AI Project

Team Members:

No.	Name	Code
1	Reham Hamdi Mohamed	20812021201109
2	Shahd Elsayed Ahmed	20812021200543
3	Matilda Ashraf Malak	20812021201241
4	Mariam farouk slama	20812021201289
5	Nancy Ayman Nabil	20812021200506

SELF-DRIVING CAR SIMULATION WITH REINFORCEMENT LEARNING

Overview:-

This project presents a reinforcement learning-based self-driving car simulation, where an intelligent agent learns to drive autonomously in a virtual environment. The main idea is to allow the agent to explore and interact with its surroundings, receiving rewards or penalties depending on its driving behavior. Through repeated training episodes, the agent gradually improves its decision-making to navigate roads safely and avoid collisions. After the training phase, the agent is tested independently, using a pre-trained model to predict the best driving actions based on real-time visual input. This system highlights the power of combining deep learning and computer vision to build adaptive, intelligent driving agents that mimic real-world autonomous vehicle behavior. The project serves as a foundational step toward understanding how AI can be applied in real-time, safety-critical applications like autonomous driving.

File train_self_driving_agent.py:-

This code implements a **Deep Q-Network (DQN)** agent for autonomous driving in CARLA simulator using reinforcement learning and a **convolutional Neural Network (CNN)**

- ## Implementation

```
import glob
import os
import sys
import random
import time
import numpy as np
import cv2
import math
from collections import deque
from keras.applications.xception import Xception
from keras.layers import Dense, AveragePooling2D, Flatten
from keras.optimizers import Adam
from keras.models import Model
from keras.callbacks import TensorBoard
from keras.callbacks import ModelCheckpoint

import tensorflow as tf
from keras.layers import Activation, Dense
import keras.backend.tensorflow_backend as backend
from keras import Sequential
from keras.layers.convolutional import Conv2D
from threading import Thread

from tqdm import tqdm

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
import carla


SHOW_PREVIEW = False
IM_WIDTH = 640
IM_HEIGHT = 480
SECONDS_PER_EPISODE = 10
REPLAY_MEMORY_SIZE = 5_000
MIN_REPLAY_MEMORY_SIZE = 1_000
MINIBATCH_SIZE = 16
PREDICTION_BATCH_SIZE = 1
TRAINING_BATCH_SIZE = MINIBATCH_SIZE // 4
UPDATE_TARGET_EVERY = 5
MODEL_NAME = "Sequential"

MEMORY_FRACTION = 0.4
MIN_REWARD = -200

EPISODES = 12000
DISCOUNT = 0.99
epsilon = 1
EPSILON_DECAY = 0.95 ## 0.9975 99975
MIN_EPSILON = 0.001

AGGREGATE_STATS_EVERY = 10
  return go(f, seed, [])
}
```

```
# Own Tensorboard class
class ModifiedTensorBoard(TensorBoard):

    # Overriding init to set initial step and writer (we want one log file for all .fit()
calldef __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)
```

```python
def process_img(self, image):
    i = np.array(image.raw_data)
    #print(i.shape)
    i2 = i.reshape((self.im_height, self.im_width, 4))
    i3 = i2[:, :, :3]
    if self.SHOW_CAM:
        cv2.imshow("", i3)
        cv2.waitKey(1)
    self.front_camera = i3

def step(self, action):
    if action == 0:
        self.vehicle.apply_control(carla.VehicleControl(throttle=1.0, steer=-1*self.STEER_AMT))
    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))

    v = self.vehicle.get_velocity()
    kmh = int(3.6 * math.sqrt(v.x*2 + v.y2 + v.z*2))

    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 + SECONDS_PER_EPISODE < time.time():
        done = True

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

```python
class CarEnv:
  SHOW_CAM = SHOW_PREVIEW
  STEER_AMT = 1.0
  im_width = IM_WIDTH
  im_height = IM_HEIGHT
  front_camera = None

  def _init_(self):
      self.client = carla.Client("localhost", 2000)
      self.client.set_timeout(50.0)
      self.world = self.client.get_world()
      self.blueprint_library = self.world.get_blueprint_library()
      self.model_3 = self.blueprint_library.filter("model3")[0]

  def reset(self):
      self.collision_hist = []
      self.actor_list = []

      self.transform = random.choice(self.world.get_map().get_spawn_points())
      self.vehicle = self.world.spawn_actor(self.model_3, self.transform)
      self.actor_list.append(self.vehicle)

      self.rgb_cam = self.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")

      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.process_img(data))

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

      colsensor = self.blueprint_library.find("sensor.other.collision")
      self.colsensor = self.world.spawn_actor(colsensor, transform, attach_to=self.vehicle)
      self.actor_list.append(self.colsensor)
      self.colsensor.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

  def collision_data(self, event):
      self.collision_hist.append(event)
```

```python
class DQNAgent:

    def _init_(self):
        self.model = self.create_model()
        self.target_model = self.create_model()
        self.target_model.set_weights(self.model.get_weights())

        self.replay_memory = deque(maxlen=REPLAY_MEMORY_SIZE)

        self.tensorboard = ModifiedTensorBoard(log_dir=f"logs/{MODEL_NAME}-{int(time.time())}")
        self.target_update_counter = 0
        self.graph = tf.get_default_graph()

        self.terminate = False
        self.last_logged_episode = 0
        self.training_initialized = False

    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(3, activation="linear"))
        model = Model(inputs=model.input, outputs=model.output)
        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)
```

```python
def train(self):
        if len(self.replay_memory) < MIN_REPLAY_MEMORY_SIZE:
            return

        minibatch = random.sample(self.replay_memory, MINIBATCH_SIZE)

        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)

        new_current_states = np.array([transition[3] for transition in minibatch])/255
        with self.graph.as_default():
            future_qs_list = self.target_model.predict(new_current_states, PREDICTION_BATCH_SIZE)

        X = []
        y = []

        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
            else:
                new_q = reward

            current_qs = current_qs_list[index]
            current_qs[action] = new_q

            X.append(current_state)
            y.append(current_qs)

        log_this_step = False
        if self.tensorboard.step > self.last_logged_episode:
            log_this_step = True
            self.last_log_episode = self.tensorboard.step

        with self.graph.as_default():
            self.model.fit(np.array(X)/255, np.array(y), batch_size=TRAINING_BATCH_SIZE, verbose=0,
shuffle=False, callbacks=[self.tensorboard] if log_this_step else None)


        if log_this_step:
            self.target_update_counter += 1

        if self.target_update_counter > UPDATE_TARGET_EVERY:
            self.target_model.set_weights(self.model.get_weights())
            self.target_update_counter = 0

    def get_qs(self, state):
        return self.model.predict(np.array(state).reshape(-1, *state.shape)/255)[0]

    def train_in_loop(self):
        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, callbacks=[self.tensorboard])

        self.training_initialized = True

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

```python
if __name__ == '__main__':
    FPS = 60
    # For stats
    ep_rewards = [-200]

    # For more repetitive results
    random.seed(1)
    np.random.seed(1)
    tf.set_random_seed(1)

    # Create models folder
    if not os.path.isdir('models'):
        os.makedirs('models')

    #Create checkpoint folder
    if not os.path.isdir('models\\checkpoints'):
        os.makedirs('\\models\\checkpoints')

    # Create agent and environment
    agent = DQNAgent()
    env = CarEnv()


    # Start training thread and wait for training to be initialized
    trainer_thread = Thread(target=agent.train_in_loop, daemon=True)
    trainer_thread.start()
    while not agent.training_initialized:
        time.sleep(0.01)

    # Initialize predictions - forst prediction takes longer as of initialization that has to be done
    # It's better to do a first prediction then before we start iterating over episode steps
    agent.get_qs(np.ones((env.im_height, env.im_width, 3)))
```

```
# Iterate over episodes
    for episode in tqdm(range(1, EPISODES + 1), ascii=True, unit='episodes'):
        #try:

            env.collision_hist = []

            # Update tensorboard step every episode
            agent.tensorboard.step = episode

            # Restarting episode - reset episode reward and step number
            episode_reward = 0
            step = 1

            # Reset environment and get initial state
            current_state = env.reset()

            # Reset flag and start iterating until episode ends
            done = False
            episode_start = time.time()

            # Play for given number of seconds only
            while True:

                # This part stays mostly the same, the change is to query a model for Q values
                if np.random.random() > epsilon:
                    # Get action from Q table
                    action = np.argmax(agent.get_qs(current_state))
                else:
                    # Get random action
                    action = np.random.randint(0, 3)
                    # This takes no time, so we add a delay matching 60 FPS (prediction above takes longer)
                    time.sleep(1/FPS)

                new_state, reward, done, _ = env.step(action)

                # Transform new continous state to new discrete state and count reward
                episode_reward += reward

                # Every step we update replay memory
                agent.update_replay_memory((current_state, action, reward, new_state, done))

                current_state = new_state
                step += 1

                if done:
                    break

            # End of episode - destroy agents
            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'models/{MODEL_NAME}__{max_reward:_>7.2f}max_{average_reward:_>7.2f}avg_{min_reward:_>7.2f}mi
n__{int(time.time())}.hdf5')

            # Decay epsilon
            if epsilon > MIN_EPSILON:
                epsilon *= EPSILON_DECAY
                epsilon = max(MIN_EPSILON, epsilon)

            #Manual Checkpoint
            if episode > 0:

 agent.model.save(f'models/checkpoints/{MODEL_NAME}__{episode}__{max_reward:_>7.2f}max_{average_reward:_>7.2f}av
g_{min_reward:_>7.2f}min__{int(time.time())}.hdf5')

    # Set termination flag for training thread and wait for it to finish
    agent.terminate = True
    trainer_thread.join()

 agent.model.save(f'models/{MODEL_NAME}__{max_reward:_>7.2f}max_{average_reward:_>7.2f}avg_{min_reward:_>7.2f}mi
n__{int(time.time())}.hdf5')
```

▪ Explain of this code:-

This code builds and trains a reinforcement learning agent to drive a car in the **CARLA** simulator using **Deep Q-Learning.** It initializes a car with an RGB camera and a collision sensor, and captures road images to use as input for a neural network. The agent decides how to steer (left, straight, or right) based on these images. A **CNN model** predicts Q-values for each possible action, and the agent learns by receiving rewards—penalizing collisions and slow speeds, and rewarding safe, fast driving. Past experiences are stored in replay memory for training in small batches. A separate thread handles continuous training while the agent runs episodes. The code also tracks performance with TensorBoard and saves the model regularly when progress is made.

File Test_self_driving_agent.py:-

- ## Implementation:-

```
import random
from collections import deque
import numpy as np
import cv2
import time
import tensorflow as tf
import keras.backend.tensorflow_backend as backend
from keras.models import load_model
from train_self_driving_agent import CarEnv, MEMORY_FRACTION



MODEL_PATH = "Sequential____-4.00max_-103.00avg_-202.00min__1571284603.hdf5"
if __name__ == '__main__':

    # Memory fraction
    gpu_options = tf.GPUOptions(per_process_gpu_memory_fraction=MEMORY_FRACTION)
    backend.set_session(tf.Session(config=tf.ConfigProto(gpu_options=gpu_options)))

    # Load the model
    model = load_model(MODEL_PATH)

    # Create environment
    env = CarEnv()

    # For agent speed measurements - keeps last 60 frametimes
    fps_counter = deque(maxlen=60)

    # Initialize predictions - first prediction takes longer as of initialization that has to be
done# It's better to do a first prediction then before we start iterating over episode steps
    model.predict(np.ones((1, env.im_height, env.im_width, 3)))
```

```
# Loop over episodes
    while True:

        print('Restarting episode')
        # Reset environment and get initial state
        current_state = env.reset()
        env.collision_hist = []
        done = False
        # Loop over steps
        while True:

            # For FPS counter
            step_start = time.time()

            # Show current frame
            cv2.imshow(f'Agent - preview', current_state)
            cv2.waitKey(1)

            # Predict an action based on current observation space
            qs = model.predict(np.array(current_state).reshape(-1, *current_state.shape)/255)[0]
            action = np.argmax(qs)

            # Step environment (additional flag informs environment to not break an episode by time limit)
            new_state, reward, done, _ = env.step(action)
            # Set current step for next loop iteration
            current_state = new_state
            # If done - agent crashed, break an episode
            if done:
                break
            # Measure step time, append to a deque, then print mean FPS for last 60 frames, q values and taken
action
            frame_time = time.time() - step_start
            fps_counter.append(frame_time)
            print(f'Agent: {len(fps_counter)/sum(fps_counter):>4.1f} FPS | Action: [{qs[0]:>5.2f},
{qs[1]:>5.2f}, {qs[2]:>5.2f}] {action}')

        # Destroy an actor at end of episode
        for actor in env.actor_list:
            actor.destroy()
```

- ## Explain of this code:-

This code test a trained self-driving car agent using deep reinforcement learning. It starts by loading a pre-trained model and setting up GPU memory usage to avoid overload. The simulation environment (CarEnv) is then created, which mimics a real-world driving scenario and provides the agent with visual input from the car's camera.

Before the agent begins driving, the model makes an initial dummy prediction to ensure everything is properly initialized. The main loop continuously restarts episodes where the car drives autonomously based on what it sees. At each step, the current camera image is shown, and the model predicts the best driving action (e.g., left, right, forward) using the image as input.

The chosen action is applied in the environment, and the car's new state is returned. This loop continues until the car crashes or the episode ends. The script also measures and displays the performance (FPS) and the Q-values that represent how confident the model is about each action. After each episode, any simulated objects (actors) are removed to reset the scene for the next run.

- ## Output:

We set the agent to restart at:

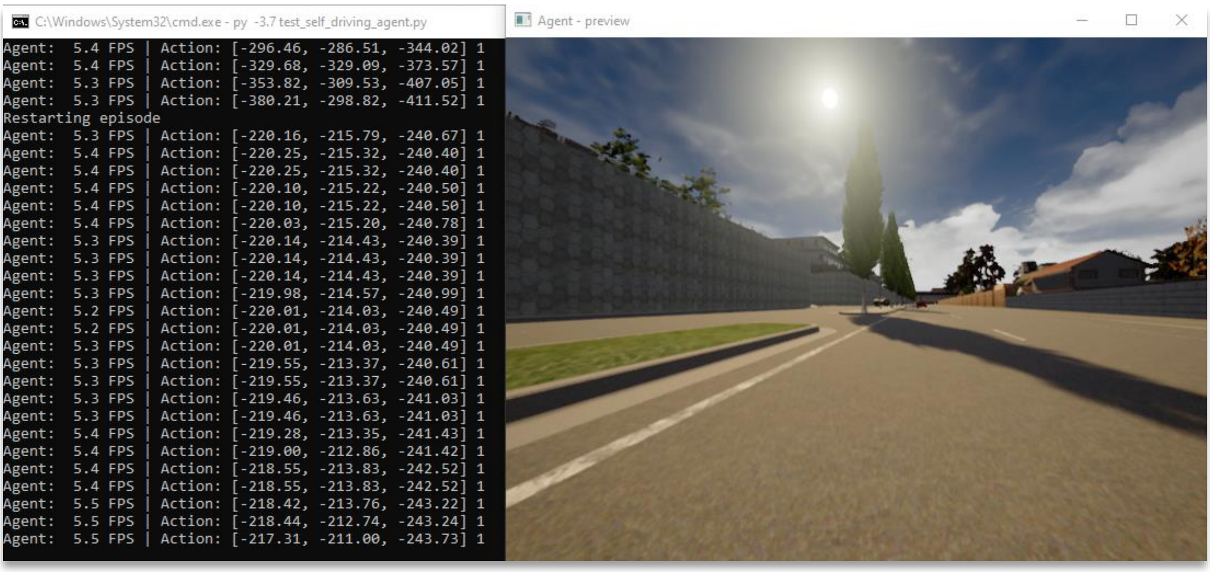
1. **Start of a New Episode:** At the beginning of each new episode, the agent resets to explore the environment and learn from scratch.

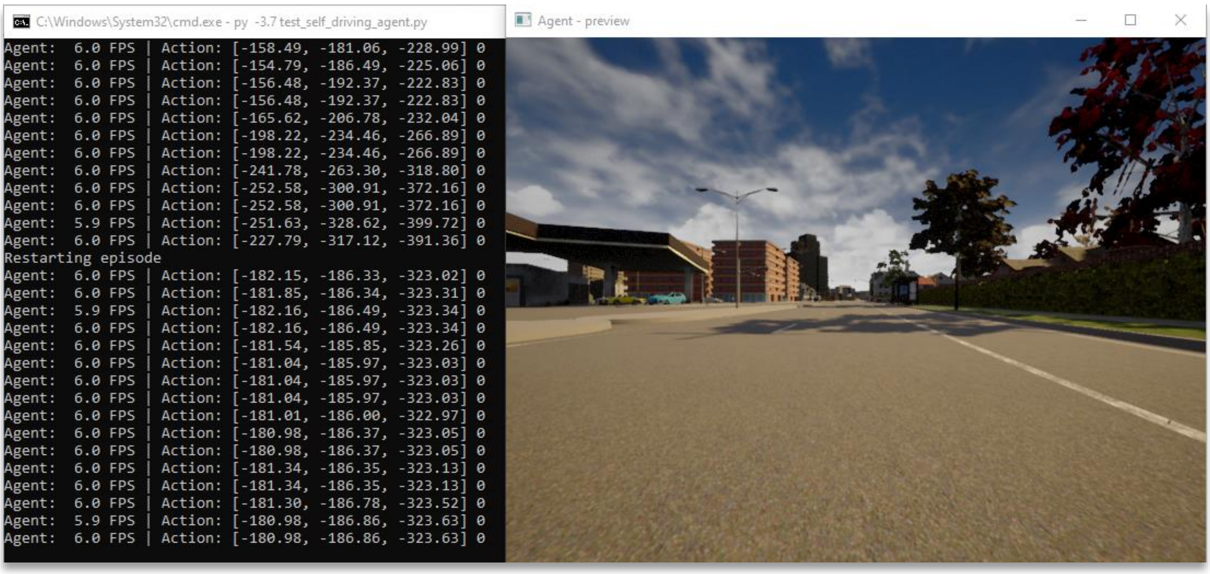
2. **Time Limit:** If the agent reaches the 10-second limit for the current episode, it restarts to begin a new one.

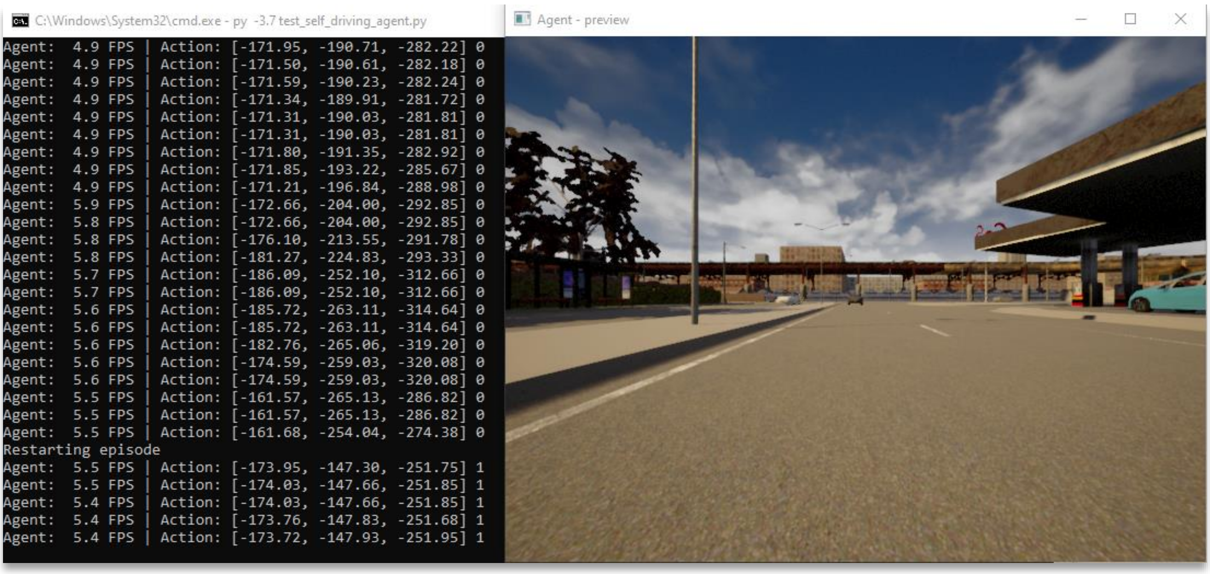
3. **Collision or Episode End:** If the agent crashes or the episode ends for any other reason, the environment resets, allowing the agent to start over and try again.

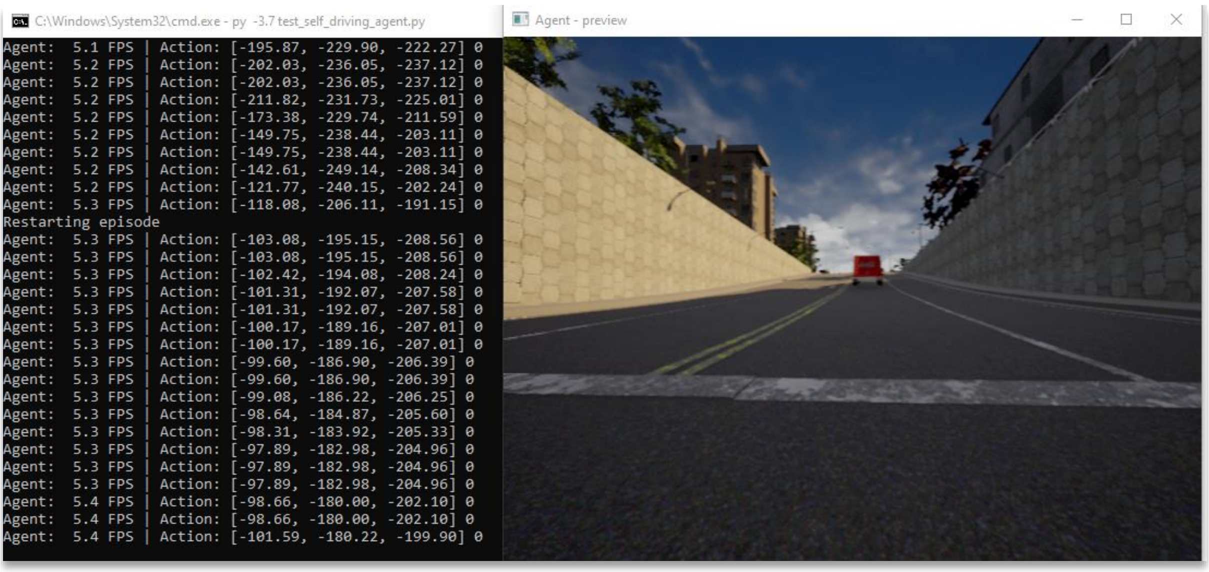
So, each restart offers the agent a fresh opportunity to explore, try different actions, and learn from its experiences. Restarting helps the agent adapt its behavior to improve its decision-making over time.

- **Different views of project running**






C:\Windows\System32\cmd.exe - py -3.7 test_self_driving_agent.py
Agent: 4.9 FPS | Action: [-171.95, -190.71, -282.22] 0
Agent: 4.9 FPS | Action: [-171.50, -190.61, -282.18] 0
Agent: 4.9 FPS | Action: [-171.59, -190.23, -282.24] 0
Agent: 4.9 FPS | Action: [-171.34, -189.91, -281.72] 0
Agent: 4.9 FPS | Action: [-171.31, -190.03, -281.81] 0
Agent: 4.9 FPS | Action: [-171.31, -190.03, -281.81] 0
Agent: 4.9 FPS | Action: [-171.80, -191.35, -282.92] 0
Agent: 4.9 FPS | Action: [-171.85, -193.22, -285.67] 0
Agent: 4.9 FPS | Action: [-171.21, -196.84, -288.98] 0
Agent: 5.9 FPS | Action: [-172.66, -204.00, -292.85] 0
Agent: 5.8 FPS | Action: [-172.66, -204.00, -292.85] 0
Agent: 5.8 FPS | Action: [-176.10, -213.55, -291.78] 0
Agent: 5.8 FPS | Action: [-181.27, -224.83, -293.33] 0
Agent: 5.7 FPS | Action: [-186.09, -252.10, -312.66] 0
Agent: 5.7 FPS | Action: [-186.09, -252.10, -312.66] 0
Agent: 5.6 FPS | Action: [-185.72, -263.11, -314.64] 0
Agent: 5.6 FPS | Action: [-185.72, -263.11, -314.64] 0
Agent: 5.6 FPS | Action: [-182.76, -265.06, -319.20] 0
Agent: 5.6 FPS | Action: [-174.59, -259.03, -320.08] 0
Agent: 5.6 FPS | Action: [-174.59, -259.03, -320.08] 0
Agent: 5.5 FPS | Action: [-161.57, -265.13, -286.82] 0
Agent: 5.5 FPS | Action: [-161.57, -265.13, -286.82] 0
Agent: 5.5 FPS | Action: [-161.68, -254.04, -274.38] 0
Restarting episode
Agent: 5.5 FPS | Action: [-173.95, -147.30, -251.75] 1
Agent: 5.5 FPS | Action: [-174.03, -147.66, -251.85] 1
Agent: 5.4 FPS | Action: [-174.03, -147.66, -251.85] 1
Agent: 5.4 FPS | Action: [-173.76, -147.83, -251.68] 1
Agent: 5.4 FPS | Action: [-173.72, -147.93, -251.95] 1
Agent - preview


C:\Windows\System32\cmd.exe - py -3.7 test_self_driving_agent.py
Agent: 5.1 FPS | Action: [-195.87, -229.90, -222.27] 0
Agent: 5.2 FPS | Action: [-202.03, -236.05, -237.12] 0
Agent: 5.2 FPS | Action: [-202.03, -236.05, -237.12] 0
Agent: 5.2 FPS | Action: [-211.82, -231.73, -225.01] 0
Agent: 5.2 FPS | Action: [-173.38, -229.74, -211.59] 0
Agent: 5.2 FPS | Action: [-149.75, -238.44, -203.11] 0
Agent: 5.2 FPS | Action: [-149.75, -238.44, -203.11] 0
Agent: 5.2 FPS | Action: [-142.61, -249.14, -208.34] 0
Agent: 5.2 FPS | Action: [-121.77, -240.15, -202.24] 0
Agent: 5.3 FPS | Action: [-118.08, -206.11, -191.15] 0
Restarting episode
Agent: 5.3 FPS | Action: [-103.08, -195.15, -208.56] 0
Agent: 5.3 FPS | Action: [-103.08, -195.15, -208.56] 0
Agent: 5.3 FPS | Action: [-102.42, -194.08, -208.24] 0
Agent: 5.3 FPS | Action: [-101.31, -192.07, -207.58] 0
Agent: 5.3 FPS | Action: [-101.31, -192.07, -207.58] 0
Agent: 5.3 FPS | Action: [-100.17, -189.16, -207.01] 0
Agent: 5.3 FPS | Action: [-100.17, -189.16, -207.01] 0
Agent: 5.3 FPS | Action: [-99.60, -186.90, -206.39] 0
Agent: 5.3 FPS | Action: [-99.60, -186.90, -206.39] 0
Agent: 5.3 FPS | Action: [-99.08, -186.22, -206.25] 0
Agent: 5.3 FPS | Action: [-98.64, -184.87, -205.60] 0
Agent: 5.3 FPS | Action: [-98.31, -183.92, -205.33] 0
Agent: 5.3 FPS | Action: [-97.89, -182.98, -204.96] 0
Agent: 5.3 FPS | Action: [-97.89, -182.98, -204.96] 0
Agent: 5.3 FPS | Action: [-97.89, -182.98, -204.96] 0
Agent: 5.4 FPS | Action: [-98.66, -180.00, -202.10] 0
Agent: 5.4 FPS | Action: [-98.66, -180.00, -202.10] 0
Agent: 5.4 FPS | Action: [-101.59, -180.22, -199.90] 0
Agent - preview

You can view the demonstration video of the project by clicking the following link:

https://drive.google.com/file/d/1-eiE0cNZjwP4qMBH3nCF6N8SktPg_qxX/view?usp=sharing

Refernces:

https://carla.readthedocs.io/en/latest/start_quickstart/#b-package-installation

https://github.com/carla-simulator/carla/blob/master/Docs/download.md

Thank you