Chandler Justice - A02313187

ST: Reinforcement Learning - Midterm Exam
Due: October 6, 2025 @ 08:30

---

1 Multiple-Choice

Explain why the correct answer is the correct answer since we gave you the answers

1. What is the difference between on-policy and off-policy algorithms?
   (a) On-policy algorithms use a single policy for both acting and learning, while off-policy algorithms use a target policy for acting and a behavioral policy for learning from rewards. (Correct)
   The correct policy approach depends on the application, but a primary advantage of an off-policy is the ability to decouple learning the optimal policy from the policy currently being acted on. In our first homework, we learned a policy from observing a random actor, but by observing this actor we were able to create a policy that when deployed as the actor was much more proficient at the task. Off-policy learning also has the advantage of not being as susceptible to converging into a local minima as the behavior can differ from the greedy policy target. In some applications it may make more sense to use an on-policy technique, such as training a robot to walk using real hardware.
   (b) On-policy algorithms use deterministic policies while off-policy algorithms use stochastic polices.
   (c) On-policy algorithms choose the best action without any exploration. Off-policy algorithms chooses the best action with exploration.
   (d) On-policy algorithms uses models to learn while off-policy algorithms don't.
2. What are the inputs and output of the State Value Function?
   (a) Inputs: Current state, Output: Expected total reward (Correct)
   The reason we use the current state is because in the State Value Function we assume the environment has the Markcov property which means the future only depends on the present, so we only need the current state to determine the value of the state we are in. We output the expected total reward as the value function is meant to provide insight on the long-horizon impact of the current state, providing context where the immediate reward value might be misleading.
   (b) Inputs: Next state, chosen action Output: Expected immediate reward
   (c) Inputs: Reward of current state Output: Proposed action
   (d) Inputs: Next State Output: Updated policy
3. In a MDP, what is the separation between agent and environment?
   (a) The environment is everything that the agent can not control. (Correct)
   This is important for correctly formulating the control mechanisms available to the agent. While the battery powering the computer the agent is running on might semantically be a part of the agent, making the battery a part of the environment is the correct decision as the agent cannot manipulate the current charge of the battery.
   (b) The environment chooses the actions and the agent records the rewards.
   (c) The agent can control the environment and the environment provides feedback.

(d) There is no real separation between environment and agent.

4. In DQN, what is the role of the replay buffer?

   (a) To break correlation by randomly sampling from the agent's past experiences stored in the buffer. (Correct)
   The replay buffer stores experiences (s, a, r, s') and allows random sampling during training. A key advantage of the replay buffer is it prevents the agent from falling into a local minima. If an agent falls into a certain traversal path through greedy traversal of states this can result in a sub-optimal policy. Using the replay buffer can diminish this effect as it adjusts the policy with regards to the experiences sampled. Additionally, this can result in faster convergence as the past experiences are reinforced multiple times over meaning less samples are needed overall.

   (b) The time it takes for an agent to analyze what the best action would be in a state

   (c) Used to store only high-reward actions

   (d) Computes the reward signal for each action

5. With a Q-Learning, how can we guarantee that we will converge on an optimal solution (assuming a proper decaying step-sized is used)?

   (a) As long as every state-action pair is continually updated then you will converge. (Correct)
   Analytical proofs for this claim are not provided in the textbook; however, there are many properties which Q-learning possess which imply convergence under specific conditions. Specifically

   - *Every state-action pair is visited infinitely often (in implementation this means enough for 'sufficient' exploration)*
   - *Q-Learning is an off-policy TD control method*
   - *Q-Learning is applied to an environment with finite states and actions. (Note: most proofs make this assumption, but some literature has shown that Q-learning can converge in continuous cases)*
   - *Rewards are bounded ($r_t \neq \infty$ as $t \to \infty$)*

   (b) Always choosing the action that yields the highest reward will guarantee convergence.

   (c) Setting $\gamma = 1$ will cause convergence.

   (d) The Q-values should always be increasing to cause convergence.

2 Short Response

1. Explain how the Action-Value Function works. Write the Bellman equation and explain each component.

   Solution: The bellman *expectation* equation states

$$\begin{aligned} q(s, a) &= \mathbb{E}_\pi \left[G_t | S_t = s, A_t = a\right] \\ &= \sum_{s', r} p(s', r|s, a) \left[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')\right] \end{aligned}$$

   Lets go through what each component of this equation is doing:

   - $p(s', r|s, a)$: This gives us the probability of getting a reward in a future state s' from our current state. When we take the sum of all of these values they will add up to 1, meaning we are exploring all possible future states
   - $[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')]$: This gives us the expected future value given all possible future actions and states, and our current policy.

The allows us to get a long-horizon outlook on the quality of a state action pair. When we are picking an action, there may be a choice which looks optimal from a greedy perspective, but which actually only leads to a locally optimal solution, meaning we converge to a sub-optimal policy. By evaluating the long-term quality of a state-action pair, we can pick an action which immediately gives a lower reward, but in the long term will result in a higher value overall.

2. How does the Q-Table relate to the Action-Value function in Question 1?
Solution: The Q-Table relates a given state and action and provides the expected cumulative discounted return given the pair. The entries in the Q-table are updated with

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

When an action and reward is visited, the entry in the Q-Table is updated to reflect the reward obtained, plus the estimated value of taking optimal future values. As time increases, the Q-table will converge towards the true action-value function. This provides a significant computational advantage over directly computing the action-value function.

3. How does DQN differ from Q-Learning? What are some improvements that DQN brings?
Solution: DQN utilizes a neural network instead of using a Q-table. This allows for utilizing the same principals of Q-learning without having to keep a tabulation of all possible key-value pairs, which in turn means this learning technique can be applied to much larger environments. Additionally, DQN provides a replay buffer which randomly samples the environment allowing for faster convergence to an optimal policy alongside breaking up breaking up correlations which can arise when following sequential state-action pairs.

Another key distinction is the target network which provides additional stability through less frequent updates versus the main Q-network. These less frequent updates stabilize training which can arise from the constant change in target values.

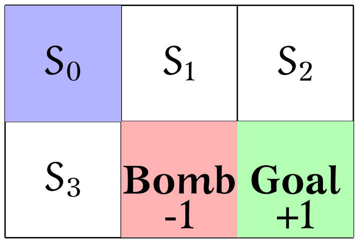
3 Q-Table by Hand

You get to generate by hand what the Q-Table representation of this gridworld is going to be! Recall that the formula for updating your Q-Table is:

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

Where $a \in A, A = \{\text{Up}, \text{Down}, \text{Right}, \text{Left}\}$ and $s \in S, S = \{S_0, S_1, S_2, S_3\}$ Assume $\alpha = 0.5$ and $\gamma = 1.0$

(For every action taken, assume 0 reward if not in a terminal condition.) Please compute 5 iterations of the Q-table as it learns.



We have provided for you what you should get on the second iteration:

State	**Up**	**Down**	**Left**	**Right**
S_0	0	0	0	0
S_1	0	-0.75	0	0.25
S_2	0	0.75	0.125	0
S_3	0	0	0	-0.75

Table 1: Iteration 2

Solution: This was painful to compute by hand.

Iteration 1

State	Up	Down	Left	Right
S0	0	0	0	0
S1	0	-0.5	0	0
S2	0	0.5	0	0
S3	0	0	0	-0.5

Iteration 2

State	Up	Down	Left	Right
S0	0	0	0	0
S1	0	-0.75	0	0.25
S2	0	0.75	0.125	0
S3	0	0	0	-0.75

Iteration 3

State	Up	Down	Left	Right
S0	0	0	0	0.125
S1	0	-0.875	0.0625	0.5
S2	0	0.875	0.3125	0
S3	0.0625	0	0	-0.875

Iteration 4

State	Up	Down	Left	Right
S0	0	0.03125	0	0.3125
S1	0	-0.9375	0.1875	0.6875
S2	0	0.9375	0.5	0
S3	0.1875	0	0	-0.9375

Iteration 5

State	Up	Down	Left	Right
S0	0	0.109375	0	0.5
S1	0	-0.96875	0.34375	0.8125
S2	0	0.96875	0.65625	0
S3	0.34375	0	0	-0.96875

4 Graduate Students: Probabilistic Q-Table by Hand

Please add two complexities to this problem:

- Teleportation on wall hit: whenever you hit a "wall" or edge of the gridworld, you have an equal chance to randomly teleport to any of the six cells
- Random motion: On moving, every action has a 20% chance of randomly picking a direction to move

You are welcome to either take the expectations/probabilities directly or emulate with a random number.

Compare the above with a situation where you have an action cost (for each action taken) of -0.25, and comment on the changes in the Q table?

Solution: Let's do this with a program.[1] We will use `python` to make life easy. I wrote the following implementation of the above requirements:

```
import random

ALPHA = 0.5

A={'S0':['Up', 'Down', 'Left', 'Right'],
   'S1':['Up', 'Down', 'Left', 'Right'],
   'S2':['Up', 'Down', 'Left', 'Right'],
   'S3':['Up', 'Down', 'Left', 'Right'],}

Q={s:{a:0.0 for a in A[s]} for s in ['S0','S1','S2','S3']}

rewards = {
```

[1] I am not very proud of this program. I think it is written poorly. If I was using this for longer than an exam I would rewrite it.

```
           'S0':    0,
           'S1':    0,
           'S2':    0,
           'S3':    0,
           'Bomb': -1,
           'Goal':  1,
          }

inbound_transitions = {
    ('S0', 'Right'): ['S1', 0],
    ('S0', 'Down'):  ['S3', 0],
    ('S1', 'Down'):  ['Bomb', -1],
    ('S1', 'Right'): ['S2', 0],
    ('S1', 'Left'):  ['S0', 0],
    ('S2', 'Down'):  ['Goal', 1],
    ('S2', 'Left'):  ['S1', 0],
    ('S3', 'Right'): ['Bomb', -1],
    ('S3', 'Up'):    ['S0', 0]
}

def wall_hit(s, teleport=False):
    if teleport:
        new_cell = random.choice(list(rewards.keys()))
        return new_cell, rewards[new_cell]
    else:
        return s, 0


def step(state, action, teleport=False):
    if (state, action) in inbound_transitions:
        sprime, r = inbound_transitions[(state, action)]
        return sprime, r

    return wall_hit(state, teleport)


def maxQ(s):
    if s in ('Bomb','Goal'):
        return 0.0
    return max(Q[s].values())

def update(s, a, teleport=False, random_action=False, random_p=0.2):
    if random_action:
        coin = 1 if random.random() < random_p else 0
        if coin:
            a = random.choice(['Up', 'Down', 'Left', 'Right'])

    sprime, r = step(s,a, teleport)
    if sprime == s and not teleport:
        return # we are trying to transition into a wall
    target = r + (0 if sprime in ('Bomb','Goal') else maxQ(sprime))
    Q[s][a] = Q[s][a] + ALPHA * (target - Q[s][a])

def print_Q(Q):
    for state, values in Q.items():
        print(state, end='\t')
        for value in values.values():
            print(f'{value:8.5f}', end=' ')
        print()

if __name__ == "__main__":
    random.seed(42)
    ITERATIONS = 5
    TELEPORT = True
    RANDOM_MOTION = True
    for iter in range(ITERATIONS):
        for s in A.keys():
            for a in A[s]:
                update(s,a, TELEPORT, RANDOM_MOTION)
        print(f"Q-table at iteration {iter}")
        print_Q(Q)
```

This program lets us toggle our new features (so I can verify the previous part is correct), and it lets us adjust the number of iterations. Let's try this for 10 iterations and see if we converge to an optimal policy:

```
Q-table at iteration 0
S0  0.00000  0.00000  0.00000  0.00000
S1 -0.25000  0.00000  0.00000  0.00000
S2 -0.50000  0.00000  0.00000  0.00000
S3  0.00000  0.00000  0.00000 -0.75000
Q-table at iteration 1
S0  0.00000  0.00000 -0.50000  0.00000
S1 -0.25000 -0.50000  0.00000  0.00000
S2  0.25000  0.50000  0.00000  0.50000
S3  0.00000  0.12500  0.06250 -0.87500
Q-table at iteration 2
S0  0.25000  0.00000  0.37500  0.00000
S1 -0.62500 -0.75000  0.18750  0.25000
S2  0.37500  0.75000  0.12500  0.75000
S3  0.18750  0.25000  0.21875 -0.93750
... Middle skipped for brevity ...
Q-table at iteration 8
S0  0.71008  0.73145  0.65210  0.77441
S1  0.19531 -0.99902  0.72314  0.98486
S2  0.81934  0.99609  0.93433  0.77295
S3  0.74097  0.30127  0.95044 -0.99951
Q-table at iteration 9
S0  0.89035  0.84094  0.65210  0.87964
S1  0.57288 -0.99951  0.80675  0.99048
S2  0.90967  0.99805  0.96240  0.83165
S3  0.81566  0.65063  0.92039 -0.99976
```

I'm not sure if this counts as *long-run*, but at 100 iterations (which for this small of a problem I would argue is *long-run*) the Q-table looks like:

```
Q-table at iteration 99
S0  0.74307  1.00000  0.99988  1.00000
S1 -0.06836 -1.00000  1.00000  1.00000
S2  0.98822  1.00000  1.00000  0.68744
```

```
S3  1.00000  0.99984  0.99902 -1.00000
```

Which means we converge to a policy of "move basically anywhere except towards the bomb".

We can adjust our `rewards` and `inbound_transitions` variables to induce a penalty of -0.25 for all steps (including terminal steps, meaning goal=0.75 and bomb=-1.25) to complete the second part of the question. Let's look at the same iteration points of this version as the previous simulation:

```
Q-table at iteration 0
S0 -0.12500 -0.12500 -0.12500 -0.12500
S1 -0.43750  0.00000  0.00000 -0.18750
S2 -0.62500  0.00000 -0.12500 -0.25000
S3 -0.18750  0.00000 -0.12500 -0.93750
Q-table at iteration 1
S0 -0.25000 -0.12500 -0.68750 -0.18750
S1 -0.43750 -0.62500 -0.18750 -0.23438
S2  0.06250  0.37500 -0.28125  0.25000
S3 -0.18750 -0.18750 -0.25000 -1.09375
Q-table at iteration 2
S0 -0.06250 -0.12500  0.09375 -0.31250
S1 -0.84375 -0.93750 -0.17188 -0.05469
S2  0.09375  0.56250 -0.29297  0.50000
S3 -0.17188 -0.17188 -0.20312 -1.17188
... Middle skipped for brevity ...
Q-table at iteration 8
S0  0.14789 -0.04608  0.06905  0.08307
S1 -0.22656 -1.24878 -0.09531  0.48868
S2  0.21750  0.74707  0.20134  0.03784
S3 -0.17374 -0.33035  0.29564 -1.24939
Q-table at iteration 9
S0  0.07120 -0.00022  0.06905  0.16087
S1 -0.09046 -1.24939 -0.09222  0.49287
S2  0.48375  0.74854  0.22211 -0.02564
S3 -0.13144  0.20982  0.10326 -1.24969
... Middle skipped for brevity ...
Q-table at iteration 99
S0  0.20468  0.13250  0.72686  0.25000
S1 -0.52628 -1.25000  0.45056  0.50000
S2  0.53019  0.75000  0.25000  0.26930
S3  0.40372  0.51069  0.49169 -1.25000
```

The action penalty is clearly described by this new Q-table. We can see that movements which are closer to the goal are closer to the maximum reward of 0.75, while movements which would create more future steps are further from the goal. Additionally, the new lower bound for reward -1.25 can be seen in some state-action pairs.

FIN,