机器学习 第五次作业

RL 作业

201250076 袁家乐

2023 年 4 月 21 日

一、实现环境、策略、价值编码

本次 RL 作业实现了悬崖漫步，采用一个 4×12 的网格世界，其中第四行除首尾外的网格设定为悬崖，第四行首格为起点，第四行末格为终点（悬崖和终点均为终止状态）。各网格表示状态，分为上下左右四种动作（以'^', 'v', '<', '>'表示），每走一步的奖励为-1，掉入悬崖的奖励为-100，初始时各状态的价值函数均为 0，初始时的策略为上下左右完全随机走动。

本次实验使用 python3.9 完成。

二、策略迭代和价值迭代伪代码描述

1、策略迭代伪代码

$\pi(s) \leftarrow random(i)$
$V(s) \leftarrow 0$
$iteration$:
$while\ \Delta > \theta\ do$:
  $\Delta \leftarrow 0$
  $for\ s \in S$:
    $v \leftarrow V(S)$
    $V(s) \leftarrow r(s,\pi(s)) + \gamma \sum_{s'} P(s' \mid s,\pi(s))V(s')$
    $\Delta \leftarrow max(\Delta, |v - V(s)|)$
$end\ while$

$\pi_{old} \leftarrow \pi$
$for\ s \in S$:
  $\pi(s) \leftarrow arg\ max_a\ r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)V(s')$
$if\ \pi_{old}! = \pi$:
  $jmp\ iteration$

在策略迭代中，初始策略为完全随机，初始价值函数均置为 0。然后对当前策略进行策略评估，得到其价值函数。再根据该价值函数进行策略提升以得到更

好的新策略。持续进行策略评估、策略提升，直至策略不再改变（收敛到最优）为止。

2、价值迭代伪代码

$\pi(s) \leftarrow random(i)$
$V(s) \leftarrow 0$
$while\ \Delta > \theta\ do:$
$\quad \Delta \leftarrow 0$
$\quad for\ s \in S:$
$\quad\quad v \leftarrow V(S)$
$\quad\quad V(s) \leftarrow max(r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)V(s'))$
$\quad\quad \Delta \leftarrow max(\Delta, |v - V(s)|)$
$end\ while$

$for\ s \in S:$
$\quad \pi(s) \leftarrow arg\ max_a\ r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)V(s')$

在价值迭代中，初始策略为完全随机，初始价值函数均置为 0。然后对当前策略进行策略评估，得到其价值函数，只在策略评估中进行一轮价值更新，然后直接根据更新后的价值进行策略提升。价值迭代中不存在显式的策略，我们只维护一个状态价值函数。

三、代码详述

1、悬崖漫步环境设计

将悬崖漫步环境设计为一个单独的类，其成员变量主要有行数（此作业为 4）、列数（此作业为 12）、转移矩阵。

每个转移矩阵中包含了各格点上下左右四种动作下的转移情况（p 乘数，下一个状态的下标，奖励，是否终止）

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

    def __init__(self, ncol=12, nrow=4):
        """
            类构造方法
            :param ncol 列数
            :param nrow 行数
        """
        self.ncol = ncol  # 定义网格世界的列
        self.nrow = nrow  # 定义网格世界的行
        # 转移矩阵P[state][action] = [(p, next_state, reward, done)]包含下一个状态和奖励
        self.P = self.createP()
```

在初始化该类时，调用 createP()方法来填充转移矩阵。填充时，需考虑一般位置、下一位置在悬崖或终点、当前位置为悬崖和终点三大类情况。下一步掉入悬崖的，奖励为-100；已处于终止状态的，奖励为 0；其它的奖励均为-1。

```
P = [[[] for j in range(4)] for i in range(self.nrow * self.ncol)]
# 4种动作, change[0]:上,change[1]:下, change[2]:左, change[3]:右。坐标系原点(0,0)
change = [[0, -1], [0, 1], [-1, 0], [1, 0]]
for i in range(self.nrow):
    for j in range(self.ncol):
        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.nrow - 1 and j > 0:
                P[i * self.ncol + j][a] = [(1, i * self.ncol + j, 0, True)]
                continue
            # 其他位置
            next_x = min(self.ncol - 1, max(0, j + change[a][0]))
            next_y = min(self.nrow - 1, max(0, i + change[a][1]))
            next_state = next_y * self.ncol + next_x
            reward = -1
            done = False
            # 下一个位置在悬崖或者终点
            if next_y == self.nrow - 1 and next_x > 0:
                done = True
                if next_x != self.ncol - 1:  # 下一个位置在悬崖
                    reward = -100
            P[i * self.ncol + j][a] = [(1, next_state, reward, done)]
return P
```

2、描述策略迭代过程

将策略迭代算法设计为一个单独的类。其成员变量主要有悬崖漫步环境、策略评估阈值（此作业中设为 0.001）、折扣因子（此作业中设为 0.9）、价值函数（初始化为 0）、策略（初始化为完全随机）。

```
class PolicyIteration:
    """
        策略迭代算法类
    """

    def __init__(self, env, theta, gamma):
        """
            类构造方法
            :param env 悬崖漫步环境
            :param theta 策略评估收敛阈值
            :param gamma 折扣因子
        """
        self.env = env
        self.v = [0] * self.env.ncol * self.env.nrow  # 初始化价值为0
        self.pi = [[0.25, 0.25, 0.25, 0.25] for i in range(self.env.ncol * self.env.nrow)]  # 初始化为均匀随机策略
        self.theta = theta  # 策略评估收敛阈值
        self.gamma = gamma  # 折扣因子
```

将策略评估封装为policy_evaluation()方法，使用贝尔曼方程来迭代计算当前策略在各状态的价值函数（用上一轮的状态价值函数来计算当前轮次的状态价值函数）。

$$V^{\pi}(s) \leftarrow r(s,\pi(s)) + \gamma \sum_{s'} P(s' \mid s,\pi(s)) V^{\pi_{old}}(s')$$

由于需要不断地进行贝尔曼期望方程的迭代，策略评估实际上计算代价较大，我们以 0.001 为阈值，当差异小于此阈值时即停止迭代，并不苛求完全无差异时才停止。

```
cnt = 1  # 计数器
while 1:
    max_diff = 0
    new_v = [0] * self.env.ncol * self.env.nrow
    for s in range(self.env.ncol * self.env.nrow):
        qsa_list = []  # 开始计算状态s下的所有Q(s,a)价值
        for a in range(4):
            qsa = 0
            for res in self.env.P[s][a]:
                p, next_state, r, done = res
                qsa += p * (r + self.gamma * self.v[next_state] * (1 - done))
                # 环境比较特殊,奖励和下一个状态有关,所以需要和状态转移概率相乘
            qsa_list.append(self.pi[s][a] * qsa)
        new_v[s] = sum(qsa_list)  # 状态价值函数和动作价值函数之间的关系
        max_diff = max(max_diff, abs(new_v[s] - self.v[s]))
    self.v = new_v
    if max_diff < self.theta:
        break  # 满足收敛条件,退出评估迭代
    cnt += 1
print("策略评估进行%d轮后完成" % cnt)
```

将策略提升封装为policy_improvement()方法。在策略评估计算出当前策略的状态价值函数后，使用该方法来优化当前策略。依次计算各状态的各个动作价值函数，从中选出最大价值函数的动作作为新的策略（若有多个动作同时具备最大值，则将这些动作概率均分作为新的策略）。这是一种局部最优的贪心做法。

```
for s in range(self.env.nrow * self.env.ncol):
    qsa_list = []
    for a in range(4):
        qsa = 0
        for res in self.env.P[s][a]:
            p, next_state, r, done = res
            qsa += p * (r + self.gamma * self.v[next_state] *(1 - done))
        qsa_list.append(qsa)
    maxq = max(qsa_list)
    cntq = qsa_list.count(maxq)  # 计算有几个动作得到了最大的Q值
    # 让这些动作均分概率
    self.pi[s] = [1 / cntq if q == maxq else 0 for q in qsa_list]
print("策略提升完成")
return self.pi
```

该类真正对外提供的即为policy_iteration()方法，此方法循环执行策略评估

和策略提升，直至策略不再发生变化（收敛至最优）。

```
while 1:
    self.policy_evaluation()
    old_pi = copy.deepcopy(self.pi)  # 将列表进行深拷贝,方便接下来进行比较
    new_pi = self.policy_improvement()
    if old_pi == new_pi:
        break
```

3、描述价值迭代过程

将价值迭代算法设计为一个单独的类。其成员变量主要有悬崖漫步环境、策略评估阈值（此作业中设为 0.001）、折扣因子（此作业中设为 0.9）、价值函数（初始化为 0）、策略（初始化为完全随机）。

```
class ValueIteration:
    """
        价值迭代算法类
    """

    def __init__(self, env, theta, gamma):
        """
            类构造方法
            :param env 悬崖漫步环境
            :param theta 策略评估收敛阈值
            :param gamma 折扣因子
        """
        self.env = env
        self.v = [0] * self.env.ncol * self.env.nrow  # 初始化价值为0
        self.theta = theta  # 价值收敛阈值
        self.gamma = gamma
        # 价值迭代结束后得到的策略
        self.pi = [None for i in range(self.env.ncol * self.env.nrow)]
```

将价值迭代封装为 value_iteration()方法。价值迭代只需要在策略评估中进行一轮价值更新，然后直接根据更新后的价值进行策略提升。价值迭代中不存在显式的策略，只维护一个状态价值函数。

价值迭代是一种动态规划的过程，使用贝尔曼最优方程：

$$V^{\pi}(s) \leftarrow max\{r(s,a) + \gamma \sum_{s'} P(s' \mid s,a) V^{\pi_{old}}(s')\}$$

同样地，我们以 0.001 为阈值，当差异小于此阈值时即停止策略评估的迭代，并不苛求完全无差异时才停止。

```
cnt = 0
while 1:
    max_diff = 0
    new_v = [0] * self.env.ncol * self.env.nrow
    for s in range(self.env.ncol * self.env.nrow):
        qsa_list = []  # 开始计算状态s下的所有Q(s,a)价值
        for a in range(4):
            qsa = 0
            for res in self.env.P[s][a]:
                p, next_state, r, done = res
                qsa += p * (r + self.gamma * self.v[next_state] * (1 - done))
            qsa_list.append(qsa)  # 这一行和下一行代码是价值迭代和策略迭代的主要区别
        new_v[s] = max(qsa_list)
        max_diff = max(max_diff, abs(new_v[s] - self.v[s]))
    self.v = new_v
    if max_diff < self.theta: break  # 满足收敛条件,退出评估迭代
    cnt += 1
print("价值迭代一共进行%d轮" % cnt)
self.get_policy()
```

价值更新完成后，调用 get_policy()获取最优策略，借助下面的公式：

$$\pi(s) \leftarrow arg\ max_a\, r(s,a) + \gamma \sum_{s'} P(s' \mid s,a)V(s')$$

```
# 根据价值函数导出一个贪婪策略
for s in range(self.env.nrow * self.env.ncol):
    qsa_list = []
    for a in range(4):
        qsa = 0
        for res in self.env.P[s][a]:
            p, next_state, r, done = res
            qsa += r + p * self.gamma * self.v[next_state] * (1 - done)
        qsa_list.append(qsa)
    maxq = max(qsa_list)
    cntq = qsa_list.count(maxq)  # 计算有几个动作得到了最大的Q值
    # 让这些动作均分概率
    self.pi[s] = [1 / cntq if q == maxq else 0 for q in qsa_list]
```

依次计算各状态的各个动作价值函数，从中选出最大价值函数的动作作为新的策略（若有多个动作同时具备最大值，则将这些动作概率均分作为新的策略）。

四、实验结果

分别实践策略迭代算法和价值迭代算法，得到迭代过程、状态价值、最优策略输出如下（^, v,<,>表示上下左右，o 表示不采用该动作，*为悬崖，E 为终点）：

【策略迭代】

```
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：
ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovoo
ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovoo
ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ovoo
^ooo **** **** **** **** **** **** **** **** **** **** EEEE
```

【价值迭代】

```
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：
ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovoo
ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovo> ovoo
ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ooo> ovoo
^ooo **** **** **** **** **** **** **** **** **** **** EEEE
```

五、策略迭代和价值迭代对比讨论

策略迭代中的策略评估需要进行很多轮才能收敛，计算量很大（特别是状态和动作空间较大的情况）。有可能价值函数还未收敛，但之后无论如何更新状态的价值函数，策略均不再发生变化。因而我们可以只在策略评估中进行一轮价值更新，然后直接根据更新后的价值进行策略提升。在价值迭代中不存在显式的策略，只维护一个价值函数。