Deep Reinforcement Learning Applications on Autonomous vehicles

Sagar Nandeshwar

Student ID: 260920948

Abstract

In the project, I have built two different deep reinforcement learning algorithms to explore OpenAI Gym CarRacing environment in order to understand the complexity behind Automated vehicles. Additionally, I have implemented several techniques such as image processing, discrete rewards system, varying hyperparameters and minibatch training to ease the training process.

1 Introduction

Automated vehicles or self-driving cars were once seen as a distant future but are now becoming a reality. These advancements have numerous benefits, however, developing such technologies are not easy. In order to learn and understand the complexity behind such advanced technologies, I have implemented two deep reinforcement learning models - Deep Q Learning and Double Deep Q learning - to control direction, speed and brakes of the race car in a OpenAI Gym CarRacing environment.

2 Background

2.1 Environment (OpenAI Gym's CarRacing-v0)

It is essentially an F1 racing track. Each state of environment is a 96 x 96 RGB pixel frame that represents a top-down view of a car and its surroundings i.e., track and field. The Action Space has three components to control direction, gas and brakes. The reward is -0.1 every frame and +1000/N for every track tile visited, where N is the total number of tiles visited in the track.

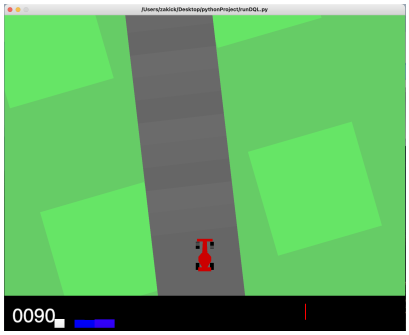


Figure 1: Race Track or State Observation

37th Conference on Neural Information Processing Systems (NeurIPS 2023).

2.2 Simple Q learning

$Q - learning$[1] is a way for an agent to learn how to make decisions based on feedback it receives from its environment. The Q-learning algorithm works by maintaining a table of values for each state-action pair, called the Q-table. The values in the Q-table represent the expected future reward that the agent can receive by taking a particular action in a particular state. The agent uses the Q-table to determine which action to take in each state.

$$q_*(s,a) = E\left[R_{t+1} + \gamma \max_{a'} q_*\left(s', a'\right)\right]$$

Figure 2: bellman equation

2.3 Deep Q learning (DQL)

$Deep\ Q - learning$[2] is a variant of Q-learning that uses a deep neural network as a function approximator to learn the Q-function. In deep Q-learning, a deep neural network is used to estimate the Q-values for each state-action pair. The neural network takes the state as input and outputs a Q-value for each possible action. The neural network is trained using the Q-learning algorithm, where the target Q-value is computed using the Bellman equation and the experience replay technique is used to break the correlations between consecutive samples.

$$Q(S_t, A_t) = (1 - \alpha)\, Q(S_t, A_t) + \alpha * (R_t + \lambda * max_a\, Q(S_{t+1}, a))$$

Figure 3: bellman equation

2.4 Double Deep Q learning (DDQL)

$Double\ Deep\ Q - Networks$[3] (DDQN) is a variant of deep Q-learning that addresses the problem of overestimation of Q-values that can occur in standard Q-learning and other Q-learning variants. In standard Q-learning, the Q-values can be overestimated because the maximum Q-value for the next state is used in the update rule, which may not always be accurate, especially in noisy or complex environments. DDQN uses a two network, called the target networks, to estimate the target Q-values for the update rule. The target network is a copy of the main network, which is periodically updated to match the parameters of the main network. The target network is used to estimate the Q-value for the next state and action, while the main network is used to select the action to take in the current state. By decoupling the selection of the action from the estimation of the target Q-value, DDQN is less prone to overestimation of Q-values, which can lead to more stable and accurate learning

$$\theta' \leftarrow \tau\theta + (1 - \tau)\theta'$$

Figure 4: Updating target model

3 Methodology

3.1 Goal

The goal is to make the car stay on track as long as possible.

3.2 Deep Q learning (DQL)

3.2.1 Architecture

Loss function: Mean Squared Error Optimization: Adam with Learning rate = 0.001 and epsilon=1e-7

Layers	Details
Convolution	6 filters of size 7x7, stride of 3, and RELU activation
Max Pooling	pool size of 2x2
Convolution	12 filters of size 4x4, and RELU activation
Max Pooling	pool size of 2x2
Flatten	-
Dense	size 216
Dense	size Action space

3.2.2 Algorithm

For each episode, the model stacks three consecutive steps taken with the same action as a single state. This state stack is then converted into grey scale image and fed into the DQL model. The model then predicts q values of all the action in action space for this state stack. It then selects the action based on Greedy-Epsilon policy. It then records the observation (current state, action, next state, done) which is used during mini batch training. This process is repeated until the episode has been terminated.

3.3 Double Deep Q learning (DDQL)

3.4 Architecture

Loss function: Mean Squared Error Optimization: Adam wirh Learning rate = 0.00001

Layers	Details
Convolution	64 filters of size 8x8, stride of 4, and RELU activation
Convolution	128 filters of size 4x4, stride of 2, and RELU activation
Convolution	128 filters of size 3x3, stride of 1, and RELU activation
Flatten	-
Dense	size 1024
Dense	size Action space

3.4.1 Algorithm

For each episode, the model stacks three consecutive steps taken with the same action as a single state. This state stack fed into the DDQL model. The target model then predicts q values of all the action in action space for this state stack. The algorithm then selects the action based on Greedy-Epsilon policy. It then records the observation (current state, action, next state, done) which is used during mini batch training to train the main model. This process is repeated until the episode has been terminated. Periodically we update the target model using the main model.

3.5 Training

3.5.1 Time

It took almost 25 hours to train each models. I could not run the program continuously, so instead I saved the model and epsilon value periodically after every 100th run. It took almost a week to train these models. Took me almost two weeks to get some decent results.

3.5.2 Some techniques to reduce the complexity

Episode Termination Policy: According to the environment, the episode finishes when either all of the tiles are visited, or the car goes outside of the playfield in which case it will receive -100 reward. However, I also terminate the process if the car stays on field for more than the tolerance (i.e., 25 steps)

Image processing: Unlike colored images, where each pixel represents three RBG components (3 * 8 bits per pixel), in grayscale images the value of each pixel only represents an amount of light, i.e., it carries only intensity information (8 bits per pixel). In DQL, every three consecutive states are first

stacked together and then converted int grey scale image. They are then turned into a 1-Dimention data array that is feeding into the model.

Discrete Action Space: In order to reduce the complexity, I reduce the action space into 12 desecrate actions that are combination of three direction (left, right and center), two speed (0 or full) and two breaks (0 or 0.2). I give an extra reward for full speed, in order for the car to find short cuts.

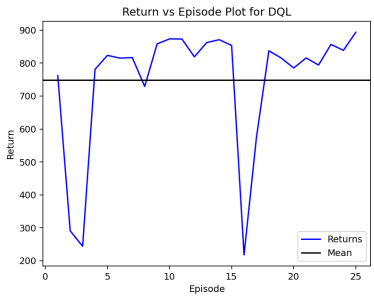
Mini-batch training: Periodically I sample a batch from the dataset and train the main model, after every 65 steps. For DDQL, I then trained the target model after every 10 episodes. For both methods, I saved the model after every 100 episodes

Decaying epsilon for greedy epsilon policy: I began the training that epsilon equals 1.0, i.e. total exploration, however slowly, I reduced the epsilon to 0.0 making it complete exploitation. During evaluation I set the epsilon to be 0.0, e.i. total exploitation.

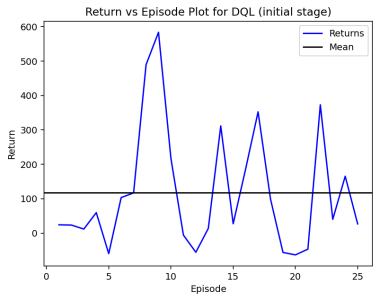
3.6 Evaluation

I ran 25 episodes and recorded the observation. In the evaluation I set the epsilon to be 0.0, i.e., complete exploitation.

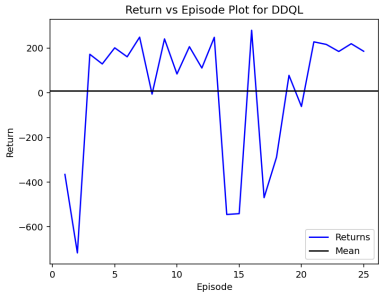
3.7 Result



(a) Result of DQL



(b) Result of DQL early stage



(c) Result of DQL early stage

Figure 5: DQL Return

4 Conclusion

My DQL implementation has, evidently, outperformed the DDQL methods. However, when I compare the early stage DQL with DDQL, their performance were quite similar. This may indicate the DDQL need a much more resources to train or maybe I need to change hyper parameters such frame depth and exploration rate.

5 Future Work

The model might not give a very comfortable ride to the passenger if used in real life. For comfortable journeys, it is important that the model understands the dynamics of cars such as jerks etc. This issue may be resolved if we use continues action space with more control over motion.

Secondly, OpenAI is a relatively simple environment; the car is practically isolated. A model should be trained for much more complex and real-life scenarios, which may include other cars, obstacles, traffic rules etc.

Third, these models could be trained with a powerful GPU. This would give much more flexibility in terms of training processes and data manipulation.

Lastly, other reinforcement learning algorithms, such as PPO (Proximal Policy Optimization) and DDPG (Deep Deterministic Policy Gradient) could also be tested and evaluated for this environment.

6 Reference

[1] "Q learning"(1992) - Christopher J. C. H. Watkins, Peter Dayan
[2] "Playing Atari with Deep Reinforcement Learning" (2013) - Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, Martin Riedmiller
[3] "Deep Reinforcement Learning with Double Q-learning" (2015) - Hado van Hasselt, Arthur Guez, David Silver