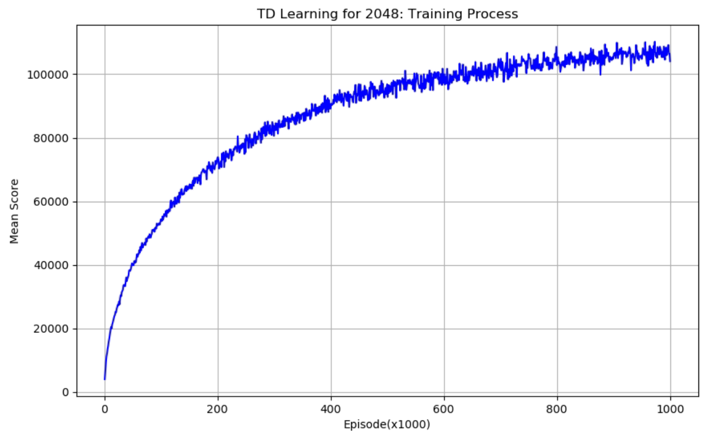
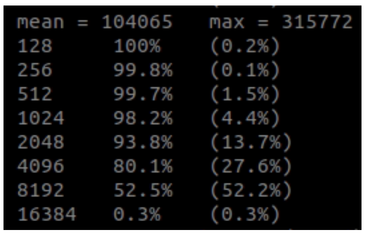
Lab1: Temporal Difference Learning in 2048

Student Name: 歐庭維

Student ID: 312605001

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





```
mean = 104065     max = 315772
128       100%     (0.2%)
256       99.8%    (0.1%)
512       99.7%    (1.5%)
1024      98.2%    (4.4%)
2048      93.8%    (13.7%)
4096      80.1%    (27.6%)
8192      52.5%    (52.2%)
16384     0.3%     (0.3%)
```

Figure 1: Mean Score vs. Episode

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

In this project, n-tuple network is used to increase the effectiveness of the computation, each pattern has wight different isomorphic pattern with rotation and reflection. These isomorphic patterns recorded the results in different position on the board. Allow us to effectively estimate the board's value by summarizing these patterns. For instance, if the board have a pattern {1, 5, 9, 13}, as shown in Figure1. The number inside the pattern is {0, 2, 8, 0}, convert to an exponent is {0, 1, 3, 0}. We can find the pattern's weight on the weight table in n-tuple network on index of 0130. This weight is represented as estimate value in the inference stage, or an update value in the training stage. In the inference stage, we must summarize all the isomorphic pattern's weight as the pattern's score. On the other hand, in the training stage. we must evenly update the score to all the isomorphic patterns.

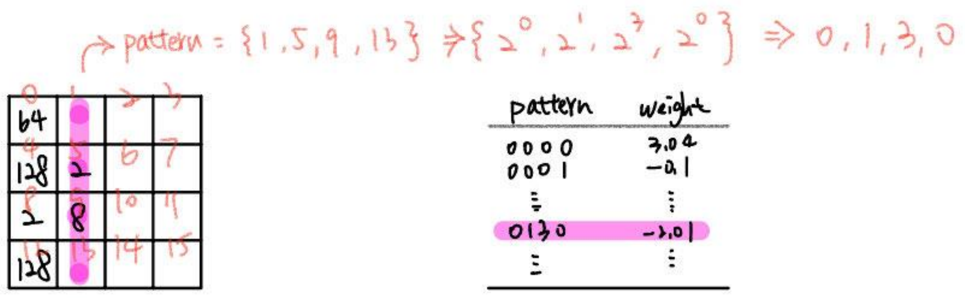


Figure 2: Example of the n-tuple network.

Explain the mechanism of TD

TD (Temporal Difference) is a prediction method used in reinforcement learning to update predictions based on experience. It combines elements of Monte Carlo methods and dynamic programming, updating value functions through online learning without relying on a model.

1. Prediction Update

   TD update the value of the current state based on the difference between the estimated values of the current state and the next state. The difference is derived from the reward and the estimate of the future state value.

2. TD Error

   The TD error is the difference between the reward plus the estimated value of the next state and the estimated value of the current state:

$$TD\ Error = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

3. Value Update

   The value of the current state is updated using the TD error. This update is typically done using a learning rate to control the extent of the adjustment:

$$V(S_t) \leftarrow V(S_t) + \alpha * TD\ Error$$

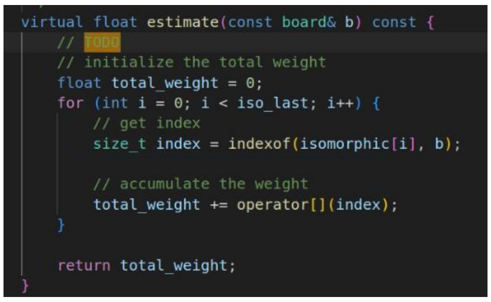
   Where α is learning rate.

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

In this project, we must finish five to-do part:

1. estimate the value of a given board.

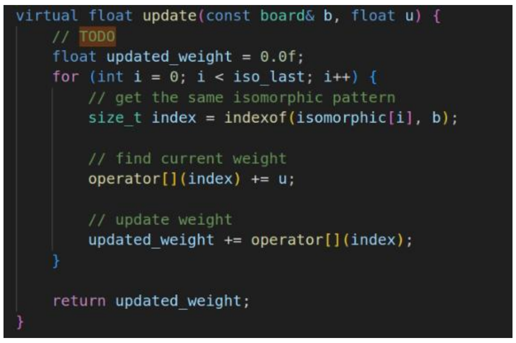
   In this part, the function "estimate" is used to calculate the estimated value by considering the board state "b". In this function, "total_weight" is used to accumulate the weight. The function then iterates through all isomorphic patterns, using "indexof" function to obtain the index of each pattern in the current board state. It retrieves the corresponding weight form the weight array and adds it to "total_weight". Finally, the function returns the calculated "total_weight" as the estimated of the board state "b".



```
virtual float estimate(const board& b) const {
    // TODO
    // initialize the total weight
    float total_weight = 0;
    for (int i = 0; i < iso_last; i++) {
        // get index
        size_t index = indexof(isomorphic[i], b);

        // accumulate the weight
        total_weight += operator[](index);
    }

    return total_weight;
}
```

2. update the value of a given board and return its updated value.

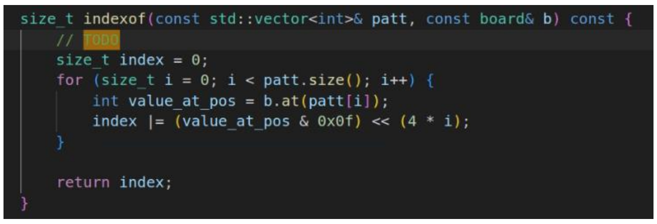
   In this part, the function "update" is used to update the weights of all isomorphic patterns in the board state "b" by adding the value "u" to each weight and returns the sum of the updated weights. I do this by iterating through each isomorphic pattern, retrieving its index for the current board state, updating the corresponding weight, and accumulating the updated weight value.



```
virtual float update(const board& b, float u) {
    // TODO
    float updated_weight = 0.0f;
    for (int i = 0; i < iso_last; i++) {
        // get the same isomorphic pattern
        size_t index = indexof(isomorphic[i], b);

        // find current weight
        operator[](index) += u;

        // update weight
        updated_weight += operator[](index);
    }

    return updated_weight;
}
```

3. find the index of n-tuple network.

   The "indexof" function calculates and returns an index based in the given pattern "patt" and the board state "b". It starts by initializing an "index" variable to 0, which will store the computed index. It then iterates through each position in the pattern "patt", retrieves the corresponding value from the board "b" using the pre-defined function "at()". Each value is marked with 0x0f and then shifted left by 4*i bits to ensure it is placed in the correct position within the final index. The resulting value is accumulated into "index" using bitwise OR. After processing all positions, the function returns the computed index.



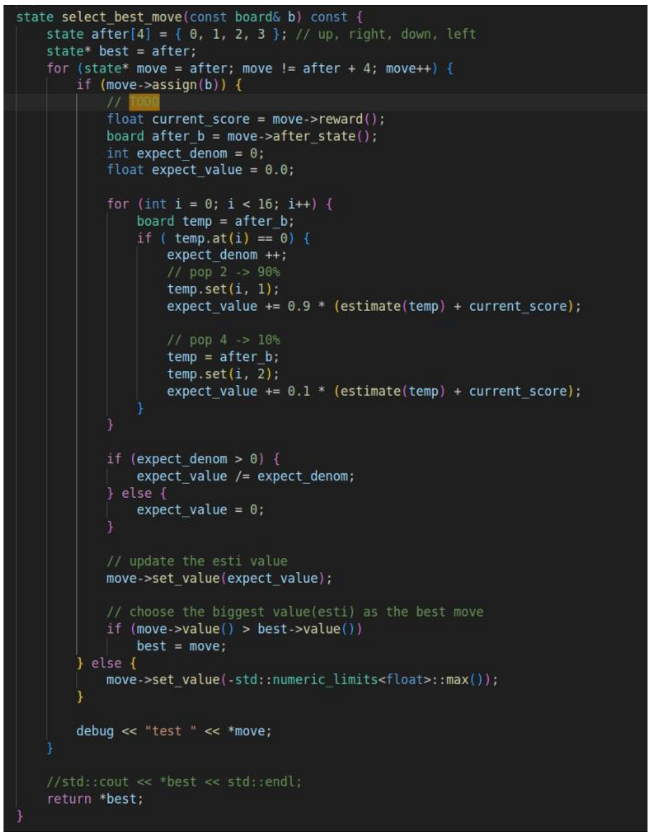
```
size_t indexof(const std::vector<int>& patt, const board& b) const {
    // TODO
    size_t index = 0;
    for (size_t i = 0; i < patt.size(); i++) {
        int value_at_pos = b.at(patt[i]);
        index |= (value_at_pos & 0x0f) << (4 * i);
    }

    return index;
}
```

4. select a best move of a before state board.

   In this part, we are limited to using popout probability in the code. My approach is to first iterate over all empty position on the current board and calculate empty positions on the current board and calculate the expected value of placing a 2 (with a 90% probability) or a 4 (with a 10% probability) at each empty position. I then accumulated the expected values for all empty blocks. Finally, I divide the total accumulated expected value by the number of empty blocks to get the average value for the move, which is return as the move's evaluation.

Next, based in the highest evaluation value, the direction to move is chosen:

$$\pi(S) = argmax_a(R_{t+1} + \mathrm{E}[V(S_{t=1} | S_t = s, A_t = a])$$



```
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
            float current_score = move->reward();
            board after_b = move->after_state();
            int expect_denom = 0;
            float expect_value = 0.0;

            for (int i = 0; i < 16; i++) {
                board temp = after_b;
                if ( temp.at(i) == 0) {
                    expect_denom ++;
                    // pop 2 -> 90%
                    temp.set(i, 1);
                    expect_value += 0.9 * (estimate(temp) + current_score);

                    // pop 4 -> 10%
                    temp = after_b;
                    temp.set(i, 2);
                    expect_value += 0.1 * (estimate(temp) + current_score);
                }
            }

            if (expect_denom > 0) {
                expect_value /= expect_denom;
            } else {
                expect_value = 0;
            }

            // update the esti value
            move->set_value(expect_value);

            // choose the biggest value(esti) as the best move
            if (move->value() > best->value())
                best = move;
        } else {
            move->set_value(-std::numeric_limits<float>::max());
        }

        debug << "test " << *move;
    }

    //std::cout << *best << std::endl;
    return *best;
}
```

5. update the tuple network by an episode.

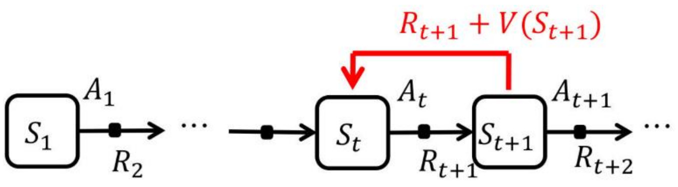
The process of this code starts by iterating through the path in reverse, beginning from the last to two states (in front of the end of a game). For each state, it first retrieves the board state using "before_state()", which means the S_t state, and then gets the reward for the subsequent state using "reward()". The TD error is calculated as the difference between the target value for the next state and the estimated value for the current state:

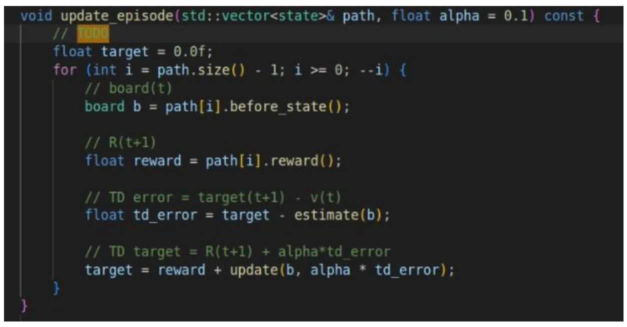
$$TD\ Error = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$$

Using the learning rate "α", this TD error is adjusted and passed to the "update()" function to update the estimated value of the current state:

$$V(S_t) \leftarrow V(S_t) + \alpha * TD\ Error$$

Finally, the updated target value is set as the target for the next state, and the process continues until all states in the path have been updated.





```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    float target = 0.0f;
    for (int i = path.size() - 1; i >= 0; --i) {
        // board(t)
        board b = path[i].before_state();

        // R(t+1)
        float reward = path[i].reward();

        // TD error = target(t+1) - v(t)
        float td_error = target - estimate(b);

        // TD target = R(t+1) + alpha*td_error
        target = reward + update(b, alpha * td_error);
    }
}
```