DLP Lab3 - Temporal Difference Learning Report

0816124 Hao-Chen Lin

April 1, 2022

1 Introduction

In this lab, we will using temporal difference learning (TD) algorithm and n-tuple network to solve the 2048 game, try to understand the concept of before-state and after-state and update the value function of before-state instead of after-state that using in source code.

2 Report Questions

2.1 A plot shows episode scores of at least 100,000 training episodes.

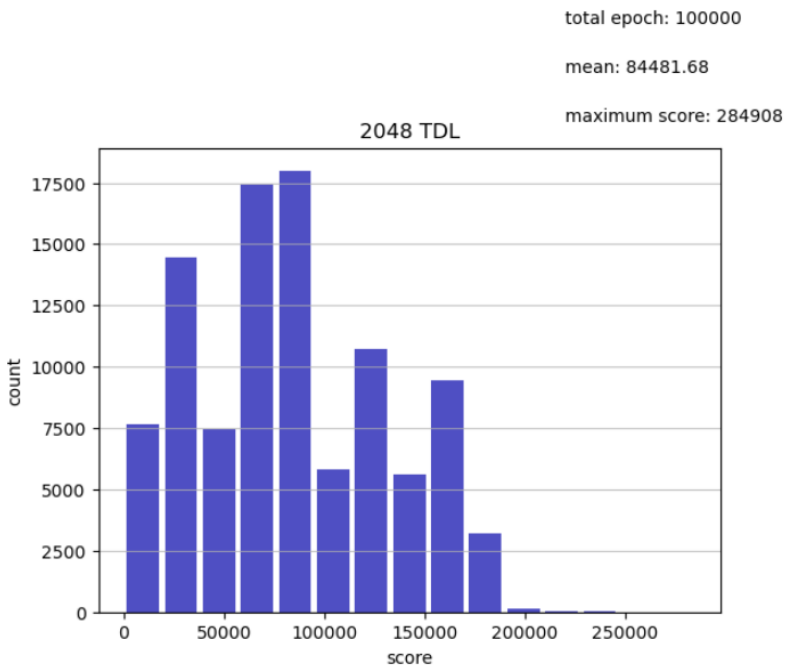

Figure 1: This is a plot shows episodes scores for 200,000 training epochs.

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

The n-tuple network need to choose n positions in game grid as n cells, and store the power of 2 as inputs in each cells, if there's (0, 2, 32, 8) in game grid, we store (0, 1, 5, 3) as inputs in n-tuple.

To calculate the index of a n-tuple network in value function, consider the 2048 game is hard to exceed $2^{15} = 32768$ in one cell, so we can take out inputs as hexadecimal number and transform them into decimal number, if the inputs in n-tuple is (0, 1, 5, 3), the corresponding index to value function is $0 * 16^3 + 1 * 16^2 + 5 * 16^1 + 3 * 16^0 = 339$, then we do the same thing for all 8 of isomorphism of this n-tuple network, total value of this n-tuple network will be the summary of value of all isomorphism.

Also we can have several n-tuple network, just repeat the step above for each n-tuple network, and the final value of a game grid will be the summary of value of all n-tuple networks.

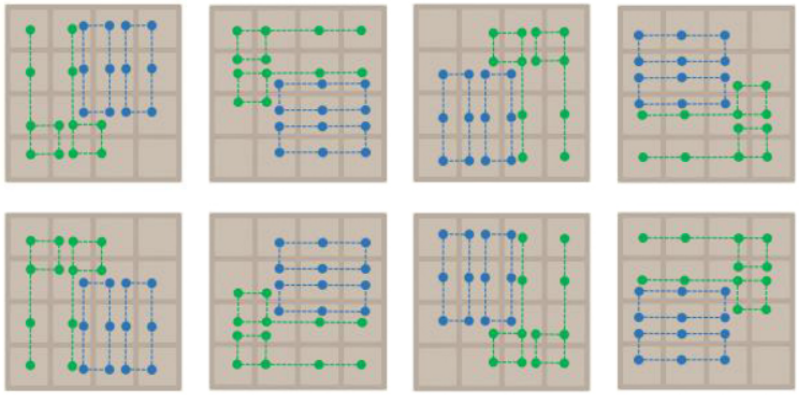

Figure 2: Example of 8 isomorphism for 4 of 6-tuple network.

The usage of n-tuple network is to provide a value function approximator, since the number of state is huge that about $16^{16} \approx 10^{19}$ states, there's no such huge storage to use, we can reduce the number of states by using n-tuple network method. There's 4 of 6-tuple network in this lab, thus reduce number of states to $4 * 16^6 \approx 6 * 10^7$.

2.3 Explain the mechanism of TD(0).

The update function for Monte-Carlo Learning is formula (1), it play through the whole trajectory to get total return G_t, but in TD(0) we only focus on next state and get the estimated return $R_t + \gamma V(s_{t+1})$, then we replace G_t to this estimated return and we can get formula (2).

$$V(s_t) = V(s_t) + \alpha(G_t - V(s_t)) \tag{1}$$

$$V(s_t) = V(s_t) + \alpha(R_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{2}$$

2.4 Explain the TD-backup diagram of V(after-state).

For a 5-tuple(s, a, r, s', s''), we firstly find the best action a_{next} to the given after state s', then compute next after state s'_{next} and reward r_{next}, then update the value of s' using TD(0).

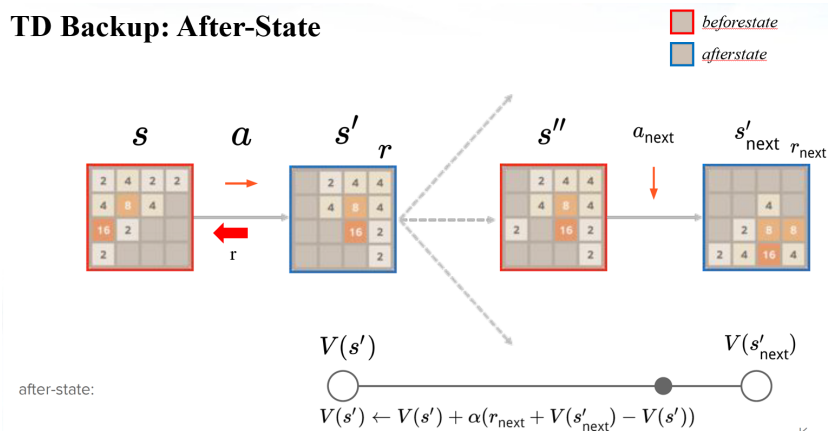

Figure 3: TD-backup diagram of V(after-state).

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

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

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

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

2.5 Explain the action selection of V(after-state) in a diagram.

For the input (s, a), we compute the after state s' and reward r that state s take action a, then since there's only one possible outcome after state, we can simply return $r + V(s')$, to select the best action to a given state, just need to find $\underset{a}{argmax}$ EVALUATE(s, a).

function EVALUATE(s, a)

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

return $r + V(s')$

2.6 Explain the TD-backup diagram of V(state).

For a 5-tuple(s, a, r, s', s''), we want to update the value function of before state s, just using update formula of TD(0).

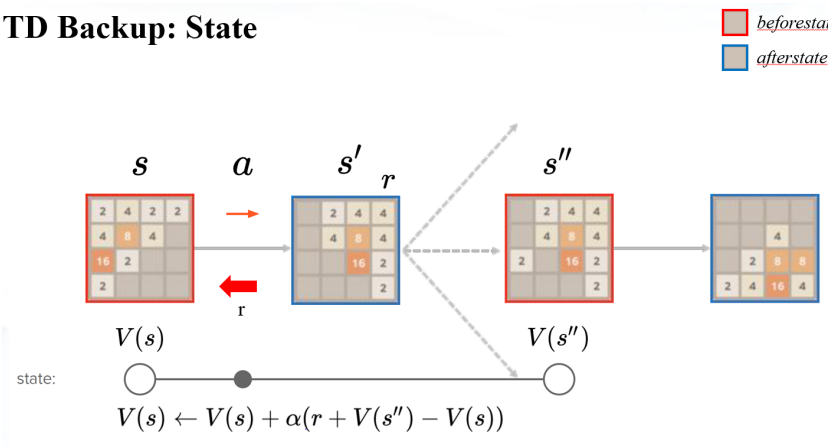

Figure 4: TD-backup diagram of V(state).

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

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

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

For the input (s, a), we compute the after state s' and reward r that state s take action a, then compute all possible next state s' into a set of states S'', then the value of this state-action pair will be reward r plus sum of value all possible next state multiply its corresponding probability.

function EVALUATE(s, a)

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

$S'' \leftarrow$ ALL POSSIBLE NEXT STATES(s')

return $r + \Sigma_{s'' \in S''} P(s, a, s'') V(s'')$

2.8 Describe your implementation in detail

There's all 5 of TODO blocks that need to implement.

2.8.1 estimate

For a given board, we want to calculate the total value of a specific n-tuple network, just sum up value of all 8 isomorphism of the given pattern.

2.8.2 update

For a given board and update value, we firstly split this value into 8 since there's 8 kind of isomorphism, then update value function of all isomorphism with this update value after split.

2.8.3 indexof

For a given pattern and board, we want to calculate the index in value function. We firstly extract the input of this pattern at corresponding position of given board, then convert this input from decimal into hexadecimal by shifting $4 * i$ bits at ith position since $2^4 = 16$.

2.8.4 select best move

For a given state-action pair, aim to calculate its value by using formula in 2.7, so firstly find all of its space position, then for each space, pop up 2 and 4 in turn and calculate value of board after pop up, their corresponding probability should be (0.9 / number of space) and (0.1 / number of space). Finally, sum up all term multiply its probability and reward of this move, and it will be our value.

```
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
            board after = move->after_state();
            std::vector<int> space;
            for(int i=0; i<16; i++) if(after.at(i) == 0) space.push_back(i);
            float value = 0.0;
            for(auto idx: space){
                after.set(idx, 1);
                value += estimate(after) * 0.9 / space.size();
                after.set(idx, 2);
                value += estimate(after) * 0.1 / space.size();
                after.set(idx, 0);
            }
            move->set_value(move->reward() + value);
            if (move->value() > best->value()) best = move;
        } else {
            move->set_value(-std::numeric_limits<float>::max());
        }
        debug << "test " << *move;
    }
    return *best;
}
```

Figure 5: Function implementation of select best move.

2.8.5 update episode

In this function, we do TD-backup update using a vector of state, i implement this function by applying formula in 2.6, in each step, calculate TD-error of each state then update value of its before-state, then store this value for purpose of calculating TD-error in next step.

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

Figure 6: Function implementation of update episode.

2.9 Other discussions or improvements.

In view of efficiency, formula in 2.7 need to compute value of all possible next state to, and do this computation for each action to select a single action in one step, it will be costly and will run too slow, so i change calculation method in this part, i just randomly pop up to produce the next state, and evaluate value only using this next state instead of compute all possible next state.

In this way, computation be faster than origin method by about 10 times, but it will cost performance as well, after 100,000 episodes, the ratio that score exceed 2048 is 61% while using origin method it will be about 90%.