RL (Reinforcement learning)

强化学习关注智能体和环境交互过程中的学习，这是一种试错型学习（trial-and-error learning）范式

Since RL requires a lot of data, therefore it is most applicable in domains where **simulated data** is readily available **like gameplay, robotics**.

状态：

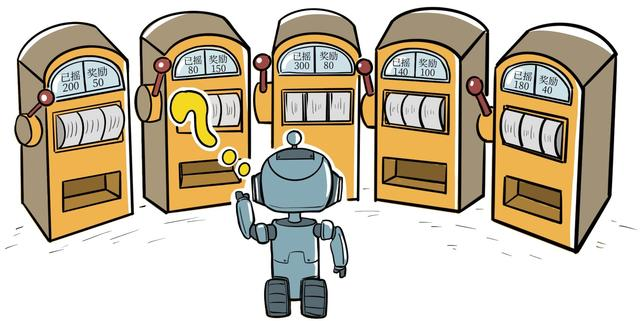
Policy策略：状态->动作的映射。如表格，函数或DNN

奖励信号：一个数值，代表着强化学习机采取行动后的即时奖励。最大化所得奖励是强化学习机的最终目标，为了完成这个目标，我们不断地调整策略。

价值函数：一个函数，用于描述给定状态下的可能获得的远期奖励。

环境模型：

多臂老虎机: 无状态的强化学习, 与环境的交互并不会改变环境



有一个拥有K根拉杆的老虎机，拉动每一根拉杆都对应一个关于奖励的概率分布 。我们每次拉动其中一根拉杆，就可以从该拉杆对应的奖励概率分布中获得一个奖励 。采用怎样的操作策略才能使获得的累积奖励最高?

我们在**各根拉杆的奖励概率分布未知**的情况下，从头开始尝试，**目标是在操作T次拉杆后获得尽可能高的累积奖励**。由于奖励的概率分布是未知的，因此我们需要在“探索拉杆的获奖概率”和“根据经验选择获奖最多的拉杆”中进行权衡。

**为了知道拉动哪一根拉杆能获得更高的奖励，我们需要估计拉动这根拉杆的期望奖励。由于只拉动一次拉杆获得的奖励存在随机性，所以需要多次拉动一根拉杆，然后计算得到的多次奖励的期望**

**探索（exploration）**是指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。例如，对于一个 10 臂老虎机，我们要把所有的拉杆都拉动一下才知道哪根拉杆可能获得最大的奖励。

**利用（exploitation）**是指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。例如，对于一个 10 臂老虎机，我们只拉动过其中 3 根拉杆，接下来就一直拉动这 3 根拉杆中期望奖励最大的那根拉杆，但很有可能期望奖励最大的拉杆在剩下的 7 根当中

* 常用的思路：开始多做探索，对每根拉杆有比较准确的估计后，再进行利用

def run(num\_steps, K=10, init\_prob=1.0):

counts = np.zeros(K) # 每根拉杆的尝试次数

estimates = np.array([init\_prob] \* K) # 每根拉杆的期望奖励

actions = [] # 维护一个列表,记录每一步的动作

# 随机生成K个0～1的数, 作为拉动每根拉杆的获奖概率

probs = np.random.uniform(size=K)

for i in range(num\_steps):

**# 根据策略选择动作（前1000次随机选择，之后根据经验）**

k = np.random.randint(0, K) if i < 1000 else np.argmax(estimates)

counts[k] += 1

actions.append(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或0

r = 1 if np.random.rand() < probs[k] else 0

# 更新期望奖励估值

estimates[k] += 1. / counts[k] \* (r - estimates[k])

拉杆奖励分布: [0.42 0.72 0. 0.3 0.15 0.09 0.19 0.35 0.4 0.54] --Ground Truth

期望奖励估计: [0.49 0.73 0. 0.26 0.15 0.08 0.15 0.37 0.35 0.61]

累积懊悔： 362.1 # 懊悔：每次拉杆与最优拉杆的期望奖励差

* ϵ-贪心算法

完全贪婪算法即在每一时刻采取期望奖励估值最大的动作（拉动拉杆），ϵ-贪婪算法在完全贪婪算法的基础上添加了噪声，每次以概率1- ϵ选择以往经验中期望奖励估值最大的那根拉杆（利用），以概率ϵ随机选择一根拉杆（探索），随着探索次数的不断增加，我们对各个动作的奖励估计得越来越准，此时我们就没必要继续花大力气进行探索。所以在 ϵ -贪婪算法的具体实现中，我们可以令ϵ随时间衰减，即探索的概率将会不断降低。

**# 根据eps贪心算法策略选择动作**

epsilon = 1 / (i+1) # epsilon值随时间衰减

k = randint(0, K) if random() < epsilon else np.argmax(self.estimates)

拉杆奖励分布：[0.84 0.18 0.25 0.77 0.83 0.21 0.59 0.82 0.7 0.67] --Ground Truth

期望奖励估计：[0.85 0. 0.5 0.5 0.8 0. 0.67 0.5 0.5 0.78]

累积懊悔： 7.7

* 上置信界算法

基于不确定性，某些拉杆只被拉动过几次，它的不确定性很高。一根拉杆的不确定性越大，它就越具有探索的价值

不确定性度量，它会随着一个动作被尝试次数的增加而减小

上置信界（upper confidence bound，UCB）算法是一种经典的基于不确定性的策略算法，UCB 算法在每次选择拉杆前，先估计每根拉杆的期望奖励的上界，使得拉动每根拉杆的期望奖励只有一个较小的概率超过这个上界，接着选出期望奖励上界最大的拉杆，从而选择最有可能获得最大期望奖励的拉杆。

N = i+1

ucb = estimates +coef \* np.sqrt(np.log(N) / (2 \* (counts + 1))) # 计算上置信界

k = np.argmax(ucb) # 选出上置信界最大的拉杆

马尔可夫性质：将来状态只取决于现在，与过去无关

马尔可夫过程：状态空间，其中下一个时刻的状态 只取决于当前状态，则状态转移矩阵满足：

马尔可夫决策过程

策略函数：状态空间到动作空间的映射图，定义了给定状态下做出给定动作的概率

**策略状态的价值函数：**从状态开始，agent按照策略进行决策所获得的回报的概率期望值

其中是折扣因子，

如果 则agent目光短浅，只关心最大化当前收益，一般来说，最大化当前收益会减少未来的收益，以至于实际上的收益变少了。

如果接近1,则agent更多考虑未来收益，变得更有远见了

**策略状态的动作价值函数:** 从状态开始，agent按照策略进行决策，执行动作之后，所获得的回报的概率期望值

价值函数和都能从经验中估算得到

Diagram

Description automatically generated A picture containing shape

Description automatically generated

对于马尔可夫过程，从当前状态开始，根据概率进行状态转移至下一状态

对于马尔可夫决策过程，在状态之间多了一层动作，我们需要先根据策略函数采取动作, 之后再根据概率进行状态转移至下一状态