

悬崖漫步实验报告

201250182 郑义

悬崖漫步环境设计

悬崖漫步的环境是一个 4×12 的网格世界，每一个世界都代表一个状态。智能体的起点是左下角的状态，目标是右下角的状态，智能体在每一个状态都可以采取四种动作：上、下、左、右。如果智能体采取动作后触碰到边界墙壁则状态不发生改变，否则就会转移到下一个状态。环境中存在着一段悬崖，如果智能体的下一个状态是处于悬崖的网格，那么智能体就会被赋予 -100 的奖励，并终止漫步。除此之外，网格世界中还存在着一个目标状态，智能体到达目标时也被视为终止状态。智能体每走一步的奖励是 -1。

因此，悬崖漫步的目的就变成如何最大化奖励来让目标到达目标状态。

使用 python 来模拟这个网格世界：

```
class CliffWalkingEnv:
    """
    悬崖漫步环境
    """

    def __init__(self, col_num=12, row_num=4):
        # 定义网格世界的列
        self.col_num = col_num
        # 定义网格世界的行
        self.row_num = row_num
        # 转移矩阵 transition_matrix[state][action] = [(p, next_state,
        reward, done)] 包含下一个状态和奖励
        self.transition_matrix = self.create_transition()

    def create_transition(self):
        # init
        transition = [[[[] for _ in range(4)] for _ in range(self.row_num *
        self.col_num)]]
        # 四种动作, change[0]: 上; change[1]: 下; change[2]: 左; change[3]: 右
        change = [[0, -1], [0, 1], [-1, 0], [1, 0]]
        for i in range(self.row_num):
            for j in range(self.col_num):
                for a in range(4):
                    # 位置在悬崖或者目标状态，因为无法继续交互，任何动作奖励都为 0
                    if i == self.row_num - 1 and j > 0:
                        transition[i * self.col_num + j][a] = [(1, i *
        self.col_num + j, 0, True)]
                    continue
                    # 其他位置
                    next_x = min(self.col_num - 1, max(0, j + change[a]
        [0]))
                    next_y = min(self.row_num - 1, max(0, i + change[a]
        [1]))
                    next_state = next_y * self.col_num + next_x
```

```
        reward = -1
        done = False
        # 下一个位置在悬崖或终点
        if next_y == self.row_num - 1 and next_x > 0:
            done = True
            # 下一个位置在悬崖
            if next_x != self.col_num - 1:
                reward = -100
        transition[i * self.col_num + j][a] = [(1, next_state,
reward, done)]

    return transition
```

策略迭代过程

策略迭代实际上就是两个步骤：

1. 策略评估
2. 策略提升

策略迭代实际上就是这两个步骤不断交替，直到得到了最后的最优策略的过程。策略评估代码：

```
def policy_evaluation(self):
    cnt = 1
    while True:
        max_diff = 0
        new_val = [0] * self.env.col_num * self.env.row_num
        for s in range(self.env.col_num * self.env.row_num):
            # 开始计算状态 s 下的所有 Q(s, a) 价值
            qsa_list = self.calc_qsa(s, True)
            new_val[s] = sum(qsa_list)
            max_diff = max(max_diff, abs(new_val[s] - self.val[s]))
        self.val = new_val
        if max_diff < self.theta:
            break
        cnt += 1
    print("策略评估进行 %d 轮后完成" % cnt)
```

策略评估的过程实际上就是去进行迭代，不断的去探寻策略，通过比较上一轮策略和这一轮策略的差异（`max_diff`），只有当其小于阈值（这里是 `self.theta`）时认为其收敛，此时对其的评估完成。

策略提升代码：

```
def policy_improvement(self):
    for s in range(self.env.row_num * self.env.col_num):
        qsa_list = self.calc_qsa(s, False)
        max_qsa = max(qsa_list)
        # 计算有几个动作得到了最大的 Q 值
        cnt_qsa = qsa_list.count(max_qsa)
```

```

        # 让这些动作均分概率
        self.pi[s] = [1 / cnt_qsa if q == max_qsa else 0 for q in
qsa_list]
        print("策略提升完成")
        return self.pi

```

策略提升的动作实际上也是去计算每个状态的奖励，只有获得了最大 Q 值的动作可以被纳入，并均分执行该动作的概率。

为代码：

初始化：

初始化状态值函数 $V(s)$ 为 0，策略 π 为一个二维数组，表示每个状态下的动作选择概率。

对每个状态 s ，都将 $\pi[s][a]$ 初始化为 $1/|A(s)|$ ，其中 $|A(s)|$ 表示状态 s 下可行动作的数量。

循环直至收敛：

进入策略评估阶段：

对于每个状态 s ：

利用当前策略 π 以及 bellman 方程计算出新的状态值函数 $V(s)$ 。

直至状态值函数收敛。

进入策略改进阶段：

对于每个状态 s ：

计算新的最优策略 π_{new} ，使其在当前状态 s 下的期望收益最大化。

若 π_{new} 与当前策略 π 相同，则跳过该状态；否则将 π 更新为 π_{new} 。

若策略已经收敛，则退出循环。

返回最优策略 π 。

价值迭代过程

伪代码：

初始化：

初始化状态值函数 $V(s)$ 为 0

循环直到收敛：

对于每个状态 s ：

计算状态 s 下执行每个动作 a 的收益 $Q(s, a)$ ：

$Q(s, a) = R(s, a) + \gamma \sum(p(s', r|s, a) * V(s'))$

将状态值函数 $V(s)$ 更新为 $Q(s, a)$ 中的最大值

如果 $V(s)$ 的变化小于一个给定的阈值 θ ，则跳出循环

返回最优策略 π ，对于每个状态 s ，选择使得 $Q(s, a)$ 最大的动作 a

代码：

```
while True:
    max_diff = 0
    new_val = [0] * self.env.col_num * self.env.row_num
    for s in range(self.env.col_num * self.env.row_num):
        # 开始计算状态 s 下的所有 Q(s, a) 价值
        qsa_list = []
        for a in range(4):
            qsa = 0
            for res in self.env.transition_matrix[s][a]:
                p, next_state, r, done = res
                qsa += p * (r + self.gamma * self.val[next_state] * (1 -
done))

            # 这一行和下一行是价值迭代和策略迭代的主要区别
            qsa_list.append(qsa)
        new_val[s] = max(qsa_list)
        max_diff = max(max_diff, abs(new_val[s] - self.val[s]))
    self.val = new_val
    if max_diff < self.theta:
        break
    cnt += 1
```

总结

策略迭代结果：

策略评估进行 60 轮后完成

策略提升完成

策略评估进行 72 轮后完成

策略提升完成

策略评估进行 44 轮后完成

策略提升完成

策略评估进行 12 轮后完成

策略提升完成

策略评估进行 1 轮后完成

策略提升完成

状态价值:

```
-7.712 -7.458 -7.176 -6.862 -6.513 -6.126 -5.695 -5.217 -4.686 -4.095 -3.439 -2.710
-7.458 -7.176 -6.862 -6.513 -6.126 -5.695 -5.217 -4.686 -4.095 -3.439 -2.710 -1.900
-7.176 -6.862 -6.513 -6.126 -5.695 -5.217 -4.686 -4.095 -3.439 -2.710 -1.900 -1.000
-7.458 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
```

策略:

```
OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV00
OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV00
000> 000> 000> 000> 000> 000> 000> 000> 000> 000> 000> OV00
^000 ***** EEEE
```

价值迭代结果:

价值迭代一共进行 14 轮

状态价值:

```
-7.712 -7.458 -7.176 -6.862 -6.513 -6.126 -5.695 -5.217 -4.686 -4.095 -3.439 -2.710
-7.458 -7.176 -6.862 -6.513 -6.126 -5.695 -5.217 -4.686 -4.095 -3.439 -2.710 -1.900
-7.176 -6.862 -6.513 -6.126 -5.695 -5.217 -4.686 -4.095 -3.439 -2.710 -1.900 -1.000
-7.458 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000 0.000
```

策略:

```
OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV00
OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV0> OV00
000> 000> 000> 000> 000> 000> 000> 000> 000> 000> 000> OV00
^000 ***** EEEE
```

策略迭代和价值迭代的区别:

1. Policy Iteration 通常是 policy evaluation + policy improvement 交替执行直到收敛
2. Value iteration 通常是寻找 Optimal value function + 一次 policy extraction, 它们不用交替执行, 因为值函数最优, 策略通常也是最优

3. 寻找 optimal value function 也可以被看作是 policy improvement (due to max) 和截断版的 policy evaluation 的组合(仅在一次扫描所有状态后重新分配 $V(s)$ 而不考虑其收敛性的组合)