# Reinforcement Learning: A Tutorial

Xiaoshan Yu
yxsleo@gmail.com
Date: 12/20/2024

with special thanks to https://hrl.boyuai.com

# Outline

强化学习概念 (Introduction)

多臂老虎机 (MAB)

马尔可夫决策过程 (MDP)

时序差分算法 (TD)

- Sarsa算法
- Q-learning

深度Q网络 (DQN)

- DQN
- Double DQN
- Dueling DQN

策略梯度 (PG)

演员-批评家 (AC)

- TRPO
- PPO

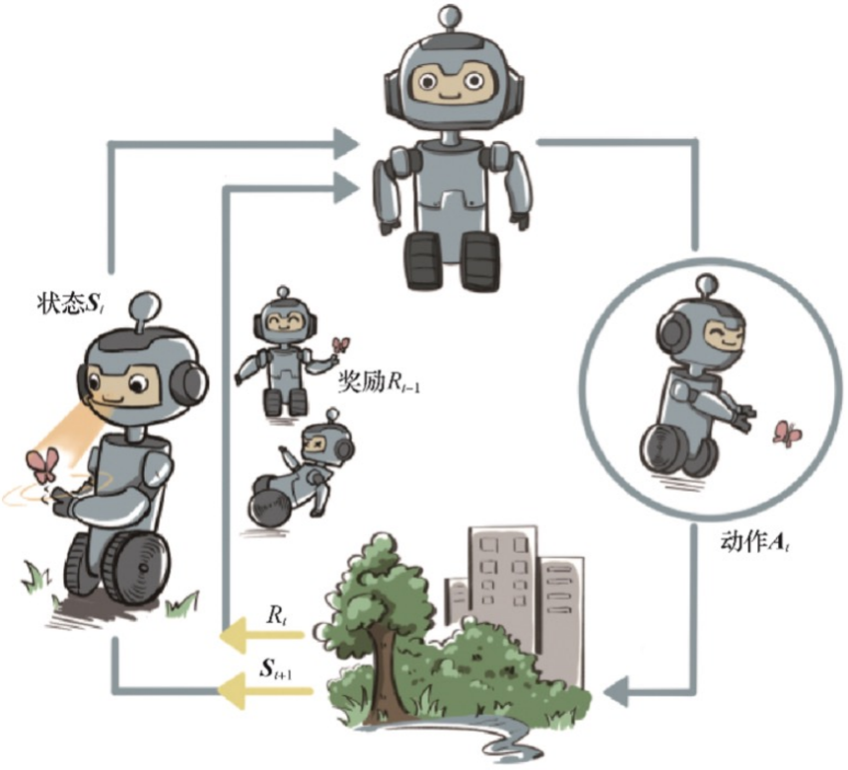
深度确定性策略梯度 (DDPG)

Policy-based Methods

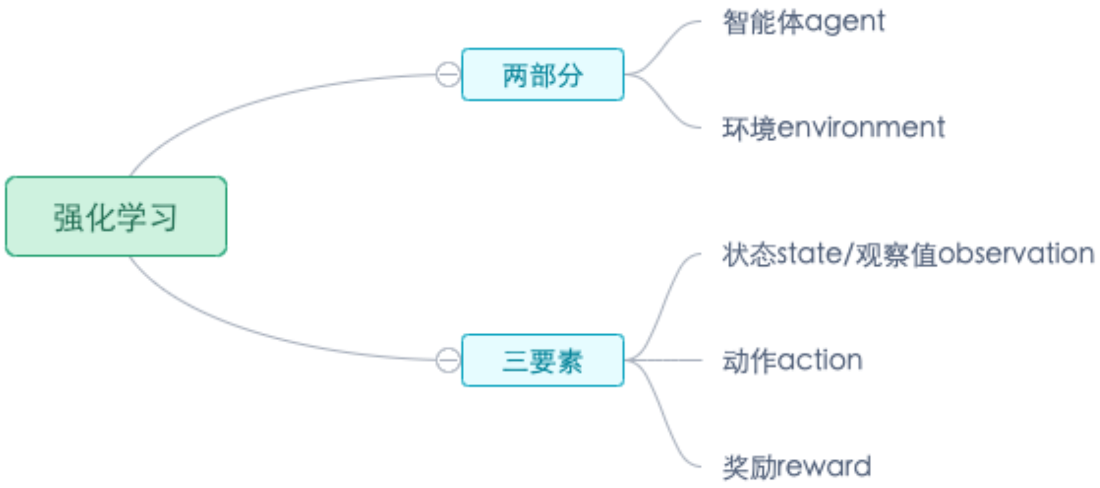
Value-based Methods

## ■ Reinforcement Learning

- 强化学习是机器通过与环境交互来实现目标的一种计算方法




✓ 迭代式交互




✓ 元素

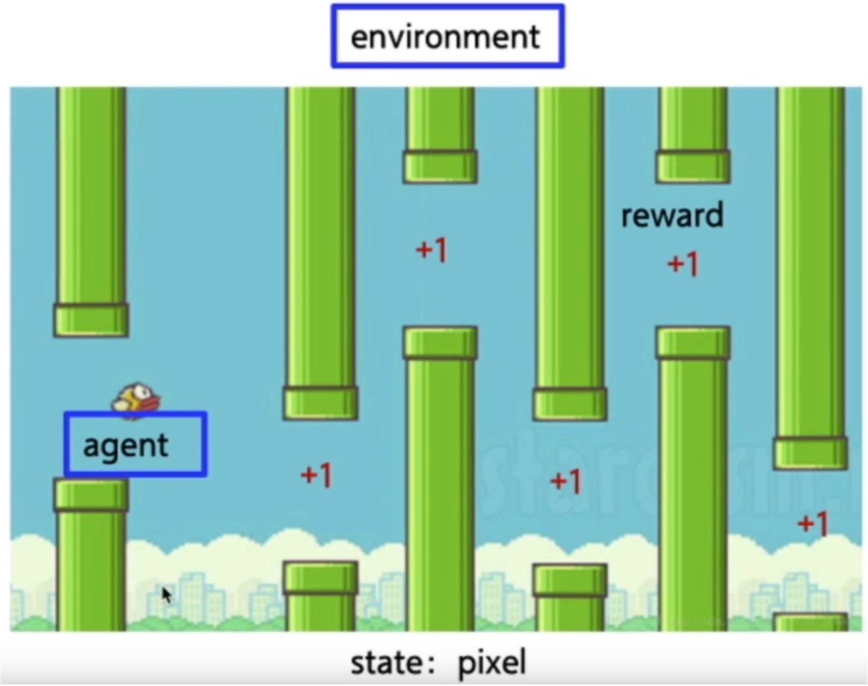
✓ 目标

$$J(\pi)=\mathbb{E}_\pi\left[\sum_{t=0}^{\infty}\gamma^t r_t\right]$$

回报(return)：累积奖励

策略(policy)：智能体的行为准则

## ■ Example




| 名称 | 对应上图中的内容 |
|---|---|
| agent | 鸟 |
| environment | 鸟周围的环境，水管、天空（包括小鸟本身） |
| state | 拍个照（目前的像素） |
| action | 向上向下动作 |
| reward | 距离（越远奖励越高） |

✓ Flappy Bird 示例

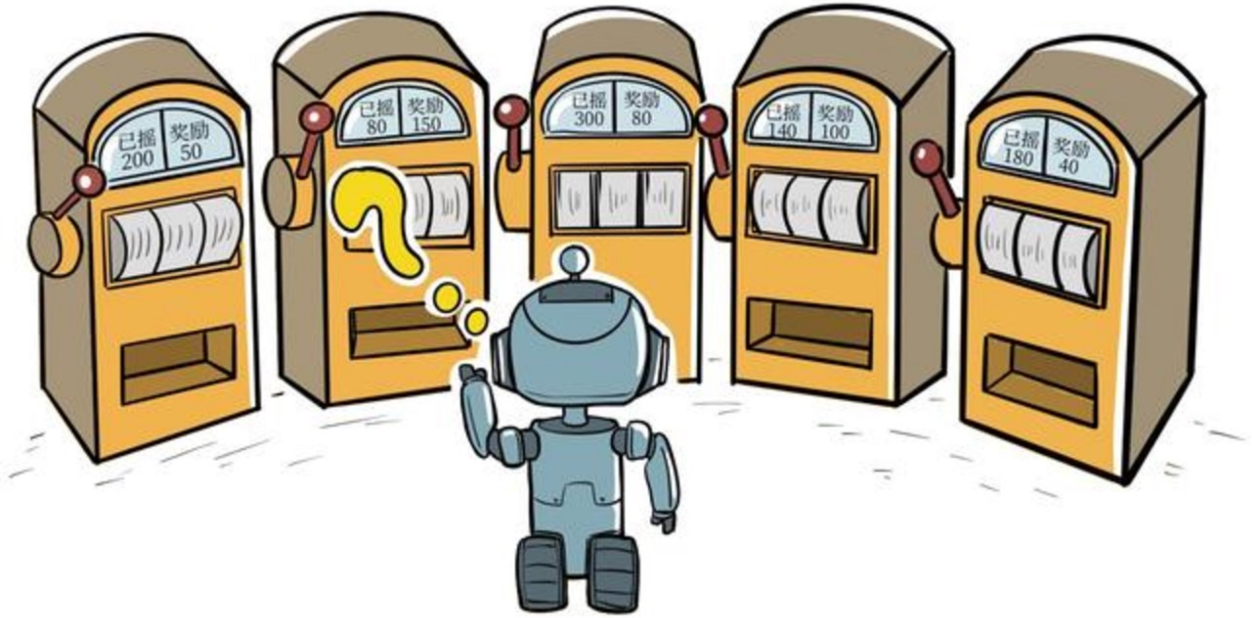
## ■ Characteristics

- 序列决策（Sequential Decision Making）
  - 强化学习是一个序列决策问题，其中当前的行动不仅影响即时奖励，还会影响未来状态和奖励。
- 试错学习（Trial and Error）
  - 强化学习一般没有直接的指导信息，Agent需要以不断与Environment进行交互，通过试错的方式来学习最佳策略。
- 探索与利用（Exploration vs. Exploitation）
  - 强化学习需要在探索（exploring）和利用（exploiting）之间取得平衡。Agent既需要探索新的行动策略，以发现更好的回报路径，又需要利用已经学到的知识来最大化当前的回报。

## ■ 多臂老虎机（Multi-Armed Bandit, MAB）

✓ 多臂老虎机问题可以表示为一个元组 $\langle \mathcal{A}, \mathcal{R} \rangle$，其中：

- $\mathcal{A}$ 为动作集合，其中一个动作表示拉动一个拉杆。若多臂老虎机一共有$K$根拉杆，那动作空间就是集合 $\{a_1, \ldots, a_K\}$，我们用$a_t \in \mathcal{A}$ 表示任意一个动作；
- $\mathcal{R}$ 为奖励概率分布，拉动每一根拉杆的动作$a$都对应一个奖励概率分布$\mathcal{R}(r|a)$，不同拉杆的奖励分布通常是不同的。




✓ 优化目标：最大化一段时间步T内累积的奖励

$$\max \sum_{t=1}^{T} r_t,\ r_t \sim \mathcal{R}(\cdot|a_t)$$

## ■ 累积懊悔（Cumulative Regret）

- ✓ 懊悔（Regret）
  - ➢ 每个动作a的期望奖励：$Q(a) = \mathbb{E}_{r \sim \mathcal{R}(\cdot|a)}[r]$
  - ➢ 最优期望奖励：$Q^* = \max_{a \in \mathcal{A}} Q(a)$
  - ➢ 懊悔：$R(a) = Q^* - Q(a)$
- ✓ 累积懊悔：操作T次拉杆后累积的懊悔总量

$$\sigma_R = \sum_{t=1}^{T} R(a_t)$$

MAB问题的目标为最大化累积奖励，等价于最小化累积懊悔！

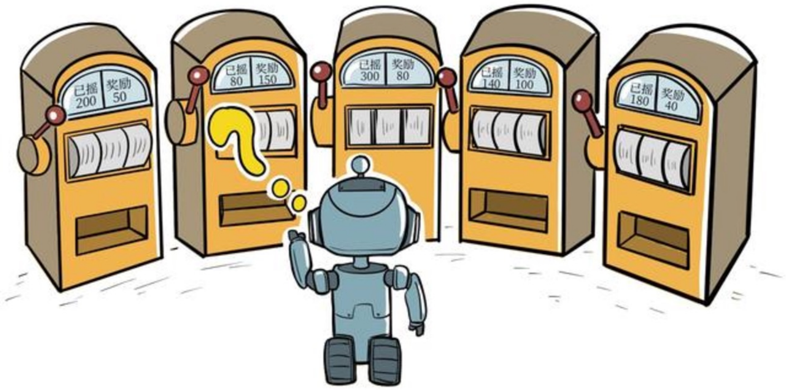
# MAB



## ■ 估计期望奖励

### ✓ 算法流程

- 对于$\forall a \in \mathcal{A}$，初始化计数器 $N(a)=0$和期望奖励估值 $\hat{Q}(a)=0$
- **for** $t=1 \rightarrow T$ **do**
- 选取某根拉杆，该动作记为 $a_t$
- 得到奖励 $r_t$
- 更新计数器: $N(a_t)=N(a_t)+1$
- 更新期望奖励估值：$\hat{Q}(a_t)=\hat{Q}(a_t)+\frac{1}{N(a_t)}\left[r_t-\hat{Q}(a_t)\right]$
- **end for**




$$
\begin{aligned}
Q_k &= \frac{1}{k}\sum_{i=1}^{k} r_i \\
&= \frac{1}{k}\left(r_k+\sum_{i=1}^{k-1} r_i\right) \\
&= \frac{1}{k}\left(r_k+(k-1)Q_{k-1}\right) \\
&= \frac{1}{k}\left(r_k+kQ_{k-1}-Q_{k-1}\right) \\
&= \boxed{Q_{k-1}+\frac{1}{k}\left[r_k-Q_{k-1}\right]}
\end{aligned}
$$

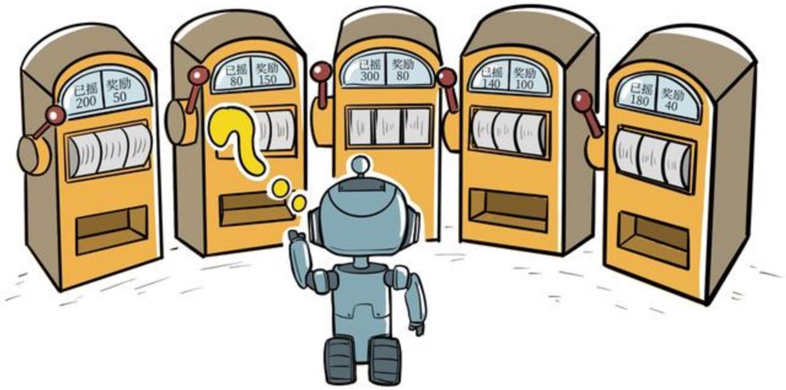
✓ 增量式（递归）更新（O(1)）

## ■ 探索vs.利用

- 对于$\forall a \in \mathcal{A}$，初始化计数器 $N(a)=0$和期望奖励估值 $\hat{Q}(a)=0$
- **for** $t=1 \rightarrow T$ **do**
- 选取某根拉杆，该动作记为 $a_t$
- 得到奖励 $r_t$
- 更新计数器: $N(a_t)=N(a_t)+1$
- 更新期望奖励估值：$\hat{Q}(a_t)=\hat{Q}(a_t)+\frac{1}{N(a_t)}\left[r_t-\hat{Q}(a_t)\right]$
- **end for**

```python
class BernoulliBandit:
    """ 伯努利多臂老虎机,输入K表示拉杆个数 """
    def __init__(self, K):
        self.probs = np.random.uniform(size=K)  # 随机生成K个0~1的数,作为拉动每根拉杆的获奖
        # 概率
        self.best_idx = np.argmax(self.probs)  # 获奖概率最大的拉杆
        self.best_prob = self.probs[self.best_idx]  # 最大的获奖概率
        self.K = K

    def step(self, k):
        # 当玩家选择了k号拉杆后,根据拉动该老虎机的k号拉杆获得奖励的概率返回1（获奖）或0（未
        # 获奖）
        if np.random.rand() < self.probs[k]:
            return 1
        else:
            return 0
```



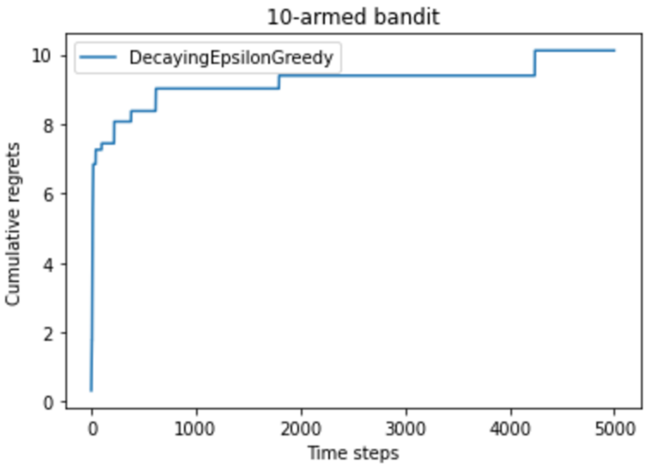

✓ **探索**：尝试拉动更多可能的拉杆，这根拉杆不一定会获得最大的奖励，但这种方案能够摸清楚所有拉杆的获奖情况。

✓ **利用**：拉动已知期望奖励最大的那根拉杆，由于已知的信息仅仅来自有限次的交互观测，所以当前的最优拉杆不一定是全局最优的。

## ■ 采样策略

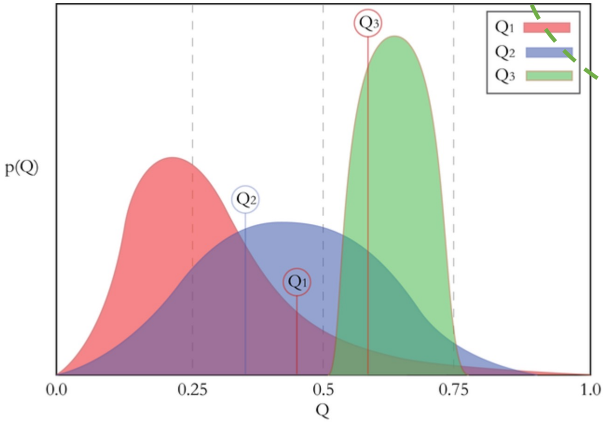
✓ $\epsilon$ −Greedy算法

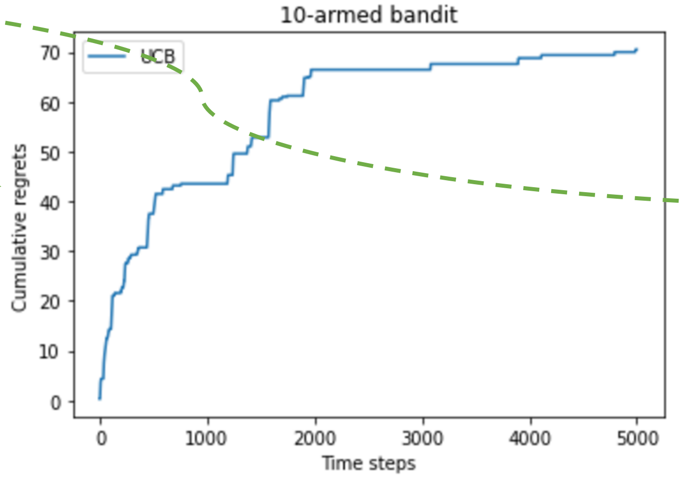
$$a_t = \begin{cases} \arg\max_{a\in\mathcal{A}} \hat{Q}(a), & \text{采样概率: } 1\text{-}\epsilon \\ \text{从 } \mathcal{A} \text{ 中随机选择}, & \text{采样概率: } \epsilon \end{cases}$$

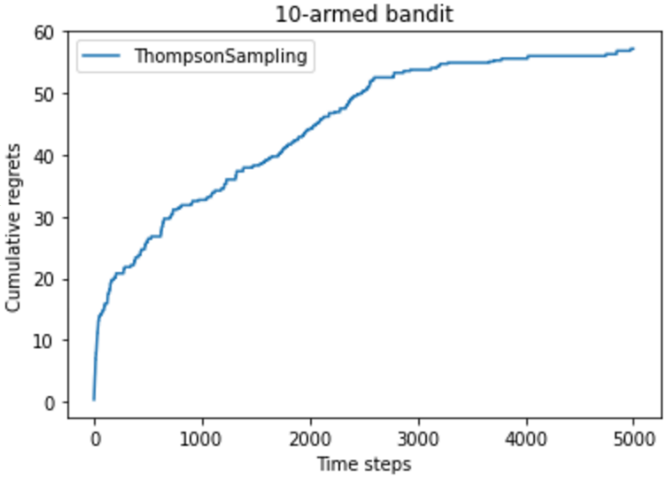



✓ 上置信界算法

✓ 汤普森采样算法



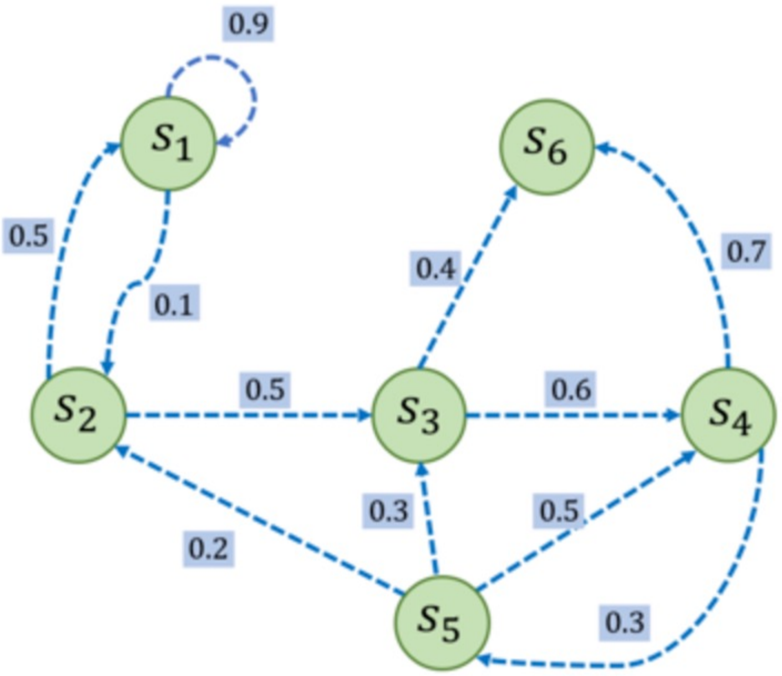



## ■ 马尔可夫过程（Markov Process, MP）

- **随机过程**（Stochastic Process）
  - ➢ 随机过程指一个系统状态随时间变化受随机因素影响的数学模型。我们将已知历史信息$(S_1, \ldots, S_t)$时下一个时刻状态为$S_{t+1}$的概率表示为$P(S_{t+1}|S_1, \ldots, S_t)$
- **马尔可夫性质**（Markov Property）
  - ➢ 当前仅当某时刻的状态只取决于上一时刻的状态时，一个随机过程被称为具有马尔可夫系性质，即$P(S_{t+1}|S_t) = P(S_{t+1}|S_1, \ldots, S_t)$
- **马尔可夫过程**（Markov Process）
  - ➢ 马尔可夫过程指具有马尔可夫性质的随机过程，也被称为马尔可夫链（Markov Chain）

■ 马尔可夫过程（Markov Process, MP）

- 通常使用元组$<S,P>$描述一个马尔可夫过程，其中$S$是有限数量的状态集合$\mathcal{S}=\{s_1,s_2,\ldots,s_n\}$，$P$ 是状态转移矩阵（state transition）。




✓ 马尔可夫过程示例

$$\mathcal{P}=\begin{bmatrix}P(s_1|s_1) & \cdots & P(s_n|s_1)\\ \vdots & \ddots & \vdots \\ P(s_1|s_n) & \cdots & P(s_n|s_n)\end{bmatrix}$$

$$\mathcal{P}=\begin{bmatrix}0.9 & 0.1 & 0 & 0 & 0 & 0\\ 0.5 & 0 & 0.5 & 0 & 0 & 0\\ 0 & 0 & 0 & 0.6 & 0 & 0.4\\ 0 & 0 & 0 & 0 & 0.3 & 0.7\\ 0 & 0.2 & 0.3 & 0.5 & 0 & 0\\ 0 & 0 & 0 & 0 & 0 & 1\end{bmatrix}$$

✓ 给定一个马尔可夫过程，我们就可以从某个状态出发，根据它的状态转移矩阵生成一个状态序列（episode），这个步骤也被叫做采样（sampling）。

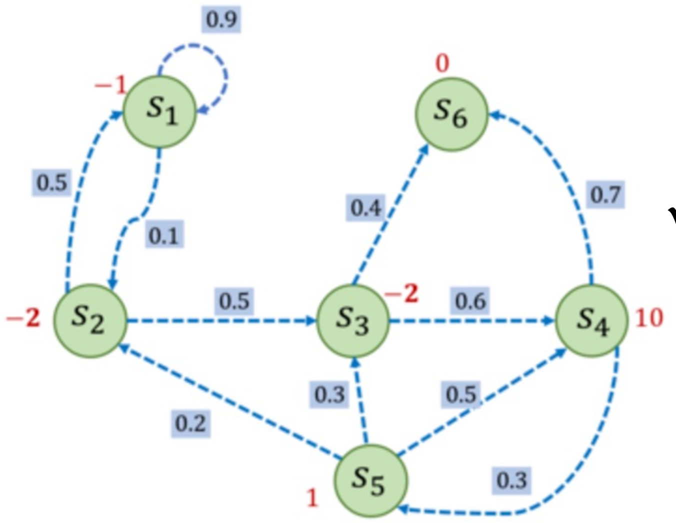
# MDP



## ■ 马尔可夫奖励过程（Markov Reward Process, MRP）

- 在马尔可夫过程的基础上加入奖励函数和折扣因子，就可以得到马尔可夫奖励过程。其由$<S,P,r,\gamma>$构成。

  ✓ $S$：有限状态的集合，
  ✓ $P$：状态转移矩阵
  ✓ $r$：奖励函数
  ✓ $\gamma$：折扣因子，$\in [0, 1)$




✓ 马尔可夫奖励过程示例

- 回报（Return）

  ✓ 在一个马尔可夫奖励过程中，从第t时刻状态$S_t$开始，直到终止状态时，所有奖励的衰减之和称为回报$G_t$

$$G_t = R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$$

# MDP

## ■ 马尔可夫奖励过程（Markov Reward Process, MRP）

### ● 价值函数（Value Function）

✓ 在马尔可夫奖励过程中，一个状态的期望回报（即从这个状态出发的未来累积奖励的期望）被称为这个状态的价值（value），所有状态的价值就构成了价值函数（value function）。其输入为状态，输出为该状态的价值。

$$
\begin{aligned}
V(s) &= \mathbb{E}[G_t|S_t = s] \\
&= \mathbb{E}[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots |S_t = s] \\
&= \mathbb{E}[R_t + \gamma (R_{t+1} + \gamma R_{t+2} + \ldots)|S_t = s] \\
&= \mathbb{E}[R_t + \gamma G_{t+1}|S_t = s] \\
&= \mathbb{E}[R_t + \gamma V(S_{t+1})|S_t = s]
\end{aligned}
$$

$$
V(s) = r(s) + \gamma \sum_{s' \in S} p(s'|s) V(s')
$$

贝尔曼方程

(Bellman equation)

矩阵化

$$
\mathcal{V} = \mathcal{R} + \gamma \mathcal{P} \mathcal{V}
$$

$$
\begin{bmatrix} V(s_1) \\ V(s_2) \\ \ldots \\ V(s_n) \end{bmatrix} = \begin{bmatrix} r(s_1) \\ r(s_2) \\ \ldots \\ r(s_n) \end{bmatrix} + \gamma \begin{bmatrix} P(s_1|s_1) & p(s_2|s_1) & \ldots & P(s_n|s_1) \\ P(s_1|s_2) & P(s_2|s_2) & \ldots & P(s_n|s_2) \\ \ldots & & & \\ P(s_1|s_n) & P(s_2|s_n) & \ldots & P(s_n|s_n) \end{bmatrix} \begin{bmatrix} V(s_1) \\ V(s_2) \\ \ldots \\ V(s_n) \end{bmatrix}
$$

解析解

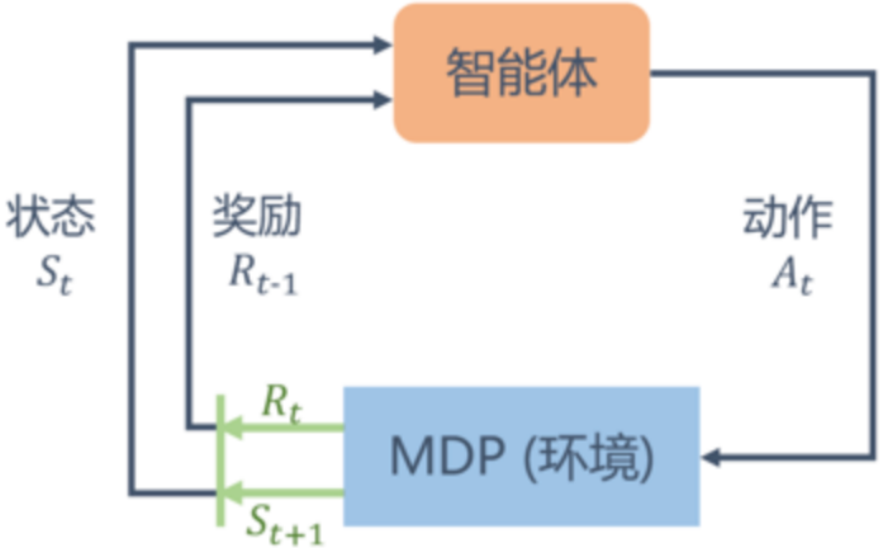
$$
\begin{aligned}
\mathcal{V} &= \mathcal{R} + \gamma \mathcal{P} \mathcal{V} \\
(I - \gamma \mathcal{P}) \mathcal{V} &= \mathcal{R} \\
\mathcal{V} &= (I - \gamma \mathcal{P})^{-1} \mathcal{R}
\end{aligned}
$$

## ■ 马尔可夫决策过程（Markov Decision Process, MDP）

- 前面的马尔可夫过程和马尔可夫奖励过程都是自发改变的随机过程，而如果有一个外界的“刺激”来共同改变这个过程，就有了马尔可夫决策过程。这里的“刺激”就是智能体（Agent）的动作。

- 因此，在MRP的基础上加入动作，就得到了MDP。其由$< S, A, P, r, \gamma >$构成。

✓ $S$：状态的集合，
✓ $A$：动作的集合，
✓ $P(s'|s,a)$：状态转移函数
✓ $r(s,a)$：奖励函数
✓ $\gamma$：折扣因子，$\in [0, 1)$




✓ 智能体与MDP环境交互

■ 马尔可夫决策过程（Markov Decision Process, MDP）

- 策略（Policy）

$$\pi(a|s) = P(A_t = a|S_t = s)$$

  ✓ 确定性策略（deterministic policy）

  ✓ 随机性策略（stochastic policy）

- 状态价值函数（State-Value Function）

  ✓ 从状态s出发遵循策略π能获得的期望回报

$$V^{\pi}(s) = \mathbb{E}_{\pi}[G_t|S_t = s]$$

■ 马尔可夫决策过程（Markov Decision Process, MDP）

● 动作价值函数（Action-Value Function）

✓ 遵循策略$\pi$，对当前状态s执行动作a获得的期望回报

$$Q^\pi(s,a)=\mathbb{E}_\pi[G_t|S_t=s,A_t=a]$$

贝尔曼期望方程

非常重要！

✓ 状态价值与动作价值的关系

$$V^\pi(s)=\sum_{a\in A}\pi(a|s)Q^\pi(s,a)$$

$$Q^\pi(s,a)=r(s,a)+\gamma\sum_{s'\in S}P(s'|s,a)V^\pi(s')$$

● 贝尔曼期望（Bellman Expectation Equation）

$$\begin{aligned}V^\pi(s)&=\mathbb{E}_\pi[R_t+\gamma V^\pi(S_{t+1})|S_t=s]\\&=\sum_{a\in A}\pi(a|s)\left(r(s,a)+\gamma\sum_{s'\in S}p(s'|s,a)V^\pi(s')\right)\end{aligned}$$

$$\begin{aligned}Q^\pi(s,a)&=\mathbb{E}_\pi[R_t+\gamma Q^\pi(S_{t+1},A_{t+1})|S_t=s,A_t=a]\\&=r(s,a)+\gamma\sum_{s'\in S}p(s'|s,a)\sum_{a'\in A}\pi(a'|s')Q^\pi(s',a')\end{aligned}$$

■ 蒙特卡洛方法（Monte-Carlo Methods）

- 蒙特卡洛方法（ Monte-Carlo Methods ）
  ✓ 也被称为统计模拟方法，是一种基于概率统计的数值计算方法，从抽样结果中归纳出要求的目标的数值估计。

- 估计状态价值函数

$$V^\pi(s)=\mathbb{E}_\pi[G_t|S_t=s]\approx\frac{1}{N}\sum_{i=1}^{N}G_t^{(i)}$$

$$\frac{\text{圆的面积}}{\text{正方形的面积}}=\frac{\text{圆中点的个数}}{\text{正方形中点的个数}}$$

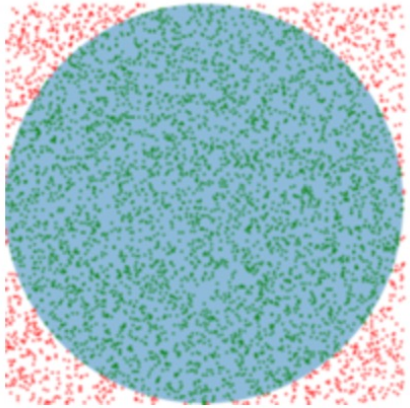


图3-5 用蒙特卡洛方法估计圆的面积

(1) 使用策略$\pi$采样若干条序列：

$$s_0^{(i)}\xrightarrow{a_0^{(i)}}r_0^{(i)},s_1^{(i)}\xrightarrow{a_1^{(i)}}r_1^{(i)},s_2^{(i)}\xrightarrow{a_2^{(i)}}\cdots\xrightarrow{a_{T-1}^{(i)}}r_{T-1}^{(i)},s_T^{(i)}$$

(2) 对每一条序列中的每一时间步$t$的状态$s$进行以下操作：

- 更新状态$s$的计数器 $N(s)\leftarrow N(s)+1$;
- 更新状态$s$的总回报 $M(s)\leftarrow M(s)+G_t$;

(3) 每一个状态的价值被估计为回报的平均值 $V(s)=M(s)/N(s)$。

增量式更新

- $N(s)\leftarrow N(s)+1$
- $V(s)\leftarrow V(s)+\frac{1}{N(s)}(G-V(S))$

## ■ 时序差分算法（Temporal Difference, TD）

- 无模型强化学习（ Model-Free RL）

  ✓ 指无需显式学习环境的动态模型（如无显式状态转移概率），仅通过与环境交互直接优化策略或价值函数的强化学习方法

- 时序差分方法（Temporal Difference）

$$
\begin{aligned}
V_\pi(s) &= \mathbb{E}_\pi[G_t|S_t=s] \\
&= \mathbb{E}_\pi[\sum_{k=0}^{\infty}\gamma^k R_{t+k}|S_t=s] \\
&= \mathbb{E}_\pi[R_t+\gamma\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t=s] \\
&= \mathbb{E}_\pi[R_t+\gamma V_\pi(S_{t+1})|S_t=s]
\end{aligned}
$$

$$V(s_t) \leftarrow V(s_t) + \alpha[G_t - V(s_t)]$$

$$V(s_t) \leftarrow V(s_t) + \alpha\underbrace{[r_t + \gamma V(s_{t+1}) - V(s_t)]}_{\text{时序差分误差（TD Error）}}$$

■ 时序差分算法（Temporal Difference, TD）

● Sarsa算法

✓ 直接用时序差分算法来估计动作价值函数

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha[r_t + \gamma Q(s_{t+1},a_{t+1}) - Q(s_t,a_t)]$$

✓ 采样策略

$$\pi(a|s) = \begin{cases} \epsilon/|\mathcal{A}| + 1 - \epsilon & \text{如果} a = \arg\max_{a'} Q(s,a') \\ \epsilon/|\mathcal{A}| & \text{其他动作} \end{cases}$$

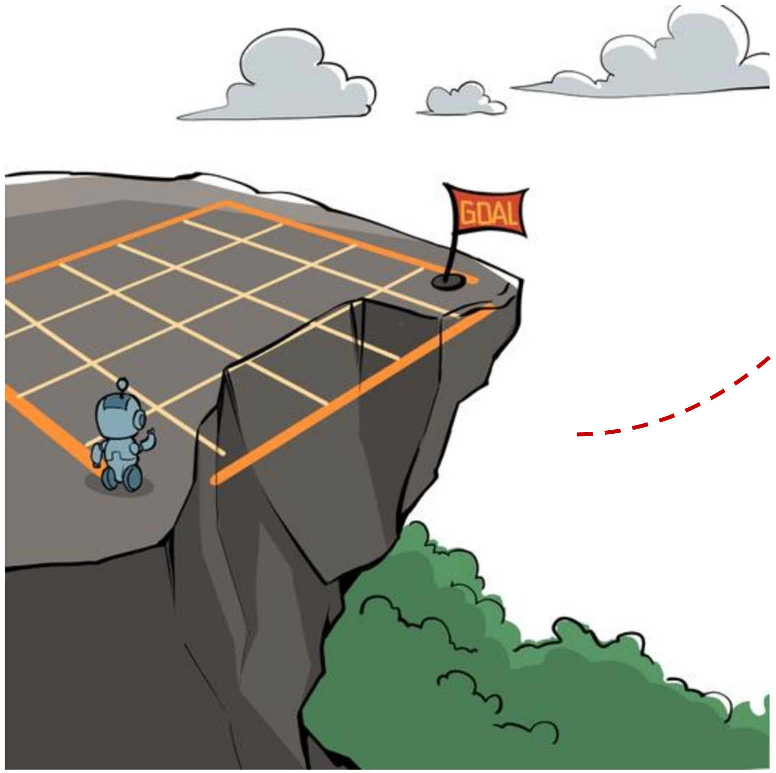
✓ 算法流程

- 初始化$Q(s,a)$
- **for** 序列$e = 1 \to E$ **do**:
- 　得到初始状态$s$
- 　用$\epsilon$-greedy 策略根据$Q$选择当前状态$s$下的动作$a$
- 　**for** 时间步$t = 1 \to T$ **do**：
- 　　得到环境反馈的$r$, $s'$
- 　　用$\epsilon$-greedy 策略根据$Q$选择当前状态$s'$下的动作$a'$
- 　　$Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
- 　　$s \leftarrow s', a \leftarrow a'$
- 　**end for**
- **end for**

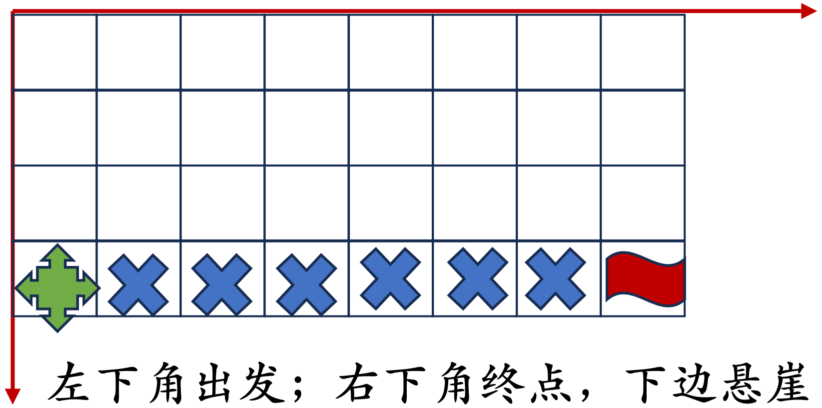
# TD

■ 时序差分算法（Temporal Difference, TD）

- Sarsa算法 [示例]



Cliff Walking



左下角出发；右下角终点，下边悬崖

```python
def step(self, action):  # 外部调用这个函数来改变当前位置
    # 4种动作, change[0]:上, change[1]:下, change[2]:左, change[3]:右。坐标系原点(0,0)
    # 定义在左上角
    change = [[0, -1], [0, 1], [-1, 0], [1, 0]]
    self.x = min(self.ncol - 1, max(0, self.x + change[action][0]))
    self.y = min(self.nrow - 1, max(0, self.y + change[action][1]))
    next_state = self.y * self.ncol + self.x
    reward = -1
    done = False
    if self.y == self.nrow - 1 and self.x > 0:  # 下一个位置在悬崖或者目标
        done = True
        if self.x != self.ncol - 1:
            reward = -100
    return next_state, reward, done
```

```python
""" Sarsa算法 """
def __init__(self, ncol, nrow, epsilon, alpha, gamma, n_action=4):
    self.Q_table = np.zeros([nrow * ncol, n_action])  # 初始化Q(s,a)表格
```

```python
def update(self, s0, a0, r, s1, a1):
    td_error = r + self.gamma * self.Q_table[s1, a1] - self.Q_table[s0, a0]
    self.Q_table[s0, a0] += self.alpha * td_error
```

■ 时序差分算法（Temporal Difference, TD）

● Q-Learning算法

✓ 时序差分的更新方式有所差别

Sarsa $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_t + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[R_t + \boxed{\gamma \max_a Q(s_{t+1}, a)} - Q(s_t, a_t)]$$

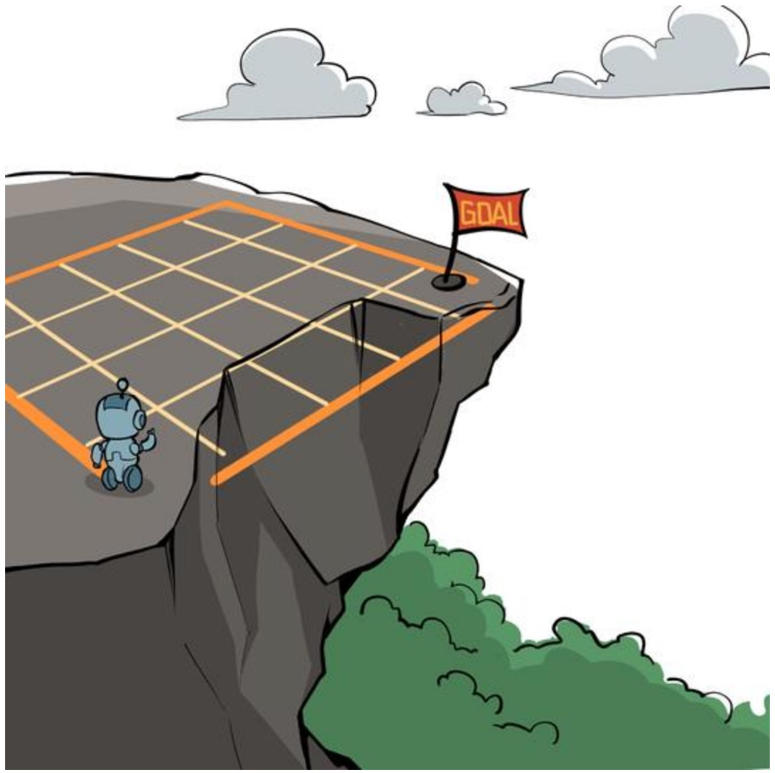
✓ 算法流程

- 初始化$Q(s, a)$
- **for** 序列$e = 1 \to E$ **do**:
- 得到初始状态$s$
- **for** 时间步$t = 1 \to T$ **do** :
- 用$\epsilon$-greedy 策略根据$Q$选择当前状态$s$下的动作$a$
- 得到环境反馈的$r, s'$
- $Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_{a'} Q(s', a') - Q(s, a)]$
- $s \leftarrow s'$
- **end for**
- **end for**

Anhui University

■ 时序差分算法（Temporal Difference, TD）

● Q-learning算法 [示例]




Cliff Walking

```
def take_action(self, state):  #选取下一步的操作
    if np.random.random() < self.epsilon:
        action = np.random.randint(self.n_action)
    else:
        action = np.argmax(self.Q_table[state])
    return action
```

当前动作a的采样

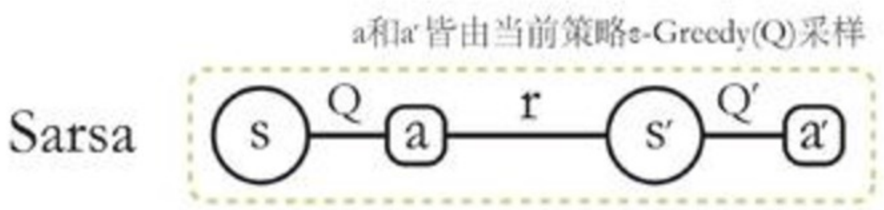
```
def update(self, s0, a0, r, s1):
    td_error = r + self.gamma * self.Q_table[s1].max(
    ) - self.Q_table[s0, a0]
    self.Q_table[s0, a0] += self.alpha * td_error
```

未来动作a'的采样

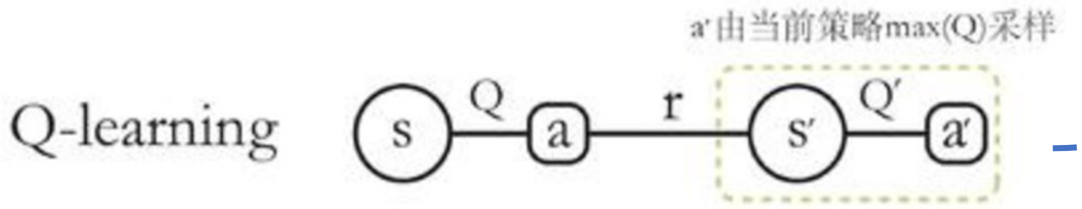
■ 时序差分算法（Temporal Difference, TD）

- 在线策略算法（On-Policy）&离线策略算法（Off-Policy）

✓ 采样数据的策略为行为策略（behavior policy）
✓ 使用数据更新的策略为目标策略（target policy）
✓ 两种策略一致的算法为在线策略，否则为离线策略




Sarsa为在线策略算法




Q-Learning为离线策略算法

■ 深度Q网络（DQN）

- Motivation

  ✓ Q-learning：维护一个存储每个状态下所有动作Q值的表格

  ✓ 这种方法要求状态和动作均为离散的，如何适应连续状态空间?

- Method

  ✓ DQN：使用函数拟合的方法来估计Q值，从而解决连续状态下离散动作的问题。

■ 深度Q网络（DQN）

● CartPole环境 [示例]

CartPole

表 7-1 CartPole环境的状态空间

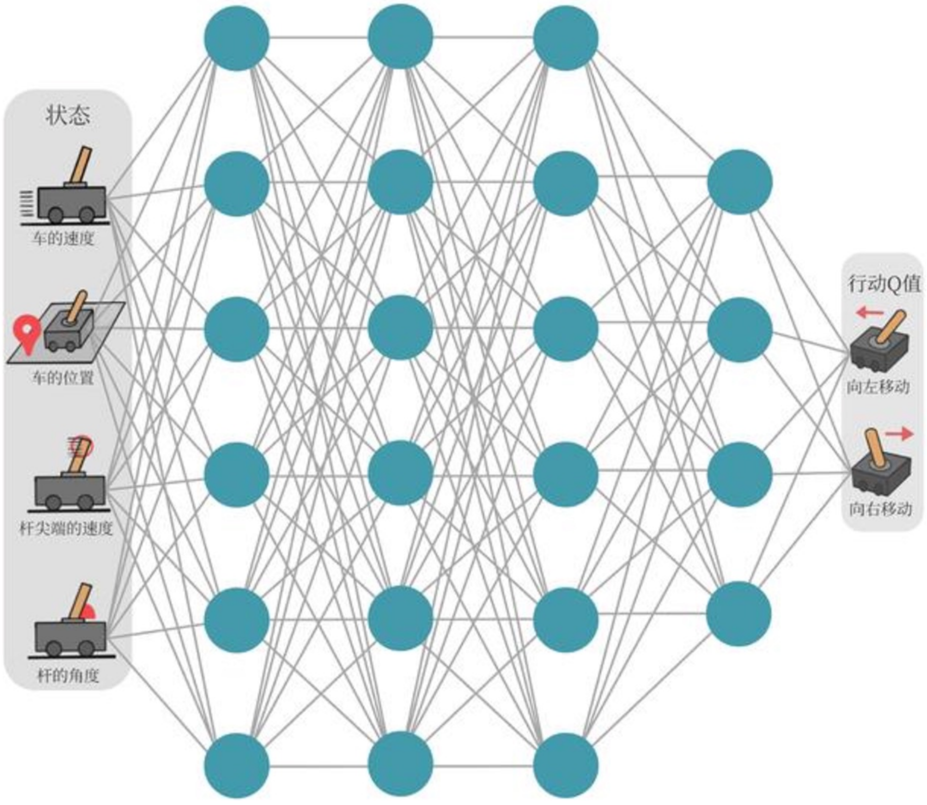
| 维度 | 意义 | 最小值 | 最大值 |
|---|---|---|---|
| 0 | 车的位置 | -2.4 | 2.4 |
| 1 | 车的速度 | -Inf | Inf |
| 2 | 杆的角度 | ~ -41.8° | ~ 41.8° |
| 3 | 杆尖端的速度 | -Inf | Inf |

表7-2 CartPole环境的动作空间

| 标号 | 动作 |
|---|---|
| 0 | 向左移动小车 |
| 1 | 向右移动小车 |

## ■ 深度Q网络（DQN）

### ● CartPole环境 [示例]




Q网络

### ● 优化目标

✓ 构建好Q网络，如何优化？即损失函数是什么？

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left[ r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a) \right]$$

$$V(s_t) \leftarrow V(s_t) + \alpha [G_t - V(s_t)]$$

$$\omega^* = \arg\min_{\omega} \frac{1}{2N} \sum_{i=1}^{N} \left[ Q_\omega(s_i, a_i) - \left( r_i + \gamma \max_{a'} Q_\omega(s_i', a') \right) \right]^2$$

均方误差损失函数

■ 深度Q网络（DQN）

- 经验放回（experience replay）

✓ 维护一个回放缓冲区，将每次从环境中采样的四元组数据（状态、动作、奖励、下一状态）存储，训练Q网络时再从缓冲区中采样batch_size个数据训练。
✓ 满足样本独立假设：打破样本相关性，有助网络训练
✓ 提高样本使用效率：每个样本多次使用，适合深度网络的训练与梯度学习

- 目标网络

✓ 为了训练的稳定性，使用目标网络来计算TD目标项，然后用Q网络来预测动作价值

目标网络

$$\frac{1}{2}\left[Q_\omega(s,a)-\left(r+\gamma\max_{a'}Q_{\omega^-}(s',a')\right)\right]^2$$

$$\omega^*=\arg\min_{\omega}\frac{1}{2N}\sum_{i=1}^{N}\left[Q_\omega(s_i,a_i)-\left(r_i+\gamma\max_{a'}Q_\omega(s_i',a')\right)\right]^2$$

✓ 同网络不同参数，按C步频率同步

## ■ 深度Q网络（DQN）

### ● 算法流程

- 用随机的网络参数$\omega$初始化网络$Q_\omega(s,a)$
- 复制相同的参数$\omega^- \leftarrow \omega$来初始化目标网络$Q_{\omega'}$
- 初始化经验回放池$R$
- **for** 序列$e=1\rightarrow E$ **do**
-     获取环境初始状态$s_1$
-     **for** 时间步$t=1\rightarrow T$ **do**
-         根据当前网络$Q_\omega(s,a)$以$\epsilon$-贪婪策略选择动作$a_t$
-         执行动作$a_t$，获得回报$r_t$，环境状态变为$s_{t+1}$
-         将$(s_t,a_t,r_t,s_{t+1})$存储进回放池$R$中
-         若$R$中数据足够，从$R$中采样$N$个数据$\{(s_i,a_i,r_i,s_{i+1})\}_{i=1,\dots,N}$
-         对每个数据，用目标网络计算$y_i=r_i+\gamma\max_a Q_{\omega^-}(s_{i+1},a)$
-         最小化目标损失$L=\frac{1}{N}\sum_i(y_i-\boxed{Q_\omega(s_i,a_i)})^2$，以此更新当前网络$Q_\omega$
-         更新目标网络
-     **end for**
- **end for**

```python
class Qnet(torch.nn.Module):
    ''' 只有一层隐藏层的Q网络 '''
    def __init__(self, state_dim, hidden_dim, action_dim):
        super(Qnet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))  # 隐藏层使用ReLU激活函数
        return self.fc2(x)
```

```python
self.q_net = Qnet(state_dim, hidden_dim,
                  self.action_dim).to(device)  # Q网络
# 目标网络
self.target_q_net = Qnet(state_dim, hidden_dim,
                         self.action_dim).to(device)
```

```python
q_values = self.q_net(states).gather(1, actions)  # Q值
# 下个状态的最大Q值
max_next_q_values = self.target_q_net(next_states).max(1)[0].view(
    -1, 1)
q_targets = rewards + self.gamma * max_next_q_values * (1 - dones
                                                        )  # TD误差目标
dqn_loss = torch.mean(F.mse_loss(q_values, q_targets))  # 均方误差损失函数
```

## DQN改进算法

- Double DQN
  - ✓ 解决DQN中常见的Q值过高估计问题

$$r+\gamma \max_{a'} Q_{\omega^-}(s',a')$$

Q网络选取下一状态最佳动作

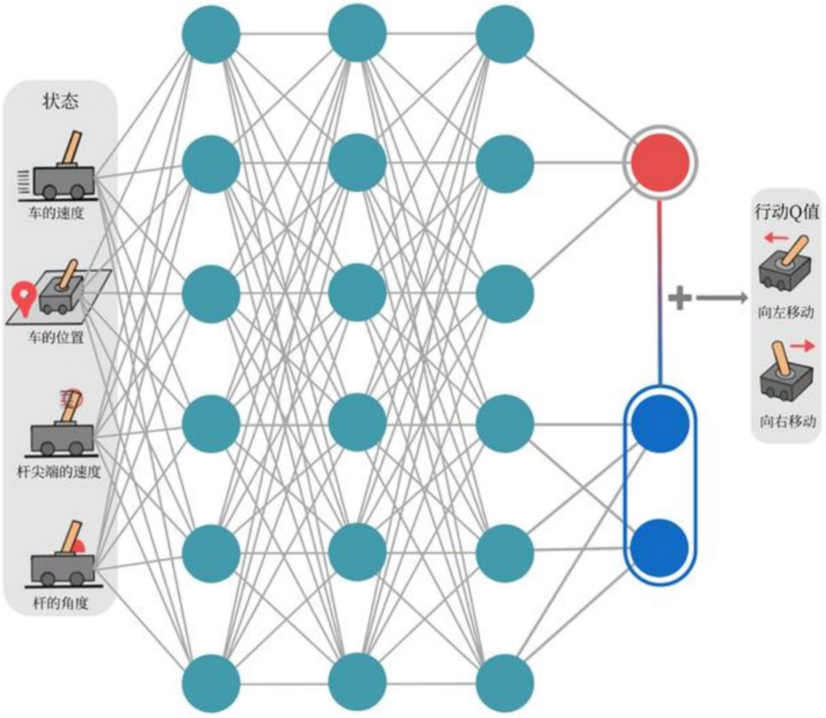
$$Q_{\omega^-}\left(s', \arg\max_{a'} Q_{\omega^-}(s',a')\right)$$

$$r+\gamma Q_{\omega^-}\left(s', \arg\max_{a'} Q_{\omega}(s',a')\right)$$

- Dueling DQN
  - ✓ 使用优势函数A建模Q网络。

$$A(s,a)=Q(s,a)-V(s)$$

$$Q_{\eta,\alpha,\beta}(s,a)=V_{\eta,\alpha}(s)+A_{\eta,\beta}(s,a)$$



## ■ 策略梯度算法（Policy Gradient）

- Motivation
  - ✓ Q-learning：处理有限状态
  - ✓ DQNs：处理连续状态

  均Value-based，有没有方法能直接学习动作策略呢？

- 策略梯度
  - ✓ 使用线性模型或者神经网络模型对策略函数建模，输入状态，输出动作的概率分布（随机性策略）。目标函数：

$$\boxed{J(\theta) = \mathbb{E}_{s_0}[V^{\pi_\theta}(s_0)]}$$

$$\begin{aligned}
\nabla_\theta J(\theta) &\propto \sum_{s\in S} \nu^{\pi_\theta}(s) \sum_{a\in A} Q^{\pi_\theta}(s,a)\nabla_\theta \pi_\theta(a|s) \\
&= \sum_{s\in S} \nu^{\pi_\theta}(s) \sum_{a\in A} \pi_\theta(a|s) Q^{\pi_\theta}(s,a) \frac{\nabla_\theta \pi_\theta(a|s)}{\pi_\theta(a|s)} \\
&= \mathbb{E}_{\pi_\theta}[Q^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)]
\end{aligned}$$

策略梯度为在线策略算法

## ■ 策略梯度算法（Policy Gradient）

- REINFORCE
  - ✓ 利用蒙特卡洛方法估计动作价值函数

$$= \mathbb{E}_{\pi_\theta}[Q^{\pi_\theta}(s,a)\nabla_\theta \log \pi_\theta(a|s)]$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\sum_{t=0}^{T}\left(\sum_{t'=t}^{T}\gamma^{t'-t}r_{t'}\right)\nabla_\theta \log \pi_\theta(a_t|s_t)\right]$$

- 初始化策略参数$\theta$
- **for** 序列 $e = 1 \to E$ **do**：
- 用当前策略$\pi_\theta$采样轨迹$\{s_1, a_1, r_1, s_2, a_2, r_2, \ldots s_T, a_T, r_T\}$
- 计算当前轨迹每个时刻$t$往后的回报$\sum_{t'=t}^{T}\gamma^{t'-t}r_{t'}$，记为$\psi_t$
- 对$\theta$进行更新，$\theta = \theta + \alpha\sum_t^T \psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)$
- **end for**

```
G = 0
self.optimizer.zero_grad()
for i in reversed(range(len(reward_list))):  # 从最后一步算起
    reward = reward_list[i]
    state = torch.tensor([state_list[i]],
                         dtype=torch.float).to(self.device)
    action = torch.tensor([action_list[i]]).view(-1, 1).to(self.device)
    log_prob = torch.log(self.policy_net(state).gather(1, action))
    G = self.gamma * G + reward
    loss = -log_prob * G  # 每一步的损失函数
    loss.backward()  # 反向传播计算梯度
self.optimizer.step()  # 梯度下降
```

Anhui University

## ■ 演员-批评家算法（Actor-Critic）

● Actor-Critic

$$g = \mathbb{E}\left[\sum_{t=0}^{T} \psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)\right]$$

1. $\sum_{t'=0}^{T} \gamma^{t'} r_{t'}$：轨迹的总回报；
2. $\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$：动作$a_t$之后的回报；
3. $\sum_{t'=t}^{T} \gamma^{t'-t} r_{t'} - b(s_t)$：基准线版本的改进；
4. $Q^{\pi_\theta}(s_t, a_t)$：动作价值函数；
5. $A^{\pi_\theta}(s_t, a_t)$：优势函数；
6. $r_t + \gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)$：时序差分残差。

$\boxed{A(s,a) = Q(s,a) - V(s)}$

$\boxed{Q = r + \gamma V}$


✓ 时序差分error：1) critic估计状态价值，2) actor预测动作分布

# Actor-Critic



■ 演员-批评家算法（Actor-Critic）

- Actor-Critic
  - Critic如何更新?

$$\boxed{\mathcal{L}(\omega)=\frac{1}{2}(r+\gamma V_\omega(s_{t+1})-V_\omega(s_t))^2}$$

$$\boxed{\nabla_\omega\mathcal{L}(\omega)=-(r+\gamma V_\omega(s_{t+1})-V_\omega(s_t))\nabla_\omega V_\omega(s_t)}$$

- 初始化策略网络参数$\theta$，价值网络参数$\omega$
- **for** 序列 $e=1\rightarrow E$ **do**：
- 用当前策略$\pi_\theta$采样轨迹$\{s_1,a_1,r_1,s_2,a_2,r_2,\ldots\}$
- 为每一步数据计算: $\delta_t=r_t+\gamma V_\omega(s_{t+1})-V_\omega(s_t)$
- 更新价值参数$w=w+\alpha_\omega\sum_t\delta_t\nabla_\omega V_\omega(s_t)$
- 更新策略参数$\theta=\theta+\alpha_\theta\sum_t\delta_t\nabla_\theta\log\pi_\theta(a_t|s_t)$
- **end for**

```python
class PolicyNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim, action_dim):
        super(PolicyNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, action_dim)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return F.softmax(self.fc2(x), dim=1)
```

```python
class ValueNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim):
        super(ValueNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, 1)

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return self.fc2(x)
```

```python
# 时序差分目标
td_target = rewards + self.gamma * self.critic(next_states) * (1 -
                                                               dones)
td_delta = td_target - self.critic(states)  # 时序差分误差
log_probs = torch.log(self.actor(states).gather(1, actions))
actor_loss = torch.mean(-log_probs * td_delta.detach())
# 均方误差损失函数
critic_loss = torch.mean(
    F.mse_loss(self.critic(states), td_target.detach()))
```

■ AC改进方法：AC训练不稳定

- TRPO算法（信任区域策略优化）
  - ✓ 找到一块信任区域，确保策略更新的安全性

$$
\begin{aligned}
J(\theta) &= \mathbb{E}_{s_0}[V^{\pi_\theta}(s_0)] \\
&= \mathbb{E}_{\pi_{\theta'}}\left[\sum_{t=0}^{\infty}\gamma^t V^{\pi_\theta}(s_t) - \sum_{t=1}^{\infty}\gamma^t V^{\pi_\theta}(s_t)\right] \\
&= -\mathbb{E}_{\pi_{\theta'}}\left[\sum_{t=0}^{\infty}\gamma^t \left(\gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)\right)\right]
\end{aligned}
$$

$$
\begin{aligned}
J(\theta') - J(\theta) &= \mathbb{E}_{s_0}\left[V^{\pi_{\theta'}}(s_0)\right] - \mathbb{E}_{s_0}\left[V^{\pi_\theta}(s_0)\right] \\
&= \mathbb{E}_{\pi_{\theta'}}\left[\sum_{t=0}^{\infty}\gamma^t r(s_t, a_t)\right] + \mathbb{E}_{\pi_{\theta'}}\left[\sum_{t=0}^{\infty}\gamma^t \left(\gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)\right)\right] \\
&= \mathbb{E}_{\pi_{\theta'}}\left[\sum_{t=0}^{\infty}\gamma^t \left[r(s_t, a_t) + \gamma V^{\pi_\theta}(s_{t+1}) - V^{\pi_\theta}(s_t)\right]\right]
\end{aligned}
$$

$$
\begin{aligned}
&= \mathbb{E}_{\pi_{\theta'}}\left[\sum_{t=0}^{\infty}\gamma^t A^{\pi_\theta}(s_t, a_t)\right] \\
&= \sum_{t=0}^{\infty}\gamma^t \mathbb{E}_{s_t\sim P_t^{\pi_{\theta'}}}\mathbb{E}_{a_t\sim\pi_{\theta'}(\cdot|s_t)}\left[A^{\pi_\theta}(s_t, a_t)\right] \\
&= \frac{1}{1-\gamma}\mathbb{E}_{s\sim\nu^{\pi_{\theta'}}}\mathbb{E}_{a\sim\pi_{\theta'}(\cdot|s)}\left[A^{\pi_\theta}(s, a)\right]
\end{aligned}
$$

# Actor-Critic

## ■ AC改进方法：AC训练不稳定

● PPO算法（近似策略优化）

✓ 简化更新过程，简单有效

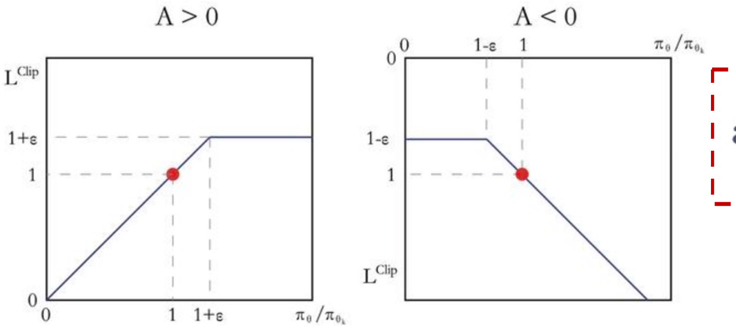
$$\max_{\theta} \quad \mathbb{E}_{s\sim\nu^{\pi_{\theta_k}}}\mathbb{E}_{a\sim\pi_{\theta_k}(\cdot|s)}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a)\right]$$
$$\text{s.t.} \quad \mathbb{E}_{s\sim\nu^{\pi_{\theta_k}}}[D_{KL}(\pi_{\theta_k}(\cdot|s),\pi_\theta(\cdot|s))] \le \delta$$

PPO-惩罚：拉格朗日乘数法将KL散度放进目标

$$\arg\max_{\theta}\mathbb{E}_{s\sim\nu^{\pi_{\theta_k}}}\mathbb{E}_{a\sim\pi_{\theta_k}(\cdot|s)}\left[\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a)-\beta D_{KL}[\pi_{\theta_k}(\cdot|s),\pi_\theta(\cdot|s)]\right]$$

PPO-截断：对目标函数进行限制，确保新-旧参数差距不会太大

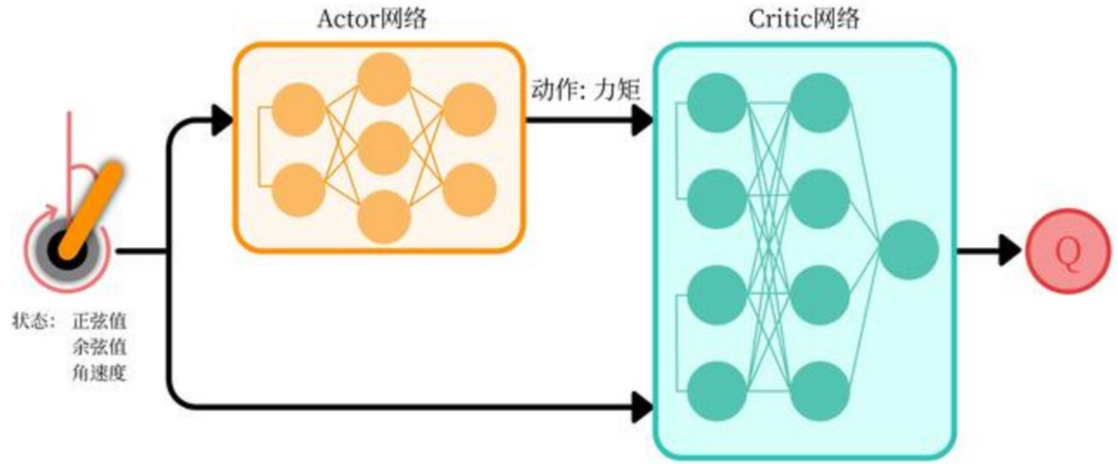
$$\arg\max_{\theta}\mathbb{E}_{s\sim\nu^{\pi_{\theta_k}}}\mathbb{E}_{a\sim\pi_{\theta_k}(\cdot|s)}\left[\min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a),\text{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)},1-\epsilon,1+\epsilon\right)A^{\pi_{\theta_k}}(s,a)\right)\right]$$



■ DDPG算法（Deep Deterministic Policy Gradient, DDPG）

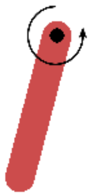
- Motivation
  - ✓ 基于策略梯度的算法REINFORCE、Actor-Critic（以及改进TRPO&PPO），这些方法均为在线策略算法，意味着样本效率较低。
  - ✓ DQN可以直接估计最优函数Q，做到离线策略学习，但是处理动作空间有限，对于连续动作空间问题处理粗糙。
- 确定性策略梯度：

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim\nu^{\pi_\beta}}\left[\nabla_\theta \mu_\theta(s) \nabla_a Q_\omega^\mu(s,a)|_{a=\mu_\theta(s)}\right]$$



■ DDPG算法（Deep Deterministic Policy Gradient, DDPG）

● Pendulum环境 [示例]



倒立摆

表8-1 Pendulum环境的状态空间

| 标号 | 名称 | 最小值 | 最大值 |
| --- | --- | --- | --- |
| 0 | $\cos\theta$ | -1.0 | 1.0 |
| 1 | $\sin\theta$ | -1.0 | 1.0 |
| 2 | $\dot{\theta}$ | -8.0 | 8.0 |

表8-2 Pendulum环境的动作空间

| 标号 | 动作 | 最小值 | 最大值 |
| --- | --- | --- | --- |
| 0 | 力矩 | -2.0 | 2.0 |

# DDPG

## ■ DDPG算法：类似DQN+AC

- 随机噪声可以用$\mathcal{N}$来表示，用随机的网络参数$\omega$和$\theta$分别初始化 Critic 网络$Q_\omega(s,a)$和 Actor 网络$\mu_\theta(s)$
- 复制相同的参数$\omega^- \leftarrow \omega$和$\theta^- \leftarrow \theta$，分别初始化目标网络$Q_{\omega^-}$和$\mu_{\theta^-}$
- 初始化经验回放池$R$
- **for** 序列$e = 1 \to E$ **do**：
- 初始化随机过程$\mathcal{N}$用于动作探索
- 获取环境初始状态$s_1$
- **for**时间步 $t = 1 \to T$ **do**：
- 根据当前策略和噪声选择动作$a_t = \mu_\theta(s_t) + \mathcal{N}$
- 执行动作$a_t$，获得奖励$r_t$，环境状态变为$s_{t+1}$
- 将$(s_t, a_t, r_t, s_{t+1})$存储进回放池$R$
- 从$R$中采样$N$个元组$\{(s_i, a_i, r_i, s_{i+1})\}_{i=1,\dots,N}$
- 对每个元组，用目标网络计算$y_i = r_i + \gamma Q_{\omega^-}(s_{i+1}, \mu_{\theta^-}(s_{i+1}))$
- 最小化目标损失$L = \frac{1}{N}\sum_{i=1}^{N}(y_i - Q_\omega(s_i, a_i))^2$，以此更新当前 Critic 网络
- 计算采样的策略梯度，以此更新当前 Actor 网络：

$$\nabla_\theta J \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \mu_\theta(s_i) \nabla_a Q_\omega(s_i, a)|_{a=\mu_\theta(s_i)}$$

- 更新目标网络：

$$\omega^- \leftarrow \tau\omega + (1-\tau)\omega^- \theta^- \leftarrow \tau\theta + (1-\tau)\theta^-$$

- **end for**
- **end for**

```python
class PolicyNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim, action_dim, action_bound):
        super(PolicyNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, action_dim)
        self.action_bound = action_bound  # action_bound是环境可以接受的动作最大值

    def forward(self, x):
        x = F.relu(self.fc1(x))
        return torch.tanh(self.fc2(x)) * self.action_bound
```

```python
class QValueNet(torch.nn.Module):
    def __init__(self, state_dim, hidden_dim, action_dim):
        super(QValueNet, self).__init__()
        self.fc1 = torch.nn.Linear(state_dim + action_dim, hidden_dim)
        self.fc2 = torch.nn.Linear(hidden_dim, hidden_dim)
        self.fc_out = torch.nn.Linear(hidden_dim, 1)

    def forward(self, x, a):
        cat = torch.cat([x, a], dim=1) # 拼接状态和动作
        x = F.relu(self.fc1(cat))
        x = F.relu(self.fc2(x))
        return self.fc_out(x)
```

```python
next_q_values = self.target_critic(next_states, self.target_actor(next_states))
q_targets = rewards + self.gamma * next_q_values * (1 - dones)
critic_loss = torch.mean(F.mse_loss(self.critic(states, actions), q_targets))
self.critic_optimizer.zero_grad()
critic_loss.backward()
self.critic_optimizer.step()

actor_loss = -torch.mean(self.critic(states, self.actor(states)))
self.actor_optimizer.zero_grad()
actor_loss.backward()
self.actor_optimizer.step()

self.soft_update(self.actor, self.target_actor)  # 软更新策略网络
self.soft_update(self.critic, self.target_critic)  # 软更新价值网络
```

# Conclusion



强化学习概念 (Introduction) → 多臂老虎机 (MAB) → 马尔可夫决策过程 (MDP) → 时序差分算法 (TD)

时序差分算法 (TD):
- Sarsa算法
- Q-learning

时序差分算法 (TD) → 深度Q网络 (DQN)

深度Q网络 (DQN):
- DQN
- Double DQN
- Dueling DQN

深度Q网络 (DQN) → 策略梯度 (PG) → 演员-批评家 (AC)

演员-批评家 (AC):
- TRPO
- PPO

Policy-based Methods: 演员-批评家 (AC) → 深度确定性策略梯度 (DDPG)

Value-based Methods: 深度Q网络 (DQN) → 深度确定性策略梯度 (DDPG)