强化学习基础篇（二十）k-bandit问题

最近回顾继续回顾了一遍第二版的《Reinforcement Learning: An Introduction》，发现有必要对那些基本的实验进行复现回顾，本文先针对多臂Bandit问题进行复现。

1、k-bandit问题设定

k-bandit问题考虑的是如下的学习问题：你要重复地在k个选项或者动作中进行选择。每次做出选择后，都会得到一定数值的收益，收益由你选择的动作决定的平稳概率分布产生。目标是在某一段时间内最大化总收益的期望。

k-bandit问题是源于老虎机（或者叫单臂Bandit），不同之处在于它有k个控制杆，而不是老虎机的一个控制杆。

在k-bandit的问题中，k个动作中的每一个在被选择时都有一个期望或者平均收益，我们称之为“价值”。我们将在时刻t选择的动作记为A_t，并将对应的收益记为R_t。任一动作a对应的价值记为$q_*(a)$，即为给定动作a时收益的期望：

$$q_*(a) = E[R_t|A_t = a]$$

如果我们知道了每个动作的价值，那么k-bandit问题就很简单，只需要每次选择价值最高的动作。但是我们不知道每个动作的价值的情况下，就只能对其进行归集。所以我们将对动作a在时刻t时的价值估计记为$Q_t(a)$，其估计的目标是希望他接近于$q_*(a)$。

Exploration与Exploitation:

既然我们会对动作a的价值进行估计，那么在任一时刻会至少有一个动作的估计价值是最高的。我们将这些对应最高估计值的动作称之为贪心(greedy)动作。我们持续选择greedy动作，则称为Exploitation。

$$A_t = argmax_a Q_t(a)$$

如果我们一直进行greedy选择动作，有可能陷入局部最优，为了能够获得全局最优。我们还需要在任务运行中去探索那些我们未知价值的动作，这就称为Exploration。

如果我们只是以ϵ的概率去探索未知的动作，而以$1-\epsilon$的概率持续进行greedy选择，那么这种规则称为$\epsilon - greedy$方法。

2、k-bandit建模

为了大致评估贪心方法和$\epsilon - greedy$方法，我们在一个10臂Bandit问题上进行相应的验证。

首先我们导入会用到的库函数。

```
import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from tqdm import trange
from pylab import mpl
import seaborn as sns
np.random.seed(sum(map(ord, "aesthetics")))
mpl.rcParams['font.sans-serif'] = ['SimHei']
```

接下来我们构建k-臂Bandit模型。

环境设置

在用代码实现Bandit时，我们需要考虑多方因素，下面先列出一个bandit初始化时需要考虑到的各种参数：

- k_arm：

  Bandit的臂数，其决定了可选动作空间的大小，例如当k_arm=10时意味着：每个时刻有10个可选动作，其分别决定着不同的收益。

- epsilon：

  在每次进行动作选取时，一个朴素的思想就是每次都选带来最大收益的动作，即 $A_t = agrmax_a Q_t(a)$，我们称之为“贪心”，贪心有利于exploitation但不利于exploration，为了平衡开发和试探，可采用$\epsilon - greedy$方法进行动作的选择，即在每个时刻，使用$1-\varepsilon$的概率选取最优动作，使用ϵ概率选取随机动作。

- initial：

  在对动作价值估计之前，我们通常为每个动作的价值$Q(a)$初始化为0。使用乐观初始值(即高于真实价值的均值)估计会去鼓励bandit进行explore，因为无论哪种动作被选取，其带来的收益均要小于乐观初始值，因此bandit会转而采取另一种动作，从而使每一个动作在收敛之前都被尝试很多次。

- step_size：

  步长参数，有固定步长和非固定步长两种类别可选。用于动作估计值的更新：$Q_{n+1} = Q_n + stepsize * (R_n - Q_n)$

- sample_averages：

  用非固定步长$1/n$作为步长参数$stepsize$，用于动作估计值的更新：$Q_{n+1} = Q_n + (R_n - Q_n)/n$

- UCB_param：

  UCB即对应置信度上界，即动作的选取从$At = agrmax_a Q_t(a)$改变为：

  $$A_t = argmax_a[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}]$$

  其中UCB_param即动作选取公式中的c，可解释为：平方根项是对a动作值估计的不确定性或方差的度量，参数c决定了置信水平，但随着a选取次数$N_t(a)$的增多，其不确定性会逐渐减少。

- gradient：

  对应梯度Bandit算法，算法中每个动作被选取的概率为softmax分布所确定：

  $$P(A_t = a) = \frac{e^{H_t(a)}}{\sum_{b-1}^{k} e^{H_t(a)}} = \pi_t(a)$$

  其中$\pi_t(a)$代表时刻t动作a被选择的概率，而$H_t(a)$则代表动作a的偏好函数。

- gradient_baseline：在梯度Bandit算法中，待学习的变量为偏好函数$H_t(a)$，可以把其理解为动作a的收益，但其本身不重要，重要的是一个动作对另一个动作的相对偏好。偏好函数的更新如下：

  $$H_{t+1}(a) = H_t(a) + stepsize * (R_t - baseline)(1 - \pi_t(A_t))$$

  当使用baseline时，baseline被赋值为到t为止的平均收益；不使用时则等于0。使用基准项可以实现缩减方差的作用，使算法快速收敛。

```
class Bandit:
    def __init__(self, k_arm=10, epsilon=0., initial=0., step_size=0.1, sample_averages=False, UCB_param=None,
                 gradient=False, gradient_baseline=False, true_reward=0.):
        self.k = k_arm  # 设定Bandit臂数
        self.step_size = step_size # 设定更新步长
```

```
        self.sample_averages = sample_averages # bool变量，表示是否使用简单增量式算法
        self.indices = np.arange(self.k) # 创建动作索引（和Bandit臂数长度相同）
        self.time = 0 # 计算所有选取动作的数量，为UCB做准备
        self.UCB_param = UCB_param # 如果设定了UCB的参数c，就会在下面使用UCB算法
        self.gradient = gradient # bool变量，表示是否使用梯度Bandit算法
        self.gradient_baseline = gradient_baseline # 设定梯度Bandit算法中的基准项（baseline），通常都用时刻t内的平均收益表示
        self.average_reward = 0 # 用于存储baseline所需的平均收益
        self.true_reward = true_reward  # 存储一个Bandit的真实收益均值，为下面制作一个Bandit的真实价值函数做准备
        self.epsilon = epsilon  # 设定epsilon-贪婪算法的参数
        self.initial = initial # 设定初始价值估计，如果调高就是乐观初始值
```

状态重置函数定义

reset(self)函数主要用于在episode结束后重置状态。

这次实验中我们会考虑在k-bandit中，动作的真实价值是从一个均值为0，方差为1的状态分布中进行选择的。

我把每一步的具体含义也打在代码中了，值得注意的是其中true_reward的用意，我是根据上下文才推测出来的，由于之后要对比使用和不使用baseline的训练效果，而真实价值是一个标准正态分布，这就导致它回报的平均值会接近0，即baseline接近0，这和不使用baseline的效果是类似的，所以在对比baseline效果的实验中我们通过true_reward把真实价值垫高。

```
    # 初始化训练状态，设定真实收益和最佳行动状态
    def reset(self):
        # #设定真实价值函数，在一个标准高斯分布上抬高一个外部设定的true_reward，true_reward设置为非0数字以和不使用baseline的方法相区分
        self.q_true = np.random.randn(self.k) + self.true_reward

        # 设定初始估计值，在全0的基础上用initial垫高，表现乐观初始值
        self.q_estimation = np.zeros(self.k) + self.initial

        # 计算每个动作被选取的数量，为UCB做准备
        self.action_count = np.zeros(self.k)
        # 根据真实价值函数选择最佳策略，为之后做准备
        self.best_action = np.argmax(self.q_true)
        # 设定时间步t为0
        self.time = 0
```

动作选择函数定义

其中考虑了如下几种动作选择情况：

- $\epsilon-greedy$动作选择：$A_t = \begin{cases} argmax_a Q(a), & 以1-\epsilon概率选择 \\ 随机动作 & 以\epsilon概率选择 \end{cases}$
- 置信度上界的动作选择：$A_t = argmax_a[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}}]$
- 基于梯度的动作选择：$P(A_t = a) = \frac{e^{H_t(a)}}{\sum_{b-1}^{k} e^{H_t(a)}} = \pi_t(a)$

```
    # 动作选取函数
    def act(self):
         # epsilon概率随机选取一个动作下标
        if np.random.rand() < self.epsilon:
```

```
            return np.random.choice(self.indices)

        # 基于置信度上界的动作选取
        if self.UCB_param is not None:
            # 按照公司计算UCB估计值
            UCB_estimation = self.q_estimation + \
                self.UCB_param * np.sqrt(np.log(self.time + 1) / (self.action_count + 1e-5))
            # 选择不同动作导致的预测值中最大的
            q_best = np.max(UCB_estimation)
            # 返回基于UCB的动作选择下值最大的动作
            return np.random.choice(np.where(UCB_estimation == q_best)[0])
        # 基于梯度Bandit算法
        if self.gradient:
            exp_est = np.exp(self.q_estimation)
            self.action_prob = exp_est / np.sum(exp_est)
            return np.random.choice(self.indices, p=self.action_prob)
        # 在未使用UCB与基于梯度的方法后，继续返回epsilon概率随机选择的greedy动作
        q_best = np.max(self.q_estimation)
        return np.random.choice(np.where(self.q_estimation == q_best)[0])
```

价值更新

由bandit建模部分可以得知，在选取完动作后，与动作价值更新有关的参数为**sample_averages、gradient、gradient_baseline**，

step函数的任务是接受agent动作，根据agent选择的方法更新价值函数估计，并返回reward以进行平均reward的计算。

给出代码如下：

```

    # take an action, update estimation for this action
    # 选择动作并进行价值更新
    def step(self, action):
        # generate the reward under N(real reward, 1)
        # 按照之前产生的真实价值作为均值，产生一个遵从正态分布的随机奖励
        reward = np.random.randn() + self.q_true[action]
        # 执行动作后时间步加一
        self.time += 1
        ## 动作选择数量计数
        self.action_count[action] += 1
        # 计算平均回报（return），为baseline做好准备
        self.average_reward += (reward - self.average_reward) / self.time

        if self.sample_averages:
            # update estimation using sample averages
            # 增量式实现（非固定步长1/n）
            # Qn+1 = Qn + [Rn-Qn]/n
            self.q_estimation[action] += (reward - self.q_estimation[action]) / self.action_count[action]

        # 使用梯度Bandit
        elif self.gradient:
            # 将k臂做onehot编码
            one_hot = np.zeros(self.k)
            # 将选择的动作位置置为1.
```

```
            one_hot[action] = 1
            # 在梯度Bandit上使用baseline
            if self.gradient_baseline:
                # 平均收益作为baseline
                baseline = self.average_reward
            else:
                #不使用baseline的情况
                baseline = 0
            # 梯度式偏好函数更新:
            # 对action At: Ht+1(At) = Ht(At) + å(Rt-R_avg)(1-π(At))
            self.q_estimation += self.step_size * (reward - baseline) * (one_hot - self.action_prob)
        else:
            # update estimation with constant step size
            # 如过上面两种方法都没有选取，就选用常数步长更新
            # Qn+1 = Qn + å(Rn-Qn)
            self.q_estimation[action] += self.step_size * (reward - self.q_estimation[action])
        return reward
```

3、辅助函数

simulate函数用来进行循环实验并返回不同策略指导下Bandit的平均回报和做出最佳选择的概率。

simulate由三个循环组成:

- 第一个循环是有对不同的实验组进行循环。
- 第二个循环是对实验次数进行循环，每次实验有着相同超参数但是不同奖励分布。
- 第三个循环是对每个Bandit进行预定时间步进行实验。在其中进行实际的动作与回报计算。最后返回每个Bandit选到最佳动作的概率和每个Bandit的平均回报，即选择不同计算策略和不同参数Bandit的表现，方便后面画图对比。

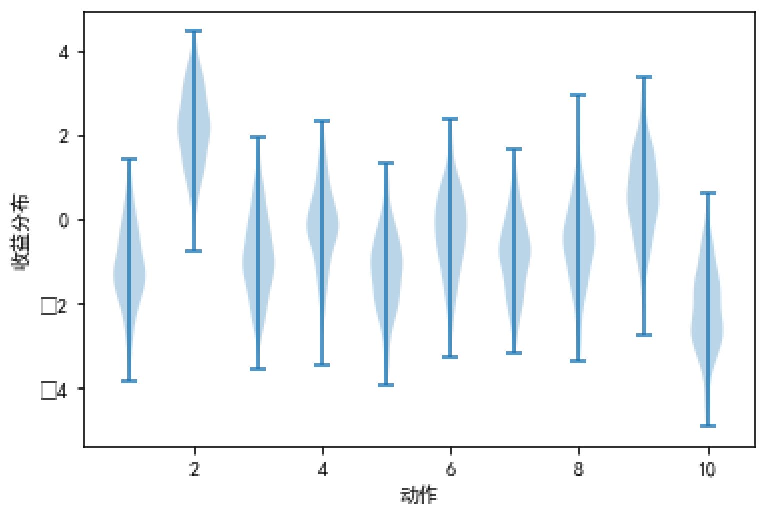
```
def simulate(runs, time, bandits):
    # 这里定义结果存储，按照实验的对照组数量，每个组实验的次数，以及每次实验的时间步数量进行存储
    rewards = np.zeros((len(bandits), runs, time))
    # 计算选择到最佳动作的数量，为算法对比做准备
    best_action_counts = np.zeros(rewards.shape)
    # 在每个实验组中循环
    for i, bandit in enumerate(bandits):
        # 轮询实验次数
        for r in trange(runs):
            # 重置环境
            bandit.reset()
            # 在实验的时间步内循环
            for t in range(time):
                # 进行动作选择
                action = bandit.act()
                # 在环境中执行动作后，获取回报
                reward = bandit.step(action)
                # 结果存储
                rewards[i, r, t] = reward
                # 统计在哪些时刻agent选择了最优的动作
                if action == bandit.best_action:
                    best_action_counts[i, r, t] = 1
```

```
    # 计算每个Bandit平均选到最佳动作的概率
    mean_best_action_counts = best_action_counts.mean(axis=1)
    # 计算每个Bandit收到的平均回报
    mean_rewards = rewards.mean(axis=1)
    return mean_best_action_counts, mean_rewards
```

4、测试平台可视化

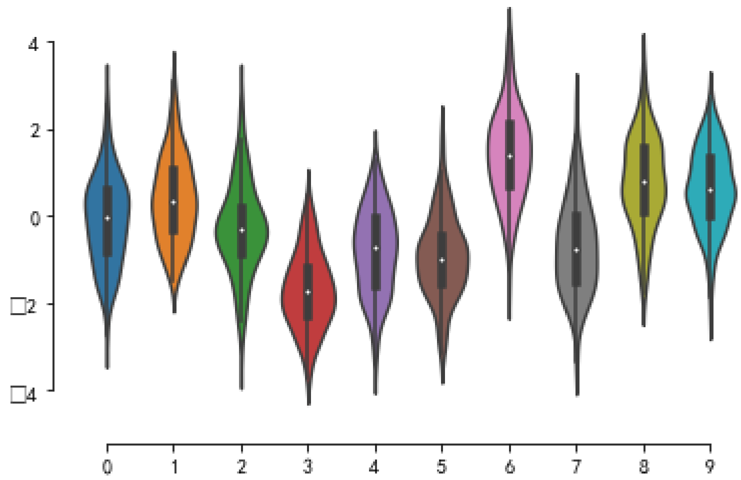
该次实验的10臂多搏击问题，具有10个动作，每个动作的真实值$q_*(a)$分别从一个零均值单位方差的正态分布中生成，然后实际的收益从均值为$q_*(a)$且为单位方差的正态分布中生成，下面代码可视化该过程的小提琴图(Violin Plot)。

```
def figure_2_1():
    plt.violinplot(dataset=np.random.randn(200, 10) + np.random.randn(10))
    plt.xlabel("动作")
    plt.ylabel("收益分布")
    plt.show()
figure_2_1()
```



当然还可以用seaborn画出来:

```
dataset=np.random.randn(200, 10) + np.random.randn(10)
f, ax = plt.subplots()
sns.violinplot(data=dataset)
sns.despine(offset=10, trim=True)
```



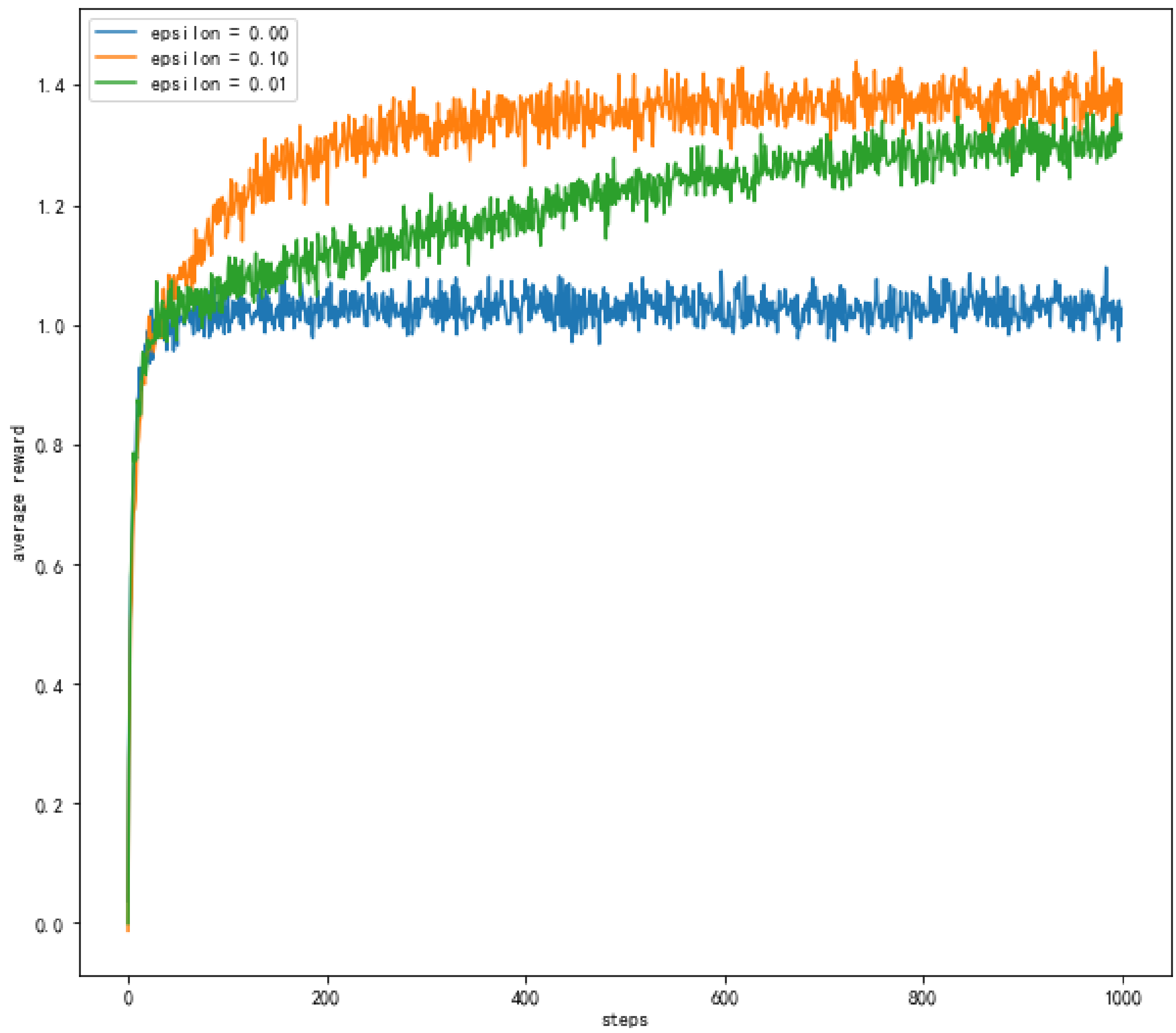
5、贪心算法与$\epsilon - greedy$算法比较

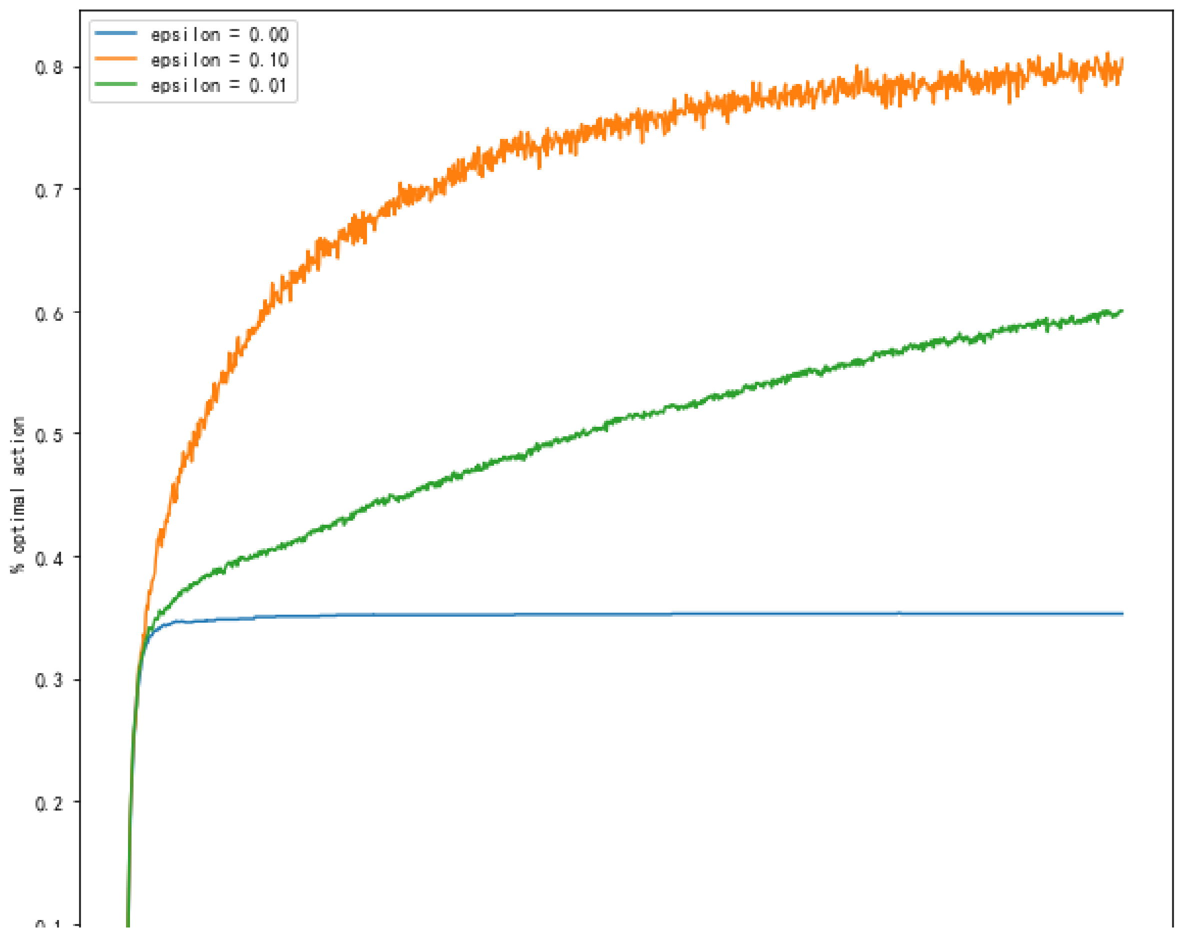
下面我们会比较贪心算法（$\epsilon = 0$），$\epsilon = 0.01$的$\epsilon - greedy$以及$\epsilon = 0.1$的$\epsilon - greedy$这三种方法。这里对2000次不同的10臂Bandit验收进行了平均，所有的方法都采用采样平均作为对动作价值的估计。

具体代码如下：

```
def figure_2_2(runs=2000, time=1000):
    # 实验的三种epsilon场景
    epsilons = [0, 0.1, 0.01]
    # 这里用了一个list存储了三个相同参数的Bandit（除了epsilon），方便下面迭代对比
    bandits = [Bandit(epsilon=eps, sample_averages=True) for eps in epsilons]
    best_action_counts, rewards = simulate(runs, time, bandits)

    plt.figure(figsize=(10, 20))

    plt.subplot(2, 1, 1)
    for eps, rewards in zip(epsilons, rewards):
        plt.plot(rewards, label='epsilon = %.02f' % (eps))
    plt.xlabel('steps')
    plt.ylabel('average reward')
    plt.legend()

    plt.subplot(2, 1, 2)
    for eps, counts in zip(epsilons, best_action_counts):
        plt.plot(counts, label='epsilon = %.02f' % (eps))
    plt.xlabel('steps')
    plt.ylabel('% optimal action')
    plt.legend()
    plt.show()

figure_2_2()
```

实验结果如下：


epsilon = 0.00
epsilon = 0.10
epsilon = 0.01
average reward
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
0
200
400
600
800
1000
steps


epsilon = 0.00
epsilon = 0.10
epsilon = 0.01
% optimal action
0.8
0.7
0.6
0.5
0.4
0.3
0.2

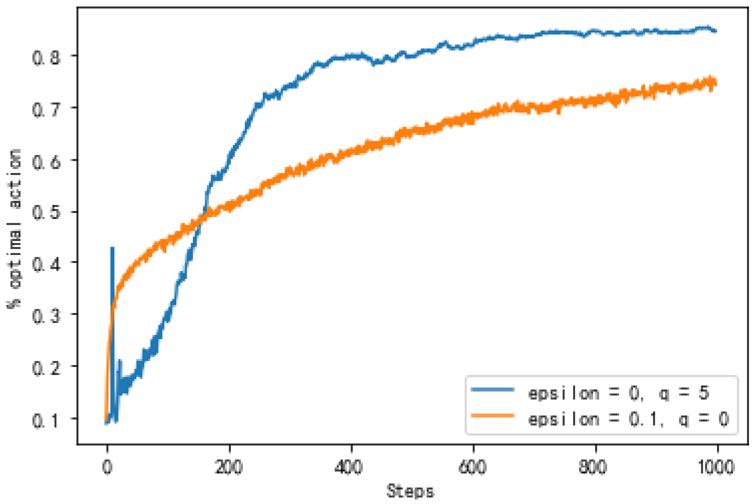
这里所有方法都用采样平均策略来形成对动作价值的估计。上部的图显示了期望的收益随着经验的增长而增长。贪心方法在最初增长得略微快一些，但是随后稳定在一个较低的水平。相对于在这个测试平台上最好的可能收益1.55，这个方法每时刻只获得了大约1的收益。从长远来看，贪心的方法表现明显更糟，因为它经常陷入执行次优的动作的怪圈。下部的图显示贪心方法只在大约三分之一的任务中找到最优的动作。在另外三分之二的动作中，最初采样得到的动作非常不好，贪心方法无法跳出来找到最优的动作。$\epsilon-greedy$方法最终表现更好，因为它们持续地试探并且提升找到最优动作的机会。$\epsilon=0.1$的方法试探得更多，通常更早发现最优的动作，但是在每时刻选择这个最优动作的概率却永远不会超过91%（因为要在$\epsilon=0.1$的情况下试探）。$\epsilon=0.01$的方法改善得更慢，但是在图中的两种测度下，最终的性能表现都会比$\epsilon=0.1$的方法更好。为了充分利用高和低的c值的优势，随着时刻的推移来逐步减小ϵ也是可以的。

6. 乐观初始值测试

该次实验会比较初始值设为$Q_1(a)=5$与初始值为$Q_1(a)=0$两种情况的对比。

```
def figure_2_3(runs=2000, time=1000):
    bandits = []
    # 实验对照组
    bandits.append(Bandit(epsilon=0, initial=5, step_size=0.1))
    bandits.append(Bandit(epsilon=0.1, initial=0, step_size=0.1))
    best_action_counts, _ = simulate(runs, time, bandits)

    plt.plot(best_action_counts[0], label='epsilon = 0, q = 5')
    plt.plot(best_action_counts[1], label='epsilon = 0.1, q = 0')
    plt.xlabel('Steps')
    plt.ylabel('% optimal action')
    plt.legend()
    plt.show()

figure_2_3()
```

实验结果如下图所示，刚开始乐观初始值表现很差，这是因为他需要更多的试探次数，随着时间的推移，实验次数减少，乐观初始值表现得更好。

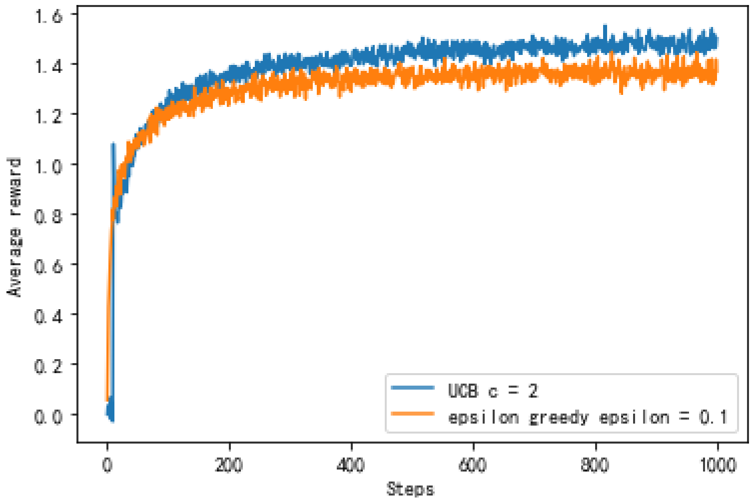


7、UCB测试

下面将比较UCB算法在10臂测试平台上的平均表现

```
def figure_2_4(runs=2000, time=1000):
    bandits = []
    bandits.append(Bandit(epsilon=0, UCB_param=2, sample_averages=True))
    bandits.append(Bandit(epsilon=0.1, sample_averages=True))
    _, average_rewards = simulate(runs, time, bandits)

    plt.plot(average_rewards[0], label='UCB c = 2')
    plt.plot(average_rewards[1], label='epsilon greedy epsilon = 0.1')
    plt.xlabel('Steps')
    plt.ylabel('Average reward')
    plt.legend()
    plt.show()
figure_2_4()
```

实验结果如下，除了刚开始的k步随机选择让位尝试过的动作外，在一般情况下，UCB算法会比$\epsilon - greedy$算法表现更好。



8、梯度Bandit算法测试

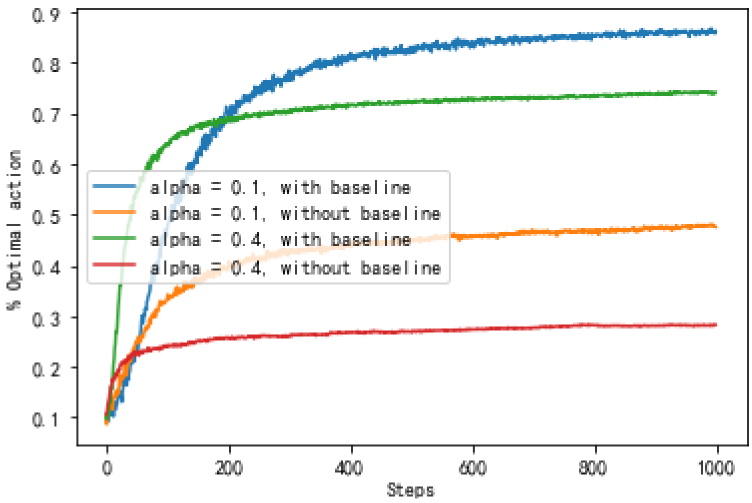
下面会比较四种情况的算法效果：

- 包含baseline，并且$\alpha = 0.1$
- 包含baseline，并且$\alpha = 0.4$
- 不包含baseline，并且$\alpha = 0.1$
- 不包含baseline，并且$\alpha = 0.4$

```
def figure_2_5(runs=2000, time=1000):
    bandits = []
    bandits.append(Bandit(gradient=True, step_size=0.1,
gradient_baseline=True, true_reward=4))
    bandits.append(Bandit(gradient=True, step_size=0.1,
gradient_baseline=False, true_reward=4))
    bandits.append(Bandit(gradient=True, step_size=0.4,
gradient_baseline=True, true_reward=4))
    bandits.append(Bandit(gradient=True, step_size=0.4,
gradient_baseline=False, true_reward=4))
    best_action_counts, _ = simulate(runs, time, bandits)
    labels = ['alpha = 0.1, with baseline',
              'alpha = 0.1, without baseline',
              'alpha = 0.4, with baseline',
              'alpha = 0.4, without baseline']
```

```

    for i in range(len(bandits)):
        plt.plot(best_action_counts[i], label=labels[i])
    plt.xlabel('Steps')
    plt.ylabel('% Optimal action')
    plt.legend()
    plt.show()
figure_2_5()
```

实验结果如下，其中可以看到包含baseline的算法表现更好。



9、多算法融合比较

最后将比较四种算法的融合比较，包括：

- $\epsilon - greedy$算法
- 乐观初始值算法
- UCB算法
- 梯度Bandit算法

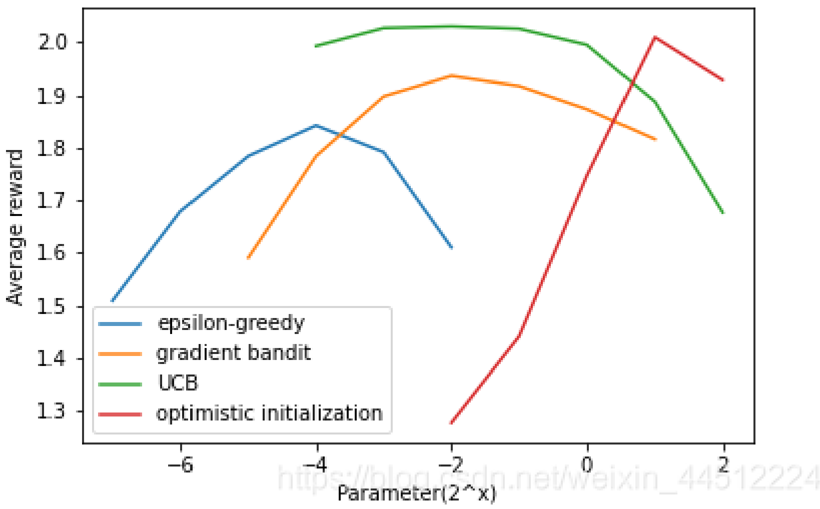
对比过程代码如下：

```
def figure_2_6(runs=2000, time=1000):
    labels = ['epsilon-greedy', 'gradient bandit',
              'UCB', 'optimistic initialization']
    generators = [lambda epsilon: Bandit(epsilon=epsilon, sample_averages=True),
                  lambda alpha: Bandit(gradient=True, step_size=alpha, gradient_baseline=True),
                  lambda coef: Bandit(epsilon=0, UCB_param=coef, sample_averages=True),
                  lambda initial: Bandit(epsilon=0, initial=initial, step_size=0.1)]
    parameters = [np.arange(-7, -1, dtype=np.float),
                  np.arange(-5, 2, dtype=np.float),
                  np.arange(-4, 3, dtype=np.float),
                  np.arange(-2, 3, dtype=np.float)]

    bandits = []
    for generator, parameter in zip(generators, parameters):
        for param in parameter:
            bandits.append(generator(pow(2, param)))
```

```
    _, average_rewards = simulate(runs, time, bandits)
    rewards = np.mean(average_rewards, axis=1)

    i = 0
    for label, parameter in zip(labels, parameters):
        l = len(parameter)
        plt.plot(parameter, rewards[i:i+l], label=label)
        i += l
    plt.xlabel('Parameter(2^x)')
    plt.ylabel('Average reward')
    plt.legend()
    plt.show()
figure_2_6()
```

对比结果如下，这张图分别给出了每一种算法及参数随时间推移的学习曲线。这个曲线中展示了每种算法和参数超过1000步的平均收益值，这个值与学习曲线下的面积成正比。这种图叫参数研究图，每个算法的性能都呈现为倒U形，所有算法在其参数的中间值表现最好，参数既不能太大也不能太小。

在评估一种方法时，我们不仅要关注它在最佳参数设置上的表现，还要注意它对参数值的敏感性。所有这些算法都是相当不敏感的，它们在一系列的参数值上表现得很好，这些参数值的大小是一个数量级的。总的来说，在这个问题上，UCB似乎表现最好。



https://blog.csdn.net/weixin_44512224