AI 3603 Artificial Intelligence: Principles and Techniques

By: Huangji Wang(519030910100) and Zhiteng Li(519030910105)

HW2: Reinforcement Learning

October 18, 2021

Contents

I. INTRODUCTION

A. Purpose

The goal of today's lab is to implement two reinforcement agents. The first is to find a safe path to the goal in a grid-shaped maze through the cliffs with a reinforcement learning algorithm. The second is to train intelligent agents to play the Sokoban game utilizing Sarsa, Q-learning, and Dyna-Q algorithms. Reinforcement learning is one type of power learning algorithm that is widely used in the artificial intelligence field and it can often lead to a beneficial and efficient solution after it is applied in the agents for a while. Thus, with the help of this lab, we want to understand the mechanisms of reinforcement learning algorithms.

In this lab, we will train agents with at most 4 kinds of reinforcement learning algorithms as the core code. In the cliff-walking task, we manage to find the best reward path after training the agent with learning rate lr and random direction factor ϵ. In the Sokoban game, we manage to train the agent to move the boxes in the best way with the same factors. After we finish the tasks, we adjust three parameters to optimize the results:

- **Learning rate:**

  Learning rate depends on the weight of past experiences. As the learning rate gets closer to 1, it means we just pay attention to the current result. However, past training experiences are very important in the learning algorithm. Thus, finding one good learning rate is important.

- **Epsilon ϵ:**

  Epsilon depends on the randomness of the directions the agent chooses. At first, we want the agent can walk as randomly as possible. In this way, we can update the map and Q-table on time. As the number of training batches gets larger, the value of epsilon should get smaller so that the agent can perform well according to the policies.

At last, we learned reinforcement learning algorithms and try to realize them.

B. Equipment

There is a minimal amount of equipment to be used in this lab. A few requirements are listed below:

- Visual Studio Code: Version 1.60.
- Windows environment: Python 3.6, Numpy 1.19.5, and matplotlib 3.1.1.
- Required environment: Anaconda 3.6, conda activate gym and imageio installation.
- Python original library environment (If you don't have): math, os, Heapq and random.

C. Procedure

1. Install the essential environment.
2. Start VS code. Open all the main codes.
3. Run the codes in the terminal. Observe the trajectory of the agent.
4. Train the walk of the agent, plot it and analyze the situation.
5. Stop the simulation, examine the graphs, and make a summary.

II. PATH DIAGRAMS

This section consists of screenshots taken during the procedure in two tasks.

A. Cliff-walking

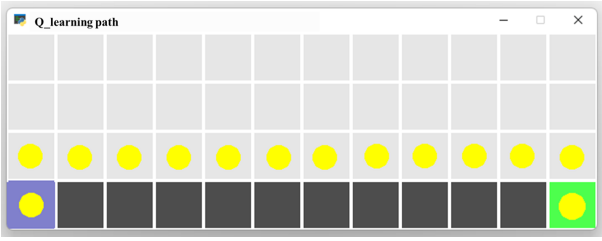


FIG. 1: Q-learning path for Cliff-walking.

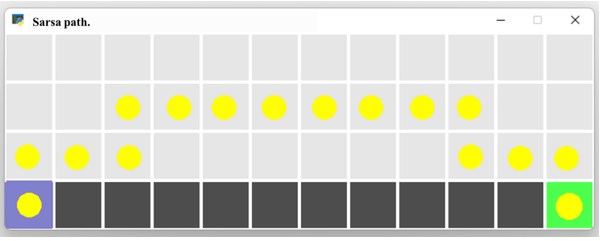
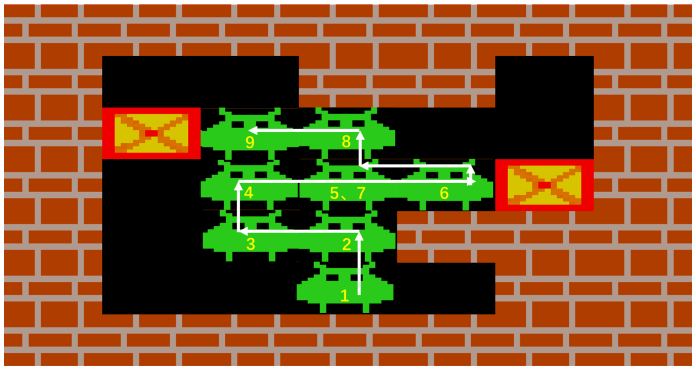
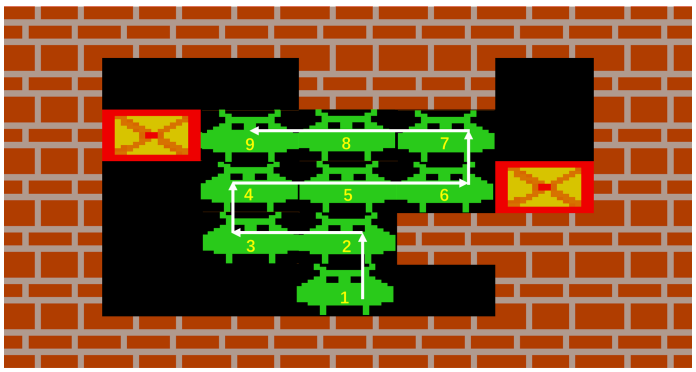


FIG. 2: Sarsa path for Cliff-walking.

B. Sokoban[1]



(a) Q-learning and Sarsa path for Sokoban.



(b) Dynaq and New exploration path for Sokoban.

FIG. 3: Two paths for Sokoban game.

III. EXPERIMENT DATA

This section will consist of the important code and the main idea of all the algorithms which displays our comprehension of these algorithms to meet the requirements of the lab.

A. Reinforcement Learning in Cliff-walking Environment

1. Cliff-walking with Sarsa Algorithm

a) SasarAgent Class

We add two attributes, num_actions and num_space to the initial function. They represent the size of action space and state-space respectively. Also, we build a Q-table in the initial function, with all the entries are set to 0. Besides, we initially set the epsilon and learning rate to 1.

In the choose_action function, we first generate a random number between 0 and 1. If the random number is smaller than epsilon, then we choose action randomly. Otherwise, we choose the best action according to the Q-table. For the best action, we just compare the Q-value of 4 next states due to different actions, then choose the action that maximizes the Q-value of the next state.

As for the learning function, we add five more parameters to it, which are the current state, the next state, the action chose on the current state, the living reward, and the discounting factor (It is fixed to 0.9 in this experiment). And we update the value of the Q-table based on the following formula

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s^{'}, a^{'}) - Q(s,a)]$$

b) cliff_walk_sarsa

When the agent is done, we assign the living reward to the Q-value of the current state, which is +10 if the agent arrives at the goal or -100 if the agent arrives at the cliff.

In addition, we let epsilon and learning rate decay exponentially with a decay rate of 0.95 per episode. At the beginning of the game, we need to let the agent explore more, so epsilon will be set higher, but as the agent continues to learn, we should be more inclined to deterministic strategies, so epsilon is set to exponentially decay and is finally set to 0 after the strategy converges.

2. Cliff-walking with Q-learning Algorithm

a) QLearningAgent Class

The initial function and the choose_action function are the same as Sarsa.

As for the learn function, we add the same parameters as Sarsa. And we update the value of Q-table based on the following formula (**different from Sarsa**)

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

b) cliff_walk_qlearning

When the agent is done, we do the same operations as Sarsa.

In addition, we let epsilon and learning rate decay exponentially with decay rate 0.9 and 0.99 per episode respectively. And the epsilon is finally set to 0 after the strategy converges.

3. *Reasons for Differences between Sarsa and Q-learning Algorithms*

These two pictures below show the difference between Sarsa and Q-learning algorithms. These two pictures are about the downward trend of episode and epsilon in these two algorithms.

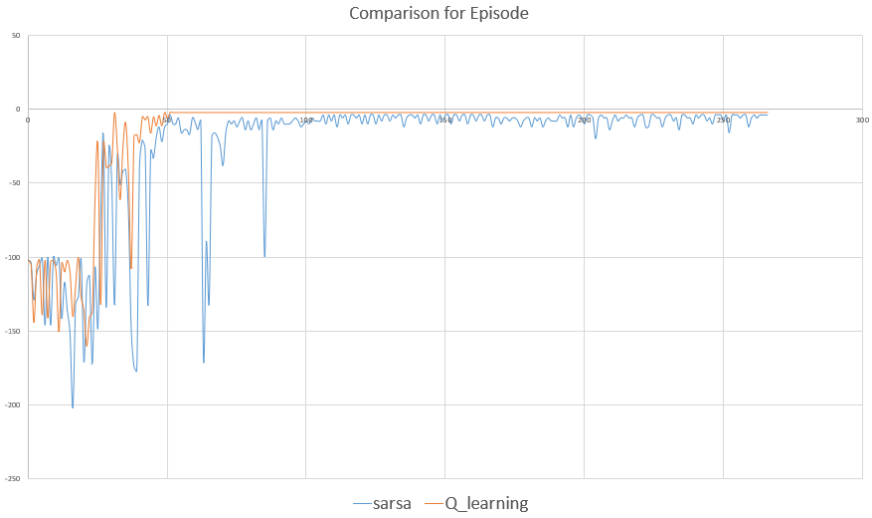


FIG. 4: Comparison for Episode.

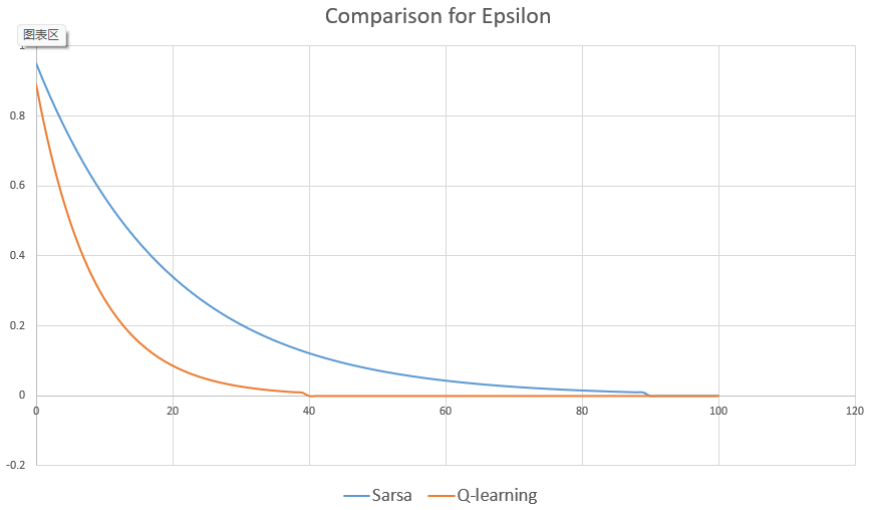


FIG. 5: Comparison for Epsilon.

a) **Learning Process of Sarsa**

At the very beginning, initialize the Q-table arbitrarily. After that, for each episode, initialize state s, and choose an action based on $\epsilon - greedy$ strategy, then repeat the following step until s is terminal.

The step is first to choose the next action based on $\epsilon - greedy$, then update the Q-value of the current state based on the formula below. Finally, assign the next state to the current state and the next

action to the current action.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s^{'}, a^{'}) - Q(s,a)]$$

Since we initialize ϵ as 1, and set it to decay exponentially. The agent will first try to walk randomly, and then follow the best action chose according to the Q-table as time goes by. What's more, since Sarsa update Q-value based on the **actual** chose the next action (**different from Q-learning**), Sarsa takes a more conservative path, updating his Q with his actions planned for the future and is sensitive to errors and death.

b) **Learning Process of Q-learning**

At the very beginning, initialize the Q-table arbitrarily. After that, for each episode, initialize state s, and then repeat the following step until s is terminal.

The step is first to choose the current activity based on $\epsilon - greedy$, then update the Q-value of the current state based on the formula below. Finally, assign the next state to the current state.

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

Since we initialize ϵ as 1, and set it to decay exponentially. The agent will first try to walk randomly, and then follow the best action chose according to the Q-table as time goes by. What's more, since Q-learning updates Q-value based on the **best** next action (**different from Sarsa**) but does not always choose it at the next step, Q-learning takes a more adventurous path. Q-learning maximizes the direction of the Q-table every time it updates and re-selects the action in the next state. Q-learning is a reckless, bold, and greedy algorithm that does not care about death and mistakes.

c) **Difference and Reasons**

From the principle of these algorithms, we can see the difference between the two algorithms. Sarsa is an on-policy algorithm, while Q-learning is an off-policy algorithm. Sarsa takes a conservative approach, updating his Q with his actions planned for the future, and is sensitive to errors and death. Q-learning, on the other hand, maximizes the direction of Q every time it updates and re-selects the action in the next state. Q-learning is a reckless, bold, and greedy algorithm that does not care about death and mistakes.

This difference is the reason for different paths in different algorithms. Q-learning tries to find the fastest way to the goal and ignores the danger on the road while Sarsa firstly considers its safety and then the goal. Thus, in the resulting path, Sarsa is more likely away from the cliffs and converges slower while Q-learning likes dancing on the edge of the cliffs and converges faster.

B. Reinforcement Learning in the Sokoban Game[2]

Different from cliff-walking, we need to define a high-dimensional state space in the Sokoban Game, which must combine the information of the positions of the agent and two boxes. On the one hand, the state we define must correspond to the actual game state one-to-one. On the other hand, we want to reduce the use of space as much as possible. Thus, we define the following state expression

$$state = 49^0 \times (agent.x + 7 \times agent.y) + 49^1 \times (box_1.x + 7 \times box_1.y) + 49^2 \times (box_2.x + 7 \times box_2.y)$$

First, we transfer the coordinate into a real number by $num = x + 7 \times y$. Second, we assign a weight of each real number based on the space of each coordinate ($7 \times 7 = 49$). Through this encoding method, we can ensure that there is no conflict between states and reduce the use of space at the same time.

1. Sarsa Algorithm in the Sokoban Game

a) SasarAgent Class

The same as Cliff-walking.

b) sokoban_sarsa

First, we assign 200000 to the state space (may be larger than need) and use the formula above to transfer state expression. The last things are similar to Cliff-walking.

2. Q-learning Algorithm in the Sokoban Game

a) QLearningAgent Class

The same as Cliff-walking.

b) sokoban_qlearning

First, we assign 200000 to the state space (may be larger than need) and use the formula above to transfer state expression. The last things are similar to Cliff-walking.

3. Dyna-Q Algorithm in the Sokoban Game

a) DynaQAgent Class

Different from model-free learning algorithms (like Sarsa and Q-learning), we build an experience pool to store experience gained during previous interactions with the environment, which is defined as a 4-tuple $(s, a, s^{'}, r)$.

In addition, we add one more method called $TD-error$[3] to calculate the TD error based on the formula below

$$\delta_t = r + \gamma \max_{a'} Q(s_t, a_t) - Q(s_{t-1}, a_{t-1})$$

It can be used to represent the value of one experience. If the td-error is large, then we have more space to update it and thus it need to be sampled more times. Based on the TD-error, we defined a prioritiaed experience replay method to sample from the experience pool.

b) sokoban_dynaq

To achieve prioritized experience replay, we build a heap to store all the experience and encapsulate experience into a class, which defined the compare principle between TD errors.

In each training, we only interact with the environment once to obtain the corresponding experience for learning. In addition, we also sample a certain amount of experience from the experience pool for learning. In this way, the agent interacts with the environment once but can learn from multiple experiences. After learning an experience from the experience pool, we will update the TD error of the current experience and push it back to the pool.

4. Reasons for Differences among These Algorithms

a) **Learning Process of Sarsa**

At the very beginning, initialize the Q-table arbitrarily. After that, for each episode, initialize state s, and choose an action based on $\epsilon - greedy$ strategy, then repeat the following step until s is terminal.

The step is first to choose the next action based on $\epsilon - greedy$, then update the Q-value of the current state based on the formula below. Finally, assign the next state to the current state and the next action to the current action.

$$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s^{'}, a^{'}) - Q(s,a)]$$

Since we initialize ϵ as 1, and set it to decay exponentially. The agent will first try to walk randomly, and then follow the best action chose according to the Q-table as time goes by. What's more, since Sarsa update Q-value based on the **actual** chose the next action (**different from Q-learning**), Sarsa takes a more conservative path. However, since Sokoban Game only has a positive reward (push two boxes into targets), the final path will be the same as Q-learning.

b) **Learning Process of Q-learning**

At the very beginning, initialize the Q-table arbitrarily. After that, for each episode, initialize state s, and then repeat the following step until s is terminal.

The step is first to choose the current activity based on $\epsilon - greedy$, then update the Q-value of the current state based on the formula below. Finally, assign the next state to the current state.

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

Since we initialize ϵ as 1, and set it to decay exponentially. The agent will first try to walk randomly, and then follow the best action chose according to the Q-table as time goes by. What's more, since Q-learning updates Q-value based on the **best** next action (**different from Sarsa**) but does not always choose it at the next step, Q-learning takes a more adventurous path. However, since Sokoban Game only has a positive reward (push two boxes into targets), the final path will be the same as Sarsa.

c) **Learning Process of Dyna-Q**

At the very beginning, initialize the Q-table arbitrarily. After that, for each episode, initialize state s, and then repeat the following step until s is terminal.

The step is first to choose the current activity based on $\epsilon - greedy$, then update the Q-value of the current state based on the formula below. **Unlike Sarsa and Q-learning, we don't just learn from the current experience. Instead, we will sample 10 more experiences from the experience pool with prioritized experience replay.** After that, push the current experience into the pool. Finally, assign the next state to the current state.

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

Since we sample experience based on the TD error, those states with more upside room for Q-value will be updated first. Thus, the final path will be a little different from Sarsa and Q-learning.

d) **Difference and Reasons**

These two pictures below show the difference among Sarsa, Q-learning, and Dynaq algorithms. These two pictures are about the downward trend of episode and epsilon in these two algorithms.

FIG. 6: Comparison for Episode.

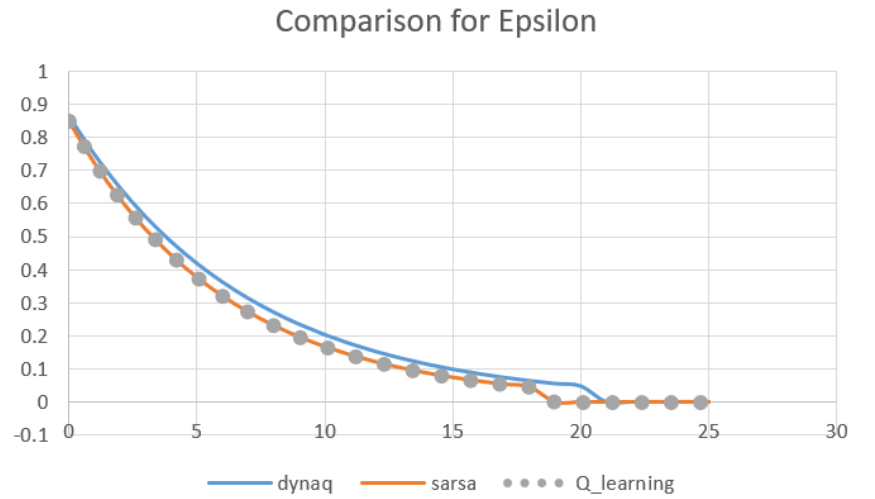


FIG. 7: Comparison for Epsilon.

Since Dyna-Q is based on Q-learning and learns more experience in one episode, it is nothing strange that Dyna-Q will converge faster than Q-learning. And since Sokoban Game doesn't have a negative final reward, Sarsa doesn't need to take a conservative path, thus the update rule of Sarsa will be more suitable and faster to converge than Q-learning.

5. *Summary and New Exploration Method for* ***Explore-Exploit Dilemma***

a) **Comparison among Different** $\epsilon values$ **and** $\epsilon - decay$ **Schemes**

First, we test different exponent for exponent decay from $\epsilon = \epsilon \times 0.9$ to $\epsilon = \epsilon \times 0.1$. The decay speed of ϵ and the learning speed of different exponents are shown below.

We can easily find that it isn't much difference between different exponent, they all converge very quickly. However, we can still find that the reward is converging fastest when the exponent is 0.4 and when the exponent increases or decreases, the reward will converge slower. From the result of the experiment, we know that it's better to set exponent in a suitable range to get faster learning speed.

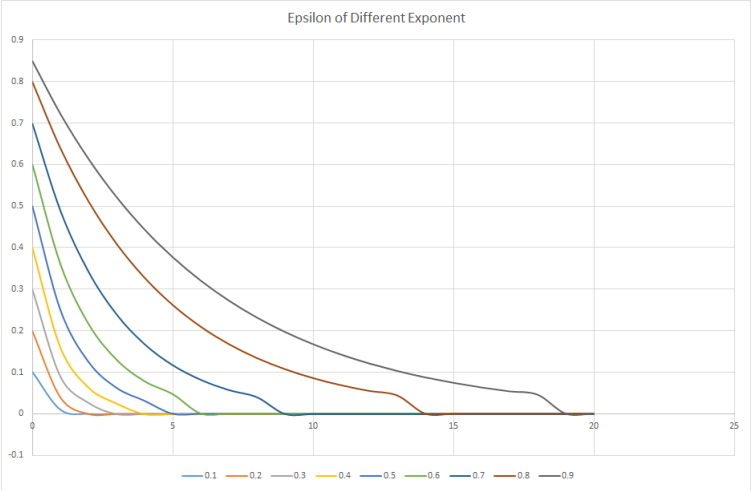


FIG. 8: Epsilon of Different Exponent.

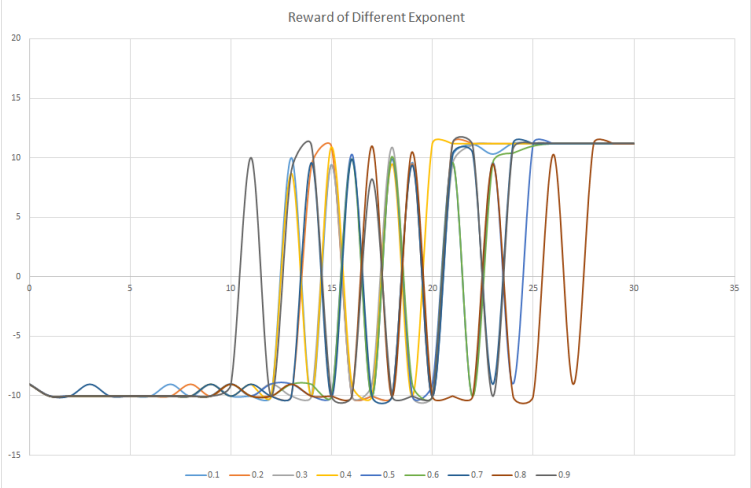


FIG. 9: Reward of Different Exponent.

Second, we test different decay methods for ϵ, including linear decay, segmented decay, and exponential decay. The decay speed of ϵ and the learning speed of different decay methods are shown below. Since the reward fluctuates greatly when the episode is small, we use average reward instead to make it easier to analyze.

We can easily find that the learning speed of exponential decay is fastest while the learning rate of segmented decay is the slowest, the learning speed of linear decay is between these two. Thus, we can conclude that exponential decay is a better $\epsilon - decay$ scheme.

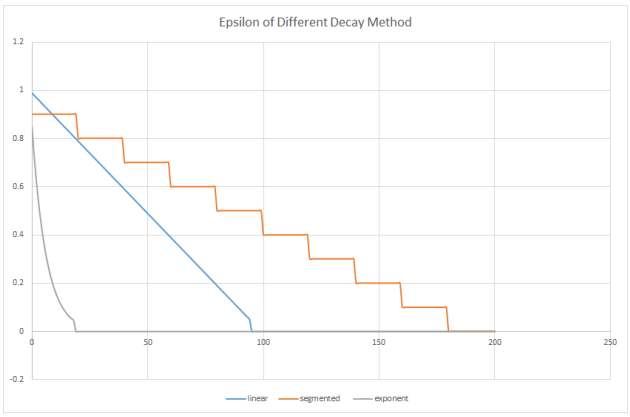


FIG. 10: Epsilon of Different Decay Method.

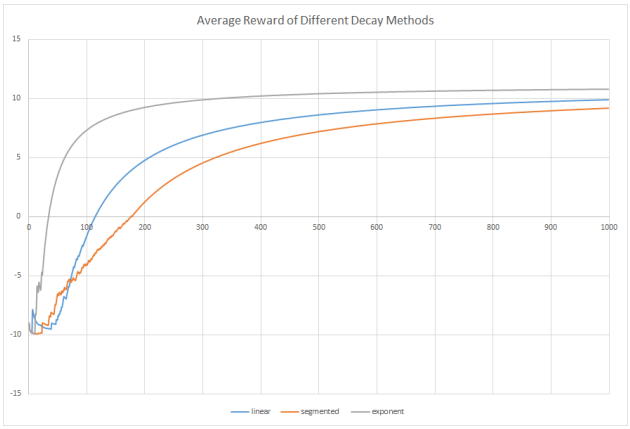


FIG. 11: Average Reward of Different Decay Method.

b) **New Exploration Method: UCB**

Besides the $\epsilon - greedy$, we learn one new exploration method called Upper Confidence Bound, which defines a function to choose the next action

$$score(a) = Q(s,a) + \sqrt{\frac{2\ln n}{N}}$$

n is the number of times that action is selected, and N is the total number of times that the four actions are selected.

UCB assigns a value to each action based on the corresponding Q-value and the frequency we choose it. The higher Q-value is, the more score will be assigned. The lower frequency we choose the action, the higher score will be assigned.

To record the times we choose one action, we build a time list to store it for each state. Each time, we calculate the score of 4 actions, then choose the highest one as the next action.

IV. DISCUSSION

The goal of this lab was to apply reinforcement learning in the simulation system to run the agent well. By pressing different combinations of the epsilon, learning rate and agent classes, the agent on the simulation tool is to act in different ways using these algorithms.

A. Strengths

1. **Effective Library: priority queue (heapq)**

   We use the priority as the fringe of the Dynaq algorithm, which can significantly reduce the time complexity from $O(n^2)$ to $O(n \log n)$. Also, we use the set as the fringe of the experiences, which can avoid adding repetitive elements to reduce time complexity.

2. **Comprehensive Parameters Adjustment**

   To find the perfect parameters that let the algorithms converge as fast as possible, we set plenty of different values of the parameters. For example, the Q-learning algorithm in the Sokoban game converges after 100 iterations at first, but when we adjust the parameters it converges after just 25 iterations.

3. **Intuitive Visual Interface**

   Although the **gym_recoder** file can not work in Windows 10, we find a good way to record each step of the agents. We combined all the steps so that we can directly recognize the final path and it will help us analyze the formulation and steps of the algorithm. By the way, we can easily obtain information about the difference among these algorithms.

4. **Strong Connection with Encoding Data**

   To find connections between states and the position of agents and boxes, we encode the data and transfer it into forty-nine base numbers, leading to a one-to-one connection. Thus, in this lab, we successfully apply information theory to data compression.

B. Weaknesses

1. **More Experience Calculation in Dynaq**

   Although we use priority queue and set to reduce time complexity, to achieve more requirements like the TD error and updating states, it will pay much more than we think. For example, when we choose one state in the experience pool in each step, we should choose at least 10 experiences to calculate.

2. **Terrible Time Cost for Huge Scale**

   Considering the time complexity, we know that the time complexity of these algorithms is at least $O(s^2A)$ and this is just the cost for Q-table in one iteration. Thus, if the scale is huge, the efficiency of our algorithms will be very bad.

V. CONCLUSION

Nowadays we should notice that artificial intelligence has made great progress with the help of new training and searching algorithms. Algorithms have played a more and more important role in helping agents to perfectly finish the tasks. A preliminary analysis of this homework project is of great significance to our understanding of the high-tech frontier technology.

In this project, we finish the reinforcement algorithms by improving some factors that are not easy to be quantified in modeling. We try to find linear approximation and some other direct features by setting exact values. We compared these two tasks and the algorithms applied in these tasks. At last, we test our algorithms, present their result figures, and analyze the comparison and their strengths and weaknesses. Such exploration can provide some useful and reasonable results which can be applied to real situations.

1. **Task 1: Cliff-walking**

   We provide each initial status with Q-learning and Sarsa paths and analyze the result and differences.

2. **Task 2: Sokoban Game**

   We provide each initial status with four types of paths and analyze the result and differences. We also present the summary for Explore-Exploit Dilemma.

All in all, this laboratory gave us an insight into how algorithms work in reality and we hope to be able to apply them to the following labs.

---

[1] M. Schrader, guyfreund, M. Schrader, Olloxan, maximilianigl, F. Schober, Jaromír, and Jaromír, *gym-sokoban* (2021), URL https://github.com/mpSchrader/gym-sokoban.

[2] N. Grinsztajn, J. Ferret, O. Pietquin, P. Preux, and M. Geist, *There is no turning back: A self-supervised approach for reversibility-aware reinforcement learning* (2021), arXiv:2106.04480.

[3] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, *Prioritized experience replay* (2015), arXiv:1511.05952.