强化学习：第六次作业

修改蒙特卡洛——探索初始化方法的错误

上次课上老师讲解了探索初始化方法的正确实现方式，因此在完成本次作业的时间差分方法之前，我先修正了上次作业的错误。此外，由于随着强化学习算法种类的增加，我重构了我的`lovebird`项目，将不同的算法分别存放在不同的文件夹中，下图展示的是代码重构后的工程目录树：

```
├── algorithm // 算法
│   ├── __init__.py
│   ├── markov.py
│   ├── monte_carlo.py
│   ├── temporal_difference.py
│   └── util.py
├── archieve // pdf作业存档
│   ├── ...
├── assets // 游戏图片
│   ├── background.png
│   ├── bird.png
│   └── brick.png
├── config // 游戏配置和算法参数配置
│   ├── algorithm_config.py
│   ├── game_config.py
│   ├── __init__.py
├── game // 游戏交互界面
│   ├── __init__.py
│   ├── lovebird.py
├── main.py // 程序入口
├── output // 作业图片及数据输出
│   ├── ...
├── README.md
├── test
│   ├── ...
```

```
├── video // 作业视频
|   ├── ...
```

各种强化学习算法均放在`algorithm`的文件夹中，本次作业的算法代码位于`temporal_difference.py`文件中。修改后的探索初始化算法位于`monte_carlo.py`文件中。

修改的要点在于生成数据集的时候需要根据策略π来更新动作选择，而之前在生成数据集的时候，我采用随机选择的策略：

```
action = np.random.choice(self.actions)
```

因此，修改后使用策略π来更新的代码如下：

```
action_candidate = []
for action_id, action in enumerate(self.actions):
    if self.q_values[state[0]][state[1]][self.actions.index(action)] ==
np.max(self.q_values[state[0]][state[1]]):
    action_candidate.append(action)
    action = np.random.choice(action_candidate)
```

经过100次迭代并使用上次作业的参数进行训练，寻路结果正确（23步）。修正后的探索初始化方法的寻路策略如下表所示（参见`output/06/05_modify`文件夹）：

```
[ -142.82692308 -1000.           -47.68396226 -1000.        ] s
[ -137.7173913  -1000.           -33.70430108  -142.13157895] s
[ -129.51219512 -1000.           -31.95299145  -138.12121212] s
[  -31.34275618 -1000.          -133.5         -132.475     ] e
[-126.66666667 -127.98039216  -25.68188105 -137.41666667] s
[ -21.83428107 -127.51020408 -131.66666667 -123.11864407] e
[ -21.25490196 -124.2745098  -122.0625     -123.59090909] e
[ -17.66734694 -119.55555556 -124.79069767 -119.53488372] e
[ -16.25025407 -118.66666667 -117.68421053 -117.78181818] e
[  -15.01367139  -115.93877551 -1000.          -113.91666667] e
[ -13.56020942  -99.78947368 -137.69811321 -117.76315789] e
[ -12.56007394 -127.28571429 -116.73333333 -120.66666667] e
[ -11.41518677 -121.31707317 -123.07692308 -117.7027027 ] e
[ -10.42746284 -109.4        -116.46666667 -125.11538462] e
[   -9.33632229  -116.13333333  -111.54545455 -1000.        ] e
[ -95.66666667 -114.88888889 -103.28571429   -8.24824737] n
[   -7.29464837 -1000.           -99.27272727  -107.83333333] e
[-91.46666667 -88.875      -98.31034483  -6.14616657] n
[  -5.18367347 -108.875      -100.94444444  -94.15      ] e
[ -4.12835388 -87.95238095 -93.54545455 -69.66666667] e
[ -3.05885472 -84.         -79.68421053 -59.3125    ] e
[-1000.           -65.4375       -75.68421053    -2.06384656] n
[-1000.      -61.375   -72.5      -1.    ] n
```

阅读《RL》第六章，理解两种算法的区别

Sarsa算法

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]. \tag{6.7}$$

Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q (e.g., ε-greedy)
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
    $S \leftarrow S'$; $A \leftarrow A'$;
  until S is terminal

Q-learning算法

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]. \tag{6.8}$$

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$
Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+, a \in \mathcal{A}(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    $S \leftarrow S'$
  until S is terminal

理解两种算法的区别

从伪代码的角度

从伪代码的角度来看，除了更新Q值公式的不同，Sarsa算法和Q-learning算法的主要区别在于：Sarsa算法是先进行动作A选择、观察奖赏R和状态S'，然后根据Q值选择状态S'下的动作A；而Q-learning算法则是先根据Q值选择状态S下的动作A，然后观察奖赏R和状态S'。这两者存在时序上的差异。

此外，Sarsa算法在迭代的过程中会同时更新状态S和动作A为状态S'和动作A'，也就是说在状态S'处，Sarsa算法就知道应该采取哪个动作A'，并真的采取了这个动作，因此才会在伪代码的最后将状态S和动作A都更新；而Q-learning算法则在状态S'时，仅计算采取哪个动作A'会得到更大的Q值，但是在状态S'处并没有真的采取这个动作A'，因此伪代码的最后仅更新状态S。

从公式的角度

从公式的角度来看，Sarsa算法能够利用状态S、动作A、奖赏R、下一个状态S'和在状态S'下的动作选择A'共五个数据，这也是Sarsa名字的由来。而Q-learning则只能利用状态S、动作A、奖赏R、下一个状态S'四个数据。

参考资料

1. Stack Overflow: What is the difference between Q-learning and SARSA?
2. Quora: What is the difference between Q-learning and SARSA learning?

利用时间差分方法解决鸳鸯找朋友的问题

时间差分方法通过`TemporalDifference`类实现，类似于前两次作业的蒙特卡洛方法类`MonteCarlo`和马尔科夫决策类`Markov`的实现。初始化需要提供奖赏空间`reward_grid`、动作空间`actions`、折扣因子`gamma`、方法空间`mode_space`、贪心策略参数ϵ及学习步长α。这些参数通过`AlgorithmConfig`类进行配置。重构以后奖赏空间的初始化通过`algorithm/util.py`中的`algorithm_init()`函数实现。

在做完蒙特卡洛方法和马尔科夫决策方法两次作业后，这次的作业实现起来就比较容易了。本次作业可以借用上两次作业中很多的函数和代码段。由于Sarsa算法和Q-learning算法都是通过计算`argmax Q()`来求取策略π的，因此，只需要记录对应状态及动作的q值：`q_values`，以及对该值的初始化：`init_q_values()`。

```
def init_q_values(self):
    # arbitrarily Q(s, a)
    self.q_values = -np.random.random(...) //代码太长，我就用省略号代替了

    # Q(terminal, *) = 0
    for e_state in self.e_states:
    for action_id, action in enumerate(self.actions):
    self.q_values[e_state[0]][e_state[1]][action_id] = 0

    # can't arrive
    for s_state in self.s_states:
    for action_id, action in enumerate(self.actions):
    if s_state == self.get_next_state(s_state, action):
    self.q_values[s_state[0]][s_state[1]][action_id] = -1000
    for c_state in self.c_states:
    for action_id, action in enumerate(self.actions):
    self.q_values[c_state[0]][c_state[1]][action_id] = -1000
```

两种算法都是都是通过计算`argmax Q()`来求取策略π的，因此他们的求取策略函数`give_action_advice()`函数的实现方式相同：

```
def give_action_advice(self, state):
    action_candidate = []
    for action_id, action in enumerate(self.actions):
    if action == self.actions[np.argmax(self.q_values[state[0]][state[1]])]:
    action_candidate.append(action)
    action = np.random.choice(action_candidate)
    print(state, action_candidate, action, self.q_values[state[0]][state[1]])
    return action
```

两种算法的共同的参数如下所示，分别表示两种算法的名称、选择哪种算法、学习步长参数α、贪心策略参数ϵ。

```
temporal_mode_space = ['sarsa', 'qlearning']
temporal_mode_count = 1
temporal_difference_alpha = 0.1
temporal_difference_epsilon = 0.1
```

Sarsa算法

Sarsa算法由`sarsa()`函数实现。该函数中涉及使用贪心策略`gredy(state)`求取当前状态的动作、获取下个时刻的状态`get_next_state(state, action)`、更新Q值`sarsa_update_q_values(state, action, reward, next_state, next_action)`等函数。

在判断Q值收敛方面，我偷懒了一下。就简单的判断前次Q值和本次Q值的差是否小于`1`。其实应该判断连续几次的Q值是否波动较小（这个容易实现）或者Q值是否保持不变（经过计算我发现Q值保持不变不太可能）。

```
if (last_q_values - self.q_values < 1).all():
    break
```

Sarsa算法的训练信息存放在`output/06/sarsa_train_info.txt`文件中，可以看到，经过390次迭代后，上下两次Q值的差值小于1（0.98810）。

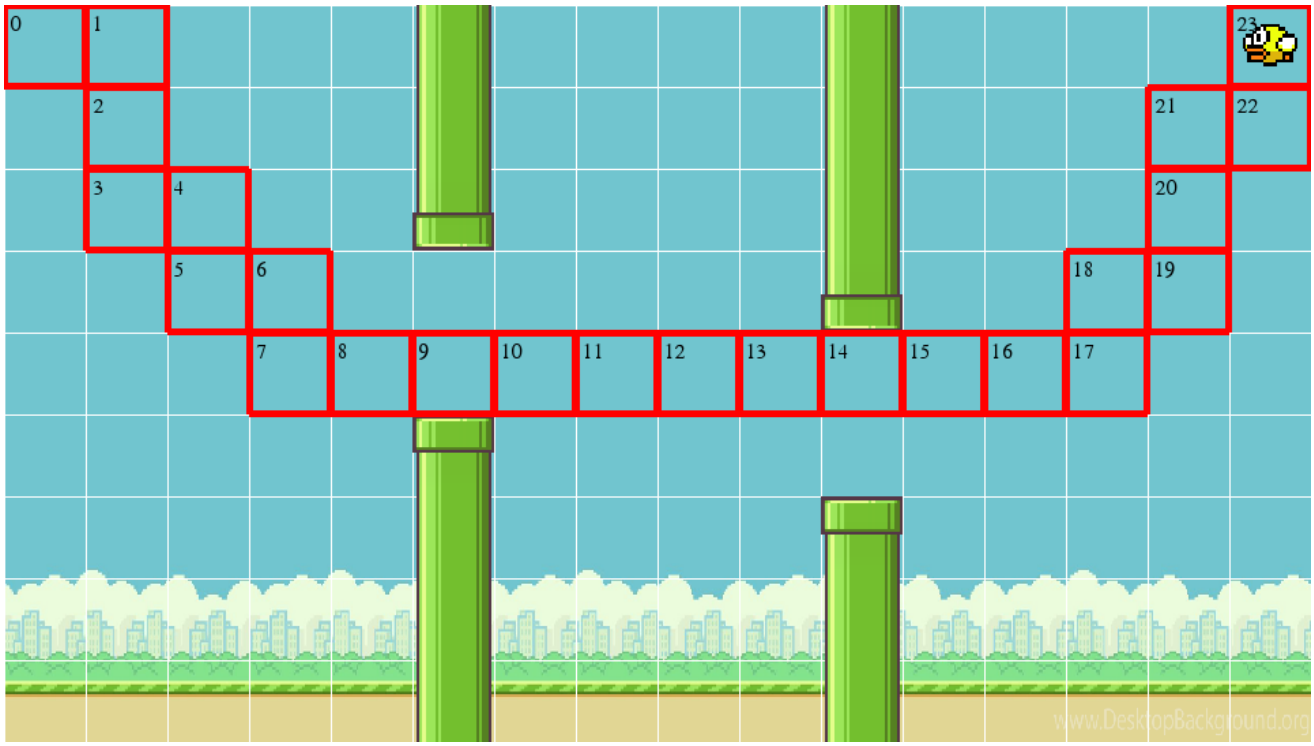
```
Episode: 0, 188.6949322050363
Episode: 1, 100.07959196444412
Episode: 2, 117.48215716269604
Episode: 3, 111.62432321617996
...
Episode: 388, 3.906522457316079
Episode: 389, 2.7332934767706654
Episode: 390, 0.9881047158182596
```

寻路策略存放在`output/06/sarsa_strategy.txt`文件中。文件中的数据分别表示：当前的状态、动作选择的后选集（有可能动作的选择不唯一）、动作选择、当前状态下各个动作的Q值。

```
[0, 0] ['e'] e [ -26.9217294  -350.39506894  -78.14131484 -374.35912411]
[0, 1] ['s'] s [ -45.50626529  -54.31138164  -25.4156393  -104.22951265]
[1, 1] ['s'] s [-27.65862836 -41.95686564 -23.15578674 -34.99910258]
[2, 1] ['e'] e [-22.58581346 -37.24907387 -23.45545642 -26.74681944]
[2, 2] ['s'] s [-21.41241716 -24.00837691 -21.14101854 -25.86296464]
[3, 2] ['e'] e [-19.65898955 -23.3141606  -21.95388343 -22.24349639]
[3, 3] ['s'] s [-19.43650693 -21.3019542  -18.61851486 -20.76839897]
[4, 3] ['e'] e [-17.54020705 -22.09034784 -20.6951277  -19.87210307]
[4, 4] ['e'] e [-16.09522124 -19.25679681 -20.17322694 -18.59725252]
[4, 5] ['e'] e [-15.03466197 -17.96032143 -16.65757215 -17.2547364 ]
[4, 6] ['e'] e [-13.77655421 -16.42749504 -25.43082446 -16.57964124]
[4, 7] ['e'] e [-12.72861396 -15.02308576 -15.13565938 -14.90889877]
[4, 8] ['e'] e [-11.54789479 -13.75318247 -14.03382642 -13.7648608 ]
[4, 9] ['e'] e [-10.27599616 -12.86903547 -12.41029061 -12.41099746]
[4, 10] ['e'] e [ -9.01557453 -11.49687237 -11.28045508 -10.11816431]
[4, 11] ['e'] e [ -7.98777831 -10.31138105 -10.15431414  -8.44427907]
[4, 12] ['e'] e [-6.93588744 -9.2131374  -9.29919868 -7.1347309 ]
```

```
[4, 13] ['n'] n [-6.59274269 -7.97359568 -8.08120598 -5.99129026]
[3, 13] ['e'] e [-4.4670451  -6.90574216 -6.85755272 -5.01008838]
[3, 14] ['n'] n [-22.91910652  -5.74725573  -6.16296179  -3.59334624]
[2, 14] ['n'] n [-4.85838705 -4.8050725  -5.4300259  -2.27523437]
[1, 14] ['e'] e [-1.15526795 -3.49556863 -3.45692333 -1.51380662]
[1, 15] ['n'] n [-1.54959637 -2.32496632 -7.18369376  0.        ]
```

Sarsa算法的寻路结果如图所示，可以看到算法找到了最优路径（23步）。



Q-learning算法

Q-learning算法由`qlearning()`函数实现。与Sarsa算法相似，该函数中涉及使用贪心策略`gredy(state)`求取当前状态的动作、获取下个时刻的状态`get_next_state(state, action)`、更新Q值（**注意不会有next_action**）`qlearning_update_q_values(state, action, reward, next_state)`等函数。

在判断Q值收敛方面，同样地我偷懒了一下。就简单的判断前次Q值和本次Q值的差是否小于`0.1`。

```
if (last_q_values - self.q_values < 0.1).all():
    break
```

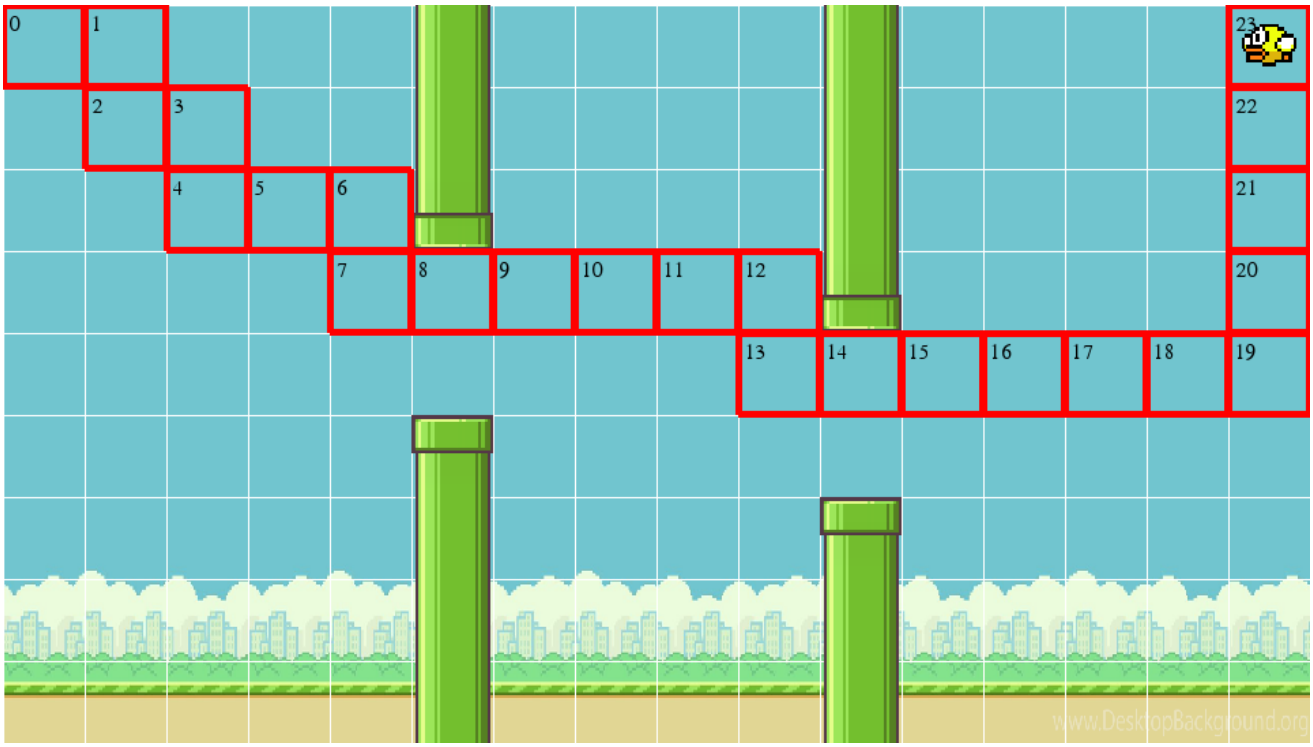
Q-learning算法的训练信息存放在`output/06/qlearning_train_info.txt`文件中，可以看到，经过127次迭代后，上下两次Q值的差值小于0.1（0.088）。

```
Episode: 0, 6.973796053133263
Episode: 1, 3.3150953722609824
Episode: 2, 2.0586038772828577
Episode: 3, 1.5759358089172988
...
Episode: 125, 0.16283720847184036
Episode: 126, 0.14049869115320845
Episode: 127, 0.08843081227820271
```

寻路策略存放在`output/06/qlearning_strategy.txt`文件中。文件中的数据分别表示：当前的状态、动作选择的后选集（有可能动作的选择不唯一）、动作选择、当前状态下各个动作的Q值。

```
[0, 0] ['e'] e [ -22.15788307 -200.86907676  -22.15956933 -241.00873901]
[0, 1] ['s'] s [-21.17836912 -22.0316722  -21.1782913  -95.43767587]
[1, 1] ['e'] e [-20.17959847 -20.7572698  -20.17959869 -21.0522385 ]
[1, 2] ['s'] s [-19.17960284 -20.23634006 -19.17960284 -20.74634151]
[2, 2] ['e'] e [-18.17960288 -19.6723108  -18.17960288 -19.14078024]
[2, 3] ['e'] e [-17.17960288 -18.83036649 -17.17960288 -18.82850333]
[2, 4] ['s'] s [-30.05380294 -18.01952996 -16.17960288 -18.08112915]
[3, 4] ['e'] e [-15.17960288 -17.15430271 -15.17960288 -17.06937597]
[3, 5] ['e'] e [-14.17960288 -16.13034208 -14.17960288 -25.60657529]
[3, 6] ['e'] e [-13.17960288 -15.08877997 -13.17960288 -15.01681798]
[3, 7] ['e'] e [-12.17960288 -13.9892598  -12.17960288 -14.02695639]
[3, 8] ['e'] e [-11.17960288 -12.89053998 -11.17960288 -13.04421917]
[3, 9] ['s'] s [-12.45696797 -12.15681523 -10.17960288 -12.17003471]
[4, 9] ['e'] e [ -9.17960288 -11.17960288 -11.17960288 -11.17960288]
[4, 10] ['e'] e [ -8.17960288 -10.17960288 -10.17960288  -9.17960288]
[4, 11] ['e'] e [-7.17960288 -9.17960283 -9.17960212 -7.17960288]
[4, 12] ['e'] e [-6.17960288 -8.17910259 -8.17935334 -6.17960288]
[4, 13] ['e'] e [-5.17960288 -7.17783301 -7.17288869 -5.17960288]
[4, 14] ['e'] e [-4.17960288 -6.15161841 -6.1310962  -4.17960288]
[4, 15] ['n'] n [-4.19695253 -5.17923486 -5.17941854 -3.17960288]
[3, 15] ['n'] n [-3.18033898 -4.1796018  -4.1796025  -2.17960288]
[2, 15] ['n'] n [-2.17960322 -3.17960288 -3.17960288 -1.17960288]
[1, 15] ['n'] n [-1.17960288 -2.17960288 -2.17960288 -0.17960288]
```

Q-learning算法的寻路结果如图所示，可以看到算法找到了最优路径（23步）。



总结

总的来说，由于有了前两次作业的铺垫，在理解时间差分两种算法的基础上，两种算法的代码实现都很容易。由于视频都是找朋友，只不过路径不同，因此这次我就没有录制视频。

在Sarsa算法和Q-learning算法的实现上，相较于Sarsa算法，Q-learning算法不需要求next_action，并且在更新Q值的时候也不需要利用next_action，而是使用`argmax()`来求取`next_state`下最大的Q值：

```
next_q_values = np.max(self.q_values[next_state[0]][next_state[1]])
```

而且，在更新状态和动作时，Q-learning算法也不需要更新动作，只更新状态。以上就是在实现时Sarsa算法和Q-learning算法的区别。