CS 5392 Spring 2023 Assignment-1 Tutorial-2

Title: -

Reinforcement Learning for Self-Driving Cars: Training an Autonomous Agent using Q-learning and Deep Q-Network (DQN) in Carla Simulator using Python.

Objective: -

- The primary objective is to provide hands-on experience in implementing a reinforcement learning algorithm for self-driving cars using the Carla Python API and gaining practical knowledge in the field of autonomous driving and reinforcement learning.
- Learn to implement a reinforcement learning algorithm using Q-learning or Deep Q-Network (DQN) to train a self-driving car. The agent will learn to navigate the environment by acting based on observations and receiving reward signals for its performance.

Level: Hard

Background:

i. Reinforcement learning

Reinforcement learning is a type of machine learning that involves training an agent to make decisions in an environment based on trial and error [4]. In this tutorial, we will explore how to use reinforcement learning to train a self-driving autonomous car in the CARLA simulator, a popular open-source simulator for autonomous driving research.

ii. Carla: -

Carla (Car Learning to Act) is an open-source autonomous driving simulator the Computer Vision Center developed. This simulator provides a realistic 3D environment for testing and training autonomous driving algorithms. CARLA has a Python API that allows users to interact with the simulator and control the virtual vehicles, sensors, and traffic. Additionally, Carla allows for the creation of a wide range of scenarios, including different road layouts, weather conditions, and traffic patterns, which can be used to evaluate the robustness of autonomous driving algorithms [5].

iii. Software Requirements:

It is recommended to have a more powerful computer with a higher-end GPU and a faster internet connection [1].

- Operating system: Any 64-bit version of Windows or Ubuntu 18.04
- Disk space: At least 165 GB of free space is required, with 32 GB used by the Carla simulator and 133 GB used by related software installations, including the Unreal Engine.
- GPU: An adequate GPU is required to run the simulator, with at least 6 GB of memory, although 8 GB is recommended. A dedicated GPU is highly recommended for machine learning.
- Network ports: The simulator requires two TCP ports, 2000 and 2001, which must not be blocked by firewalls or other applications.
- You need to have a GitHub account linked to Unreal Engine's account.

iv. Installation Requirements: -

- Install Visual Studio 2019 or later with C++ support.
- Install Git, CMake, a file compression software(7Zip), and Make.
- Install Python 3.7 or later.
- Install the Unreal Engine prerequisites (Visual Studio Build Tools, DirectX, Windows 10 SDK)
- Clone the Carla repository from GitHub.
- Configure Carla using CMake.
- Build the Carla client and server executables using Visual Studio
- Download the required assets for Carla.
- Run the Carla server and launch a client to test the installation.

v. **Libraries installed: -**

- Keras-2.2.5---

Generally, In Keras is a high-level deep learning API for building and training neural networks. It is user-friendly, modular, and supports multiple backends[6]. With built-in preprocessing and data augmentation, Keras allows for fast prototyping of different architectures and hyperparameters

- Keras-Applications:1.0.8---

So, basically in this Keras-Applications 1.0.8 provides pre-built deep learning models, such as VGG16, VGG1, Xception etc.., that can be used for Q-learning and Deep Q-Networks. We used Xception models to easily adapt to reinforcement learning tasks by replacing the output layer with a Q-value output layer. This can save time and effort compared to building and training a new model from scratch, allowing for faster experimentation and iteration.

- Keras-Preprocessing:1.1.2---

Keras-Preprocessing 1.1.2 provides a set of tools for data preprocessing and augmentation, which can be useful for Q-learning and Deep Q-Networks[6]. For example, it includes image preprocessing functions such as image resizing, cropping, and normalization, as well as data augmentation techniques such as random rotations and flips. These tools can help to improve the performance and robustness of the models by increasing the diversity of the training data. One of the latest features introduced in Keras-Preprocessing 1.1.2 is the ability to perform stateful data augmentation.

- Tensorboard:1.15.0---

Tensor Board 1.15.0 is a powerful visualization tool that is well-suited for monitoring and analyzing the training of Q-learning and Deep Q-Networks. It provides a range of visualizations, including graphs of loss and accuracy over time, as well as visualizations of the model architecture and performance metrics.

- Tensorflow:1.14.0---

TensorFlow 1.14.0 is a popular open-source machine-learning library that can be used for Q-learning and Deep Q-Networks. It provides a range of building blocks for building and training neural networks, such as layers, optimizers, and loss functions. TensorFlow also supports automatic differentiation, which is essential for backpropagation-based learning algorithms like Q-learning.

- Numpy:1.21.6---

NumPy 1.21.6 is a fundamental library for scientific computing in Python and is commonly used in Q-learning and Deep Q-Networks. NumPy also includes a range of functions for working with random numbers, which can be useful in generating the stochasticity required in reinforcement learning tasks. One of the best features of NumPy 1.21.6 is its improved performance for large-scale array computations due to optimized algorithms and caching mechanisms.

- opencv-python:4.7.0.72---

OpenCV-Python 4.7.0.72 is a versatile computer vision library that can be applied in Q-learning and Deep Q-Networks for a variety of image and video processing tasks, including object detection and segmentation [5]. One of the best features of OpenCV-Python 4.7.0.72 is its support for real-time video processing and analysis, which makes it an excellent choice for developing Q-learning and Deep Q-Network models for video-based reinforcement learning tasks.

vi. Software Solution and Evaluation:-

In the project, to train an agent, the first step taken is to create an environment that provides a 'Step' function and a 'reset' function. The step function takes an action as input and returns the resulting observation, reward, and additional info. The reset function resets the environment to an initial state or a new episode based on the done flag. Later take the collision history and decide on the reward function. This creates our agent with sensors and acceleration etc.

Generally, the Carla simulator runs in a separate process and communicates with the Python code through a client-server architecture. The client code in Python sends commands to the Carla simulator to control the self-driving car, and the simulator returns observations and rewards based on the current state of the environment. For this, we use threading to handle asynchronous interactions with Carla[1].Here we incorporate experience replay(replay memory), a technique commonly used in DQN[9], where past experiences (i.e., state, action, reward, and next state) are stored in a replay buffer, and the RL agent samples from this buffer during training to break the temporal correlation between experiences and improve learning stability[4]. The different exploration/exploitation strategies, such as epsilon-greedy and decaying epsilon-greedy, balance exploration (trying new actions), and exploitation (choosing actions with the highest Q-values) during training, which is critical for discovering optimal policies.

We have used Deep Q-Network (DQN) as the reinforcement learning algorithm to train the RL agent. The neural network takes the current state of the environment as input and outputs a set of Q-values for each possible action. These Q-values are continuous float values, and they represent the predicted expected cumulative reward for each action[3]. The action with the highest predicted Q-value is selected as the next action to be taken by the agent.To update the Q-values, the agent uses a process called "Q-learning". The Q-learning algorithm uses the observed state, action, reward, and next state to compute a target Q-value that the agent should aim for.

vii. Software Evaluation on performance

We evaluated the performance of our RL agent using the following metrics:

- Accuracy: We monitored the accuracy of the agent's actions by comparing the chosen actions with the ground truth actions or desired actions. This helped us assess the agent's ability to make accurate decisions based on the current state of the environment.
- Epsilon: We tracked the value of epsilon, which determined the exploration rate of the agent during training. Epsilon represented the probability of choosing a random action instead of exploiting the current Q-values. Monitoring epsilon helped us assess the agent's exploration strategy and how it changed over time.
- Average Reward: We calculated the average reward obtained by the agent during training episodes. This metric helped us assess the overall performance of the agent in terms of the cumulative rewards it received from the environment.
- Minimum Reward: We recorded the minimum reward obtained by the agent during training episodes. This metric helped us identify episodes or scenarios where the agent struggled and had difficulty achieving positive rewards.

- Maximum Reward: We recorded the maximum reward obtained by the agent during training episodes. This metric helped us identify episodes or scenarios where the agent performed exceptionally well and achieved high rewards.

We visualized these metrics using TensorBoard, a popular visualization tool for TensorFlow. We saved the model weights and logs during training for further analysis and comparison.

viii. Reinforcement algorithms

- **Deep Q network: -** DQN is a deep neural network-based reinforcement learning algorithm that utilizes experience replay and a target network to stabilize the learning process. It estimates the expected future rewards for each possible action in a given state using a Q-function [9]. DQN employs an epsilon-greedy policy for action selection, gradually shifting from exploration to exploitation over time. DQN is a powerful and effective algorithm for training agents in an environment, leveraging the power of neural networks to learn optimal action-selection strategies[4].
- ***Replay Buffer:*** - The Replay Buffer serves as a storage for training data in a reinforcement learning algorithm. It operates as a queue, where episodes are generated sequentially and stored at the rear of the buffer. By randomly sampling batches of transitions from the buffer, the algorithm minimizes episode correlation, as training the network on consecutive episodes could lead to overfitting. It allows the algorithm to learn from past experiences and prevents the agent from getting stuck in suboptimal actions by providing a diverse set of training data for updating the neural network[4].
- **Q-Learning: -** Q – learning is model–free reinforcement learning which does not require the model of the environment. As discussed in tutorial 1 we use the Bellman equation.

Equation 1: Bellman Equation [2]

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}$$

Advantages over other approaches: -

- The project using Q-learning and DQN allows for the customization and optimization of various aspects, such as hyperparameters, neural network architecture, and reward functions. This flexibility makes users to fine-tune the RL agent based on specific requirements and performance metrics, making it suitable for different use cases and domains.
- This helps users to understand the underlying principles and rationale behind the implementation, and basics of RL, including Q-learning, DQN, experience replay, exploration vs. exploitation, and evaluation metrics[8].
- In general, Q-learning and DQN [13] play a better part in the tutorial compared to other approaches due to their simplicity, ability to handle large state spaces, utilization of neural networks, handling of continuous state spaces, flexibility for customization, and proven track record in various applications, including self-driving cars.

Requirements: -

- Understanding of Python 3 programming including concepts such as variables, loops, conditionals, functions, and classes.
- Familiarity with Python libraries such as NumPy, Pandas, and Keras, TensorFlow may also be helpful.
- You should be familiar with the Carla Python API, which is the interface for interacting with the Carla Simulator using Python.

- Familiarity with concepts like exploration vs. exploitation, epsilon-greedy strategy, and experience replay (for DQN) would also be beneficial.
- Reinforcement learning can be a complex and iterative process, requiring experimentation, debugging, and optimization. You may need to invest time and effort in troubleshooting, tuning hyperparameters, and fine-tuning the model to achieve satisfactory results.

Tasks:

In the project we have considered, two agents, one Carla environment agent and deep Q-network (DQN)

1) We have imported the TensorFlow and Keras libraries to get the required modules like Xception, layers, optimizers, and models.
2) Later imported threading to import thread for running the training and inference processes concurrently in separate threads within the same Python process.
3) We consider the TensorFlow gpu options by configuring the config and session.
4) Created a Carla Environment agent: - we have used the help of Carla's documentation [1] to get the Carla environment, vehicle, and sensors like the front camera and collision sensors attached to the vehicle as discussed in tutorial 1.
5) In the step function, we set the reward based on the criteria of collision history and velocity. We set the throttle based on the action taken [8].

```
def set_reward(self,len_colli,velocity):
    #calculate the velocity magnitude
    velocity_mag_square = velocity.x**2 + velocity.y**2 + velocity.z**2
    #Convert into velocity to kmh
    kmh = int(3.6 * math.sqrt(velocity_mag_square))
    # if there are collissions we consider the leat reward of -200 and makes the episode as done in that epoch
    if len_colli != 0:
        flag_done = True
        reward = -200
    # if the speed is less than we consider the reward -1
    elif kmh < 50:
        flag_done = False
        reward = -1
    else:
        flag_done = False
        reward = 1
    return flag_done, reward
```

Figure 1: Code Snippet for assigning the type of rewards set.

6) After done with creating the environment agents, we worked on the DQN agent. DQN agent interacts with Carla's environment, it observes the current states and the DQN model outputs a Q- value for each possible action, the agent can take in the current state.

```
def calculate_newqvalue(self,future_qstates,index,flag_done,reward):
    # we get the future q states list , reward from the replay memory batch and based on whether done or not
    if not flag_done:
        # we calculate max value from future states and then calculate new value using bellman equation
        max_future_q = np.max(future_qstates[index])
        new_qvalue = reward + DISCOUNT * max_future_q
    else:
        new_qvalue = reward
    return new_qvalue
```

Figure 2: Code Snippet for new q value calculation using Bellman Equation.

It updates the new q value using the **Bellman equation** to minimize the difference between the predicted Q-values and the target Q-values

7) Now for training our program, we import both the Carla environment agent and DQN agent and use threading.
   - Before iterating over we initialize the first prediction with ones

```
dqn_agent.get_qstate (np. ones((carla_env.HEIGHT, carla_env.WIDTH, 3))
```

Figure 3: Code Snippet for initializing the first prediction

   - We iterate over episodes, update the tensor board at every episode, and reset the environment to the initial state. We create a model Xception [9] from the Keras application for training, the agent samples batches of experiences from the replay buffer and uses them to train the DQN model.

```
def create_model(self):
    # Create the Xception base model with random weights
    base_model = Xception(weights=None, include_top=False, input_shape=(IM_HEIGHT, IM_WIDTH,3))

    # Apply global average pooling to reduce spatial dimensions
    x = GlobalAveragePooling2D()(base_model.output)

    # Add a dense layer for predictions with linear activation function
    predictions = Dense(3, activation="linear")(x)

    # Create the final model with specified inputs and outputs
    model = Model(inputs=base_model.input, outputs=predictions)

    # Compile the model with MSE loss, Adam optimizer, and accuracy metric
    model.compile(loss="mse", optimizer=Adam(lr=0.001), metrics=["accuracy"])
```

Figure 4: Code Snippet for initializing creating model

      - Here the model is Xception from the Keras application with weights that can be initialized with random weights and not loaded with pre-trained weights [10].
      - The input shape parameter is set to (IM_HEIGHT, IM_WIDTH, 3), which defines the input shape of the images to be passed to the Xception model.
      - Global average pooling reduces the spatial dimensions of the feature maps to a single value per channel by taking the average of all values in the feature map along the spatial dimensions.
      - Dense layer with 3 units (for 3 output classes) and a linear activation function to the output of the x layer. This will produce the final predictions for the model, which will be continuous values due to the linear activation function[10].
      - For compiling the loss parameter is set to "mse", which stands for mean squared error, and is commonly used for regression tasks. The optimizer parameter is set to Adam(lr=0.001), which specifies the Adam optimizer with a learning rate of 0.001. The metrics parameter is set to ["accuracy"], which specifies that the accuracy metric should be computed during training.

   - Now we loop over until we make the done variable true to calculate action. For calculating action we use Epsilon greedy policy to get the action from Q-table.

```python
def calculate_action():
    # Generate a random number
    random_num = np.random.random()
    # Checking if the random number is greater than epsilon
    if epsilon < random_num:
        # Get action from Q table
        action = np.argmax(dqn_agent.get_qstate(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)
    return action
```

Figure 5: Code snippet of epsilon greedy policy to calculate action

- We calculate the new state, reward, and flag done from the step function .Update the replay memory for every episode
- Destroy the actor at the end of the episode and save the model.
- We calculate the minimum, maximum, and average rewards. If the minimum reward is greater than or equal to the predetermined reward, then we save the model.

```python
def calculate_reward(ep_rewards):
    # Using list slicing to extract the last AGG_ALL_STATS elements from ep_rewards
    recent_ep_rewards = ep_rewards[-AGG_ALL_STATS:]

    # Calculating the sum, length, minimum, and maximum values of recent_ep_rewards
    total_reward = sum(recent_ep_rewards)
    num_rewards = len(recent_ep_rewards)
    min_reward = min(recent_ep_rewards)
    max_reward = max(recent_ep_rewards)

    # Calculating the average reward by dividing the total reward by the number of rewards
    average_reward = total_reward / num_rewards

    return average_reward,min_reward,max_reward
```

Figure 6: Code Snippet for calculating the rewards.

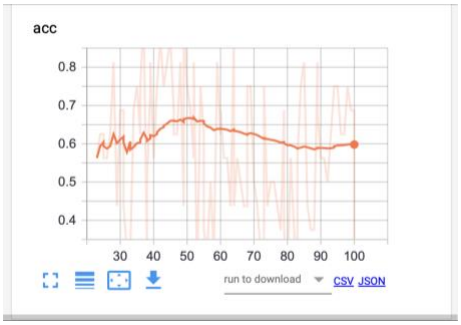
- After saving the model, we calculate the epsilon if it is greater than MIN_EPSILON.
- While training, in each episode, it calculates action using the epsilon greedy policy(exploration-exploitation strategy) and updates the replay memory.
- Set the termination flag to true or wait for the thread to finish and save model

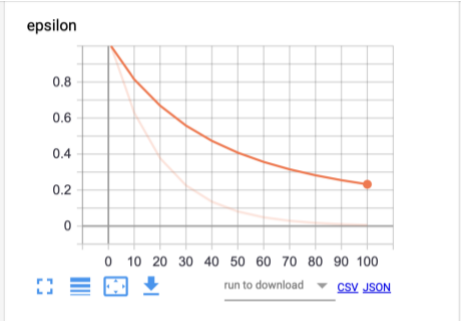
Test Cases: -

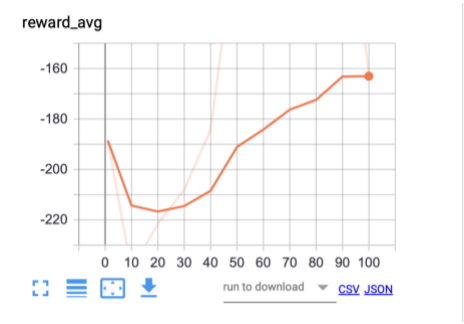
After training for 500 episodes, we saved the model and logs created. We have trained the data for 100 episodes, and 500 episodes. We have used a tensor board for visualizations for tracking accuracy and loss and rewards calculated for each episode during training.

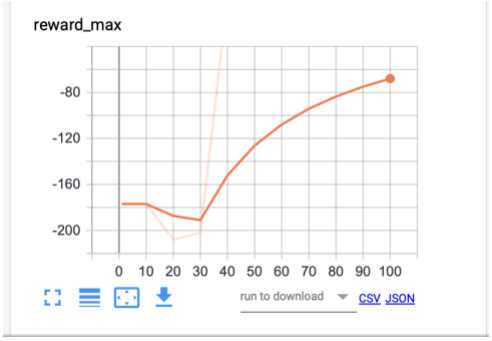
The following are the test cases of the training data:-

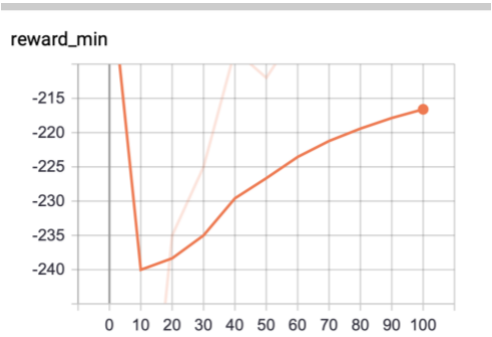
100 Episodes:-



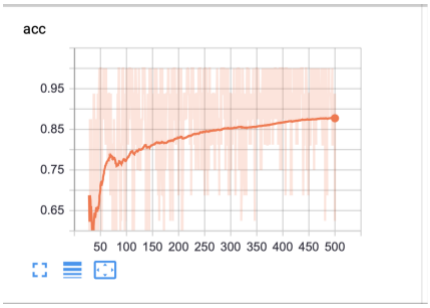


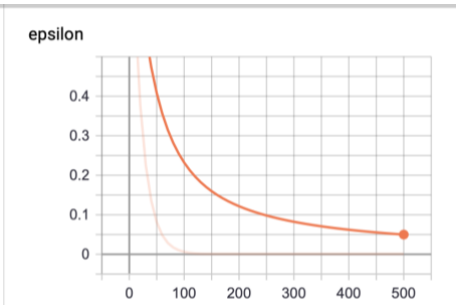


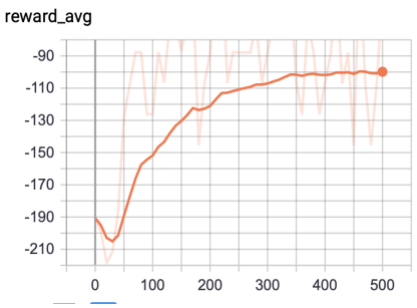


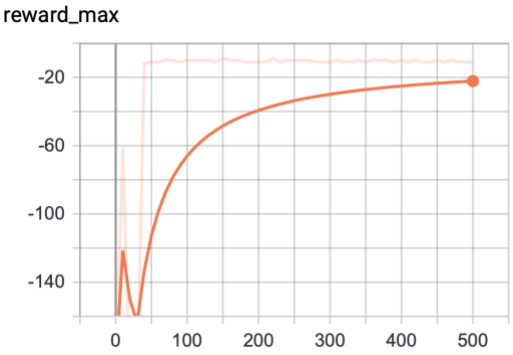


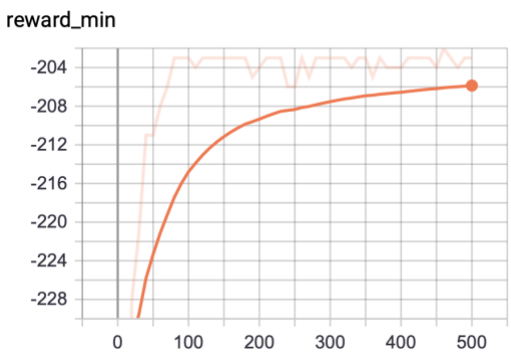
500 Episodes:-











Episodes count	Accuracy	Epsilon	Reward_avg	Reward_max	Reward_min
100	0.6	0.21	179	70	217
500	0.88	0.05	100	22	206

Table 1: Comparison of results for training for different episodes

The evaluation results show that as the number of episodes increases from 100 to 500, there is an improvement in the accuracy of the RL agent indicating that the agent is learning and improving its decision-making ability over time. With the decrease in epsilon, the agent is gradually shifting from exploration (trying new actions) to exploitation (choosing actions with the highest Q-values) as it gains more experience from rewards, the agent is learning to make better decisions and achieving higher rewards over time, but there is still some variability in the rewards obtained. Further analysis and fine-tuning may be needed to optimize the performance of the RL agent.

Learning outcomes:

After completing the tutorial, below are the learning outcomes.

- Learn how to use Python 3 and Python libraries. You will gain practical experience in implementing reinforcement learning algorithms using Python programming language.
- Learn what is Q-Learning, Bellman Equation, and its role in developing a reinforcement learning agent.
- Learn about the environment and different actions in the Carla simulator. Becoming familiar with the Carla Python API, which is a powerful tool for simulating autonomous driving scenarios.
- Understanding the basics of reinforcement learning: The project can help you gain a clear understanding of the basic concepts of reinforcement learning that include the Markov Decision Process (MDP), Q-learning, DQNs, and how they can be applied to train an autonomous agent.
- You will learn how to troubleshoot issues, fine-tune hyperparameters, and optimize the performance of the reinforcement learning model.

Exercises:

Once we have done with the tutorial, we can further enhance our understanding and skills in reinforcement learning as well as autonomous driving.

- Experiment with hyperparameter tuning: The tutorial provides default hyperparameter values, but you can experiment with different values to see how they affect the training process and the performance of the self-driving car. You can try tuning hyperparameters such as learning rate, discount factor, exploration rate, and batch size to find optimal values for your specific problem and find out which hyperparameter tunes better. [11].
- Modify the reward function: The tutorial provides a simple reward function based on the distance to the closest waypoint. You can experiment with different reward functions to see how they affect the performance of the self-driving car. For example, you can try adding penalties for collisions, rewards for staying within the lanes, or rewards for following traffic rules.
- Implement different RL algorithms: The tutorial uses a basic Q-learning algorithm and Deep Q-Networks (DQNs) for training the self-driving car. You can explore other popular RL algorithms such as Double DQNs [14], or Dueling DQNs, and implement them in the tutorial's framework. Compare their performance and understand how they differ in terms of convergence and stability.
- Implement additional sensors: The tutorial uses only a front-facing camera as an input to the RL agent. You can experiment with other types of sensors available in CARLA, such as lidar, radar, or GPS, and incorporate them into the RL agent's observation space. This can provide the agent with more information about its environment and potentially improve its decision-making capabilities [1].
- Visualization of the training process: Visualize the training process of your reinforcement learning model using various techniques, such as learning curves, loss curves, or action-value function visualization. Analyze the training progress, convergence, and stability of the model during the training process. Experiment with different visualization techniques to gain insights into how the model is learning and adapting its policy over time.
- Collect real-world driving data from actual self-driving cars or real-world driving scenarios and use this data to train and evaluate your reinforcement learning model.

Cited References:

1. Carla. (n.d.). Documentation for Windows build. Read the Docs. Retrieved March 13, 2023, from https://carla.readthedocs.io/en/latest/build_windows/

2. Wikipedia. (Date Accessed: 2023, march 04). Q-learning. Wikipedia. https://en.wikipedia.org/wiki/Q-learning

3. García Cuenca, L., Puertas, E., Fernandez Andrés, J., & Aliane, N. (2019). Autonomous Driving in Roundabout Maneuvers Using Reinforcement Learning with Q-Learning. Electronics, 8(12), 1536–1536. https://doi.org/10.3390/electronics8121536 (Date Accessed: 2023, March 30)

4. Sutton, R. S., & Barto, A. G. (2018). Reinforcement Learning: An Introduction. MIT Press. (Date Accessed: 2023, March 31)

5. OpenCV. (n.d.). OpenCV. https://opencv.org/ . (Date Accessed: 2023, March 18)

6. Keras Library https://keras.io/ (Date Accessed: 2023, April 09)

7. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A., Koltun, V., & Ferrari, V. (2017). Carla: An Open Urban Driving Simulator. In Proceedings of the 1st Annual Conference on Robot Learning (Vol. 78, pp. 1-16). (Date Accessed: 2023, April 05)

8. Carla. Code recipes. Read the Docs. https://carla.readthedocs.io/en/0.9.9/ref_code_recipes/ (Date Accessed: 2023, April 02)

9. Srivastava, A. (2021, January 7). Deep Q-Learning Tutorial. Towards Data Science. https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc (Date Accessed: 2023, April 03)

10. Chollet F. Xception: deep learning with depthwise separable convolutions. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pages 1251–1258, 2017.

11. Hyperparameter tuning https://towardsdatascience.com/experiments-on-hyperparameter-tuning-in-deep-learning-rules-to-follow-efe6a5bb60af Accessed April 11, 2023.

12. Towards Data Science. https://towardsdatascience.com/review-xception-with-depthwise-separable-convolution-better-than-inception-v3-image-dc967dd42568. Accessed April 06, 2023.

13. Yang, H., Guo, Y., & Song, L. (2020). Unsupervised discovery of interpretable directions in the GAN latent space. In International Conference on Machine Learning (ICML) (Vol. 120, pp. 10686-10695). Retrieved from https://proceedings.mlr.press/v120/yang20a.html Date accessed: 2023-04-14.

14. Hinton, G. E., & Salakhutdinov, R. R. (2010). Reducing the dimensionality of data with neural networks. In Proceedings of the Science Conference on Neural Information Processing Systems (NeurIPS) (pp. 1-9). from https://proceedings.neurips.cc/paper/2010/hash/091d584fced301b442654dd8c23b3fc9-Abstract.html Date accessed: 2023-04-06