第 I 部分 表格型求解方法

第2章：多臂赌博机

2.1 一个 k 臂赌博机问题

赌博机问题的原始形式：你要重复的在 k 个选项或动作中进行选择。每次做出选择之后，你都会的=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决定。你的目标是在某一段时间内最大化总收益的期望，比方说，1000次选择或者1000时刻之后。

在 k 臂赌博机问题中， 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，我们称为这个动作的"价值"。我们将在时刻 t 时选择的动作记作 A_t ,并将对应的收益记作 R_t 。任一动作 a 对应的价值，记作 $q_*(a)$.

$$q_*(a) \doteq \mathbb{E}[R_t|A_t = a].$$

当我们知道每个动作的价值，则解决 k 臂赌博机问题就很简单：每次都选择价值最高的动作。我们假设你不能确切的知道每个动作的价值，但是你可以进行估计。我们将对动作 a 在时刻 t 时的价值的估计记作 $Q_t(a)$,我们希望它接近 $q_*(a)$.

我们持续的对动作的价值进行估计，那么在任意时刻都会至少有一个动作的估计价值是最高的，我们把这些估计价值最高的动作称为**贪心**的动作。当从这些"贪心的动作"中选择行为时，我们称此为**开发**当前你所知道的关于动作价值的知识。如果不是如此，而是选择非贪心的动作，我们称此为**试探**。因为是价值的估计值，难免会存在一些误差，试探操作会改善对非贪心动作（估计的非贪心动作实际上可能是价值最高的动作）的价值估计，减少由不确定性带来的误差。值得一提的是，在同一次动作选择中，开发和试探是不可能同时进行的，这种情况就是我们常常提到的开发和试探之间的冲突。

在一个具体的案例中，到底选择"试探"还是"开发"一种复杂的方式依赖于我们得到的函数估计、不确定性和剩余时间的精确数值。目前有很多复杂的方法去平衡两者，但大多需要对平稳情况和先验知识做出很强的假设。接下来会给出几个针对 k 臂赌博机问题的非常简单的平衡方法。

2.2 动作-价值方法

我们使用这些价值的估计来进行动作的选择，这一类方法被统称为"动作-价值方法"。

$$Q_t(a) \doteq \frac{t\text{时刻前通过执行动作}a\text{得到的收益总和}}{t\text{时刻前执行动作}a\text{的次数}} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{I}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{I}_{A_i=a}} \quad (2.1)$$

其中，$\mathbb{I}_{predicate}$ 表示随机变量，当 $predict$ 为真时其值为1，反之为0。当分母为0时，我们将 $Q_t(a)$ 定义为某个默认值，比如 $Q_t(a) = 0$ 。当分母趋向无穷大时，根据大数定理 $Q_t(a)$ 会收敛到 $q_*(a)$。我们将这种估计动作价值的方法称为采样平均法。

接下来采用最简单的动作选择规则，即选择具有最高估计值的动作。如果有多个贪心动作，那就任意选择一个，比如随机挑选。记作

$$A_t \doteq \underset{a}{argmax} Q_t(a) \qquad (2.2)$$

这种贪心的策略大部分时间都表现得贪心，但偶尔（比如以一个很小的概率 ϵ ）以独立于动作-价值估计值的方式从所有动作中等概率随机的做出选择，我们将使用这种近乎贪心的选择规则的方法称为 ϵ -贪心方法。这个算法的问题是它永远无法进行确定性的选择，选择最优动作的概率最后会收敛到 $1-\epsilon$ 。

2.3 10 臂测试平台

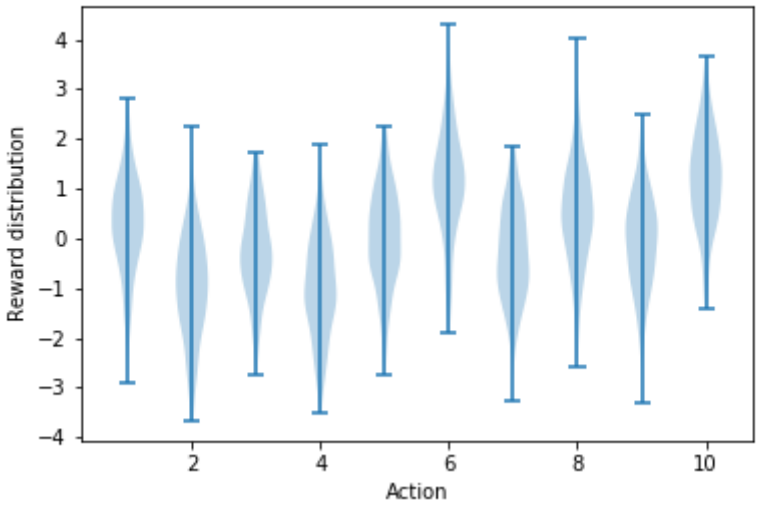
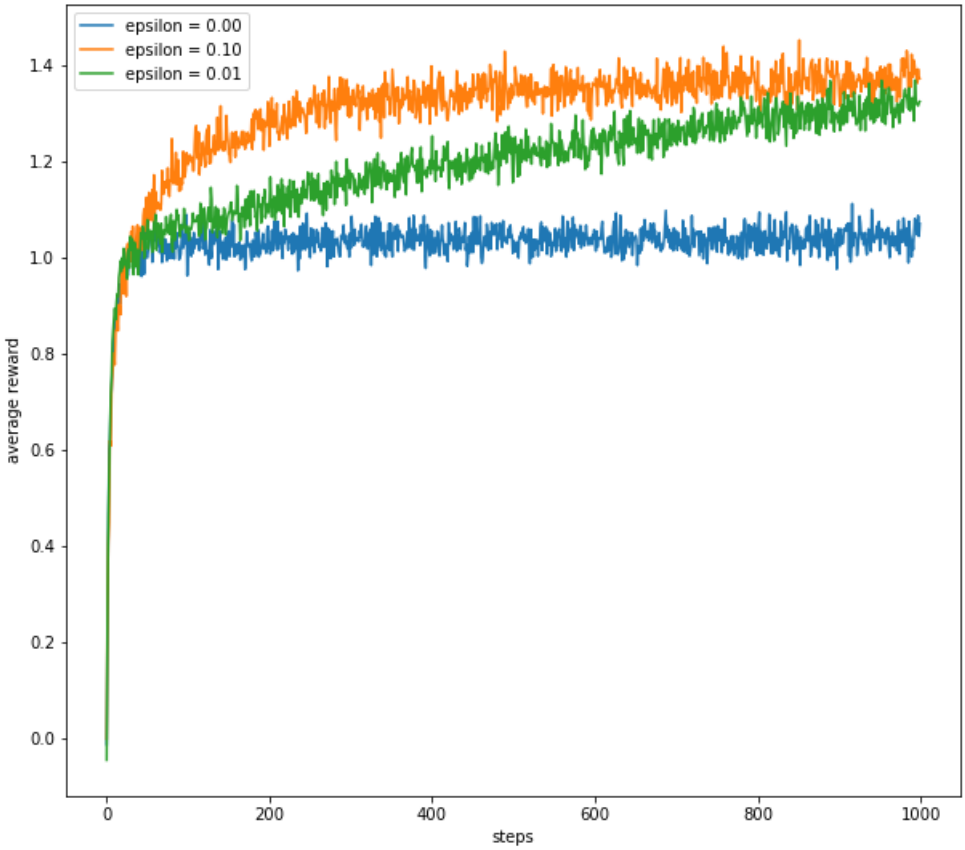


图2.1 10 臂赌博机的问题平台

如图 2.1 显示的那样，动作的真实价值为 $q_*(a), a = 1, \ldots, 10$。从一个均值为 0 方差为 1 的标准正态（高斯）分布中选择，这些分布显示为蓝色区域。当对应于该问题的学习方法在时刻 t 选择 A_t 时，实际收益 R_t 则由一个均值为 $q_*(A_t)$ 方差为 1 的正态分布决定。



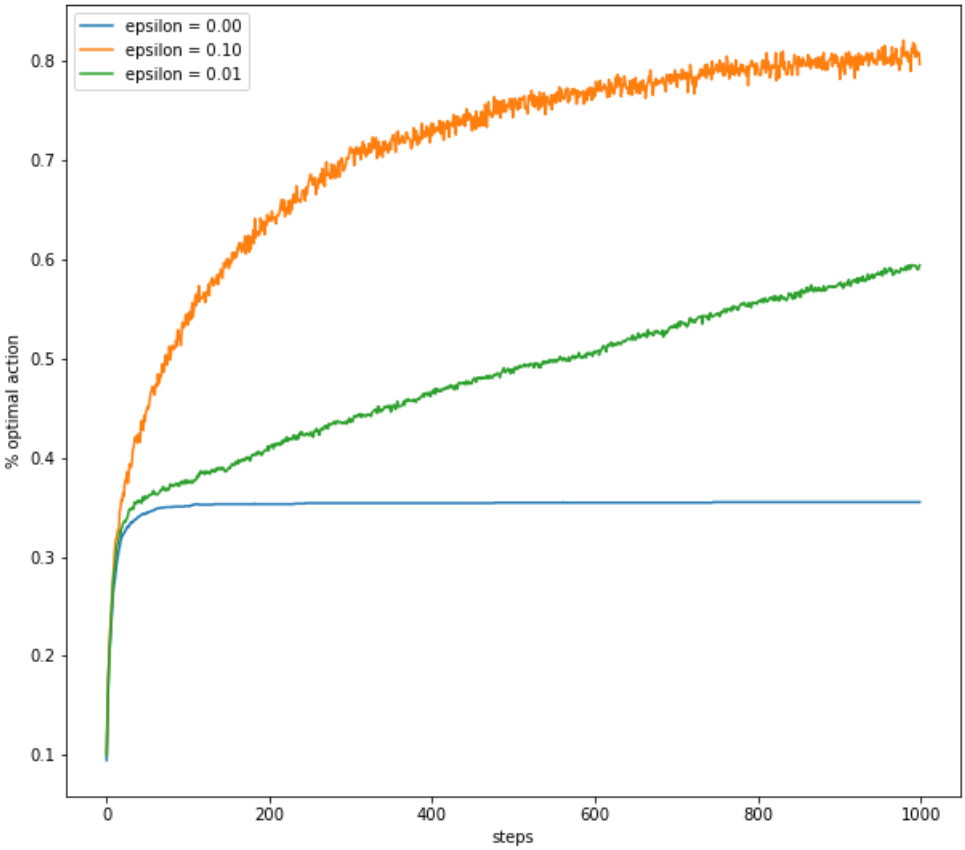


图2.2 “动作-价值”方法在 10 臂赌博机的问题平台上的平均表现

很明显，在该问题平台上加入参数 ϵ 的算法表现得更好，但是 ϵ-贪心算法的问题是当 ϵ 过大会导致最后收敛的概率过低，而 ϵ 若是过小的话又会导致收敛速度过慢，所以设置一个随着时刻可变的 ϵ 值是一个非常好的方法。

2.4 增量式实现

令 R_i 表示这一动作被选择 i 次获得的收益，Q_n 表示被选择 $n-1$ 次后它的估计的动作价值，可以简单地把它写为

$$Q_n \doteq \frac{R_1 + R_2 + \cdots + R_{n-1}}{n-1}$$

这种简明的实现需要维护所有收益的记录，由于已知的收益越来越多，内存和计算量也会随着时间增长。这样对资源的消耗过大。很显然，这并不是必要的，我们很容易设计增量式公式以小而恒定的计算来更新平均值。给定 Q_n 和第 n 次的收益 R_n ，所有 n 个收益的新的均值可以这样计算

$$\begin{aligned}
Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\
&= \frac{1}{n}(R_n + \sum_{i=1}^{n-1} R_i) \\
&= \frac{1}{n}(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i) \\
&= \frac{1}{n}(R_n + (n-1)Q_n) \\
&= \frac{1}{n}(R_n + nQ_n - Q_n) \\
&= Q_n + \frac{1}{n}[R_n - Q_n] \qquad (2.3)
\end{aligned}$$

对于每一个新的收益，这种实现只需要存储 Q_n 和 n 即可。

更新公式 (2.3) 的形式会贯穿本书始终。它的一般形式是：

$$\text{新估计值} \leftarrow \text{旧估计值} + \text{步长} \times \left[\text{目标} - \text{旧估计值}\right] \quad (2.4)$$

一个完整的使用以增量式计算的样本均值和 ϵ -贪心动作选择的赌博机问题算法的伪代码如下面的框中所示。假设函数 $bandit(a)$ 表示接受一个动作作为参数并且返回一个对应的收益。

一个简单的多臂赌博机算法

初始化，令 $a = 1$ 到 k ：
$Q_a \leftarrow 0$
$N(a) \leftarrow 0$
无限循环：
$A \leftarrow \begin{cases} argmax_a Q(a) & \text{以} 1-\epsilon\text{概率（随机跳出贪心）} \\ \text{一个随机的动作} & \text{以}\epsilon\text{概率} \end{cases}$
$R \leftarrow bandit(A)$
$N(A) \leftarrow N(A) + 1$
$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}[R - Q(A)]$

2.5 跟踪一个非平稳过程

到目前为止我们讨论的取平均的方法对平稳的赌博机问题是合适的，即收益的概率分布不随着时间得变化。但是如果赌博机问题的收益概率是随着时间变化的，也就是我们经常遇到的非平稳的强化学习问题。在这种情形下，给**近期的收益赋予比过去很久的收益更高的权值**就是一种合理的处理方式。比如说，用于更新 $n-1$ 个过去的收益的均值 Q_n 的增量更新规则 (2.3) 可以改为

$$Q_n \doteq Q_n + \alpha[R_n - Q_n] \quad (2.5)$$

式中，步长参数 $\alpha \in (0,1]$ 是一个常数。这使得 Q_{n+1} 成为对过去的收益和初始的估计 Q_1 的加权平均（证明过程过于简单，略）。因为权值以指数形式递减，所以这个方法有时候也被称为**“指数近因加权平均”**。

有时候随着时刻一步步改变步长参数是很方便的。设 $\alpha_n(a)$ 表示用于处理第 n 次选择动作 a 后收到的收益的步长参数。随机逼近理论中的 一个著名结果给出了保证收敛概率为 1 所需的条件

$$\sum_{n=1}^{\infty} \alpha_n(a) = \infty \quad \text{且} \quad \sum_{n=1}^{\infty} \alpha_n^2(a) < \infty \quad (2.6)$$

上式说明，收敛性不能保证对任何 $\{\alpha_n(a)\}$ 序列都满足。选择 $\alpha_n(a) = \frac{1}{n}$ 将会得到采样平均法，大数定理（或调和级数）保证它的收敛概率为 1 。但是常数步长 $\alpha_n(a) = \alpha$ 却不能保证收敛概率为 1 ，也就是说估计永远无法完全收敛，它会随着最近得到的收益变化，这也是我们在非平稳环境中所需要的特性。值得注意的是，符合条件 (2.6) 的步长参数序列常常收敛的很慢，或者需要大量的调试才能得到一个满意的收敛率，所以这类方法也只在理论证明中较常出现，在实际应用和实验研究中很少用到。

2.6 乐观初始值

目前为止我们讨论的所有方法都在一定程度上依赖于初始动作值 $Q_1(a)$ 的选择。从统计学角度来说，这些方法（由于初始估计值）是有偏的。对于采样平均法来說，当所有动作都至少被选择一次时，偏差就会消失。而对于步长为常数 α 的情况来说，偏差不会消失，但是会随着时间减小。在实际中，这种偏差通常不是一个问题，有时甚至会有好处。我们可以在可选择的参数集中选择初始估计值从而简单地设置关于预期收益水平的先验知识。

初始动作的价值同时也提供了一种简单的试探方法。还是从 10 臂的测试平台来看，我们替换掉原先的初始值 0 ，将他们全部设为 $+5$ 。注意，如前所述，在这个问题中，$q_*(a)$ 是按照均值为 0 方差为 1 的正态分布选择的。因为 $+5$ 的初始值是一个过度乐观的估计。但是这种乐观的初始估计却会鼓励动作-价值方法去试探。因为无论哪一种动作被选择，收益都比开始的估计值要小；因此学习器会对得到的收益感到“失望”，从而转向另一个动作。这就导致所有动作在估计值收敛之前都被尝试了好几次。我们把这种鼓励试探的技术叫做**乐观初始价值**。需要注意的是，这个简单的技巧，在平稳问题中非常有效，但是不太适合非平稳过程，因为它试探的驱动力天生是暂时的。

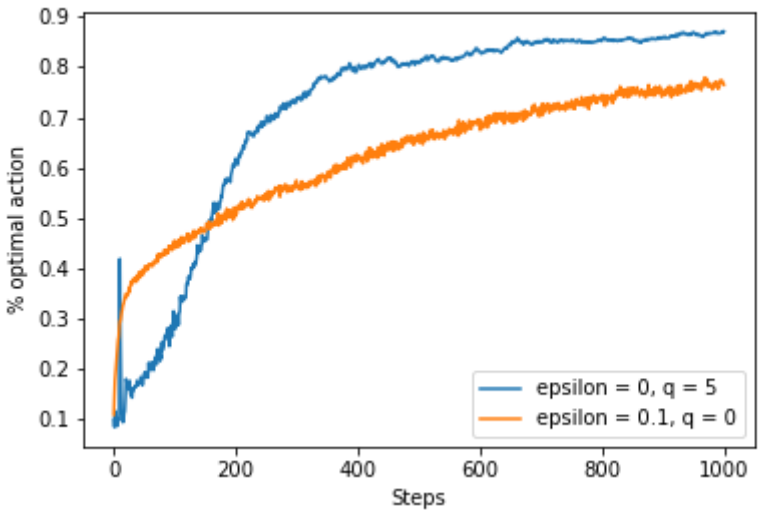


图2.3 乐观的初始“动作-价值”方法在 10 臂赌博机的问题平台上的运行结果

从图 2.3 可以看出，刚开始乐观初始化表现得比较糟糕，因为它需要试探的更多次，但是随着时间的推移，试探的次数减少，它的表现也变得更好。

2.7 基于置信度上界的动作选择

对动作-价值的估计总会存在不确定性，所以试探是必须的。贪心动作虽然在当时时刻看起来最好，但实际上其他动作可能从长远看更好。而 ϵ - 贪心算法虽然会尝试选择非贪心的动作，但那也只是一种盲目的选择，因为它不太会选择那种接近贪心的动作或者不确定性特别大的动作。在非贪心动作中，最好根据他们的潜力来选择可能事实上是最优的动作。一个有效的方法是按照以下公式选择动作

$$A_t \doteq \underset{a}{argmax}[Q_t(a) + c\sqrt{\frac{lnt}{N_t(a)}}] \quad (2.7)$$

在这里简述一下置信度上界（UCB）的动作选择思想，在这里我们把式子中的平方根项称为对 a 动作值估计的不确定性的度量，参数 c 决定了置信水平，接下来我们把这一项叫做不确定性度量项。不确定性度量项会随着 $N_t(a)$ 的增大而减小；也会随着 t 的推移而增加。这对于一个动作来说，如果它随着时间的推移一直没有被选到，那么它的不确定性就会变得越来越大（由于是关于时间的对数函数，所以变化的幅度会越来越小，但是还是会无限的变大下去），这也就意味着它终将会被选到。总之，整体价值越大，被选到的次数就会越多，这满足了我们对算法的期望（根据动作的潜力来试探动作，而不是简单地选择贪心动作）。

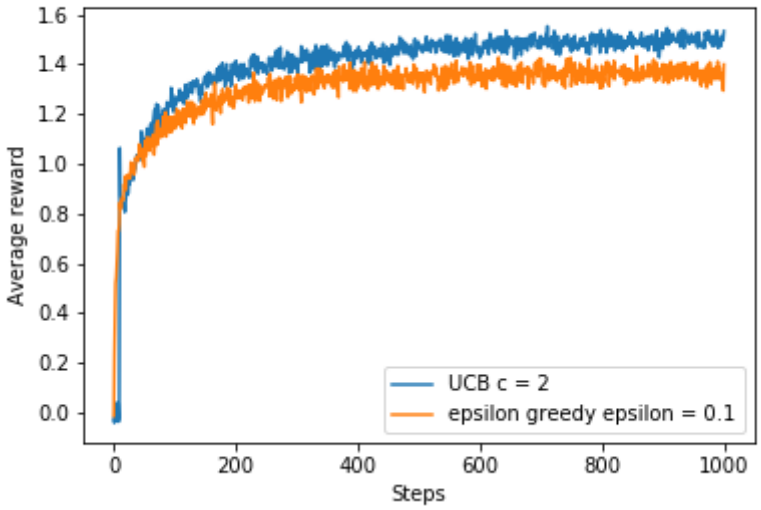


图2.4 UCB算法在 10 臂赌博机的问题平台上的平均表现

如图所示，UCB算法往往会表现很好，但是很难推广到一般的强化学习方法中。在一些更复杂的问题中，目前还没有已知的实用方法利用UCB动作选择的思想。

2.8 梯度赌博机算法

针对每一个动作 a 考虑学习一个数值化的偏好函数 $H_t(a)$。偏好函数越大，动作就越频繁的被选择。它表示的是一个相对的概念。如果我们给每一个动作的偏好函数都加上 1000 ，那么对于按照如下 $softmax$ 分布（吉布斯或玻尔兹曼分布）确定的动作概率没有任何影响。

$$Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a) \quad (2.8)$$

其中，$\pi_t(a)$ 表示动作 a 在时刻 t 时被选择的概率。所有偏好函数的初始值都是一样的（例如，$H_1(a) = 0, \forall a$），所以每个动作被选择的概率是相同的。

在选择动作 A_t 并获得收益 R_t 之后，偏好函数将按照如下方式更新

$$\begin{aligned} H_{t+1}(A_t) &\doteq H_t(A_t) + \alpha(R_t - \bar{R}_t)(1 - \pi_t(a)) \quad else \\ H_{t+1}(a) &\doteq H_t(a) - \alpha(R_t - \bar{R}_t)\pi_t(a) \qquad\quad a \neq A_t \end{aligned} \quad (2.9)$$

注意，两个式子中的 R_t 均是指选择 A_t 动作时，所对应的即时收益。$\bar{R}_t$ 是指在时刻 t 内所有收益的平均值。$\bar{R}_t$ 项作为基准，如果收益高于它，那么未来选择动作 A_t 的概率就会增加，反之概率会降低。而未选择的动作变化与之相反。

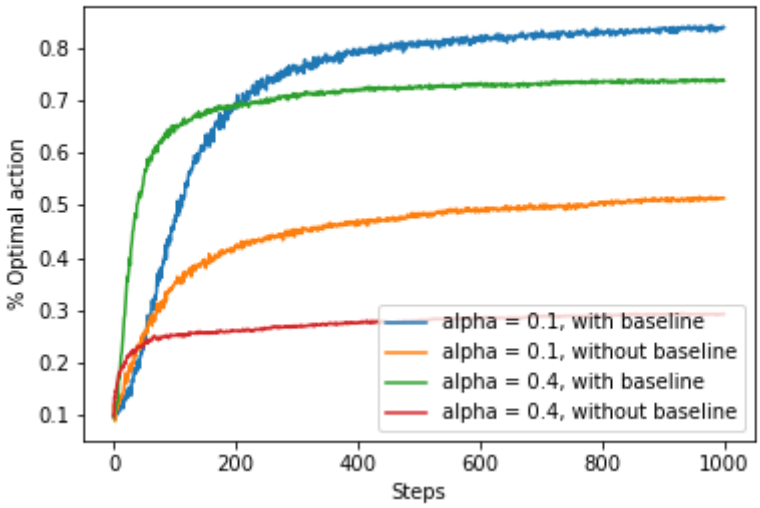


图2.5 含收益基准项与不含收益基准项的梯度赌博机算法在 10 臂赌博机的问题平台上的平均表现

通过随机梯度上升实现梯度赌博算法

在精确地梯度上升算法中

$$H_{t+1}(a) \doteq H_t(a) + \alpha \frac{\delta \mathbb{E}[R_t]}{\delta H_t(a)} \quad (2.10)$$

其中

$$\mathbb{E}[R_t] = \sum_x \pi_t(x) q_*(x)$$

经过一系列推导得到

$$H_{t+1}(a) \doteq H_t(a) + \alpha(R_t - \bar{R}_t)(\mathbb{I}_{\mathrm{a}=\mathbb{A}_{\mathrm{t}}} - \pi_{\mathrm{t}}(\mathrm{a})) \quad \forall a$$

其中，$\mathbb{I}_{a=A_t}$ 表示如果$a = A_t$ 就取 1 ，否则取 0 。可以发现上式和原始算法中的式子 2.9 是一致的。

2.9 关联搜索（上下文相关的赌博机）

本章目前为止，还没有将不同的动作与不同的情景联系起来。然而，在一般的强化学习任务中，往往有不止一个情景，他们的目标是学习一种策略：一个从特定情境到最优动作的映射。例如，用我们所看到的颜色作为信号，把每个任务和该任务下最优的动作直接关联起来，比如，如果为红色，则选择 1 号臂；如果为绿色，则选择 2 号臂。这便是与任务相关的策略。

如果在关联搜索的基础上再加上允许动作可以影响下一时刻的情景和收益这个条件，这便是一个完整的强化学习问题。这也是我们以后的研究重点。

2.10 本章小结（附源码）

本章的提到的算法虽然简单，但是在我们看来，它们已经被公认为最先进的技术，我们接下来提出的解决完整的强化学习问题的学习方法，它们也都部分的使用了本章的处理方法。

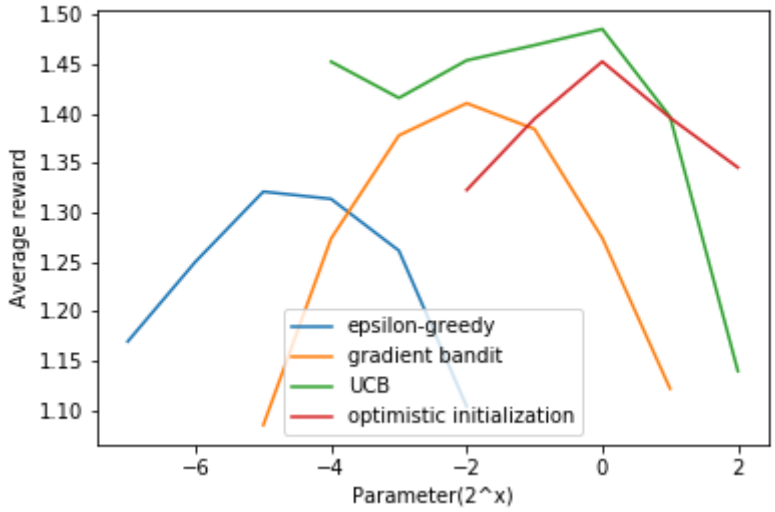


图2.6 本章中不同赌博机算法的参数调整研究

因为每种算法都有自己不同的参数，为了便于观察，上图中的横坐标用单一的尺度显示了所有的参数。这些算法对参数值的变化都不是很敏感。总的来说，在这个问题上，UCB 算法似乎表现得最好。

10 臂赌博机问题平台搭建的源码（摘自GitHub）如下：

```
# -*- coding: utf-8 -*-
"""
Created on Tue Oct 29 15:13:34 2019

@author: YUYU
"""

import matplotlib
import matplotlib.pyplot as plt
import numpy as np
from tqdm import trange

matplotlib.use('Agg')


class Bandit:
    # @k_arm: # of arms
    # @epsilon: probability for exploration in epsilon-greedy algorithm
    # @initial: initial estimation for each action
    # @step_size: constant step size for updating estimations
    # @sample_averages: if True, use sample averages to update estimations
instead of constant step size
    # @UCB_param: if not None, use UCB algorithm to select action
    # @gradient: if True, use gradient based bandit algorithm
    # @gradient_baseline: if True, use average reward as baseline for gradient
based bandit algorithm
    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
        self.step_size = step_size
        self.sample_averages = sample_averages
        self.indices = np.arange(self.k)
        self.time = 0
```

```
        self.UCB_param = UCB_param
        self.gradient = gradient
        self.gradient_baseline = gradient_baseline
        self.average_reward = 0
        self.true_reward = true_reward
        self.epsilon = epsilon
        self.initial = initial

    def reset(self):
        # real reward for each action
        self.q_true = np.random.randn(self.k) + self.true_reward

        # estimation for each action
        self.q_estimation = np.zeros(self.k) + self.initial

        # # of chosen times for each action
        self.action_count = np.zeros(self.k)

        self.best_action = np.argmax(self.q_true)

        self.time = 0

    # get an action for this bandit
    def act(self):
        if np.random.rand() < self.epsilon:
            return np.random.choice(self.indices)

        if self.UCB_param is not None:
            UCB_estimation = self.q_estimation + \
                self.UCB_param * np.sqrt(np.log(self.time + 1) /
(self.action_count + 1e-5))
            q_best = np.max(UCB_estimation)
            return np.random.choice(np.where(UCB_estimation == q_best)[0])

        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)

        q_best = np.max(self.q_estimation)
        return np.random.choice(np.where(self.q_estimation == q_best)[0])

    # take an action, update estimation for this action
    def step(self, action):
        # generate the reward under N(real reward, 1)
        reward = np.random.randn() + self.q_true[action]
        self.time += 1
        self.action_count[action] += 1
        self.average_reward += (reward - self.average_reward) / self.time

        if self.sample_averages:
            # update estimation using sample averages
            self.q_estimation[action] += (reward - self.q_estimation[action]) /
self.action_count[action]
        elif self.gradient:
            one_hot = np.zeros(self.k)
            one_hot[action] = 1
            if self.gradient_baseline:
```

```
                baseline = self.average_reward
            else:
                baseline = 0
            self.q_estimation += self.step_size * (reward - baseline) * (one_hot
- self.action_prob)
        else:
            # update estimation with constant step size
            self.q_estimation[action] += self.step_size * (reward -
self.q_estimation[action])
        return reward


def simulate(runs, time, bandits):
    rewards = np.zeros((len(bandits), runs, time))
    best_action_counts = np.zeros(rewards.shape)
    for i, bandit in enumerate(bandits):
        for r in trange(runs):
            bandit.reset()
            for t in range(time):
                action = bandit.act()
                reward = bandit.step(action)
                rewards[i, r, t] = reward
                if action == bandit.best_action:
                    best_action_counts[i, r, t] = 1
    mean_best_action_counts = best_action_counts.mean(axis=1)
    mean_rewards = rewards.mean(axis=1)
    return mean_best_action_counts, mean_rewards


def figure_2_1():
    plt.violinplot(dataset=np.random.randn(200, 10) + np.random.randn(10))
    plt.xlabel("Action")
    plt.ylabel("Reward distribution")
    plt.savefig('figure_2_1.png')
    plt.close()


def figure_2_2(runs=2000, time=1000):
    epsilons = [0, 0.1, 0.01]
    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.savefig('figure_2_2.png')
    plt.close()


def figure_2_3(runs=2000, time=1000):
    bandits = []
    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.savefig('figure_2_3.png')
    plt.close()


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.savefig('figure_2_4.png')
    plt.close()


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.savefig('figure_2_5.png')
    plt.close()


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.savefig('figure_2_6.png')
    plt.close()


if __name__ == '__main__':
    figure_2_1()
    figure_2_2()
    figure_2_3()
    figure_2_4()
    figure_2_5()
    figure_2_6()
```