上海交通大學

SHANGHAI JIAOTONG UNIVERSITY

强化学习课程设计 - Project1

Dynamic Programming

姓名：薛春宇

学号：518021910698

完成时间：2021/3/24

1 实验目的

基于动态规划的思想，分别使用 Policy Iteration 和 Value Iteration 优化随机策略，以解决 Gridworld 下的最短路径问题。

2 实验准备

在本节中，我们将分别介绍 Policy Iteration 和 Value Iteration 的相关内容，以为接下来的代码实现做准备。

2.1 策略迭代 Policy Iteration

Policy Iteration 是本次实验中实现起来最具挑战性的模块，其可以分为两个部分，其名称及作用分别是：

(1) 策略评估 Policy Evaluation：基于已有的策略 π，遍历 MDP 中的每一个状态 s，使用贝尔曼方程对值函数 V 进行迭代更新，直到满足结束条件趋于收敛。

(2) 策略提升 Policy Improvement：基于已有的值函数 V，使用贝尔曼方程对策略 π 遍历 MDP 中的每一个状态 s 进行一次更新，并判断策略是否发生了改变，若没有改变，则将值函数 V 和策略 π 返回；否则重复 policy evaluation 的步骤。

整个 policy iteration 的伪代码如下所示：

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s',r|s,\pi(s))\left[r + \gamma V(s')\right]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s',r|s,a)\left[r + \gamma V(s')\right]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

我们的 python 代码也是基本按照上述伪代码的思想进行实现的。

2.2 值迭代 Value Iteration

相较于策略迭代，value iteration 在整个运行过程中只会在最终输出确定性策略的时候，使用贝尔曼方程，基于已有的值函数 V 求出策略 π 。Value iteration 的主体部分是一个针对值函数 V 的迭代更新，同样是基于贝尔曼方程，并设置一个阈值来控制迭代的结束。

Value iteration 的伪代码如下所示：

Value Iteration, for estimating $\pi \approx \pi_*$

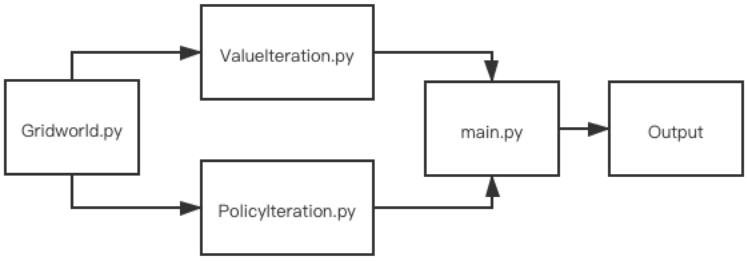
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a)\left[r + \gamma V(s')\right]$

3 实验内容

本次实验的代码及逻辑结构如下所示：



接下来，我们将对四个主要文件进行逐一分析，并分析两类迭代算法的性能差异。相关代码位于./code 目录下。

3.1 Gridworld.py

该模块的实现参考了 Sutton 版 RL 提供的 Gridworld 类，实现了一个可实例化的格子世界，可以通过如下指令进行实例化：

```
# Grid Map Definition
env_for_value_iter = GridworldEnv([6, 6])
env_for_policy_iter = GridworldEnv([6, 6])
```

需要注意的是，本项目中的 Terminal State、Action Tpye、Reward 和概率转移矩阵 P

均需要在 Gridworld.py 中声明。本项目中的 Gridworld 需要被实例化为如下结构（其中 1 和 35 号 grid 被设置为 terminal state）：

0	1	2	3	4	5
6	7	8	9	10	11
12	13	14	15	16	17
18	19	20	21	22	23
24	25	26	27	28	29
30	31	32	33	34	35

3.2 PolicyIteration.py

该模块的实现基本遵循了 2.1 节中给出的伪代码的思想，并将整体内容设置成一个可调用的函数 policy_iteration()，供用户在 main.py 中调用。

```
    # Value function
    V = np.zeros(env.nS)
    # Iteration step
    iteration_step = 0
    # Update time for value function
    update_times_for_value_function = 0
    # Create a deterministic policy using the optimal value function.
    my_policy = np.zeros([env.nS, env.nA])
    # Policy Initialization. Policy is initialized to be 0.25 for all 4 directions
    for state in range(env.nS):
        for action in range(env.nA):
            my_policy[state][action] = 0.25
```

在函数的开始部分，我们根据已知的参数大小（env.nS 为 Gridworld 的全部状态数，env.nA 为 Gridworld 的全部行为数）来创建值函数及策略的容器，并进行策略的初始化（四个方向各位 0.25）。

```
while True:
    iteration_step += 1

    # Policy Evaluation, Update value function in current policy
    while True:...

    # Policy Improvement
    policy_stable = True  # Whether this policy is stable
    for state in range(env.nS):...

    # Stable
    if policy_stable:
        print("Stop. Totally", iteration_step, "iterations for Policy Iteration,", update_times_for_value_function, "times for the update
        return my_policy, V
```

函数主体为一个 while 循环，分为 policy evaluation 和 policy improvement 两个子模块，最后若策略稳定，则结束循环将策略和值函数返回。

```
        # Policy Evaluation, Update value function in current policy
        while True:
            update_times_for_value_function += 1
            # Stop condition
            _delta = 0
            # Update each state
            for state in range(env.nS):
                origin_value = V[state]
                # Update value function in current policy
                V[state] = calculate_action_value(env, state, V, _discount_factor, my_policy)
                # Calculate _delta across all states seen so far
                _delta = max(_delta, np.abs(origin_value - V[state]))
            if _delta < _theta:
                break
```

第一个模块为策略评估 policy evaluation，设置_delta 为停止条件，在一个 while 循环中，调用 calculate_action_value()函数实现贝尔曼方程对值函数 V 进行更新迭代，并更新_delta，当小于预设的阈值_theta 时退出该模块。子函数的实现如下：

```
# Calculate the value for all actions in a given state, one-step lookahead
# state: state (int)
# V: value function, vector with the length of env.nS
# _discount_factor: discount factor
# my_policy: current policy (FIXED in Policy Evaluation part)
# Return: a
def calculate_action_value(env, state, V, _discount_factor, my_policy):
    A = 0
    for action in range(env.nA):
        for prob, next_state, reward, isDone in env.P[state][action]:
            A += my_policy[state][action] * prob * (reward + _discount_factor * V[next_state])
    return A
```

Calculate_action_value()函数严格按照 policy evluation 的伪代码进行实现。

```
        # Policy Improvement
        policy_stable = True  # Whether this policy is stable
        for state in range(env.nS):
            # Get the old action (with the largest probability in my_policy[state])
            maxElm = -100000000.0
            old_actions = []
            for action in range(env.nA):
                if my_policy[state][action] > maxElm:
                    maxElm = my_policy[state][action]
                    old_actions.clear()
                    old_actions.append(action)
                elif my_policy[state][action] == maxElm:
                    old_actions.append(action)

            # Get the currently best action by using greedy search
            best_actions = greedy_policy(env, state, V, _discount_factor)
            # Policy is changed in this update round
            # Note that the policy grid may be more than one direction!
            if len(old_actions) != len(best_actions):
                policy_stable = False
            else:
                for i in range(0, len(best_actions), 1):
                    if best_actions[i] not in old_actions:
                        policy_stable = False
                        break

            if not policy_stable:
                prob = 1.0 / float(len(best_actions))
                for action in range(env.nA):
                    my_policy[state][action] = 0.0
                for i in range(0, len(best_actions), 1):
                    my_policy[state][best_actions[i]] = prob
```

第二个模块为策略提升 policy improvement，设置一个布尔变量 policy_stable 指示

在本次执行中策略是否发生了变化。执行主体为一个遍历所有状态 s 的 for 循环，需要特别的注意的是，在判断 old_policy 和 best_policy 是否相同的时候，需要考虑到二者均可为一个包含一个以上 action 的列表，原因是在调用贪心算法 greedy_policy()寻找当前值函数的最优策略时，每个 state 对应的最优 action 可能并不止一个：

```python
# Greedy policy
def greedy_policy(env, state, V, _discount_factor):
    A = np.zeros(env.nA)
    for action in range(env.nA):
        for prob, next_state, reward, isDone in env.P[state][action]:
            A[action] += prob * (reward + _discount_factor * V[next_state])
    maxElm = -100000000.0
    # Note that the actions corresponding to the maximum A may be more than one!
    ret = []
    for action in range(env.nA):
        if A[action] > maxElm:
            maxElm = A[action]
            ret.clear()
            ret.append(action)
        elif A[action] == maxElm:
            ret.append(action)
    return ret
```

因此在判断最大值的时候需要特别注意，需要使用一个 list 来容纳可能不止一个的 best action 并返回。此外，在更新 policy 的时候，需要将位于 list 中的 action 对应的 my_policy[state][action]置为 1/len(list)，其他则置为 0。

3.3 ValueIteration.py

该模块的实现基本遵循了 2.2 节中给出的伪代码的思想，并将整体内容设置成一个可调用的函数 value_iteration()，供用户在 main.py 中调用

```python
    # Value function
    V = np.zeros(env.nS)
    # Iteration step
    iteration_step = 0
    # policy
    my_policy = np.zeros([env.nS, env.nA])
```

在函数的开始部分，我们根据已知的参数大小（env.nS 为 Gridworld 的全部状态数，env.nA 为 Gridworld 的全部行为数）来创建值函数及策略的容器。由于本模块是对值函数而非策略进行迭代，我们不需要单独对 my_policy 容器进行初始化。

```python
    # print("Current iteration step: ", iteration_step)
    iteration_step += 1
    # Stop condition
    _delta = 0
    # Update each state
    for state in range(env.nS):
        origin_value = V[state]
        # Calculate best action, one-step lookahead, and update the value function
        V[state] = calculate_action_value(env, state, V, _discount_factor, is_output=False)
        # Calculate _delta across all states seen so far
        _delta = max(_delta, np.abs(origin_value - V[state]))

    if _delta < _theta:
        print("Stop. Totally", iteration_step, "iterations for Value Iteration (update times of value function).")
        break
```

模块主体由一个 while 循环组成，第一个子模块为对值函数 V 的迭代。这里调用了 calculate_action_value()函数来实现贝尔曼方程：

```python
# Calculate the value for all actions in a given state, one-step lookahead
# state: state (int)
# V: value function, vector with the length of env.nS
# _discount_factor: discount factor
# is_output: a flag indicates that whether it is called in output stage
# Return: max value among these possible values or its arg (depends on whether in output stage)
def calculate_action_value(env, state, V, _discount_factor, is_output):
    A = np.zeros(env.nA)
    for action in range(env.nA):
        for prob, next_state, reward, isDone in env.P[state][action]:
            A[action] += prob * (reward + _discount_factor * V[next_state])
    if not is_output:
        return np.max(A)
    else:
        return np.argmax(A)
```

值得注意的是，在最后输出确定性策略的时候，我们也会调用该函数，不过会将 is_output 设置为 True，以获得每个 state 的最优 action。

```python
    # Output a deterministic policy
    for state in range(env.nS):
        # Get optimal direction
        direction = calculate_action_value(env, state, V, _discount_factor, is_output=True)
        # Update policy (make choice)
        for action in range(env.nA):
            if action == direction:
                my_policy[state][action] = 1
            else:
                my_policy[state][action] = 0

    return my_policy, V
```

输出确定性策略的子模块如上所示。

3.4 main.py

最后的主函数主要负责 Gridworld 类的实例化，两类迭代函数的调用以及输出的格式化。

```python
import numpy as np
from Gridworld import GridworldEnv
from ValueIteration import value_iteration
from PolicyIteration import policy_iteration

# Grid Map Definition
env_for_value_iter = GridworldEnv([6, 6])
env_for_policy_iter = GridworldEnv([6, 6])


def value_iteration_policy(env):...

def policy_iteration_policy(env):...


if __name__ == '__main__':
    # Value Iteration
    value_iteration_policy(env_for_value_iter)
    # Policy Iteration
    policy_iteration_policy(env_for_policy_iter)
```

其中，value_iteration()和 policy_iteration()分别在 value_iteration_policy()和 policy_iteration_policy()两个封装函数中调用，输出的格式化也在里面实现。

3.5 Policy Iteration 和 Value Iteration 的性能对比

策略迭代的运行结果如下：

```
Stop. Totally 3 iterations for Policy Iteration, 223 times for the update of value function
----------------------------------
Policy Iteration
----------------------------------
Reshaped Policy (0=UP, 1=RIGHT, 2=DOWN, 3=LEFT):
[[1 0 3 3 3 3]
 [0 0 0 0 0 2]
 [0 0 0 0 1 2]
 [0 0 0 1 1 2]
 [0 0 1 1 1 2]
 [1 1 1 1 1 0]]
----------------------------------
Final Value function:
[[-1.  0. -1. -2. -3. -4.]
 [-2. -1. -2. -3. -4. -4.]
 [-3. -2. -3. -4. -4. -3.]
 [-4. -3. -4. -4. -3. -2.]
 [-5. -4. -4. -3. -2. -1.]
 [-5. -4. -3. -2. -1.  0.]]
```

值迭代的运行结果如下：

```
Stop. Totally 6 iterations for Value Iteration (update times of value function).
----------------------------------
Value Iteration
----------------------------------
Reshaped Policy (0=UP, 1=RIGHT, 2=DOWN, 3=LEFT):
[[1 0 3 3 3 3]
 [0 0 0 0 0 2]
 [0 0 0 0 1 2]
 [0 0 0 1 1 2]
 [0 0 1 1 1 2]
 [1 1 1 1 1 0]]
----------------------------------
Final Value Function:
[[-1.  0. -1. -2. -3. -4.]
 [-2. -1. -2. -3. -4. -4.]
 [-3. -2. -3. -4. -4. -3.]
 [-4. -3. -4. -4. -3. -2.]
 [-5. -4. -4. -3. -2. -1.]
 [-5. -4. -3. -2. -1.  0.]]
```

根据上述结果可以看到，虽然 policy iteration 的值函数更新次数（223 次）要远多于 value iteration 的值函数更新次数（6 次），但其总的迭代次数（3 次）相比之下要比 value iteration 的迭代次数（6 次）少一半，收敛速度也比较快。但由于其每次迭代都需要进行策略的生成，总体的运行速度还是要比 value iteration 要慢。实际上，我们可以采取一定的方法让 policy iteration 提前终止，以提高其运行速度。

4 实验结果

对比两种迭代方案的运行结果可以看到，两类迭代方案的结果完全相同且正确。本次实验成功。

5 实验心得

在本次实验的过程中，我首先在课堂授课内容的基础上，系统性地学习了 policy iteration 和 value iteration 的相关知识，并掌握了其在代码层面上的实现方法。在实现的过程中，由于从伪代码到 python 代码的差别还是比较大的，我遇到了一些困难，其中就包括了在判别策略是否稳定的过程中，没有考虑到一个 state 可能会对应多种最优 action 的情况，因此走了不少弯路。

整个过程约花费半天时间，在本次实验中，我不仅了解了两种迭代方案的基本知识，掌握了从伪代码到可运行代码的复现方法，更是提高了自身发现 bug，解决 bug 的能力。希望在接下来的实验中也能收获满满！