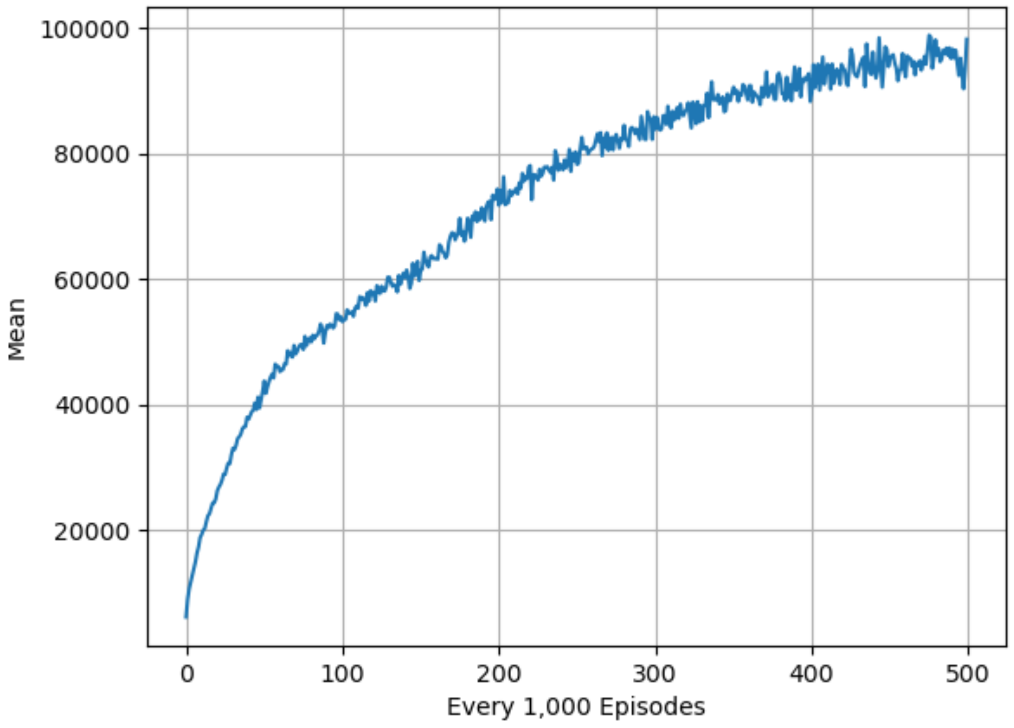
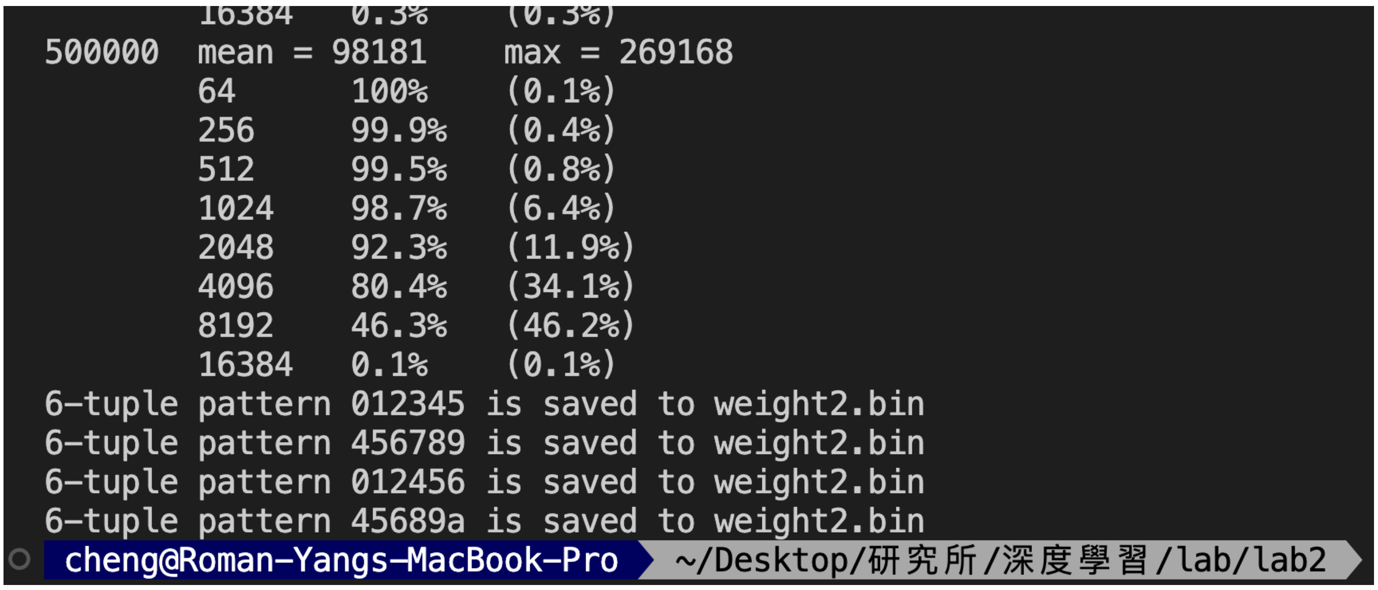
311551180_資工所_楊佳誠

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





```
         16384    0.3%     (0.3%)
500000   mean = 98181     max = 269168
         64       100%     (0.1%)
         256      99.9%    (0.4%)
         512      99.5%    (0.8%)
         1024     98.7%    (6.4%)
         2048     92.3%    (11.9%)
         4096     80.4%    (34.1%)
         8192     46.3%    (46.2%)
         16384    0.1%     (0.1%)
6-tuple pattern 012345 is saved to weight2.bin
6-tuple pattern 456789 is saved to weight2.bin
6-tuple pattern 012456 is saved to weight2.bin
6-tuple pattern 45689a is saved to weight2.bin
○  cheng@Roman-Yangs-MacBook-Pro  ~/Desktop/研究所/深度學習/lab/lab2
```

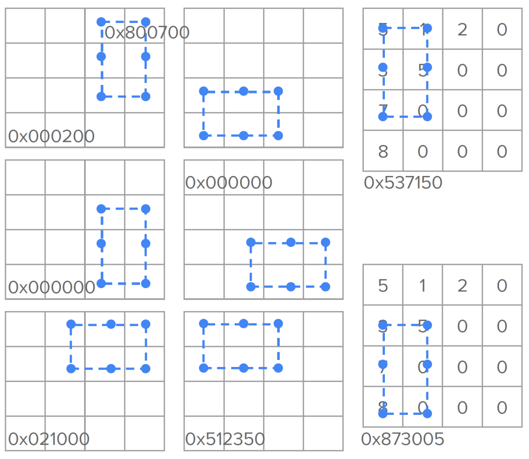
- **Describe the implementation and the usage of *n*-tuple network.**

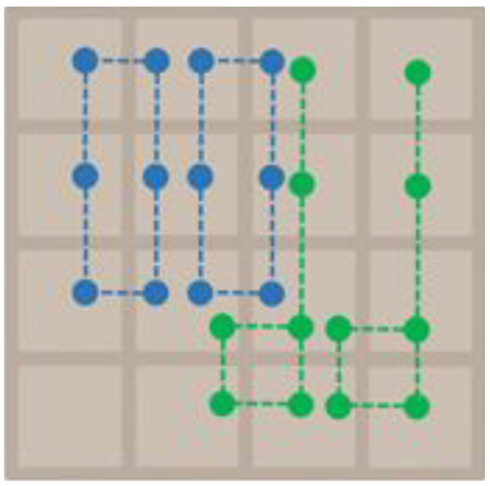
To implement 2048 reinforcement learning, what we should do is to save the board's state. With this value, we can continually update it to improve the greedy policy. However, storing all possible board states is impossible due to the immense number of possibilities.

To address this issue, the n-tuple network decomposes the board into several smaller pieces, reducing the memory needed to store the board's features. These features enable the recognition of the state and allow for easy computation of the board's value.

The sample code provides four features. By rotating and mirroring each feature, we get eight isomorphisms for each. We store an estimated value that corresponds to the feature that captures it.

A data structure is designed to directly find the given weight corresponding to the feature. In this example, a feature has a memory size of 16^6 because each piece of the board can have 16 possible values and the feature consists of 6 pieces. To query the value of a given board, we simply use the board's value as an index to find the corresponding feature weight.





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

The TD(0) algorithm is given as follow:

```
1. Initialize the state s
2. Initialize the Q-value table Q(s,a) for all states s and actions a
3. Set the learning rate alpha and discount factor gamma
4. Repeat for each episode:
    a. Set the current state s as the initial state
    b. Repeat until the terminal state is reached:
        i. Choose an action a based on a policy derived from Q(s,a) (e.g.
epsilon-greedy)
        ii. Take the action a and observe the reward r and the new state s'
        iii. Update the Q-value for the current state and action using the
temporal difference (TD) target:
                Q(s,a) = Q(s,a) + alpha * (r + max(Q(s',a')) - Q(s,a))
        iv. Set the new state s' as the current state
5. Repeat Step 4 for a sufficient number of episodes or until convergence
```

Temporal differences is a reinforcement learning method. Compared to dynamic programming method and Monte Carlo method, temporal differences have two advantages.

First, temporal differences method does not need to go through the whole episode to update the estimated value, which makes it have lower variance than Monte Carlo method.

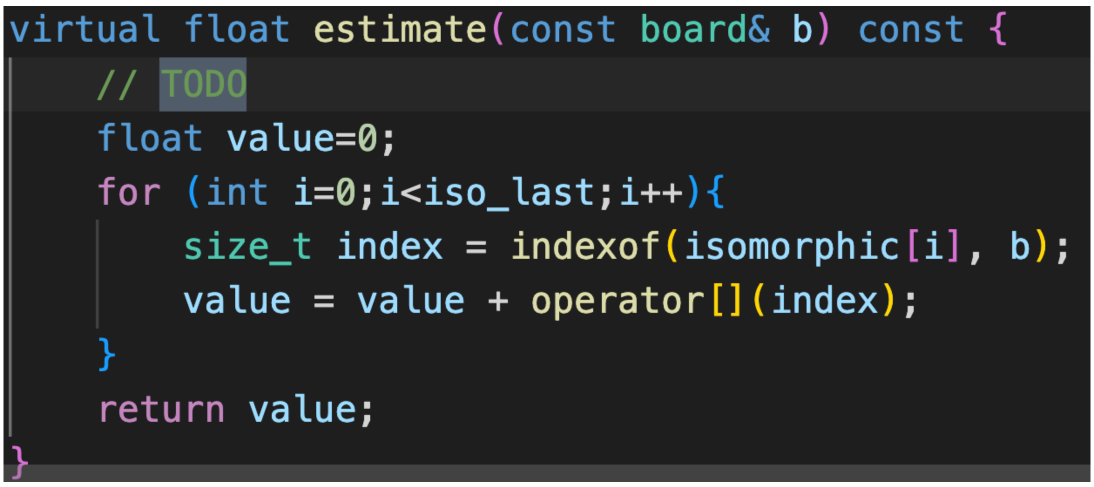
Second, unlike dynamic programming, temporal differences method takes advantage of sampling, which does not need to compute the whole state value when updating the estimate value.

The main idea of temporal difference is to update each state's value by taking a step further. The equation given below shows that we hope to make the Q-value as close as possible to r + max(Q(s',a')). As the game is played numerous times, the estimate value will gradually become closer to the real value. By getting a more accurate Q-value, there is a greater possibility that the action chosen by the largest estimate value (greedy policy) will be the most promising choice.

Q(s,a) = Q(s,a) + alpha * (r + max(Q(s',a')) - Q(s,a))

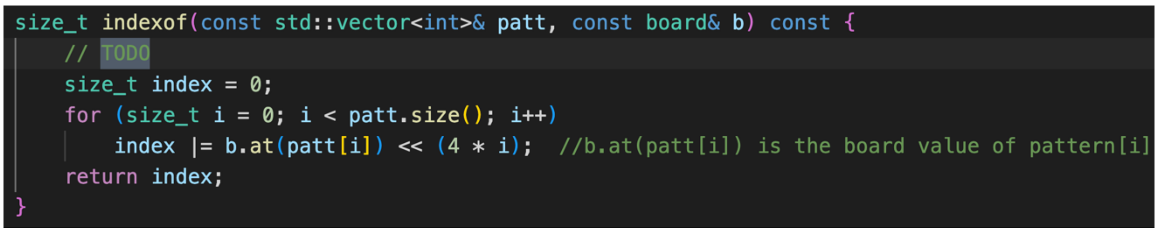
- **Describe your implementation in detail including action selection and TD-backup diagram.**

- This estimate function is to return the feature’ s estimate value. The value is stored in an array indexing by the board’ s value correspond to feature. Note that each feature has 8 isomorphic, so we need to return the accumulate value.



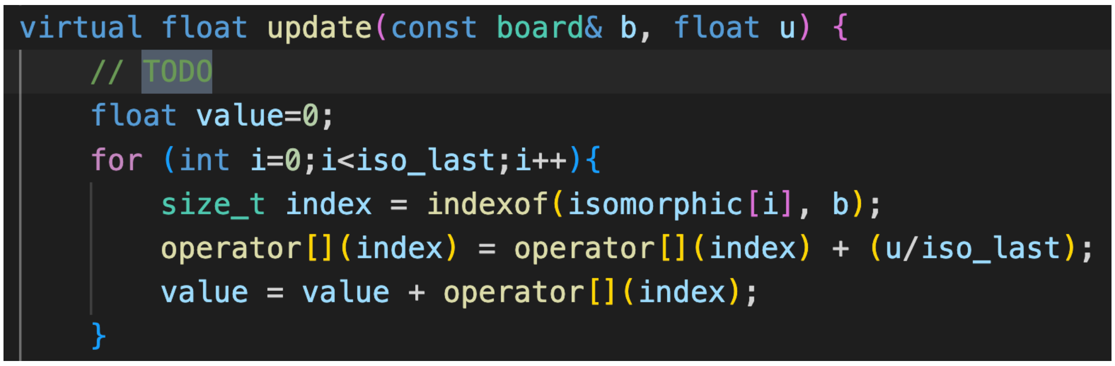
```
virtual float estimate(const board& b) const {
    // TODO
    float value=0;
    for (int i=0;i<iso_last;i++){
        size_t index = indexof(isomorphic[i], b);
        value = value + operator[](index);
    }
    return value;
}
```

- This indexof function is to return the weight’ s index of what feature get in the board.



```
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);  //b.at(patt[i]) is the board value of pattern[i]
    return index;
}
```

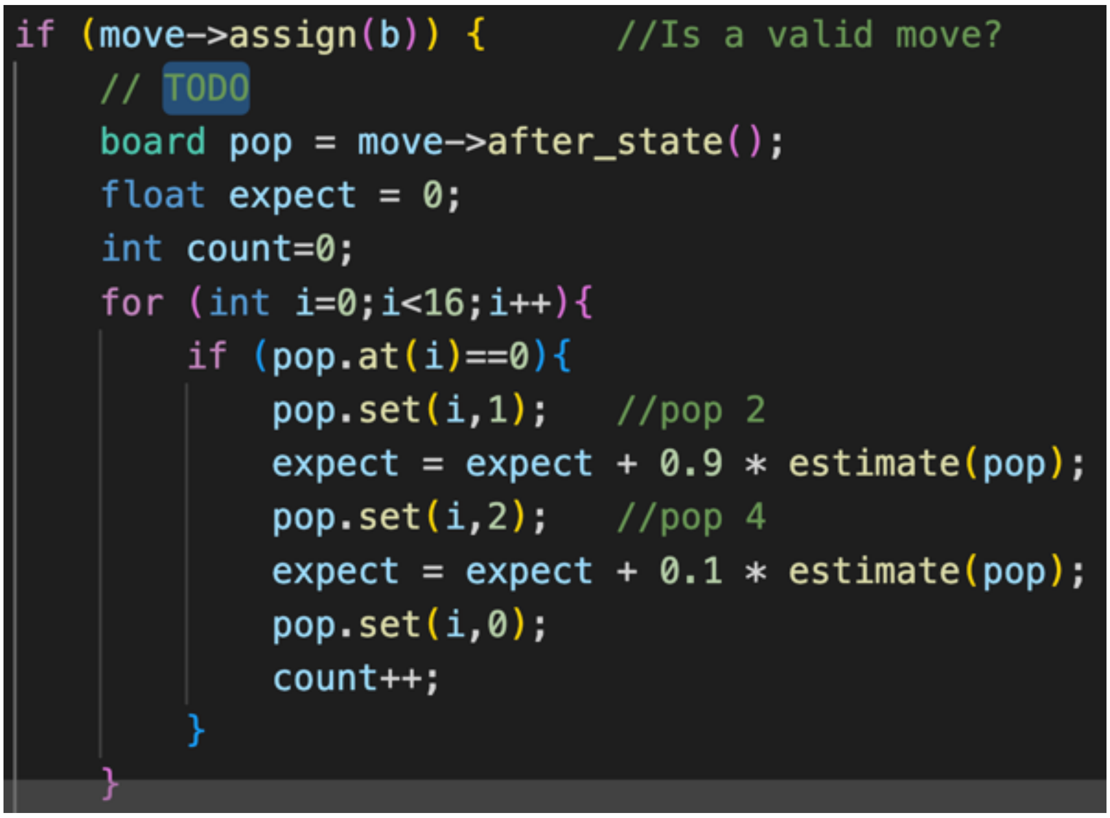
- The update function is to update each weight of what feature get. Float u is the value needed to updated. Since there are 8 isomorphic, u has to divide 8. The new accumulative weight of this feature is returned.



```
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) = operator[](index) + (u/iso_last);
        value = value + operator[](index);
    }
```

- Since we need to use the information of P(popup tile 2) = 0.9 and P(popup tile 4) = 0.1, when choosing next move, we need to compute the expectation value of the board after we move. The tile will pop in the empty tile(denote as 0 in raw), and the probability are different for 2 and 4.

  We take advantage of the class board to use set and at function, and compute the expectation after moved Then we choose the action based on largest value.



```
if (move->assign(b)) {      //Is a valid move?
    // TODO
    board pop = move->after_state();
    float expect = 0;
    int count=0;
    for (int i=0;i<16;i++){
        if (pop.at(i)==0){
            pop.set(i,1);    //pop 2
            expect = expect + 0.9 * estimate(pop);
            pop.set(i,2);    //pop 4
            expect = expect + 0.1 * estimate(pop);
            pop.set(i,0);
            count++;
        }
    }
```

- After playing the entire episode, we have to update the estimate value.

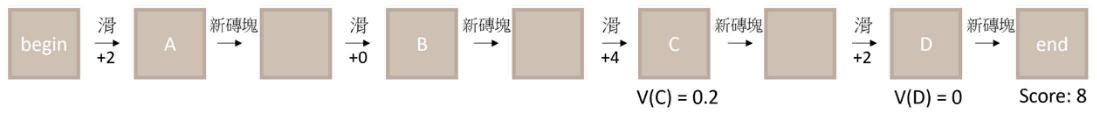
Update_w is the value of alpha * [r + max(Q(s',a')) - Q(s,a)], which is needed to transfer to each feature' s weight.

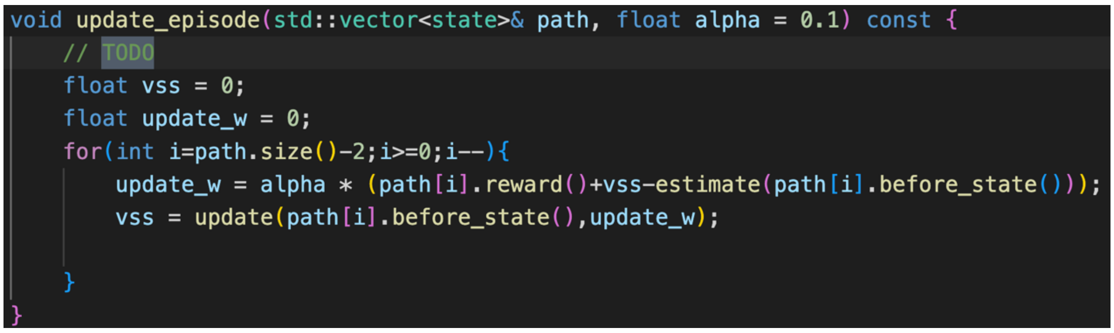
When calling update function, the weight will be update, and the new estimate value will be return. This value will be the next Q(s',a'), say VSS.

The update procedure is illustrated as below, it give us a clear explanation of update the estimate value.

https://ko19951231.github.io/2021/01/01/2048/

接著，對於 C 盤面來說，它在這場遊戲中的「下一個盤面」是 D 盤面，並且是賺了 2 分後成為 D 盤面的，因此它的學習目標是 $2 + V(D)$。注意，這個目標絲毫不管真正到結束後賺了幾分，它只全然相信 $V(D)$ 會是對的，又以目前的 $V(D)$ 值來說，$2 + V(D) = 2$。
因此更新 $V(C) \leftarrow 0 + 0.1 * (2 - 0) = 0.2$





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

    }
}
```