强化学习：第三次作业

编程实现多臂赌博机的其他两个探索-利用平衡的策略

初始框架

我重构了`multi-arms.py`，将ε-贪心策略、ucb策略、波尔兹曼策略三种动作选择策略整合在`KGamble`类中，具体代码请查阅`k_gamble.py`文件。

开发环境

- ubuntu 18.04，i7-7700，16GB RAM，GeForce GTX 1080Ti
- python==3.6.6
- gym==0.10.8

框架介绍

`KGamble`类中的主要函数如下：

- `__init__()`：多臂赌博机环境的初始化（如动作的均值/标准差）
- `step(action_idx)`：在给定选择的动作后，给出该动作在该时步的奖赏
- `action_strategy(strategy, **kwargs)`：在给定动作选择的策略及相关策略参数（ε/c/α）的前提下，返回该时步的动作选择

- `update_q(strategy, action_idx, reward)`：更新值函数$Q(a)$
- `update_h_with_pi(action_idx, reward, a_ratio, step)`：更新玻尔兹曼策略中的动作偏好值$H(a)$以及在某个时步`step`时采取动作`action_idx`的可能性$\pi(a)$
- `train(steps, strategy, **kwargs)`：将上述函数封装起来，分别实现多种动作选择策略下的学习逻辑
- `reset()`：重置多臂赌博机环境
- `plot(self, colors, strategy)`：依据值函数绘制当前动作选择策略的学习情况

ε-贪心策略

动作选择

利用`np.random.uniform(0, 1)`和`e_ratio`参数来决定选择的动作是按照最大$Q(a)$值对应的动作还是随机选择的动作。

```
if strategy == 'e_greedy':
    e_ratio = kwargs['e_ratio']
    if np.random.uniform(0, 1) > e_ratio:
        action_idx = np.argmax(self.q_values)
    else:
        np.random.randint(len(self.actions))
    return action_idx
```

学习逻辑

确定动作，计算奖赏，更新$Q(a)$值

```
if strategy == 'e_greedy':
    action_idx = self.action_strategy(strategy, e_ratio=kwargs['e_ratio'])
    reward = self.step(action_idx)
    self.update_q(strategy, action_idx, reward)
```

UCB策略

动作选择

根据公式$A_t = argmax[Q_t(a) + c\sqrt{\frac{lnt}{N_t(a)}}]$来计算动作选择，需要注意如果某个动作的$N_t(a) = 0$，那么该动作就是当前时步的被选动作；否则，就从根据公式计算的列表中选择对应最大值的动作。

```
if strategy == 'ucb':
    c_ratio, step = kwargs['c_ratio'], kwargs['step']
    if 0 in self.action_counts:
        action_idx = self.action_counts.index(0)
    else:
        values_ucb = [item+c_ratio*np.sqrt(np.log(step) / self.action_counts[i]) for i,
item in enumerate(self.q_values)]
    action_idx = np.argmax(values_ucb)
    return action_idx
```

学习逻辑

同ε-贪心策略的学习逻辑，只是参数改变（如增加时步参数`step`）。

```
if strategy == 'ucb':
    action_idx = self.action_strategy(strategy, c_ratio=kwargs['c_ratio'], step=item)
    reward = self.step(action_idx)
    self.update_q(strategy, action_idx, reward)
```

玻尔兹曼策略

动作选择

根据`np.random.uniform(0, 1)`落在可能性$\pi(a)$（即`pi_values`）的区间（即`interval`）来决定动作选择。

```
if strategy == 'boltzmann':
    interval = [(np.sum(self.pi_values[0:i-1]), np.sum(self.pi_values[0:i-1])+item) for
i, item in enumerate(self.pi_values)]
    action_idx = 0
    for i, item in enumerate(interval):
        if item[0] < np.random.uniform(0, 1) < item[1]:
            action_idx = i
            break
    return action_idx
```

可能性$\pi(a)$的计算依据书中公式2.11及公式2.12来计算。

```
def update_h_with_pi(self, action_idx, reward, a_ratio, step):
    pi_sum = np.sum([np.exp(item) for item in self.h_values])
    self.pi_values = [np.exp(item) / pi_sum for item in self.h_values]
    for i in range(len(self.actions)):
        if i != action_idx:
            self.h_values[i] += -a_ratio * (reward - np.sum(self.q_values) / step) *
self.pi_values[i]
        else:
            self.h_values[i] += a_ratio * (reward - np.sum(self.q_values) / step) * (1
- self.pi_values[i])
```

学习策略

需要在获取奖赏及更新$Q(a)$值之间插入更新可能性$\pi(a)$及偏好值$H(a)$的操作

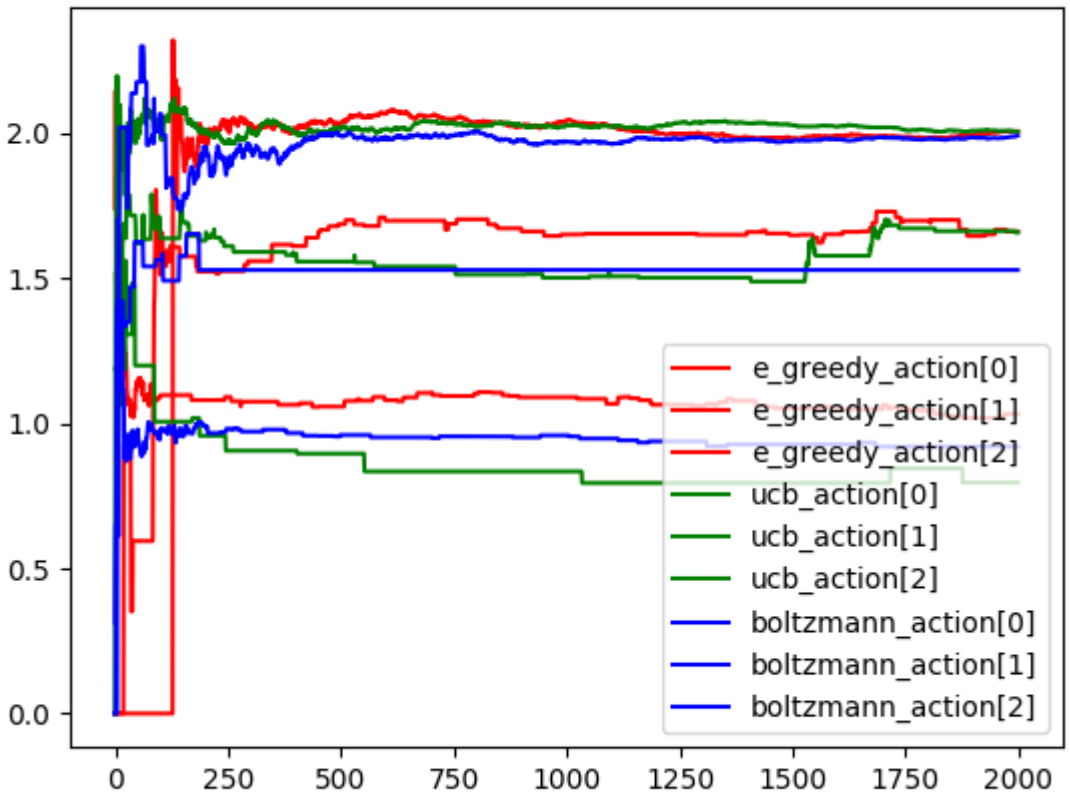
```
if strategy == 'boltzmann':
    action_idx = self.action_strategy(strategy, a_ratio=kwargs['a_ratio'], step=item)
    reward = self.step(action_idx)
    # 更新可能性π(a)及偏好值H(a)
    self.update_h_with_pi(action_idx, reward, a_ratio=kwargs['a_ratio'], step=item+1)
    self.update_q(strategy, action_idx, reward)
```

三种策略的比较

分别使用上述三种策略，进行2000次学习后，各策略的值函数情况如下图所示。

```
np.random.seed(0)
k_gamble = KGamble()
k_gamble.train(steps=2000, strategy='e_greedy', e_ratio=0.1)
k_gamble.plot(colors='rrr', strategy='e_greedy')
k_gamble.reset()
k_gamble.train(steps=2000, strategy='ucb', c_ratio=2)
k_gamble.plot(colors='ggg', strategy='ucb')
k_gamble.reset()
k_gamble.train(steps=2000, strategy='boltzmann', a_ratio=0.1)
k_gamble.plot(colors='bbb', strategy='boltzmann')
plt.show()
```

可以看出：在当前参数下，绿色（UCB策略）的值函数情况是最好的，另外两种策略的值函数区别不大。这可能是受`np.random.seed(0)`的影响，改变`np.random.seed(0)`的取值会导致生成的图形存在非常大的差异。



构建鸳鸯环境，体会MDP问题

Pygame环境配置及学习

安装

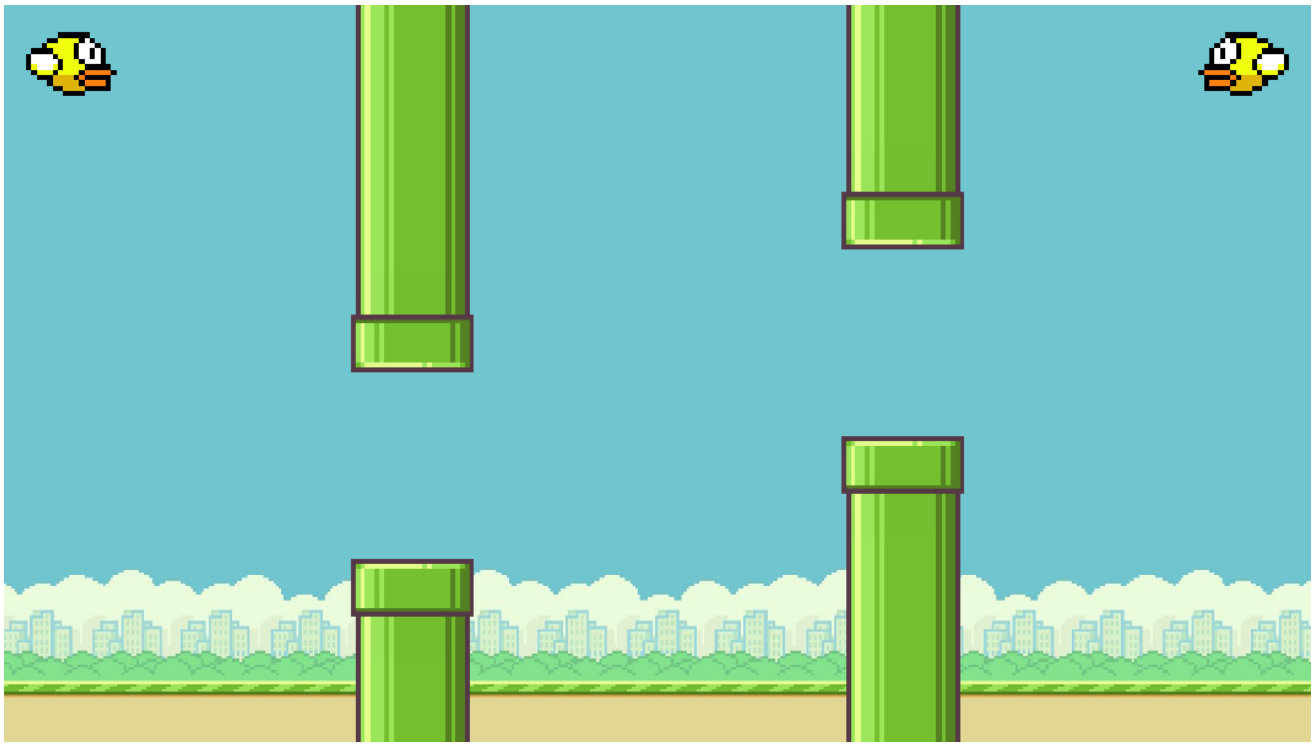
```
pip install pygame
```

入门教程

- pygame-a-primer
- pygame documentation 1.9.2

鸳鸯环境

程序文件参见`lovebirds.py`，`./assets`文件夹包含背景图、挡板、~~鸳鸯~~的素材图片，环境界面如下图所示：



构建环境元素

由于环境中存在2只鸳鸯与4个挡板，因此，我继承`pygame.sprite.Sprite`类，分别构建挡板和鸳鸯的动画类。由于环境中的挡板是固定的，因此挡板类`Brick`只实现了`__init__()`；而鸳鸯需要在场景中移动，因此鸳鸯类`Bird`中实现了`__init__()`、`update()`、`collision()`。`update(self, step, direction)`根据步长`step`和方向`direction`来更新鸳鸯的位置，`collision(self, step, bricks, window_size)`负责检测鸳鸯和挡板`bricks`以及环境边界`window_size`是否发生碰撞，并返回鸳鸯可以移动的方向。

检测碰撞

利用`pygame.sprite.spritecollide(bird, bricks, False, pygame.sprite.collide_mask)`来检测鸳鸯图片中非透明区域与环境边界的碰撞情况。因此需要利用`pygame.mask.from_surface()`分别提取鸳鸯图片以及挡板的非透明区域的mask。

利用`pygame.sprite.collide_mask()`来检测两只鸳鸯是否相遇。具体实现参见`find_mate()`。

环境逻辑

窗口的界面大小`window_size`为`(1066, 600)`。环境帧率`clock.tick()`设置为30fps，在回车键按下后，两只鸳鸯开始移动；ESC键按下后，程序会以`screen_{序号}.png`的形式保存游戏场景到程序运行的当前文件夹。只有点击窗口的关闭按钮，程序才会退出。每只鸳鸯的移动步长`step`均为50。如下图所示，两只鸳鸯相遇后就不再移动。

```
window_size = (1066, 600)
while True:
```

```
    clock.tick(30)
    for event in pygame.event.get():
        if event.type == KEYDOWN:
            if event.key == K_RETURN:
                start_flag = not start_flag
            if event.key == K_ESCAPE:
                pygame.image.save(screen, f'screen_{save_screen_no}.png')
                save_screen_no += 1
            elif event.type == QUIT:
                pygame.quit()
                sys.exit()
    screen_blit(screen, bg, birds, bricks, window_size)
    pygame.display.update()
    if start_flag:
        step = 50
        for bird in birds:
            direction = bird.collision(step, bricks, window_size)
            bird.update(step, direction)
            if find_mate(bird_male, bird_female):
                start_flag = False
```