AI 3603 Artificial Intelligence: Principles and Techniques

By: 518030910380 Xiaolin Bu
519030910089 Jiaxin Song

HW#: 2

October 26, 2021

I. INTRODUCTION

Contents

A. Purpose

The goal of this lab is to implement Reinforcement Learning agents using Sarsa, Q-Learning and dyna-Q algorithms. Then the agent is used in learning cliff walking and sokoban game. The agentsknows nothing at the beginning of the game. However, as the episode increases, the agent learns from the experience from the reward to choose a proper action at every state. Finally, the agent forms a strategy to get the highest reward in the game.

The agent also needs to keep a balance between exploration and exploitation. If it explores too much and learns little, it won't converge for a long time. If it explores a little then leads the final result, it may converge at local optimal, rather than global optimal. Hence, we do further research on the relationship between exploration and exploitation as well as other exploration methods.

B. Equipment

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

- OS: Fedora 34
- python 3.8
- Anaconda 1.7
- Python modules including gym, matplotlib, numpy, etc.

II. CLIFF WALKING

In the cliff walking part, we implement two reinforcement algorithms: Q-learning and SARSA. The basic idea of reinforcement is: when a state is given, the agent chooses an action and receives a reward, then moves to another state and repeats the loop until the strategy converges.

For the cliff walking environment:

- state: 12×4 maze which consists of safe region, cliff and exit
- action: up, down, left and right, represented by integer 0 to 4
- reward: 1 as living cost, -100 for falling off the region and +10 for reaching the exit

The Q-learning and SARSA algorithms are pretty similar. The key point of both algorithms is to maintain a Q-table and choose the next according to the Q-table. The main procedure of the algorithms are basically like: initialize the Q-table $\rightarrow$ choose an action $\rightarrow$ perform the action $\rightarrow$ measure the reward $\rightarrow$ update Q-table $\rightarrow$ choose action again $\rightarrow$ ...

Moreover, action in both algorithms is chosen by ϵ-greedy scheme. The only difference is how to update the Q table.

A. Q-learning

- **Description** In the Q-learning algorithm, the update of Q-value in one state uses the maximal Q-value in the next state. It can be written as:

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

  We choose the max Q-value for updating, but in the next state, there's a chance that we do not choose the action that leads to the max value. There's a difference between what we choose and what we do. That why the algorithm is called off-policy learning.

- **Implementation** We implement the algorithm and the pseudo code is as follows. Each state is calculated from the *observation* given in the environment and the Q-table is represented as a $(12 \times 4 \times 4)$ array. We implement the choose action function and learn function. Further, to get the best performance and understand the influence of different parameters, we choose learning rate α to be 0.1, 0.5, 0.8 respectively, reward decay γ to be 0.9 and 1. We also initialize the ϵ with 1 and try three different ϵ-decay schemes. The result will be visualized and analysed in the following section.

Algorithm 1: Q-learning Algorithm

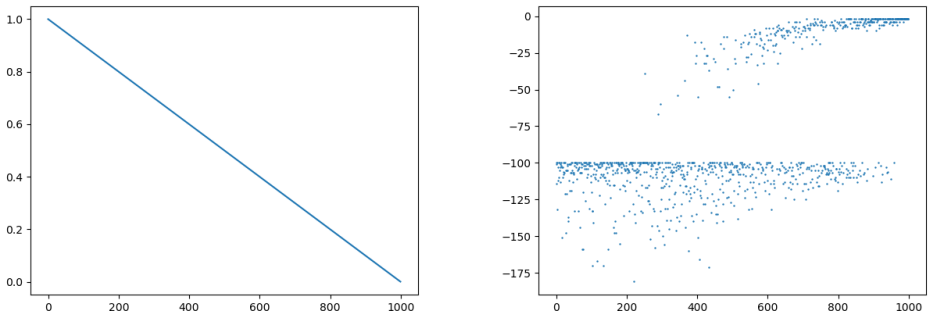
Input: The state space is S, all actions are a_1, a_2, a_3, a_4
Output: A converged solution

$Q(s, a_i) \leftarrow 0, \forall s \in S, i \in [1, 4]$;
episode $\leftarrow$ 0;
while not converged **do**
  foreach *iteration in range(500)* **do**
    Get current enviroment and state s;
    $a \leftarrow \epsilon - greedy$(s, Q)
    Take action a, observe reward R and state s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a_i} Q(s', a_i) - Q(s,a)]$
  episode +=1;

- **Result Visualization**

  a. ***ϵ-decay scheme:*** We first choose the learning rate α and reward decay γ to be 0.1 and 0.9 respectively. We try three different ϵ-decay schemes. The results are shown below.

  We first try the simplest linear function: $\epsilon = 1 - 0.001 \times episode$. When the episode reaches 1000, ϵ will fall down to 0.

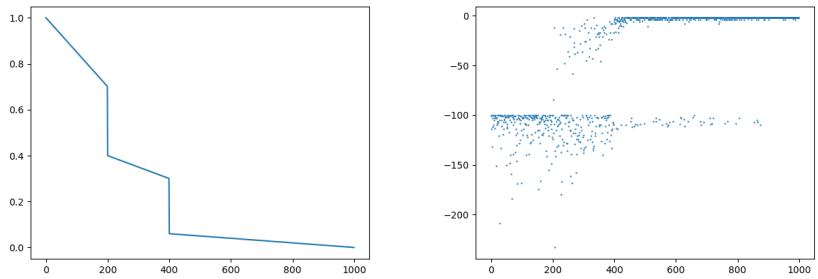


(a)ϵ-decay function (b)episode reward

FIG. 1: $\epsilon = 1 - 0.001 \times episode$

From the figure we can see that the episode reward reaches 1000 eventually, but it converges very slowly. So we choose a piecewise function to make the decay more rapid.

$$\epsilon = \begin{cases} 1 - 0.0015 \times episode & episode < 200 \\ 0.5 - 0.0005 \times episode & 200 \leq episode < 400 \\ 0.1 - 0.0001 \times episode & otherwise \end{cases}$$

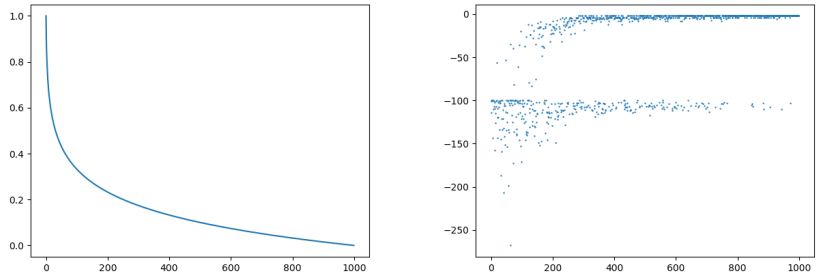


(a)ϵ-decay function (b)episode reward

FIG. 2: Piecewise Function

It finds the right path at around 300 episodes and begins to converge at around 400 episode, because from the 400th episode, ϵ becomes very small.

We try to further increase the descent rate and choose the log function $\epsilon = 1 - log_{1000}(episode + 1)$.



(a)ϵ-decay function (b)episode reward

FIG. 3: $\epsilon = 1 - log_{1000}(episode + 1)$

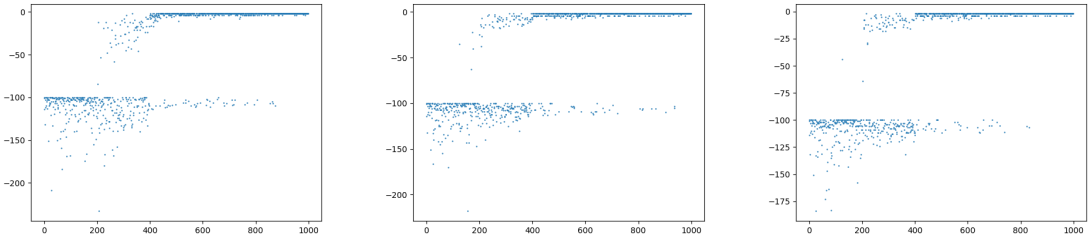
It converges more faster. However, during one experiment, we accidentally found that the path falls into a local optimal instead of global optimal, because the exploration of the agent is not enough and the ϵ already becomes small.

We plot the path we find as below. We can conclude that the agent with log function as ϵ-decay does not explore enough about the environment.

(a)First two functions (b)log function

FIG. 4: Path Visualization

b. ***learning rate*** α***:*** We now turn our attention to how does the learning rate affect the result. We choose α to be 0.1, 0.5 and 0.8 respectively and use the piecewise function as the ϵ-decay function.



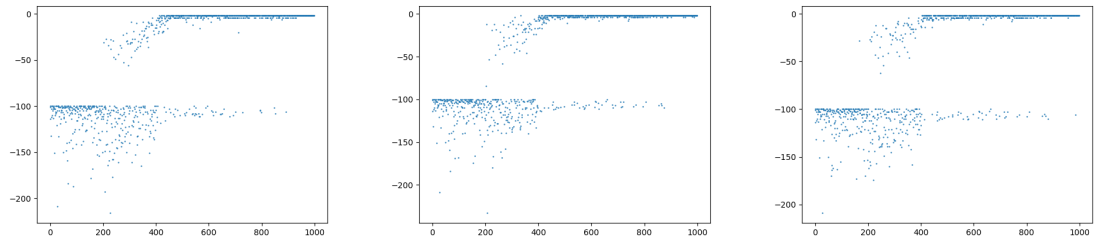
(a)$\alpha = 0.1$ (b)$\alpha = 0.5$ (c)$\alpha = 0.9$

FIG. 5: Learning Rate Visualization

A larger learning rate means to update the Q-table more quickly. The three figures look pretty similar to each other, except that with a larger α, the rewards for around 300th episodes are closer to the

final reward. This is because the environment is simple and the agent does not make some complex moves. The relative size of each state will not change hence the agent can always find the optimal path. Actually, they do find the same path as $FIG.4(a)$ shows.

c. ***reward decay*** γ***:*** Finally, we choose different γ: 0.7, 0.9 and 1. We fix the learning rate to be 1 and use the piecewise function.



(a)$\gamma = 0.7$ (b)$\gamma = 0.9$ (c)$\gamma = 1.0$

FIG. 6: Reward Decay Visualization

The three figures are almost identical in the reward distribution. This is because the environment is not complex and the exit reward (for both falling down from cliff and arriving at the exit safely) are relatively large. Even the smallest γ we try does not change the convergence of the Q-table. They also find the same path shown in $FIG.4(a)$.

- **Learning Process** The learning process of the agent is introduced briefly in the description part. The agent maintains a Q-table. In each episode, the agent first chooses an action according to ϵ-greedy method. Then the agent executes the action and gets a reward as well as the next state. The agent choose the max value in the Q-table of the next state to update the Q-table. The agent repeatedly chooses the action in the next state, moves and updates the Q-table. It stops when the process is done, for example, falling off the cliff or exit. Then the next episode begins and the agent will repeat the procedure.

  Take the piecewise function as an example. When episode is less than 200, ϵ is large and the agent explores the environment with large randomness. ϵ drops at the 200th episode and when episode is between 200 and 400, the agent begins to find a way that leads to the maximal reward. When episode is more than 400, ϵ is very small and the final reward converges.

B. SARSA

- **Description** In the SARSA algorithm, the update of Q-value is based on the next action chosen by ϵ-greedy algorithm in the next position, written as

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

  Then the action chosen for the next state is passed to the next iteration, which means at the next state, the agent do follows the action used to update. What we choose and what we do coincide. Hence, SARSA is called on policy algorithm.

- **Implementation** The implementation of agent is the same as in Q-learning. In the algorithm, the only difference is that instead of choosing the second action with the max Q-value, we choose the action using ϵ-decay and pass it to the next iteration.

Algorithm 2: SARSA Algorithm

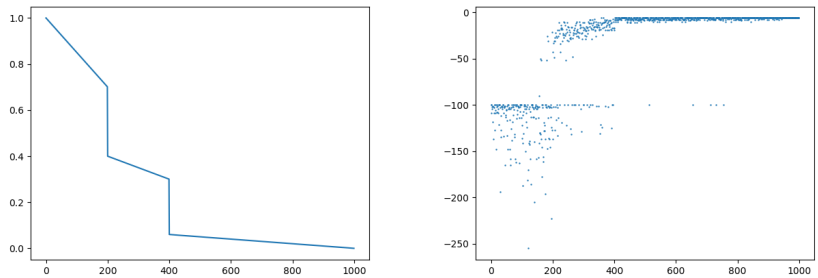
Input: The state space is S, all actions are a_1, a_2, a_3, a_4
Output: A converged solution

$Q(s, a_i) \leftarrow 0, \forall s \in S, i \in [1, 4]$;
episode $\leftarrow 0$;
while not converged **do**
  $a \leftarrow \epsilon - greedy$(s, Q)
  foreach *iteration in range(500)* **do**
    Get current environment and state s;
    Take action a, observe reward R and state s'
    $a' \leftarrow \epsilon - greedy$(s', Q)
    $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s', a') - Q(s,a)]$
    $a \leftarrow a'$
  episode +=1;

- **Result Visualization** As in the Q-learning algorithm, we choose α to be 0.1, γ to be 0.9, ϵ-decay function to be

$$\epsilon = \begin{cases} 1 - 0.0015 \times episode & episode < 200 \\ 0.5 - 0.0005 \times episode & 200 \leq episode < 400 \\ 0.1 - 0.0001 \times episode & otherwise \end{cases}$$

  This leads us to the result of episode reward



(a)ϵ-decay function (b)episode reward

FIG. 7: Result

And the final path is

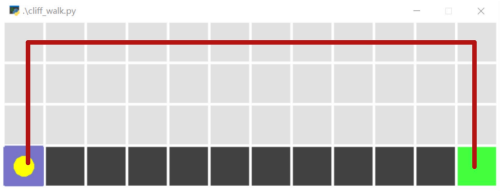


FIG. 8: Path

- **Learning Process** The learning process of the agent is introduced briefly in the description part. The agent maintains a Q-table. In each episode, the agent first chooses an action according to ϵ-greedy method. Then the agent executes the action and gets a reward as well as the next state. At the next state, the agent uses ϵ-greedy method again to choose a new action, and use the action to update the Q-table. In the new state, the agent already chooses the state, so it simply moves and chooses an action in the second new state. The agent stops when the process is done, for example, falling off the cliff or exit. Then the next episode begins and the agent will repeat the procedure.

  We can see from FIG.7(b) that there are two turning points around the 200th episode and 400th episode, where the agent begins to find the path with maximal reward and the path converges respectively. This is because at these special points ϵ has a great drop.

C. Differences of the Paths

The path found by Q-learning and SARSA algorithms are respectively

(a)Q-learning (b)SARSA

FIG. 9: Final Path

For Q-learning, it updates $Q(s,a)$ with the max value of $Q(s',a_i)$. Although some directions' Q value of a grid may be low, $max_{a_i}(Q(s',a_i))$ can still be large. In our example, although the Q value for each grid on this path is small in the downward direction, the agent is still likely to choose move right due to the seductive reward for moving directly to the target point.

For SARSA, this is because grid (1, 1) has a smaller Q value on the downward direction, which will gradually reduce the Q value of the grid $(0,1)$ in its right direction. And when the agent arrive $(0,1)$, it's more likely to choose to move upward.

III. SOKOBAN

In this section, we implement intelligent agents to play Sokoban game utilizing Sarsa, Q-learning, dyna-Q and UCB algorithms.

For the Sokoban game environment:

- state: contains three parts: agent coordinates (x_0, y_0), box 1 coordinates (x_1, y_1), box 2 coordinates (x_2, y_2), $x_i, y_i \in [0, 5)$.
- action: up, down, left and right, represented by integer 0 to 4
- reward: -0,1 as living cost, -1 for pushing a box off the target, 1 for pushing one box on a target, 10 for pushing two boxes onto the targets,

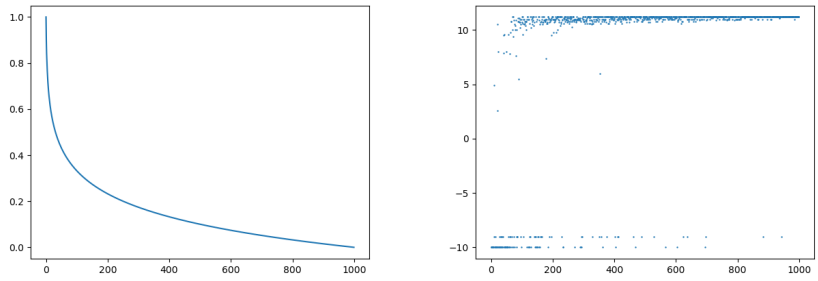
A. Q-learning && SARSA

The implementation of Q-learning and SARSA algorithms are almost the same as in the cliff walking part, except a minor change in the dimension of the Q-table and the ϵ-decay function.

For both algorithm, we choose α to be 0.1, γ to be 0.9, ϵ-decay function to be

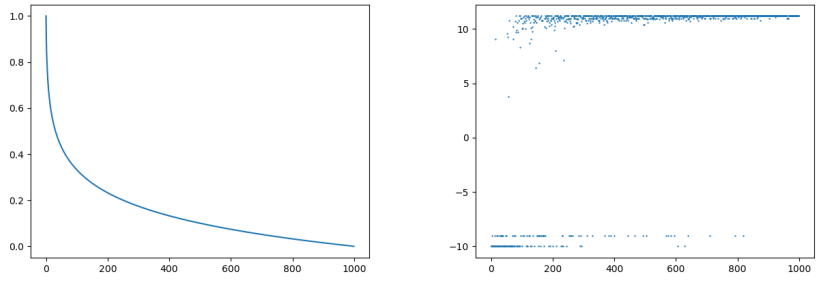
$$\epsilon = 1 - log_{1000}(episode + 1)$$

Both algorithms converge and find the right path to push the box into the target.



(a)ϵ-decay function (b)episode reward

FIG. 10: Q-learning result



(a)ϵ-decay function (b)episode reward

FIG. 11: SARSA result

B. Dyna-Q

- **Description:** Dyna-Q algorithm is similar to Q-learning, but it contains a part of pseudo model learning. It not only maintains a Q-table, but also a model. In Dyna-Q, after exploring the environment, it also updates its model. Then it randomly chooses some previously observed states and actions, and update their Q value in model.

- **Implementation:** Suppose the state space of sokoban game is S, all actions are a_1, a_2, a_3, a_4, represent move leftward, rightward, upward,downward respectively. The Q value of state s, action a_i is $Q_s(a_i)$.

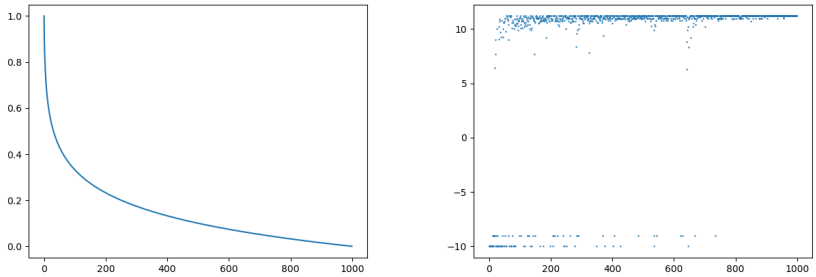
  Each time after updating the environment, we run model learning for $N = 10$ times. You can see this process in the red pseudo code below:

Algorithm 1: Dyna-Q algorithm

Input: The state space is S, all actions are a_1, a_2, a_3, a_4
Output: A converged solution

$Q(s, a_i) \leftarrow 0, \forall s \in S, i \in [1, 4]$;
$Model(s, a_i) \leftarrow \{\}, \forall s \in S, i \in [1, 4]$;
$N = 10$;
episode $\leftarrow 0$;
while not converged **do**
  foreach *iteration in range(500)* **do**
    Get current enviroment and state s;
    $a \leftarrow \epsilon - greedy$(s, Q)
    Take action a, observe reward R and state s'
    $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma max_{a_i} Q(s', a_i) - Q(s, a)]$
    $Model(s, a) \leftarrow r, s'$
    foreach *i in range(N)* **do**
      $s \leftarrow$ previously observed state
      $a \leftarrow$ random action previously taken in s.
      $r, s' \leftarrow Model(s, a)$
      $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma max_{a_i} Q(s', a_i) - Q(s, a)]$
  episode +=1;

- **Result Visualization**



(a)ϵ-decay function (b)episode reward

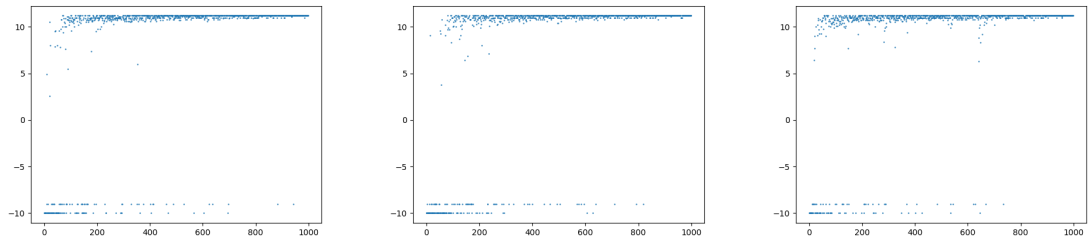
FIG. 12: dyna-Q result

C. Result Analysis

- **Learning Process** The learning processes of **Q-learning** and **SARSA** are similar to the process in cliff walking part, except the dimension of state changes. There's no need to introduce the algorithms again. Because we choose the log function as the ϵ-decay function, ϵ drops rapidly before the 100th episode, and the reward converges at around the 200th episode.

  For **dyna-Q**, the reason of adding the model is that interaction with real environment is expensive in many circumstances and to get a value fully updates, a path has to be explored several times. Learning in model is relatively cheap and the Q-value can be correctly updated. After a single movement, the agent updates the Q-value for 10 times in the model and Q-table gets updated more quickly. That why it converges more quickly (less than 100 episodes).

- **Learning Rate** All of the three algorithms use $\epsilon = 1 - log_{1000}(episode + 1)$ as the ϵ-decay function with the same parameters, and the learning speed can be shown from the below figure.



(a)Q-learning (b)SARSA (c)Dyna

FIG. 13: Learning Rate Visualization

Learning speed (also convergence speed): Dyna-Q > Q-learning > SARSA

Reason: In the Dyna-Q algorithm, when the agent makes one move, it updates the Q-table in the model for 10 times. It's obvious that it has the largest convergence speed.

Compared to the Q-learning algorithm, even if SARSA's optimal policy converges, the SARSA algorithm may still choose a random action and explore a safer policy due to its conservative, yet the Q-learning algorithms will simply follow the policy which have the max Q-value. This makes the learning speed of SARSA smaller than Q-learning.

D. Explore-Exploit Dilemma

1. *Different ϵ-decay scheme*

We select different $\epsilon - decay$ for SARSA, 15 shows the convergence of the same segmented function as before and $1 - log_{1000}(episode + 1)$.

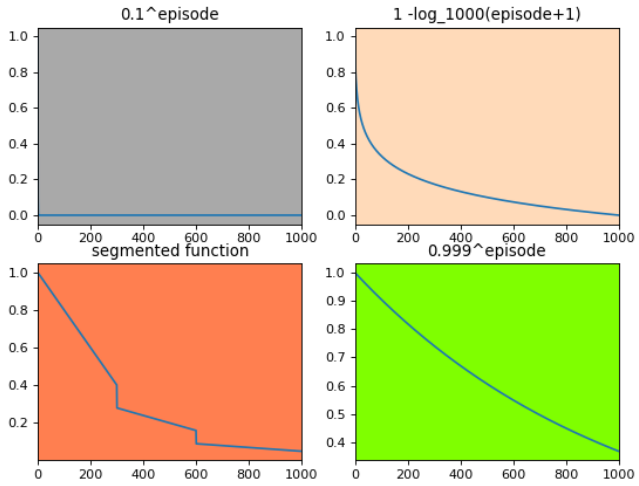
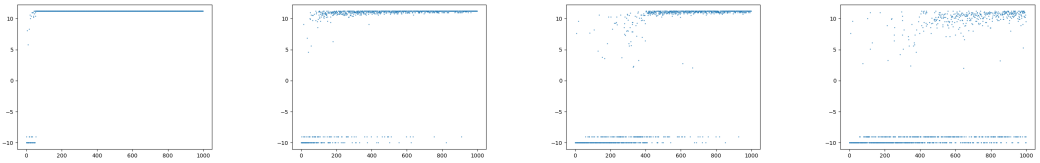


FIG. 14: ϵ-decay Function



(a)$0.1^{episode}$ (b)$1 - log_{1000}(episode + 1)$ (c)segmented function (d)$0.999^{(episode+1)}$

FIG. 15: Episode Reward

Relationship between different exploration schemes and learning speeds and result: For SARSA algorithm, if ϵ decreases faster, the agent has less randomness and hence the learning speed is faster, i.e. the episode reward converges in less episodes.

In the experiment, the final reward stays the same because the environment is simple and the agent eventually gains the same policy. However, in most cases, if ϵ drops too fast, the final policy may possibly fall into a local optimal instead of global optimal.

*2. **(Bonus)** Upper Confidence Bound*

In this section, we implement a new exploration strategy named Upper Confidence Bound.

- **Description:** The main idea of UCB algorithm is simple. At first, the actual reward of each action is unknown, we must make lots of attempts and estimates the actual reward of each action.

  It defines a confidence interval for each action at every state, the confidence interval estimates the expected reward of taking that action. And the upper bound of this interval is called UCB. When the actual reward obtained is lower or greater than expected, the confidence interval will make the corresponding changes.

  Every time we make a choice, we choose the action with the max UCB.

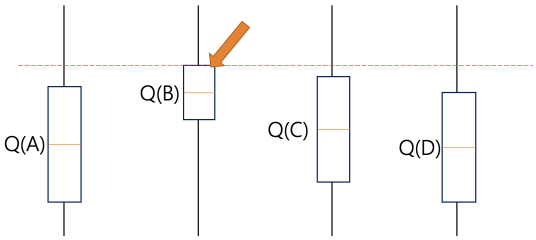


FIG. 16: choose the action with the max UCB.

- **Implementation:** Suppose the state space of sokoban game is S, all actions are a_1, a_2, a_3, a_4, represent move leftward, rightward, upward,downward respectively. The UCB of action a_i at state $s \in S$, episode $= n$ is: $UCB_{s,n}(a_i)$.

  According to the formulas of UCB algorithm,

$$UCB_{s,n}(a_i) = r_{s,n}(a_i) + c\sqrt{\frac{ln(N(s,n))}{N_{a_i}(s,n)}} \tag{1}$$

$$Action(s,n) = argmax_{a_i}(UCB_{s,n}(a_i)) \tag{2}$$

  Notes:

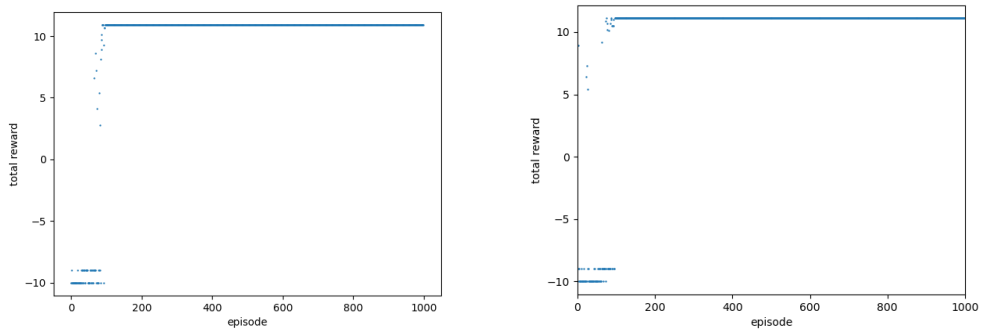
  1. At formula (1), $r_{s,n}(a_i)$ represents the average reward for action a_i at episode n, state s. The blue part is called explore, we add this term for exploring the actions not chosen yet. $N(s,n)$ represents total choice number at episode n, state s, $N_{a_i}(s,n)$ represents the total number of action a_i be chosen at episode n, state s .
  2. At formula (2), $Action(s,n)$ represents the action we choose at at episode n, state s.
  3. We also update Q table while calculating UCB and use $Q(s,a_i)$ as the actual reward of take action a_i at state s.

Algorithm 2: Sokoban UCB Algorithm

Input: The state space is S, all actions are a_1, a_2, a_3, a_4
Output: A converged solution

$r_{s,n}(a_i) \leftarrow 0, \forall s \in S, n \in [0, 1000), i \in [1, 4]$;
$Q(s, a_i) \leftarrow 0, \forall s \in S, i \in [1, 4]$;
$N(s, n) \leftarrow 0, \forall s \in S, n \in [0, 1000)$;
$N_{a_i}(s, n) \leftarrow 0, \forall s \in S, n \in [0, 1000), i \in [1, 4]$;
episode $\leftarrow 0$;
while not converged **do**
  foreach *iteration in range(500)* **do**
    Get current enviroment and state s;
    foreach *i in range(1,5)* **do**
      $UCB_{s,episode}(a_i) \leftarrow r_{s,n}(a_i) + 2\sqrt{\frac{ln(N(s,n))}{N_{a_i}(s,n)}}$
    $Action(s, episode) = argmax_{a_i}(UCB_{s,episode}(a_i))$
    $N(s, n) += 1$; $N_{Action(s,episode)}(s, episode) += 1$;
    Update $r_{s,n}(Action(s, episode))$;
    Update $Q(s, a_i)$;
  episode +=1;

- **Result Visualization and Analysis** The demo video is in the 'Demo/' folder, and the following 17 is episode-total reward convergence with batch episode =1000, c = $\sqrt{2}, 2$.



(a)c = $\sqrt{2}$ (b)c = 2

FIG. 17: episode-total reward

We can see that the UCB algorithm combined with Q-learning converges very fast, and it also operates very fast in actual running .

IV. CONCLUSION

In this lab, we focus on reinforcement learning and implement Q-learning, SARSA, Dyna-Q and UCB algorithms. We apply these algorithms on cliff walking and sokoban game. It's amazing to watch the little agents explore the world and form a optimal policy in the end. We also try different parameters, which really takes us some time, yet we gain a better understanding about how the parameters affect the agent's policy and learning speed. We also try different exploration exploitation strategies for ϵ-greedy, and implement UCB, which has a better result.