Reinforcement Learning for Car Racing in OpenAI Gym using Double Deep Q-Learning (DDQN)

Qi Jiachen
2021533125
qijch@shanghaitech.edu.cn

Shi Junyan
2021533154
shijy@shanghaitech.edu.cn

Xu Chenhao
2021533119
xuchh@shanghaitech.edu.cn

Zhao Zeyu
2021533136
zhaozy2@shanghaitech.edu.cn

Zhu Yuhang
2021533173
zhuyh3@shanghaitech.edu.cn

Abstract

Car racing games hold significant practical importance and present a series of challenging technical problems. This final project endeavors to construct a model utilizing reinforcement learning techniques, specifically Double Q-learning (DDQN), within the OpenAI Gym environment. We establish a reward system and conduct experiments involving image processing and model architectures.

GitHub Repository: https://github.com/Sleepless9/CS181-2023fall

1 Introduction

Car racing games can be simulated to replicate real-world scenarios, allowing the definition of various constraints and goals. The original OpenAI Gym's CarRacing problem aimed for the car to traverse all track tiles, rather than completing a full lap. However, this implies that if the car misses some tiles, it will need to run at least one more lap. In this project, we modify the end condition to expedite training and testing, concluding an epoch if the car cannot traverse all tiles and fails to accrue new points within a specified frame limit. This unique end condition aims to encourage the agent to consistently progress on the road, swiftly concluding when yawning—a feature potentially beneficial in the realm of autonomous driving.

2 Problem Analysis

2.1 Environment

In our investigation, we have opted to utilize the `CarRacing-v0` version environment provided by OpenAI Gym to train our autonomous car agent. The track layout is dynamically generated at the commencement of each simulation, denoted as an "episode". Each state is represented by a 96×96 RGB pixel frame, serving as the sole perceptible information for the agent. This frame offers a top-down perspective of the surroundings, encompassing the current position of the car.

Between consecutive frames, the car is presented with the decision to take an action or maintain its current state. The action space is defined as a combination of steering direction, acceleration (gas),

Preprint. Under review.

and braking. The numerical values for each component of the action space range between -1 and 1. The complete set of actions is enumerated below:

Table 1: Action Space

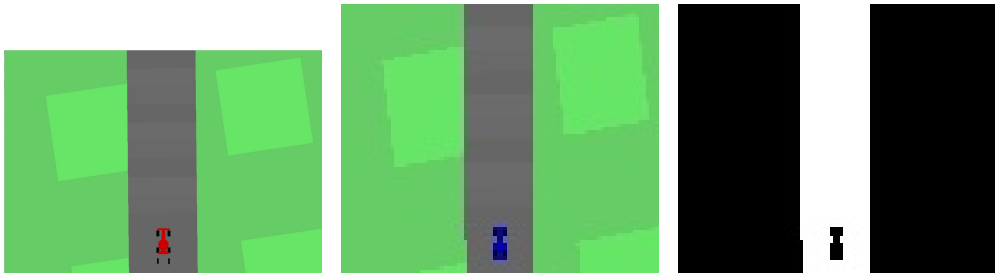
(-1, 1, 0.5)	(0, 1, 0.5)	(1, 1, 0.5)
(-1, 1, 0)	(0, 1, 0)	(1, 1, 0)
(-1, 0, 0.5)	(0, 0, 0.5)	(1, 0, 0.5)
(-1, 0, 0)	(0, 0, 0)	(1, 0, 0)

2.2 Grayscale Convertion

How to perform environment recognition to assist the agent in understanding the environment? This problem can lead to many solutions and be applied to the field of autonomous driving.

We choose to do grayscale convertion to help the agent learn about the environment from observation. Such a solution is enough to help us complete the recognition in the box2d environment, and will not produce too much calculation to drag down the training of the model.

Figure 1: Grayscale Convertion Process



(a) Observed Image (b) RGB Image (c) Grayscale Image

3 Related Model

3.1 Double Deep Q Learning (DDQN) Overview

The conventional Q-learning algorithm often encounters the challenge of overestimating Q-values. Following our team's initial evaluation of Q-learning's efficacy, we introduced the Double Deep Q Learning (DDQN) algorithm, particularly tailored for addressing challenges in dynamic environments like car racing games. DDQN extends the original DQN algorithm by incorporating two neural networks (the main network and the target network) to alleviate the issue of Q-value overestimation.

In the update process, DDQN leverages the target network to compute the Q-value for the subsequent state s' and subsequently selects the optimal action using the main network. This approach plays a crucial role in diminishing instability arising from overestimation, a critical consideration in the context of car racing scenarios. DDQN seamlessly integrates the updating mechanism of Q-learning but employs deep neural networks to estimate Q-values, thereby enhancing its adaptability to challenges in complex environments. The inclusion of a dual-network structure further bolsters the stability of the learning process, making DDQN particularly well-suited for tasks in our navigating car racing environments.

3.2 Double Q-learning

We initialize the Q-learning algorithm by setting parameters related to the environment and action space.

3.2.1 Experience Replay (`experience`)

When the number of experiences in the experience pool surpasses a certain threshold, the experience replay mechanism comes into play. This involves:

- `store_transition`: Storing information about the state, action, reward, next state, and whether it's done for each time step into an experience replay buffer.
- `get_batch`: Once the observed tuple count surpasses a predefined minimum, the model undergoes iterative improvement by sampling a batch of historical experience tuples from the buffer. This batch is then employed to train the model weights in each iteration.
- For each experience sample, calculating the target model's Q-value and updating the Main Model's Q-value:

$$Q_{main}[a] = \begin{cases} R & \text{if the episode is done} \\ R + \gamma \max_{a'} Q_{target}(s', a'; \theta^-) & \text{otherwise} \end{cases}$$

  Here, θ^- represents the parameters of the target network.
- `update_model`: Updating the model by copying the parameters of the neural network `self.model` to the target network `self.target_model`.

3.2.2 Choosing an Action (`choose_action`)

In the context of our exploration strategy, the selection of an action a_t given the current state s_t involves utilizing the neural network `self.model` to predict Q-values for each action in the current state. The decision-making process adheres to an epsilon-greedy policy, where the action with the maximum Q-value is chosen with a probability of $1 - \epsilon$, and a random action is selected with a probability of ϵ. The action a_t is determined by:

$$a_t = \begin{cases} \arg\max_a Q(s_t, a; \theta) & \text{with probability } 1 - \epsilon \\ \text{random action} & \text{with probability } \epsilon \end{cases}$$

To elaborate on our specific setup, our ϵ value is initially set to 1.0 during the early stages of training. After each experience, ϵ undergoes exponential decay, diminishing with a decay factor of 0.9999. This decay process continues until ϵ reaches a minimum value of 0.05. This dynamic adjustment of ϵ serves as a mechanism for balancing exploration and exploitation throughout the training iterations.

3.3 Neural Network Architecture

The core of our Double Deep Q Learning (DDQN) model resides in its Convolutional Neural Network (CNN). This network is designed to process raw state representations, extracting essential features and translating them into meaningful Q-value estimates. The architecture is summarized in the table below:

Table 2: CNN Architecture

Layer	Operation	Parameters
Conv1	`ReLU(Conv2D)`	Filters=6, Kernel=(7,7), Strides=3
MaxPool1	`MaxPooling2D`	Pool Size=(2,2)
Conv2	`ReLU(Conv2D)`	Filters=12, Kernel=(4,4)
MaxPool2	`MaxPooling2D`	Pool Size=(2,2)
Flatten	`Flatten`	-
Dense1	`ReLU(Dense)`	Units=216
Output	`Dense`	Units=12, Activation="linear"

Given the inherent strengths of CNN in image processing tasks, we deliberately chose this architecture as the foundation of our DDQN. The inclusion of multiple layers enables the network to capture

hierarchical features, enhancing its ability to discern intricate patterns within the input data. This architectural choice, coupled with the model's capacity for learning intricate spatial representations, suggests promising training outcomes. The potential effectiveness of the model's training process can be influenced by the depth of the CNN, allowing it to grasp both low-level and high-level features essential for making informed decisions in the dynamic environment of the car racing game.

During the compilation phase, we employ Huber Loss as the loss function and the Adam optimizer, which further enhances the model's effectiveness in learning Q-values . This strategic combination emphasizes the importance of precise Q-value estimation throughout the DDQN training process.

4 Approach

4.1 Grayscale Convertion

Get a 96×96 RGB three-color image from the game. Multiply the y-axis by 0.85 to remove the following scores and the status bar of the vehicle to get a complete picture, i.e. reduce the number of pixel blocks to be processed. Use cv2 to use color segmentation to convert roads into white, lawns and vehicles into black.

4.2 Training

We set `episode_num` times to loop, and run the game once in each loop.If the game is over or the road cannot be recognized in the image or the totalreward is less than the minimum value or no positive reward is obtained for 100consecutive frames (that is, no blocks are eaten, it is purely a time deduction),the loop ends. Each time, the obtained state is converted from the color stateto a grayscale image, and `road_visibility` is obtained from it to ensure thatthe road can be recognized in the image and avoid meaningless training sets.From the obtained grayscale image, the agent finds the optimal action correspondingto the image in the model, and uses the action to run the game to obtain thenext state. The `skip_frames` method is used to introduce time continuity andprovide the agent with more information for decision-making.At the same time, we process the reward obtained and set an upper limit for it. Becausewe don't want it to run too fast, so that it will be easily thrown away when turning.Therefore, no matter how fast the car runs, it can only get so many rewards at most, andgoing too fast will result in a low score, which will drive the car to drive rationally.Then convert the resulting new state from the color state to a grayscale image,and update `road_visibility`.After that, the old and new status, action, reward, and whether it is completed are storedin the agent, and experience playback is performed. Finally, reward is added to the total.The process would update the agent's action_value network every 5 cycles.

4.3 Test

This part is to load the model from model to agent. Same as training part,We set `test_num` times to loop, and run the game once in each loop.If the game ends or the total reward is less than the minimum value or no positivereward is obtained for 300 consecutive frames (that is, no tiless have been traversed, it ispurely a time deduction), the loop ends. Then convert the obtained state fromcolor state to grayscale image each time. From the obtained grayscale image,the agent finds the optimal action corresponding to the image in the model, and usesthe action to run the game to obtain the next state. The `skip_frames`method is usedto introduce time continuity and provide the agent with more information for decision-making.After that, the old and new status, action, reward, and whether it is completed are storedin the agent, and experience playback is performed. Later, reward is added to the total. Finally, calculate the maximum, minimum, average, and variance of the reward obtained, and store the data in the ".csv" file.

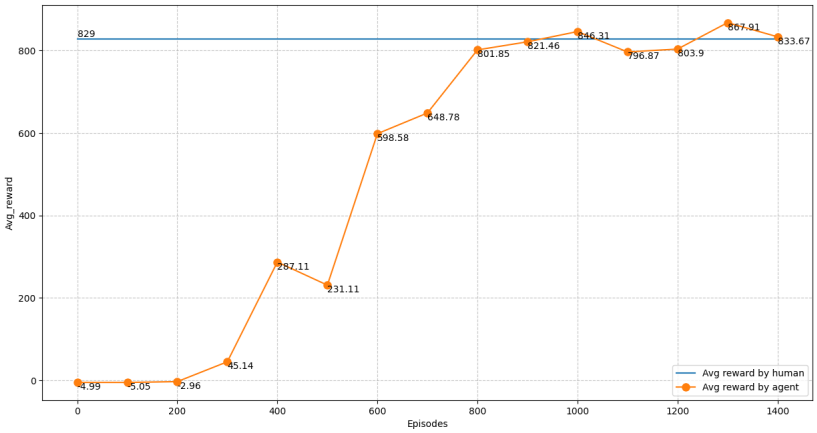


Figure 2: Average Rewards

References

[1] andywu0913 (2020). OpenAI-GYM-CarRacing-DQN. GitHub Repository. Retrieved from `https://github.com/andywu0913/OpenAI-GYM-CarRacing-DQN`

[2] Gym Library. Car Racing - Gym Document. Retrieved from `https://www.gymlibrary.dev/environments/box2d/car_racing/`

[3] van Hasselt, H., Guez, A., & Silver, D. (2015). Deep Reinforcement Learning with Double Q-learning. Retrieved from `https://arxiv.org/abs/1509.06461`

[4] Changmao Li (2019). Challenging On Car Racing Problem from OpenAI Gym. Retrieved from `https://arxiv.org/pdf/1911.04868.pdf`