MLMI7-report

Tony RuiKang OuYang

Abstract

This is the course work report of *MLMI 7: Reinforcement Learning and Decision Making*, where codes are open-sourced in my GitHub repo: https://github.com/tonyauyeung/MLMI7-Reinforcement-Learning. Aknowledge to the RL-textbook[1].

Contents

1 Question 1

In this question, we first implement Value Iteration. In details, we update the value function V by code1 for each iteration.

```
for _ in tqdm(range(maxit)):
    V_new = np.zeros_like(V)
    for s in model.states:
        values = []
        for a in Actions:
            R = model.reward(s, a)
            values.append(compute_value(s, a, lambda *_: R))
        V_new[s] = max(values)
    delta = np.max(np.abs(V_new - V))
    V = np.copy(V_new)
    if delta <= threshold:
        break
```

Listing 1: Value Iteration

1.1 Policy Iteration v.s. Value Iteration

While policy iteration is terminated when the policy $\pi(\cdot|s)$ for every state s converges (i.e. $\forall(a,s) \in A \times S, \pi_k(a|s) = \pi_{k+1}(a|s)$, where k is the index of iteration), we terminate the Value Iteration when the maximum changes of the value function is less than a threshold ϵ, i.e. $\max_{s\in S} \|V_{k+1}(s) - V_k(s)\| < \epsilon$.

We run Value Iteration in the grid world, small world and cliff world respectively, setting maximum iteration 100, a l_1 distance to measure the changes and threshold 0.001. For comparison, we also run Policy Iteration in these three worlds with maximum iteration 100.

Figure1 shows that the optimal policies obtained by Value Iteration and Policy Iteration are exactly the same, on each world configuration, since both of them are guaranteed to converge to the optimal policy. And both the Value Iteration and Policy Iteration could find the optimal path from initial state to goal state under each configuration.

Further more, we explore the convergence of both these two algorithms. Policy iteration is optimizing the policy $\pi(a|s)$ directly and thus it converges towards a stable distribution of action given state directly. We thus measure the convergence of Policy Iteration via the number of action changes for each iteration. While Value Iteration optimizes the value function and is terminated when it doesn't change (with a small tolerant ϵ), we measure its convergence via the L_1 distance of V-value changes at each iteration.

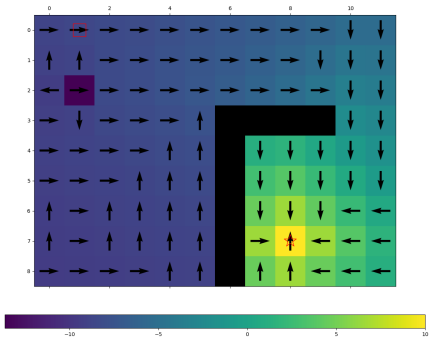
Figure2 shows the convergence of Policy Iteration and Value Iteration respectively. It is shown that Policy Iteration would converge in fewer iterations compared with Value Iteration.

1.2 Computational Complexity between Policy Iteration and Value Iteration

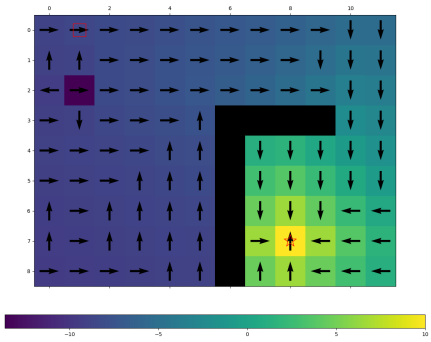
Alternatively, we would like to compare the computational efficiency between Policy Iteration and Value Iteration as well. Suppose there are $|S|$ states and $|A|$ possible actions, one step policy evaluation requires $O(|S|^2)$ computation and one step policy improvement requires $O(|S|^2|A|)$ computation; suppose that one Policy Iteration is composed with n steps policy evaluation and one step policy improvement, then the computational complexity of one iteration of Policy Iteration would be:

$$O((n + |A|)|S|^2) \tag{1}$$

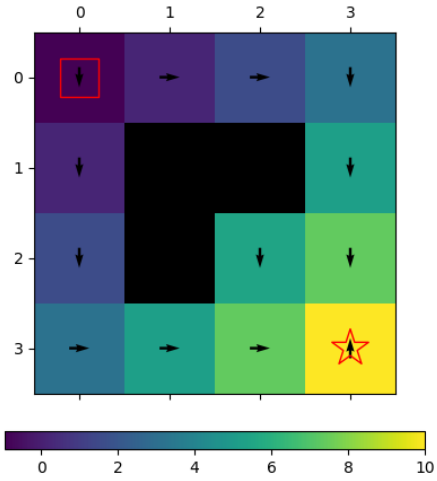
For value iteration, we consider all possible actions given a state s and update the value function $v(s)$ according to the maximum predicted return. Thus the computational complexity



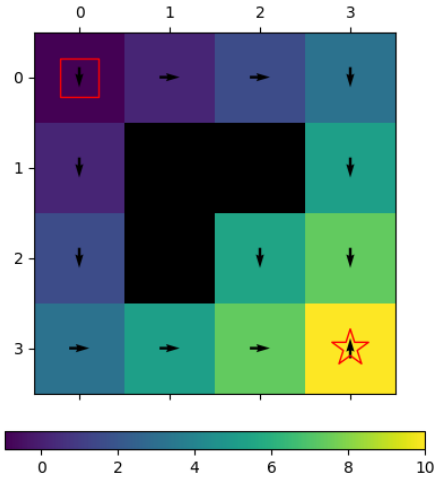
(a) Value Iteration on Grid world



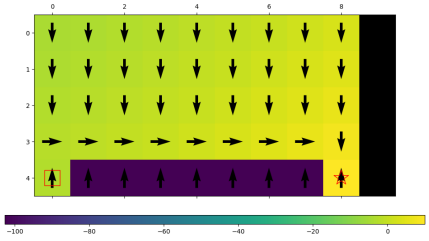
(b) Policy Iteration on Grid world



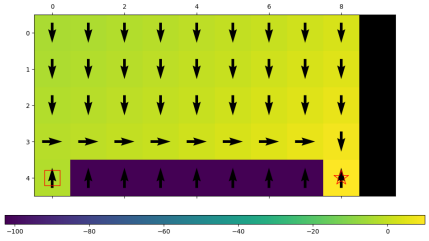
(c) Value Iteration on small world



(d) Policy Iteration on small world



(e) Value Iteration on cliff world



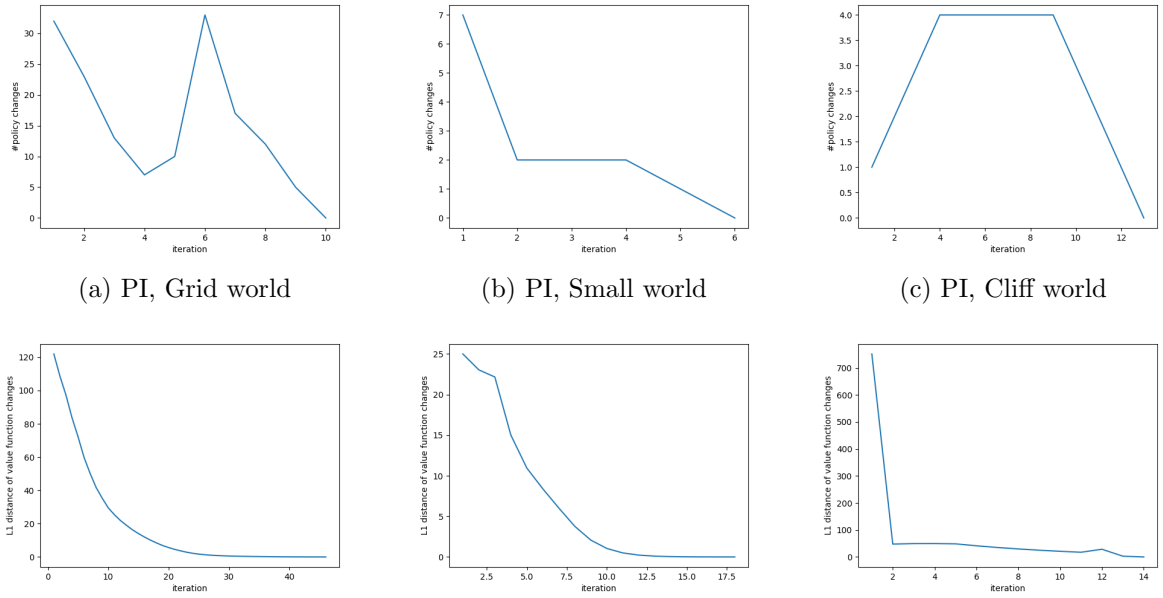
(f) Policy Iteration on cliff world

Figure 1: Comparison of the optimal policy on different world obtained by Value Iteration and Policy Iteration. UP: Grid; MIDDLE: Small; BOTTOM: Cliff.

of one iteration of Value Iteration would be:

$$O(|A||S|^2) \tag{2}$$

Though Policy Iteration is more complex than Value Iteration in each iteration, it converges faster in general, where results in figure2 support this conclusion. Thus Policy Iteraion is more efficient than Value Iteration.



(a) PI, Grid world (b) PI, Small world (c) PI, Cliff world

(d) VI, Grid world (e) VI, Small world (f) VI, Cliff world

Figure 2: Comparison of the Policy Iteration and Value Iteration Convergence on each world. UP: Policy Iteration; BOTTOM: Value Iteration.

1.3 Synchronise and Asynchronise Value Iteration

Instead of updating the value function after seeing all the states, Asynchronise Value Iteration updates the value function immediately during the for-loop of states (see code2) which is more efficient:

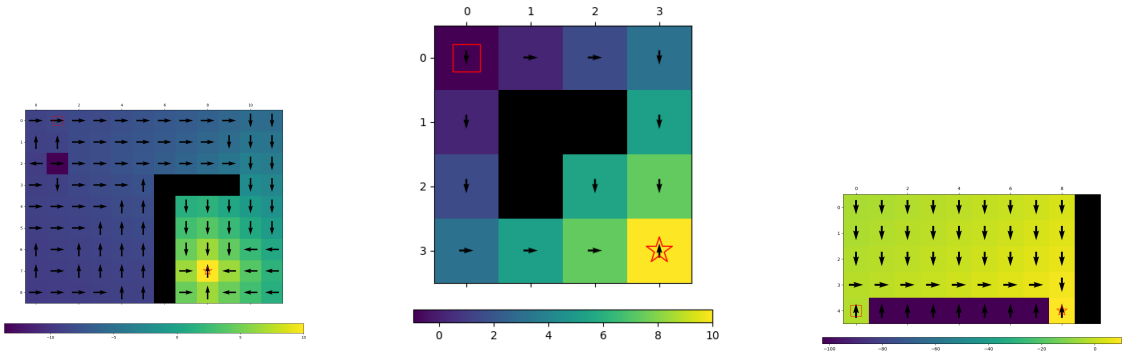
```
for _ in tqdm(range(maxit)):
    delta = -1e+6
    for s in model.states:
        values = []
        for a in Actions:
            R = model.reward(s, a)
            values.append(compute_value(s, a, lambda *_: R))
        delta = max(delta, np.abs(V[s] - max(values)))
        V[s] = max(values)
    if delta <= threshold:
        break
```

Listing 2: Asynchronise Value Iteration

We then compare the optimal policy and convergence of these two different Value Iteration algorithms. Figure3 shows that Asynchonise Value Iteration obtains the same optimal policy compared with synchonise one compared to figure1. While figure4 shows that the asynchronise Value Iteration acts similar to Value Iteration in terms of convergence. And thus, the asynchronize one is much more efficient as it doesn't need store the old value function V (a vector in python with space complexity $O(|S|)$) but update it directly. When the number of states are large, then it would reduce the memory usage a lot which is desirable.

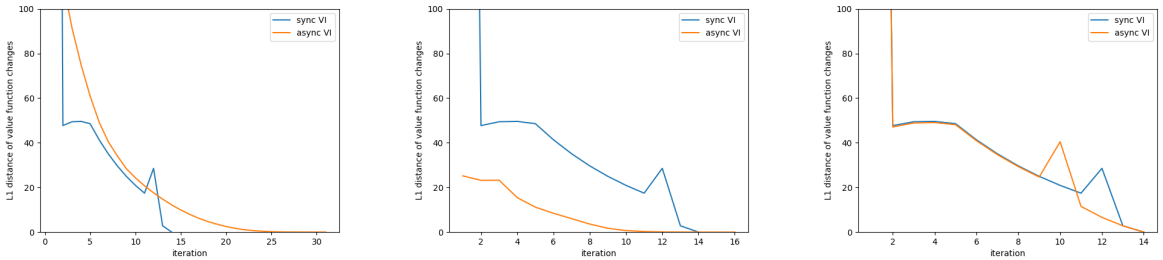
2 SARSA

In this question, we explore the tabular Q-value based RL method - State-Action-Reward-State-Action (SARSA). We first implement SARSA and code3 shows the key steps in the



(a) Grid world (b) Small world (c) Cliff world

Figure 3: The optimal policy on different world obtained by Asynchonise Value Iteration.



(a) Grid world (b) Small world (c) Cliff world

Figure 4: Comparison of convergence of asynchronise and synchonise Value Iteration on different world.

implementation.

```
for i in range(n_episode):
    s = model.start_state
    coin = np.random.choice([0, 1], size=1, p=[1 - epsilon, epsilon])
    a = np.random.randint(0, len(Actions)) if coin else np.argmax(Q[s, :])
    for _ in range(maxit):
        r = model.reward(s, a)
        s_ = model.next_state(s, a)
        coin = np.random.choice([0, 1], size=1, p=[1 - epsilon, epsilon])
        a_ = np.random.randint(0, len(Actions)) if coin else np.argmax(Q[s_, :])
        Q[s, a] = Q[s, a] + alpha * (r + model.gamma * Q[s_, a_] - Q[s, a])
        s, a = s_, a_
        if s == model.goal_state:
            break
```

Listing 3: SARSA

We then run a default experiment on small world with 1000 episodes, $\alpha = 0.1$, $\epsilon = 0.1$ and $maxit = 100$ (the maximum length of each episode). Figure5 shows the policy learned from SARSA, which achieves the optimality.

2.1 Settings of hyper-parameters

In this subsection, we discuss the setting of the four hyper-parameters we mentioned:

1. α is the learning rate of SARSA, which controls the "step-size" of the Q-value function updates in each iteration. Thus, a large α might result to divergence of SARSA; while a

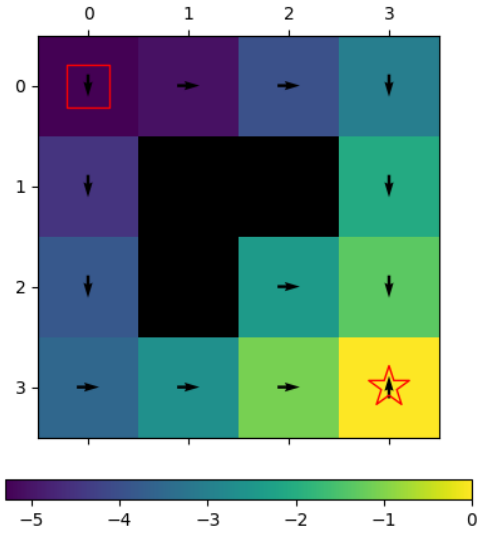


Figure 5: The optimal policy obtained by SARSA on Grid World

small α might result to a very slow convergence rate. In our default setting, we choose a moderate value $\alpha = 0.1$.

2. ϵ controls the trade-off between exploitation and exploration. A large ϵ results to a high probability to choose a random action, which means exploration more; while a small ϵ means exploiting the experience more, which might cause the model to be over-confident.

3. n_epsidoe is the number of iteration of Q-value updates, where we should use a moderate number to ensure the convergence of SARSA.

4. maxit is the maximum length of episode in each iteration. To make sure the algorithm could find the optimal policy from source to goal, the maxit should at least as long as the shortest path from source to goal.

2.2 Hyper-parameter tuning

To find the optimal α and ϵ, we run multiple experiments with different hyper-parameter setting. Figure6, where a large slope refers faster convergence, shows that a small value of ϵ results to faster convergence as the slope of the plot would stablelize to higher value. Especially, $\epsilon = 0$ results to the fastest convergence because the small world environment is too easy to learn that we're not necessary to explore; while too much exploration would be harmful (i.e. $\epsilon > 0.5$) and result to divergence. On the other hand, $\alpha = 0.3$ is the best choice which is a moderate option, where large α ($\alpha \geq 2$) and small ones ($\alpha = 0.01$) would perform bad.

2.3 Extreme cases of hyper-parameters

Further more, we explore SARSA's performance under different extreme hyper-parameter settings:

Figure7 shows that: 1. SARAS can't find the optimal policy under extreme αs; 2. a fully exploitation strategy (i.e. $\alpha = 0$) can still find the optimal policy, while an almost random strategy (i.e. $\alpha = 0.99$) results to disconvergence; 3. Under small value of episodes and maxit, SARSA can't find the optimal policy.

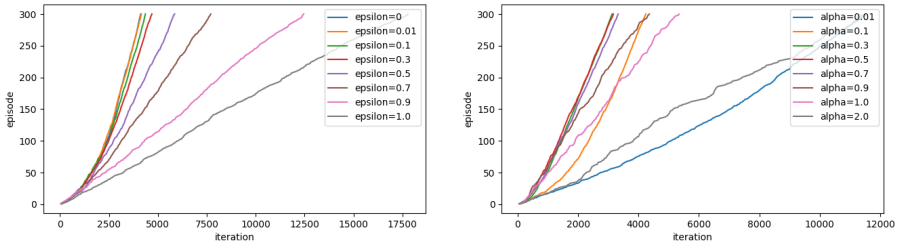


Figure 6: SARSA convergence under different parameter settings

2.4 Expected SARSA

In this subsection, we implemented the expected SARSA in code4

```
for i in range(n_episode):
    s = model.start_state
    coin = np.random.choice([0, 1], size=1, p=[1 - epsilon, epsilon])
    a = np.random.randint(0, len(Actions)) if coin else np.argmax(Q[s, :])
    for _ in range(maxit):
        r = model.reward(s, a)
        rewards[i] += r
        s_ = model.next_state(s, a)
        coin = np.random.choice([0, 1], size=1, p=[1 - epsilon, epsilon])
        a_ = np.random.randint(0, len(Actions)) if coin else np.argmax(Q[s_, :])
        a_s = np.where(Q[s_, :] == np.max(Q[s_, :]))[0]
        expected_q = 0.
        for j in range(4):
            expected_q += ((1 - epsilon) + epsilon / len(Actions)) *
                          Q[s_, j] if j in a_s else epsilon / len(Actions) * Q[s_, j]

        Q[s, a] = Q[s, a] + alpha * (r + model.gamma * expected_q - Q[s, a])

        s, a = s_, a_
        if s == model.goal_state:
            break
```

Listing 4: Expected SARSA

Expected SARSA can be a little bit more computationally expensive than SARSA in one iteration because it involves an additional step of estimating the expected value of a policy. However, this additional complexity can lead to faster learning (faster convergence) and improved performance. Figure8 shows that Expected SARSA can converge fsater than SARSA. This is because Expected SARSA incorporates the expected value of a policy, which can help guide the exploration and exploitation strategy.

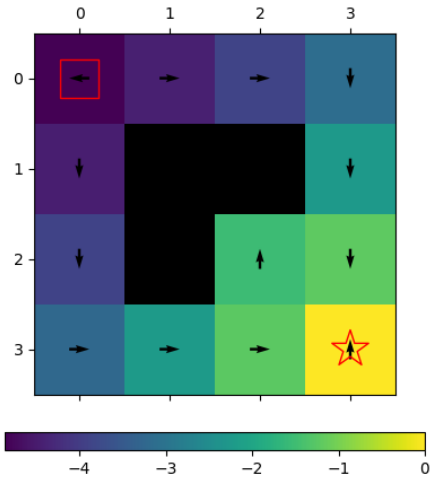
Thus, expected SARSA is more efficient than SARSA.

3 Question 3

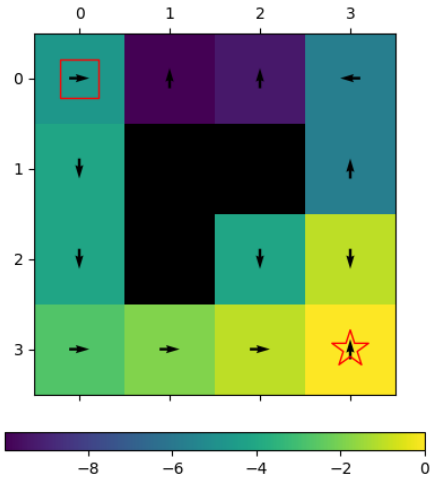
In this question, we implement Q-learning, which is shown in code5:

Analogous to the hyper-parameter tuning in SARSA, figure9 shows that $\alpha = 0.5$ and $\epsilon = 0.01$ result to the fastest convergence rate. And figure10 shows the optimal policies found by Q-learning and SARSA under this setting, which shows that both of them are able to find the optimal path in the small world configuration.

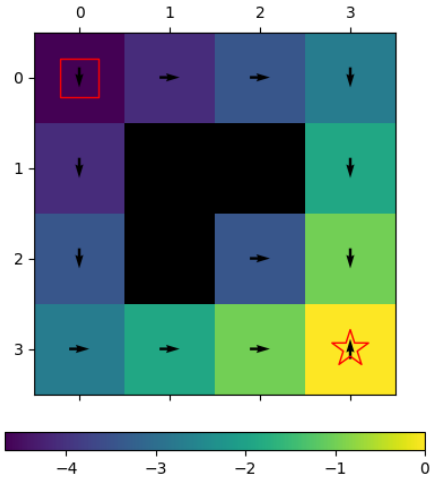
Moreover, we compare the convergence rate between Q-learning and SARSA in figure11. The slope there shows that Q-learning could converge faster than SARSA, which aligns to the difference between these two models: Q-learning can converge faster to the optimal policy since it always considers the best future action regardless of the exploration strategy; while SARSA



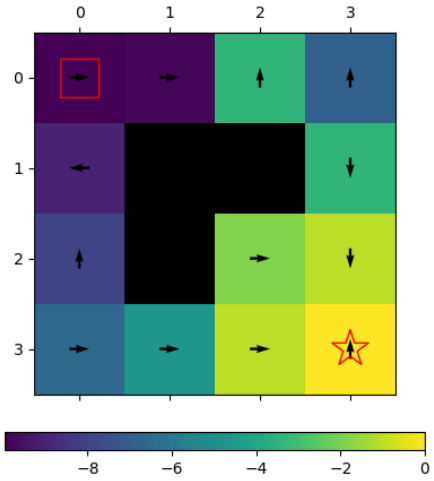
(a) $\alpha = 0.001$



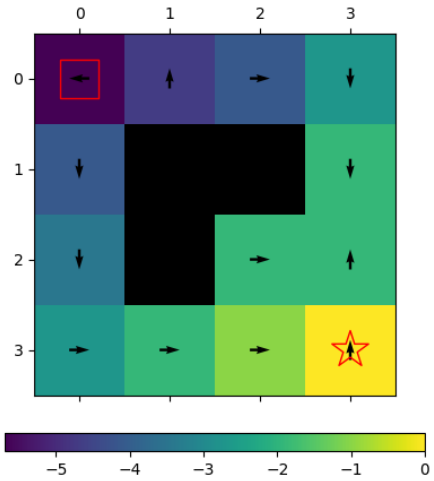
(b) $\alpha = 2.0$



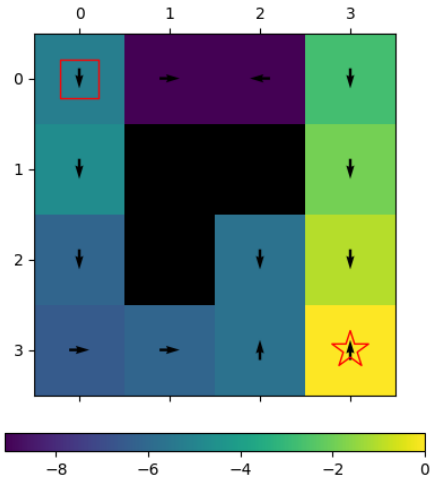
(c) $\epsilon = 0.$



(d) $\epsilon = 0.99$



(e) n_episode= 10



(f) maxit = 10

Figure 7: Comparison of found policy in small world, under extreme cases of hyper-parameters , using SARSA.

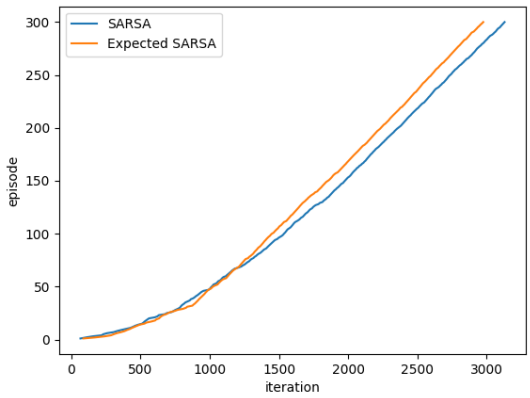


Figure 8: Convergence rate comparison of SARSA and Expected SARSA

```
for i in range(n_episode):
    s = model.start_state
    for _ in range(maxit):
        coin = np.random.choice([0, 1], size=1, p=[1 - epsilon, epsilon])
        a = np.random.randint(0, len(Actions)) if coin else np.argmax(Q[s, :])
        r = model.reward(s, a)
        rewards[i] += r
        s_ = model.next_state(s, a)
        Q[s, a] = Q[s, a] + alpha * (r + model.gamma * np.max(Q[s_, :]) - Q[s, a])
        s = s_
        if s == model.goal_state:
            break
```

Listing 5: Q-learning

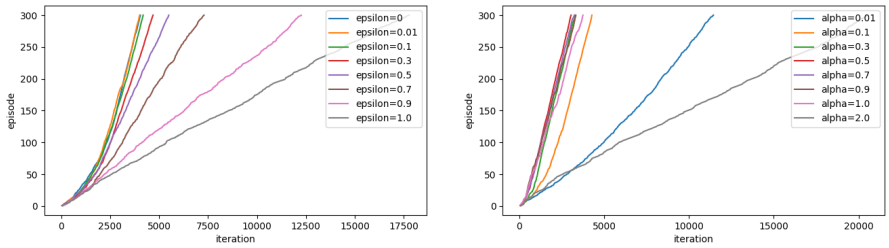


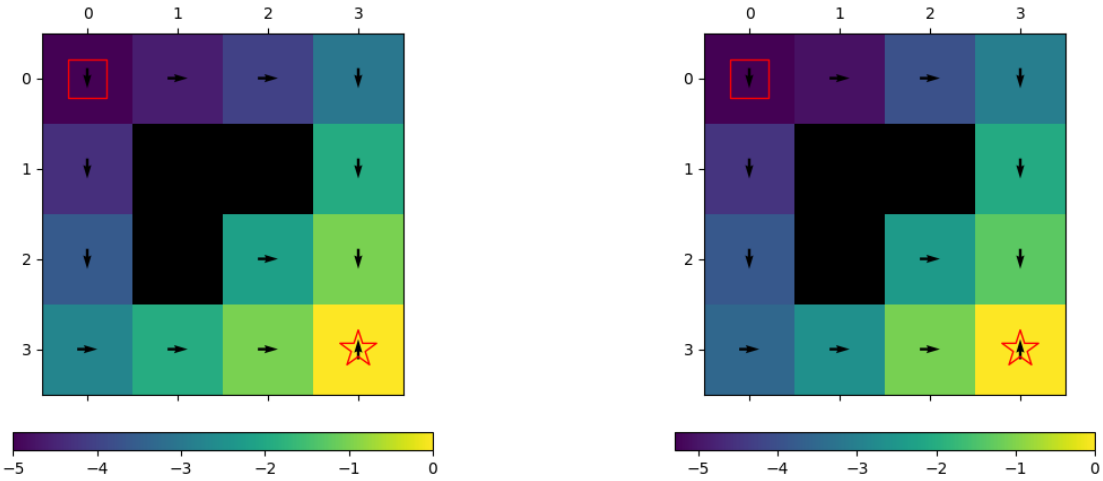
Figure 9: Q-Learning convergence under different parameter settings

might learn slower due to its on-policy nature, making it vary with the policy's exploration strategy. Thus, Q-learning might be more "efficient" in environments where the objective is to learn the optimal policy as quickly as possible without concern for the risks associated with exploratory actions.

4 Question 4

In this section, we modified the codes and store the accumulative rewards of each episode in both Q-learning and SARSA and then run experiments in the Cliff world. Since the performance of Q-learning and SARSA would be influenced by the exploration parameter ϵ, we use different ϵs (i.e., 0, 0.01, 0.1 and 0.3) and fixed $\alpha = 0.3$ for experiments.

Figure12 shows that, SARSA tends to make safer choices, as it is an on-policy algorithm and considers the policy's explorations which might be risky; While Q-learning is an off-policy algorithm, aiming for the optimal policy without regard to exploration risks, can sometimes



(a) Q-learning

(b) SARSA

Figure 10: Optimal policy found by Q-learning and SARSA with $\alpha = 0.5$ and $\epsilon = 0.1$

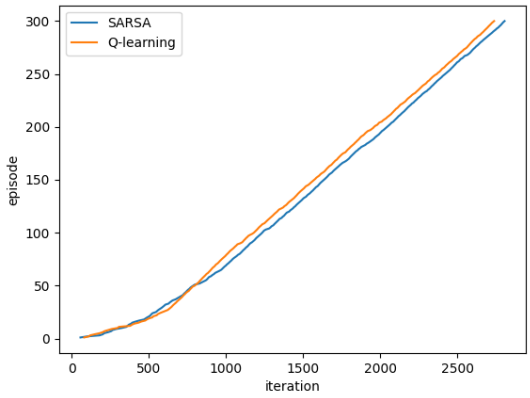


Figure 11: Convergence rate of Q-learning and SARSA

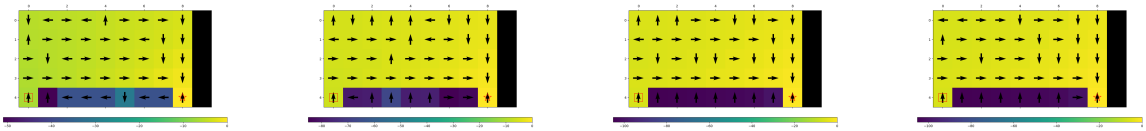
engage in risky moves if they lead to higher rewards.

As a result (see figure13), SARSA has higher accumulative rewards per episode as it always chooses a safer action; while Q-learning would have lower rewards since it's more risky. Besides, as the degree of exploration increases (i.e. ϵ increases), the gap of accumulative rewards between SARSA and Q-learning would increase since the exploratory of model is increasing and thus the risky actions in Q-learning are more likely to result in a "bad" states. Also, when the degree of model exploration is small (tends to 0, i.e. $\epsilon = 0.01$), the gap of accumulative rewards between these two algorithm vanishes; and when the model is fully-exploitation (i.e. $\epsilon = 0$), then Q-learning is equivalent to SARSA, where they obtain the same optimal policy and accumulative rewards.

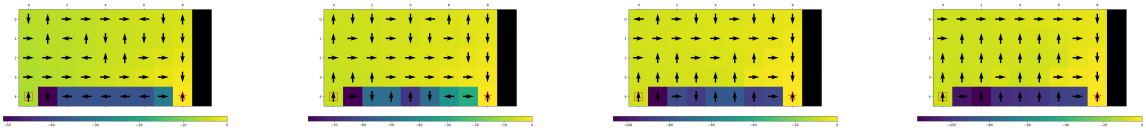
5 Question 5

5.1 Why function approximation?

In this section, we discuss the usage of function approximation in the small world with varying barries. Since whether the 3 cells contain barriers is sampled independently, there're 2^3 cases

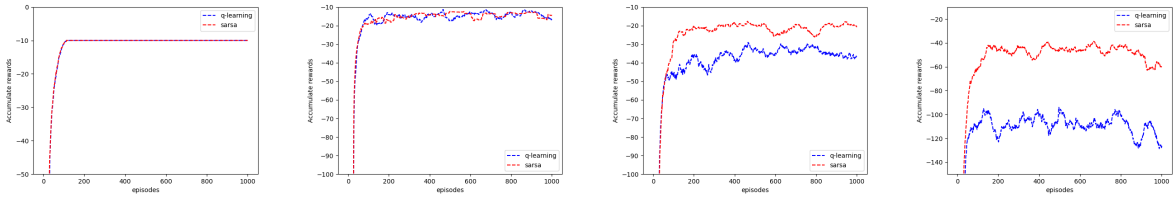


(a) Q-learning, $\epsilon = 0$ (b) Q-learning, $\epsilon = 0.01$ (c) Q-learning, $\epsilon = 0.1$ (d) Q-learning, $\epsilon = 0.3$



(e) Q-learning, $\epsilon = 0$ (f) SARSA, $\epsilon = 0.01$ (g) SARSA, $\epsilon = 0.1$ (h) SARSA, $\epsilon = 0.3$

Figure 12: Optimal policy found by Q-learning and SARSA with $\alpha = 0.3$ and different ϵs in cliff world.



(a) $\epsilon = 0$ (b) $\epsilon = 0.01$ (c) $\epsilon = 0.1$ (d) $\epsilon = 0.3$

Figure 13: Accumulative rewards per episode of Q-learning and SARSA with $\alpha = 0.3$ and different ϵs in cliff world.

and thus result to a world contains $16 \times 2^3 = 128$ states.

In Value Iteration, we have to maintain a V-table; while in SARSA and Q-learning, we have to maintain a Q-table. Both of them scale to $O(|S|)$ when the number of states are large. Besides, It requires us to visit every reachable state many times (and apply every action, for tabular Q-value methods) many times to get a good estimate of $V(s)$(or $Q(s, a)$). Thus, if we never visit a state s, we have no estimate of $V(s)$ (or $Q(s, a)$), even if we have visited states that are very similar to s.

In such case, using function approximation, the value function $V(s)$ could be estimated by $\hat{V}(s, W)$ where W is the parameters that we're fitting. And usually, we choose a feature mapping $\phi : S \to \mathcal{R}^D$ ($D << |S|$) and apply linear regression over the mapping space:

$$\hat{V}(s, W) = \sum_{d=1}^{D} W_d \phi_d(s) \tag{3}$$

Leveraging function approximation would help us: 1. save the memory usage by avoiding saving the Q-table; 2. could handle unseen states which is impossible for tabular V(or Q)-value methods.

And thus, in this modified small world, we could use function approximation to avoid saving a 8-times larger V-table, which is much more efficient.

5.2 How to apply function approximation in the modified small world?

Using function approximation in this world is more efficient, but a simple linear function approximation using $\phi(s) = s$ is not sufficient. Since the state space is now $S \times \mathbf{Z}_2^3$, the simple s could not encode information about locations of barriers. Thus, we at least need $\hat{s} = (s, z_1, z_2, z_3)$ to encode information, where z_i is indicators that clarify where the cell-i is barrier ($i = 1, 2, 3$).

However, this is still not enough. If we simply use a linear feature (linear basis function), then the prediction of V-value would be

$$\hat{V}(\hat{s}) = \hat{w}^T \phi(\hat{s}) = \hat{w}^T s + w_1 z_1 + w_2 z_2 + w_3 z_3 \tag{4}$$

It is obvious that, no matter which cell is barrier (i.e. $\forall z_i \in \{0, 1\}, i = 1, 2, 3$), the approximated value at the initial state should be the same, i.e.:

$$\hat{w}^{*T} s_0 + w_1^* z_1 + w_2^* z_2 + w_3^* z_3 \equiv C, \forall z_i \tag{5}$$

$$\Longrightarrow w_1^* = w_2^* = w_3^* = 0 \tag{6}$$

thus, a simple linear feature would not be helpful. Instead, we should consider non-linear interaction among the current location information and the barriers' information and thus we should choose another basis function here. For example, we could consider a polynomial basis function like $\phi(s) = (1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1^2 s_2, s_1 s_2^2)$, where we could generally formulate as:

$$\phi(s) = (\phi_1(s), ..., \phi_D(s)) \tag{7}$$

$$\phi_d(s) = \Pi_{j=1}^k s_j^{c_{ij}} \tag{8}$$

$$c_{ij} \in \{0, 1, ..., n\} \tag{9}$$

These features make up the order-n polynomial basis for dimension k, which contains $(n+1)k$ different features.

In summary, function approximation could help us to save large computational expense and appropriate basis function could boost the performance of linear approximation to achieve competitive (or even better) policy as tabular-based methods.

References

[1] Sutton, R. S., and Barto, A. G. *Reinforcement Learning: An Introduction.* A Bradford Book, Cambridge, MA, USA, 2018.