Report Assignment 2

Thi Tuyet Nhi Nguyen

Faculty of Applied Science, Memorial University of Newfoundland

The code is available at: Reinforcement_Learning/Assignment 2 at main · snoween/Reinforcement_Learning

I. Part 1

To do this assignment, first need to import necessary libraries as below:

```
import numpy as np
```

Also, we need to set up gridworld for this Part as shown below:

```
# Gridworld setup
grid_size = 5
n_states = grid_size * grid_size
actions = ['up', 'down', 'left', 'right']
gamma = 0.95

# Special squares (by index)
blue = 1 * grid_size + 1      # (1,1)
green = 3 * grid_size + 1     # (3,1)
red = 1 * grid_size + 3       # (1,3)
yellow = 3 * grid_size + 3    # (3,3)
```

A function to perform result as a matrix:

```
# Map (row, col) to state index
def state_index(row, col):
    return row * grid_size + col
```

A function to perform movement transitions:

```
# Movement transitions
def next_state(state, action):
    row, col = divmod(state, grid_size)
    if action == 'up':
        row2 = max(row - 1, 0)
        col2 = col
    elif action == 'down':
        row2 = min(row + 1, grid_size - 1)
        col2 = col
    elif action == 'left':
        row2 = row
        col2 = max(col - 1, 0)
    elif action == 'right':
        row2 = row
        col2 = min(col + 1, grid_size - 1)
    return state_index(row2, col2)
```

A function to return a reward

```python
# Reward and next_state logic
def transition(s, a):
    if s == blue:
        return red, 5
    if s == green:
        return np.random.choice([red, yellow]), 2.5
    s2 = next_state(s, a)
    if s == s2:  # Hitting boundary
        return s, -0.5
    return s2, 0
```

1. Question 1:

(1) Solving the system of Bellman equations explicitly:

The code for solving the system of Bellman equations explicitly:

```python
# Solving system Bellman
def solve_bellman_system():
    A = np.zeros((n_states, n_states))
    b = np.zeros(n_states)
    for s in range(n_states):
        for a in actions:
            s2, r = transition(s, a)
            A[s, s2] += 0.25 * gamma
            b[s] += 0.25 * r
        A[s, s] -= 1  # Move all terms to one side: A * V = b
    V = np.linalg.solve(A, -b)
    return V.reshape((grid_size, grid_size))

# Show the results
V_exact = solve_bellman_system()
print("\nValue Function (Exact Solution):\n", V_exact.round(2))
```

The results:

```
Value Function (Exact Solution):
 [[ 2.28  2.94  1.74  0.66 -0.11]
 [ 3.15  5.95  2.53  1.    0.15]
 [ 2.42  3.22  1.94  0.87  0.13]
 [ 1.93  3.26  1.57  0.59 -0.08]
 [ 1.04  1.43  0.81  0.13 -0.45]]
```

(2) Iterative policy evaluation:

The code for iterative policy evaluation:

```python
# Function of Iterative policy evaluation
def iterative_policy_evaluation(threshold=1e-2):
    V = np.zeros(n_states)
    policy_prob = 0.25
    iteration = 0
    while True:
        delta = 0
        new_V = np.zeros(n_states)
        for s in range(n_states):
            v = 0
            for a in actions:
                s2, r = transition(s, a)
                v += policy_prob * (r + gamma * V[s2])
            new_V[s] = v
            delta = max(delta, abs(v - V[s]))
        V = new_V
        iteration += 1
        if delta < threshold:
            break
    print(f"Converged after {iteration} iterations.")
    return V.reshape((grid_size, grid_size))
```

```
V_iter = iterative_policy_evaluation()
print("Value Function (Iterative):\n", V_iter.round(2))
```

The result is:

```
Converged after 179 iterations.
Value Function (Iterative):
 [[ 2.28  2.94  1.74  0.66 -0.11]
 [ 3.16  5.95  2.53  1.    0.15]
 [ 2.43  3.22  1.94  0.87  0.13]
 [ 1.93  3.26  1.57  0.59 -0.08]
 [ 1.05  1.43  0.81  0.13 -0.45]]
```

Discussion: Which states have the highest value? Does this surprise you?

The states that have a higher chance of reaching a positive reward sooner (or avoid negative reward) have higher values. Typically, in grid-worlds with terminal states, states closer to terminal states (goal) have higher values.

This situation does not surprise me. Because it aligns with intuition: states with better future expected rewards have higher value. Random policy leads to less difference in values compared to an optimal policy.

2. Question 2:

Determine the optimal policy for the gridworld problem by

(1) explicitly solving the Bellman optimality equation:

The code:

```
def value_iteration(grid_size, gamma, theta=1e-6):
    V = np.zeros((grid_size, grid_size))
    actions = ['up', 'down', 'left', 'right']
    policy = np.zeros((grid_size, grid_size), dtype=int)
    while True:
        delta = 0
        for i in range(grid_size):
            for j in range(grid_size):
                v = V[i, j]
                values = []
                for idx, action in enumerate(actions):
                    # Replace take_action with transition
                    s = i * grid_size + j
                    s2, r = transition(s, action)
                    row2, col2 = divmod(s2, grid_size)
                    values.append(r + gamma * V[row2, col2])
                V[i, j] = max(values)
                policy[i, j] = np.argmax(values)
                delta = max(delta, abs(v - V[i, j]))
        if delta < theta:
            break
    return V, policy
```

The result:

```
V_opt, policy_opt = value_iteration(5, 0.95)
print(np.round(V_opt, 3))
print(policy_opt)
```

```
[[31.639 33.304 31.639 30.057 28.554]
 [33.304 35.057 33.304 31.639 30.057]
 [31.639 33.304 31.639 30.057 28.554]
 [30.929 32.557 30.929 29.383 27.914]
 [29.383 30.929 29.383 27.914 26.518]]
[[1 1 1 1 1]
 [3 0 2 2 2]
 [0 0 0 0 0]
 [3 1 2 2 2]
 [0 0 0 0 0]]
```

(2) using policy iteration with iterative policy evaluation

The code:

```python
def policy_iteration(threshold=1e-2):
    V = np.zeros(n_states)
    policy = np.full((n_states,), 'up', dtype=object)  # arbitrary init
    stable = False
    iter_count = 0

    while not stable:
        # POLICY EVALUATION
        while True:
            delta = 0
            new_V = np.copy(V)
            for s in range(n_states):
                a = policy[s]
                s2, r = transition(s, a)
                new_V[s] = r + gamma * V[s2]
                delta = max(delta, abs(new_V[s] - V[s]))
            V = new_V
            if delta < threshold:
                break

        # POLICY IMPROVEMENT
        stable = True
        for s in range(n_states):
            old_a = policy[s]
            values = []
            for a in actions:
                s2, r = transition(s, a)
                values.append(r + gamma * V[s2])
            best_a = actions[np.argmax(values)]
            policy[s] = best_a
            if best_a != old_a:
                stable = False
        iter_count += 1

    print(f"Policy Iteration converged after {iter_count} improvement steps.")
    return V.reshape((grid_size, grid_size)), policy.reshape((grid_size, grid_size))
```

The result:

```python
V_pi, policy_pi = policy_iteration()

print("Optimal Value Function (Policy Iteration):\n", V_pi.round(2))
print("Optimal Policy (Policy Iteration):\n", policy_pi)
```

(3) policy improvement with value iteration.

The code:

```python
def value_iteration(threshold=1e-4):
    V = np.zeros(n_states)
    iter_count = 0
    while True:
        delta = 0
        new_V = np.copy(V)
        for s in range(n_states):
            values = []
            for a in actions:
                s2, r = transition(s, a)
                values.append(r + gamma * V[s2])
            new_V[s] = max(values)
            delta = max(delta, abs(new_V[s] - V[s]))
        V = new_V
        iter_count += 1
        if delta < threshold:
            break

    # Derive policy
    policy = np.full((n_states,), 'up', dtype=object)
    for s in range(n_states):
        values = []
        for a in actions:
            s2, r = transition(s, a)
            values.append(r + gamma * V[s2])
        policy[s] = actions[np.argmax(values)]

    print(f"Value Iteration converged after {iter_count} iterations.")
    return V.reshape((grid_size, grid_size)), policy.reshape((grid_size, grid_size))
```

The result:

```python
V_vi, policy_vi = value_iteration()

print("\nOptimal Value Function (Value Iteration):\n", V_vi.round(2))
print("Optimal Policy (Value Iteration):\n", policy_vi)
```

```
Value Iteration converged after 213 iterations.

Optimal Value Function (Value Iteration):
 [[31.64 33.3  31.64 30.06 28.55]
 [33.3  35.06 33.3  31.64 30.06]
 [31.64 33.3  31.64 30.06 28.55]
 [30.93 32.56 30.93 29.38 27.91]
 [29.38 30.93 29.38 27.91 26.52]]
Optimal Policy (Value Iteration):
 [['down' 'down' 'down' 'down' 'down']
 ['right' 'up' 'left' 'left' 'left']
 ['up' 'up' 'up' 'up' 'up']
 ['right' 'up' 'left' 'left' 'left']
 ['up' 'up' 'up' 'up' 'up']]
```

II. Part 2

The grid world in this part has changed some:

```python
# BLACK = terminal states, put it at top left and bottom right
black_states = [0, 24]

# Overwrite transition() to include terminal logic and new reward
def transition_with_terminal(s, a):
    if s in black_states:
        return s, 0  # Terminal state
    if s == blue:
        return red, 5
    if s == green:
        return np.random.choice([red, yellow]), 2.5
    s2 = next_state(s, a)
    if s == s2:
        return s, -0.5  # hitting wall
    return s2, -0.2
```

The code to show format in this part:

```python
# Show the result in a matrix form
def format_policy(policy_dict):
    table = np.full((grid_size, grid_size), '', dtype=object)
    for s in range(n_states):
        r, c = divmod(s, grid_size)
        if isinstance(policy_dict[s], str):
            table[r, c] = policy_dict[s]
        else:
            best_a = max(policy_dict[s], key=policy_dict[s].get)
            table[r, c] = best_a
    return table
```

1. Question 1: Use the Monte Carlo method with (1) exploring starts

The code:

```python
def mc_control_exploring_starts(num_episodes=1000, gamma=0.95):
    Q = { (s, a): 0.0 for s in range(n_states) for a in actions }
    returns = { (s, a): [] for s in range(n_states) for a in actions }
    policy = { s: np.random.choice(actions) for s in range(n_states) }

    for ep in range(num_episodes):
        # Exploring Start: random (s,a)
        s = np.random.choice([s for s in range(n_states) if s not in black_states])
        a = np.random.choice(actions)

        episode = []
        while True:
            s2, r = transition_with_terminal(s, a)
            episode.append((s, a, r))
            if s2 in black_states:
                break
            s = s2
            a = policy[s]

        G = 0
        visited = set()
        for t in reversed(range(len(episode))):
            s, a, r = episode[t]
            G = gamma * G + r
            if (s, a) not in visited:
                returns[(s, a)].append(G)
                Q[(s, a)] = np.mean(returns[(s, a)])
                visited.add((s, a))
                # Improve policy greedily
                best_a = max(actions, key=lambda a_: Q[(s, a_)])
                policy[s] = best_a

    return Q, policy
```

The result:

```python
# Exploring Starts
Q_es, policy_es = mc_control_exploring_starts()
policy_grid_es = format_policy(policy_es)
print("Policy (Exploring Starts):\n", policy_grid_es)
```

(2) without exploring starts

The code:

```
def mc_control_epsilon_soft(num_episodes=10000, gamma=0.95, epsilon=0.1):
    Q = { (s, a): 0.0 for s in range(n_states) for a in actions }
    returns = { (s, a): [] for s in range(n_states) for a in actions }
    policy = { s: { a: 1/len(actions) for a in actions } for s in range(n_states) }

    for ep in range(num_episodes):
        # Start from random non-terminal state
        s = np.random.choice([s for s in range(n_states) if s not in black_states])
        episode = []

        while True:
            a = np.random.choice(actions, p=[policy[s][a_] for a_ in actions])
            s2, r = transition_with_terminal(s, a)
            episode.append((s, a, r))
            if s2 in black_states:
                break
            s = s2

        G = 0
        visited = set()
        for t in reversed(range(len(episode))):
            s, a, r = episode[t]
            G = gamma * G + r
            if (s, a) not in visited:
                returns[(s, a)].append(G)
                Q[(s, a)] = np.mean(returns[(s, a)])
                visited.add((s, a))
                # Improve policy using epsilon-greedy
                best_a = max(actions, key=lambda a_: Q[(s, a_)])
                for a_ in actions:
                    if a_ == best_a:
                        policy[s][a_] = 1 - epsilon + (epsilon / len(actions))
                    else:
                        policy[s][a_] = epsilon / len(actions)

    return Q, policy
```

The result:

```
# Epsilon-Soft
Q_eps, policy_eps = mc_control_epsilon_soft()
policy_grid_eps = format_policy(policy_eps)
print("\nPolicy (Epsilon-Soft):\n", policy_grid_eps)
```

2. Question 2:

```
def mc_offpolicy_importance_sampling(num_episodes=10000, gamma=0.95):
    Q = { (s, a): 0.0 for s in range(n_states) for a in actions }
    C = { (s, a): 0.0 for s in range(n_states) for a in actions }
    target_policy = { s: np.random.choice(actions) for s in range(n_states) }

    for ep in range(num_episodes):
        # Generate episode using behaviour policy (random)
        s = np.random.choice([s for s in range(n_states) if s not in black_states])
        episode = []

        while True:
            a = np.random.choice(actions)  # behaviour = random
            s2, r = transition_with_terminal(s, a)
            episode.append((s, a, r))
            if s2 in black_states:
                break
            s = s2

        G = 0
        W = 1.0
        for t in reversed(range(len(episode))):
            s, a, r = episode[t]
            G = gamma * G + r
            C[(s, a)] += W
            Q[(s, a)] += (W / C[(s, a)]) * (G - Q[(s, a)])

            # Improve target policy
            best_a = max(actions, key=lambda a_: Q[(s, a_)])
            target_policy[s] = best_a

            if a != target_policy[s]:
                break  # importance weight becomes 0 from here on
            W = W * (1.0 / 0.25)  # (a|s)=1, b(a|s)=0.25

    return Q, target_policy
```

The result:

```
Q_off, policy_off = mc_offpolicy_importance_sampling()
policy_grid_off = format_policy(policy_off)

print("Optimal Policy (Off-policy MC with Importance Sampling):\n", policy_grid_off)
```