國立陽明交通大學

NATIONAL YANG MING CHIAO TUNG UNIVERSITY

Deep Learning
Lab2: Temporal Difference Learning

Name: 許子駿

Student ID: 311551166

Institute of Computer Science and Engineering

2023 年 4 月 9 日

Table of Contents

Chapter 1

Introduction

This task is to implement Temporal Difference (TD) learning algorithm to solve 2048 game using an n-tuple network.

Chapter 2

Details

Details are listed in this chapter including 5 parts, n-tuple network, TD(0) algorithm, TD-backup diagram, action selection, and implementation details.

2.1 n-tuple network

1. Isomorphic patterns are created in constructor of class `pattern` (See Listing 2.1).

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

Listing 2.1: C++ code of class pattern constructor.

2. Get index of an isomorphic pattern of the board.

```
size_t indexof(const std::vector<int>& patt, const board& b) const {
    // TODO
    size_t idx = 0;
    for (size_t i = 0; i < patt.size(); i++)
        idx |= b.at(patt[i]) << (4 * i); // Shift left 16 bits.

    return idx;
}
```

Listing 2.2: C++ code of indexof function of class pattern.

3. Values of isomorphic patterns are sum up to represent the value of the board (See Listing 2.3).

```
virtual float estimate(const board& b) const {
    // TODO
    // Sum up values of all isomorphic patterns.
    float sum_iso_patterns = 0;
    for (int i = 0; i < iso_last; i++)
        sum_iso_patterns += weight[indexof(isomorphic[i], b)];

    return sum_iso_patterns;
}
```

Listing 2.3: C++ code of estimate function of class pattern.

4. Update isomorphic patterns with mean error, and then return the updated weight (See Listing 2.4).

```
virtual float update(const board& b, float u) {
    // TODO
    // Update all isomorphic patterns with mean.
    float mean = u / iso_last;
    float sum_iso_patterns = 0;
    for (int i = 0; i < iso_last; i++) {
        size_t idx = indexof(isomorphic[i], b);
        weight[idx] += mean;
        sum_iso_patterns += weight[idx];
    }

    return sum_iso_patterns;
}
```

Listing 2.4: C++ code of update function of class pattern.

2.2 TD(0) algorithm

This algorithm, starting from second last move of the episode, calculates error between before-state of the move s and next move s'', then updates before-state of s with the error, and finally calculates TD target with sum of reward of s and updated weight of before-state of s. Keep the TD target for next step, repeat steps above till the start of the episode (See Figure 2.1 and Listing 2.5).

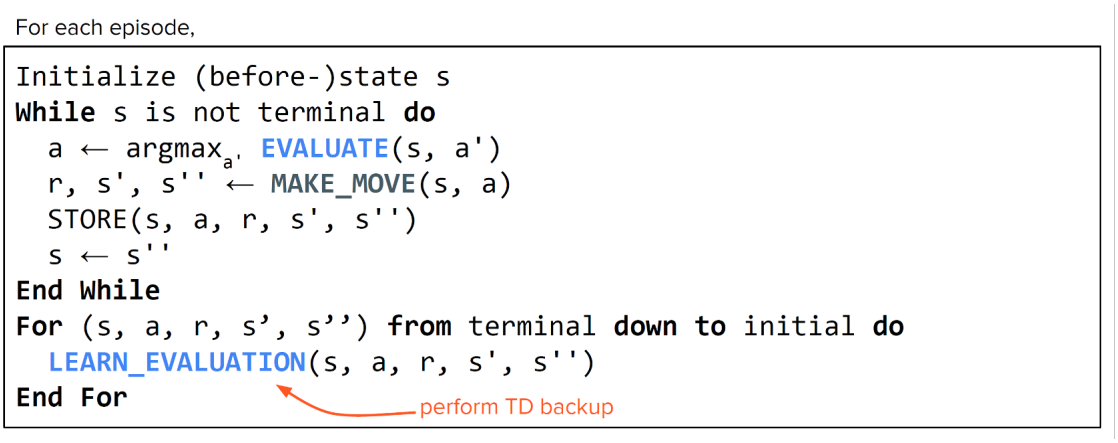

Figure 2.1: Pseudo-code of TD(0) algorithm.

```cpp
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    // Starts from second last move, and update state backwards iteratively.
    float target = 0.0f;
    for (path.pop_back(); path.size(); path.pop_back()) {
        state& move = path.back();
        target = move.reward() +
            update(move.before_state(), alpha * (target - (move.value() - move.reward())));
    }
}
```

Listing 2.5: C++ code of update_episode function of class learning.

2.3 TD-backup diagram

1. Before-state: use before-state of move s as value function, and use reward r and before-state of next move s'' as TD target to update before-state of the move s (See Figure 2.2).

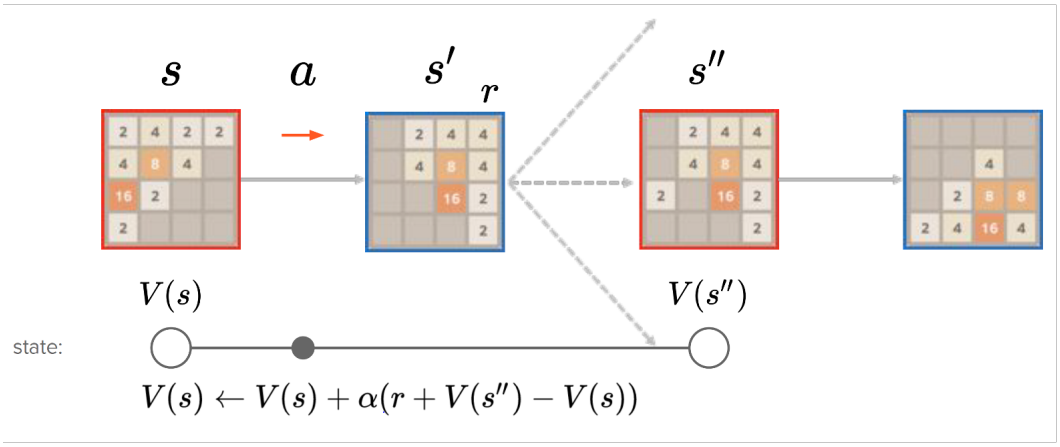

Figure 2.2: TD backup diagram of before-state.

2. After-state: use after-state of move s' as value function, and use reward r_{next} and after-state of next move s'_{next} as TD target to update after-state of the move s' (See Figure 2.3).

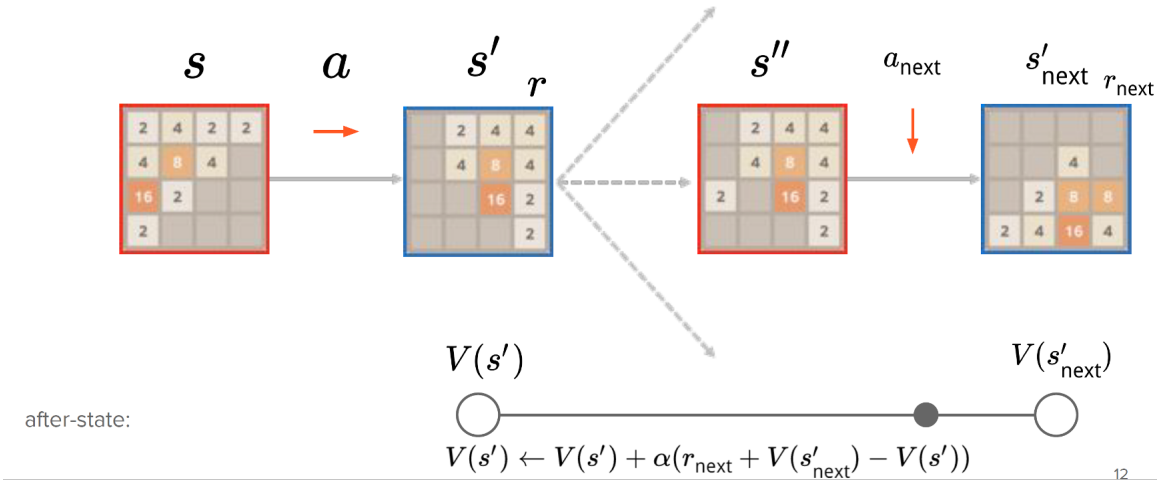

Figure 2.3: TD backup diagram of after-state.

2.4 Action selection

1. Before-state: after choosing an action A_t, randomly generate one 2-tile or 4-tile, then calculate and weighted sum up value functions of each new state S_{t+1} by the probability that 2-tile and 4-tile pop-up and number of empty tiles (See Figure 2.4).

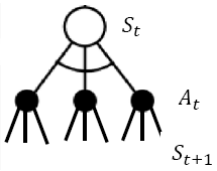

Figure 2.4: Action selection of before-state.

2. After-state: consider that before-state of next move S_{t+1} is fixed, and when choosing action of next move A_{t+1} is the same situation, so this approach chooses the action that results in highest value of after-state. (See Figure 2.5).

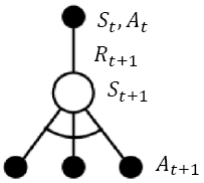

Figure 2.5: Action selection of after-state.

2.5 Implementation details

1. `indexof` (See Listing 2.2), `estimate` (See Listing 2.3), `update` (See Listing 2.4), `update_episode` (See Listing 2.5) functions have been listed in previous sections.

2. First, find all empty tiles during after-state, then randomly generate 2-tile or 4-tile, and calculate weighted sum to be the value of before-state of the move, and finally update the highest value for further comparisons, and return best move (See Listing 2.6).

```
state select_best_move(const board& b) const {
    state after[4] = { 0, 1, 2, 3 }; // up, right, down, left
    float weight_2_tile = 0.9f;
    float weight_4_tile = 1.0f - weight_2_tile;

```

```
    state* best = after;
    float best_move_val = -std::numeric_limits<float>::max();
    for (state* move = after; move != after + 4; move++) {
        if (move->assign(b)) {
            // TODO
            board after_state = move->after_state();
            int tiles[16];
            int numEmptyTiles = 0;

            // Find all empty tiles during after-state.
            for (int i = 0; i < 16; i++)
                if (after_state.at(i) == 0)
                    tiles[numEmptyTiles++] = i;

            // Find best move during after-state.
            float val = move->reward();
            for (int i = 0; i < numEmptyTiles; i++) {
                board* cur_board = new board(uint64_t(after_state));

                cur_board->set(tiles[i], 1); // 2-tile.
                val += weight_2_tile * (estimate(*cur_board) /
    numEmptyTiles);

                cur_board->set(tiles[i], 2); // 4-tile.
                val += weight_4_tile * (estimate(*cur_board) /
    numEmptyTiles);
            }

            move->set_value(estimate(move->before_state()));

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

Listing 2.6: C++ code of select_best_move function of class learning.

3. With original 4 patterns (See Listing 2.7 and Figure 2.6), the average reached 2048-tile percentage is about **90%** after 350K iterations. However, with additional 4 patterns, totally 8 patterns (See Listing 2.8 and Figure 2.7), it's about **95%** (Details are in Chapter 3).

```
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 }));
```

Listing 2.7: C++ code of original 4 patterns.

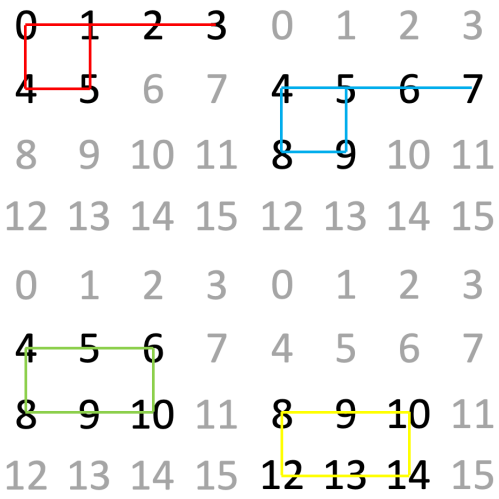

Figure 2.6: Illustration of original 4 patterns.

```
tdl.add_feature(new pattern({ 0, 2, 5, 10 }));
tdl.add_feature(new pattern({ 4, 6, 9, 14 }));
tdl.add_feature(new pattern({ 1, 4, 5, 6, 9, 13 }));
tdl.add_feature(new pattern({ 1, 4, 9, 14 }));
```

Listing 2.8: C++ code of new 4 patterns.

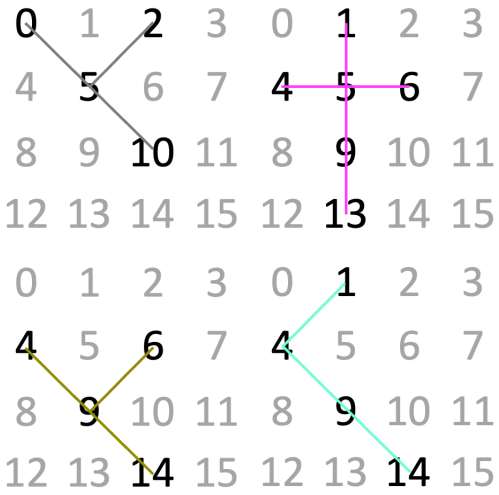

Figure 2.7: Illustration of new 4 patterns.

Chapter 3

Results

1. Mean score reaches about **77510** after 350K iterations training (See Figure 3.1).

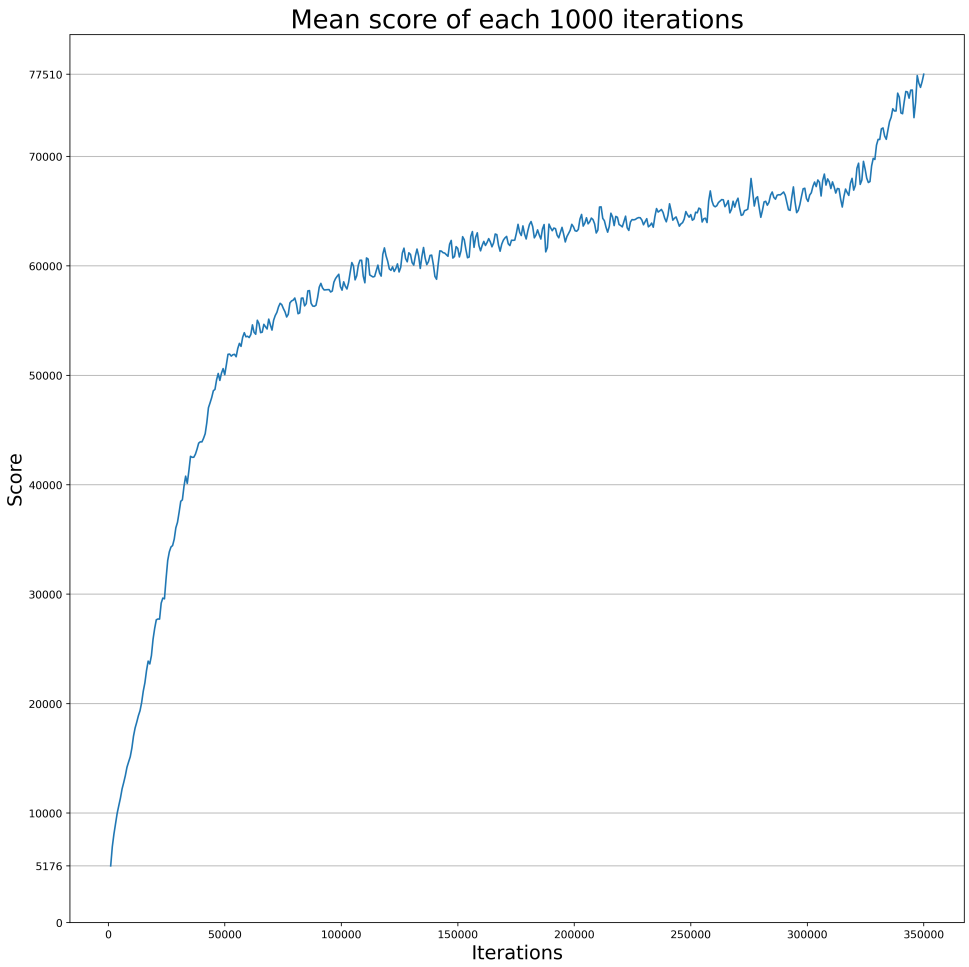

Figure 3.1: Mean score of each 1K iterations for total 350K iterations.

2. Average reached 2048-tile percentage is about **95%** after 350K iterations training (See Figure 3.2).

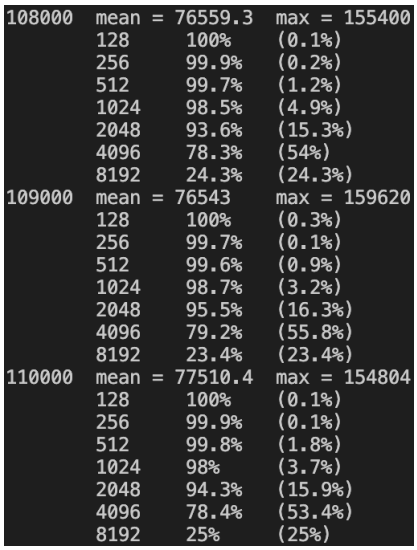

Figure 3.2: Score details after 350K training.[1]

[1]Note that the printed iterations here are wrong, since the model is trained 240K iterations at first and stop, then load the weight, and trained for 110K iterations for the second time. Totally 350K iterations training.

Chapter 4

Conclusion

After this experiment, I found that **patterns** play an important role in this task.
With original 4 patterns, the average reached 2048-tile percentage is only about **90%** after 350K iterations training. However, with another 4 patterns, totally 8 patterns are used, it reaches about **95%**.