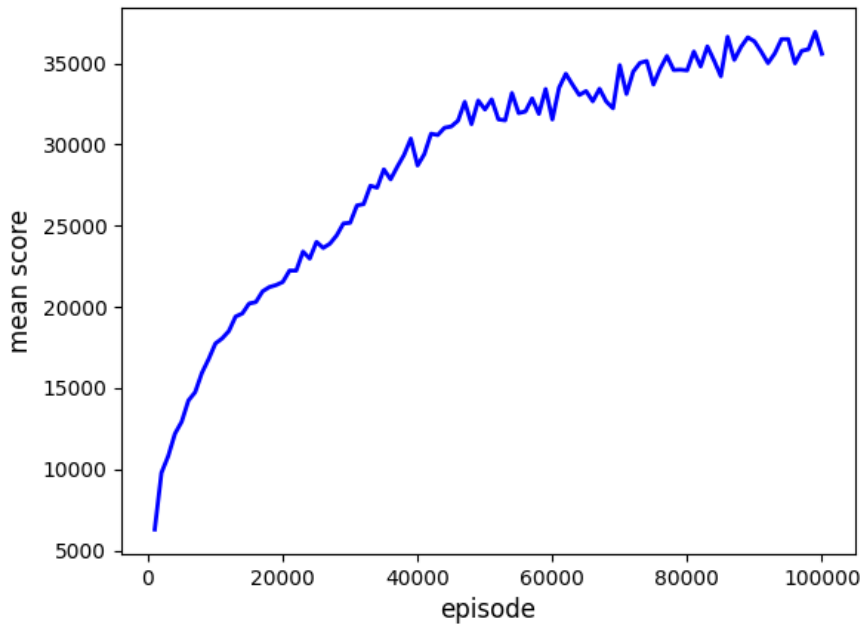
311551069 余忠旻 Lab2 : Temporal Difference Learning

1. A plot shows scores (mean) of at least 100k training episodes

alpha = 0.1, run 100000 episodes (x: #episode, y: mean score)



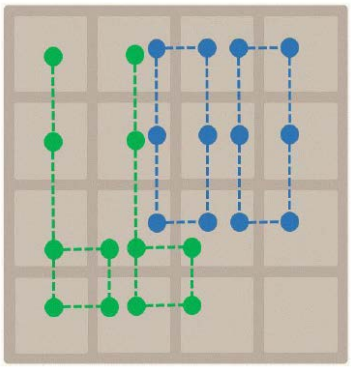
2. Describe the implementation and the usage of n-tuple network.

在 2048 這 4*4 盤面的遊戲中，盤面能玩出的的最高數字是 65536 (2^{16})，也就是在這盤面可能會出現: 2的零次方、2的一次方、2的二次方、2的三次方、……、二的十六次方，總共有17個數可能會出現在盤面上，而盤面有16個格子，因此所有可能性是17^{16} 種可能，這很明顯是很大的量。

我們要做的事是把這 4*4 盤面轉為盤面的局部資訊來儲存，並且透過這些局部資訊來估計這個盤面的好壞。有一個好的盤面代表說在後續的 action 能更容易拿到更高分。為了好儲存盤面資訊，我們假設盤面會出現的數字是 32768(2^{15}) ，並將每個格子數字轉成2的次方數，每個格子用 4-bit 儲存，所以能存 0~15 (代表數字 $2^0 \sim 2^{15}$)，這樣一個盤面可以用 64(4*16) bits 儲存，恰好是一個 unsigned long long 的大小。

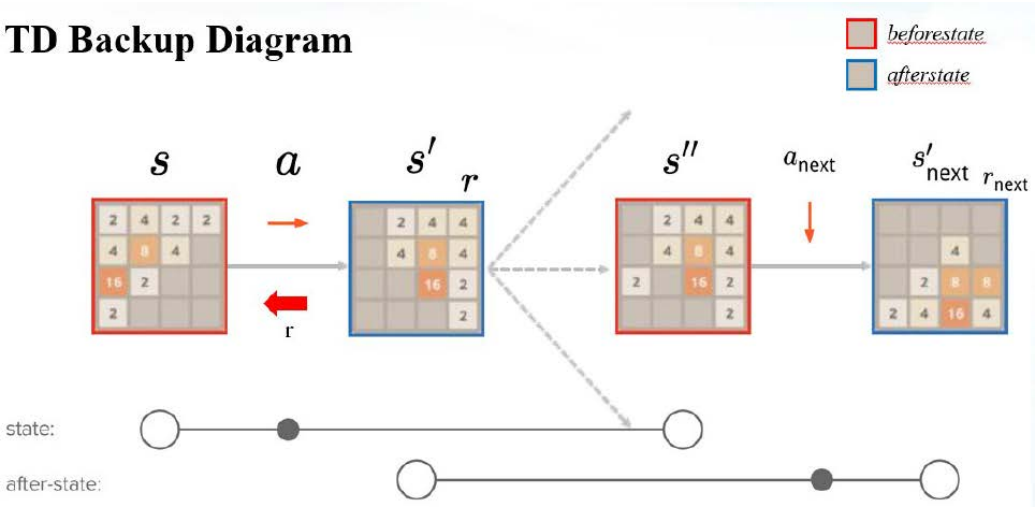
而 tuple network 的特性是 provide with a large number of features & easily update，我們在這次作業中是使用 4*6-tuple network，我們會取得 4 個局部資訊的 value，把他們相加起來代表一個盤面的估計值，一個 6-tuple 需要 24(4*6)bits 來儲存，4*6-tuple 則需要 112(4*24)bits 儲存，而一個 6-tuple 有 16777216(2^{24}) 種可能性，這些可能性就代表局部盤面的好壞程度，盤面越好value越高，將四個 6-tuple 的 value 值相加起來就能估計整個盤面的好壞程度。另外在訓練時會考慮盤面的 isomorphic patterns，也就是透過旋

轉和鏡射會出現8種 patterns (也就是旋轉0°, 90°, 180°, 270° 以及水平鏡射後的旋轉0°, 90°, 180°, 270°)，考慮 isomorphic patterns 能加速訓練速度，而且 4*6-tuple (考慮isomorphic patterns) 相比 8*4-tuple (不考慮isomorphic patterns)可以大量節省所需記憶體。下圖是 4*6-tuple 的樣子:



3. Explain the mechanism of TD(0).

TD learns from incomplete episodes, by bootstrapping，並且 TD updates a guess towards a guess，而 TD(0) 代表是用下一個 state V(s_{t+1}) 來更新當下這個 state V(s_t)，也就是走了一步後做 bootstrapping。



更新的方法則是用 TD-backup，當運行完整個 episode，我們會由後往前去對每個 state 的估計值 V(s) 去更新。更新的公式如下:

$$V(s) \leftarrow V(s) + \alpha(r + V(s'') - V(s))$$

state s 執行 action a 後會得到一個 reward r & state s'，並且由環境領導至 state s''，若 r+V(s'') 的值比 V(s) 高，代表當下 state s 是一個好的 state，因為它可以透過後續的動作取得更高分，所以我們應該增加 V(s) 值，反之，假如 r+V(s'') 的值比 V(s) 低，代表當下 state s 不是一個好的 state，所以我們應該減少 V(s) 的值來避免之後的 episode 走到這個 state，這也

就 TD 的運作原理和上述公式代表的意義。其中 α 代表 learning rate，控制每次更新的幅度大小。

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

這次作業有五個 TODO 的地方

第一個出現在 pattern class 的 estimate function，一個 tuple 取出來要考慮8種 isomorphic patterns (也就是旋轉0°, 90°, 180°, 270° 以及水平鏡射後的旋轉0°, 90°, 180°, 270°)，因此我們需要把這8種局面盤面估計值相加起來，取值時會用到 index function，這是後面 TODO 的其中一個。

```
/**
 * 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){
        // [] operator defined in class feature
        size_t index = indexof(isomorphic[i], b);
        value += (*this)[index];
    }
    return value;
}
```

第二個是 pattern class 的 update function，在更新時也是要考慮8種 isomorphic patterns，也就是每一個 pattern 更新的值要平均分給這8種 isomorphic patterns。
因此整個 4*6-tuple network，更新時首先會在 learning class 的 update function 除以 4 來分給 4 個 pattern，每個 pattern 有8種isomorphic patterns，會在這裡除以 8 分給 isomorphic patterns，因此 td-error 會除以 32 來平均分給所有 pattern 的 isomorphic patterns。

```
/**
 * 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++){
        // sum of splits for isomorphic patterns (one feature)
        size_t index = indexof(isomorphic[i], b);
        (*this)[index] += u_split;
        // update weights
        value += (*this)[index];
    }
    return value;
}
```

下一個是 index function，也就是前面所講的，每一個格子要用 4-bit 代表，存 0~15 (代表數字 $2^0 \sim 2^{15}$)，而每一個 tuple 就是取六個格子(24-bit)。實作就是用 b.at() function 把每一個格子的值取出來，接著要把 tuple 對應的六個格子 concatenate，透過向左 shift 四位再做 or，這樣 bit 之間就不會有所重疊了。最後的到 24-bit 就是 index 值。

```
size_t indexof(const std::vector<int>& patt, const board& b) const {
    // TODO
    /*
    +------------------------+
        |     2     8   128     4|
        |     8    32    64   256|
        |     2     4    32   128|
        |     4     2     8    16|
        +------------------------+
        board index 15 = b.at(15) = 16 => 4
        board index  1 = b.at(1)  = 8 => 3

        1-tile => 4 bits
        pattern.size = 6 => 24 bits
    */

    size_t index = 0;
    for (int i = 0; i < patt.size(); i++){
        index |= (b.at(patt[i]) << (i * 4));
    }

    // 24-bit index for pattern
    return index;
}
```

接著是 action selection 部分，state s 執行 action a 之後會得到 reward r & state s’，並由環境領導至 state s’’，但是選擇 action 時，我們並不知道環境會將我們引領到哪個 state，也就是 s’’ 是未知的，所以我們應該透過計算 V(s’’) 的期望值，來當作下一個 state s’’ 的值，因此要選擇的 action 就是 max(r + E(V(s’’)))。
實作部分，我則是額外寫一個 evaluate function 來計算 E(V(s’’))，而計算期望值就會需要用的P(popup tile 2) = 0.9 and P(popup tile 4) = 0.1 這個資訊加上每個空格子 popup tile 的機率是相等的這個資訊來做計算。

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

```
float evaluate(const board& b) const{
    float value = 0.0;
    int emptySpace = 0;
    for (int i = 0; i < 16; i++){
        if (b.at(i) == 0){
            board b1 = b;
            b1.set(i, 1); // place 2 at i, occurrence probability: 0.9
            board b2 = b;
            b2.set(i, 2); // place 4 at i, occurrence probability: 0.1
            value += (estimate(b1) * 0.9 + estimate(b2) * 0.1);
            emptySpace += 1;
        }
    }
    if (emptySpace == 0){
        return 0;
    }else{
        return value / emptySpace;
    }
}
```

最後是 update_episode function

根據上述 Q3. 所提到 TD-backup，當運行完整個 episode，我們會由後往前去對每個 state 的估計值 V(s) 去更新。而 state s 執行 action a 後會得到一個 reward r & state s'，並且由環境領導至 state s''，藉由 r+V(s'') 的值比 V(s) 的值比較可以得知當下 state s 好壞，下面公式就是藉由此過程來更新整條 episode:

$$V(s) \leftarrow V(s) + \alpha(r + V(s'') - V(s))$$

```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    //We have to update 0 to terminal afterstate so that it can converge.
    path.pop_back(); // remove final invalid move
    update(path[path.size() - 1].before_state(), 0);
    //The backward method updates the afterstates from the end to the beginning.
    for(int i = path.size() - 2; i >= 0; i--){
        float td_target = path[i].reward() + estimate(path[i+1].before_state());
        float td_error =  td_target - estimate(path[i].before_state());
        float adjustment = alpha * td_error;
        update(path[i].before_state(), adjustment);
    }

}
```

Advanced skill : expectimax

Ref: K. Matsuzaki, “Systematic selection of N-tuple networks with consideration of interinfluence for game 2048,” DOI: 10.1109/TAAI.2016.7880154.

在上述 action selection 中，我們選擇的是 r + E(V(s’’)) 值最大的 action，我們可以在往下多看一層盤面，也就是選擇 r + E(max(r’ + V(s’’’))) 值最大的 action，往下多看一層盤面能讓 action 選擇更加精確。

```
// evaluate one more level
float evaluate_expectimax(const board& b) const{
    float value = 0.0;
    int emptySpace = 0;
    for (int i = 0; i < 16; ++i){
        if (b.at(i) == 0){
            board b1 = b;
            b1.set(i, 1); // place 2 at i, occurrence probability: 0.9
            board b2 = b;
            b2.set(i, 2); // place 4 at i, occurrence probability: 0.1

            state after1[4] = { 0, 1, 2, 3 }; // up, right, down, left
            float value1 = -std::numeric_limits<float>::max();
            for (state* move = after1; move != after1 + 4; move++) {
                if (move->assign(b1)) {
                    float v1 = move->reward() + estimate(move->after_state());
                    if (v1 > value1) value1 = v1;
                }
            }
            state after2[4] = { 0, 1, 2, 3 }; // up, right, down, left
            float value2 = -std::numeric_limits<float>::max();
            for (state* move = after2; move != after2 + 4; move++) {
                if (move->assign(b2)) {
                    float v2 = move->reward() + estimate(move->after_state());
                    if (v2 > value2) value2 = v2;
                }
            }
            value += (value1 * 0.9 + value2 * 0.1);
            emptySpace += 1;
        }
    }
    if (emptySpace == 0){
        return 0;
    }else{
        return value / emptySpace;
    }
}
```

這個方法雖然能使 action 選擇更加精確，但計算量多很多，所以我拿前面用一般 evaluate function 來訓練，而這 evaluate_expectimax function 則是拿來跑最後結果，可以看到 winrate_{2048} 提高了不少。
如下圖，原本的 action selection 的 winrate_{2048} 是 71.4%，用了 expectimax(多看一層盤面) winrate_{2048} 提高到 94.4%。

action selection: r + E(V(s’’))

```
TDL2048-Demo
alpha = 0
total = 1000
seed = 4084115498
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)
6-tuple pattern 012345 is loaded from weights.bin
6-tuple pattern 456789 is loaded from weights.bin
6-tuple pattern 012456 is loaded from weights.bin
6-tuple pattern 45689a is loaded from weights.bin
1000    mean = 43669.4  max = 110532
        32      100%    (0.3%)
        64      99.7%   (0.8%)
        128     98.9%   (0.2%)
        256     98.7%   (1.2%)
        512     97.5%   (5.3%)
        1024    92.2%   (20.8%)
        2048    71.4%   (26.7%)
        4096    44.7%   (44.3%)
        8192    0.4%    (0.4%)
```

action selection: r + E(max(r’ + V(s’’’)))

```
TDL2048-Demo
alpha = 0
total = 1000
seed = 3166521824
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)
6-tuple pattern 012345 is loaded from weights.bin
6-tuple pattern 456789 is loaded from weights.bin
6-tuple pattern 012456 is loaded from weights.bin
6-tuple pattern 45689a is loaded from weights.bin
1000    mean = 71173.5  max = 153868
        256     100%    (0.3%)
        512     99.7%   (0.4%)
        1024    99.3%   (4.9%)
        2048    94.4%   (12.8%)
        4096    81.6%   (73.1%)
        8192    8.5%    (8.5%)
```