Lab2: Temporal Difference Learning

311551147 高振群

April 2023

1 Result

The result is shown in Figure 1.

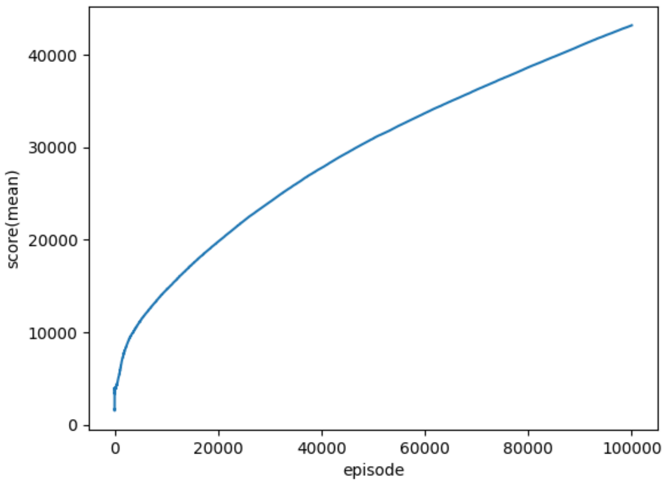

Figure 1: The mean score to episode curve.

2 Usage and implementation of n-tuple network

2.1 Usage

The n-tuple network is used as a feature extractor in this lab. By exploiting isomorphism (rotating or flipping the board doesn't change the context and information of the game), it captures 8 patterns on the board and estimates the expected future reward. The estimation derived from the n-tuple network will be used during the decision-making and TD update.

2.2 Implemenation

In this subsection, we only focus on the TODO part specified in the code template.

Source Code 1: TODO code in pattern class

```cpp
virtual float estimate(const board& b) const {
    // TODO
    float estimation = 0;
    for(const vector<int> iso: isomorphic) {
        size_t index = indexof(iso, b);
        estimation += weight[index];
    }
    return estimation;
}

/**
 * update the value of a given board, and return its
 ↪  updated value
 */
virtual float update(const board& b, float u) {
    /*
    TODO
    */
    u /= iso_last;
    float updated_value = 0;
    for(const vector<int> iso: isomorphic) {
        size_t index = indexof(iso, b);
        weight[index] += u;
        updated_value += weight[index];
    }

    return updated_value;
}

size_t indexof(const std::vector<int>& patt, const
 ↪  board& b) const {
```

```
30      // TODO, given a pattern and the board position,
     ↪  what is the index of that position in the weight
     ↪  table
31      size_t index = 0;
32      for(int i = 0; i < patt.size(); i++) {
33          index |= b.at(patt[i]);
34
35          if(i != patt.size() - 1)
36              index <<= 4; // "go for the next square"
37      }
38      return index;
39  }
```

Source code of the pattern class is shown in Source Code 1. In the estimate function, the pattern class checks all the isomorphism pattern on the board at line 4 and accumulates all the expected return. The update function update the weights of all the isomorphic pattern in the for loop starting from line 20. The indexof function calculates the index of the board position(or pattern on the board, to me more specifically) by using bitwise operation, the index is going to be used to update or retrieve the estimation value from the n-tuple weight.

3 Explanation of TD(0)

$TD(0)$ utilizes one-step look-ahead search to update the value estimation, by following the formula

$$V(s) \leftarrow V(s) + \alpha(r + V(s') - V(s))$$

, where α is the learning rate or update step size and s' is the next state. The mechanism of $TD(0)$ is to update the value function toward the direction of $r + V(s') - V(s)$, which is the TD error. Since $TD(0)$ only exploits one-step look-ahead, it can be used in non-episodic task, which makes it a more handy tool than Monte-Carlo method.

4 Implementation of action selection and TD backup

Source code of this section is implemented in learning class and is shown in Source Code 2. In the select_best_move function, we first consider every possible moves at line 4 and derive the immediate reward of taking that move, then we consider every possible popup situations at line 14 and derive the long term reward using the n-tuple network estimation. By taking both immediate reward and long term reward into account, we update the best move variable, best, at line 39 and 40. In the update_episode function, we update the weight array by

TD error, which is derived from backward TD update. The TD backup diagram is shown in Figure 2.

Source Code 2: TODO code in learning class

```cpp
state select_best_move(const board& b) const {
    state after[4] = { 0, 1, 2, 3 }; // up, right, down, left, read the constructor, the number represents the opcode
    state* best = after;
    for (state* move = after; move != after + 4; move++) {
        //(move->assign(b) returns a boolean, it is true if the action is legal)
        if (move->assign(b)) {
            // TODO, calculate the move->value() by considering every possible next state
            float immediate_reward = move->reward();
            float long_term_reward = 0;
            int empty_squares = 0;


            // Consider all possible state transitions
            for(int i = 0; i < 16; i++) {
                board after_state = move->after_state();
                // if that position is empty (0-tile), then it is possible that the new popup() tile can be there
                if(after_state.at(i) == 0) {
                    empty_squares++;
                    board board_with_4tile = after_state;
                    board board_with_2tile = after_state;

                    board_with_4tile.set(i, 2);
                    board_with_2tile.set(i, 1);

                    // if the next state is terminal state, then there is no long term reward
                    if(!is_terminal(board_with_4tile)) {
                        long_term_reward += 0.1 * (this -> estimate(board_with_4tile));
                    }
                    if(!is_terminal(board_with_2tile)) {
                        long_term_reward += 0.9 * (this -> estimate(board_with_2tile));
                    }
                }
            }

            if(empty_squares != 0)
                move->set_value(immediate_reward + long_term_reward / empty_squares); // each square is equally likely to have a new popup tile

            // update the cur best move, value() returns the esti
            if (move->value() > best->value())
                best = move;
        } else {
            move->set_value(-std::numeric_limits<float>::max());
        }
        debug << "test " << *move;
    }

    // return the best move based on the condition above
    return *best;
}

/**
 * update the tuple network by an episode
 *
 * path is the sequence of states in this episode,
 * the last entry in path (path.back()) is the final state
 *
 * for example, a 2048 games consists of
 *  (initial) s0 --(a0,r0)--> s0' --(popup)--> s1 --(a1,r1)--> s1' --(popup)--> s2 (terminal)
 *  where sx is before state, sx' is after state
 *
 * its path would be
 *  { (s0,s0',a0,r0), (s1,s1',a1,r1), (s2,s2,x,-1) }
 *  where (x,x,x,x) means (before state, after state, action, reward)
 *
 * (s0,s0',a0,r0) = (state.before_state(), state.after_state(), state.action(), state.reward())
 */
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    // probably should call the update function in public functions
    // probably should iterate over every states:
    // each state should be used to update the pattern in features(its actually the pattern class instances)
    // backward update
    float next_state_value = 0;
    for(int i = path.size() - 2; i >= 0; i--) {
        float TD_target = path[i].reward() + next_state_value;
        float TD_error = TD_target - (this -> estimate(path[i].before_state()));
        next_state_value = this -> update(path[i].before_state(), TD_error * alpha);
    }
}
```

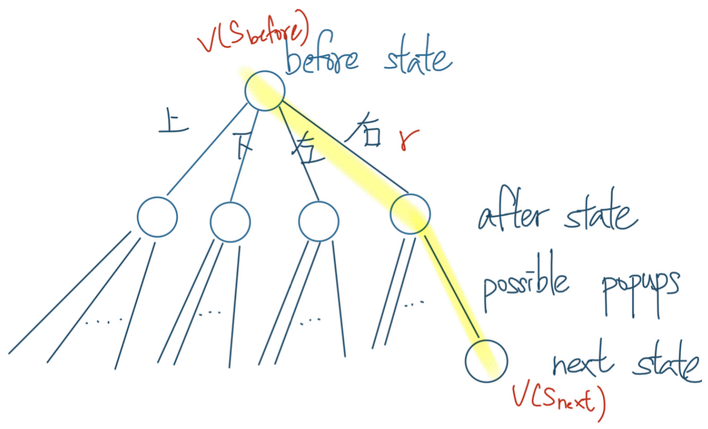

Figure 2: TD backup diagram. The yellow path is the sampled state transition from the trajectory.