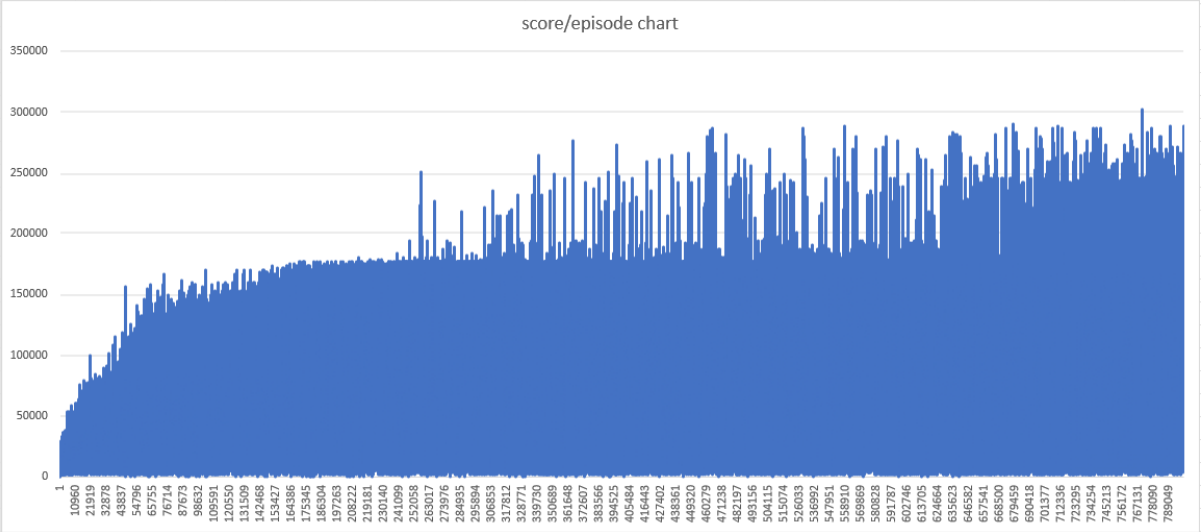
1.A plot shows episode scores of at least 100,000 training episodes



我對6個6-tuple訓練了800000episode，上圖為score/episode圖。
折線圖之所以如此上下波動極大，是因為分數跟實際上該局達到的最高數字有關，且我也沒有取平均分數來畫圖。不過從這張圖中，還是可以看的出來有越訓練越好，且最後收斂的趨勢。

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

實作主要可以分成兩個重點:
玩遊戲和更新參數，這有點像DL中forward和back propagation+gradiant descent的過程。
首先是玩遊戲，這次作業透過計算/評估V(state)的值，去計算每個狀態的action。
每玩完一場遊戲後，就更新參數來訓練V(state)，而這份作業選擇了Temporal-difference方法，去更新每個V(state)。

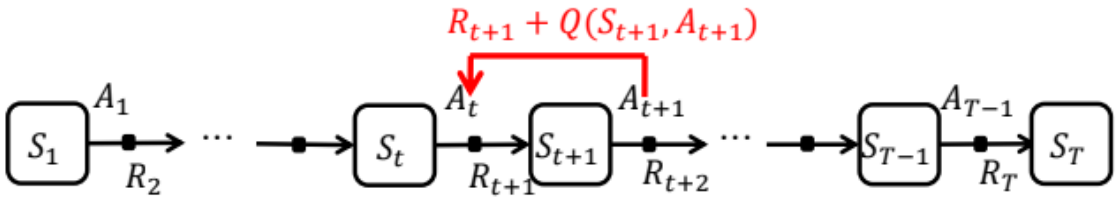
上面的作法很理想，不幸的，如果每種盤面都算一種state的話，每一格總共有18種數字的可能(2^1~2^17+empty)，一個盤面總共有16格，所以總共需要18^16~=1.2*10^21格array去存V(state)，而且有許多盤面幾乎不可能發生，顯然不實際。
為了省空間，且讓程式只看重要的部分(甚麼部分重要滿看經驗與嘗試)，於是我們讓程式一次只看部分重要的區域(像這次作業6-tuple就是一次只看6格)，假如兩個不同盤面，其部分區域一樣，那程式就認為說這兩個盤面是相同state。
這就是n-tuple的用途，然後n-tuple的n代表一次看幾格。

至於以上的實作細節，留到後面回答。

3.Explain the mechanism of TD(0)

TD(0)是一種更新state/after state的value的方法，其結合了蒙地卡羅法與動態規劃。雖然不像蒙地卡羅法unbiased，不過他考慮了下一步的value，在選擇action時比較考慮近期未來的state，比較快收斂，至於TD細節留到4和6回答。

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

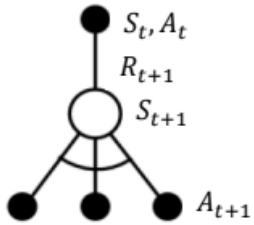


圖如上，該episode結束後，從最後面after_state的value更新回前面after_state的value。
After-state是指說做出action，且環境還沒因其他因素變動(像2048這邊就是每移動一次後會隨機出現一個tile)後的state。然後這個方法是對有走過的after state更新其value 。
公式如下:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

助教給的after state 虛擬碼比較像是sarsa，然後r=1。
公式有點gradiant descent的感覺，就是對原本V(after state)多增加一個TD error。TD error的部份，跟蒙地卡羅法的TD target相比，就是將total reward換成下一回合的reward與下回合的V(after state)。
其好處是可以從action selection時順便存取下個state的V(after state)，跟V(state)比速度較快。

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

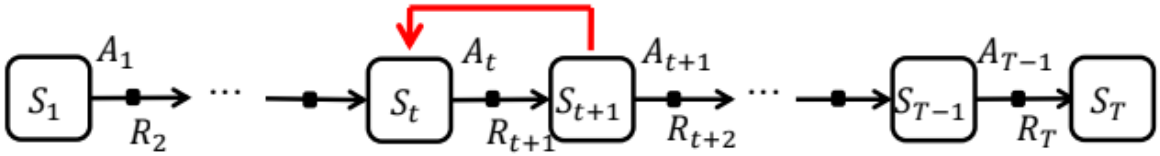


公式如下:

$$R_{t+1} + Q(S_{t+1}, A_{t+1})$$

Action selection的部份，會對移動前的state的所有動作去計算reward+其V(after state)=Q(s,a)，然後取最大的值的action當作action。跟action selection of V(state)比，好處是只要取一次Q(s,a)，速度顯著提升。

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

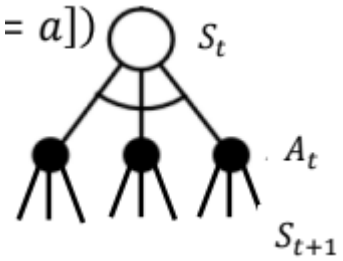


圖如上，該episode結束後，從最後面state的value更新回前面state的value。
State是指說做出action前的狀態，然後這個方法是對還沒走之前的state更新其狀態。
公式如下:

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

公式有點gradiant descent的感覺，就是對原本V(state)多增加一個TD error。TD error的部份，跟蒙地卡羅法的TD target相比，就是將total reward換成下一回合的reward與下回合的V(state)。
但在實作時發現V(s)似乎不能在action selection時先提前存起來，所以重新計算V(s)算是這個方法的缺點吧。

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



Action selection的部份，會對移動前的state的所有動作去計算以下公式:

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

也就是會算動作的reward和未來可能發生的state的V(s)期望值，然後取最大的值的action當作action。
在實作時顯然感覺得出計算期望值的緩慢，實測結果action selection of V(state)大概比action selection of V(after state)慢16倍。

8.Describe your implementation in detail.

這份作業總共有5個TODO要實作:
(1)function indexof

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

這個function是要將n-tuple取到的board的value進行編碼，最後變成pattern array的index。

像說假如有個盤面如下
board index:
 0 1 2 3
 4 5 6 7
 8 9 10 11
12 13 14 15

value:
2 4 2 8
8 8 8 8
8 8 8 8
8 8 8 8

pattern取(0,1)
因為board[0]=2^1 board[1]=2^2
所以最後用4bit編碼會變成00100001

(2)function estimate

```
/**
 * estimate the value of a given board
 */
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;
}
```

默認情況下，isomorphic的用途是用來對n-tuple進行鏡射和旋轉0,90,180,270度，一共8個值。這樣做才會對盤面的評分有旋轉/鏡射不變性。
然後function estimate的功能就是取出目前board的n-tuple對應到的分數。
這裡的做法就是將n-tuple進行鏡射和旋轉0,90,180,270度，將八種情況下的分數加起來再return。
(3)function update

```
/**
 * update the value of a given board, and return its updated value
 */
virtual float update(const board& b, float u) {
    // TODO
    float u_split = u / iso_last;
    float value = 0;
    for (int i = 0; i < iso_last; i++) {
        size_t index = indexof(isomorphic[i], b);
        operator[](index) += u_split;
        value += operator[](index);
    }
    return value;
}
```

function update的功能就是更新目前board的n-tuple對應到的分數。
跟function estimate一樣，同時得考慮鏡射/旋轉共8個value。
所以update這邊就把error分成八等分，平均更新在每個value。

(4)function select best move

```
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
            float error=0;
            int free_space_num = 0;
            board after_state = move->after_state();
            for(int i = 0; i < 16; i++){
                if(after_state.at(i) == 0){
                    free_space_num++;
                    board add_2_state = after_state;
                    board add_4_state = after_state;
                    add_2_state.set(i,1);
                    add_4_state.set(i,2);
                    error += 0.9* estimate(add_2_state) + 0.1* estimate(add_4_state);
                }
            }
            if(free_space_num == 0){
                free_space_num = 1 ;
            }
            move->set_value(move->reward() + error/float(free_space_num));
            if (move->value() > best->value())
                best = move;
        } else {
            move->set_value(-std::numeric_limits<float>::max());
        }
        debug << "test " << *move;
    }
    return *best;
}
```

其實就是照著7.提到的演算法刻，也就是把action上下左右分別做一次，去計算說哪個動作的reward和期望值V(s)總和最大，然後做出相對應的動作。
期望值的計算，就先將每一個空的tile先產生2(90%)後的V(s)和產生4(10%)後的V(s)做加權平均，然後再將每一個空的tile得出的加權平均再做一次一般的平均。

(5)function update episode

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

其實就是照著6.提到的演算法刻，也就是計算TD error(reward+V(s_next)-V(s))後乘上alpha，然後再引用function update(這個function不是(1)提到的)更新。

這裡有值得注意的細節，首先要在for迴圈外放一個V(s_next)變數，去紀錄上次更新完的V(s_next)，當然也是可以用function estimate取V(s_next)，但是速度會變慢。然後V(s)沒辦法先存起來，只能用function estimate取出。

9.Other discussions or improvements.

我有分別跑了

4*6pattern(就完全沒改過的) 50萬episode

6*6pattern(我繳交的檔案) 80萬episode

```
tdl.add_feature(new pattern({ 0, 1, 2, 3, 4, 5 }));
tdl.add_feature(new pattern({ 4, 5, 6, 7, 8, 9 }));
tdl.add_feature(new pattern({ 0, 1, 2, 4, 5, 6 }));
tdl.add_feature(new pattern({ 4, 5, 6, 8, 9, 10 }));
tdl.add_feature(new pattern({ 0, 1, 2, 3, 4, 8 }));
tdl.add_feature(new pattern({ 4, 5, 6, 7, 8, 12 }));
```

和10*6pattern 50萬episode

```
tdl.add_feature(new pattern({ 0, 1, 2, 3, 4, 5 }));
tdl.add_feature(new pattern({ 4, 5, 6, 7, 8, 9 }));
tdl.add_feature(new pattern({ 0, 1, 2, 4, 5, 6 }));
tdl.add_feature(new pattern({ 4, 5, 6, 8, 9, 10 }));
tdl.add_feature(new pattern({ 0, 1, 2, 3, 4, 8 }));
tdl.add_feature(new pattern({ 4, 5, 6, 7, 8, 12 }));
tdl.add_feature(new pattern({ 0, 1, 2, 4, 5, 8 }));
tdl.add_feature(new pattern({ 0, 1, 2, 3, 5, 6 }));
tdl.add_feature(new pattern({ 4, 5, 6, 7, 9, 10 }));
tdl.add_feature(new pattern({ 1, 4, 5, 6, 7, 9 }));
```

我觀察到，對於1024以下(含)的勝率，基本上只要不要重複或是出現subset，想辦法堆pattern，1024的勝率都會分別提高個0.5~1%。

但高一點的數字就不一定，像16384。

我覺得是因為小的數字很早就達到收斂，基本上能拉高小數字的做法只有拉高upper bound，也就是要給更多的information，就算訊息過多帶來干擾，也因為盤面很空，不至於有過多影響，因此我認為堆pattern的做法對小數字是有效的。

但大數字的組成，因為空間不足，我認為跟有沒有走對每一步比較有關，這時增多pattern不一定有效。

然而這麼做也是要付出代價的，像我10*6的pattern在跑100000~200000episode花的時間，比我跑6*6的pattern在跑600000~700000花的時間還多。

可惜的是，因為demo時間有限，我不可能在5分鐘內跑完10*6的pattern 1000次，所以我只能繳交6*6pattern的版本了。