多臂老虎机问题

问题介绍

问题定义：

多臂老虎机问题是一个简化版的强化学习，与强化学习不同的是，多臂老虎机不存在状态信息，只有动作和奖励。

多臂老虎机问题就是在一个多根拉杆的老虎机中每拉动一根拉杆就会获得一个随机的奖励，在各根拉杆的奖励概率分布未知的情况下，从头开始尝试，希望在T次拉杆后尽可能获得更高的累积奖励

探索与利用：

在这个过程就会遇到**探索与利用**这一经典问题，**探索**指的是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**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。

- 探索：尝试更多可能的决策，不一定会是最优收益
- 利用：执行能够获得已知最优收益的决策

因此，多臂老虎机问题转换为“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。从而判断采用怎样的操作策略才能使获得的累积奖励最高

问题实现

问题形式化书上说的很清楚，动手学强化学习重点还是在动手，多臂老虎机的求解方案在书上已经有了公式推导，所以学习的重点就是代码的实现

伯努利多臂老虎机：

现定义一个多臂老虎机，拉杆数为 10，其中拉动每根拉杆的奖励服从伯努利分布，即每次拉下拉杆有p的概率获得的奖励为 1，有$1-p$的概率获得的奖励为 0。奖励为 1 代表获奖，奖励为 0 代表没有获奖

```
# 导入需要使用的库,其中numpy是支持数组和矩阵运算的科学计算库,而matplotlib是绘图库
import numpy as np
import matplotlib.pyplot as plt


class BernoulliBandit:
    """ 伯努利多臂老虎机,输入K表示拉杆个数 """
    def __init__(self, K): # 实例化对象时自动初始化
        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
    #成员变量:
    #每根获得奖励的概率probs[i],获奖率最大的拉杆号best_idx,最大的获奖概率best_prob,拉杆数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

#测试
np.random.seed(1)  # 设定随机种子,使实验具有可重复性
K = 10
bandit_10_arm = BernoulliBandit(K)
print("随机生成了一个%d臂伯努利老虎机" % K)
print("获奖概率最大的拉杆为%d号,其获奖概率为%.4f" %
      (bandit_10_arm.best_idx, bandit_10_arm.best_prob))

随机生成了一个10臂伯努利老虎机
获奖概率最大的拉杆为1号,其获奖概率为0.7203
```

定义一个**Solver** 基础类来实现上述的多臂老虎机的求解方案。

在该类中需要实现以下函数功能：

- 根据策略选择动作
- 根据动作获取奖励
- 更新期望奖励估值
- 更新累积懊悔和计数

在下面的 MAB 算法（要么探索要么利用）基本框架中，我们将根据策略选择动作、根据动作获取奖励和更新期望奖励估值放在 run_one_step() 函数中，由每个继承 Solver 类的策略具体实现。而更新累积懊悔和计数则直接放在主循环 run() 中。

```
class Solver:
    """ 多臂老虎机算法基本框架 """
    def __init__(self, bandit):
        self.bandit = bandit # 一个未知类型的老虎机
        self.counts = np.zeros(self.bandit.K)  # 每根拉杆的尝试次数,初始每根拉杆都为0次
        self.regret = 0.  # 当前步的累积懊悔
        self.actions = []  # 维护一个列表,记录每一步的动作
        self.regrets = []  # 维护一个列表,记录每一步的累积懊悔

    def update_regret(self, k):
        # 计算累积懊悔并保存,k为本次动作选择的拉杆的编号
        self.regret += self.bandit.best_prob - self.bandit.probs[k]
        self.regrets.append(self.regret) # 添加到每步懊悔列表中

    def run_one_step(self):
        # 返回当前动作选择哪一根拉杆,由每个具体的策略实现
        raise NotImplementedError

    def run(self, num_steps):
        # 运行一定次数,num_steps为总运行次数
        for _ in range(num_steps):
            k = self.run_one_step()
            self.counts[k] += 1 # 第K根拉杆拉动次数
```

```
            self.actions.append(k) # 将拉下第K根拉杆这一举措,添加到行动列表中
            self.update_regret(k) # 计算当前懊悔并累积保存
```

好啦，基本的算法框架就有了，唯一需要实现的就是算法中拉动杆子的策略。而策略的选择就是**探索与利用**问题

策略选择

前面也说到了，我们需要在探索和利用做一个权衡，在实现伯努利多臂老虎机进行测试时，得到了1号拉杆获奖的概率是最大的，所以我们可以选择**完全贪婪**的策略，即每次我们都去拉1号杆这种获得期望奖励估值最大的动作，这就是纯粹的利用，而没有探索

但需要注意的是如果一直不探索新策略，那么当前我们选的1号拉杆这个认知也是不确保为最优的，比如2号拉杆获奖概率虽为0.2，但在某次决策中选取了1号拉杆没有中奖而获奖概率低的2号反倒中奖。而这样就会导致我们的懊悔值是递增的无法收敛的

同样的，当我们一直探索新策略，而探索新的策略也不能保证是最优解，同样会增加懊悔值，导致懊悔值不断递增无法收敛。

所以是否存在一个**探索和利用不断权衡**的方法，使得懊悔值具有次线性从而保证收敛呢?

下面是几个策略选择的算法。

ε—贪婪算法:

前面提到的，完全贪婪下去一直去利用是不可行的，但从直观上来说我们一直选择概率最大的拉杆去拉，累积的奖励概率应该是很大的。所以我们需要对完全贪婪算法进行一些修改，增加一些探索

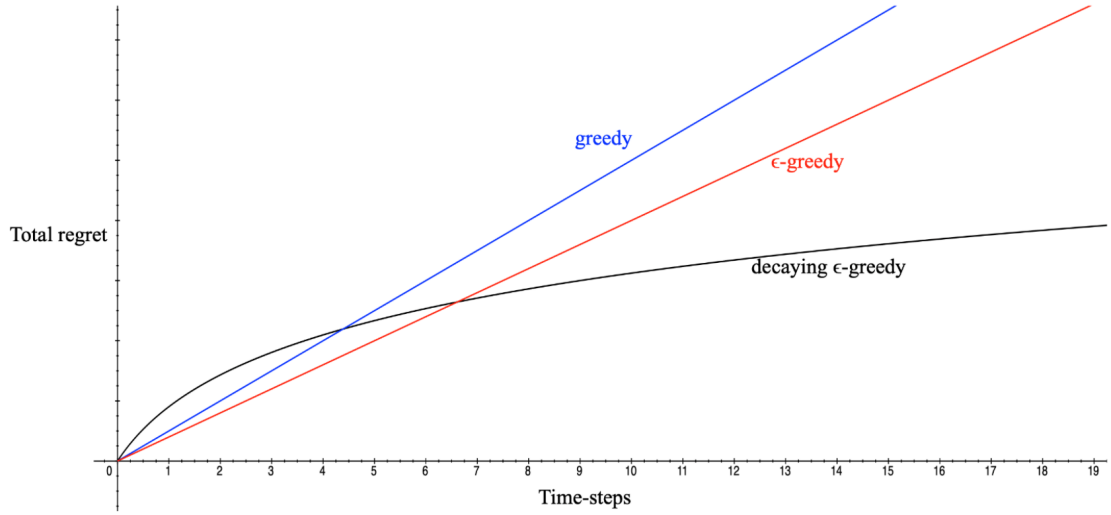
比较经典的算法就是ε—贪婪算法，它的思想是以1-ε较大的概率去采取贪心策略去利用（一直去拉中奖概率最大的杆子），以比较小的ε概率去做一个均匀分布的探索（编号1~10，均匀分布1/10的概率去选）

如果一直选择的是常量的概率ε，那么经过计算推导懊悔值仍是线性递增的，只是增长率是比贪心策略小的。

但是我们对ε随着时间增大进行衰减，就能得到在理论上对数进行渐进收敛的懊悔值。这是因为随着时间不断增加，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。

⚠️需要注意的是，不会在有限的步数内衰减至 0，因为基于有限步数观测的完全贪婪算法仍然是一个局部信息的贪婪算法，永远距离最优解有一个固定的差距。所以很难找到合适的衰减规划



上图是完全贪婪、ε—贪婪算法、ε衰减的贪婪算法，随着拉杆次数的增加累积懊悔值的对比

书上有ε—贪婪算法和ε衰减的贪婪算法，要学就学最好的，所以来看一下ε衰减的贪婪算法是如何实现的吧

首先为了判断该策略的优劣，利用**matplotlib**定义一个可视化的类，来体现累积的懊悔值

```
def plot_results(solvers, solver_names):
    """生成累积懊悔随时间变化的图像。输入solvers是一个列表,列表中的每个元素是一种特定的策略。
    而solver_names也是一个列表,存储每个策略的名称"""
    for idx, solver in enumerate(solvers):
        time_list = range(len(solver.regrets))
        plt.plot(time_list, solver.regrets, label=solver_names[idx])
    plt.xlabel('Time steps')
    plt.ylabel('Cumulative regrets')
    plt.title('%d-armed bandit' % solvers[0].bandit.K)
    plt.legend()
    plt.show()
```

matplotlib绘图还是搞不懂，一直看别人怎么实现的也不是办法，有机会一定要去系统的学一学！！！

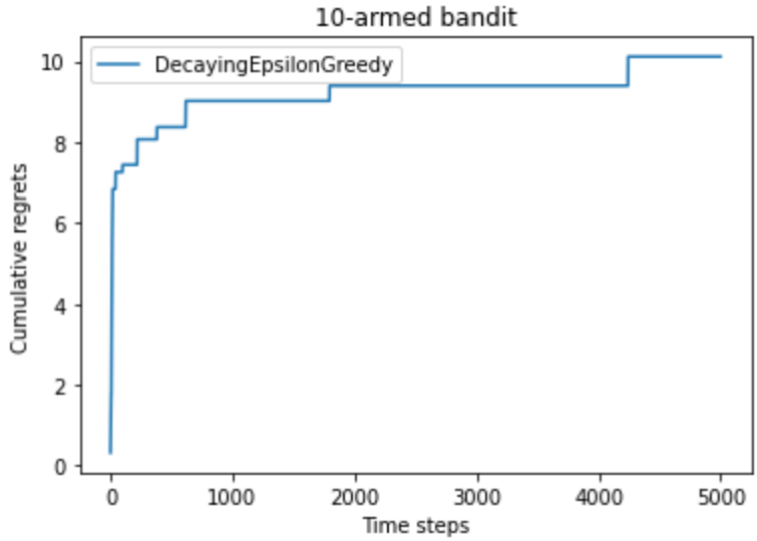
ε值随时间衰减的 ε-贪婪算法，采取的具体衰减形式为反比例衰减，公式为：

$$\epsilon_t = \frac{1}{t}$$

```
class DecayingEpsilonGreedy(Solver):
    """ epsilon值随时间衰减的epsilon-贪婪算法,继承Solver类 """
    def __init__(self, bandit, init_prob=1.0):
        super(DecayingEpsilonGreedy, self).__init__(bandit)
        #将一个老虎机作为自己的成员并初始化
        self.estimates = np.array([init_prob] * self.bandit.K)
        #初始化拉动所有拉杆的期望估计值
        self.total_count = 0 # 表示时间的累积,用于确定

    def run_one_step(self):
        self.total_count += 1 # 每拉动一次,拉动次数加1
```

```
12
13          #决策拉动哪个拉杆
14          if np.random.random() < 1 / self.total_count:  # epsilon值随时间衰减
15              k = np.random.randint(0, self.bandit.K)
16              #若概率小于ϵ则随机选择一根拉杆
17          else:
18              #若概率大于等于1-ϵ则选择期望奖励估值最大的拉杆
19              k = np.argmax(self.estimates)
20
21          r = self.bandit.step(k) # 得到拉动k杆动作的奖励
22          self.estimates[k] += 1. / (self.counts[k] + 1) * (r -
23                                                             self.estimates[k])
24          #更新期望奖励的估计
25
26          return k # 返回决策拉动的杆号
27
28  #累积5000次拉杆后该策略的懊悔值,测试
29  np.random.seed(1)
30  decaying_epsilon_greedy_solver = DecayingEpsilonGreedy(bandit_10_arm)
31  decaying_epsilon_greedy_solver.run(5000)
32  print('epsilon值衰减的贪婪算法的累积懊悔为：',
    decaying_epsilon_greedy_solver.regret)
33  plot_results([decaying_epsilon_greedy_solver], ["DecayingEpsilonGreedy"])
34
35  epsilon值衰减的贪婪算法的累积懊悔为： 10.114334931260183
```



从实验结果图中可以发现，随时间做反比例衰减的ε -贪婪算法能够使累积懊悔与时间步的关系变成**次线性**的，这明显优于固定ε值的ε -贪婪算法。

我们还可以结合贪婪算法，对动作的奖励**积极初始化**，即我们对每一个动作a^i一开始就赋值一个比较高的期望奖励值$Q(a^i)$，这样就会使我们认为每个选项都可以是最优的，所以给予的值也比较高

然后以**蒙特卡洛方法**去更新这个$Q(a^i)$，虽然$Q(a^i)$是有偏的，但是随着迭代次数的增加这个偏差的影响会越来越小（原因我也不清楚🐻），但是在选择次数很多次中可能存在有的动作被选择的比较少（比如拉3号杆这一动作），那么$Q(a^3)$这个值还是会比较高。那么我们根据期望选择奖励较高的策略，我们就会尝试去探索选择这些比较少选择的动作，就能达到探索没有探索到的或者少探索的一些动作

利用这种方法和贪婪算法的结合可以提高我们最优动作的选择率，⚠️但仍然会陷入局部最优。（原因我没有听明白...）

这是我们去提高偏差来考虑动作的价值，我们还可以显示的根据动作价值的分布来平衡和探索和利用，来选择动作

上置信界算法 UCB：

一个经验性的指导，如果说对于某个动作a尝试的次数越多（N(a)越大），那么我们对这个动作a所获得奖励的认知越清晰（U(a)不确定性越小），如果对于动作a尝试次数越少，那么不确定性越大

于是可以做一个这样的权衡，不确定性越大的Q(a^i)，越具有探索的价值，有可能会是最好的策略

即策略有：

$$\pi：\ a = \arg\max_{a\in\mathcal{A}} \hat{Q}(a) + \hat{U}(a)$$

策略中的**U(a)**即为上置信界

不确定如何度量的数学推导已经让我🐷🧠过载了，所以这里直接上结论

$$\hat{U}_t(a) = \sqrt{-\frac{\log p}{2N_t(a)}}$$

其中公式中的概率*p*是自己确定的，所以设定一个概率*p*后就可以计算相应的不确定性度量**U(a)**了，可以笼统的看作U(a) = c/N(a)，即a尝试的次数越多，U(a)不确定性越小

该策略可以直观的概括为在每次选择拉杆前，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*p*概率超过这个上界，然后选出期望奖励上界最大的拉杆，这根拉杆也代表着最有可能获得最大期望奖励的拉杆。

下面编写UCB算法，根据不确定性度量的公式，选择概率p = 1/t（t为迭代的总次数），并对分母加上一个任意的常数（这里选择为1）来避免分母出现0的情况，得：

$$\hat{U}_t(a) = \sqrt{\frac{\log t}{2(N_t(a)+1)}}$$

⚠️别看分母加了个1，U的公式形式改变了，但是我们每个动作的不确定性度量都加的是固定值1，所以相当于都没加

同时设定一个常系数coef来控制不确定性这个单位在期望奖励上界所占的比重，使得采取的策略为：

$$a = \arg\max_{a\in\mathcal{A}} \hat{Q}(a) + c\cdot\hat{U}(a)$$

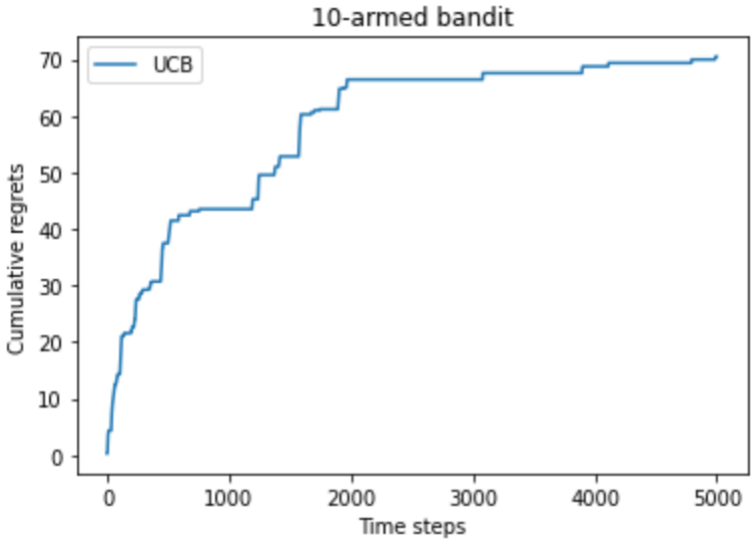
```
class UCB(Solver):
    """ UCB算法,继承Solver类 """
    def __init__(self, bandit, coef, init_prob=1.0):
        super(UCB, self).__init__(bandit)
        self.total_count = 0
        self.estimates = np.array([init_prob] * self.bandit.K)
        #初始化拉动所有拉杆的期望估计值
        self.coef = coef # 常数,控制不确定性在奖励期望中所占的比重

    #决策
    def run_one_step(self):
        self.total_count += 1 # 拉杆总次数
        ucb = self.estimates + self.coef * np.sqrt(
```

```
            np.log(self.total_count) / (2 * (self.counts + 1)))  # 计算上置信界
        k = np.argmax(ucb)  # 选出上置信界最大的拉杆
        r = self.bandit.step(k) # 得到拉动k杆动作的奖励
        self.estimates[k] += 1. / (self.counts[k] + 1) * (r - self.estimates[k])
        #更新拉杆k的期望奖励估计
        return k #返回决策

#累积5000次拉杆后该策略的懊悔值,测试
np.random.seed(1)
coef = 1  # 控制不确定性比重的系数
UCB_solver = UCB(bandit_10_arm, coef)
UCB_solver.run(5000)
print('上置信界算法的累积懊悔为：', UCB_solver.regret)
plot_results([UCB_solver], ["UCB"])

上置信界算法的累积懊悔为： 70.45281214197854
```



同样可以看出累积懊悔和时间也是次线性的

汤普森采样算法：

我们根据动作价值的分布，来思考到底选择哪个动作

我们不用显示的去弄清每个动作价值分布的具体细节，而是根据每个动作成为最优的概率来选择动作这一策略。

那么怎样得到当前每个动作a的奖励概率分布并且在尝试过程中进行更新？

在实际情况中，我们通常用 **Beta 分布**对当前每个动作的奖励概率分布进行建模，这个奖励概率肯定是从0~1分布的，初始贝塔分布时，就是Beta (1,1)。其中左边的1表示刚开始拉杆拿到奖励的次数，而右边的1表示没拿到奖励的次数

随着拉杆被选择了m次，其中m_1次拿到了奖励，m_2 次没拿到奖励。在拉动m次后，拉这根杆子这个动作获得奖励的概率分布在初始为Beta(1,1)分布的基础上，奖励服从变为了**Beta(m_1+1 , m_2+1)**

然后从每个拉杆的奖励分布中随机采样得到对应拉杆获得的奖励概率，比较这些概率选取最大的概率拉杆来作为决策，然后根据决策在判断拿没拿到奖励，如果拿到奖励贝塔分布第一个参数就加1，没拿到奖励贝塔分布第二个参数就加1。这样就保证了每个动作a在不断尝试中还更新了奖励的概率分布，然后不断循环完成汤普森采样

实现：

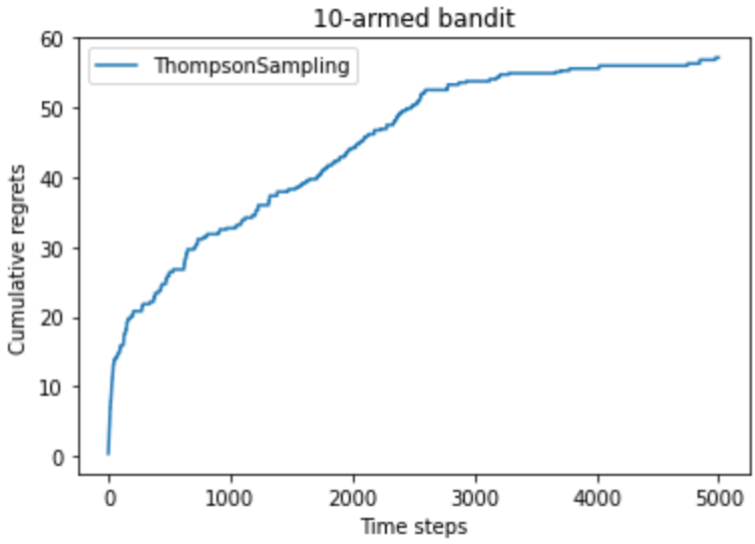
```
class ThompsonSampling(Solver):
    """ 汤普森采样算法,继承Solver类 """
    def __init__(self, bandit):
        super(ThompsonSampling, self).__init__(bandit)
        self._a = np.ones(self.bandit.K)  # 列表,表示每根拉杆奖励为1的次数
        self._b = np.ones(self.bandit.K)  # 列表,表示每根拉杆奖励为0的次数
        #np.ones初始化保证每根拉杆服从Beta(1,1)分布

    #决策
    def run_one_step(self):
        samples = np.random.beta(self._a, self._b)  # 按照Beta分布采样一组奖励样本
        k = np.argmax(samples)  # 选出采样奖励最大的拉杆
        r = self.bandit.step(k) # 判断该拉杆有没有拿到奖励

        self._a[k] += r  # 更新Beta分布的第一个参数
        self._b[k] += (1 - r)  # 更新Beta分布的第二个参数
        return k # 返回决策

#累积5000次拉杆后该策略的懊悔值,测试
np.random.seed(1)
thompson_sampling_solver = ThompsonSampling(bandit_10_arm)
thompson_sampling_solver.run(5000)
print('汤普森采样算法的累积懊悔为：', thompson_sampling_solver.regret)
plot_results([thompson_sampling_solver], ["ThompsonSampling"])

汤普森采样算法的累积懊悔为： 57.19161964443925
```



同样也是次线性的

不过三种算法进行比较，对于伯努利型老虎机，ε衰减的贪婪算法懊悔度比较低，相较于最优

小结:

- 探索与利用是强化学习的试错法的必备技术
- 多臂老虎机可以被看作无状态的强化学习
- 多臂老虎机是研究探索与利用技术理论的最佳环境
    - 理论的渐近最优regret为O(logT)
- ε—贪婪算法、上置信界算法、汤普森采样算法在多臂老虎机任务中十分常用，强化学习的探索中也十分常用，最常用的就是ε—贪婪算法