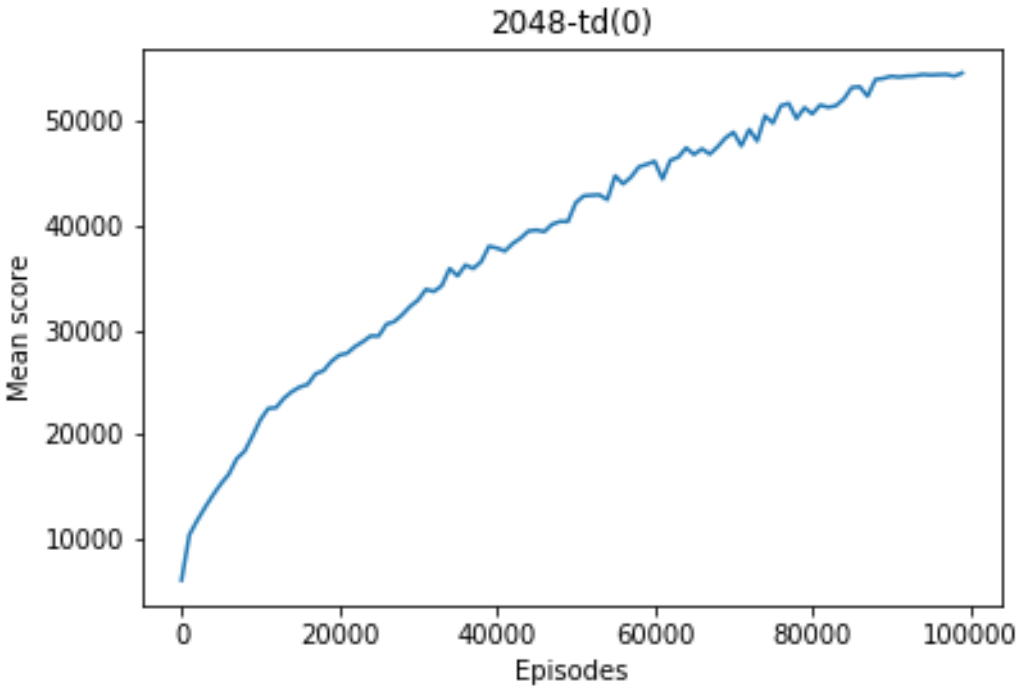
Lab5: Temporal Difference Learning

Report

1. **A plot shows episode scores of at least 100,000 training episodes (10%)**



2. Describe your implementation in detail. (10%)

TD-after-state

function EVALUATE(s, a)

$s', r \leftarrow$ COMPUTE AFTERSTATE(s, a)

return $r + V(s')$

Code:

```
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)) {
            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;
}
```

For the function here, we compute the four after state and estimate each after state reward. In the end, return the best move.

function LEARN EVALUATION(s, a, r, s', s'')

$a_{next} \leftarrow \underset{a \in A(s'')}{\operatorname{argmax}} EVALUATE(s'', a')$

$s'_{next}, r_{next} \leftarrow COMPUTE\ AFTERSTATE(s'', a_{next})$

$V(s') \leftarrow V(s') + \alpha(r_{next} + V(s'_{next}) - V(s'))$

Code:

```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    float exact = 0;
    for (path.pop_back() /* terminal state */; path.size(); path.pop_back()) {
        state& move = path.back();
        float error = exact - (move.value() - move.reward());
        debug << "update error = " << error << " for after state" << std::endl << move.after_state();
        exact = move.reward() + update(move.after_state(), alpha * error);
    }
}
```

The error is $\mathrm{r}_{\mathrm{next}} + V(s'_{next}) - V(s')$, and update the $\mathrm{V}(\mathrm{s}') \leftarrow V(s') + \alpha(error)$. The path parameter save the (s0, s0', a0, r0), (s1, s1', a1, r1) means (before state, after state, action, reward).

==========

TD-state

function EVALUATE(s, a)
  $s', r \leftarrow$ COMPUTE AFTERSTATE(s, a)
  $S'' \leftarrow$ ALL POSSIBLE NEXT STATES(s')
  return $r + \Sigma_{s'' \in S''} P(s, a, s'') V(s'')$

Code:

```
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)) {
            //TD-state
            int space[16], num = 0;
            for (int i = 0; i < 16; i++)
                if (move->after_state().at(i) == 0) {
                    space[num++] = i;
                }
            float E = 0.0;
            if (num){
                for(int i = num-1; i>=0; i--){
                    board temp_board;
                    temp_board = move->after_state();
                    temp_board.set(space[i], 1);
                    E = E + estimate(temp_board)*0.9/num;
                    temp_board.set(space[i], 2);
                    E = E + estimate(temp_board)*0.1/num;
                }
            }
            move->set_value(move->reward() + E);
            if (move->value() > best->value())
                best = move;
        } else {
            move->set_value(-std::numeric_limits<float>::max());
        }
        debug << "test " << *move;
    }
    return *best;
}
```

First for loop compute all possible after state for pop up. Second, we estimate all possible next state and times the probability for pop up. Finally return the best move.

function LEARN EVALUATION(s, a, r, s', s'')
  $V(s) \leftarrow V(s) + \alpha(r + V(s'') - V(s))$

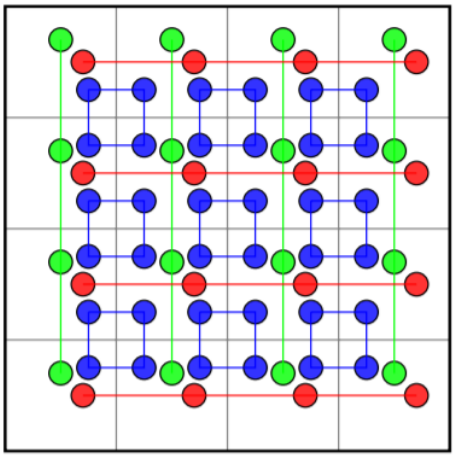
Code:

```
void update_episode(std::vector<state> &path, float alpha = 0.1) const {
    float exact = 0;
    update(path.back().before_state(), alpha * -estimate(path.back().before_state()));
    for (path.pop_back() /* terminal state */; path.size(); path.pop_back()) {
        state &move = path.back();
        float error = move.value() - estimate(move.before_state());
        update(move.before_state(), alpha * error);
    }
}
```

The TD error here is $\mathrm{r} + \mathrm{V}(\mathrm{s}'') - V(s)$ and move.value is $\mathrm{r} + \mathrm{V}(\mathrm{s}'')$, estimate(move.before_state()) is $\mathrm{V}(\mathrm{s})$. after that update the $\mathrm{V}(\mathrm{s}) \leftarrow \mathrm{V}(\mathrm{s}) + \alpha * error$.

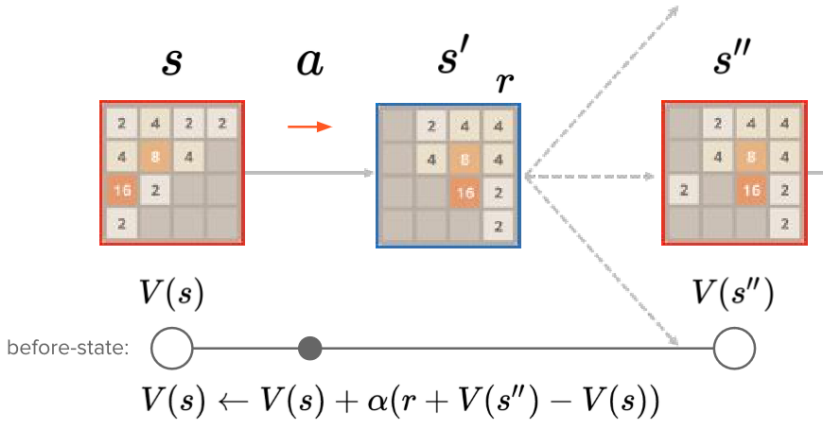
3. Describe the implementation and the usage of nn-tuple network. (10%)

```
// 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}));
```



n-Tuple network is used to find out the feature in the board. By rotating and reflecting, the tuple network will give different values. The tuple network is no hidden layers and the input vector is sparse vector with 0 or 1 with just one output node. In practice we use multiple n-tuple to improve our work, by capturing many different features in the board.

4. Explain the TD-backup diagram of V(state). (5%)

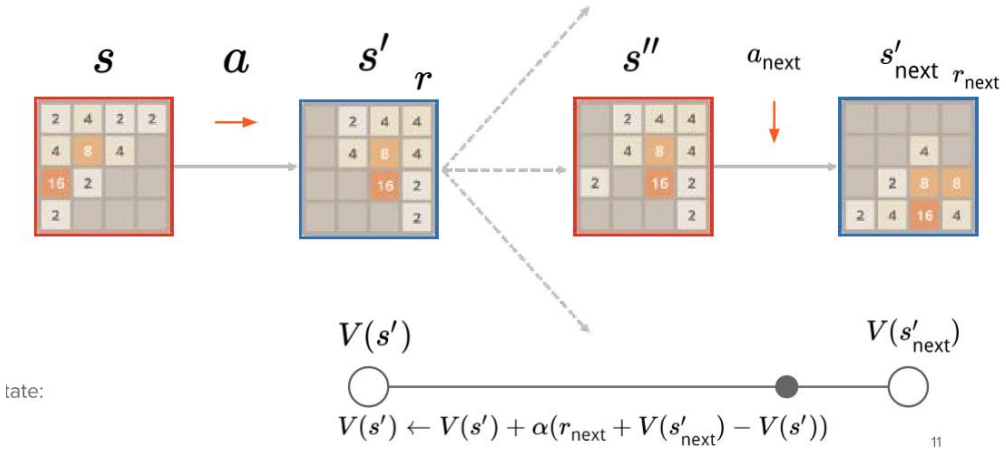


TD-state use state s'' (sample update) and reward to refresh the current state of value. $r + V(s'')$ estimated rewards after 1 step.

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

First we will compute after state given state and action, then we will estimate all possible next state and times the possible state probability and add the reward. Finally, we will return the best move with max estimate.

6. Explain the TD-backup diagram of V(after-state). (5%)

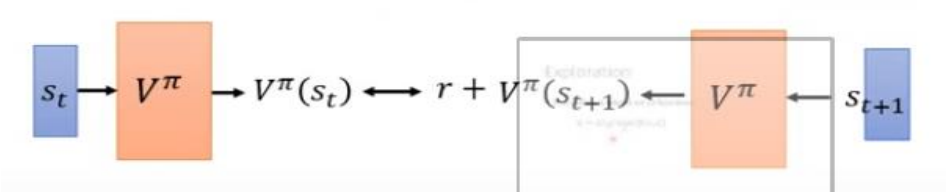


We should minimize the TD loss between V(after-state) and V(next after-state) and update the V(after-state)

7. Explain the action selection of V(after-state) in a diagram. (5%)

Action selection of V(after-state) is quite simpler than V(state), just estimate the V(s') and add the reward.

8. Explain the mechanism of temporal difference learning. (5%)



TD learning need to calculate the error between V(s) and V(s') + reward and keep update the v(s) each step until episode terminate. The λ in TD(λ) control the steps we should consider, the larger λ means we care about the reward in future.

TD learns from incomplete episodes by bootstrapping.

TD updates a guess towards a guess

TD can get smaller variance but higher bias.

TD converges to V(s)

9. Explain whether the TD-update perform bootstrapping. (5%)

TD-update perform bootstrapping, cause we need to estimate the expectation of new number pop up on the board.

10. Explain whether your training is on-policy or off-policy. (5%)

My training is on-policy TD (Sarsa), cause we have record the path by (s1, a1, r1, s1', s2), and update the V(state, action):

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big].$$

11. Other discussions or improvements. (5%)

This game is very time consuming, when I was training game of 2048. I found my 2048 get 75% win rate in 43000 episode, but grow up slowly to 86.8% win rate in 100,000 episode. Most of the training time the win rate is around 80%, I think is my alpha's problem or the more feature pattern I should add.

Performance