

Reinforcement Learning

Assignment #1 Report

Adam Tomc (2024017874)

2026-10-28

1. Exercise 1

- (a) I implemented policy iteration using iterative policy evaluation. Each iteration performs policy evaluation and then policy improvement. An example of both is shown below for iteration 1 and the start state $S = (3, 0)$.

Policy evaluation:

$$\begin{aligned}v_{\pi_0}(S) &= -0.45 + 0.95 [0.5 v_{\pi_0}(2, 0) + 0.25 v_{\pi_0}(3, 0) + 0.25 v_{\pi_0}(3, 1)] \\&= -0.45 + 0.95 [0.5(-3.73594296) + 0.25(-4.18560263) + 0.25(-4.07136464)] \\&= -0.45 + 0.95(-3.93221330) = -4.18560263\end{aligned}$$

Policy improvement:

$$\begin{aligned}q_{\pi_0}(S, U) &= -0.45 + 0.95 [0.5(-3.7359) + 0.25(-4.1856) + 0.25(-4.0714)] = -4.1856 \\q_{\pi_0}(S, D) &= -0.45 + 0.95 [0.75(-4.1856) + 0.25(-4.0714)] = -4.3992 \\q_{\pi_0}(S, L) &= -0.45 + 0.95 [0.75(-4.1856) + 0.25(-3.7359)] = -4.3195 \\q_{\pi_0}(S, R) &= -0.45 + 0.95 [0.5(-4.0714) + 0.25(-3.7359) + 0.25(-4.1856)] = -4.2653\end{aligned}$$

U has the largest value, so $\pi_1(S) = \arg \max_a q_{\pi_0}(S, a) = U$.

Both $V(s)$ and $\pi(s)$ produced by the program match the exact solution in Q1.ipynb.

ex1.py:

```
import numpy as np

pdir = 0.5
pper = 0.25
sr = -0.45
gamma = 0.95
theta = 1e-12

pi = np.full((4, 4), 'U')
pi[0, 2] = pi[0, 3] = pi[2, 1] = 'X'

V = np.zeros((4, 4))
V[0, 2] = 1
V[0, 3] = -1
V[2, 1] = 2

acts = ['U', 'D', 'L', 'R']

states = [(sy, sx) for sy in range(4) for sx in range(4)]

def states_for_action(s, a):
    if a == 'U':
        return [
```

```

        ((s[0] - 1, s[1]), pdir),
        ((s[0], s[1] - 1), pper),
        ((s[0], s[1] + 1), pper)
    ]

    if a == 'D':
        return [
            ((s[0] + 1, s[1]), pdir),
            ((s[0], s[1] - 1), pper),
            ((s[0], s[1] + 1), pper)
        ]

    if a == 'L':
        return [
            ((s[0], s[1] - 1), pdir),
            ((s[0] - 1, s[1]), pper),
            ((s[0] + 1, s[1]), pper)
        ]

    if a == 'R':
        return [
            ((s[0], s[1] + 1), pdir),
            ((s[0] - 1, s[1]), pper),
            ((s[0] + 1, s[1]), pper)
        ]

    raise Exception("Unreachable")

def p(sd, s, a):
    cds = states_for_action(s, a)

    fcds = [
        c if c[0][0] >= 0 and c[0][0] <= 3 and c[0][1] >= 0
        and c[0][1] <= 3 and (c[0] != (2, 1)) else (s, c[1])
        for c in cds
    ]

    return sum([c[1] for c in fcds if c[0] == sd])

for it in range(3):
    print()
    print(f"Iteration #{it + 1}:")

    while True:
        delta = 0
        for s in states:
            if s in [(2, 1), (0, 2), (0, 3)]:
                continue

            v = V[s]
            V[s] = sum([p(sd, s, pi[s]) * (sr + gamma * V[sd])
                        for sd in states])
            delta = max(delta, abs(v - V[s]))
        if delta < theta:
            break

    print()
    print("V(s):")

```

```

print(V)

for s in states:
    if s in [(2, 1), (0, 2), (0, 3)]:
        continue

    old = pi[s]
    pi[s] = max(
        acts,
        key=lambda a: sum([p(sd, s, a) * (sr + gamma * V[sd])
                           for sd in states])
    )
print()
print("pi[s]")
print(pi)

```

Below are the results of my program compared with those produced by Q1.ipynb.

Iteration 1:

-3.03305785	-1.7768595	1.	-1.
-3.18183169	-2.25450184	-0.86235177	-1.48171612
-3.73594296	W	-1.77045549	-2.06465356
-4.18560263	-4.07136464	-2.91951921	-2.7857

ex1.py: $v_{\pi_0}(s)$

R	R	X	X
R	R	U	U
U	X	U	U
U	R	R	U

ex1.py: $\pi_1(s)$

-3.033	-1.777	1.000	-1.000
-3.182	-2.255	-0.862	-1.482
-3.736	W	-1.770	-2.065
-4.186	-4.071	-2.920	-2.786

Q1.ipynb: $v_{\pi_0}(s)$

→	→	G	H
→	→	↑	↑
↑	W	↑	↑
↑	→	→	↑

Q1.ipynb: $\pi_1(s)$

Iteration 2:

-1.34533519	-0.28444674	1.	-1.
-1.85560371	-1.01848691	-0.54534771	-1.38297716
-2.5360224	W	-1.53054331	-1.92841729
-3.20022124	-3.30761287	-2.70841423	-2.6350775

ex1.py: $v_{\pi_1}(s)$

R	R	X	X
U	R	U	U
U	X	U	U
U	R	U	U

ex1.py: $\pi_2(s)$

-1.345	-0.284	1.000	-1.000
-1.856	-1.018	-0.545	-1.383
-2.536	W	-1.531	-1.928
-3.200	-3.308	-2.708	-2.635

Q1.ipynb: $v_{\pi_1}(s)$

→	→	G	H
↑	→	↑	↑
↑	W	↑	↑
↑	→	↑	↑

Q1.ipynb: $\pi_2(s)$

Iteration 3:

-1.3027747	-0.28444674	1.	-1.
-1.71896213	-1.01848691	-0.54534771	-1.38297716
-2.41239431	W	-1.53054331	-1.92841729
-3.07564676	-3.15491941	-2.53964777	-2.5825109

ex1.py: $v_{\pi_2}(s)$

R	R	X	X
U	R	U	U
U	X	U	U
U	R	U	U

ex1.py: $\pi_3(s)$

-1.303	-0.284	1.000	-1.000
-1.719	-1.018	-0.545	-1.383
-2.412	W	-1.531	-1.928
-3.076	-3.155	-2.540	-2.583

Q1.ipynb: $v_{\pi_2}(s)$

→	→	G	H
↑	→	↑	↑
↑	W	↑	↑
↑	→	↑	↑

Q1.ipynb: $\pi_3(s)$

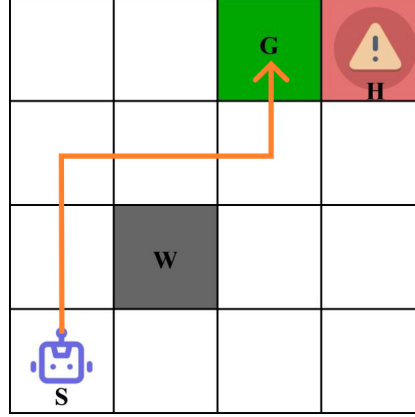
- (b) When Q1.ipynb is run with $\gamma = 0.1$, the policy converges in 4 iterations. The value function and policy are:

-0.496038	-0.422927	1	-1
-0.499609	-0.494168	-0.424806	-0.498060
-0.499979	W	-0.496138	-0.499758
-0.499999	-0.499990	-0.499806	-0.499983

$v_4(s)$

→	→	G	H
→	→	↑	↓
↑	W	↑	←
↑	→	↑	↑

$\pi_4(s)$



Route from S to G for $\gamma = 0.1$

2. Exercise 2

- (a) The transition probability matrix is:

(s, a)	Tired	Energetic	Healthy
(Tired, Work)	1.0	0	0
(Tired, Gym)	0.5	0.5	0
(Tired, Sleep)	0.2	0.8	0
(Energetic, Gym)	0	0	1.0
(Energetic, Work)	0.8	0.2	0
(Healthy, Work)	1.0	0	0

- (b) Below are the Bellman expectation equations for the state-value function $v_\pi(s)$ under a generic policy π .

$$v(T) = \pi(W|T)[20 + \gamma v(T)] + \pi(G|T)[0.5(-25 + \gamma v(T)) + 0.5(-10 + \gamma v(E))] + \pi(S|T)[0.2\gamma v(T) + 0.8\gamma v(E)]$$

$$v(E) = \pi(W|E)[0.8(50 + \gamma v(T)) + 0.2(20 + \gamma v(E))] + \pi(G|E)[-20 + \gamma v(H)]$$

$$v(H) = \pi(W|H)[100 + \gamma v(T)]$$

Substituting the policy values:

- i. $p = 0.65, \gamma = 0.8$

$$\begin{aligned} v(T) &= 0.65[20 + 0.8v(T)] + 0.175[0.5(-25 + 0.8v(T)) + 0.5(-10 + 0.8v(E))] \\ &\quad + 0.175[0.16v(T) + 0.64v(E)] \\ &= 9.9375 + 0.618v(T) + 0.182v(E) \end{aligned}$$

$$\begin{aligned} v(E) &= 0.65[0.8(50 + 0.8v(T)) + 0.2(20 + 0.8v(E))] + 0.35[-20 + 0.8v(H)] \\ &= 21.6 + 0.416v(T) + 0.104v(E) + 0.28v(H) \end{aligned}$$

$$v(H) = 100 + 0.8v(T)$$

ii. $p = 0.9, \gamma = 0.25$

$$\begin{aligned}
v(T) &= 0.9[20 + 0.25v(T)] + 0.05[0.5(-25 + 0.25v(T)) + 0.5(-10 + 0.25v(E))] \\
&\quad + 0.05[0.05v(T) + 0.2v(E)] \\
&= 17.125 + 0.23375v(T) + 0.01625v(E) \\
v(E) &= 0.9[0.8(50 + 0.25v(T)) + 0.2(20 + 0.25v(E))] + 0.1[-20 + 0.25v(H)] \\
&= 37.6 + 0.18v(T) + 0.045v(E) + 0.025v(H) \\
v(H) &= 100 + 0.25v(T)
\end{aligned}$$

(c) Value iteration:

$$v_{k+1}(s) = \max_a \sum_{s'} T(s, a, s') [R(s, a, s') + \gamma v_k(s')]$$

Iteration 1:

$$\begin{aligned}
v_1(T) &= \max \begin{pmatrix} 1 \cdot (20 + 1 \cdot 0) + 0 + 0 \\ 0.5 \cdot (-25 + 0) + 0.5 \cdot (-10 + 0) \\ 0.2 \cdot (0 + 0) + 0.8 \cdot (0 + 0) \end{pmatrix} \\
&= \max(20, -17.5, 0) = 20 \\
v_1(E) &= \max(44, -20) = 44 \\
v_1(H) &= \max(100) = 100
\end{aligned}$$

Iteration 2:

$$\begin{aligned}
v_2(T) &= \max \begin{pmatrix} 1 \cdot (20 + 1 \cdot 20) + 0 + 0 \\ 0.5 \cdot (-25 + 20) + 0.5 \cdot (-10 + 44) \\ 0.2 \cdot (0 + 20) + 0.8 \cdot (0 + 44) \end{pmatrix} \\
&= \max(40, 14.5, 39.2) = 40 \\
v_2(E) &= \max \begin{pmatrix} 0.2 \cdot (20 + 44) + 0.8 \cdot (50 + 20) \\ 1 \cdot (-20 + 100) \end{pmatrix} \\
&= \max(68.8, 80) = 80 \\
v_2(H) &= \max(1 \cdot (100 + 20)) \\
&= \max(120) = 120
\end{aligned}$$

Iteration 3:

$$\begin{aligned}
v_3(T) &= \max \begin{pmatrix} 1 \cdot (20 + 1 \cdot 40) + 0 + 0 \\ 0.5 \cdot (-25 + 40) + 0.5 \cdot (-10 + 80) \\ 0.2 \cdot (0 + 40) + 0.8 \cdot (0 + 80) \end{pmatrix} \\
&= \max(60, 42.5, 72) = 72 \\
v_3(E) &= \max \begin{pmatrix} 0.2 \cdot (20 + 80) + 0.8 \cdot (50 + 40) \\ 1 \cdot (-20 + 120) \end{pmatrix} \\
&= \max(92, 100) = 100 \\
v_3(H) &= \max(1 \cdot (100 + 40)) \\
&= \max(140) = 140
\end{aligned}$$

Iteration 4:

$$v_4(T) = \max \begin{pmatrix} 1 \cdot (20 + 1 \cdot 72) + 0 + 0 \\ 0.5 \cdot (-25 + 72) + 0.5 \cdot (-10 + 100) \\ 0.2 \cdot (0 + 72) + 0.8 \cdot (0 + 100) \end{pmatrix}$$

$$= \max(92, 68.5, 94.4) = 94.4$$

$$v_4(E) = \max \begin{pmatrix} 0.2 \cdot (20 + 100) + 0.8 \cdot (50 + 72) \\ 1 \cdot (-20 + 140) \end{pmatrix}$$

$$= \max(121.6, 120) = 121.6$$

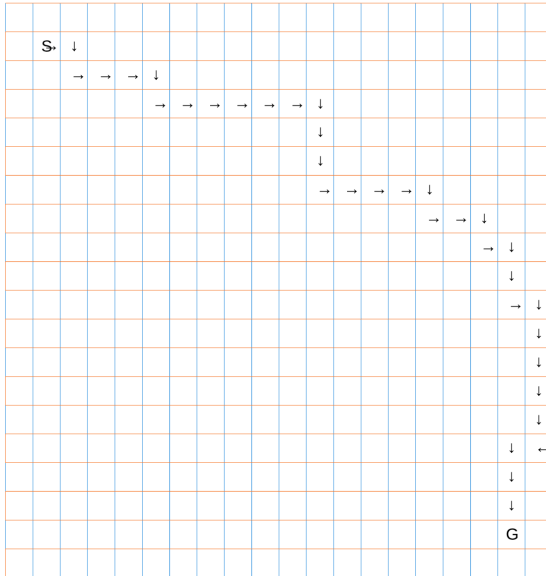
$$v_4(H) = \max (1 \cdot (100 + 72))$$

$$= \max(172) = 172$$

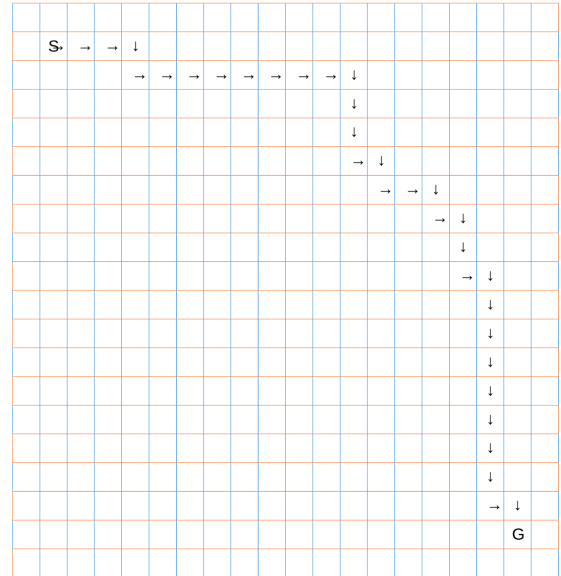
3. Exercise 3

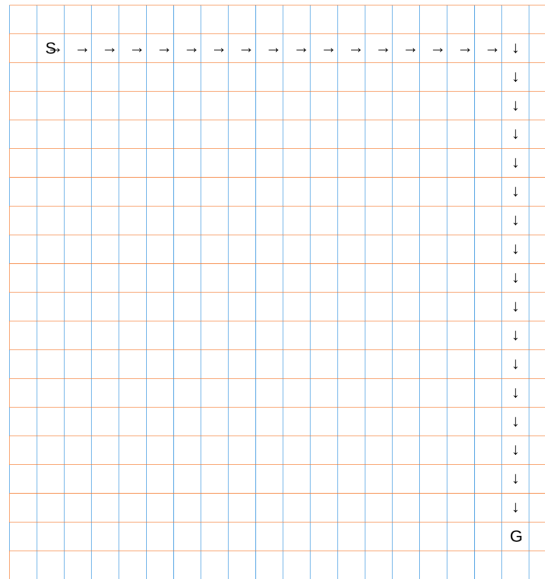
Analyzing the results of the three parameter sets for both SARSA and Q-learning, each run for 4000 episodes with every 500th episode shown starting from 1000, we notice, between sets i and iii, which differ only in their α values, that higher α values find a solution (reach the goal cell) faster: with SARSA, $\alpha = 0.9$ reaches G at episode 1500, while $\alpha = 0.3$ does so at episode 2000, and Q-learning with $\alpha = 0.9$ even converges to an optimal path by episode 1000, while with $\alpha = 0.3$ it reaches G consistently only from episode 3000. This is because new estimates have a greater impact on the stored Q-values when α is large. For SARSA, however, higher α values also make learning less stable: with $\alpha = 0.9$, it fails to reach G again at episodes 3000, 3500, and 4000. With Q-learning, we do not observe this instability: with $\alpha = 0.9$ (the third parameter set), it reaches G through the same optimal path in each of the 7 shown episodes, having converged to it by episode 1000. The reason for the different behavior is that SARSA updates its Q-values using the action actually taken in the episode, which was sampled from the ϵ -greedy policy, while Q-learning updates using the action that provides the maximum Q-value in the next state, which is calculated deterministically. Q-learning therefore avoids the randomness that makes SARSA unstable when α is large.

Simple grid-world SARSA - episode 1500 - (ϵ : 0.100), (α : 0.9000), (γ : 0.9)

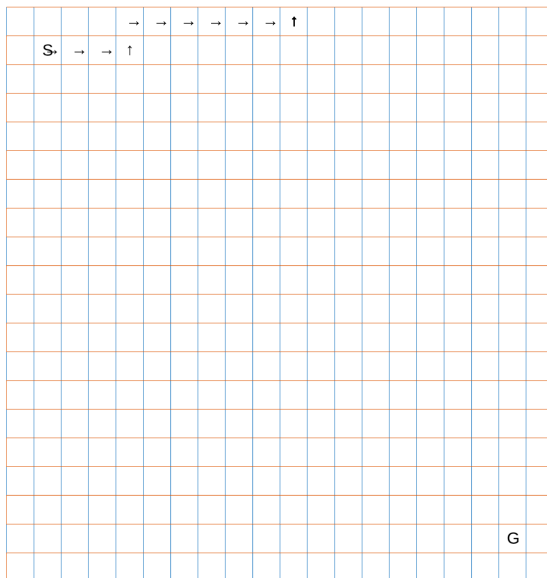
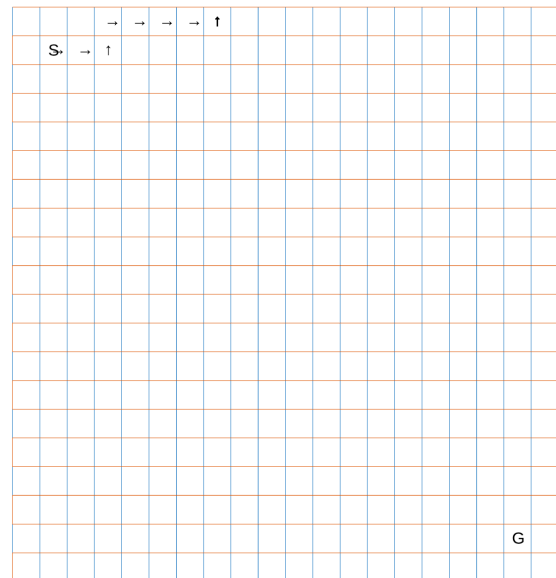


Simple grid-world SARSA - episode 2000 - (ϵ : 0.100), (α : 0.3000), (γ : 0.9)



Simple grid-world Q-Learning - episode 1000 - (ϵ : 0.100), (α = 0.9000), (γ = 0.9)

For parameter set ii, which differs from the other two in both its γ and ϵ values, we notice that neither SARSA nor Q-learning finds a solution that reaches G, even in 50000 episodes. This is because with $\epsilon = 0.9$ the agent chooses a random action (as opposed to the greedy one) with 90% probability, so the greedy path's states are visited hardly more often than any other state, making the learning very slow. In addition, $\epsilon = 0.9$ makes SARSA learn the values of an almost random policy. A discount of $\gamma = 0.5$ also means that, far from G, the goal's influence is easily outweighed by errors in the learned Q-values, leading a greedy policy to choose the wrong action.

Simple grid-world SARSA - episode 50000 - (ϵ : 0.900), (α = 0.3000), (γ = 0.5)Simple grid-world Q-Learning - episode 50000 - (ϵ : 0.900), (α = 0.3000), (γ = 0.5)

SARSA is an on-policy method, while Q-learning is off-policy. This means that, when following an ϵ -greedy policy, SARSA learns the values of the ϵ -greedy policy itself, whereas Q-learning learns the values of the greedy policy. Q-learning therefore converges to an optimal policy even with a fixed ϵ , as it did in sets i and iii, while SARSA's estimates keep changing because the ϵ -greedy policy with a fixed ϵ constantly introduces noise. We saw this with the third parameter set, where, due to the large $\alpha = 0.9$, the noise kept breaking the greedy solution.