# Markov Decision Process and Dynamic Programming

- Date: Match 2019
- Material from Reinforcement Learning:An Introduction,2nd,Rechard.S.Sutton;
- Code from dennyBritze, 部分做了修改；

**content**

- MDP problems setup
- Bellman Equation
- Optimal Policies and Optimal Value Functions
- Model-Based: Dynamic Programming
- Policy Evaluation
- Policy Improvement
- Policy Iteration
- Value Iteration
- Generalized Policy Iteration(GPI)

**Abstract**

MDP过程是RL环境中常见的范式，DP是解决有限MDP问题的可最优收敛办法，效率在有效平方级。DP算法基本思想是基于贝尔曼方程进行Bootstrapping，即用估计来学习估计（learn a guess from a guess)。DP需要经过反复的**策略评估**和**策略提升**过程，最终收敛到最优的策略和值函数。这一过程其实是RL很多算法的基本过程，即先进行评估策略（Prediction）再优化策略。

# MDP problems set up

在**RL problems set up**中我们知道RL基本要素是Agent和Enviornment, 环境的种类很多，但大多都可以抽象成一个马尔科夫决策过程（MDP）或者部分马尔科夫决策过程(POMDP);

MDPs are a mathematically idealized form of the reinforcement learning problem for which precise theoretical statements can be made.

**Key elements** of MDP：$<\mathcal{S},\mathcal{A},\mathcal{P},\mathcal{R},\gamma>$

| 名称 | 表达式 |
| --- | --- |
| 状态转移矩阵（一个Markov Matrix） | $P^a_{ss'} = P(S_{t+1} = s' \| S_t = s, A_t = a)$ |
| 奖励函数 | $R^a_s = \mathbb{E}_\pi[R_{t+1} \| S_t = s, A_t = a]$ |
| 累计奖励 | $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+1+k}$ |
| 值函数（Value Function） | $V_\pi(a) = \mathbb{E}[G_t \| S_t = s]$ |
| 动作值函数（Action Value Fucntion） | $Q_\pi(s, a) = \mathbb{E}[G_t \| S_t = s, A_t = a]$ |
| 策略（Policy） | $\pi(a \| s) = \mathbb{P}(A_t = a \| S_t = s)$ |
| 奖励转移方程 | $R_{t+1} = R_{t+1}(S_t, A_t, S_{t+1})$ |
| 某策略下的状态转移方程 | $P^\pi_{ss'} = \mathbb{P}(S_{t+1} = s' \| S_t = s) = \sum_a \pi(a \| s) P^a_{ss'}$ |
| 某状态某策略下的奖励函数 | $R^\pi_s = \sum_a \pi(a \| s) R^a_s$ |

# Bellman Equation

贝尔曼方程将某时刻的值函数与其下一时刻的值函数联系起来：$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$

$G_t = \sum\limits_{k=t+1}^{T} \gamma^{k-t-1} R_k = R_{t+1} + \gamma G_{t+1}$ 对于**动作-值函数来说**：

$$\begin{aligned} Q_\pi(s, a) &= \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma V_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= R^a_s + \gamma \sum_{s'} P^a_{ss'} V_\pi(s') \end{aligned}$$

对于**值函数来说**

$$\begin{aligned} V_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] \\ &= \sum_a \pi(a|s) Q_\pi(s, a) \\ &= \sum_a \pi(a|s) \{R^a_s + \gamma \sum_{s'} P^a_{ss'} V_\pi(s')\} \\ &= \sum_a \pi(a|s) R^\pi_s + \gamma \sum_{s'} P^\pi_{ss'} V_\pi(s') \quad (2) \\ &= \sum_a \pi(a|s) \sum_{s'} P^a_{ss'} \{R^a_{ss'} + \gamma V_\pi(s')\} \quad (3) \end{aligned}$$

基于公式2可以写成矩阵的形式：$V^\pi = R^\pi_s + P^\pi V^\pi$

# Optimal Policies and Optimal Value Functions

最优策略和最优的值函数关系如下：$v_* = \max_\pi V_\pi(s)$ $Q_*(s,a) = \max_\pi Q_pi(s,a)$ $V_*(s) = \max_a Q_*(s,a)$

Find optimal policy by:

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \arg\min_a Q_*(s,a) \\ 0 & \text{Otherwise} \end{cases}.$$

在最优策略下的贝尔曼方程为Bellman Optimality Equation:

$$\begin{aligned} Q_*(s,a) &= R_s^a + \gamma \sum_{s'} P_{ss'}^a V_*(s') \\ &= R_s^a + \gamma \sum_{s'} P_{ss'}^a \max_{a'} Q_*(s',a'). \end{aligned}$$

$$V_*(s) = \max_a \{R_s^a + \gamma \sum_{s'} P_{ss'}^a V_*(s')\}.$$

# Dynamic Programming

DP算法要求MDP的全部信息完全可知，依据Bellman Optimality Equation为基本思想进行策略迭代出最优结果；

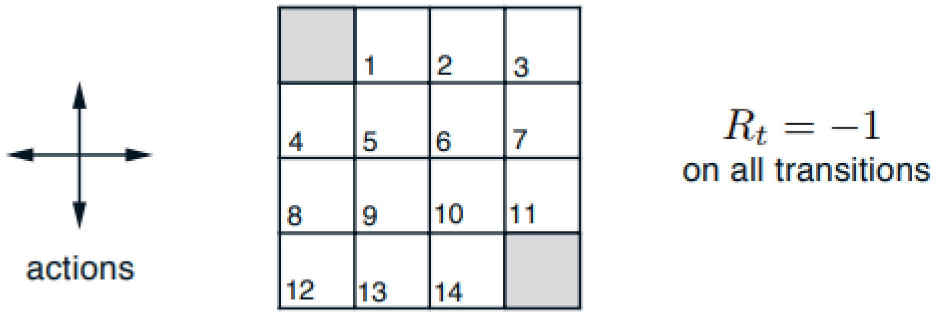
# Policy Evaluation

策略评估是在给定一个策略的情况下计算出Value Function的过程，迭代的更新规则是:

$v_{k+1}(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_k(S_{t+1})|S_t = s] = \sum_a \pi(a|s) \sum_{s'} P_{ss'}^a (R_{ss'} + \gamma v_k(s'))$ 迭代一定次数后$v_\pi$会趋于稳定，评估结束。

## Example: Gridword

用此例子来说明DP的一些算法




探索出发点到终点的路径，动作空间是四个方向， 除了终点其他坐标的Reward均为-1

```python
import gym
import gym_gridworlds
import numpy as np

# 策略评估算法
def policy_val(policy, env, GAMMA=1.0, theta=0.0001):
    V = np.zeros(env.observation_space.n)
    while True:
        delta = 0
        for s in range(env.observation_space.n):
            v = 0
            for a, action_prob in enumerate(policy[s]):
                for next_state, next_prob in enumerate(env.P[a,s]):
                    reward = env.R[a, next_state]
                    v += action_prob*next_prob*(reward + GAMMA*V[next_state])
            delta = max(delta, V[s]-v)
            V[s] = v
        if delta < theta:
            break
    return np.array(V)
```

```python
env = gym.make('Gridworld-v0')
# 初始随机策略
random_policy = np.ones((env.observation_space.n, env.action_space.n)) / env.action_space.n
# 评估这个随机策略得到稳定的Value Function:
policy_val(random_policy, env)
```

```
array([  0.        , -12.99934883, -18.99906386, -20.9989696 ,
       -12.99934883, -16.99920093, -18.99913239, -18.99914232,
       -18.99906386, -18.99913239, -16.9992679 , -12.9994534 ,
       -20.9989696 , -18.99914232, -12.9994534 ])
```

# Policy Improvement

定义：The process of making a new policy that improves on an original policy, by making it **greedy** with respect to the value function of the original policy, is called policy improvement.

## policy improvement theorem

假设$\pi$和$\pi'$是两个确定的策略，对于所有的$s \in \mathcal{S}$ 如果 $q_\pi(s, \pi'(s)) \geq v_\pi(s)$ 那么可以证明得到： $v'_\pi(s) \geq v_\pi(s)$

所以策略提升的过程就是：$\pi \leftarrow \text{Greedy}(V_\pi)$.

# Policy Iteration

Policy Iteration = Policy Evaluation + Policy Improvement

给定策略--评估策略得到各个V(s)--greedy提升出新的策略--评估新策略--直到策略不发生变化

**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 $s \in \mathcal{S}$:
           $v \leftarrow V(s)$
           $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s))\big[r + \gamma V(s')\big]$
           $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   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* $\leftarrow \pi(s)$
       $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a)\big[r + \gamma V(s')\big]$
       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

```python
def one_step_lookahead(state, V, GAMMA):
        A = np.zeros(env.action_space.n)
        for a in range(env.action_space.n):
            for next_state, prob in enumerate(env.P[a, state]):
                reward = env.R[a, next_state]
                A[a] += prob * (reward + GAMMA*V[next_state])
        return A

def policy_improvement(env, policy_eval_fun=policy_val, GAMMA=1.0):

    # 用随机策略开始
    policy = np.ones((env.observation_space.n, env.action_space.n)) / env.action_space.n

    while True:
        V = policy_eval_fun(policy, env, GAMMA)
        policy_stable = True
        for s in range(env.observation_space.n):
            chosen_a = np.argmax(policy[s])
            action_values = one_step_lookahead(s, V, GAMMA)
            best_a = np.argmax(action_values)

            # 贪心的方式更新策略
            if chosen_a != best_a:
                policy_stable = False
            policy[s] = np.eye(env.action_space.n)[best_a]

        if policy_stable:
            return policy, V
```

```python
# 得到稳定的策略和V(s)
policy_improvement(env)
```

```
(array([[1., 0., 0., 0.],
        [0., 0., 0., 1.],
        [0., 0., 0., 1.],
        [0., 0., 1., 0.],
        [1., 0., 0., 0.],
        [1., 0., 0., 0.],
        [1., 0., 0., 0.],
        [0., 0., 1., 0.],
        [1., 0., 0., 0.],
        [1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.]]),
 array([ 0.,  0., -1., -2.,  0., -1., -2., -1., -1., -2., -1.,  0., -2.,
        -1.,  0.]))
```

# Value Iteration

跟Policy Iteration的区别在于省略Policy Evaluation的过程为一步计算，这样降低了迭代次数，同时保证收敛结果依然为$v_*$，边评估边提升。

省略后的Evaluation：$v_{k+1}(s)=\max_a \sum_s{'} P_{ss'}^a(R_{s'}^a+\gamma v_k(s'))$ 而之前的评估策略迭代更多：

$$v_{k+1}(s)=\sum_a \pi(a|s)\sum_{s'} P_{ss'}^a(R_{ss'}+\gamma v_k(s'))$$

```python
def value_iteration(env, theta=0.001, GAMMA=1.0):
    V = np.zeros(env.observation_space.n)
    while True:
        delta = 0
        for s in range(env.observation_space.n):
            A = one_step_lookahead(s, V, GAMMA)
            best_action_value = np.max(A)
            delta = max(delta, np.abs(best_action_value - V[s]))
            V[s] = best_action_value
        if delta < theta:
            break
    policy = np.zeros((env.observation_space.n, env.action_space.n))
    for s in range(env.observation_space.n):
        A = one_step_lookahead(s, V, GAMMA)
        best_action = np.argmax(A)
        policy[s, best_action] = 1.0
    return policy, V

value_iteration(env)
```

```
(array([[1., 0., 0., 0.],
        [0., 0., 0., 1.],
        [0., 0., 0., 1.],
        [0., 0., 1., 0.],
        [1., 0., 0., 0.],
        [1., 0., 0., 0.],
        [1., 0., 0., 0.],
        [0., 0., 1., 0.],
        [1., 0., 0., 0.],
        [1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 0., 1., 0.],
        [1., 0., 0., 0.],
        [0., 1., 0., 0.],
        [0., 1., 0., 0.]]),
 array([ 0.,  0., -1., -2.,  0., -1., -2., -1., -1., -2., -1.,  0., -2.,
        -1.,  0.]))
```

可以看出同一个环境，Value Iteration的结果和之前Policy Iteration的结果相同；

# Generalized Policy Iteration(GPI)

GPI的思想将会贯彻RL始终，任何RL算法的过程都可以看作是一个GPI的过程。GPI描述了policy evaluation和improvement不断交互提升的过程；评估和提升的方法是多样的。