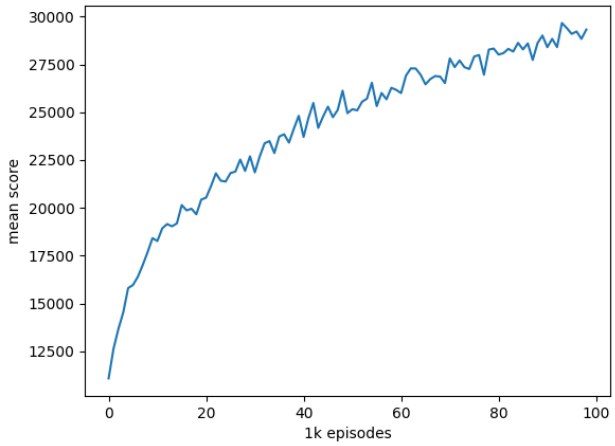
# Lab3 : Temporal Difference Learning

310605005 王映勻

## 1. A plot shows episode scores of at least 100,000 training episodes

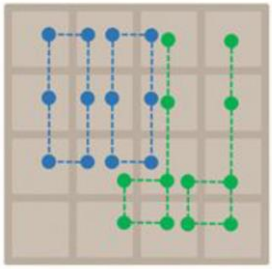



## 2. Describe the implementation and the usage of $n$-tuple network.

2048 盤面大小為 4*4 共 16 格，而每格從空白到 2048 至少有 12 種可能性，因此若要記錄每種盤面的估計值需要 12^16 的大小，若是全部紀錄的話會使記憶體用量無法負荷，而引入 n-tuple network 架構的話，則能有效地解決這個問題。

我們可以選取特定幾格作為盤面的「feature」代表當前狀態，而計算估計值時就不用查出當前 board 的分數，而就只針對盤面上那一小塊 feature，將每個 feature 的分數加總代表整個盤面的分數。以原本設定的 4 X 6-tuples network 為例，這樣計算上就只會有 4* 12^6 種 state，計算量和原本相差了大概 12^10 倍。

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



3. **Explain the mechanism of TD(0).**

Temporal-difference (TD) Learning 是強化學習核心的算法之一，它集結了 Monte Carlo Methods (MC) 和 Dynamic Programming (DP) 的優點。TD 可以直接從經驗學習，不需要知道 environment model，和 MC 一樣為 model-free 的方法；而像 DP 一樣，TD 方法不需要得到最終的 outcome 才更新 model，可以即時利用其它狀態的估計值來更新當前狀態。

TD learning 有一個參數 λ 來決定「多少比例考慮實際賺分的結果」，而 TD(0) (one-step TD)就是完全不考慮實際賺分的結果，只考慮「下一個狀態」，其公式如下：

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$$

意思是在下一個時刻 t+1 時，會馬上利用觀測到的 reward $R_{t+1}$ 和估計的 $V(S_{t+1})$ 更新當前狀態估計值 $V(S_t)$。

下圖為 TD(0)的運作流程，可以看出首先會先根據我們設定的 policy $\pi$ 決定 action $A$，利用 action 後產生的 reward $r$ 以及下一個狀態 $s'$ 的估計值 $V(s')$ 更新當前狀態 $s$ 的估計值 $V(s)$。

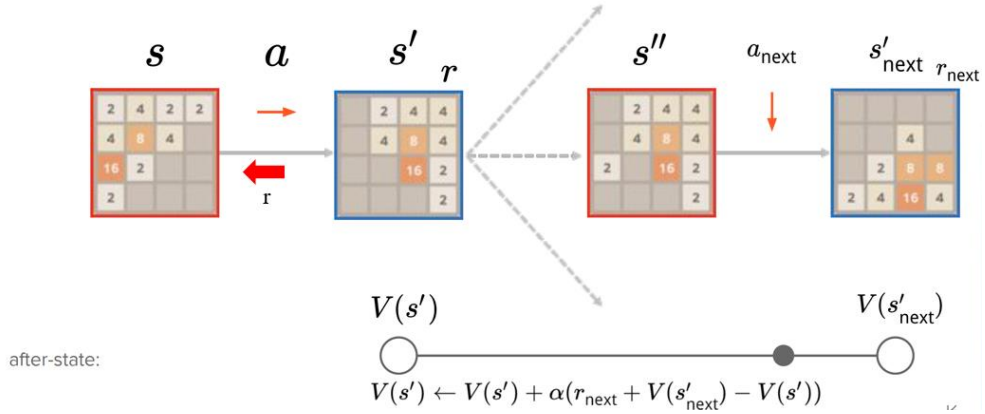
**Tabular TD(0) for estimating $v_\pi$**

Input: the policy $\pi$ to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0, \forall s \in \mathcal{S}^+$)
Repeat (for each episode):
  Initialize $S$
  Repeat (for each step of episode):
    $A \leftarrow$ action given by $\pi$ for $S$
    Take action $A$, observe $R$, $S'$
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  until $S$ is terminal



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

After-state 指的是下圖中的 $s'$，代表經過 action 後的 state。其估計值 V(after-state) 也就是 $V(s')$，是使用下一時刻 $s'_{next}$ 的估計值 $V(s'_{next})$ 以及 reward $r_{next}$ 來更新。



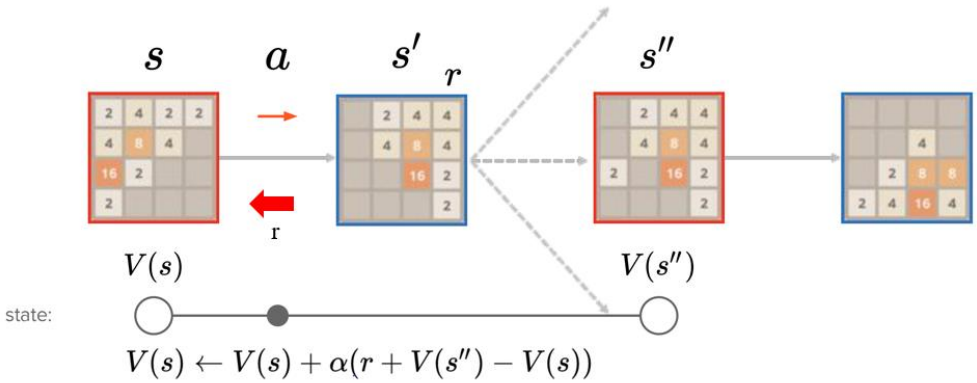

**5. Explain the action selection of V(after-state) in a diagram.**

在選擇 action 時，首先要先計算出四種 action $(a)$ 導致的 after-state $(s')$ 及 reward $(r)$，計算 $s'$ 的估計值 $V(s')$ 加上 reward $r$，以選擇出分數最高的 action。

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

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

State 指的是下圖中的 $s$ & $s''$，代表在 action 之後經過 environment 隨機 pop out 一個 tile 的 state。 計算當前狀態估計值 V(state) 也就是 $V(s)$，是使用 $V(s'')$ 以及 reward $r$ 來更新。



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

在選擇 action 時，首先一樣要先計算出四種 action $(a)$ 導致的 after-state $(s')$ 及 reward $(r)$，還要再列舉經過環境隨機產生 tile 後所有可能的 state $(s'')$，計算所有 $V(s'')$ 乘上 environment model 的狀態轉移機率 P 並加總，再加上 reward，選擇出分數最高的 action。

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

## 8. Describe your implementation in detail.

以下分別針對五個 TODO 說明：

(1) estimate

這個 function 的目的是要計算當前 board 的估計值，由於我們使用 n-tuple network，因此是要計算我們設定的 patterns 以及其 8 個 isomorphism 的 value 之和。

要得到這些 value 需要去查 weight table，而這邊就要利用 indexof 這個 function 產生 pattern 對應 weight table 的編號，再透過 operator 得到其 weight value。

```
virtual float estimate(const board& b) const {
    // TODO
    float value = 0;
    for (int i=0; i<iso_last; i++){
        size_t index = indexof(isomorphic[i], b);
        value += operator[](index);
    }
    return value;
}
```

(2) update

這個 function 是用在 TD backup 時要去更新當前 board 的估計值，input 的 u 是已經乘上 learning rate 的 TD-error。和 estimate 一樣透過查表得到 pattern 的 weight value，但這邊的 value 要再加上 TD-error u 作更新。

```
virtual float update(const board& b, float u) {
    // TODO
    float value = 0;
    for(int i=0; i<iso_last; i++){
        size_t index = indexof(isomorphic[i], b);
        operator[](index) += u;
        value += operator[](index);
    }
    return value;
}
```

(3) indexof

前面有提到在求 board 的估計值時都需要去查 weight table，為了方便查詢，需要知道 pattern 在 weight table 對應的編號。將每個 pattern 對應的位元值依照其大小用 4 個 bit 表示 (可代表 2^0~2^15 的值)。

```
size_t indexof(const std::vector<int>& patt, const board& b) const {
    // TODO
    size_t index = 0;
    for(int i=0; i<patt.size(); i++){
        index |= b.at(patt[i])<<(i*4);
    }
    return index;
}
```

(4) select_best_move

按照前面所提的以 V(state)進行 action selection 的方法，首先窮舉出所有可能的下一個狀態並計算估計值，再依照 environment model 的設定分別對 2-tile 以及 4-tile 乘上其狀態轉移機率 P，回傳 $r + \sum_{s'} P(s, a, a'')V(s'')$，最後選擇出分數最高的 action。

```
if (move->assign(b)) {
    // TODO
    board s = move->after_state();
    int count = 0;
    double sum_P_V = 0;
    float V_4, V_2;
    /*All Possibile Next States*/
    for (int i=0; i<16; i++){
        if (s.at(i) == 0){
            count ++;
            /*popup a 4-tile*/
            s.set(i, 2);
            V_4 = estimate(s);
            s.set(i, 0); //clear
            /*popup a 2-tile*/
            s.set(i, 1);
            V_2 = estimate(s);
            s.set(i, 0); //clear
            /*Calaulate P*/
            sum_P_V += (V_4*0.1 + V_2*0.9);
        }
    }
    if(count != 0){
        move->set_value(move->reward() + sum_P_V);
    }
    if(count == 0){
        move->set_value(move->reward());
    }
```

(5) update_episode

按照前面所提的以 V(state)進行 TD-backup 的方法，令 $TD-error = r + V(s[t+1]) - V(s[t])$，依序反覆更新回去。

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

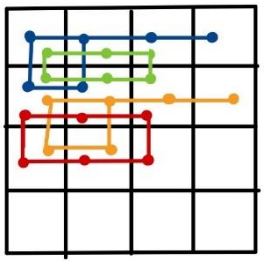
## 9. Other discussions or improvements.

依據參考資料，文中提到經實驗後發現，5-tuple network 的效果最好，因此我也嘗試了幾種不同的 tuple 組合，由於全部跑完所需的時間太長，因此以下僅列出 5000 episodes 的實驗結果：

(1) Original

```
pattern({ 0, 1, 2, 3, 4, 5 }));
pattern({ 4, 5, 6, 7, 8, 9 }));
pattern({ 0, 1, 2, 4, 5, 6 }));
pattern({ 4, 5, 6, 8, 9, 10 }));
```

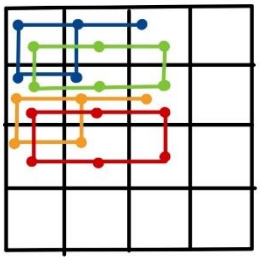
```
6-tuple pattern 012345, size = 16777216 (64MB)
6-tuple pattern 456789, size = 16777216 (64MB)
6-tuple pattern 012456, size = 16777216 (64MB)
6-tuple pattern 45689a, size = 16777216 (64MB)
1000    mean = 7683.77  max = 25856
        64      100%    (0.1%)
        128     99.9%   (5.2%)
        256     94.7%   (20.9%)
        512     73.8%   (54.6%)
        1024    19.2%   (19.1%)
        2048    0.1%    (0.1%)
2000    mean = 10914.5  max = 31804
        128     100%    (0.8%)
        256     99.2%   (7.4%)
        512     91.8%   (47.9%)
        1024    43.9%   (41.8%)
        2048    2.1%    (2.1%)
3000    mean = 12447.7  max = 36248
        128     100%    (0.6%)
        256     99.4%   (5.5%)
        512     93.9%   (37.6%)
        1024    56.3%   (51.2%)
        2048    5.1%    (5.1%)
4000    mean = 13627.5  max = 39496
        128     100%    (0.2%)
        256     99.8%   (4.9%)
        512     94.9%   (30.9%)
        1024    64%     (56.1%)
        2048    7.9%    (7.9%)
5000    mean = 14849.2  max = 50984
        128     100%    (0.6%)
        256     99.4%   (4%)
        512     95.4%   (27.9%)
        1024    67.5%   (54.8%)
        2048    12.7%   (12.6%)
        4096    0.1%    (0.1%)
```



(2)

```
pattern({ 0, 1, 2, 4, 5 }));
pattern({ 4, 5, 6, 8, 9 }));
pattern({ 0, 1, 2, 4, 5, 6 }));
pattern({ 4, 5, 6, 8, 9, 10 }));
```

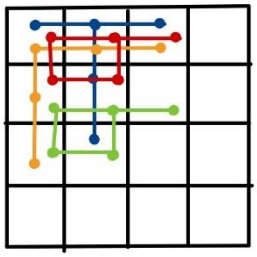
```
5-tuple pattern 01245, size = 1048576 (4MB)
5-tuple pattern 45689, size = 1048576 (4MB)
6-tuple pattern 012456, size = 16777216 (64MB)
6-tuple pattern 45689a, size = 16777216 (64MB)
1000    mean = 7796.13  max = 26504
        64      100%    (0.4%)
        128     99.6%   (4.5%)
        256     95.1%   (22.8%)
        512     72.3%   (52.4%)
        1024    19.9%   (19.5%)
        2048    0.4%    (0.4%)
2000    mean = 10913.5  max = 36244
        128     100%    (0.5%)
        256     99.5%   (9.5%)
        512     90%     (42.4%)
        1024    47.6%   (45.4%)
        2048    2.2%    (2.2%)
3000    mean = 12786.3  max = 33256
        128     100%    (0.3%)
        256     99.7%   (5.7%)
        512     94%     (32.2%)
        1024    61.8%   (56.5%)
        2048    5.3%    (5.3%)
4000    mean = 14029.8  max = 36036
        64      100%    (0.2%)
        128     99.8%   (0.6%)
        256     99.2%   (3.9%)
        512     95.3%   (27.8%)
        1024    67.5%   (59%)
        2048    8.5%    (8.5%)
5000    mean = 14981    max = 39568
        64      100%    (0.1%)
        128     99.9%   (0.3%)
        256     99.6%   (3.8%)
        512     95.8%   (26%)
        1024    69.8%   (58%)
        2048    11.8%   (11.8%)
```



(3)

```
pattern({ 0, 1, 2, 5, 9 }));
pattern({ 0, 1, 2, 4, 8 }));
pattern({ 0, 1, 2, 4, 5 }));
pattern({ 4, 5, 6, 8, 9 }));
```

```
5-tuple pattern 01259, size = 1048576 (4MB)
5-tuple pattern 01248, size = 1048576 (4MB)
5-tuple pattern 01245, size = 1048576 (4MB)
5-tuple pattern 45689, size = 1048576 (4MB)
1000    mean = 8321.94   max = 31804
        32      100%    (0.1%)
        64      99.9%   (0.4%)
        128     99.5%   (4%)
        256     95.5%   (19.7%)
        512     75.8%   (51.4%)
        1024    24.4%   (23.7%)
        2048    0.7%    (0.7%)
2000    mean = 11915.8   max = 35624
        128     100%    (1.1%)
        256     98.9%   (7.4%)
        512     91.5%   (38.7%)
        1024    52.8%   (49.3%)
        2048    3.5%    (3.5%)
3000    mean = 13991.2   max = 36404
        64      100%    (0.1%)
        128     99.9%   (0.3%)
        256     99.6%   (4.9%)
        512     94.7%   (29%)
        1024    65.7%   (57.6%)
        2048    8.1%    (8.1%)
4000    mean = 15002.9   max = 40232
        128     100%    (0.4%)
        256     99.6%   (3.6%)
        512     96%     (25.3%)
        1024    70.7%   (59.3%)
        2048    11.4%   (11.4%)
5000    mean = 15701.1   max = 52448
        64      100%    (0.1%)
        128     99.9%   (0.1%)
        256     99.8%   (3.2%)
        512     96.6%   (24.5%)
        1024    72.1%   (57.3%)
        2048    14.8%   (14.6%)
        4096    0.2%    (0.2%)
```



觀察以上數據可發現第三種 tuple 設計效果最好，但其實三者分數差異不大，也不排除是 random seed 不同的緣故。

Reference：


https://www.mxeduc.org.tw/scienceaward/history/projectDoc/18th/doc/SA18-065_final.pdf?fbclid=IwAR2ogNjtCRuspYBkydsLYJJ1orCYKAZ5k3jBQN6NcpmdSNb0MgHjPGft0Qk