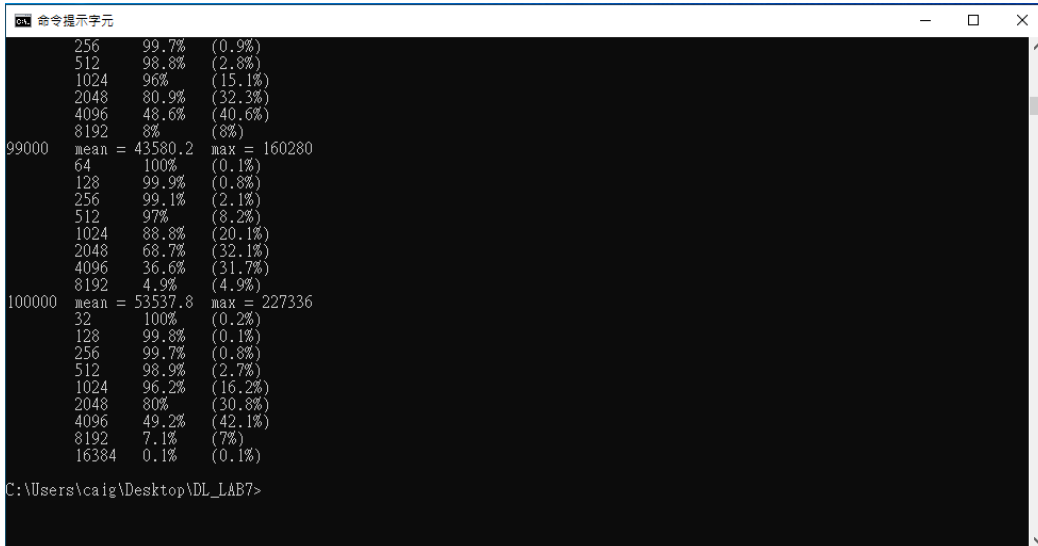


LAB 7 Report

Name: 黃兆宇

ID: 0856601

1. Episode score



```
命令提示字元
256 99.7% (0.9%)
512 98.8% (2.8%)
1024 96% (15.1%)
2048 80.9% (32.3%)
4096 48.6% (40.6%)
8192 8% (8%)
99000 mean = 43580.2 max = 160280
64 100% (0.1%)
128 99.9% (0.8%)
256 99.1% (2.1%)
512 97% (8.2%)
1024 88.8% (20.1%)
2048 68.7% (32.1%)
4096 36.6% (31.7%)
8192 4.9% (4.9%)
100000 mean = 53537.8 max = 227336
32 100% (0.2%)
128 99.8% (0.1%)
256 99.7% (0.8%)
512 98.9% (2.7%)
1024 96.2% (16.2%)
2048 80% (30.8%)
4096 49.2% (42.1%)
8192 7.1% (7%)
16384 0.1% (0.1%)
C:\Users\caig\Desktop\DL_LAB7>
```

The win rate for 2048 after 100000 iterations is 80%.

2. My implementation

Board class

```
class board {
public:
    board(uint64_t raw = 0) : raw(raw) {}
    board(const board& b) = default;
    board& operator =(const board& b) = default;
    operator uint64_t() const { return raw; }
```

The board is implemented by a 64bit value, store only the power part of the number in the variable name raw, for example, store 3 for 2^3 . Some operator and move functions are also implemented in this class.

```
/**
 * get a 4-bit tile
 */
int at(int i) const { return (raw >> (i << 2)) & 0x0f; }
```

The function at(int i) is used to return the value at the index i.

```
/**
 * set a 4-bit tile
 */
void set(int i, int t) { raw = (raw & ~(0x0fULL << (i << 2))) | (uint64_t(t & 0x0f) << (i << 2)); }
```

The function set(int i, int t) is used to set the 4-bit value in the board.

Main function

```
// 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 main function, after initialize parameters and patterns, the program will start main training loop.

```
// train the model
std::vector<state> path;
path.reserve(20000);
for (size_t n = 1; n <= total; n++) {
    board b;
    int score = 0;

    // 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);

        if (best.is_valid()) {
            debug << "best " << best;
            score += best.reward();
            b = best.after_state();
            b.popup();
        } else {
            break;
        }
    }
    debug << "end episode" << std::endl;

    // update by TD(0)
    tdl.update_episode(path, alpha);
    tdl.make_statistic(n, b, score);
    path.clear();
}
```

In main training loop, the program will first select the best move according to the current state. (The 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
            // reward of after state + value of next state
            move->set_value(move->reward() + estimate(move->after_state()));
            if (move->value() > best->value())
                best = move;
        } else {
            move->set_value(-std::numeric_limits<float>::max());
        }
        debug << "test " << *move;
    }
    return *best;
}
```

In the function **select_best_move()**, the program will move the board in 4 directions, (up, down, left, right),

And find the move which will give the highest reward of current state + the value of after state. The state which is obtained after perform the best move (the best after state) will be return by the function.

After the best after state is returned by the function **select_best_move()**, a variable path will used to store the best after state for each step.

And if the move is valid, (which means we can move current board in up, down, left, right directions), the program will update the score and the current board (state), the current board will be updated by the best after state.

At the end of training loop, the function **update_episode()** will update the value function by TD(0).

```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    float value_next_after_state = 0;
    float value_after_state = 0;
    for (path.pop_back() /* terminal state */; path.size(); path.pop_back()) {
        state& move = path.back();

        // TD update: TD error will be r + V(s'next) - V(s')
        value_after_state = move.value();
        float error = move.reward() + value_next_after_state - value_after_state;

        debug << "update error = " << error << " for after state" << std::endl << move.after_state();

        // TD update: V(s') = V(s') + alpha * (r + V(s'next) - V(s'))
        value_after_state = move.reward() + update(move.after_state(), alpha * error);
        value_next_after_state = value_after_state;
    }
}
```

Here we perform after TD update, first compute TD error, TD error will be $r + V(s'_{next}) - V(s')$. And then update the value of after state by the TD error. And because we compute the TD update from back to the head, the value of next after state in the next iteration will equal to the value of after state in this iteration.

3. Implementation and the usage of n -tuple network

```
static float* alloc(size_t num) {
    static size_t total = 0;
    static size_t limit = (1 << 30) / sizeof(float); // 1G memory
    try {
        total += num;
        if (total > limit) throw std::bad_alloc();
        return new float[num]();
    } catch (std::bad_alloc&) {
        error << "memory limit exceeded" << std::endl;
        std::exit(-1);
    }
    return nullptr;
}
size_t length;
float* weight;
```

The n -tuple network is mainly implemented by two classes, feature and pattern, the pattern weights is stored in a variable name weight, the function **alloc()** will allocate the $\text{float} * \text{size}$ of n -tuple tree memory for each pattern.

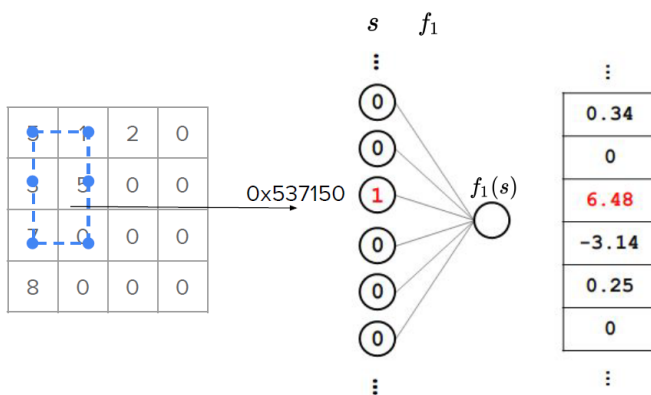
```
pattern(const std::vector<int>& p, int iso = 8) : feature(1 << (p.size() * 4)),
```

Size of n -tuple tree is different according to the size of pattern, for example, if size of pattern = 5, then the size of n -tuple tree will be 2^{20} , because we assume the value in our 2048 game will not exceed 2^{16} , and we store only the power part of value (ex. 32, we store 5) in our board, so we may have the power value from 1 ~ 15 (no 2^0 in 2048 game). We need 4 bits to store the value, and because the pattern size = 5, so we need

$5 \times 4 = 20$ bits to store the data, that's why the size is 2^{20} .

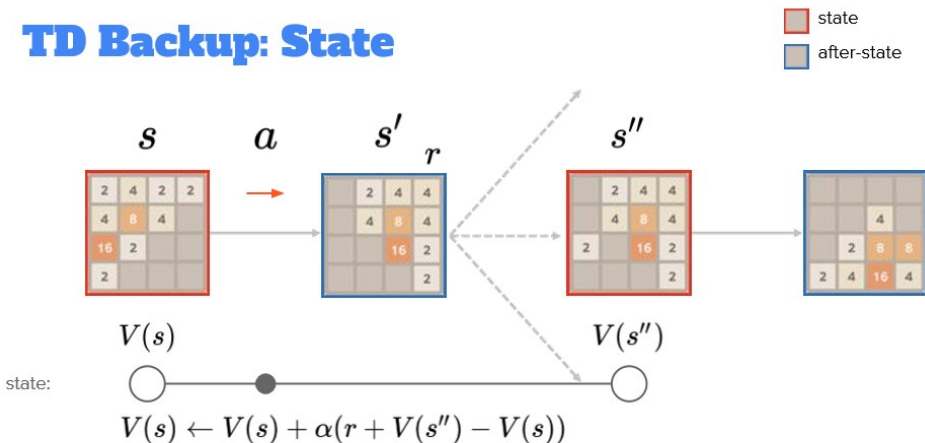
```
for (int i = 0; i < 8; i++) {
    board idx = 0xfedcba9876543210ull;
    if (i >= 4) idx.mirror();
    idx.rotate(i);
    for (int t : p) {
        isomorphic[i].push_back(idx.at(t)); // Storing the index of the iso patterns
    }
}
```

And according to the reference paper, the same pattern will be rotated several time to get more value information from board, which in the code is called isomorphic. The way to implement the isomorphic in the code is a little tricky, instead of “rotate the pattern”, the code “rotate the index board (idx)”, index board in the code is a board which is filled by the index. From left top to the right bottom, the board value will be 0, 1, 2, ... 15. And use the function mirror and rotate to rearrange the board, by doing this, we don't need to mirror and rotate the pattern to get the its isomorphic.



For the usage of n-tuple network, it is used to represent the value of the state. If we want to store the value of each state directly by a look-up table, we will have $(4 \times 4)^{16}$ states, which is impossible. But if using n-tuple network the memory usage will be reduced, in this lab, a 4-tuple only need 16^4 weights. Which is possible to implement.

4. TD-backup diagram of $V(\text{state})$



For the diagram of state TD backup, first the program will compute the after state (s') and the reward of the current state according to the action, and one 2-tile or 4 tile will emerge from the 7 empty blocks. So it may give out $7 * 2$ kinds of next state(s''). And for the Value function update, **the value function of current state $V(s)$** will be updated by adding $\alpha * (\text{reward of current state } (r) + \text{value of next state } (V(s'')) - \text{value of current state } (V(s')))$.

5. Action selection of $V(\text{state})$ in a diagram

```

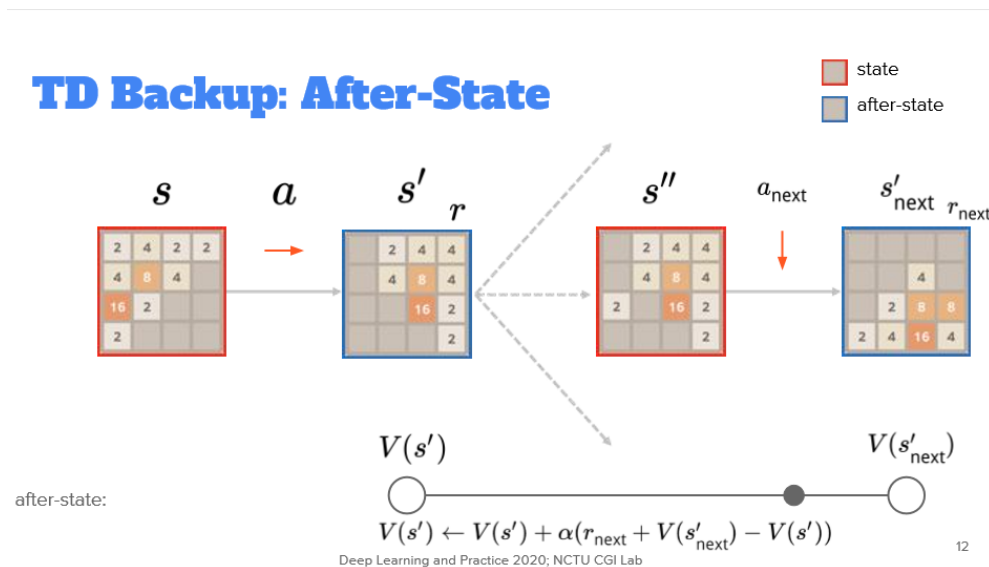
1: function EVALUATE( $s, a$ )
2:    $s', r \leftarrow \text{COMPUTE AFTERSTATE}(s, a)$ 
3:    $S'' \leftarrow \text{ALL POSSIBLE NEXT STATES}(s')$ 
4:   return  $r + \sum_{s'' \in S''} P(s, a, s'') V(s'')$ 
5:
6: function LEARN EVALUATION( $s, a, r, s', s''$ )
7:    $V(s) \leftarrow V(s) + \alpha(r + V(s'') - V(s))$ 

```

Figure 5: The state evaluation function and TD(0).

For the action selection, the program will first evaluate 4 different actions (up, down, left, right), and find the action that can bring the highest accumulate reward by the function $r + \sum_{s'' \in S''} P(s, a, s'') V(s'')$. To compute this, program need to compute all the possible $V(s'')$, (in the case of the diagram above $7 * 2$ kinds).

6. TD-backup diagram of $V(\text{after-state})$



For the diagram of after-state TD backup, the program will first compute the after-state given different actions, and then compute the next state, then using the same way to compute the after state of next state and the reward of next state, but the update method is different from the state TD-backup, the program will **update the value of next state $V(s')$** by adding $\alpha * (\text{reward of next state } (r_{\text{next}}) + \text{value of the after state of next state } V(s'_{\text{next}}) - \text{value of next state } V(s'))$

7. Action selection of V (after-state) in a diagram

```
1: function EVALUATE( $s, a$ )
2:    $s', r \leftarrow \text{COMPUTE\_AFTERSTATE}(s, a)$ 
3:   return  $r + V(s')$ 
4:
5: function LEARN EVALUATION( $s, a, r, s', s''$ )
6:    $a_{next} \leftarrow \arg \max_{a' \in A(s')} \text{EVALUATE}(s', a')$ 
7:    $s'_{next}, r_{next} \leftarrow \text{COMPUTE\_AFTERSTATE}(s'', a_{next})$ 
8:    $V(s') \leftarrow V(s') + \alpha(r_{next} + V(s'_{next}) - V(s'))$ 
```

For the action-selection of V(after-state), the program will first evaluate each actions by accumulate reward, the function reward of current action (r) + value of after state $V(s')$, the action that can bring the highest accumulate reward will be chosen. Unlike the state TD backup, we don't need to consider different states, because for each action, we will only get one next-state.

8. Mechanism of temporal difference learning

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

A model-free learning method that predict the value function by learning from the episodes. TD learning update a guess toward guess, the value function of current step is update by the TD error, and TD error is computed by reward of current state (R_{t+1}) + value of next step ($V(S_{t+1})$) – value of current state ($V(S_t)$). And when arriving next state, the value function of current state can be updated, so the TD learning can learn from the incomplete episodes.

9. TD-update perform bootstrapping

The bootstrap in RL means that estimate a value based on another estimate value. In TD update, TD error $R_{t+1} + V(S_{t+1}) - V(S_t)$ is using the estimate of next state to update the current value, so TD learning perform bootstrapping.

10. TD-update is on-policy

In main function, we first store the actions we play the game and use the same policy (same (state, action) pairs) to update the value, so it is an on-policy learning.

11. Other discussion and improvements

The value function implement by the n-tuple network play well in 2048 game, but I think some method can be used to improve the learning efficiency, first is using the $TD(\lambda)$ to replace the $TD(0)$ which may give more precise prediction of value, and another method is trying to use different patterns for n-tuple tree, which can make the program achieve higher score in fewer iterations.