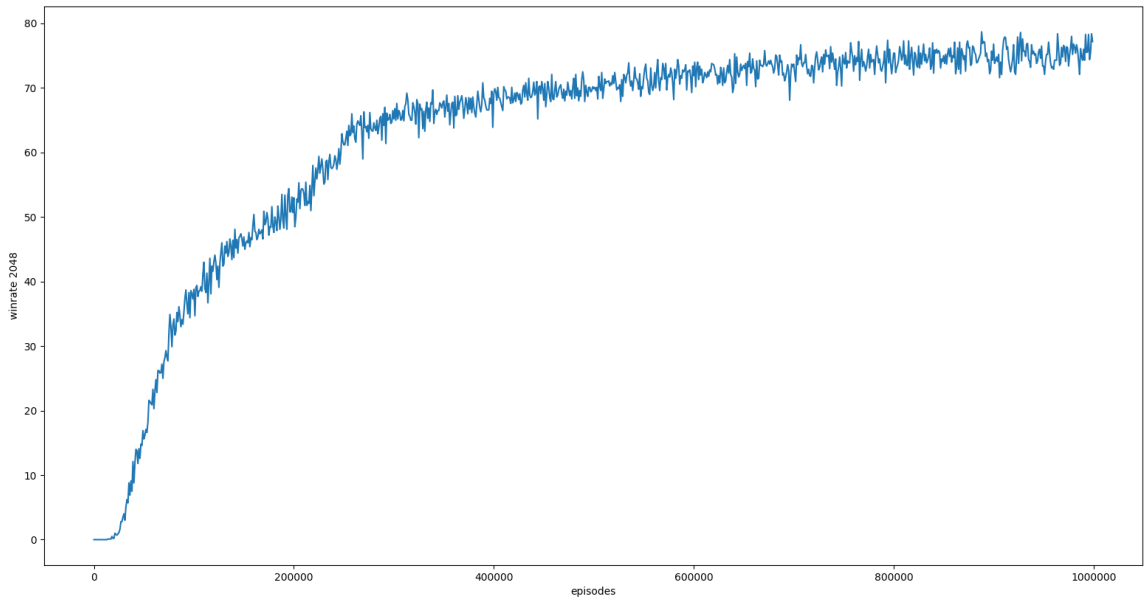
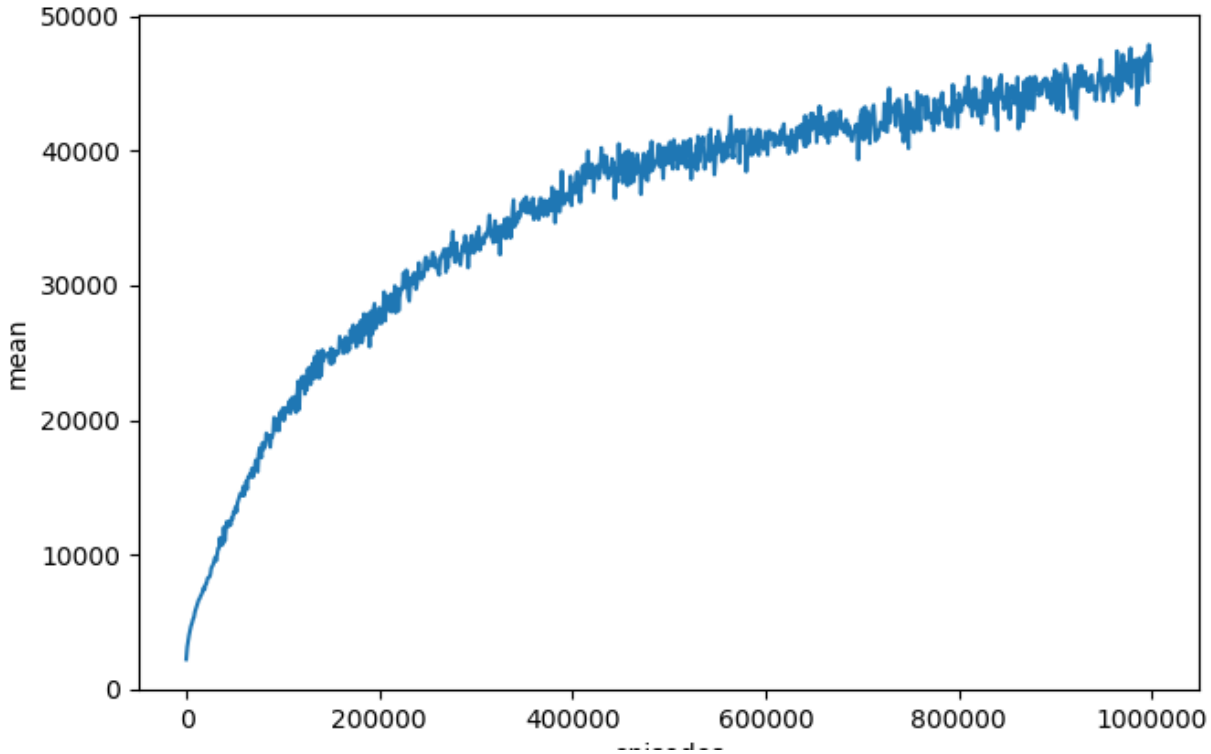
Lab5: Temporal Difference Learning

0851915 徐玉山

Report:

1. Plot





2. Describe your implementation in detail.

TD-state

function EVALUATE(s, a)
  $s', r \leftarrow$ COMPUTE AFTERSTATE(s, a)
  $S'' \leftarrow$ ALL POSSIBLE NEXT STATES(s')
  return $r + \Sigma_{s'' \in S''} P(s, a, s'') V(s'')$

```
for (int i=0;i<4;i++) {
  state *move = &after[i];
  if (move->assign(b)) {
    float sample_result = 0;
    for(int j=0;j<10;j++){
      state s(i);
      state *samplemove = &s;
      samplemove -> assign(b);
      sample_result += estimate(move->after_state());
    }
    sample_result/=10;
    move->set_value(move->reward() + sample_result );
```

用抽樣的方法來計算

$$\Sigma_{s'' \in S''} P(s, a, s'') V(s'')$$

相信只要抽得夠多，抽樣結果就會越接近實際的值。

```
bool assign(const board &s) {
  debug << "assign " << name() << std::endl << s;
  before = after = s;
  score = after.move(opcode);
  after.popup();
  esti = score;
  return score != -1;
}
```

把 `class state` 中的 `afterstate` 定義成執行完 popup 之後的結果

function LEARN EVALUATION(s, a, r, s', s'')
  $V(s) \leftarrow V(s) + \alpha(r + V(s'') - V(s))$

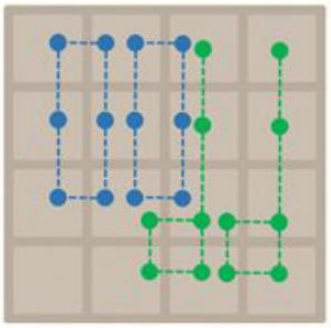
如此這個部分的修改也完成了

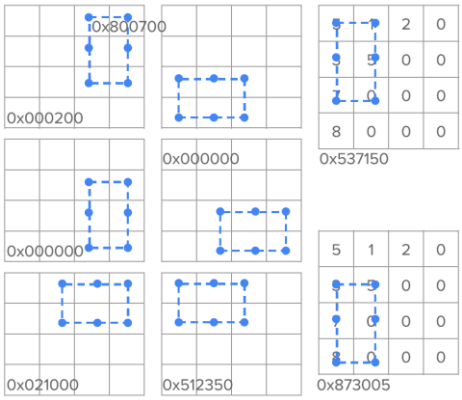
3. Describe the implementation and the usage of n-tuple network.

假設每個格子中的數字可以從 `2^0 ~ 2^15` 共16種
那總共的State數量約有 `1.8*10^19`，如果以此數量來建造 `V(s)` 表格顯然不現實
因此使用n-tuple來”壓縮”狀態量。

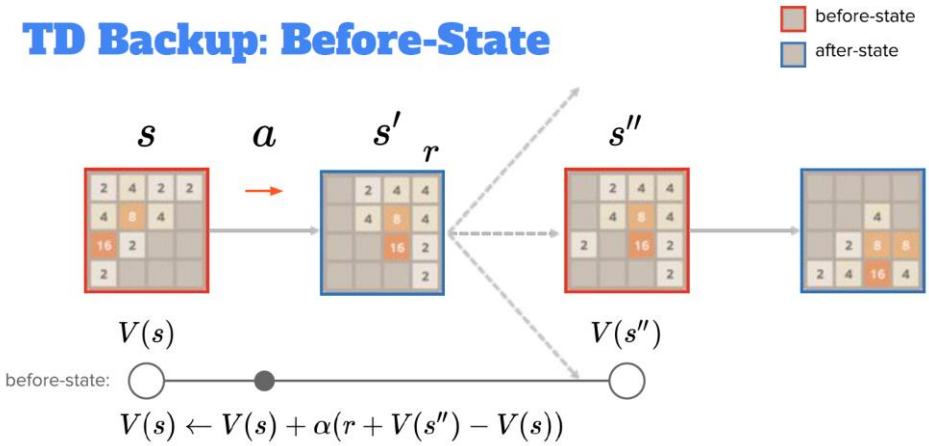
如下圖所示，一個盤面s，他的V(s)就是四種tuple分別查表並相加的結果(如下圖左，圖中有四種tuples)，而每一種tuple都會計算所有旋轉對稱的情況(如下圖右)

$$V(s) = f_1(s) + f_2(s) + f_3(s) + f_4(s)$$





4. Explain the TD-backup diagram of V(state).



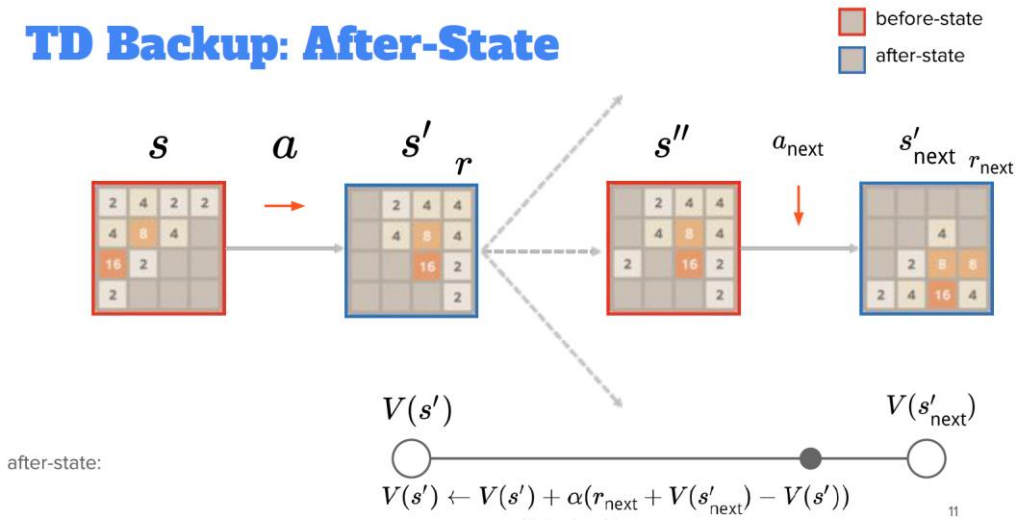
這裡的s’’是要經過popup之後的狀態，所以帶有隨機性。
因此在計算V(s’’)的時候要把所有的s’’都計算出來，十分麻煩
(實作上我是用抽樣的)
Tuple-network存的state是已經做完popup之後的盤面。

5. Explain the action selection of V(state) in a diagram.

function EVALUATE(s, a)
  $s', r \leftarrow$ COMPUTE AFTERSTATE(s, a)
  $S'' \leftarrow$ ALL POSSIBLE NEXT STATES(s')
  return $r + \Sigma_{s'' \in S''} P(s, a, s'') V(s'')$

首先，先往四個方向滑動取得reward。
要計算各種popup的情況，並用tuple-network查表返回V(s’’)累加。
選得到最大結果的那個action當作下一步的行動

6. Explain the TD-backup diagram of V(after-state).



所謂的afterstate，遊戲中的state選擇一個action(比如說右滑)，在環境還沒有給出反應之前所得到的一個狀態。
如此設計V(s)的好處是更新或是選擇action的時候不需要考慮環境給的popup
降低了訓練的成本。

在訓練中 tuple-network 保存的盤面是 afterstate

7. Explain the action selection of V(after-state) in a diagram.

function EVALUATE(s, a)
  $s', r \leftarrow$ COMPUTE AFTERSTATE(s, a)
  return $r + V(s')$

首先，先往四個方向滑動取得reward。
查tuple-network得知 V(afterstate)。
相加得到 reward + V(afterstate)
選得到最大結果的那個action當作下一步的行動

8. Explain the mechanism of temporal difference learning.

首先要有個估計狀態分數的函數 無論是 V(S) Q(s,a) 或是 policy π 這邊統一用 V(s) 來代表

$V(s_t)$ is approximate to actual return R_t.
Error: $\delta_t = R_t - V(s_t)$.
Adjust: $V(s_t) = V(s_t) + \alpha\delta_t = V(s_t) + \alpha\big(R_t - V(s_t)\big)$

α: a step-size parameter to control the learning rate
如此就可以按照episode的結果來更新V(s),使得V(s)越來越準。

9. Explain whether the TD-update perform bootstrapping.

是使用bootstrapping沒錯，雖然是等到整個episode結束之後才做更新
但是每次更新的時候都是對下一步的 Q Value 做計算 (這邊我的after_state有重新定義成popup後的狀態了)

```
void update_episode(std::vector<state> &path, float alpha = 0.1) const {
  float exact = 0;
  for (path.pop_back() /* terminal state */; path.size(); path.pop_back()) {
    state &move = path.back();
    float error = exact - (move.value() - move.reward());
    debug << "update error = " << error << " for after state" << std::endl
          << move.after_state();
    exact = move.reward() + update( move.after_state() , alpha * error);
    //exact = move.reward() + update( move.before_state() , alpha * error);
  }
}
```

10. Explain whether your training is on-policy or off-policy.
Off-policy.

Off-policy
因為使用的是 Q-Learning 的更新方式

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$$

而非 SARSA 的更新方式

$$Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma Q(S',A') - Q(S,A)]$$

11. Other discussions or improvements.

None.
我不該抽樣計算 argmax 的，我問了一下隔壁同學有乖乖寫期望值估計的都可以輕鬆練到 90%以上
不過來不及寫了= =

Performance:

```
999000  mean = 47170.1  max = 157636
    32  100%    (0.1%)
    64  99.9%   (0.4%)
   128  99.5%   (0.9%)
   256  98.6%   (3.5%)
   512  95.1%   (8.3%)
  1024    86.8%   (8.4%)
  2048    78.4%   (29.3%)
  4096    49.1%   (45%)
  8192    4.1%    (4.1%)
```