Beijing Forest Studio
北京理工大学信息系统及安全对抗实验中心

强化学习基础与实战

博士研究生　门元昊

2022年03月27日

内容提要

- 背景简介
- 基本概念
- 算法原理
- 应用总结
- 参考文献

背景简介

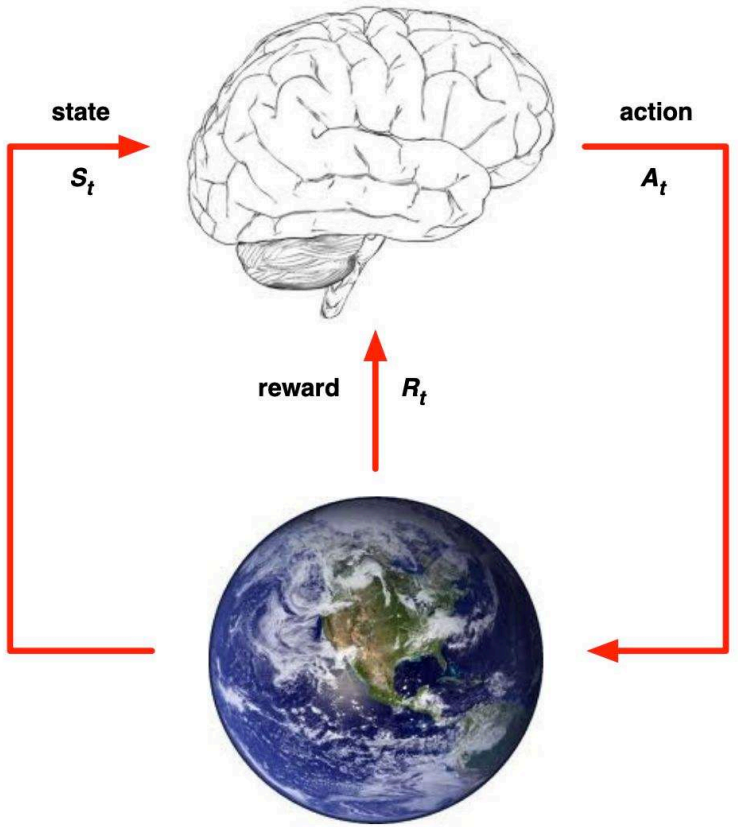
- 预期收获
  - 1. 了解强化学习领域的基础知识
  - 2. 理解Q-Learning、DQN算法原理
  - 3. 了解Q-Learning算法的实现方法

背景简介

- 选择苹果
  - 咬一口青苹果，很酸不好吃，得到知识“青苹果不好吃”
  - 咬一口红苹果，很甜好吃，得到知识“红苹果好吃”
  - 后续的选择中更加倾向于选择红苹果

基本概念

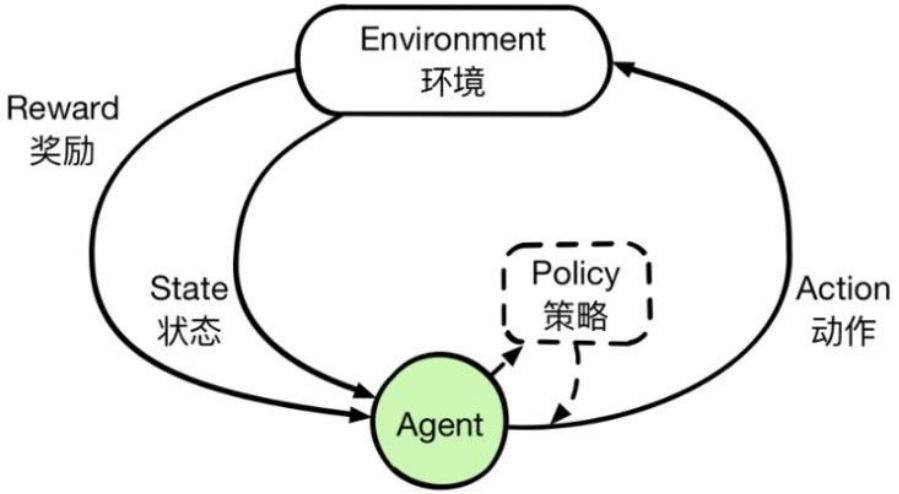
- 强化学习的概念
  - 解决智能体（代理）与环境的交互过程中，通过学习策略以达到回报最大化或实现特定目标的问题
  - 实质上是解决决策（decision-making）问题，能够进行连续自动决策



AlphaGo

基本概念

- 基本架构
  - 两个角色：代理、环境
  - 三个要素：动作A、状态S、奖励R
  - 代理依据策略（Policy）做出动作（Action）影响环境
  - 环境状态（State）改变并返回奖励（Reward）给代理

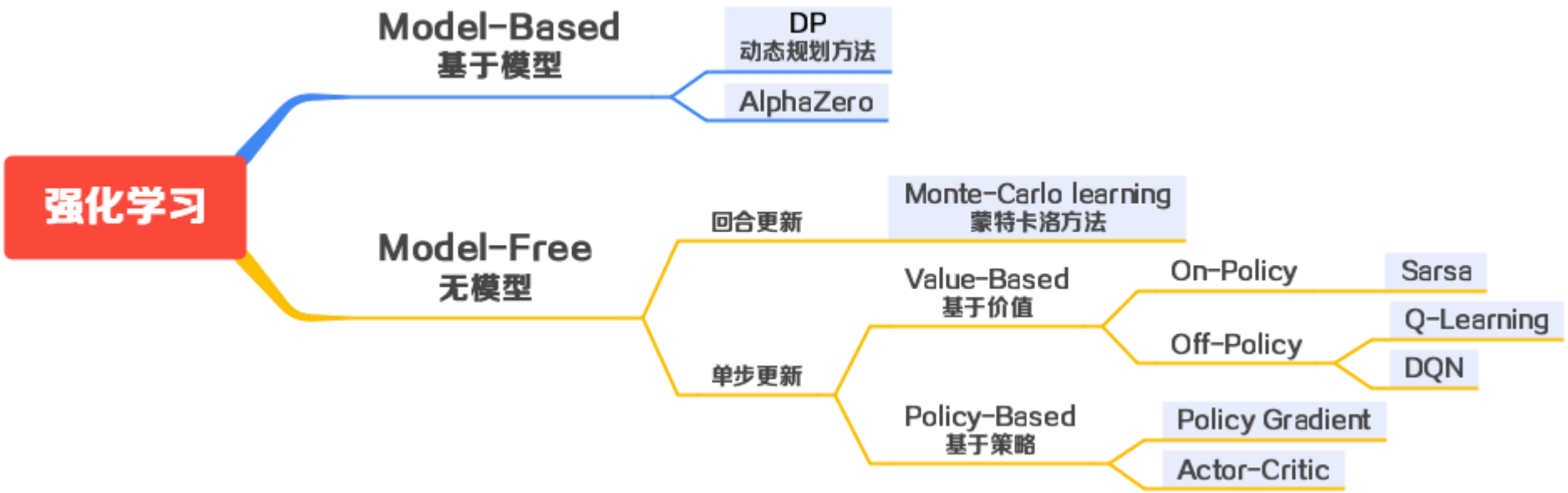


基本概念

- 其他概念
  - 策略
    - 代理采取动作的依据，通常表示为$\pi(a|s)$，表示在状态s时采取动作a的概率
    - $\pi(a|s) = P(A_t = a|S_t = s)$
  - 价值
    - 给定策略π和状态s时，代理行动后的价值（后续奖励的期望），表示为$v_\pi(s)$
    - $v_\pi = E_\pi(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots |S_t = s)$
  - 奖励衰减 γ
    - 用于调整当前延时奖励和后续奖励的权重
  - 状态转化模型
    - 环境状态变化的概率模型，$P^{\mathrm{a}}_{ss'}$，表示在状态s时采取动作a转到状态s'的概率

基本概念

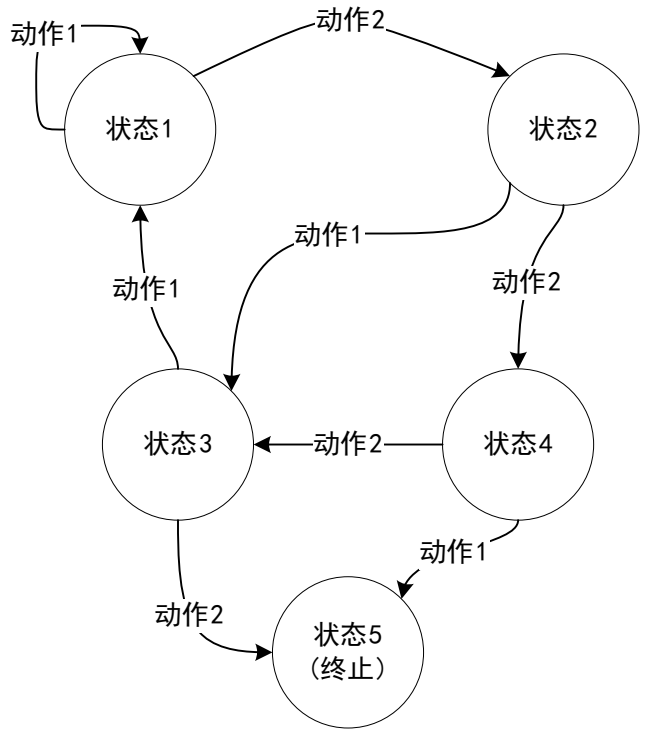
- 分类
  - Model-Based/Free
    - 代理能否理解环境，是否已知状态转移概率和奖励函数
  - Value/Policy-Based
    - 输出动作的价值或概率，据此选择动作



基本概念

- 马尔科夫决策过程（MDP）
  - 实际环境中，下一个状态s_{t+1}，既与当前状态s_t相关，也与之前的状态相关，难以进行环境的建模
  - 采用MDP模型，假定每个状态都仅与上一个状态有关，

    环境状态转化模型：$P_{ss'}^{a} = E(S_{t+1} = s' | S_t = s, A_t = a)$



价值函数：$v_\pi(s) = E_\pi(R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s)$

动作价值函数：$q_\pi(s, a) = E_\pi(R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1}) | S_t = s, A_t = a)$

Q-Learning

算法原理

T	训练代理在连续决策问题上做出最优决策
I	动作集、环境模型（状态集、奖励）、迭代轮数
P	初始化参数 For epoch =1, N: 1. 依据当前状态选择的动作 2. 执行动作，从环境中获取下一阶段状态 3. 计算奖励和价值 4. 更新策略参数（Q表）
O	能够进行最优决策的代理

P	DP方法依赖于问题模型，需要获得奖励和环境状态转换的知识才能求解
C	环境满足MDP假设，决策问题存在最优解
D	缺乏环境知识，仅能依赖奖励函数
L	经典方法

算法原理

- Q Learning算法
  - Value-Based类方法，其核心思想是利用环境状态和代理动作构建一张表格（Q-Table）用以存储Q值，依据Q值选取能获得最大收益的动作
  - Q值为当前状态和动作下的长期收益期望，即$Q_t = \Sigma_{k=0}^{\infty} \gamma^k R_{t+k+1}$

- 概念区分
  - 奖励：给定状态下的即时反馈
  - 价值：给定当前状态和代理策略下的长期收益期望
  - Q值：给定当前状态和当前动作下的长期收益期望

动作

状态

	ts_up	ts_down	tv_up	tv_down	adj_up	adj_down
0	4.717569	5.161536	5.120970	7.049681	5.795876	4.601859
1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2	1.006945	2.919058	2.822928	6.647163	2.013414	3.385360
3	6.353390	0.896469	0.799802	2.538798	2.140769	1.229461
4	0.730251	0.774865	0.906058	5.463744	0.812705	0.994933
...	...	...	...	...	...	...
69	0.000000	0.163728	0.000000	0.000000	1.406014	0.000000
70	0.000000	0.530771	0.059254	8.640511	0.584880	1.105284
71	-0.105249	9.300511	-0.147061	6.992806	1.506404	0.408057
72	0.321237	0.175072	-0.212824	0.009448	5.602252	0.000000
73	0.493080	0.313502	0.217867	0.353771	9.759383	1.097195

算法原理

- 算法流程
  - 训练过程

算法：Q-Learning

输入：环境状态集S，代理动作集A，奖励衰减指数γ，学习率α

1 随机初始化 Q 表$Q(s,a)$

2 初始化代理策略$\pi(a|s)$，$\forall s \in S, \forall a \in A, \pi(s|a) = \frac{1}{|A|}$

3 repeat

4 　初始化起始状态s

5 　repeat

6 　　依据当前状态s，选择动作a

7 　　执行动作a，得到奖励r和新状态s'

8 　　更新价值函数$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_i Q(s',i) - Q(s,a))$

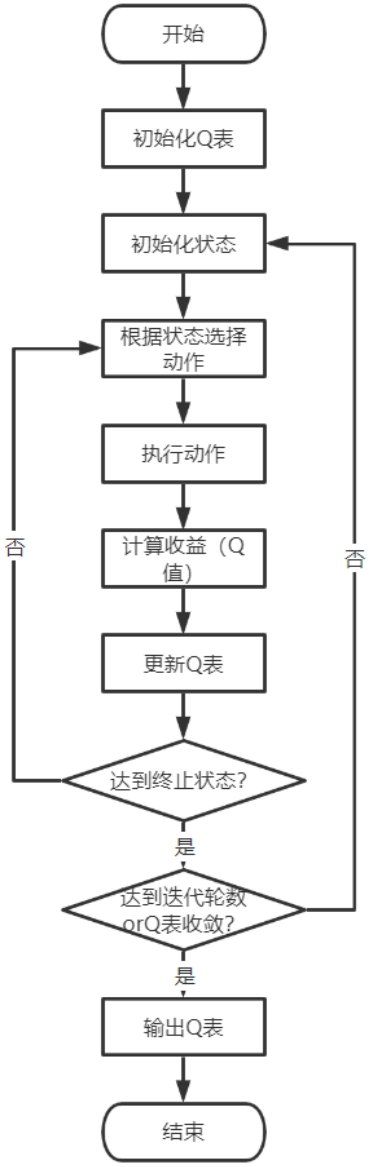
9 　　$s \leftarrow s'$

10 　until s为终止状态

11 until $\forall s \in S, \forall a \in A, Q(s,a)$收敛

输出：Q 表$Q(s,a)$

加入ε-greedy策略以保持探索能力



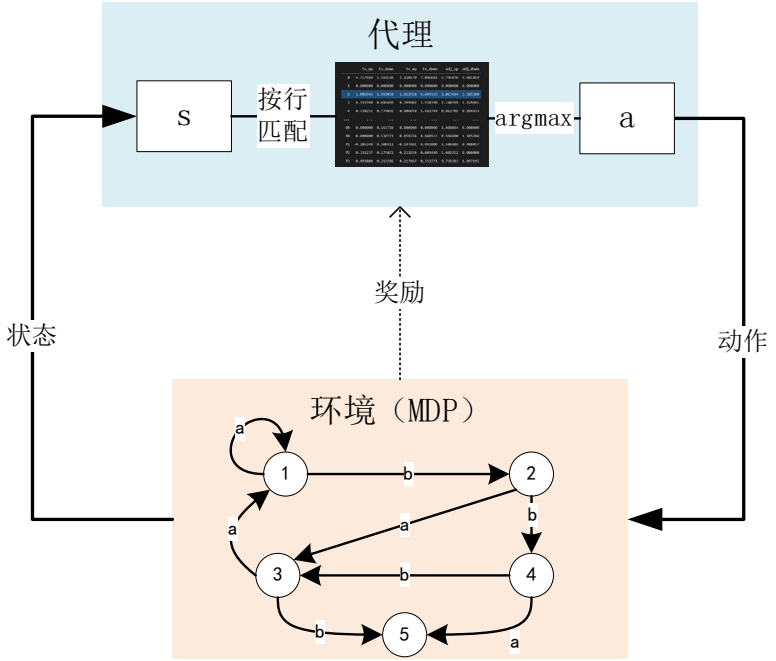
单步更新Q表： $Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_i Q(s',i) - Q(s,a))$

算法原理

- 算法流程
  - 应用过程
    - 1. 代理获取当前环境状态s
    - 2. 依据Q表和状态s，计算最优动作$a = argmax_a Q(s,a)$
    - 3. 重复上述步骤直至达到终止状态

```
if (q_state <= q_table_len)
    {
        memcpy_usr(q_reward, &(q_table[q_state].ts_up), (q_action_count * sizeof(q_reward[0U])));
        /*- 获取Q表中最大权重对应的操作 */
        for (i = 0U; i < q_action_count; i++)
        {
            if (max_reward < q_reward[i])
            {
                max_reward = q_reward[i];
                best_action = i;
            }
            else
            {
                /*- do nothing */
            }
        }
    }
```

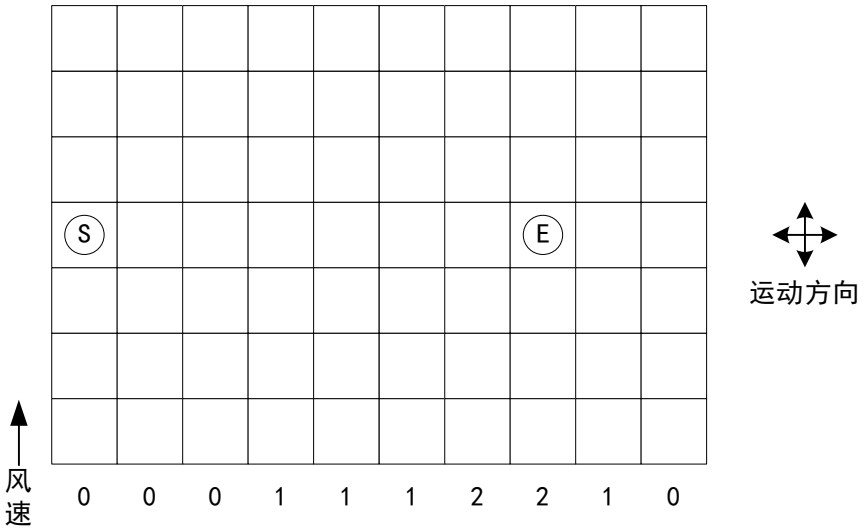
Q表应用过程的本质就是查阅
由状态和动作索引的二维数组



算法原理

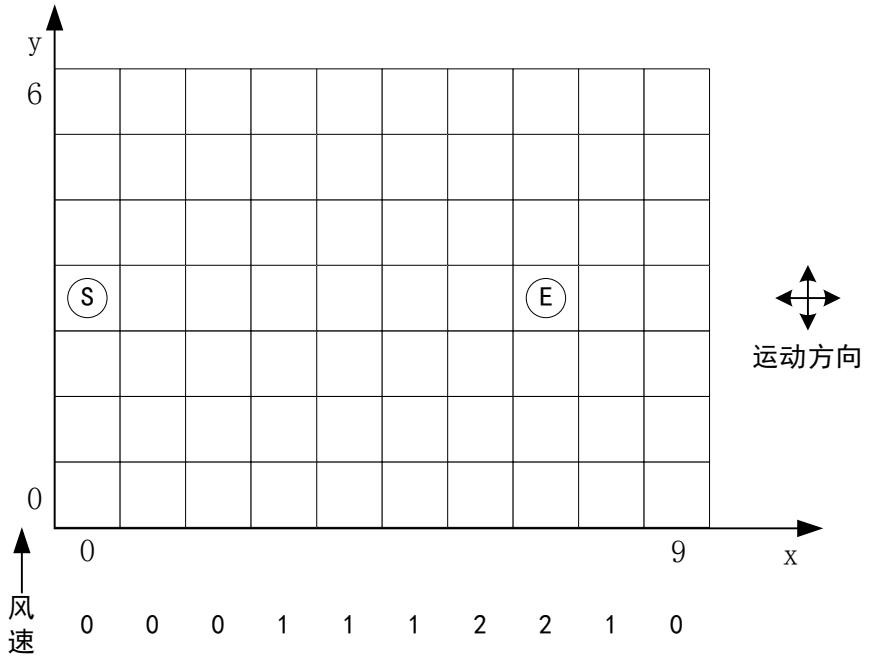
- 算法实战
  - 1. 确定状态空间S
  - 2. 确定动作集A
  - 3. 设计奖励计算函数$R(s,a)$
  - 4. 训练代理
  - 5. 应用代理

- 案例（有风的网格）
  - 二维网格空间
  - 部分列存在一个向上的风力，推动物体向上移动
  - 上下左右四种运动方式
  - 目标：由S点到达E点



算法原理

- 算法实战
  - 1. 确定状态空间S
    - $S = \{(x, y) | x \in [0.9], y \in [0,6]\}$
    - $|S| = 70$
    - 起始状态$(3,0)$
    - 终止状态$(3,7)$
  - 2. 确定动作集A
    - $A = \{up, down, left, right\}$
    - $|A| = 4$
  - 3. 设计奖励函数$R(s, a)$
    - 未到达终止状态时持续收到-1的奖励



算法原理

- 算法实战

```
def get_next_state(state, action):
    """
    get next state according to current state and action
    :param state: current state
    :param action: current action
    :return: next state
    """
    i, j = state
    if action == ACTION_UP:
        return [max(i - 1 - WIND[j], 0), j]
    elif action == ACTION_DOWN:
        return [max(min(i + 1 - WIND[j], WORLD_HEIGHT - 1), 0), j]
    elif action == ACTION_LEFT:
        return [max(i - WIND[j], 0), max(j - 1, 0)]
    elif action == ACTION_RIGHT:
        return [max(i - WIND[j], 0), min(j + 1, WORLD_WIDTH - 1)]
    else:
        assert False
```

环境模型代码

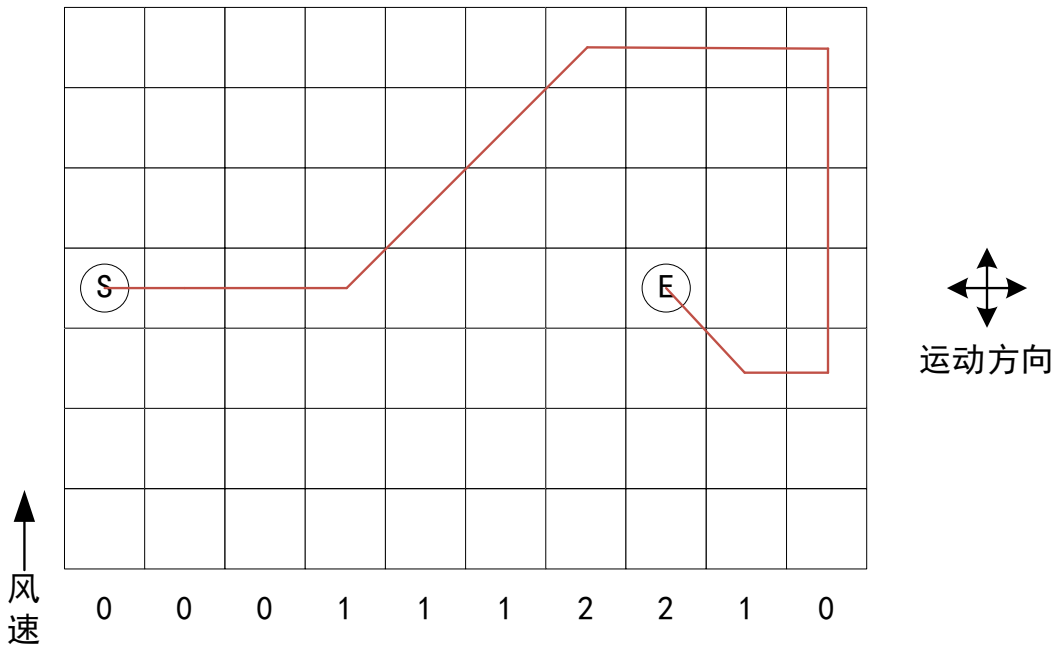
```
def play_one_round(q_value):
    """
    play for an round of windy world
    :param q_value: q table
    """
    # initialize state
    state = START

    # keep going until get to the goal state
    while state ≠ GOAL:
        # choose the action based on epsilon-greedy algorithm
        if np.random.binomial(1, EPSILON) == 1:
            action = np.random.choice(ACTIONS)
        else:
            values_ = q_value[state[0], state[1], :]
            action = np.random.choice([action_ for action_, value_ in enumerate(values_)
if value_ == np.max(values_)])

        next_state = get_next_state(state, action)
        values_ = q_value[next_state[0], next_state[1], :]
        next_action = np.random.choice([action_ for action_, value_ in enumerate(values_)
if value_ == np.max(values_)])

        # update q value table
        q_value[state[0], state[1], action] += ALPHA * (REWARD + q_value[next_state[0],
next_state[1], next_action] - q_value[state[0], state[1], action])
        state = next_state
```

单次训练代码

算法原理

- 算法实战

```
Optimal policy is:
['L', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'D']
['D', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'R', 'D']
['R', 'R', 'R', 'R', 'R', 'R', 'R', 'L', 'R', 'D']
['R', 'R', 'R', 'R', 'R', 'R', 'R', 'G', 'R', 'D']
['D', 'R', 'R', 'R', 'R', 'R', 'U', 'D', 'L', 'L']
['D', 'R', 'R', 'R', 'R', 'U', 'U', 'R', 'L', 'R']
['U', 'R', 'R', 'R', 'U', 'U', 'U', 'U', 'L', 'L']
```



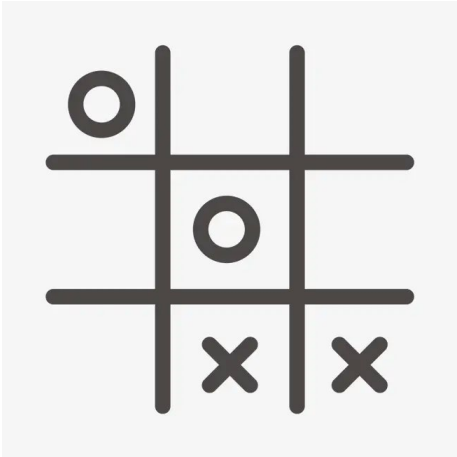
算法原理

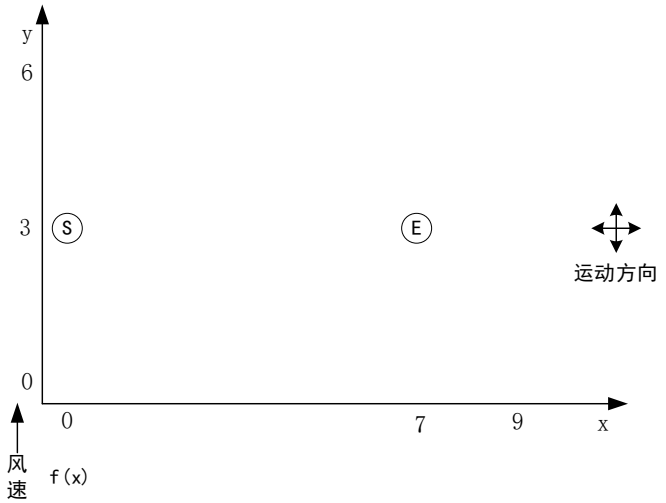
- 超参数说明
  - 学习率
    - 学习率越小，更重视之前的学习经验，性能更稳定，但收敛速度较慢，代理训练次数可能会增加
    - 学习率越大，更加重视本次学习经验，但容易出现围绕最优解震荡的情况
    - 建议设置随迭代轮数衰减的学习率
  - 奖励衰减系数
    - 取值在0~1之间
    - 越接近1，表示更加重视后续长远的收益，即代理更有“远见”
    - 越接近0，表示更看重当前的收益，及代理变得“短视”
    - 通常取值在0.5~0.8

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r + \gamma \max_i Q(s',i) - Q(s,a))$$

算法原理

- 优势
  - 模型无关方法，不需要了解环境的状态转移模型
  - Q表的状态和动作定义清晰，策略的可解释性较好
  - 模型逻辑简单，便于实现
- 劣势
  - 状态集必须是离散的，无法用于连续的状态
  - 面对复杂模型时，Q表会非常庞大，需要消耗大量时间和空间成本进行查找和存储



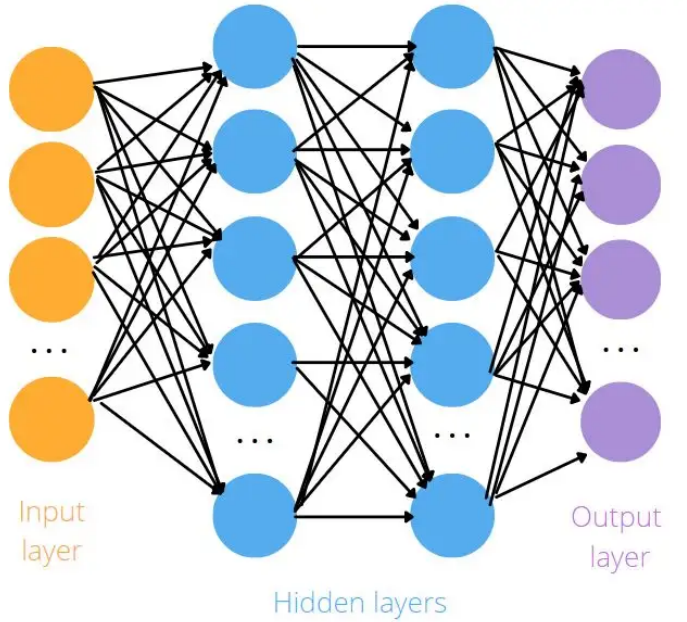


Deep Q-Learning

算法原理

- Deep Q-Learning算法（DQN）
  - 神经网络近似表示价值函数，使用神经网络（Q网络）代替Q表
    - 价值函数：$\hat{v}(s,w) \approx v_\pi(s)$
    - 动作价值函数（类比Q值）：$\hat{q}(s,a,w) \approx q_\pi(s,a)$

	ts_up	ts_down	tv_up	tv_down	adj_up	adj_down
0	4.717569	5.161536	5.120970	7.049681	5.795876	4.601859
1	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2	1.006945	2.919058	2.822928	6.647163	2.013414	3.385360
3	6.353390	0.896469	0.799802	2.538798	2.140769	1.229461
4	0.730251	0.774865	0.906058	5.463744	0.812705	0.994933
...	...	...	...	...	...	...
69	0.000000	0.163728	0.000000	0.000000	1.406014	0.000000
70	0.000000	0.530771	0.059254	8.640511	0.584880	1.105284
71	-0.105249	9.300511	-0.147061	6.992806	1.506404	0.408057
72	0.321237	0.175072	-0.212824	0.009448	5.602252	0.000000
73	0.493080	0.313502	0.217867	0.353771	9.759383	1.097195



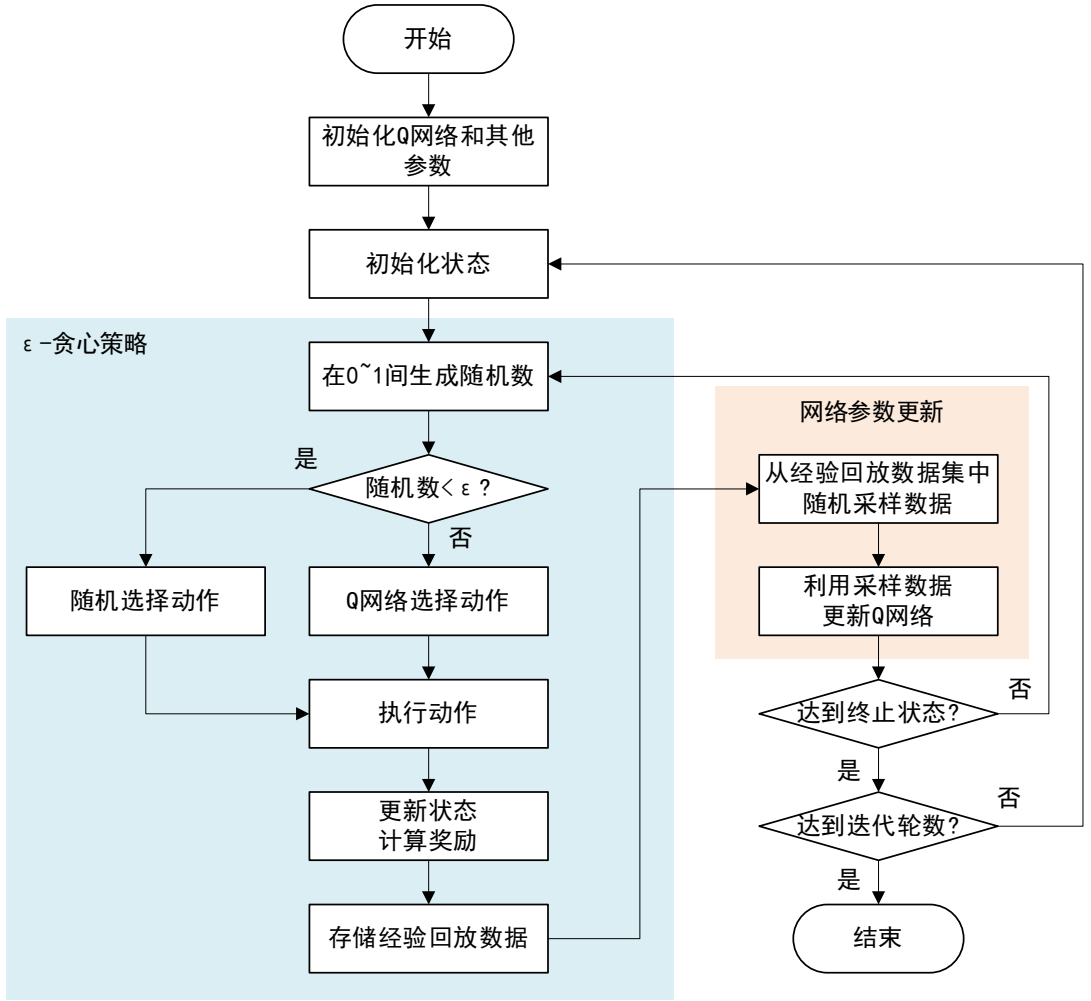
算法原理

T	训练代理在连续决策问题上做出最优决策
I	动作集、环境模型（状态集、奖励）、迭代轮数
P	初始化参数 For epoch =1, N: 1. 依据当前状态选择的动作 2. 执行动作，从环境中获取下一阶段状态 3. 计算奖励和价值 4. 更新神经网络参数
O	能够进行最优决策的代理

P	由于动作集或状态集过大导致的Q表训练困难
C	环境满足MDP假设，决策问题存在最优解
D	训练成本较高
L	Nature（2015）

算法原理

- 算法流程
  - ε-贪心策略
    - 每次以ε的概率进行探索（随机选择），以$1-\varepsilon$的概率进行利用（使用Q网络）
  - 存储经验回放
    - 记录新旧状态的特征向量$\phi(s)$ $\phi(s')$、动作a、奖励r、终止状态标识is_end，作为经验回放数据，存入集合D



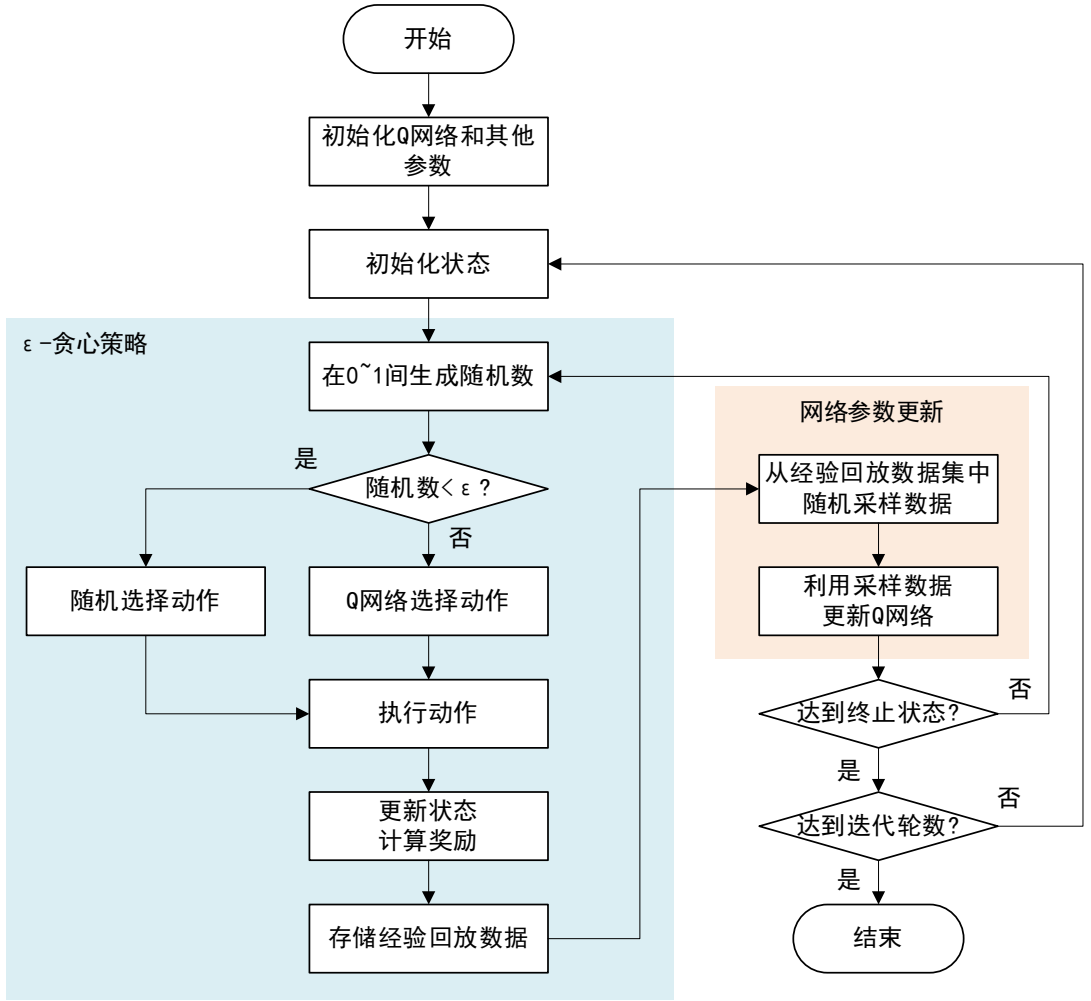
算法原理

- 算法流程
  - 网络参数更新
    - 从D中采样m个样本分别计算其Q值

$$y_j = \begin{cases} r_j & is\_end_j\ is\ true \\ r_j + \gamma \max_{a'} Q\left(\phi\left(s_j'\right), a_j', w\right) & is\_end_j\ is\ false \end{cases}$$

    - 使用MSE损失函数，反向传播更新Q网络参数

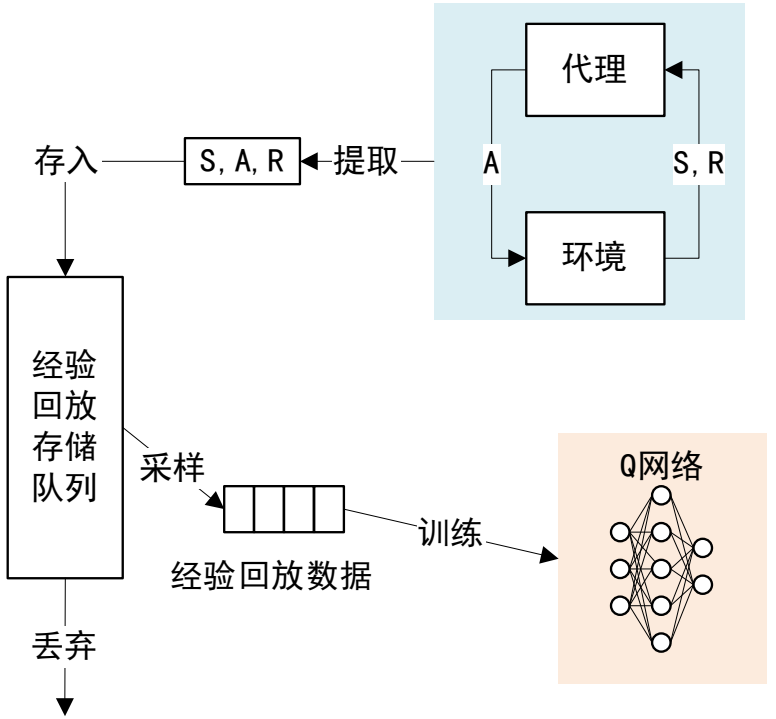
利用经验回放机制，避免丢失以前经验，
设置经验回放集合，随机采样数据更新网络



算法原理

- 经验回放
  - 序列相关性：连续的状态行动数据存在相关性，难以满足独立同分布假设
  - 数据使用效率：Q-Learning算法中，每次交互生成的数据使用后直接丢弃，若在训练Q网络时使用相同方法，数据的利用率过低，需要更多的交互次数，时间成本巨大

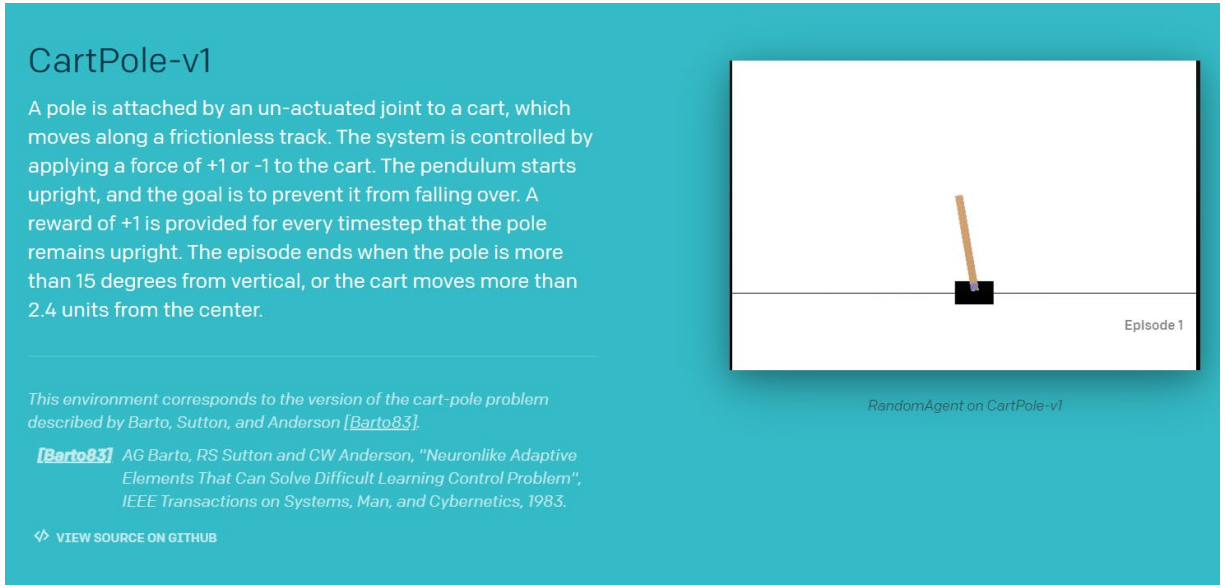
- 收集阶段
  - 从交互中收集经验回放样本
  - 按时间顺序存放，若存满则覆盖最早样本
- 采样阶段
  - 随机采样一批样本进行学习



算法原理

- 算法实战
  - Q-Learning：确定动作集和状态集
  - DQN：确定动作集，依据状态特征和动作构建Q网络

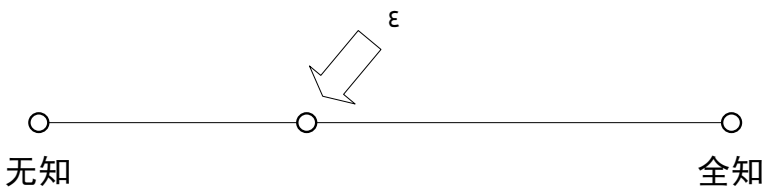
- 超参数说明
  - 学习率
  - 奖励衰减系数
  - ε策略的概率ε
  - 经验回放存储数量
  - 经验回放采样条数
  - Q网络层数和大小

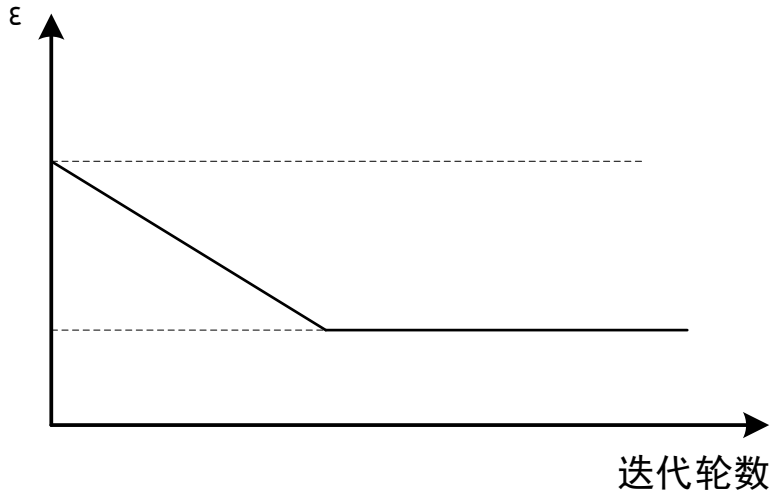


CartPole v1

算法原理

- 超参数说明
  - ε策略的概率ε
    - 以ε的概率随机选择动作，以$1-\varepsilon$的概率进行使用Q网络计算动作
    - 本质上体现的是学习过程中对已有知识利用和对未知信息探索的权衡
    - 通常随着学习的进行不断减小
    - 初始设为0.5或更高





算法原理

- 超参数说明
  - 经验回放存储数量
    - 存储数量小，Q网络训练时容易发散
    - 存储数量大，增加存储时间，导致代理训练时间变长，同时增大存储空间开销
  - 经验回放采样条数
    - 采样条数小，Q网络参数更新速度慢，代理训练时间增加
    - 采样条数大，难以打破数据关联性
  - Q网络层数与大小

利用经验回放机制，打破关联性，且避免丢失以前经验，
设置经验回放集合，随机采样数据更新网络

算法原理

- 优势
  - 能够应对环境状态连续的情况，面对复杂环境模型表现更好
- 劣势
  - 容易出现over-estimate（过估计）问题，影响模型性能
    - 解决思路：使用两个网络，分别负责动作的选择和评估->DDQN

应用总结

应用总结

- 应用领域
  - 智能交通：有轨自动驾驶（ATO）、汽车自动驾驶
  - 网络安全：入侵检测、自主网络防御
  - 强化学习调参
  - 其他领域：nlp（序列标注、文章结构判断…）
- 未来发展
  - 多目标
    - 研究在多个不同优化目标约束下，代理进行最优决策的方法
  - 多代理
    - 多个代理协同决策或者相互对抗博弈
  - 迁移学习

Decision making

参考文献

[1] Watkins C J C H. Learning from delayed rewards[J]. 1989.
[2] Mnih V, Kavukcuoglu K, Silver D, et al. Playing atari with deep reinforcement learning[J]. arXiv preprint arXiv:1312.5602, 2013.
[3] Mnih V, Kavukcuoglu K, Silver D, et al. Human-level control through deep reinforcement learning[J]. nature, 2015, 518(7540): 529-533.
[4] https://github.com/ljpzzz/machinelearning/tree/17302f708146ad46838b3782bf735364fa3cf16d/reinforcement-learning

上善若水。水善利万物
而不争，处众人之所恶，
故几於道。居善地，心
善渊与善仁，言善信，
正善治，事善能，动善
时。夫唯不争，故无尤。

谢谢！