocess_img_stack() function.

C. Training Loop

The training loop begins by instantiating a policy network and a target network. The target network has weights identical to those in the policy network. Both networks are instantiated with an output equal to the number of actions in the environment's action space.

AdamW optimization is used in the optimization step. This is a stochastic gradient descent method that is based on adaptive estimation of first-order and second-order moments with an added method to decay weights per the techniques discussed in the paper, 'Decoupled Weight Decay Regularization' by Loshchilov, Hutter et al., 2019.

According to Kingma et al., 2014, the underlying Adam method is "computationally efficient, has little memory requirement, invariant to diagonal rescaling of gradients, and is well suited for problems that are large in terms of data/parameters".

Then a replay memory class is instantiated. The replay memory is implemented as a deque with a length equal to 10,000 items. Each item is a named tuple class: 'Transition'. Each transition object stores a state, an action taken given that state, the next observed state, and the reward obtained from the environment after taking the action.

The replay memory is the cornerstone of 'experience replay', which is a technique used in deep q-learning. The reason for using replay memory is to break the correlation between consecutive samples. If the network learned only from consecutive samples of experience as they occurred sequentially in the environment, the samples would be highly correlated and would therefore lead to inefficient learning. This time correlation is characteristic of any time series. Taking random samples from replay memory breaks this correlation.

In addition to breaking the correlation typical of sequential data, experience replay avoids the problem of 'forgetting' action-value pairs taken at earlier stages of the simulation.

At each optimization step a batch of 128 states from the replay memory is passed through the policy network and using the actions stored in the replay memory the action's values are calculated. Then the next state is passed through the target network and the expected state action values are calculated. Then the loss is calculated by comparing the values calculated by the policy network and the expected values calculated by the target network. The loss is calculated using the smooth L1 loss criterion. Then a backward propagation optimization step is taken to update the policy network.

On each episode an action is selected using the policy network. The action is passed to the environment, then an observation and a reward are obtained from the environment. The initial state, the action, the observed next state, and the reward are pushed to the replay memory instance. Finally, the next state observed becomes the state observation that is passed through the agent to select the next action.

D. Tested Parameters

For this project, given the time constraints only tau and the epsilon decay parameter were varied widely. Values of tau ranged from 0.001 to 0.1. The epsilon decay factor was varied discretely for values in the set of 1000, 2500, 5000, 10000.

VI. Metrics

For a self-driving vehicle project the natural measure of success would be to get an agent capable of driving without triggering the collision sensor for an infinite amount of time. That means it is desired the agent to last as long as possible without hitting an object in the environment. Thereby, one measure collected was episode duration.

However, an agent can succeed in lasting for an indefinite amount of time without triggering the collision sensor by just standing in place. Thereby, the reward function was linked to speed (on top of being linked to the collision sensor). In the final version of the reward function a penalty of -1 at each step was imposed to the agent for not moving. Because of this reason the second measure of success was the cumulative reward obtained by the agent on each episode.

VII. Discussion and Results

The agent went for not lasting more than a couple of seconds to last on average between 20 and 40 seconds before hitting an object. This is still considered poor performance. However, some of the intervention on the approach, the environment, and the training loop did result in some improvement.

A. Replay memory

The first version of the optimization step explicitly excluded terminal states and it assigned to them a reward equal to zero. This architecture followed the design of agents that worked in several environments.

However, terminal states in the custom environment are associated with the most negative reward: -100. Masking terminal states and/or zeroing them out would remove from the replay memory action that led to a high penalty.

Given that the reward function was associated with speed, it was hypothesized that the agent could simply accelerate to the top speed it could reach to collect the high rewards awarded by the higher speed and then loose control of the vehicle (i.e. unable to turn because of the inertia) then crashing without taking into account the big negative reward consequence of the speeding behavior.

Because of this the optimization step was modified to take all states, including terminal states.

B. Action Space

The initial action space had 13 legal actions. It was hypothesized that having more actions to choose from would give the agent flexibility. Furthermore, it was assumed that legal actions conducive to negative rewards would be discarded by the agent.

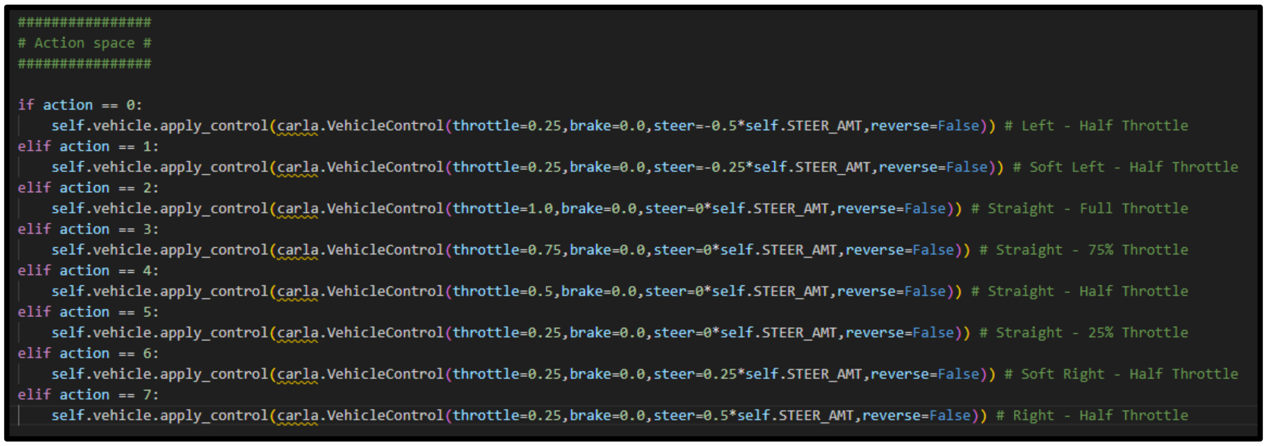
However, when observing the behavior of the agent the results were discouraging. In particular, hard turns at full speed made the agent crash. After a second thought, considerations were given to the exploration versus exploitation issue. At the beginning of the training random actions are taken, if within the legal actions hard turns at full speed are available, the exploration stage will pick them up.

CARLA's engine is trying to replicate the physics of the real world. When a hard turn is taken at full speed in the real world any vehicle gets destabilized. Back into the simulation, after the hard turn at full speed action is passed, there are a couple of seconds left before crashing, where the agent still makes decisions. Since the reward function provides a high regard due to the high speed of the action and speed data is not passed as a stack, chances are the training algorithm does not detect this action as a bad action. Hence, it learns to repeat the action.

As training progressed, it was observed that the agent would attempt to select exclusively hard turns at full throttle. This strategy would work in wide avenues, the vehicle was driven in circles by the agent. However, this strategy failed in most locations of the map.

Because of the aforementioned considerations the action space was narrowed down. Actions that would not make sense in the real world, such as hard turns at full throttle, were removed from the action space. Figure 8 depicts the final action space.

Narrowing down the action space led to improvement in the performance of the agent.



```
################
# Action space #
################

if action == 0:
    self.vehicle.apply_control(carla.VehicleControl(throttle=0.25,brake=0.0,steer=-0.5*self.STEER_AMT,reverse=False)) # Left - Half Throttle
elif action == 1:
    self.vehicle.apply_control(carla.VehicleControl(throttle=0.25,brake=0.0,steer=-0.25*self.STEER_AMT,reverse=False)) # Soft Left - Half Throttle
elif action == 2:
    self.vehicle.apply_control(carla.VehicleControl(throttle=1.0,brake=0.0,steer=0*self.STEER_AMT,reverse=False)) # Straight - Full Throttle
elif action == 3:
    self.vehicle.apply_control(carla.VehicleControl(throttle=0.75,brake=0.0,steer=0*self.STEER_AMT,reverse=False)) # Straight - 75% Throttle
elif action == 4:
    self.vehicle.apply_control(carla.VehicleControl(throttle=0.5,brake=0.0,steer=0*self.STEER_AMT,reverse=False)) # Straight - Half Throttle
elif action == 5:
    self.vehicle.apply_control(carla.VehicleControl(throttle=0.25,brake=0.0,steer=0*self.STEER_AMT,reverse=False)) # Straight - 25% Throttle
elif action == 6:
    self.vehicle.apply_control(carla.VehicleControl(throttle=0.25,brake=0.0,steer=0.25*self.STEER_AMT,reverse=False)) # Soft Right - Half Throttle
elif action == 7:
    self.vehicle.apply_control(carla.VehicleControl(throttle=0.25,brake=0.0,steer=0.5*self.STEER_AMT,reverse=False)) # Right - Half Throttle
```

Figure 8- Action Space of the Environment

C. Reward Function

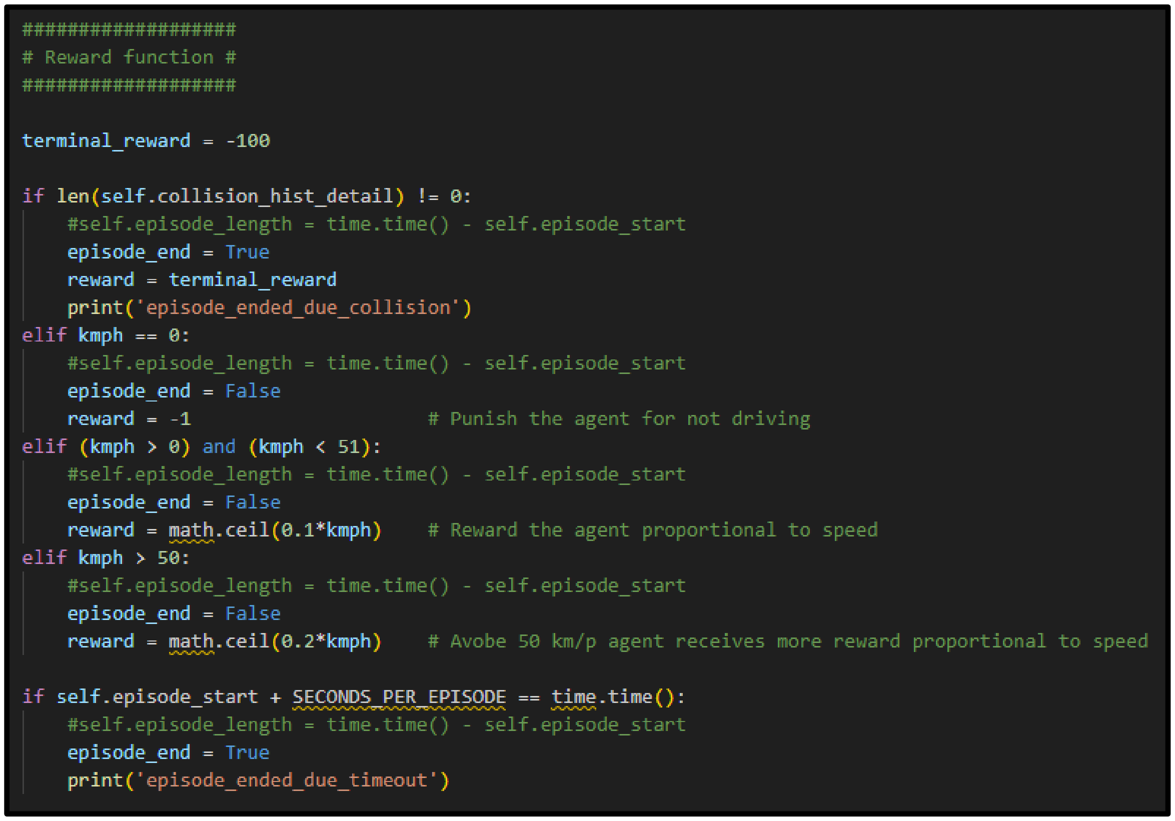
Just like with the replay memory, the reward function followed a template that work in other environments: an important penalty for failing at the task (crashing on an object), and a small negative reward for an action that is not consider optimal. In this case, the small negative reward of -1 was given at each step when the vehicle was not above a given speed.

However, with the large action space, the high rewards of speeding may become rarely available to the agent. Thereby, it might have been that the agent faced two choices:

- Driving for a few seconds and earning negative rewards at every step, just to end up crashing and getting a big negative reward, or
- Crashing as soon as possible and earning only the big negative reward (avoiding the extra penalties accumulated over time for driving slowly)

It was hypothesized that the reward function incentives were encouraging sudden crashes. The opposite of the goal set for the agent.

Because of this reason the reward function had to be redesigned to better suit this environment. Improvement was observed in the behavior of the agent once modifications made on the reward function. Figure 9 showcases the final design of the reward function.



```
###################
# Reward function #
###################

terminal_reward = -100

if len(self.collision_hist_detail) != 0:
    #self.episode_length = time.time() - self.episode_start
    episode_end = True
    reward = terminal_reward
    print('episode_ended_due_collision')
elif kmph == 0:
    #self.episode_length = time.time() - self.episode_start
    episode_end = False
    reward = -1                     # Punish the agent for not driving
elif (kmph > 0) and (kmph < 51):
    #self.episode_length = time.time() - self.episode_start
    episode_end = False
    reward = math.ceil(0.1*kmph)    # Reward the agent proportional to speed
elif kmph > 50:
    #self.episode_length = time.time() - self.episode_start
    episode_end = False
    reward = math.ceil(0.2*kmph)    # Avobe 50 km/p agent receives more reward proportional to speed

if self.episode_start + SECONDS_PER_EPISODE == time.time():
    #self.episode_length = time.time() - self.episode_start
    episode_end = True
    print('episode_ended_due_timeout')
```

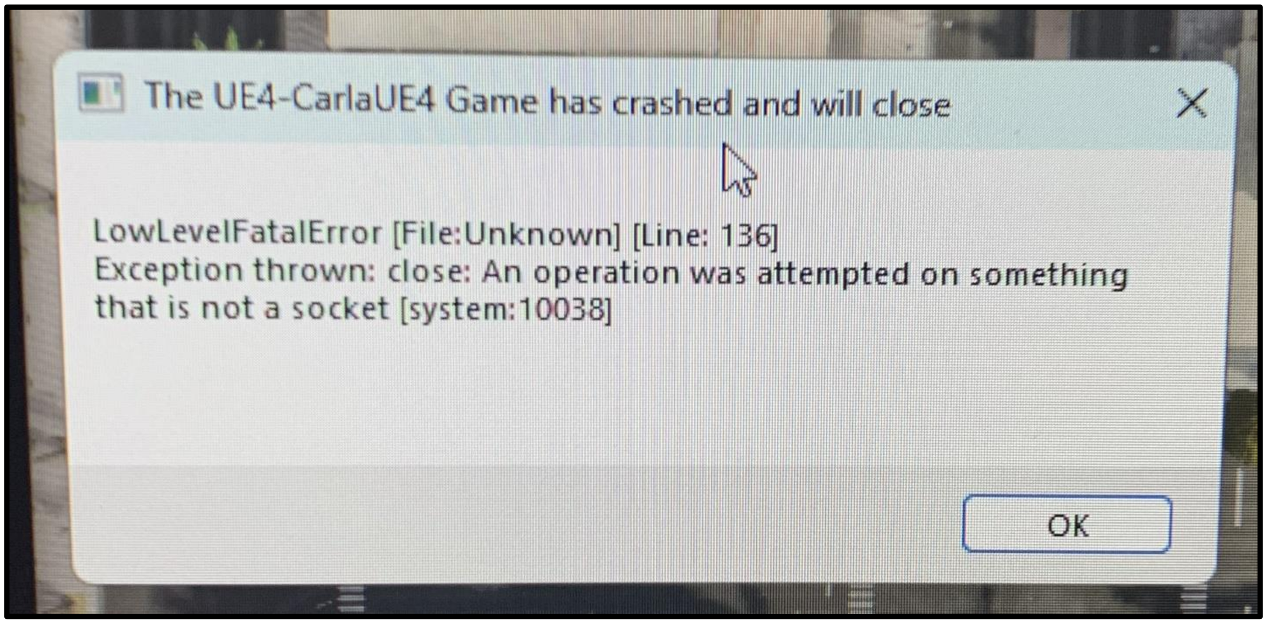
Figure 9- Final Reward Function

D. Semantic Segmentation Camera

Training trials where the semantic segmentation camera was used did not lead to good performance or desired behavior. Thereby, most experiments used the signal from the RGB camera sensor.

E. Training Results

One of the challenges faced during the trials was handling an error thrown by Unreal Engine. This issue has been raised at CARLA's GitHub forum and at the time of this writing it has not been closed. The open issue can be found in the following URL: https://github.com/carla-simulator/carla/issues/4861. Just as it is described in the issue's blog this error is thrown unexpectedly and it can happen at any time during the training process.



```
The UE4-CarlaUE4 Game has crashed and will close

LowLevelFatalError [File:Unknown] [Line: 136]
Exception thrown: close: An operation was attempted on something
that is not a socket [system:10038]
```

Figure 10. Fatal Error Thrown by Unreal Engine

To go around this unavoidable issue milestones were established in the training loop. Every 50 episodes the weights of both the target and the policy network were saved on disk. Then a 'continue' version of the training algorithm was implemented. If the training was interrupted because of the error, then the 'continue' algorithm would load the latest version of the networks' weights saved in the aforementioned milestones. After loading these weights, the training would be able to resume by continue to optimizing the trained weights. The only caveat of this approach is that every time the 'continue' algorithm was used the epsilon was set back to its initial value, thereby, exploration would start over.

This issue improved after installing the latest version of CARLA, launched only a few weeks ago.

Figure 11 showcases the number of exploration steps in the blue plot line series and the exploitation steps taken on the orange plot line series. Each series plots steps taken on each episode/epoch. Each shaded area (light blue or white) corresponds the kick in of the 'continue' algorithm.

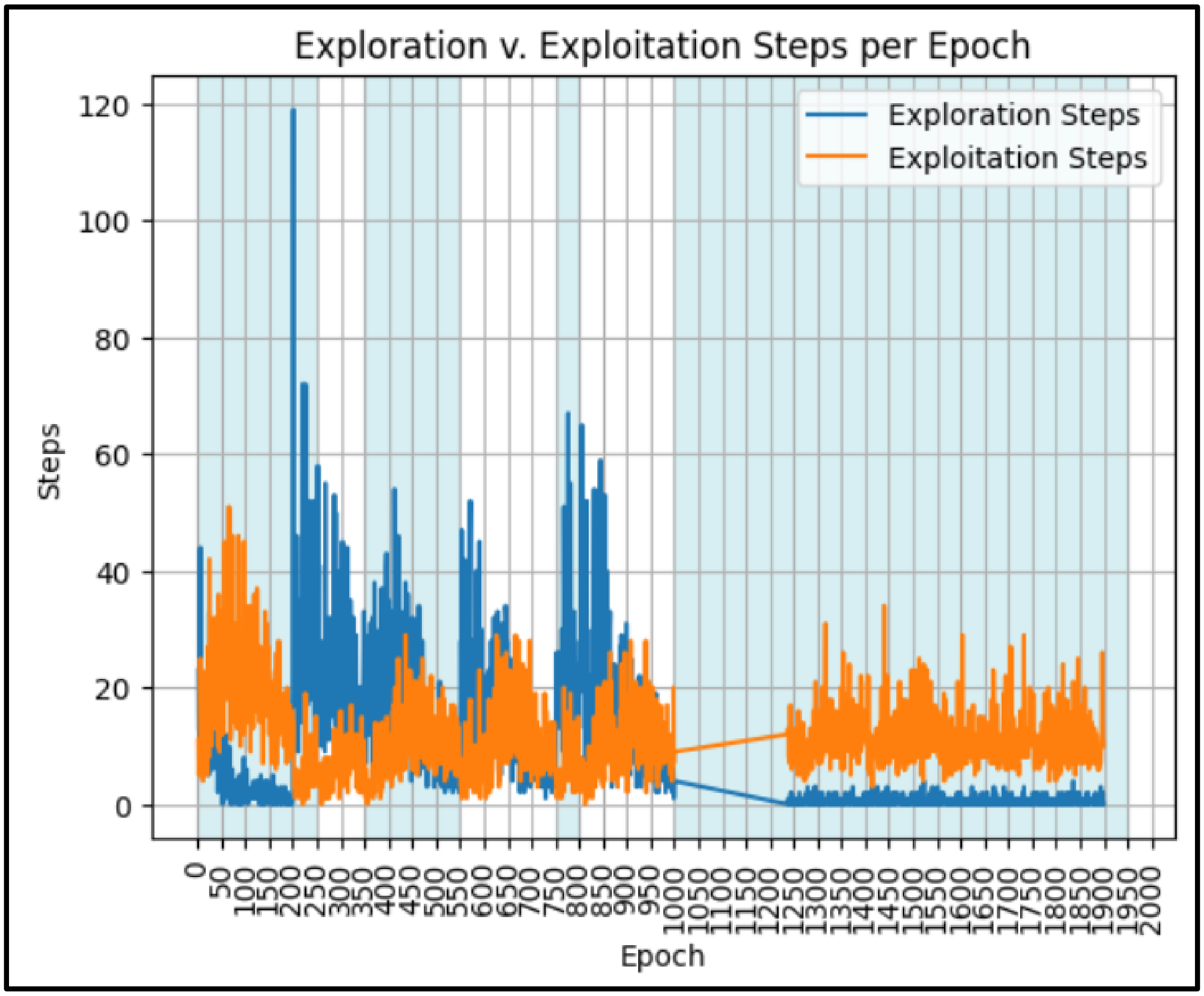


Figure 11. Exploration v. Exploitation Steps while Training the Agent for 2000 episodes/epochs

In figure 11 it can be noticed how at the beginning of each ‘continue’ trial random actions are taken more frequently, whereas policy actions are taken towards the end. It is worthwhile pointing that data between episodes 1000 through 1230 is missing. This was because the log deleted earlier outputs. Thus, the exploration peak on the final trial is not observed in the plot.

On figures 12 and 13 it can be seen that the top performance of the agent was reaching fairly quickly during the training trial. Surprisingly, training for longer time led to a deterioration in the agent’s performance.

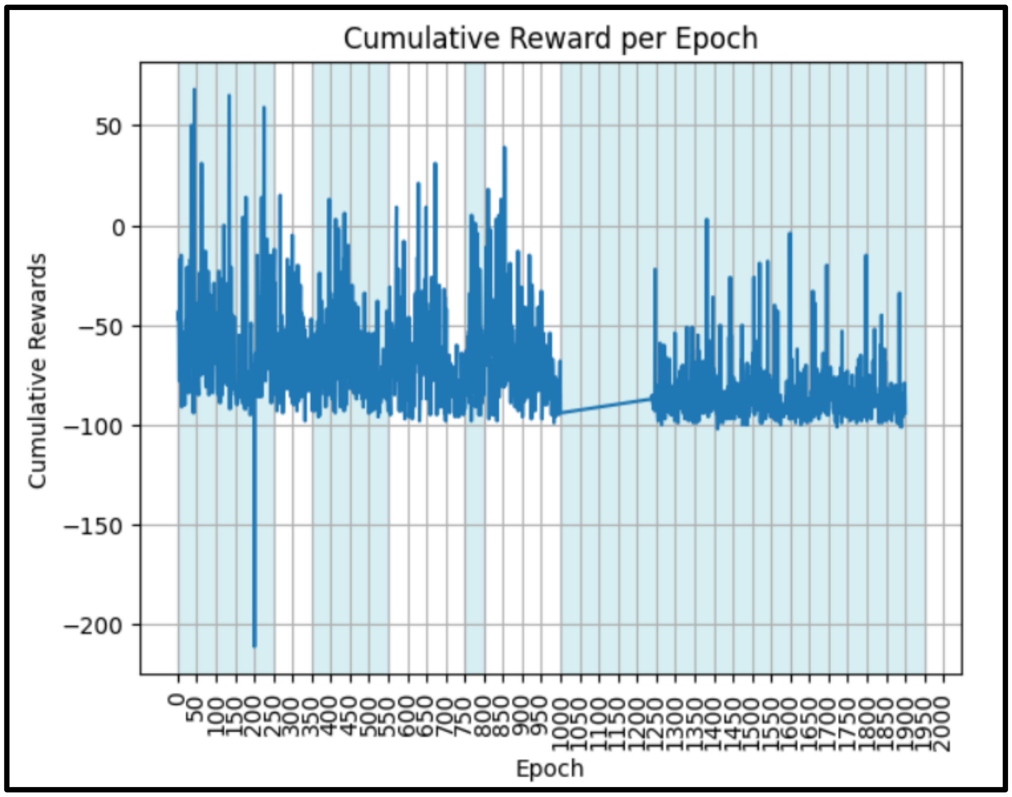


Figure 12. Cumulative Reward per Episode while Training the Agent for 2000 episodes/epochs

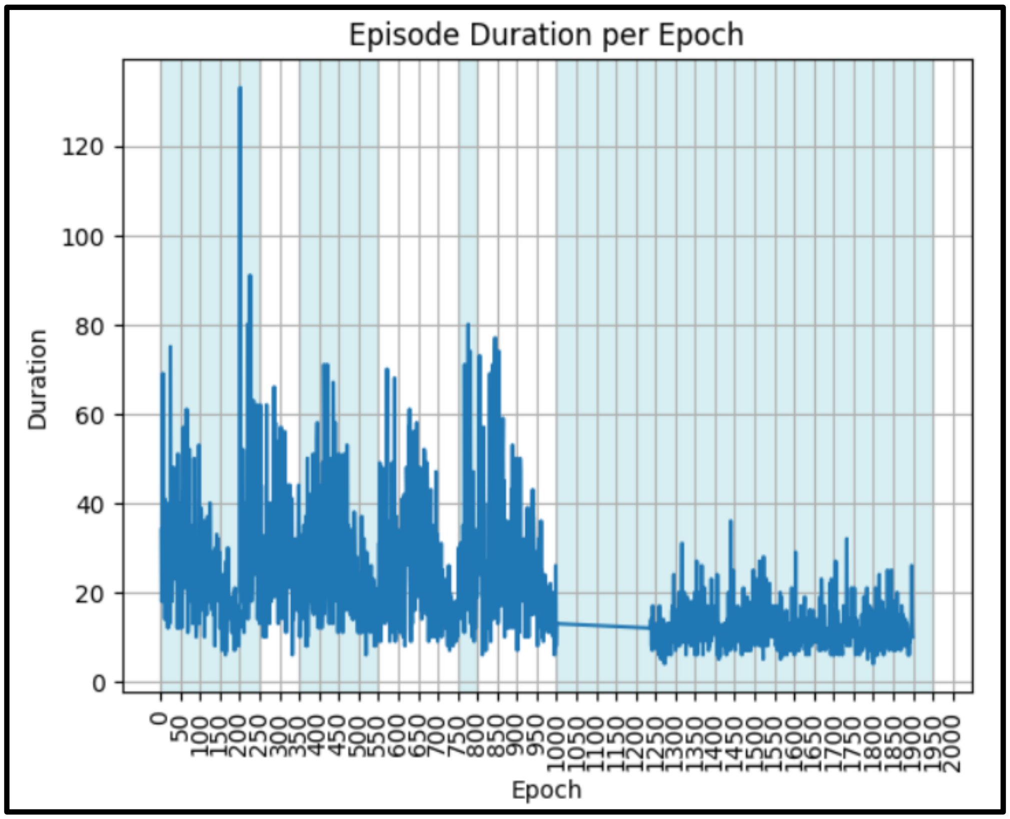


Figure 13. Episode Duration while Training the Agent for 2000 episodes/epochs

Because of the deterioration of the agent after long periods of training every experiment thereafter was run for maximum 250 epochs.

The best results were obtained with a value of tau of 0.001, a very soft update of the target policy; and with an epsilon decay factor of 5000. The deterioration of the agent with longer training times was observed in all the experiments. Thereby, based on the information derived from the plots in figure 14 and 15, for the demo the weights of the policy after 100 training episodes were selected.

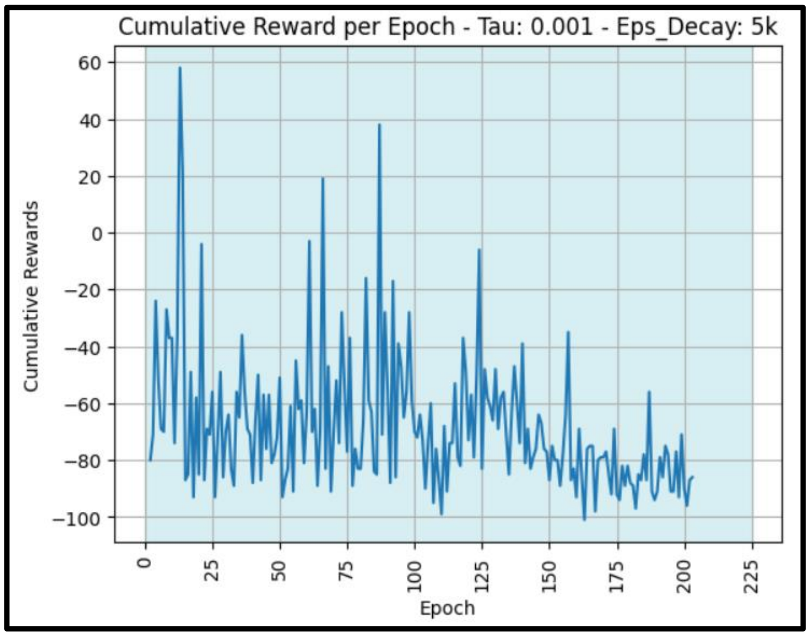


Figure 14. Cumulative Reward per Episode – Experiment 5

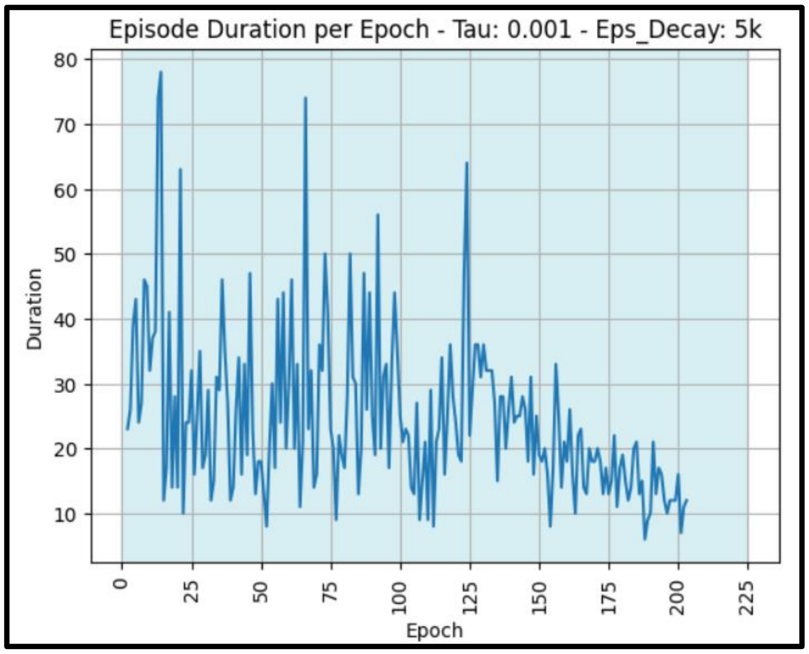


Figure 15. Episode Duration – Experiment 5

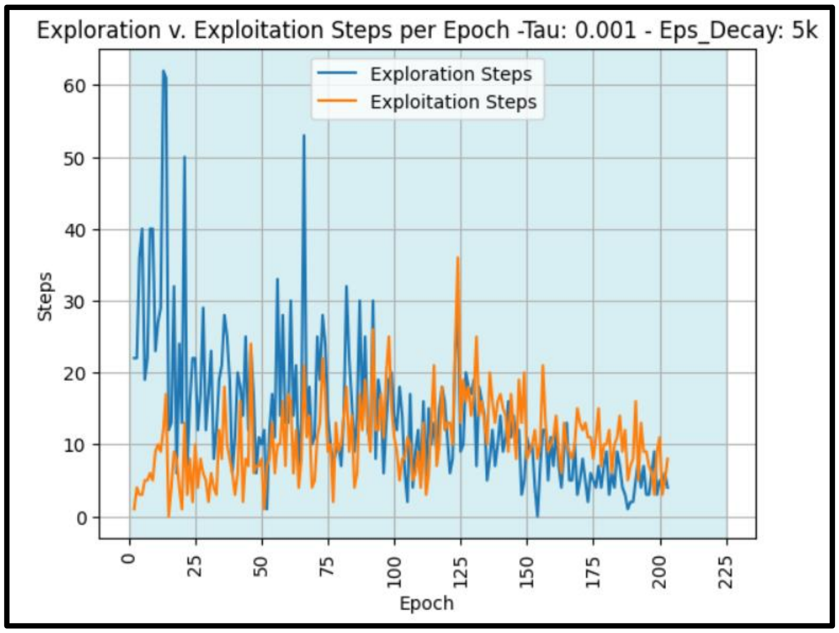


Figure 16. Exploration v. Exploitation Steps – Experiment 5

VIII. Conclusion and Future Work

The agent's performance was far from satisfactory. However, many improvements were made. Among the improvements, a few of them were related with the topic of crossing the reality gap (i.e. removing actions that a human driver would never take).

For future work, if a system with more memory and compute power becomes available, more sensors should be used. Likewise, it may be convenient to drop the speed on its present form as an input.

Another item that might require further experimentation is the reward function. It might be possible that the agent performs better if rewards are tied to the episode duration instead of the speed of the vehicle. Notwithstanding, an incentive must be included to avoid the vehicle from standing still.

Finally, the performance deterioration issue might be a sign that the soft update is still too hard at a value of tau equal to 0.001. It seems the learning is thrown in the wrong direction very easily. Perhaps the self-driving agent is a problem better suited for techniques that focus on avoiding updates in the wrong direction that the training cannot overcome, such as PPO.

IX. References

[1]. R. Sutton, A. Barto, "Reinforcement Learning: An introduction," 2nd edition, 2018.