CS 5392 Spring 2023 Assignment-1 Tutorial-3

Title: -

Reinforcement Learning for Self-Driving Cars: Training and Improvement of 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.
- In this tutorial, we have taken the reference from previous research papers to improve the accuracy, and reward system of our ongoing project. We have tried various procedures of reward assigning and deep learning models, integrated them into our project, and ran test cases by increasing epochs.

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. We are using Carla (Car Learning to Act) an open-source autonomous[7] driving simulator the Computer Vision Center developed. This simulator provides a realistic 3D environment for testing and training autonomous driving algorithms. 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].

ii. 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.

iii. Installation Requirements: -

- Install Visual Studio 2019 or later with C++ support, Git, CMake, file compression software(7Zip), and Make.
- Install Python 3.7 or later and the Unreal Engine prerequisites (Visual Studio Build Tools, DirectX, Windows 10 SDK)

- Clone the Carla repository from GitHub and configure Carla using CMake.
- Build the Carla client and server executables using Visual Studio Run the Carla server and launch a client to test the installation.

iv. Libraries installed: -

Libraries	Description
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].
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.
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].
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.
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.
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.
opencv-python:4.7.0	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.

v. Referred Research Papers Summary:-

a) Deep reinforcement learning-based control for Autonomous Vehicles in CARLA [15] focuses on deep reinforcement learning (DRL) based control approach for autonomous vehicles in the CARLA simulator. It proposes a novel architecture that combines a DRL algorithm with a convolutional neural network (CNN) to learn an end-to-end driving policy. Here we have taken the reference of using the CNN model to train our dqn agent. We have used the reward function used in this paper for our RL agent and later trained on both CNN and Xception. We have evaluated the performance metrics.
b) A Deep Q-Network Reinforcement Learning-Based Model for Autonomous Driving [16] proposes a model for autonomous driving based on Deep Q-Network (DQN) reinforcement learning. The DQN algorithm is a deep reinforcement learning technique that combines Q-learning with deep neural networks, enabling the agent to learn from high-dimensional input data, such as images from cameras or LIDAR scans. It uses five convolutional layers with 32, 64, 64, 128, and 128 neurons in layers 1 through 5, respectively. All layer’s use (3 x 3) kernel size and stride zero. Each convolutional layer is followed by a ReLU activation function. We have used the same CNN 5 layered model to train our dqn agent and evaluated the performance.
c) Review of deep learning: concepts, CNN architectures, challenges, applications, and future directions:- The paper outlines the importance of DL, presents various DL techniques and networks,

and focuses on convolutional neural networks (CNNs) which are the most commonly used DL network type. It describes the development of CNN architectures along with their main features, starting with the AlexNet network and closing with the High-Resolution network (HR.Net). Moreover helped us in understanding CNN architecture and in the training of our model [2].

d) Xception: Deep Learning with Depth-wise Separable Convolutions[12]: - Here it proposes an interpretation of Inception [10] modules in convolutional neural networks as an intermediate step between regular convolution and depth-wise separable convolution (a depth-wise convolution followed by a pointwise convolution). Here it helped us in understanding Inception architecture and in the training of our model.

e) Analysis of Reinforcement Learning in Autonomous Vehicles[5] :- This paper has given an idea of how to calculate reward based on different criteria like velocity, speed, and collision detection, Moreover gave a piece of brief information on different reinforcement models used by top companies. We have used this paper for modifying the reward function, trying different variants to improve the performance metrics.

vi. Software Solution and Evaluation: -

In the previous part of our project, the first step taken is to create a carla 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. Here we have updated our reward function based on a research paper [15] where the reward is considered as 100 if there are no collisions along with a speed less than 50.

For training our DQN agent, we have considered deep learning models like Xception, 5 layered CNN (Convolutional Neural Network)[16]. We have trained for 100 and 500 episodes using our reward system and modified reward system and calculated the metrics based on accuracy, loss, and rewards.

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[13].
- 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.
- 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.
- 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 Tensor Board, 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].

- ***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.[4].
- **Q-Learning: -** Q – learning is model–free reinforcement learning[3] which does not require the model of the environment. As discussed in tutorial 1 we use the Bellman equation.

Equation 1: Bellman Equation

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

- **Deep Learning architectures**

    - **Xception**:- Xception is based on the Inception architecture but introduces a novel concept called depthwise separable convolutions. These convolutions replace the traditional convolution operations used in Inception, resulting in a more efficient and accurate model. Depth wise separable convolutions involve two operations: depth wise convolutions, which apply a single filter per input channel, followed by pointwise convolutions, which combine the outputs of the depth wise convolution using 1x1 filters. This approach reduces the number of parameters and computations, making Xception more efficient and scalable compared to traditional CNNs[12].

    - **CNN** :- A CNN is a type of deep learning model specifically designed for processing grid-like data, such as images. CNNs consist of multiple layers, including convolutional, pooling, and fully connected layers. The convolutional layers are responsible for detecting local features in an image using filters, while pooling layers reduce the spatial dimensions of the feature maps, making the network less sensitive to small shifts in the input.

Requirements: -

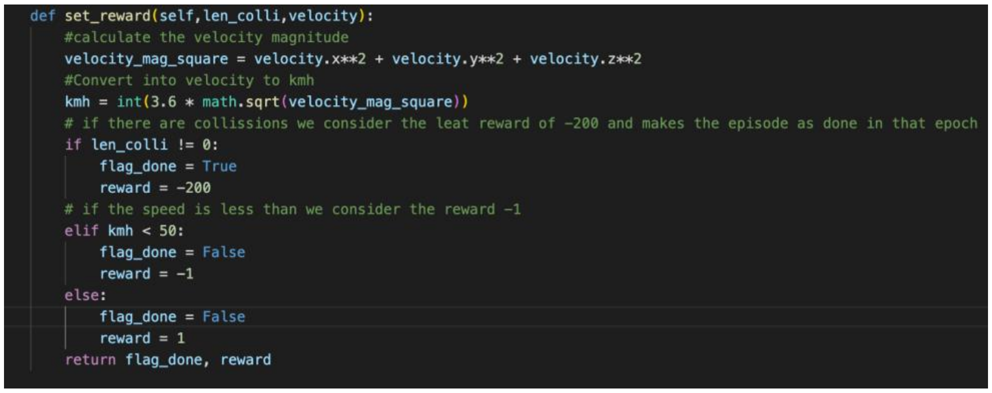
- Understanding of Python 3 programming including concepts such as variables, loops, conditionals, functions, and classes.
- Familiarity with Python libraries such as NumPy, Pandas, and Keras, TensorFlow may also be helpful.
- You should be familiar with the Carla Python API, which is the interface for 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. Moreover, have a learning curve on deep learning models will be essential.

Tasks:

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

1) We have imported the TensorFlow and Keras libraries to get the required modules like Xception, CNN, 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.

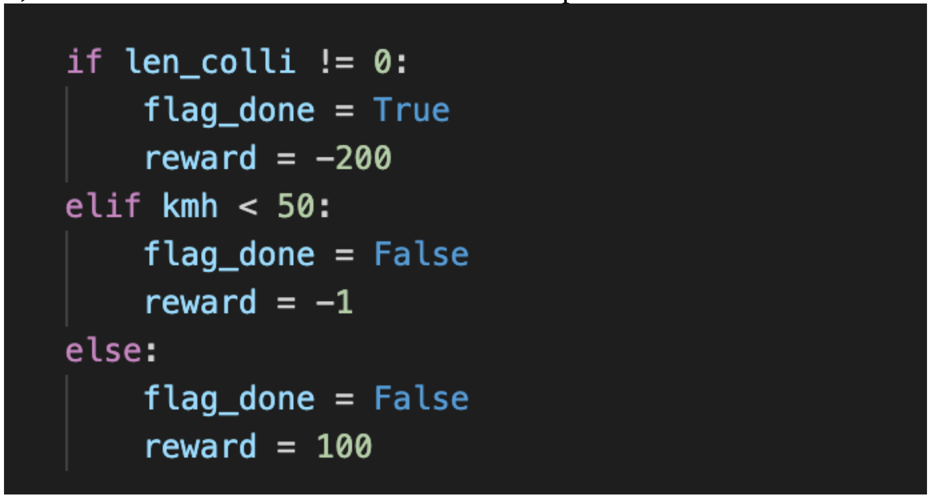
5) In the step function, we set the reward based on the criteria of collision history and velocity. We set the throttle based on the action taken [8].



```python
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) Here we used the research paper[15] reward system criteria to improve our model. Instead of taking reward 1, we considered 100 as them and then compared the results.

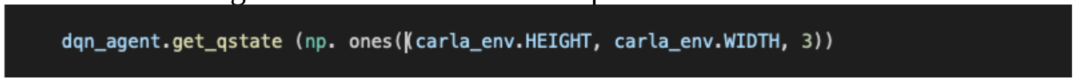


```python
if len_colli != 0:
    flag_done = True
    reward = -200
elif kmh < 50:
    flag_done = False
    reward = -1
else:
    flag_done = False
    reward = 100
```

Figure 2: Code Snippet for the modified rewards set based on the research paper.

7) Now for training our program, we import both the Carla environment agent and DQN agent and use threading.

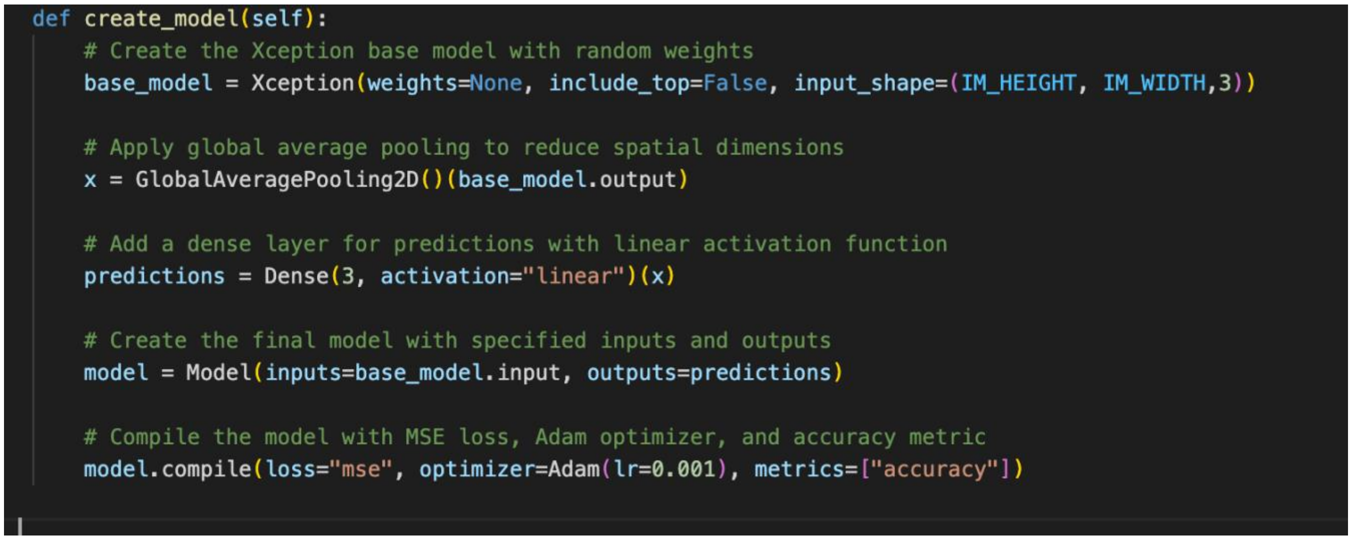
- Before iterating over we initialize the first prediction with ones



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

Figure 3: Code Snippet for initializing the first prediction

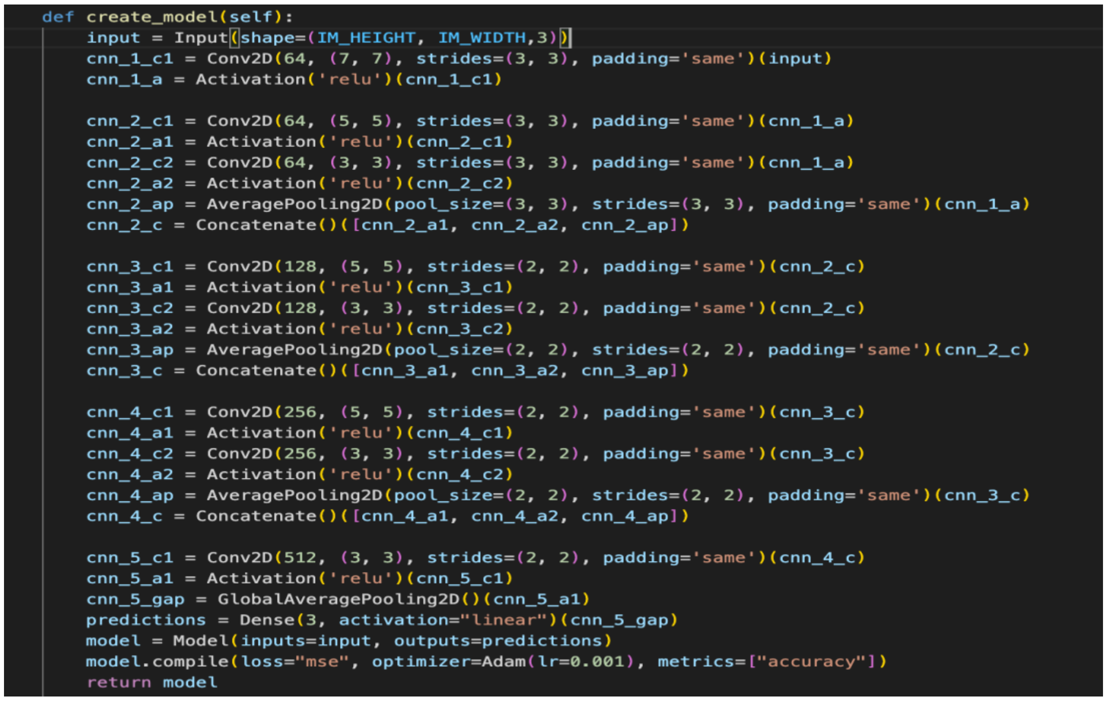
- We iterate over episodes, update the tensor board at every episode, and reset the environment to the initial state. We create a model Xception [9] from the Keras application for training, the agent samples batches of experiences from the replay buffer and uses them to train the DQN model.



```
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 a model using Xception

Along with this model, we considered the Convolutional neural network (CNN) layered network from the research paper[16] and applied this model as our base model to our project and calculated the performance metrics for 100 and 500 epochs. Modified the reward system similar to hyperparameter tuning and evaluated the metrics.



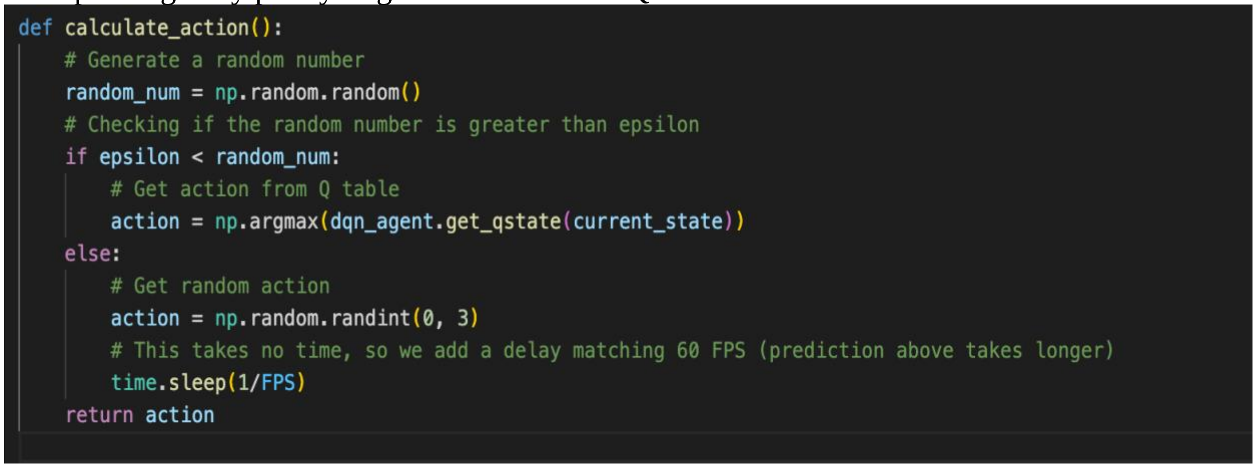
```
def create_model(self):
    input = Input(shape=(IM_HEIGHT, IM_WIDTH,3))
    cnn_1_c1 = Conv2D(64, (7, 7), strides=(3, 3), padding='same')(input)
    cnn_1_a = Activation('relu')(cnn_1_c1)

    cnn_2_c1 = Conv2D(64, (5, 5), strides=(3, 3), padding='same')(cnn_1_a)
    cnn_2_a1 = Activation('relu')(cnn_2_c1)
    cnn_2_c2 = Conv2D(64, (3, 3), strides=(3, 3), padding='same')(cnn_1_a)
    cnn_2_a2 = Activation('relu')(cnn_2_c2)
    cnn_2_ap = AveragePooling2D(pool_size=(3, 3), strides=(3, 3), padding='same')(cnn_1_a)
    cnn_2_c = Concatenate()([cnn_2_a1, cnn_2_a2, cnn_2_ap])

    cnn_3_c1 = Conv2D(128, (5, 5), strides=(2, 2), padding='same')(cnn_2_c)
    cnn_3_a1 = Activation('relu')(cnn_3_c1)
    cnn_3_c2 = Conv2D(128, (3, 3), strides=(2, 2), padding='same')(cnn_2_c)
    cnn_3_a2 = Activation('relu')(cnn_3_c2)
    cnn_3_ap = AveragePooling2D(pool_size=(2, 2), strides=(2, 2), padding='same')(cnn_2_c)
    cnn_3_c = Concatenate()([cnn_3_a1, cnn_3_a2, cnn_3_ap])

    cnn_4_c1 = Conv2D(256, (5, 5), strides=(2, 2), padding='same')(cnn_3_c)
    cnn_4_a1 = Activation('relu')(cnn_4_c1)
    cnn_4_c2 = Conv2D(256, (3, 3), strides=(2, 2), padding='same')(cnn_3_c)
    cnn_4_a2 = Activation('relu')(cnn_4_c2)
    cnn_4_ap = AveragePooling2D(pool_size=(2, 2), strides=(2, 2), padding='same')(cnn_3_c)
    cnn_4_c = Concatenate()([cnn_4_a1, cnn_4_a2, cnn_4_ap])

    cnn_5_c1 = Conv2D(512, (3, 3), strides=(2, 2), padding='same')(cnn_4_c)
    cnn_5_a1 = Activation('relu')(cnn_5_c1)
    cnn_5_gap = GlobalAveragePooling2D()(cnn_5_a1)
    predictions = Dense(3, activation="linear")(cnn_5_gap)
    model = Model(inputs=input, outputs=predictions)
    model.compile(loss="mse", optimizer=Adam(lr=0.001), metrics=["accuracy"])
    return model
```

Figure5: Code Snippet for initializing creating a model for 5 layered CNN

- Here, we have taken the input size and assigned our height and width. Later we have extracted higher-level features from the input image.
- The first line creates a convolutional layer (Conv2D) with 64 filters of size 5x5, a stride of 3x3, and padding set to 'same'. The input to this layer is a tensor called cnn_1_a.
- The second line applies the Rectified Linear Unit (ReLU) activation function to the output of the first convolutional layer.
- The third line creates a second convolutional layer with 64 filters of size 3x3, a stride of 3x3, and padding set to 'same'. The input to this layer is also cnn_1_a.
- The fourth line applies the ReLU activation function to the output of the second convolutional layer.

- The fifth line applies average pooling to the input tensor cnn_1_a with a pool size of 3x3, a stride of 3x3, and padding set to 'same'.
- Finally, the output tensors of the two convolutional layers and the average pooling layer are concatenated using the Concatenate() function along the depth dimension. The resulting tensor is called cnn_2_c and is the output of this CNN block.Repeated this for 5 layers and later compiled it.

- 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.



```
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
```

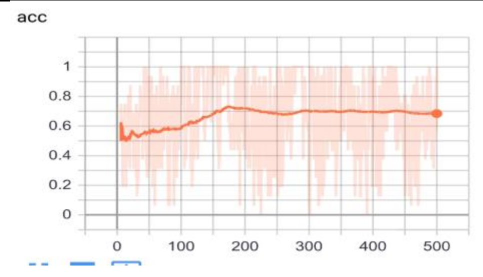
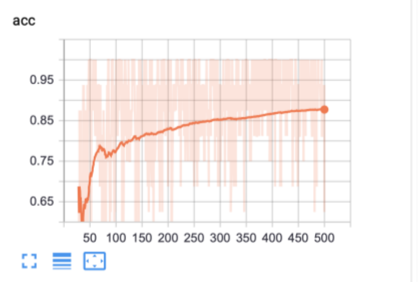
Figure 6: Code snippet of epsilon greedy policy to calculate action

- 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.
- 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: -

We have considered two models both Xception and CNN with two types of reward systems for training our DQN agent. 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:-

We have trained for 500 episodes each:-

CNN model	**Xception Model**
	

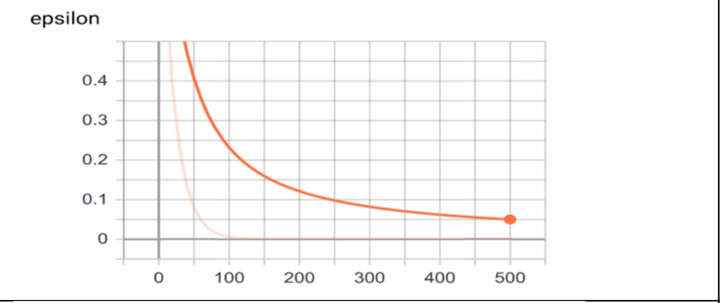
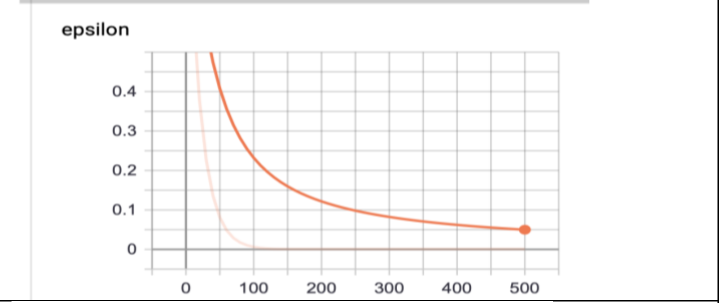
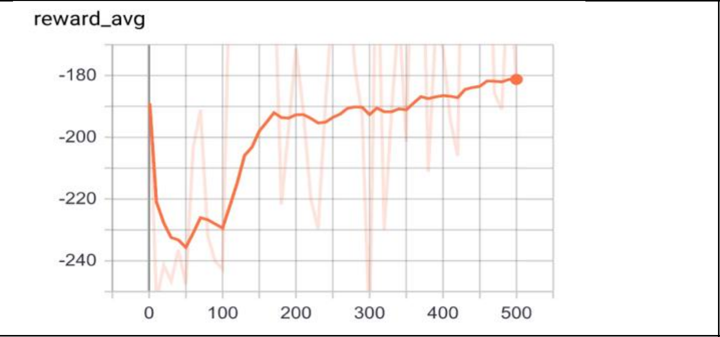
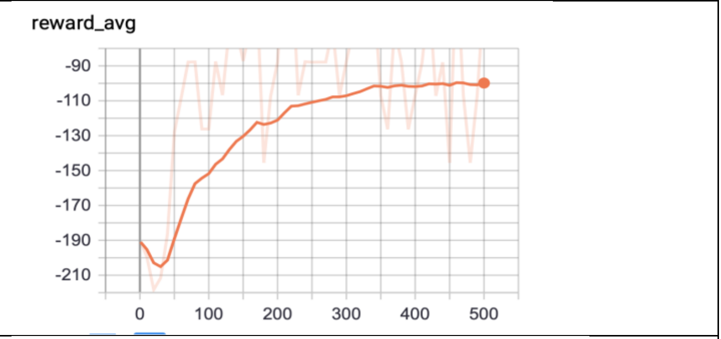
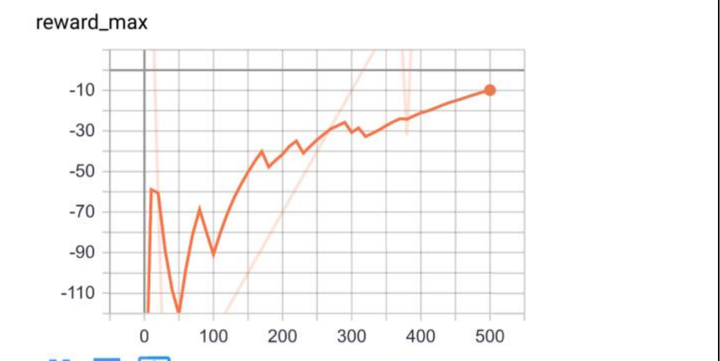
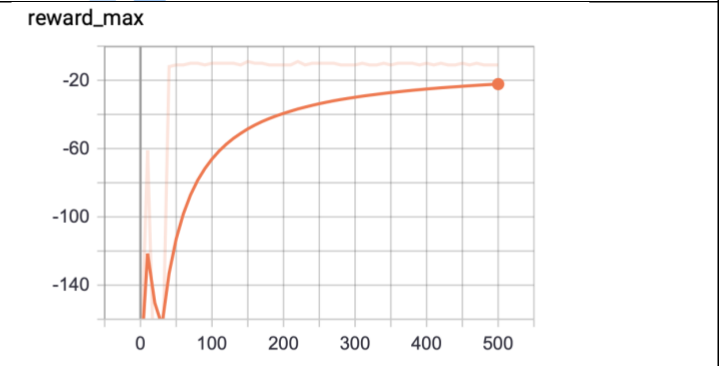
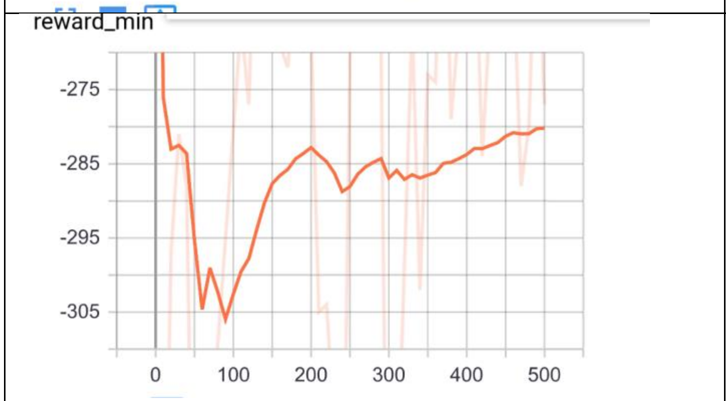
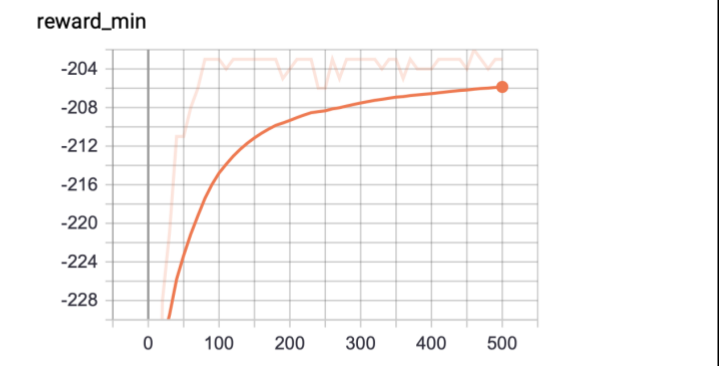
	
	
	
	

Table 1: Visualization of the Tensor board for Comparing of CNN model and Xception trained for 500 episodes

Here in the above two cases, we ran the two models for 500 epochs for both the Xception and CNN. The accuracy of Xception is far better compared to CNN but the time taken for training is much faster in CNN compared to Xception. Moreover, the reward has little difference in both models. But to train our model Xception is the best model.

Learning outcomes:

After completing the tutorial, below are the learning outcomes.

- Gain practical experience in Python 3 and Python libraries.
- Learn what is Q-Learning, Bellman Equation and different deep learning agents.

- 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.
- Gain a clear understanding of the basic concepts of reinforcement learning, including the Markov Decision Process, Q-learning, and Deep Q-Networks, 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].
- Implement multi-agent RL: Multi-agent RL involves learning a policy for multiple agents that interact with each other. You can try implementing multi-agent RL algorithms such as MADDPG or COMA and apply them to scenarios where multiple autonomous vehicles are operating in the same environment[5].
- 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].
- Implement exploration strategies: Exploration strategies involve balancing the agent's exploration of the environment with its exploitation of the learned policy. You can try implementing exploration strategies such as Boltzmann exploration, or optimistic initialization and compare their performance to pure exploitation strategies like greedy policies.

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. Alzubaidi, L., Zhang, J., Humaidi, A.J. *et al.* Review of deep learning: concepts, CNN architectures, challenges, applications, future directions. *J Big Data* **8**, 53 (2021). https://doi.org/10.1186/s40537-021-00444-8García Cuenca, L., Puertas, E., Fernandez Andrés, J., & Aliane, N. (2019). (Date Accessed: 2023, April 15)

3. 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. E. Jebessa, K. Olana, K. Getachew, S. Isteefanos and T. K. Mohd, "Analysis of Reinforcement Learning in Autonomous Vehicles," *2022 IEEE 12th Annual Computing and Communication Workshop and*

Conference (CCWC), Las Vegas, NV, USA, 2022, pp. 0087-0091, doi: 10.1109/CCWC54503.2022.9720883. (Date Accessed: 2023, April 20)

6. PythonProgramming.net:https://www.youtube.com/watch?v=J1F32aVSYaU&list=PLQVvvaa0QuDeI12McNQdnTlWz9XlCa0uo&ab_channel=sentdex (Date Accessed: 2023, April 02)

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. (Date Accessed: 2023, April 18)

11. Hyperparameter tuning https://towardsdatascience.com/experiments-on-hyperparameter-tuning-in-deep-learning-rules-to-follow-efe6a5bb60af (Date 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 April 14, 2023)

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 April 06, 2023)

15. Pérez-Gil, Ó., Barea, R., López-Guillén, E. *et al.* Deep reinforcement learning based control for Autonomous Vehicles in CARLA. *Multimed Tools Appl* **81**, 3553–3576 (2022). https://doi.org/10.1007/s11042-021-11437-3 (Date Accessed April 11, 2023)

16. M. Ahmed, C. P. Lim and S. Nahavandi, "A Deep Q-Network Reinforcement Learning-Based Model for Autonomous Driving," *2021 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Melbourne, Australia, 2021, pp. 739-744, doi: 10.1109/SMC52423.2021.9658892. (Date Accessed April 18, 2023)