基于模型的动态规划方法

动态规划算法是基于模型的一种算法，即已知所有状态以及回报函数、状态转移概率。此算法不用进行试验收集数据，根据已知模型来进行算术求解。动态规划算法分为策略评估和策略改善两个步骤，即先以某一初始化的策略计算所有状态的值函数(策略评估)，然后利用值函数对策略进行改善(策略改善)，重复评估与改善过程最终得到最优策略。

1、策略评估

在贝尔曼小节中我们得到了 $V_\pi(s) = \Sigma_a\pi(a|s)\cdot(r_i+\gamma V_i(s'))$，即用后续状态的值函数表示当前状态的值函数。在这个式子中只有状态值函数是未知的，其余都是已知的，所以这是一个未知数个数等于状态数的方程组。可以用迭代算法解这个方程组，从而得到所有的状态值函数。

复习一下线性方程组的迭代解法。用方程 AX=b 表示一般的线性方程组，所谓的迭代算法就是根据该式设计一个迭代公式，任取初始值 $X^{(0)}$，将其代入到设计的迭代公式中得到 $X^{(1)}$，再将 $X^{(1)}$代入迭代公式中得到 $X^{(2)}$，如此循环最终得到收敛的 X。

在这里，迭代公式为 $V_{k+1}(s) = \Sigma_a\pi(a|s)\cdot(r_i+\gamma V_k(s'))$，状态 s 的后继状态 s'的值函数被初始化为 0，迭代伪代码如下：

[1] 初始化 V(s)=0

[2] Repeat k=0,1,2… # 迭代过程

[3] $V_{k+1}(S) = \Sigma_a\pi(a|S)\cdot(r_i+\gamma V_k(S'))$ # S 为向量，包含所有状态 s_1,s_2,s_3…

[4] Until $V_{k+1} = V_k$ # 迭代结束条件

至此，我们通过迭代算法算出了在某一策略下每个状态的值函数，即策略评估。

在 Sutton 的书中，其更新方式为如下。

$$\begin{aligned} v_{k+1}(s) &\doteq \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1}) \mid S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r \mid s, a)\Big[r + \gamma v_k(s')\Big], \end{aligned}$$

2、策略改善

策略改善的目的是找到一个比当前策略下值函数更大的策略。其中一个方法是，在状态 s 下，选择一个动作 a，计算其行为值函数 Q(s,a)，看其是否大于 V(s)，即看选择该动作之后带来的累积回报是否要优于按照原来策略选择动作带来的累积回报。如果 Q(s,a)>V(s)，那以后每次在状态 s 下，都选择那个动作。即在每个状态采用贪婪策略对当前策略进行改进，贪婪策略：$\pi_{l+1}(a|s)=\arg_a\max Q^{\pi_l}(s, a)$。使用贪婪策略改进后，通常在每个状态只有一个最优动作可选，但如果有多个最优动作，则可分配相同的概率给这些动作，或者至少把选择非最优动作的概率置零。

3、策略迭代算法与值函数迭代算法

1、策略迭代算法

策略迭代算法包括策略评估和策略改进两个步骤。在策略评估中，给定策略，通过数值迭代解方程组计算出该策略下每个状态的值函数，然后利用该值函数和贪婪策略得到新的策略。如此循环下去，最终得到最优策略。这是一个策略收敛的过程。伪代码如下：

[1] 初始化 V(s)=0

[2] Repeat k=0,1,2… # 迭代过程

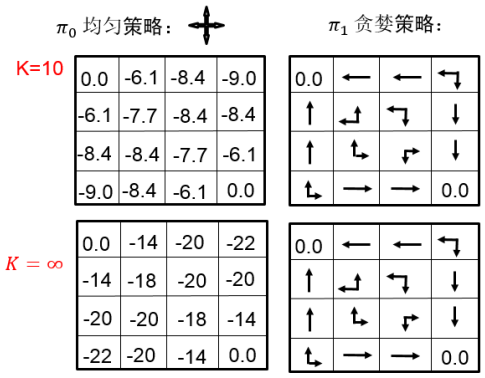
[3] find V^{π_k} # Policy evaluation：$V_{k+1}(s) = \Sigma_a \pi(a|s)\cdot(r_i+\gamma V_k(s')) = E(r_i+\gamma V_k(s'))$

[4] $\pi_{k+1}(a|s) \epsilon arg_a max Q^{\pi_k}(s, a)$ # Policy improvement

[4] Until $\pi_{k+1} = \pi_k$ # 迭代结束条件

2、值函数迭代算法

从策略迭代算法中，进行策略改进之前需要得到收敛的值函数。而值函数的收敛往往需要很多次迭代，现在的问题是进行策略改进之前一定要等到策略值函数收敛吗？下图为解状态值函数时的迭代过程中迭代次数为 10 和∞时的结果，可以看出策略评估迭代 10 次和迭代无穷次所得到的贪婪策略是一样的，所以不一定等到策略评估算法完全收敛就可以进行策略改善。如果在进行一次评估之后就进行策略改善，则称为值函数迭代算法。**值函数迭代算法无需初始化策略，因为它在操作过程中会遍历所有动作，实际上完全是一个迭代优化的过程**。在值函数迭代算法中：$v_{l+1}(s) = \max_a[R_s^a + \gamma v_l(s')]$，用最值 max 代替了平均值。



值函数迭代算法的伪代码为：第[4]条语句同时对一个状态进行策略评估和策略改善。

[1] 输入：状态转移概率 $P_{ss'}^a$ 回报函数 R_s^a ，折扣因子 γ

初始化值函数：$v(s) = 0$

[2] Repeat l=0,1,…

[3] for every s do

[4] $v_{l+1}(s) = \max_a R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_l(s')$

[5] Until $v_{l+1} = v_l$

[6] 输出：$\pi(s) = \text{argmax}_a R_s^a + \gamma \sum_{s' \in S} P_{ss'}^a v_l(s')$

$v_{l+1}(s) = \max_a[R_s^a + \gamma v_l(s')]$

而 $Q(s,a) = R_s^a + \gamma v_l(s')$

所以 $v_{l+1}(s) = \max_a Q(s,a)$

因为当在状态 s 下只有一个动作可以选择时，$Q(s,a) = V(s)$

4、代码实现

```
class DP_soft(object):   # 策略迭代算法，采用 soft-greedy 策略
    def __init__(self, env, episolon):
        self.episolon = episolon
        self.env = env
        self.gamma = 0.9
        self.V = [0.0 for _ in range(len(self.env.states))]
    # 依据初始化的值函数初始化策略，为字典结构，索引为状态，
     对应有该状态下每个动作的概率
        self.pi = self.policy_improvment()

    def policy_evaluation(self):   # 策略评估，依据当前策略计算值函数
        for i in range(10):
            delta = 0.0
            for s in self.env.states:
                # if self.env.is_terminal(s):   # 终止状态不用估计值函数也不用改善
                #         continue
                Q = []
                for a in self.env.actions:
                    self.env.state = s
                    s_, r, done = self.env.step(a)
                    if done:
                        Q.append(r)
                    else:
                        Q.append(r + self.gamma * self.V[s_])
                # 算 V[S]=∑pi(a|s)Q(s,a)
                v = 0
                for a in range(len(self.env.actions)):
                    v += Q[a] * self.pi[s][a]
                delta += abs(v - self.V[s])
                self.V[s] = v
            if delta < 1e-6:
                print("迭代了%d 次" % i)
                break
     def policy_improvment(self):
        # 策略改善就是依据评估好的状态值函数重新赋值每个动作对应的概率
        prob_s_a = dict()
        for s in self.env.states:
            # if self.env.is_terminal(s):   # 终止状态不用估计值函数也不用改善
            #         continue
            Q = []
            for a in self.env.actions:
                self.env.state = s
                s_, r, done = self.env.step(a)
                if done:
                    Q.append(r)
                else:
                    Q.append(r + self.gamma * self.V[s_])
            a_maxQ = self.env.actions[Q.index(max(Q))]
            # 重新赋值每个动作对应的概率
            a_prob = [self.episolon / len(self.env.actions) for _ in range(len(self.env.actions))]
```

```
                a_prob[a_maxQ] += 1 - self.episolon
                prob_s_a[s] = a_prob
        return prob_s_a

    def policy_iteration(self):
        for i in range(10):
            self.policy_evaluation()
            self.pi = self.policy_improvment()

    def choose_action(self, s):    # soft-greedy
        a_select = 0
        rd = random.random()
        prob = 0
        for i in range(len(self.env.actions)):
            prob += self.pi[s][i]
            if rd <= prob:
                a_select = self.env.actions[i]
                break
        return a_select


# dp_policy_iteration.py 的内容，初始化为均匀策略，然后用贪婪策略改进
    def policy_evaluate(self):
        # 策略评估，计算值函数，高斯塞德尔迭代
        for i in range(100):
            delta = 0.0
            for state in self.states:
                flag1 = yuanyang.collide(yuanyang.state_to_position(state))
                flag2 = yuanyang.find(yuanyang.state_to_position(state))
                if flag1 == 1 or flag2 == 1: continue
                action = self.pi[state]
                s, r, t = yuanyang.transform(state, action)
                # 更新值
                new_v = r + self.gamma * self.v[s]
                delta += abs(self.v[state] - new_v)    # 迭代终止条件
                # 更新值替换原来的值函数
                self.v[state] = new_v
            if delta < 1e-6:
                break

    def policy_improve(self):
        # 利用更新后的值函数，进行策略改进 v
        for state in self.states:
            flag1 = yuanyang.collide(yuanyang.state_to_position(state))
            flag2 = yuanyang.find(yuanyang.state_to_position(state))
            if flag1 == 1 or flag2 == 1: continue
            a1 = self.actions[0]
            s, r, t = yuanyang.transform(state, a1)
            Q1 = r + self.gamma * self.v[s]    # Qπ(s,ai) = ri+γVi(s')
            # 找状态 s 时，采用哪种动作，值函数最大
            for action in self.actions:
                s, r, t = yuanyang.transform(state, action)
                if Q1 < r + self.gamma * self.v[s]:    # 贪婪策略，进行更新
                    a1 = action
```

```
                    Q1 = r + self.gamma * self.v[s]
                self.pi[state] = a1    #  新的策略 self.pi[STATE]
            # print(self.pi)

        def policy_iterate(self):
            for i in range(100):
                self.policy_evaluate()    #  策略评估,变的是 v
                self.policy_improve()    #  策略改进,变的是 pi
class value_iteration:
        def __init__(self, yuanyang):
            self.states = yuanyang.states
            self.actions = yuanyang.actions
            self.v    = [ 0.0 for i in range(len(self.states) + 1)]
            self.pi = dict()
            self.yuanyang = yuanyang
            self.gamma = yuanyang.gamma
        def value_iteration(self):
            for i in range(1000):
                delta = 0.0
                for state in self.states:
                    flag1 = yuanyang.collide(yuanyang.state_to_position(state))
                    flag2 = yuanyang.find(yuanyang.state_to_position(state))
                    if flag1 == 1 or flag2 == 1: continue
                    a1= self.actions[0]
                    s, r, t = yuanyang.transform( state, a1 )
                    v1 = r + self.gamma * self.v[s]    #策略评估
                    for action in self.actions:    #策略改进
                        s, r, t = yuanyang.transform( state, action )
                        if v1 < r + self.gamma * self.v[s]:
                            a1 = action
                            v1 = r + self.gamma * self.v[s]    #  这个 r 与没有必要，可以去掉吗？
                    delta+= abs(v1 - self.v[state])
                    self.pi[state] = a1
                    self.v[state]    = v1
                if delta <    1e-6:
                    break
```