《强化学习》专业选修课

作业一

提交时间：10 月 15 日之前

提交形式：请提交一个压缩包，命名为：学号_姓名_assignment1.zip。内容包括：

- 题目 1 和 2 的解答 assignment1_solution.pdf
- 题目 3 中的 algorithm1.py
- Email 到：bit_reinforce@163.com，邮件标题为：学号_姓名_assignment1

题目 1：奖励函数的选择

已知一个离散动作状态空间 MDP，折扣因子$\gamma \in (0,1)$。该 MDP 没有终止状态，智能体将不断做出决策。假设该问题的原始最优值函数为V_1^*，最优策略为π_1^*。

a) 若对该 MDP 的每一次状态转移上都加上一个小的正实数（即：$r(s,a)$是原始奖励函数，新的奖励函数为$\hat{r}(s,a) = r(s,a) + c;\ \forall s,a$），请写出新的最优值函数表达式。在新条件下，最优策略是否会改变？请说明理由。

b) 若对该 MDP 的每一次状态转移都乘上实数$c \in R$（即$\hat{r}(s,a) = c\ \times\ r(s,a);\ \forall s,a$）。

   1) 是否存在特定情况使得新的最优策略仍然为π_1^*，且值函数能用c和V_1^*来表示？如果存在，请写出对应的值函数表达式、满足该情况的常数c。如果不存在，说明理由。

   2) 是否存在特定情况使得最优策略发生变化？如果存在，请举出一个c的例子，并说明变化理由；如果不存在，也请说明理由。

   3) 是否存在一个c，使得该 MDP 中的所有策略都是最优策略？举出该例子。

c) 如果该 MDP 存在终止状态，这是否会改变问题 a)的结果？如果是，请给出一个例子并说明理由。

题目 2：REINFORCE 算法优化

为了计算 REINFORCE 的目标估计，需要计算$(G_t)_{t=1}^{T}$的值，其中G_t的表达式是：

$$G_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$$

注意到，计算所有这些值需要$O(T^2)$时间。请给出一种在$O(T)$时间内计算的方法。

题目 3：封冻湖小游戏

在这道题中，需要使用 Gymnasium（前身为 OpenAI Gym）环境实现值迭代和策略迭代。请参照本文件中的示例代码实现。

a) 环境配置

- 使用 Python 3.8 或更高版本完成。
- 完成任务所需依赖包在 `requirements.txt` 中，可以通过 `pip install -r requirements.txt` 安装。
- 建议为创建专用虚拟环境（virtualenv 或 Anaconda），以避免软件依赖包冲突。
- 编程时，若有任何 `DeprecationWarning` 出现，请忽略。

b) 题目要求

- 编程题 1：阅读示例代码 `algorithm1.py`，并实现 `policy_evaluation`，`policy_improvement` 和 `policy_iteration` 三个函数，迭代停止条件tol $= max_s|V_{old}(s) - V_{new}(s)| < 10^{-3}$，折扣因子$\gamma = 0.9$。返回最优价值函数和最优策略。
- 编程题 2：实现 `algorithm1.py` 中的 `value_evaluation`，迭代停止条件和折扣因子与编程题 1 相同。返回最优价值函数和最优策略。
- （问题）分别在附件代码中 Deterministic-4x4-FrozenLake-v0 和 Stochastic-4x4-FrozenLake-v0 中运行上面两小问的方法。在第二个环境中，环境互动是随机的。请从定性和定量两个角度指出随机性是如何影响迭代的次数，以及最终策略。

提示：对于 Deterministic-4x4-FrozenLake-v0 环境，前三个状态的值是 0.59，0.656，0.729。作为验证，实现值迭代和策略迭代函数产生的值与它们的误差应该不超过10^{-3}。

c) 附件

requirements.txt

```
gymnasium[toy_text]==0.27.0
matplotlib
numpy
scipy
```

algorithm1.py

```
### MDP Value Iteration and Policy Iteration
import argparse
import time

import gymnasium as gym
import numpy as np
from envs import *

np.set_printoptions(precision=3)

parser = argparse.ArgumentParser(
    description="A program to run assignment 1 implementations.",
    formatter_class=argparse.ArgumentDefaultsHelpFormatter,
)

parser.add_argument(
    "--env",
    type=str,
    help="The name of the environment to run your algorithm on.",
    choices=["Deterministic-4x4-FrozenLake-v0", "Stochastic-4x4-
FrozenLake-v0"],
    default="Deterministic-4x4-FrozenLake-v0",
)

parser.add_argument(
    "--render-mode",
    "-r",
    type=str,
    help="The render mode for the environment. 'human' opens a window
to render. 'ansi' does not render anything.",
    choices=["human", "ansi"],
```

```
    default="human",
)

"""
For policy_evaluation, policy_improvement, policy_iteration and
value_iteration,
the parameters P, nS, nA, gamma are defined as follows:

    P: nested dictionary of a nested lists
        From gym.core.Environment
        For each pair of states in [1, nS] and actions in [1, nA],
P[state][action] is a
               tuple of the form (probability, nextstate, reward,
terminal) where
                      - probability: float
                             the probability of transitioning from
"state" to "nextstate" with "action"
                      - nextstate: int
                             denotes the state we transition to (in
range [0, nS - 1])
                      - reward: int
                             either 0 or 1, the reward for
transitioning from "state" to
                             "nextstate" with "action"
                      - terminal: bool
                        True when "nextstate" is a terminal state (hole
or goal), False otherwise
       nS: int
               number of states in the environment
       nA: int
               number of actions in the environment
       gamma: float
               Discount factor. Number in range [0, 1)
"""


def policy_evaluation(P, nS, nA, policy, gamma=0.9, tol=1e-3):
    """Evaluate the value function from a given policy.

       Parameters
       ----------
       P, nS, nA, gamma:
               defined at beginning of file
       policy: np.array[nS]
```

```
                The policy to evaluate. Maps states to actions.
        tol: float
                Terminate policy evaluation when
                        max |value_function(s) - prev_value_function(s)|
< tol
        Returns
        -------
        value_function: np.ndarray[nS]
                The value function of the given policy, where
value_function[s] is
                the value of state s
        """

    value_function = np.zeros(nS)

    ############################
    # YOUR IMPLEMENTATION HERE #
    ############################
    return value_function


def policy_improvement(P, nS, nA, value_from_policy, policy,
gamma=0.9):
    """Given the value function from policy improve the policy.

        Parameters
        ----------
        P, nS, nA, gamma:
                defined at beginning of file
        value_from_policy: np.ndarray
                The value calculated from the policy
        policy: np.array
                The previous policy.

        Returns
        -------
        new_policy: np.ndarray[nS]
                An array of integers. Each integer is the optimal action
to take
                in that state according to the environment dynamics and
the
                given value function.
        """
```

```
    new_policy = np.zeros(nS, dtype="int")

    ############################
    # YOUR IMPLEMENTATION HERE #
    ############################
    return new_policy


def policy_iteration(P, nS, nA, gamma=0.9, tol=1e-3):
    """Runs policy iteration.

        You should call the policy_evaluation() and policy_improvement()
methods to
        implement this method.

        Parameters
        ----------
        P, nS, nA, gamma:
                defined at beginning of file
        tol: float
                tol parameter used in policy_evaluation()
        Returns:
        ----------
        value_function: np.ndarray[nS]
        policy: np.ndarray[nS]
        """

    value_function = np.zeros(nS)
    policy = np.zeros(nS, dtype=int)

    ############################
    # YOUR IMPLEMENTATION HERE #
    ############################
    return value_function, policy


def value_iteration(P, nS, nA, gamma=0.9, tol=1e-3):
    """
        Learn value function and policy by using value iteration method
for a given
        gamma and environment.

        Parameters:
        ----------
```

```
        P, nS, nA, gamma:
                defined at beginning of file
        tol: float
                Terminate value iteration when
                        max |value_function(s) - prev_value_function(s)|
< tol
        Returns:
        ----------
        value_function: np.ndarray[nS]
        policy: np.ndarray[nS]
        """

    value_function = np.zeros(nS)
    policy = np.zeros(nS, dtype=int)
    ############################
    # YOUR IMPLEMENTATION HERE #
    ############################
    return value_function, policy


def render_single(env, policy, max_steps=100):
    """
    This function does not need to be modified
    Renders policy once on environment. Watch your agent play!

    Parameters
    ----------
    env: gym.core.Environment
      Environment to play on. Must have nS, nA, and P as
      attributes.
    Policy: np.array of shape [env.nS]
      The action to take at a given state
  """

    episode_reward = 0
    ob, _ = env.reset()
    for t in range(max_steps):
        env.render()
        time.sleep(0.25)
        a = policy[ob]
        ob, rew, done, _, _ = env.step(a)
        episode_reward += rew
        if done:
            break
```

```
    env.render()
    if not done:
        print(
            "The agent didn't reach a terminal state in {}
steps.".format(
                max_steps
            )
        )
    else:
        print("Episode reward: %f" % episode_reward)


# Edit below to run policy and value iteration on different
environments and
# visualize the resulting policies in action!
# You may change the parameters in the functions below
if __name__ == "__main__":
    # read in script argument
    args = parser.parse_args()

    # Make gym environment
    env = gym.make(args.env, render_mode=args.render_mode)

    env.nS = env.nrow * env.ncol
    env.nA = 4

    print("\n" + "-" * 25 + "\nBeginning Policy Iteration\n" + "-" *
25)

    V_pi, p_pi = policy_iteration(env.P, env.nS, env.nA, gamma=0.9,
tol=1e-3)
    render_single(env, p_pi, 100)

    print("\n" + "-" * 25 + "\nBeginning Value Iteration\n" + "-" * 25)

    V_vi, p_vi = value_iteration(env.P, env.nS, env.nA, gamma=0.9,
tol=1e-3)
    render_single(env, p_vi, 100)
```

envs.py

```
import gymnasium as gym
from gymnasium.envs.registration import register

env_dict = gym.envs.registration.registry.copy()
for env in env_dict:
    if "Deterministic-4x4-FrozenLake-v0" in env:
        del gym.envs.registration.registry[env]
    elif "Stochastic-4x4-FrozenLake-v0" in env:
        del gym.envs.registration.registry[env]


register(
    id="Deterministic-4x4-FrozenLake-v0",
    entry_point="gymnasium.envs.toy_text.frozen_lake:FrozenLakeEnv",
    kwargs={"map_name": "4x4", "is_slippery": False},
)

register(
    id="Stochastic-4x4-FrozenLake-v0",
    entry_point="gymnasium.envs.toy_text.frozen_lake:FrozenLakeEnv",
    kwargs={"map_name": "4x4", "is_slippery": True},
)
```