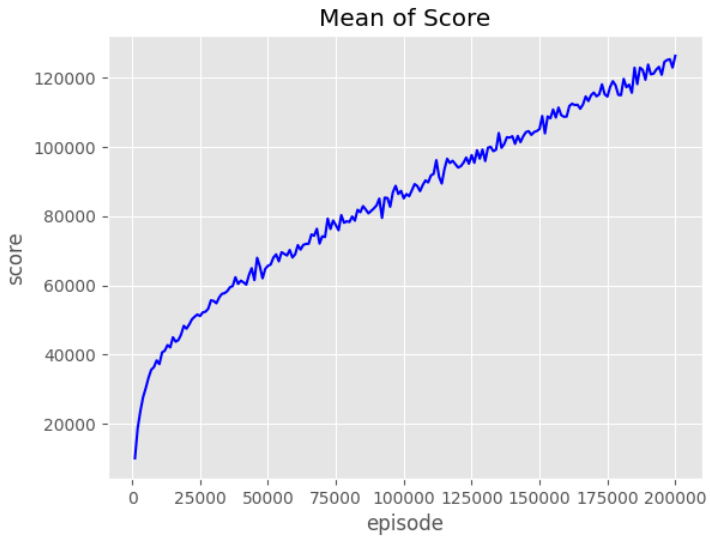
RL Lab 1 Report

資科工碩 陳冠廷 313551058

- **Basic**
  - **A plot shows scores (mean) of at least 100k training episodes**

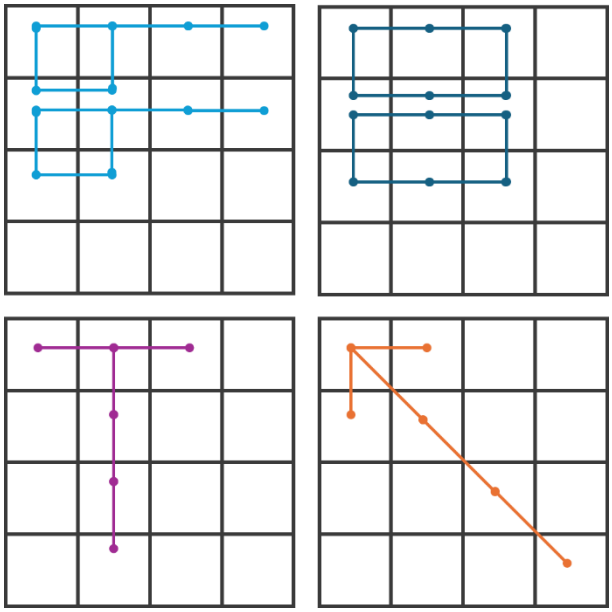


The testing score:

```
1000    mean = 129996   max = 314072
        512     100%    (0.3%)
        1024    99.7%   (1.5%)
        2048    98.2%   (10.1%)
        4096    88.1%   (21.4%)
        8192    66.7%   (61.6%)
        16384   5.1%    (5.1%)
```

- **Bonus**

■ **Describe the implementation and the usage of *n*-tuple network.**



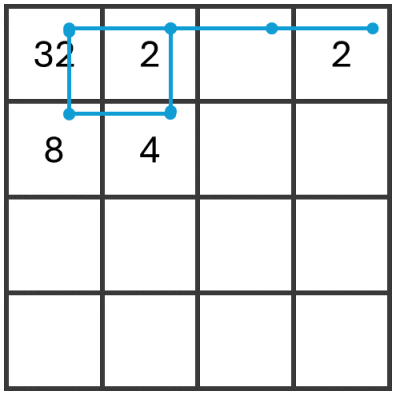
我一共使用了 6 種 6-tuple 作為盤面的特徵(如圖所示)，除了 sample code 給定的 4 種外，我額外加了 T 型的特徵與箭頭的特徵。pattern class 在建立時會產生八種不同的 isomorphic(旋轉與鏡射)，來捕捉一樣資訊的版面，類似資料擴增的概念。

$$V(s) = f_1(s) + f_2(s) + f_3(s) + f_4(s) + f_5(s) + f_6(s)$$

$$f_i(s) = \sum_{k=1}^{8} weights[\, iso_k \,] f_1(s)$$

其中$V(s)$是版面 s 的估計分數、$f_i(s)$表示第 i 個 pattern 的分數、iso_k表示第 k 個 isomorphic 的版面所代表的 weight index。以下圖為例：

藍色的為一種 isomorphic，則iso_k可以表示成 0x510132，對應的程式碼為下方程式。




```
size_t indexof(const std::vector<int>& patt, const board& b) const {
        // TODO
        size_t mask = 0x0000;
        for (int idx = 0; idx < patt.size(); ++idx) {
                unsigned int tile = b.at(patt[idx]);
                mask |= ( tile << (4*idx) ); // shift left 4 bits based on its index
        }
        return mask;
}
```

$f_i(s)$ 適用於估計一個 pattern 的分數，為所有 isomorphic 的分數和，如下方程式所示：

```
virtual float estimate(const board& b) const {
        // TODO
        float value = 0.0f;

        for (auto patt: isomorphic) {
                size_t mask = indexof(patt, b);
                value += weight[mask];
        }
        return value;
}
```

■ Explain the mechanism of TD(0).

TD(0) 是一種只考慮S_t, S_{t+1} 關係的 model-free 演算法。實作中，每一次玩的過程都會記錄為一段 episode。Episode 紀錄了S_0~S_T（T 是 episode 的長度）每個狀態做的動作a_t和得到的 reward r_t。(S_t做了動作後會轉移到S_t'，S_t'根據隨機生成 2 或 4 後，才會進到具有隨機性的狀態S_{t+1})

在進行更新參數時，由於$V(s)$是用於估計具有隨機性的版面未來的 total reward，因此不能使用S_t'(非隨機性的版面)。學習的過程中，必須從 episode 的尾端往回更新參數（也就是 T~1），因為$V(S_T) = 0$可以視為 ground truth，TD(0)希望

$V(S_T)$的估計值愈接近 0 愈好，$V(S_{T-1})$ 的估計值愈接近$V(S_T)+r_{T-1}$愈好，以此類推，可以根據以下公式更新參數：

$$V(S_t) = V(S_t) + \alpha\,(r_t + V(S_{t+1}) - V(S_t))$$

ps: learning 時，為了更快達到好效果，我使用線性的 learning rate 變動，從 0.5 遞減至 0.01，可以在早期就快速達到很好的 2048 win rate。

- **Describe your implementation in detail including action selection and TD-backup diagram.**

1. Action selection

當位於狀態S_t 時，我需要評估不同動作下可以得到的 total reward，並選擇未來 total reward 最大的動作。在採取動作a_t時，遊戲會立即給我們一個 reward r_t 表示執行a_t後，有多少方塊聚合成功得到的分數，此時的狀態為S_t'。

因為$V(s)$適用於估計具有隨機性的版面，但是S_t'為移動後的版面，不能直接套用$V(s)$估計。因此我需要根據 popup 的機率去統計所有可能的$V(S_{t+1})$的期望值，進而做出最好的動作估計$\widehat{a_t}$:

$$\widehat{a_t} = \underset{a\in\{1,2,3,4\}}{\operatorname{argmax}}\ r_t + E[V(S_{t+1})|a]$$

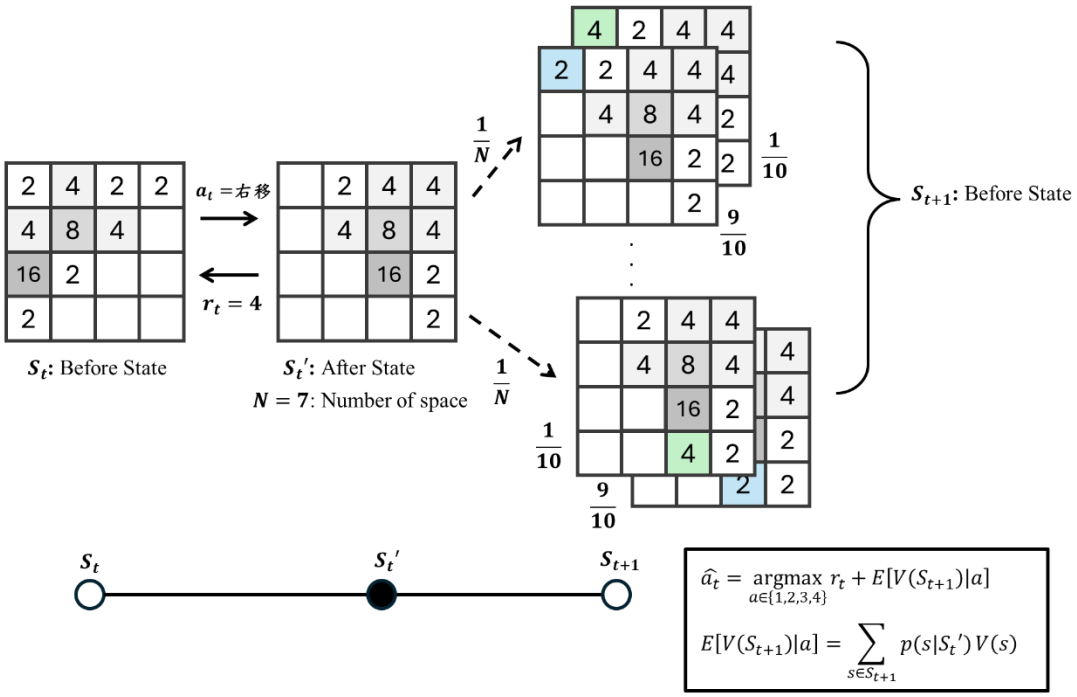
$$E[V(S_{t+1})|a] = \sum_{s\in S_{t+1}} p(s|S_t')\,V(s)$$

具體的實作方式為找出做了動作a後的 after state S_t'的所有空白區域，並考慮每個空白區域的有 0.9 的機率生成 2 和 0.1 的機率生成 4，並將所有的數值求合並除以空白區域的數量，讓機率總合為一，如下方程式碼所示：

```
state select_best_move(const board& b) const {
        state after[4] = { 0, 1, 2, 3 }; // up, right, down, left
        state* best = after;
        for (state* move = after; move != after + 4; move++) {
                if (move->assign(b)) {
                        // TODO
                        board after_b = move->after_state();
                        std::vector<int> space_positions;
                        float expected_value = 0.0f;

                        // collect all space positions
                        for (int i = 0; i < 16; ++i) {
                                if (after_b.at(i) == 0)
                                        space_positions.push_back(i);
                        }
                        for (auto pos: space_positions) {
                                board tmp(after_b);
                                // simulate 0.9 as 1
                                tmp.set(pos, 1);
                                expected_value += (estimate(tmp) * 0.9);

                                // simulate 0.1 as 2
                                tmp.set(pos, 2);
                                expected_value += (estimate(tmp) * 0.1);
                        }

                        expected_value /= (space_positions.size());

                        move->set_value( move->reward() + expected_value );

                        if (move->value() > best->value())
                                best = move;
                } else {
                        move->set_value(-std::numeric_limits<float>::max());
                }
                debug << "test " << *move;
        }
        return *best;
}
```

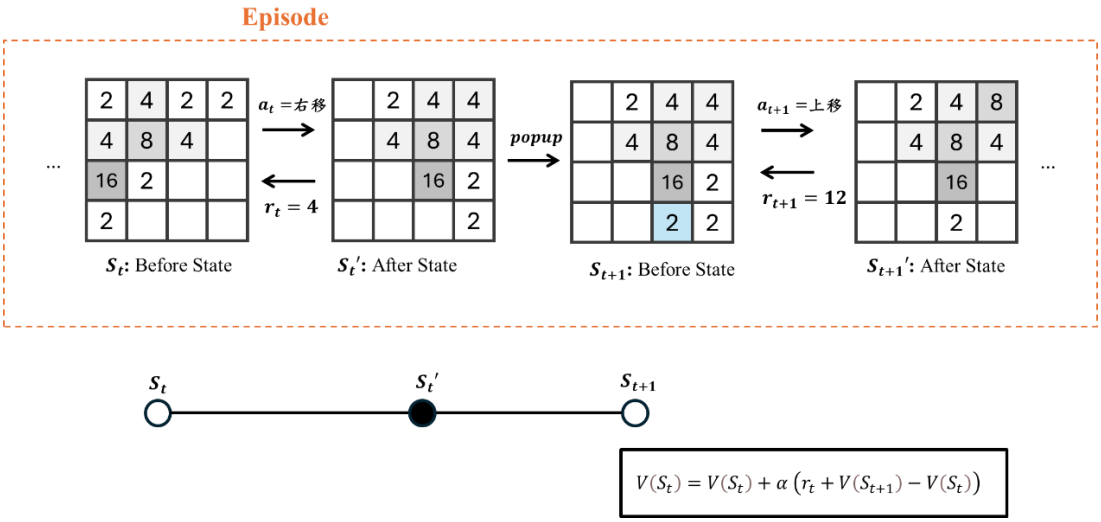
2. TD-backup diagram

做決定的 backup diagram：最佳的動作需要考慮所有未來 popup 的結果的分



數期望值。

學習的 backup diagram，嘗試讓時間 t 的預期分數逼近 t+1 與 reward 的總和，由後往前更新，因為最後一個的結束版面的估計值應該逼近 0：



實際的實作如下方程式碼：

```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
        // TODO
        float latter_val = 0.0f;
        for (int i = path.size() - 1; i >= 0; --i) {
                state former = path[i];
                int reward;
                float former_val, diff;
                reward = former.reward();
                former_val = estimate(former.before_state());
                diff = latter_val + reward - former_val;
                latter_val = update(former.before_state(), diff * alpha);
        }

}
```