

强化学习作业一

题目 1:

a)

$$\hat{V}(s) = V_i^*(s) + c$$

最优策略不会改变，因为最优策略与值函数的相对大小有关和它依赖于值函数，而添加一个常数不会改变相对大小，等同于新值函数与旧值函数只有一个常数偏移，不会影响最优策略的选择。

b)

$$\hat{V}(s) = V_i^*(s) \times c$$

1) 新的最优策略仍然为 π^* 当且仅当 $c = 0$ 。在这种情况下，没有发生变化。

2) 如果 $c \neq 0$ ，最优策略可能会发生变化。例如，考虑一个 MDP，原始奖励函数 $r(s, a)$ 为正数，导致最优策略是选择最大奖励的动作。但是，如果乘以一个正数 $c > 1$ ，新的奖励函数 $r^*(s, a)$ 将更加强调奖励，智能体可能会更倾向于选择最大奖励的动作，从而导致最优策略的改变。

3) 不存在一个特定的 c ，使得该 MDP 中的所有策略都是最优策略。因为不同策略可能对应不同的最优值函数，对于不同的 MDP，最优策略可能会改变。

c) 如果该 MDP 存在终止状态，问题 a) 的结果可能会改变。如果存在一个或多个终止状态，那么最终状态的值将在每个策略下都是固定的，因为它们不会继续转移。在这种情况下，添加常数 c 到奖励函数可能会影响最优策略，因为 c 会对终止状态的值产生影响，而最优策略可能会根据值函数的变化而改变。这取决于 c 的大小以及终止状态的重要性。如果 c 足够小，终止状态的值不会受到显著影响，那么最优策略可能不会改变。

题目 2:

使用 REINFORCE with baseline 可以将 G_t 的计算复杂度降为 $O(1)$ ，在 REINFORCE with baseline 中，将更新的项变成了

$$\theta \leftarrow \theta + \alpha \gamma^t (G_t - b(s)) \nabla_{\theta} \log \pi(A_t | S_t, \theta)$$

其中， G_t 是 Return，通常是一个序列的奖励折现总和。引入 baseline $b(S_t)$ 之后，减去了与动作 A_t 无关的项 $b(S_t)$ ，这降低了更新项的方差，从而减少了计算复杂度。将 REINFORCE 中更新时使用的 G_t 改为，从而降低计算复杂度的

$$R_{t+1} + \gamma \hat{V}(S_{t+1}) - b(S_t)$$

题目 3:

定性分析:

1.Deterministic-4x4-FrozenLake-v0 环境: 这是一个确定性环境，其中动作执行后的结果是确定的。在这种环境中，算法可能会更快地收敛，因为它们可以根据已知的转移概率准确地计算值函数和策略。

2.Stochastic-4x4-FrozenLake-v0 环境: 这是一个随机性环境，其中动作执行后的结果是随机的。在这种环境中，算法可能需要更多的迭代才能收敛，因为随机性导致值函数和策略的估计更具不确定性。由于随机性，算法需要更多的样本来确定最优策略。

定量分析:

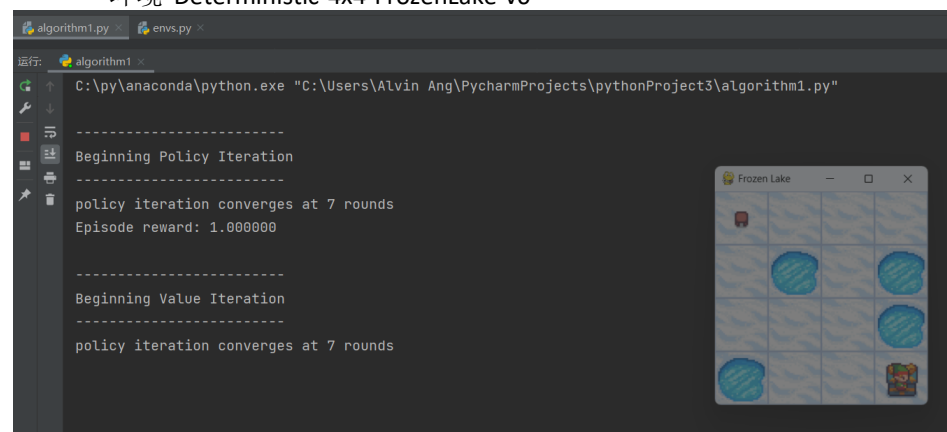
1.Deterministic-4x4-FrozenLake-v0 环境: 在这个环境中，由于确定性特质，算法可能需要较少的迭代次数才能找到最优策略。通常，值迭代和策略迭代会在较少的迭代次数内收敛。

2.Stochastic-4x4-FrozenLake-v0 环境: 在这个环境中，由于随机性，算法可能需要更多的迭代次数才能找到最优策略。随机性会导致值函数和策略的估计变得更不稳定，因此算法需要更多的迭代来平均随机性的影响。

最终，可以通过比较两个环境中算法的迭代次数和最终策略的性能来定量分析随机性的影响。在 **Stochastic-4x4-FrozenLake-v0** 环境中，算法需要更多的迭代次数才能获得与确定性环境相同的性能。

运行结果:

环境"Deterministic-4x4-FrozenLake-v0"



```
algorithm1.py  envs.py
运行: algorithm1.py
C:\py\anaconda\python.exe "C:\Users\Alvin Ang\PycharmProjects\pythonProject3\algorithm1.py"
-----
Beginning Policy Iteration
-----
policy iteration converges at 7 rounds
Episode reward: 1.000000
-----
Beginning Value Iteration
-----
policy iteration converges at 7 rounds
```

环境-"Stochastic-4x4-FrozenLake-v0"

```
C:\py\anaconda\python.exe "C:\Users\Alvin Ang\PycharmProjects\pythonProject3\algorithm1.py"
-----
Beginning Policy Iteration
-----
policy iteration converges at 6 rounds
Episode reward: 1.000000
-----
Beginning Value Iteration
-----
policy iteration converges at 23 rounds
Episode reward: 0.000000

进程已结束，退出代码为 0
|
```

实现代码

PolicyEvaluation

```
# 策略评估-固定策略 => 求最优状态值函数
def policy_evaluation(P, nS, nA, policy, gamma=0.9, tol=1e-3):
    # 初始化V
    value_function = np.zeros(nS)
    # 循环直到收敛
    while True:
        delta = 0
        # 对于每一个状态s进行循环
        for s in range(nS):
            v = value_function[s]
            new_v = 0
            a = policy[s]
            transitions = P[s][a]
            for prob, next_state, reward, terminal in transitions:
                # 使用Bellman方程更新新值
                new_v += prob * (reward + gamma * value_function[next_state])
            # 更新状态s的值函数
            value_function[s] = new_v
            # 计算最大状态值函数的变化，以检查是否收敛
            delta = max(delta, abs(v - new_v))
        # 如果变化小于容忍度 tol，则停止迭代
        if delta < tol:
            break
    return value_function
```

PolicyImprovement

```
# 策略改进-固定状态值函数 => 求最优策略
def policy_improvement(P, nS, nA, value_from_policy, policy, gamma=0.9):
    # 创建一个新的策略数组
    new_policy = np.zeros(nS, dtype="int")
    # 对每个状态s进行循环
    for s in range(nS):
        # 创建一个数组来存储每个动作的估计值
        q_values = np.zeros(nA)
        # 对每个动作a进行循环
        for a in range(nA):
            # 根据状态 s 和动作 a 查找状态转移概率
            for prob, next_state, reward, terminal in P[s][a]:
                # 使用贝尔曼方程计算状态-动作值函数 Q(s, a)
                q_values[a] += prob * (reward + gamma * value_from_policy[next_state])
            # 选择具有最大估计值的动作作为新策略
            new_policy[s] = np.argmax(q_values)
    return new_policy
```

PolicyIteration

```
# 策略迭代
def policy_iteration(P, nS, nA, gamma=0.9, tol=1e-3):
    # 创建策略
    value_function = np.zeros(nS)
    policy = np.zeros(nS, dtype=int)
    # 迭代步数
    iters = 0
    while True:
        # 更新步数
        iters += 1
        # 策略评估
        value_function = policy_evaluation(P, nS, nA, policy, gamma, tol)
        # 策略改进
        new_policy = policy_improvement(P, nS, nA, value_function, policy, gamma)
        # 如果策略相同，也就是收敛到一个最优策略
        if np.array_equal(policy, new_policy):
            print(f"policy iteration converges at {iters} rounds")
            return value_function, policy
        # 更新当前策略，继续迭代
        policy = new_policy
```

Value Iteration

```
def value_iteration(P, nS, nA, gamma=0.9, tol=1e-3, max_iter=1000):
    # 初始化状态值函数为0
    value_function = np.zeros(nS)
    # 初始化策略为0
    policy = np.zeros(nS, dtype=int)
    # 迭代步数
    iters = 0
    # 迭代多次, 最多max_iter次
    for i in range(max_iter):
        delta = 0
        iters += 1
        # 对每个状态 s 进行循环
        for s in range(nS):
            v = value_function[s]
            q_values = np.zeros(nA)
            # 对每个动作 a 进行循环
            for a in range(nA):
                q_val = 0
                transitions = P[s][a]
                # 对每个可能的状态转移进行循环
                for prob, next_state, reward, terminal in transitions:
                    # 使用 Bellman 方程计算动作值
                    q_val += prob * (reward + gamma * value_function[next_state])
                q_values[a] = q_val
            # 选择具有最大动作值的新状态值
            new_v = np.max(q_values)
            # 更新状态 s 的值
            value_function[s] = new_v
            # 计算状态值的变化
            delta = max(delta, np.abs(v - new_v))
        # 如果状态值的变化小于容忍度 tol, 则终止迭代
        if delta < tol:
            break
```