Solving Mountain Car prolem from OpenAI Gym with two reinforcement learning algorithms

Shenghui Chen (sc9by), Jiajia Liang (jl9pg), Zhengkun Xiao (zx8yz), Yike Guo (yg7jg)

***Abstract*—In this paper, we consider the problem of mountain car reaching the goal uphill, a classic control problem selected from the OpenAI Gym environments. We attempt to solve the problem with two approaches. First, we use a simple neural network to learn the weights for features that constitutes Q-value, thereby solving the problem via an approximate Q-learning way. In the second method, we directly apply the exact Q-learning algorithm after discretizing the state space of the scenario. We implement the algorithms in python and the results are presented respectively. We then analyze and compare the two algorithms from the same set of metrics such as performance, training time, and ease-of-use. With many algorithms to use, which one is optimal to choose in order to solve classic control problems? This paper tries to answer that by showcasing the pros and cons for each method. However, the results are to a large extent preliminary given the limited time and computing resources. Therefore, we hope to further investigate the problem with more experimental data, different reinforcement learning algorithms, and more sophisticated problem setting.**

***Index Terms*—Reinforcement Learning, OpenAI Gym, Q-learning**

I. Introduction

IN this coming new industrial revolution, one of the many focuses is on developing intelligent autonomous systems. One of the critical challenge is to control via AI algorithms. In this domain, the technique of reinforcement learning is particularly important as it does not require a large amount of prior data unlike in deep learning algorithms.

Since this is a rather new field, we do not have a set of standard solution to the classic control problems. Oftentimes, different researchers have proposed different algorithms in response to the same problem. Therefore, we feel the need to compare different methods in terms of their effectiveness, performance, and ease-of-use on a standard scale.

To achieve this goal, it only makes sense to measure two algorithms to one same problem. We chose the OpenAI Gym as our platform to implement two reinforcement learning algorithms for the same Mountain Car problem. OpenAI Gym is a popular open-source repository of reinforcement learning environments and development tools. We see this as an appropriate platform for our project because it is now a standard benchmarking tool for RL algorithms and are designed to be computationally tractable on modern consumer-grade hardware. [1]

We chose a classic control problem MountainCar-v0 as our benchmark setting, and implemented two reinforcement learning algorithms. In the first method, we use a deep reinforcement learning approach, which utilize a simple neural network to learn the weights for key features and thereby updating the Q values. In the second method, we adhere to the exact Q-learning equation. We then presents the results of both algorithms in the same set of metrics and compare their results. From the analysis on the respective performance of each method, we conclude some pros and cons for employing each method.

In summary, the main contributions of this paper are:

- Two implementations of solving the Mountain Car problem (deep RL, simple RL)
- Measuring the performance of each algorithm in a standard way, using the same set of metrics
- Comparison and analysis on the advantages and drawbacks for each method and corresponding suggestions on when to use which method

The rest of the paper is organized as follows. Section II surveys the existing works relevant to this paper. Section III briefly explains the OpenAI Gym platform and the method of Q-learning. Section IV introduces the problem of mountain car, specifying the goal, assumptions and parameters in this scenario. Section V describes the method of deep reinforcement learning with related results; section VI describes the method of exact Q-learning with related results. Section VII draws a comparison between the two methods used in different aspects. Section VIII concludes the paper with limitations and directions for future work. Section IX details the task appropriation among our team and briefly describes the difficulties we met and how we responded them.

II. Related Works

In recent years, game playing has gradually become a popular way to demonstrate the power of artificial intelligence. With the application of convolutional neural networks, performance of game playing among various types of games has been improving, too. Up to these days, it is even not surprising to observe AI players outperform expert human players in many fields.

A recent research[2] about game playing with Deep Q-Learning using OpenAI Gym reveals that the kind of models using deep reinforcement learning can be trained on more than one game from OpenAI Gym at the same time, or even on multiple other more complicated games. In addition, the training time of the models could be reduced by utilizing Double Deep Q-Learning and sinusoidal epsilon function. This paper is profoundly significant, because designing game players usually requires specific knowledge of the particular game to be integrated into the model for the game playing program. As a result, a model can only learn to play one game

at each time, in order to be a successful player. Therefore, with the assistance of computer vision and deep neural networks, it is now feasible to create multi-skilled computer players by obtaining digital information in pixels and training models with deep reinforcement learning.

Hence, our research group finds inspiration from the generalized model that can have excellent performance among several games, and decides to explore more about the difference between our simple AI algorithms, created for one specific environment, and complex AI algorithms, designed by scholars, for multiple environments.

III. Preliminaries

A. OpenAI Gym

OpenAI Gym is a toolkit for reinforcement learning research, which includes a growing collection of benchmark problems that demonstrates the performance of different algorithms. [3] Its library is a collection of well-established physical environments that have shared interface, and it allows users to implement their algorithms to record the performance. Its game categories range from classic control scenarios to real Atari 1600 games in the market. Our Mountain Car problem is one of the classic control problems in the OpenAI Gym.

B. Q-Learning

Reinforcement Learning can be divided into Model-Based Learning and Model-Free Learning. Q-Learning is the latter one, as it can gradually converges to optimal policy without giving any policy evaluation and reward function. The reason to use Q-Learning is that in practice, it is sometimes impossible to come up with a specific policy evaluation. In our OpenAI Gym environment, a better strategy is also to explore rather than to exploit. In general, Q-Learning update function can be presented in the following way:

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha(r + \gamma \max_{a'} Q(s',a'))$$

In this equation, α is the learning rate that has a value between 0 and 1. Setting it to 0 means that the Q-values are never updated, hence nothing is learned. Setting a high value such as 0.9 means that learning can occur quickly. Then, γ is the discount factor, also between 0 and 1. The $\max_{a'} Q(s',a'))$ represents a series of actions that attains the maximum reward by computing each subsequent node from the current state.

This function is derived from its original function of sample-based Q-value iteration, which is similar to TD learning. Starting with the state Q_0, the procedure does not differ vastly from the general Reinforcement Learning process:

1) Initialize the Q-values table, Q(s,a).
2) Observe the current state, s.
3) Choose an action, a, for that state
4) Take the action, and observe the reward, r, as well as the new state, s'.
5) Compute the reward and observe the maximum reward possible for the next state.
6) Update the state with Q-Learning update function.
7) repeat until terminal state has been reached.

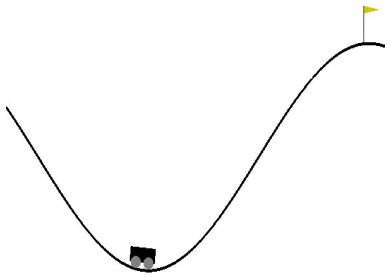

Fig. (1) Mountain car problem scenario

IV. Problem Description

The problem we are trying to solve is the mountain car problem named *mountainCar-v0*. The agent is an underpowered car dropped into the valley, and the goal is to reach the top of a hill which position equals 0.5. See Fig.1 for the visual representation of the environment. Since the car's engine is not powerful enough to drive up the hill without a head start, the car need to drive up the left hill to gain enough momentum. Essential properties for the problem are listed below[4]:

- *State Space*: For each state, the car has position between -1.2 and 0.6, and velocity between -0.07 and 0.07 as a vector. The state space is summarized as follows:

num	observation	min	max
0	position	-1.2	0.6
1	velocity	-0.07	0.07

- *Action Space*: The actions that the car can take is one of the following: push left, push right, and stay still.
- *Reward*: For each time step (action taken by the car), the reward is -1, and there is no penalty for moving in the direction opposite to the flag.
- Starting State: The car will be initialized randomly at position -0.6 to 0.4.
- *Episode Termination*: The program will terminates either after the goal position 0.5 has been reached or 200 iterations are taken.

To evaluate the performance of our algorithms, we are going to analyze the following:

- *Training time:* Training time includes the time to run the main algorithms for all episodes. It excludes the time for importing library and outputting results.
- *Success rate:* Success rate is defined as the percentage of successful trials divided by total trials. When the car reaches position 0.5 within 200 iterations, it will be counted as a success trial.
- *Success rate vs execution time:* This will be showing how success rate increase in each unit of training time.
- *Average reward vs episode:* This will be used to analyze how the average reward changes as more episodes are being learned.

V. Method 1: Deep RL

A. Supervised learning Network

We are using PyTorch neural network to build a two linear layers network. Our network takes state space as input, feeding into 200 units hidden layers, and output the action state. As discussed in the preliminaries, we will use Q-Learning as our reinforcement learning policy. Q-Learning Update Equation is

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

For our Deep reinforcement learning network, $[r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$ is the loss function. $Q(s_t, a_t)$ is the output of our neural network, and $\gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$ is our target Q value. The pseudo-code of our algorithm is as following [5][6]:

initialize Q(s,a)
for each episode **do**
  Initialize S;
  Choose A from S using policy derived from Q;
  for each step of episode **do**
    Observe Reward, S' after action A
    Choose A' from S' using policy derived from Q
    $Q(S, A) \leftarrow Q(S, A) + a[R + \gamma Q(S', A') - Q(S, A)]$
    $S \leftarrow S'; A \leftarrow A'$;
  end for
  until S is terminal
end for

B. Performance

- *Training time:* We tested our learning model for total of 10000 episodes. For training time, it takes on average 20.53 minutes to finish 10000 episodes. The run-time exclude the time for printing and plotting results, but includes the time for writing to writers. Therefore, we expect the actual run-time for main algorithms should be shorter than times we reported here. We notices, however, that the time it takes to train can varied a lot depending on computer environment. For example, training 10000 takes on average 20.53 minutes, with exception taking more than 30 minutes. To handle this problem, we run training session for 10 times and reported the average execution time.
- *Success rate:* The average success rates for 1000,3000,5000,and 10000 episodes are showed below:

episode	success rate(percent)
1000	0
3000	17
5000	22
10000	37

- *Success rate vs execution time:* Knowing that the average success rate increase as the number of episodes increase, we want to see training efficiency of the deep neural network for our problem. Specifically, in unit of time, how much increase in success rate we would obtain. In Fig. 2, we plot the execution time in seconds on x axis, and the average success rate in percent on the y axis.

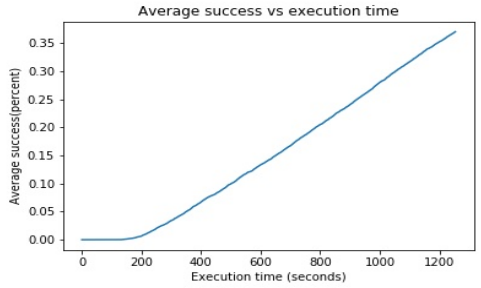

Fig. (2) Average success rate VS execution time for 10000 episodes

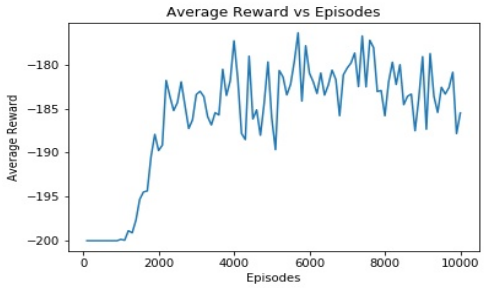

Fig. (3) Average rewards VS episodes for 10000 episodes

From the graph, we can see that after spending some time figuring out the environment, the success rate is steadily increase during each unit of time. We expect the line will continue to increase as the execution time increase, and eventually reached the ceiling and flatten out. Looking at the rate of increment, if we want to achieve a success rate of 90 percent, it would take about more than 3000 seconds to run.

- *Average reward vs episode:* Fig.3 shows the average reward for each episode we trained. The lowest reward is -200 since we terminate the trails after 200 iterations. In the first 1000 episodes, the average rewards remain -200, which correspond to our success rate of 0 percent. But once it randomly hit some successful trails, the network learns the model pretty fast, result in a big increase in success rate and average reward. Another way to interpret the average reward is to viewing it as the average number of steps it takes for the agent to reach the goal. After 2000 episodes, the average rewards is floating around -180 to -185. This might suggest that current learned strategy of reaching the goal requires about 180 steps on average.

C. Discussion and Improvement

We notice that in the first 1000 episodes, there is no increase in either success rate nor average rewards. This is caused by how we define the reward that feed into the network. Currently, we minus 1 point for each steps taken, therefore, the only way to increase reward, or reduce lost, is to reach the top in less than 200 steps. In the first thousand episodes, the agent is randomly moving around, have no knowledge about the strategy that reaching higher position is associated with reaching the goal, until later by chanced it learns the strategy

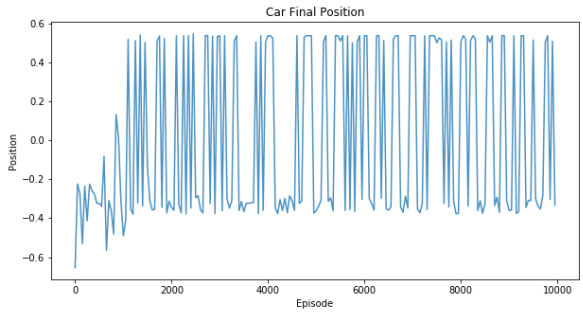

Fig. (4) Final position for original reward model

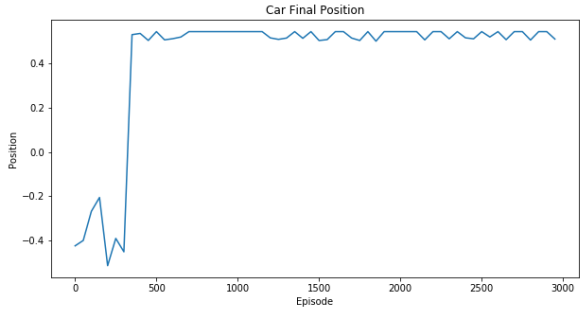

Fig. (5) Final position for modified reward model

by accidentally reaching the top. To motivate the agent move to higher position and gain more momentum, we can adjust the reward system by accounting the position of the agent reached, so that the higher it reaches, more rewards it gets[6].

We tested the modified version using smaller number of episodes. After such modification, the car learned the strategy of moving upward much faster. We plot the positions of the car for each 50 episodes using original reward system and the modified reward system in Fig.4 and Fig.5 respectively. From the plot of our original reward model(Fig.4), the agent has not yet figured out a solid strategy to success, which explained the oscillated pattern in the graph. As we can see from Fig.5, the modified version learned to reach the top immediately after 500 episodes and the algorithm is very stable at producing successful outcomes. The success rate for 3000 episodes is 89.7 percent, which is much efficient and accurate than the original model. In future comparison, however, we will still use the original reward model to remain consistent with our problem definition.

VI. Method 2: Simple RL

A. Exact Q-learning

In the simple Reinforcement learning implementation, we directly apply the exact Q-learning algorithm to this case. Compared with the previous method, we need to discretize the state space according to this specific case, use epsilon greedy strategy to determine next action, and then update the Q table according to the equation below:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

The algorithm is described in pseudo-code [7]:

for each episode **do**
  Initialize Q(s1, s2, a);
  for each step of episode **do**
    Observe the current state (s1, s2);
    Choose action, a according to the epsilon greedy strategy;
    Take action a and observe the resulting reward, r, and the new state of the environment, (s1', s2');
    Update Q(s1, s2, a) based on the update rule $Q'(S_1, S_2, a) = (1 - w) * Q(S_1, S_2, a) + w * (r + d * Q(S_1', S_2', argmax_{a'} Q(S_1', S_2', a')))$;
  end for
  until S is terminal
end for

The parameters used are 0.2 for learning rate w, 0.9 for discount d. Also we have an initial epsilon of 0.8 and a minimum epsilon of 0 in the epsilon greedy strategy.

B. Performance

- *Training time:* We tested this simple RL model for total of 10000 episodes. For training time, it takes on average 3.65 minutes to finish 10000 episodes. To handle the problem of variability among different executions, we run training session for 10 times and record the average execution time.
- *Success rate:* We tested the success rate on 1000, 3000, 5000, and 10000 episodes. The average success rate under each episodes is shown:

episode	success rate(percent)
1000	0
3000	0.04
5000	5.86
10000	8.24

- *Success rate vs execution time:* To demonstrate how the success rate changes with episodes, we plot the execution time in seconds on x axis and the average success rate in percent on the y axis in Figure 6. From the figure we can see that after spending some time exploring the environment, the success rate steadily increases with time. We expect the line to continue increase and eventually reach a plateau.
- *Average reward vs episode:* The average reward verses for each episode is shown in Figure 7. As we known the minimum reward for the scenario is -200, we can see the agent uses at least 5000 episodes to explore the environment and then pick up the momentum to succeed, thereby rapidly increasing their average reward. Although it takes much longer to dramatically boost the performance, the rate of learning seems much faster.

VII. Performance comparison

From approaching the same problem with two different algorithms, we not only solved the problem, but more importantly gain insights in the pros and cons for the use of

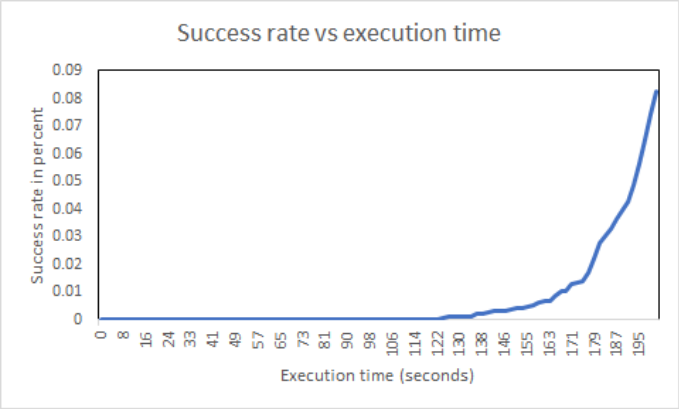

Fig. (6) Success rate vs execution time for the simple RL algorithm

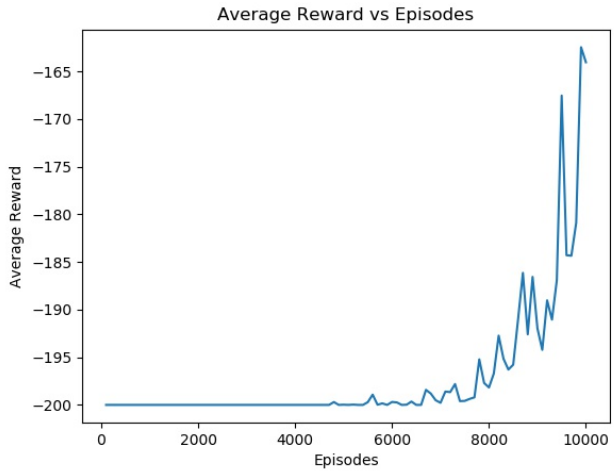

Fig. (7) Average reward vs episodes for simple RL algorithm

each algorithm. In its essence, the deep RL method utilizes an approximate Q-learning, which Q value was learned with a supervised learning network based on the following equation

$$Q(s,a) = \sum_i w_i f_i(s,a)$$

where $\{f_i(s,a)\}$ is a series of features, and the neural network was employed to optimize the values of weights for each feature. As for simple RL algorithm in method 2, it faithfully adheres the exact Q-learning equation

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t)+\alpha[r_t+\gamma \max_a Q(s_{t+1},a_{t+1})-Q(s_t,a_t)]$$

The difference in the nature of these two algorithms then determines the pros and cons for using each of the method. The algorithm for deep RL is more generalized and can be easily applied to other scenarios without many changes to the code, whereas we have to manually write the code that discretizes the state space for different scenarios in the simple RL method. The two algorithms also greatly influence their performance. Given the nature of training a neural network, deep RL method requires much more training time than that of the other method for the same amount of episodes. However, if we consider the fact that the final success rate of deep RL method is also much higher than that of simple RL, the time spent seems worthwhile.

Generally, when to choose which algorithm depends on the requirement on the accuracy of the performance. For example, if we only need the mountain car to succeed some time in the execution, there is not that much demand on the success rate of the final result. In this case, it makes sense to use the simple RL method which uses less time to gain some preliminary results. However, if we know there are a set of control problems to be solved, deep RL may be a more sensible choice since the code is more generalized and can be applied with more ease to different set-ups.

VIII. Conclusion and Future Work

In this paper, we considered the problem of enabling a car to reach its goal uphill by swinging back and forth to gain momentum. We used the simplified scenario from OpenAI Gym collection, and implemented two different reinforcement learning algorithms to solve the problem. For the first method (Deep RL), we used a simple neural network to learn the weights for features that constitutes Q-value, thereby solving the problem via an approximate Q-learning way. In the second method (Simple RL), we directly applied the exact Q-learning algorithm after discretizing the state space of the scenario. We implemented the algorithms in python and the results are presented respectively. We then analyzed and compared the two algorithms from the same set of metrics. Overall, we found the Deep RL algorithm takes much longer time to train, but can be applied to other problems with less modification in code. The Simple RL method, on the other hand, needs more effort from the programmers to properly discretize the state space, but usually hits the goal faster.

However, given the limited time and amount of computing resources at hand, we are not confident the results are final. Specifically, when we were training the neural network on our personal computer, the performance often varies greatly presumably due to the other operations being executed at the same time. With the similar methodology, we think a standardized computing environment is needed to make strong claims about their performance.

In the future, we hope to further investigate the problem with more experimental data, different reinforcement learning algorithms, and more sophisticated problem setting.

IX. Reflection

In this project, we assign the tasks based on the past experiences of our team members. Jiajia and Shenghui are reponsible for the Deep RL and Simple RL implementation, respectively. Zhengkun did much work in configuring the enviornment provided by OpenAI Gym, and Yike organizes the material in a video.

In this project, we met many difficulties. First of all, the environment set up in OpenAI Gym is much more complicated than we initially imagined. Zhengkun did a lot of work in properly setting up the dependencies on our computers so that Vivian and Jiajia can just focus on the code. Another major

obstacle in this project lies in the performance comparison part. To support the claim we would like to make about the relative performance of the two algorithms, we have to ensure the computing environment that runs our algorithms is standardized. Our personal computer clearly does not satisfy this experiment requirement. We also did not have enough time to repeat the experiment to an extent that we have strong arguments in the performance section.

REFERENCES

[1] P. Aldape and S. Sowell, "Reinforcement Learning for a Simple Racing Game," pp. 1–9, 2018.
[2] D. G. Robert Chuchro, "Game playing with deep q-learning using openai gym," 2017.
[3] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.
[4] "OpenAI Wiki," 2019. [Online]. Available: https://github.com/openai/gym/wiki/MountainCar-v0
[5] D. Silver. Ucl course on rl. UCL. [Online]. Available: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html
[6] Tim, "Solving Mountain Car with Q-Learning," 2018. [Online]. Available: https://medium.com/@ts1829/solving-mountain-car-with-q-learning-b77bf71b1de2
[7] G. Hayes, "Getting Started with Reinforcement Learning and Open AI Gym," 2019. [Online]. Available: https://towardsdatascience.com/getting-started-with-reinforcement-learning-and-open-ai-gym-c289aca874f