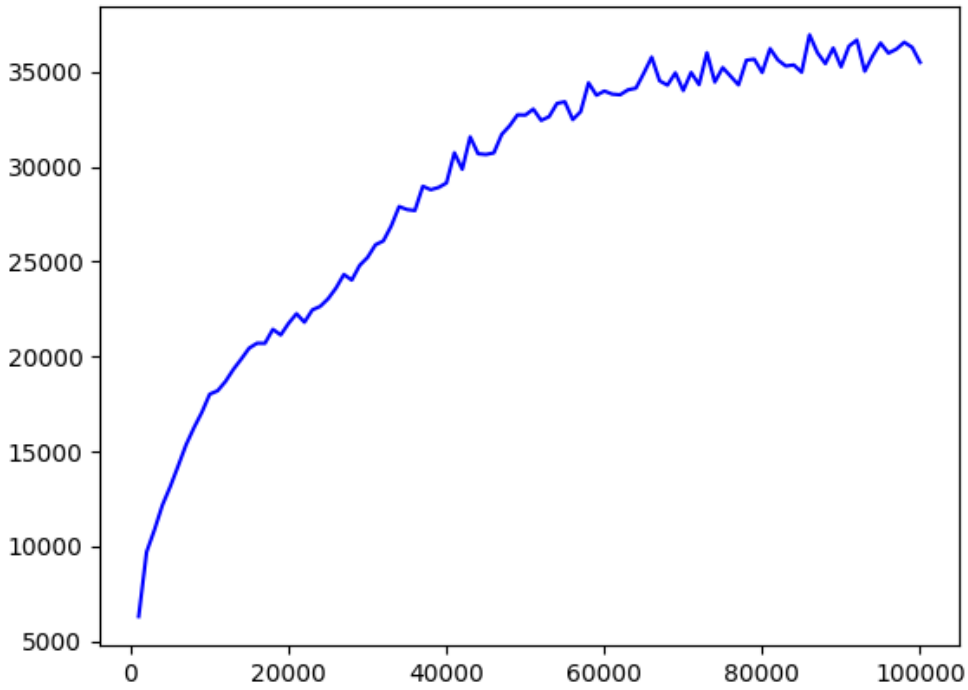
Lab3 Report

a) **A plot shows episode scores of at least 100,000 training episodes**

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



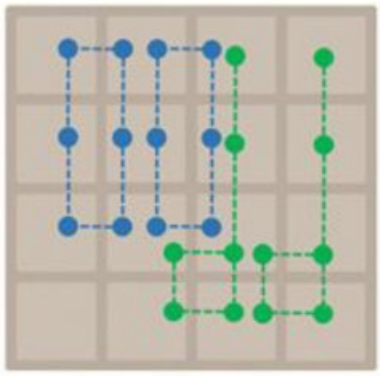
b) **Describe the implementation and the usage of *n*-tuple network**

首先，假設盤面上能玩出的最高數字是 65536，也就是 2 的 16 次方。那麼盤面的 16 個格子中，每個格子可以出現的數字就是 2 的 0 次方、1 次方、2 次方...到 2 的 16 次方。總共有可能出現 17 個數字，整個盤面就有 17 的 16 次方種可能，大約是 5*(10^19)，這明顯是電腦存不下的量。因此，我們轉為儲存一個盤面的局部資訊，並利用這些局部資訊估計這個盤面的好壞。一個好的盤面代表的意義

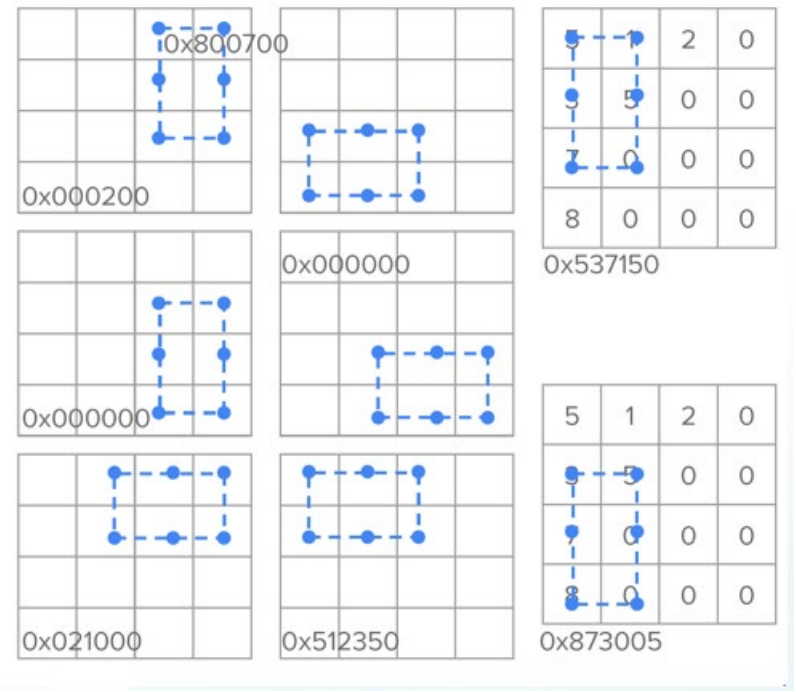
就是我們傾向於能透過後續的 action 得到更高的分數。

為了方便起見，一個格子我們用 4 個 bits 來儲存(代表數字 0~15，即假設最高出現 2^15 = 32768)，這樣一個盤面剛好是 4*16 = 64 個 bits，可以用一個 unsigned long long 來表示。以一個 6-tuple 為例，我們需要 24 個 bits 來存一個 tuple，可以用一個 int 來存。而 24 bits 可能出現的數字量就是 2^24 = 16777216，也就是說這個 tuple 會有 16777216 種可能的局部盤面，而我們要存的就是這 16777216 種局部盤面的 value 值。value 值越高，就代表此盤面越好。

而估計一個盤面可以不只用一個 tuple。以下圖講義上 4x6 tuple 為例:

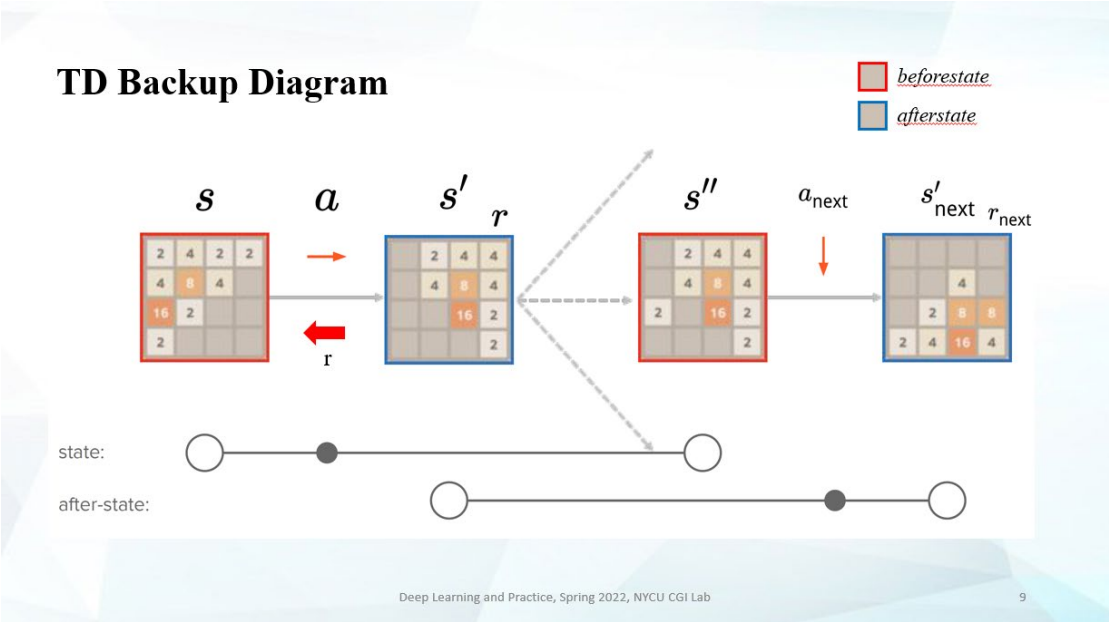


我們會取得 4 個局部盤面的 value 值，並將他們加起來，來代表一個盤面的估計值。因此，若用 4x6 tuple，要存的數字量大約是 4*16777216 個浮點數，這對現代的電腦來說並不是什麼大負擔。至此解決盤面 value 無法儲存的問題。另外，實作上每個 tuple 會有鏡射、旋轉共 8 種型態，如下圖(取自講義):



所以，用 4x6 tuple，每次估計盤面都是 4*8 共 32 個值的加總。

c) Explain the TD-backup diagram of V(state)



當運行完整個 episode，我們會由後往前對每個 state 的估計值 V(s)進行更新。state s 執行 action a 得到 reward r 和一個 state s’，並由環境引導至 state s”，若 r+V(s”)的值比 V(s)還要高，代表 s 是一

個好的 state，因為它可以透過後續的操作得到更高的分數。那麼我們就應該要增加 V(s)的值，這可以讓之後的 episode 更容易走到這個 state。反之，若 r+V(s")的值比 V(s)還低，代表 V(s)不是一個好的 state，因為它之後的 state s"比它自己還爛 (r >= 0，因此 V(s) > V(s"))，所以我們要減少 V(s)的值來盡量避免之後的 episode 走到這個 state。這也就是 TD 的運作原理。

我們使用這個公式

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

來更新 V(s)。當 r+V(s") > V(s)，後項括號內自然為正，V(s)增加；反之則為負，V(s)減少。而 alpha 是 learning rate，是為了避免一次更新造成的變動過大。

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

如 c 所述，state s 執行 action a 得到 reward r 和一個 state s'，並由環境引導至 state s"。但不同的是，選擇 action 時並不知道環境會將我們引導至哪個 state，也就是說 s"是未知的。這個時候就只能列出所有可能的 s"，並取所有可能的 V(s")的平均。因此，要選擇的 action 就是 max(r + E(V(s"))。

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

原理和 c 所述相同，只是更新的方式不同。原本是 state s 執行

action a 得到 reward r 和一個 state s’，並由環境引導至 state s”；現在是 s 由環境引導至 s’，再執行 a 得到 r 和 s”。並用一樣的更新公式。這樣做的意義就是我們將「環境引導」這件事視為 V(s)的一部分，試圖將環境中的機率因素訓練至 V(s)中。如此便不用像 d 所述列舉所有 V(s”)取平均，實作上更加容易且訓練效率更高。

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

如 e 所述，選擇 action 時只需要選出 max(r + V(s”))即可。

g) Explain the mechanism of TD(0)

TD 的原理如 c 所述。TD(0)代表只用下一個 state 來更新當下這個 state，也就是我在這個作業中實作的方法。

h) Describe your implementation in detail

接下來針對五個 TODO 的地方進行說明。第一個是在 pattern class 裡面的 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){
                value += (*this)[indexof(isomorphic[i], b)]; // [] operator defined in class feature
        }
        return value;
}
```

一個 tuple pattern 配合鏡射、旋轉總共有 8 種 isomorphic，因此我們要把 board 中的這 8 個局部盤面估計值給取出來做加總。取值時要用到定義在 feature class 中的[] operator。取的 index 由後面的函式計

算。

第二個則同上，以同樣的方式對 board 取值做 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){
                (*this)[indexof(isomorphic[i], b)] += u_split;
                value += (*this)[indexof(isomorphic[i], b)];
        }
        return value;
}
```

這邊要注意的是，一個 pattern 代表 8 種 isomorphic。所以可以想像成更新的值要平均分給這 8 種 isomorphic。另外，我用的是 4x6 tuple，因此更新的值在 learning class 那邊的 update 要先除以 4，代表平均分給 4 個 pattern。所以更新的值要先被除以 pattern 的數量，再除以一個 pattern isomorphic 的數量，才算是將 td error 的值平均分給所有的 pattern。此例中是除以 4 (#pattern)再除以 8 (#isomorphic/pattern)，共除以 32。根據 TD 的 update 公式也可將這個係數理解成並入 learning rate (alpha)的常數，即 learning rate 為 0.1/32 = 0.003125。但這裡我仍習慣將 learning rate 理解為 0.1。

第三個則是要取得前兩個部分局部盤面 pattern 的 index。

```
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){
                /*
                get 4-bit board index

                ex:
                +-------------------------+
                |     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
                */

                index |= (b.at(patt[i]) << (i * 4));

                // 1-tile => 4 bits
                // patt.size() = 6 => 24 bits
        }

        // return a 24-bit index
        return index;
}
```

我用的是 4x6 tuple，每一個 tuple 要取出 board 中 6 個格子裡的值，用 b.at()函式可以取得。而一個格子的大小是 4 bits (在 b 已解釋)，因此我們將 6 個 4-bit 的數字用 or 串起來，成一個 24-bit 的整數，所以這邊才要 left shift (i*4)再做 or，bit 之間才不會 overlap 造成值錯誤。最後得到的 24-bit 整數就是 index 值。

第四個是選擇 action 的部分。

```
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()));

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

根據 c、d 所述，我們要選擇擁有最大 r + E(V(s'))值的 action。這邊另外用一個 evaluate 函式計算 E(V(s'))。

```
float evaluate(const board& b) const{
        float value = 0.0;
        int cnt = 0;
        for (int i = 0; i < 16; ++i){
                if (b.at(i) == 0){
                        board tmp1 = b;
                        tmp1.set(i, 1); // place 2 at i
                        board tmp2 = b;
                        tmp2.set(i, 2); // place 4 at i
                        value += (estimate(tmp1) * 0.9 + estimate(tmp2) * 0.1);
                        cnt++;
                }
        }

        if (cnt == 0){
                return 0;
        }
        return value / cnt;
}
```

對每一個空格，都有 90%的機率填上 2 和 10%的機率填上 4。所以一個空格期望值就是 estimate(填上 2)*0.9 + estimate(填上 4)*0.1，將每個空格加總起來除以空格數量，就是全部的期望值。

最後是根據 TD 公式 update 整條 episode。

```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
        // TODO
        path.pop_back();
        for(int i = path.size() - 2; i >= 0; i--){
                float td_error = path[i].reward() + estimate(path[i+1].before_state()) - estimate(path[i].before_state());
                update(path[i].before_state(), alpha * td_error);
        }
}
```

這邊就是套上公式而已。值得注意的是必須由後往前更新。

i) Other discussions or improvements

在原本的 select action 中，我們選擇擁有最大 r + E(V(s'))值的 action。這邊我們可以將 V(s')再往下計算一個盤面，也就變成選擇最大 r + E(max(r' + V(s'')))的 action。這表示選擇 action 時可以比原

本多看一層的盤面，可以讓 action 選擇得更加精確。

```
float evaluate(const board& b) const{
        float value = 0.0;
        int cnt = 0;
        for (int i = 0; i < 16; ++i){
                if (b.at(i) == 0){
                        board tmp1 = b;
                        tmp1.set(i, 1); // place 2 at i
                        board tmp2 = b;
                        tmp2.set(i, 2); // place 4 at i

                        state after1[4] = { 0, 1, 2, 3 }; // up, right, down, left
                        float max_value1 = -std::numeric_limits<float>::max();
                        for (state* move = after1; move != after1 + 4; move++) {
                                if (move->assign(tmp1)) {
                                        float tmp = move->reward() + estimate(move->after_state());
                                        max_value1 = std::max(max_value1, tmp);
                                }
                        }

                        state after2[4] = { 0, 1, 2, 3 }; // up, right, down, left
                        float max_value2 = -std::numeric_limits<float>::max();
                        for (state* move = after2; move != after2 + 4; move++) {
                                if (move->assign(tmp2)) {
                                        float tmp = move->reward() + estimate(move->after_state());
                                        max_value2 = std::max(max_value2, tmp);
                                }
                        }

                        value += (max_value1 * 0.9 + max_value2 * 0.1);
                        cnt++;
                }
        }
                                                                        717,4-25
```

這個做法的缺點在於計算量多不少，拿來訓練的話效率會很差。因此我選擇不拿它來訓練、也就是不 update weight，而是拿訓練好的 weight 來直接跑結果。下圖是結果的比較:

before improvement:

```
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 = 36335.6  max = 97184
        32      100%    (0.2%)
        64      99.8%   (0.5%)
        128     99.3%   (1.1%)
        256     98.2%   (2.1%)
        512     96.1%   (10.3%)
        1024    85.8%   (26.5%)
        2048    59.3%   (18.5%)
        4096    40.8%   (40.7%)
        8192    0.1%    (0.1%)
```

after improvement:

```
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 = 49941.4  max = 123824
        128     100%    (0.1%)
        256     99.9%   (0.2%)
        512     99.7%   (3.4%)
        1024    96.3%   (12%)
        2048    84.3%   (18.1%)
        4096    66.2%   (65.5%)
        8192    0.7%    (0.7%)
```

可以看到，兩者在平均分數上有一千多分的差距，2048 的 win rate 也從 59%提升到 84%。