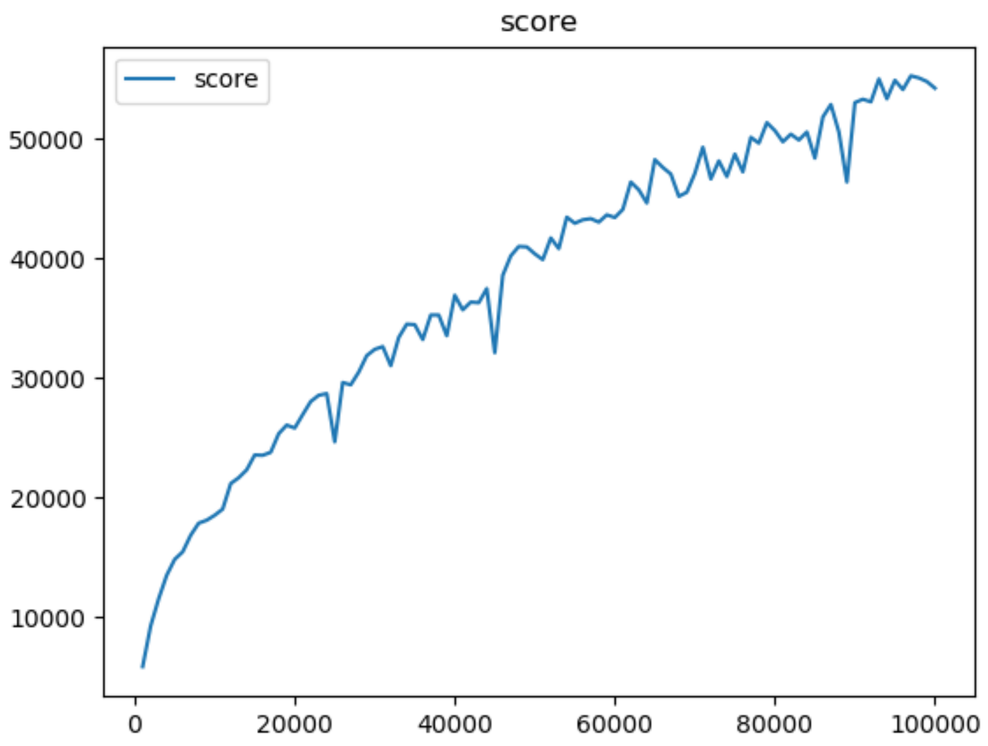
DL HW7

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



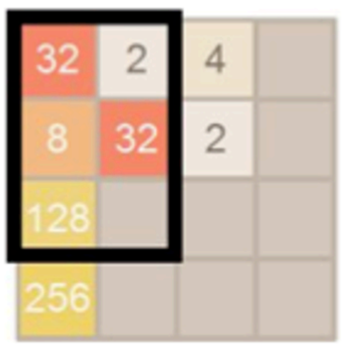
2. Describe your implementation in detail.

I implement the code of TODO part. There are five parts: indexof, estimate, move->assign(b), update, update_episode. The following images

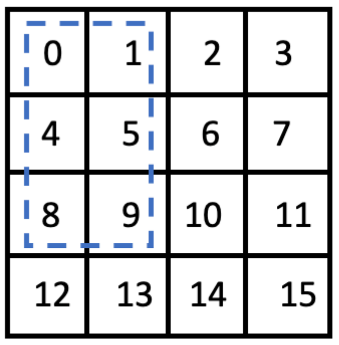
- indexof

This function will return the corresponding index of given pattern. Every number could be represented by 4 bits. The order is little endian.

For example:



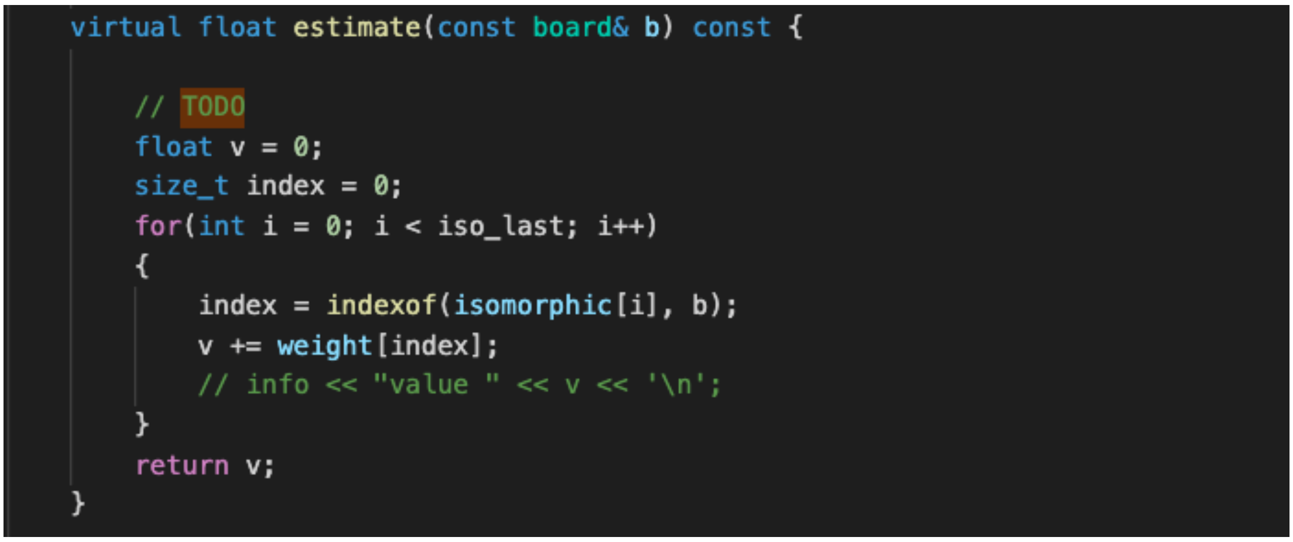
Assume index



Pattern 0 1 4 5 8 9

Return

0 0 0 0	0 1 1 1	0 1 0 1	0 0 1 1	0 0 0 1	0 1 0 1
9	8	5	4	1	0

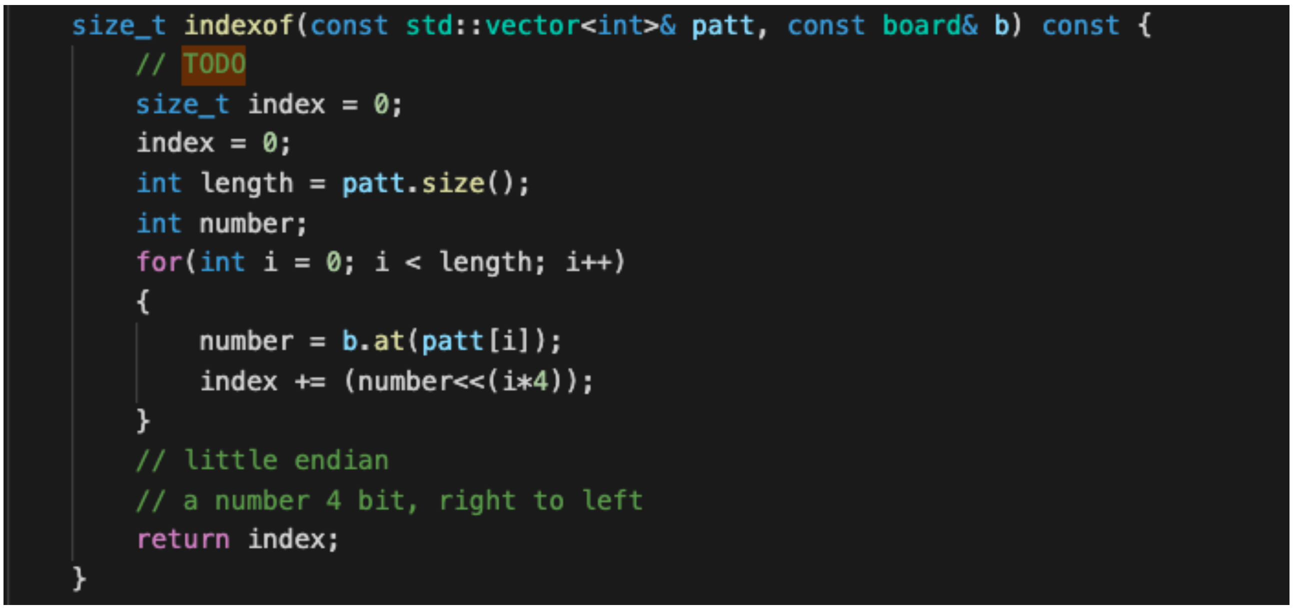


```
virtual float estimate(const board& b) const {

    // TODO
    float v = 0;
    size_t index = 0;
    for(int i = 0; i < iso_last; i++)
    {
        index = indexof(isomorphic[i], b);
        v += weight[index];
        // info << "value " << v << '\n';
    }
    return v;
}
```

- estimate

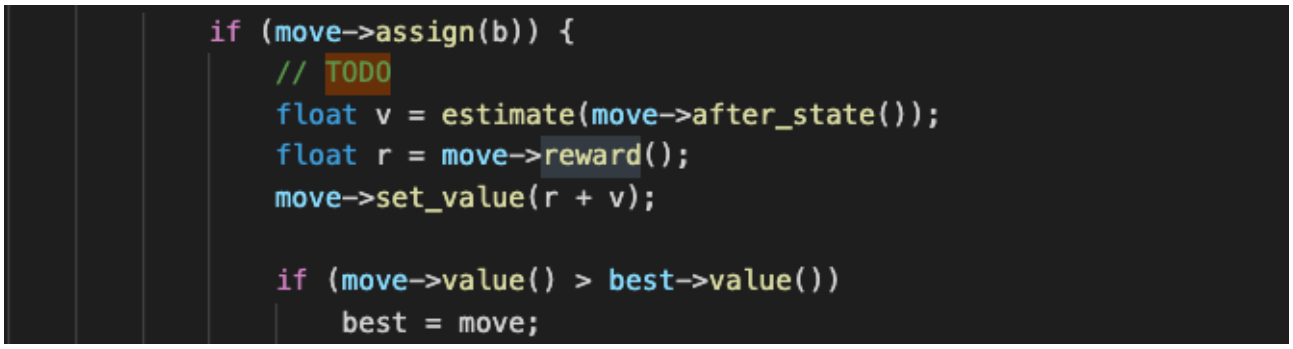
In this part, the function are used to estimate the value of selected n-tuple patterns. Because the pattern has rotations and reflections, it has a lot of isomorphisms. We sum up all values of isomorphisms.



```
size_t indexof(const std::vector<int>& patt, const board& b) const {
    // TODO
    size_t index = 0;
    index = 0;
    int length = patt.size();
    int number;
    for(int i = 0; i < length; i++)
    {
        number = b.at(patt[i]);
        index += (number<<(i*4));
    }
    // little endian
    // a number 4 bit, right to left
    return index;
}
```

- move->assign(b)

When we take an action, we should compute the value of after state. $V_\pi(S_t) = R_{t+1} + V(S_{t+1})$ If the value is best one of the all possible actions, we choose the action.

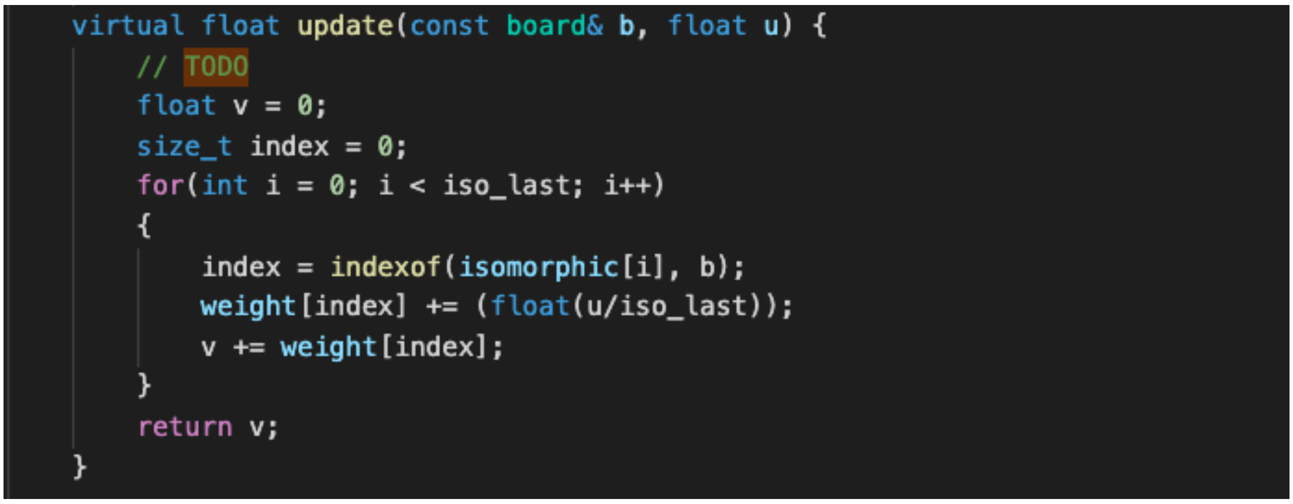


```
        if (move->assign(b)) {
            // TODO
            float v = estimate(move->after_state());
            float r = move->reward();
            move->set_value(r + v);

            if (move->value() > best->value())
                best = move;
```

- update

In this part, we update the value by given error. Because the error is summation of n-tuple pattern including isomorphisms, when we update the value, we should multiply (1/number of isomorphisms). And after updating the value, the function will return new value.



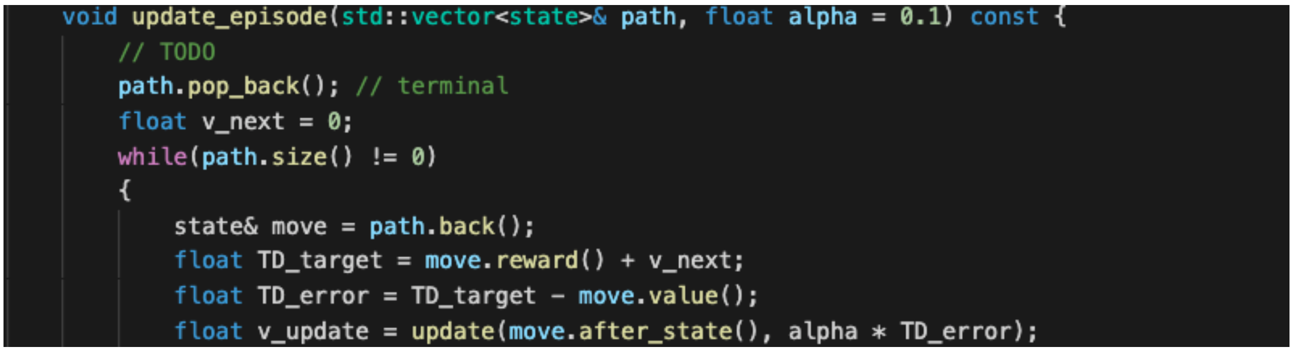
```
virtual float update(const board& b, float u) {
    // TODO
    float v = 0;
    size_t index = 0;
    for(int i = 0; i < iso_last; i++)
    {
        index = indexof(isomorphic[i], b);
        weight[index] += (float(u/iso_last));
        v += weight[index];
    }
    return v;
}
```

- update_episode

When the game is end, we can update weights of state in the episode from the terminal to the beginning. The terminal state doesn't need to update, so we pop the terminal state and starts with last second state. To update a state, we should have the value of the next state.

$$V(S_t) \leftarrow V(S_t) + \alpha\left(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right)$$

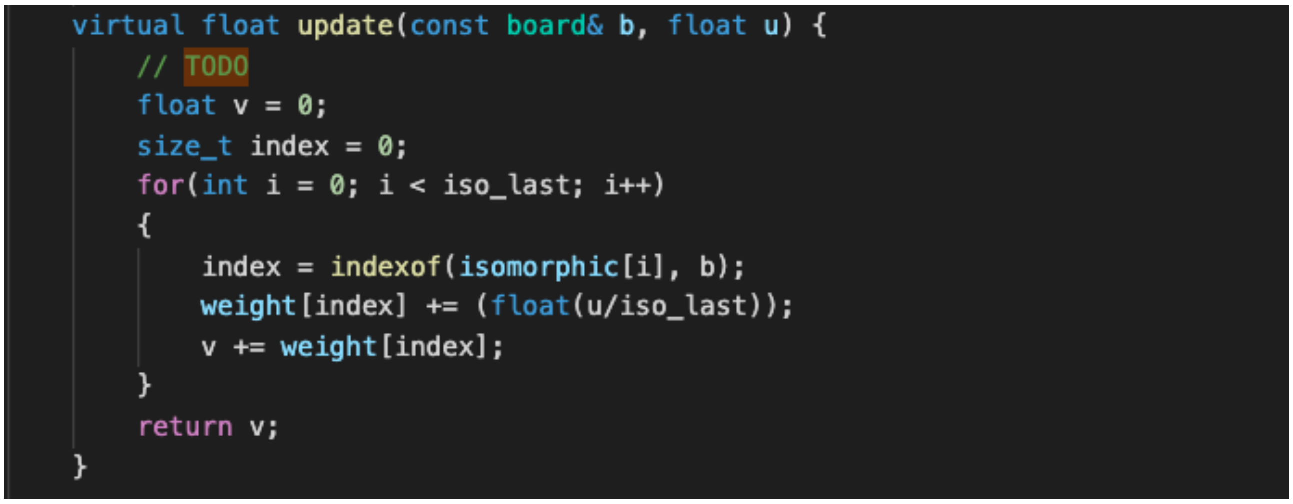
From the above formula, red part $R_{t+1} + \gamma V(S_{t+1})$ is TD target. And TD target - $V(t_s)$ is TD error. We call the *update* function. We input the after state and TD error multiply alpha (Here alpha is equal to gamma in the above formula).



```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    path.pop_back(); // terminal
    float v_next = 0;
    while(path.size() != 0)
    {
        state& move = path.back();
        float TD_target = move.reward() + v_next;
        float TD_error = TD_target - move.value();
        float v_update = update(move.after_state(), alpha * TD_error);
```

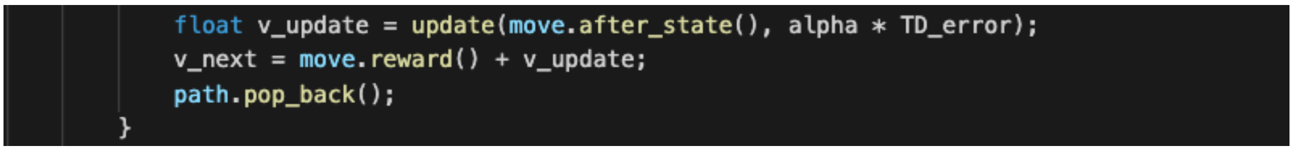
- update

In this part, we update the weights by given error. Because the error is summation of n-tuple pattern including isomorphisms, when we update the weights, we should multiply (1/number of isomorphisms). And after updating the weights, the function will return new value.



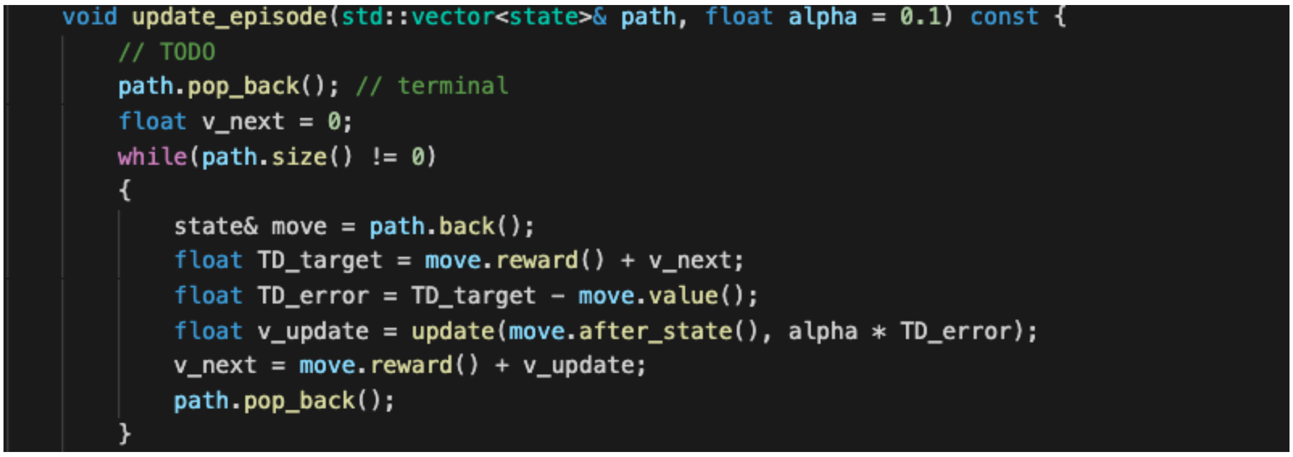
```
virtual float update(const board& b, float u) {
    // TODO
    float v = 0;
    size_t index = 0;
    for(int i = 0; i < iso_last; i++)
    {
        index = indexof(isomorphic[i], b);
        weight[index] += (float(u/iso_last));
        v += weight[index];
    }
    return v;
}
```

With the update value, we can compute the next state we should update with the formula $V_\pi(S_t) = R_{t+1} + V(S_{t+1})$. When the state finish updating, we pop the state. The following code is the rest part of the *update_episode* function.



```
        float v_update = update(move.after_state(), alpha * TD_error);
        v_next = move.reward() + v_update;
        path.pop_back();
    }
```

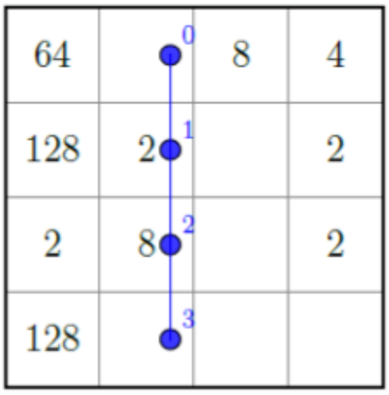
It's a total *update_episode* function code.

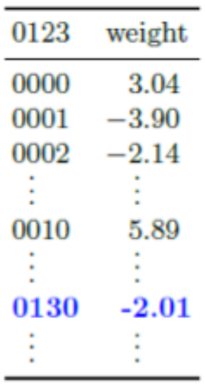


```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    path.pop_back(); // terminal
    float v_next = 0;
    while(path.size() != 0)
    {
        state& move = path.back();
        float TD_target = move.reward() + v_next;
        float TD_error = TD_target - move.value();
        float v_update = update(move.after_state(), alpha * TD_error);
        v_next = move.reward() + v_update;
        path.pop_back();
    }
```

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

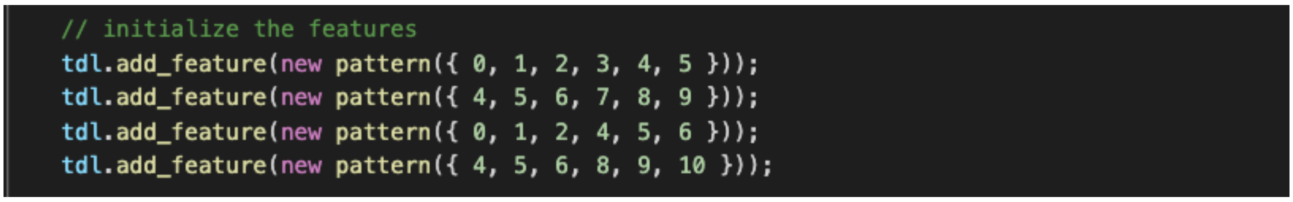
Every cell in 2048 have 16 possible value. There are totally 16 cell in the board. That means there are approximately 16^{16} combination of state could be. If we just store all state and analyze all possibility, it needs much memory and costs a lot of computation time. N-tuple network could extract the feature from the board . Each tuple represents a kind of the combination of state. Take the following image for example, the value of blue 4-tuple networks is 0130. We can build a lookup table like the form on the right hand side. 0130 is an index, and we store the corresponding weight. In this way, we can more easily to store the feature of whole board and understand the state. It's also more efficient.





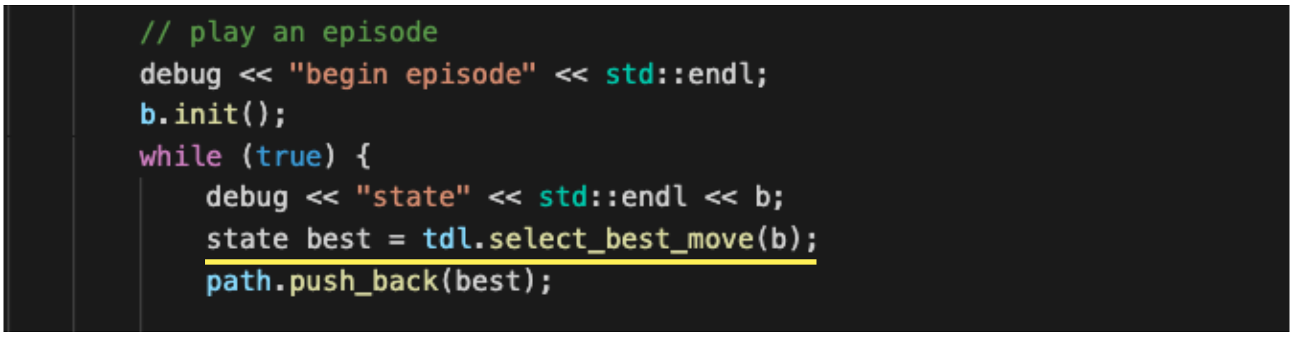
0123	weight
0000	3.04
0001	−3.90
0002	−2.14
⋮	⋮
0010	5.89
⋮	⋮
0130	-2.01
⋮	⋮

Tuples need to define and initialize in the beginning. It will assign the board index and find all possible isomorphisms.



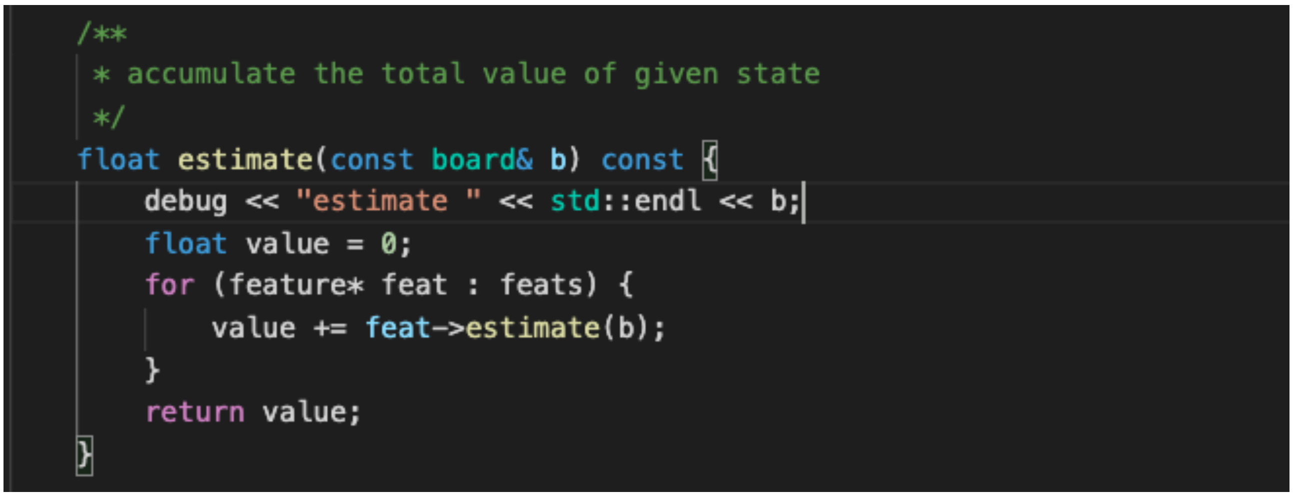
```
// initialize the features
tdl.add_feature(new pattern({ 0, 1, 2, 3, 4, 5 }));
tdl.add_feature(new pattern({ 4, 5, 6, 7, 8, 9 }));
tdl.add_feature(new pattern({ 0, 1, 2, 4, 5, 6 }));
tdl.add_feature(new pattern({ 4, 5, 6, 8, 9, 10 }));
```

In an episode, we will find the best state until the game ends. We use *select_best_move(b)* to choose the next move.



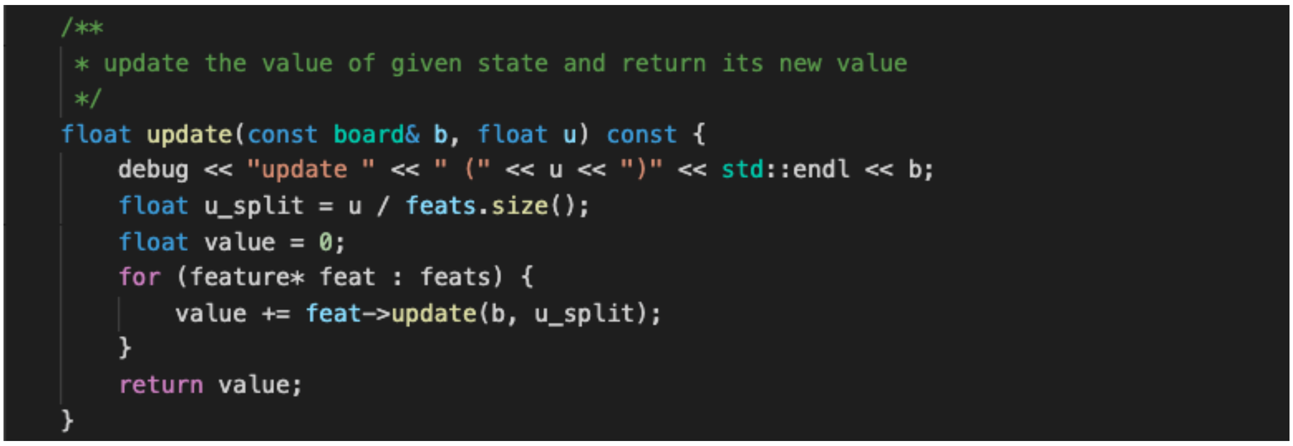
```
// play an episode
debug << "begin episode" << std::endl;
b.init();
while (true) {
    debug << "state" << std::endl << b;
    state best = tdl.select_best_move(b);
    path.push_back(best);
```

In *select_best_move* function, we use the function *move->assign(b)* (introduce in the Q2) to estimate the after state value of different actions and choose the best move. We use *estimate* function to compute the value. This function will go through every tuple we define to compute the value of each pattern and sum up.



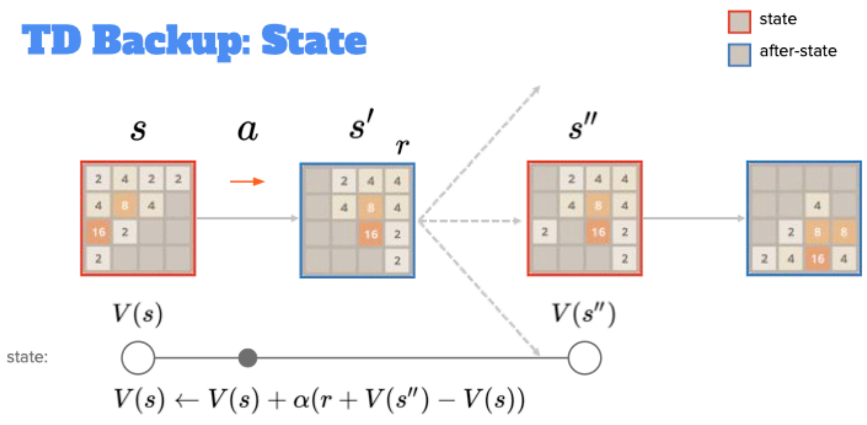
```
/**
 * accumulate the total value of given state
 */
float estimate(const board& b) const {
    debug << "estimate " << std::endl << b;
    float value = 0;
    for (feature* feat : feats) {
        value += feat->estimate(b);
    }
    return value;
}
```

Finally, when the game ends, we use the function *update_episode* to update the weights. We use *update* function to update the value. This function also go through every tuple we define and return the final value.



```
/**
 * update the value of given state and return its new value
 */
float update(const board& b, float u) const {
    debug << "update " << " (" << u << ")" << std::endl << b;
    float u_split = u / feats.size();
    float value = 0;
    for (feature* feat : feats) {
        value += feat->update(b, u_split);
    }
    return value;
}
```

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

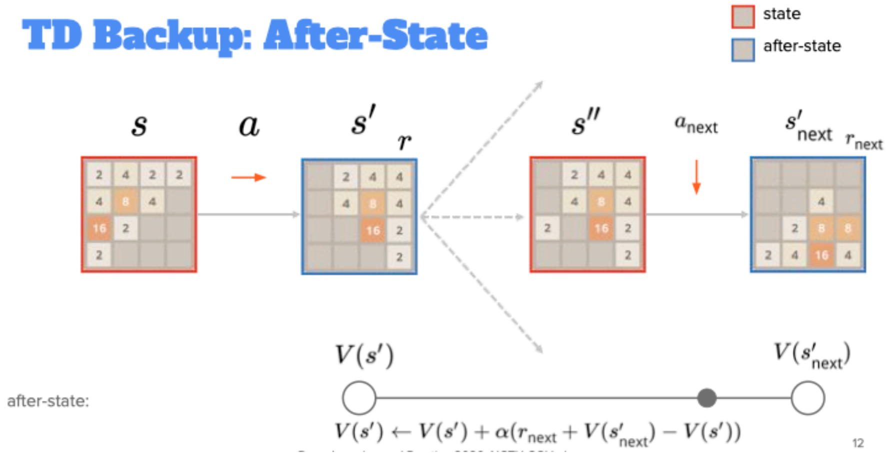


The current state is s. When we take an action, the state become s' and the environment will generate tile in arbitrary position randomly. The state with random tile is s''. TD-backup diagram of V(state) update $V(s)$ by reward of s' and $V(s'')$.

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

Because we estimate $V(s)$ by reward of s' and $V(s'')$. We not only need to consider the reward of s' after taking action, it's more important to estimate $V(s'')$. s'' is the state with random tile, we are not sure about what s'' could be. Therefore, we need to consider all possible states, and get the mean of these state to estimate the total value. By comparing total value, we can choose the best action.

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

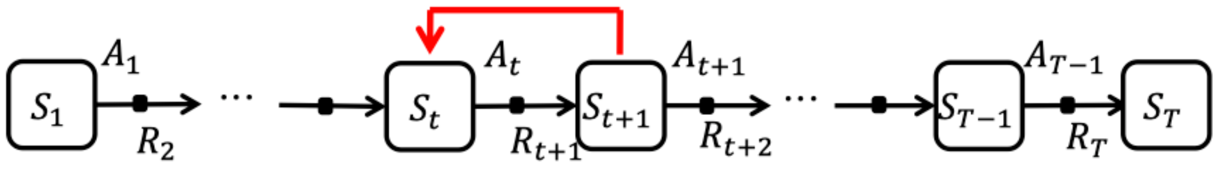


The current state is s', it's a state that has already taken an action. The state will become s'' with a random tile. After choosing an action, the state becomes s'_{next}. TD-backup diagram of V(after-state) update the $V(s')$ by the reward of s'_{next} and $V(s'_{next})$

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

We want to update $V(s')$. This time we can know what s'' is. We only need to consider s'_{next} after taking an action. Compute the reward and estimate the $V(s'_{next})$. By compare total value after taking different action, we can choose the best action.

8. Explain the mechanism of temporal difference learning.



This method learns directly from episodes of experience. It doesn’t need the complete episode and can estimate the value. It updates a guess towards a guess. It try to make each value in temporal difference similar. Unlike Monte-Carlo needs to go through all episode and get the result.

9. Explain whether the TD-update perform bootstrapping.

Yes. The concept of bootstrapping is using one or more estimated values in the update step for the same kind of estimated value. TD learning can update the s $V(s_t)$ by $V(s_{t+1})$.

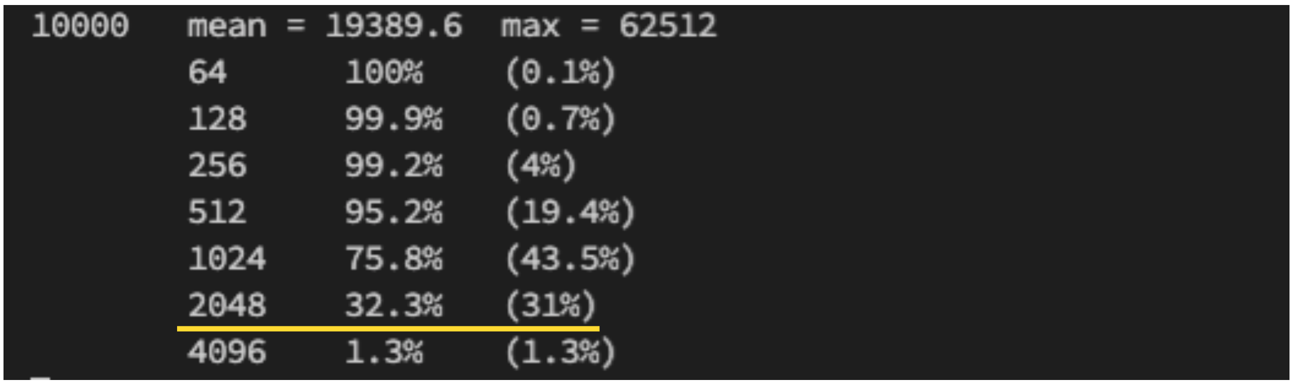
10. Explain whether your training is on-policy or off-policy.

Off-Policy, The policy in this game is the environment will generate a new tile randomly. In my training process, I don’t consider about this. And just evaluate the value of state.

11. Other discussions or improvements.

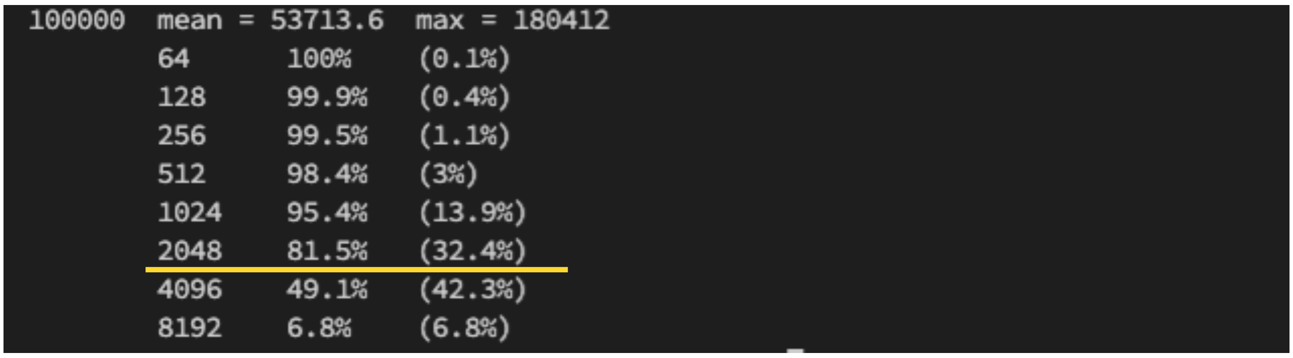
- Performance

2048-tile should appear within 10,000 episodes



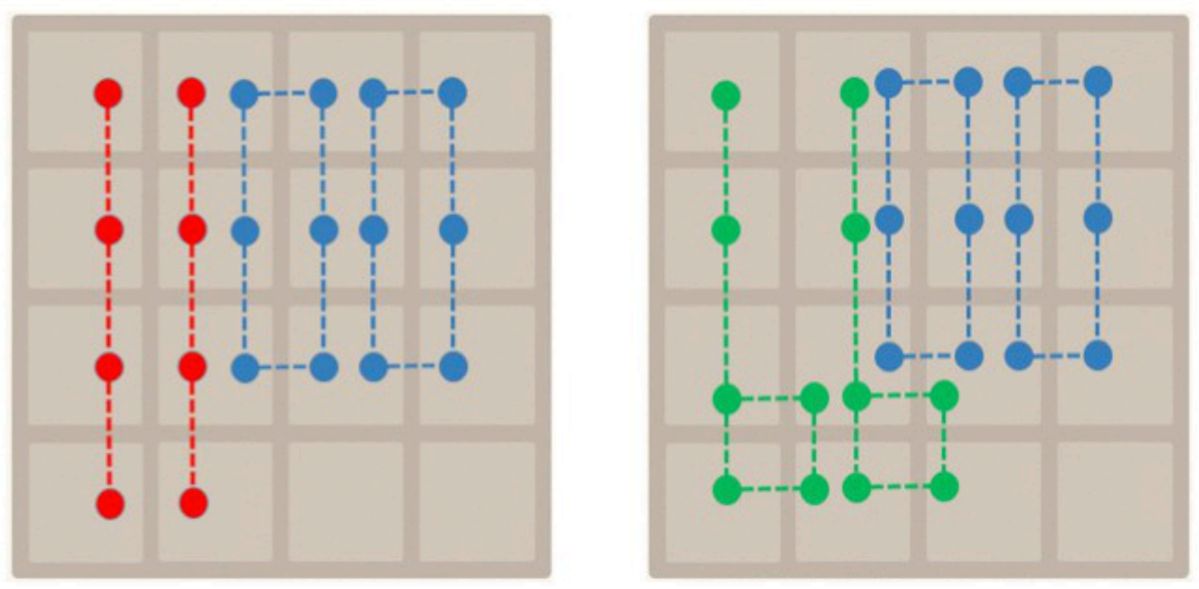
```
10000   mean = 19389.6   max = 62512
        64      100%     (0.1%)
        128     99.9%    (0.7%)
        256     99.2%    (4%)
        512     95.2%    (19.4%)
        1024    75.8%    (43.5%)
        2048    32.3%    (31%)
        4096    1.3%     (1.3%)
```

The 2048-tile win rate in 1000 games : **81.5%**

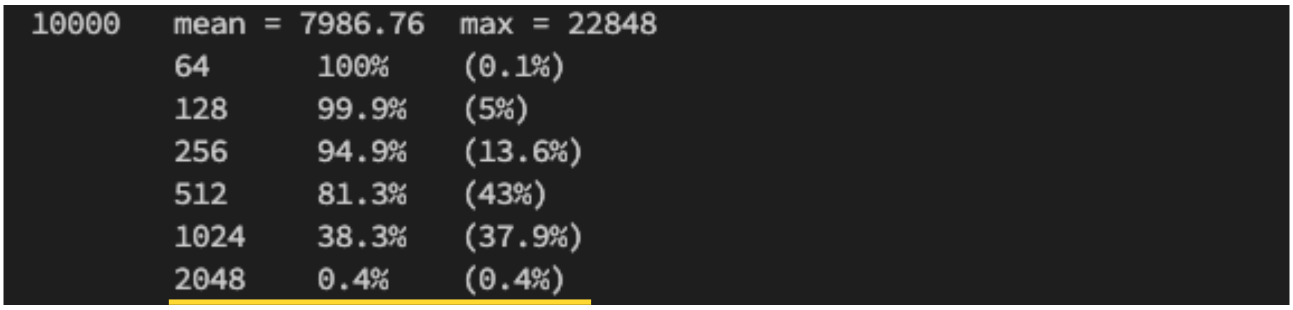


```
100000  mean = 53713.6  max = 180412
        64      100%    (0.1%)
        128     99.9%   (0.4%)
        256     99.5%   (1.1%)
        512     98.4%   (3%)
        1024    95.4%   (13.9%)
        2048    81.5%   (32.4%)
        4096    49.1%   (42.3%)
        8192    6.8%    (6.8%)
```

The following performance is I try the left tuple network. I want to check if the right tuple network are truly better than left one.



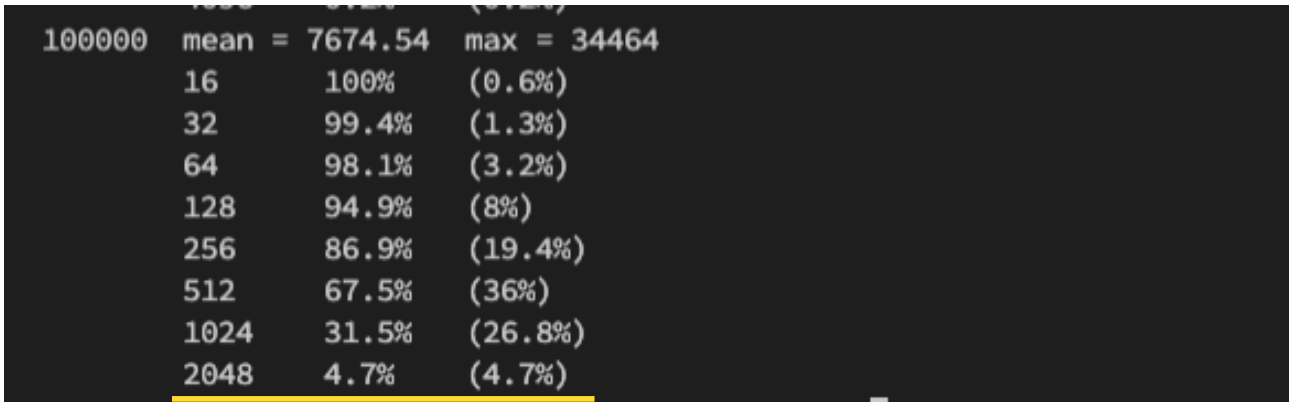
2048-tile should appear within 10,000 episodes.



```
10000   mean = 7986.76  max = 22848
        64      100%    (0.1%)
        128     99.9%   (5%)
        256     94.9%   (13.6%)
        512     81.3%   (43%)
        1024    38.3%   (37.9%)
        2048    0.4%    (0.4%)
```

2048 appear in this tuple network, but win rate is lower than origin one. 4096 appear in the origin one, and here doesn't.

The 2048-tile win rate in 1000 games : 4.7%



```
100000  mean = 7674.54  max = 34464
        16      100%    (0.6%)
        32      99.4%   (1.3%)
        64      98.1%   (3.2%)
        128     94.9%   (8%)
        256     86.9%   (19.4%)
        512     67.5%   (36%)
        1024    31.5%   (26.8%)
        2048    4.7%    (4.7%)
```

We can find the win rate is much lower than origin one. This network also can’t appear more number when episodes reach 10000.