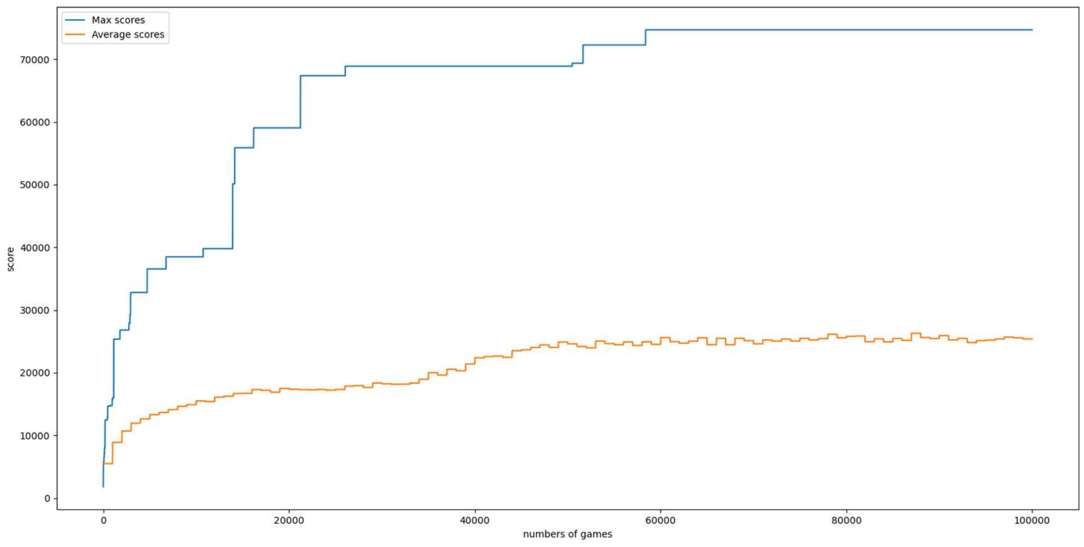
0716325 曾正豪 Lab3

1. **A plot shows episode scores of at least 100,000 training episodes:**



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

   In 2048 games, the number of all possible states is huge. In each box, they could be 2^1 to 2^16, there are 16 possible values, so there will be 16^16 possible states in total. It's too large to save in memory. Hence, in this lab, we were asked to use n-tuple network to save the memory. In the n-tuple network of this lab, it has an array called "weight". Once we want to get the value of an n-tuple feature, we would convert the feature into an integer index, then use this index to find its weight value.

3. **Explain the mechanism of TD(0)**

   Compare to Monte Carlo learning trying to learn the value from final result, the TD learning learns value from the next state. Hence, TD learning can learn at any time, even the game have not finished yet.

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

   The diagram shows that during the training, it will update the value of after-state. With TD target equals to "the next state's action's reward" plus "the estimated value of next after-state".

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

   It's to select the action that has the greatest value of "reward after performing that action" plus "the estimated value of after-state".

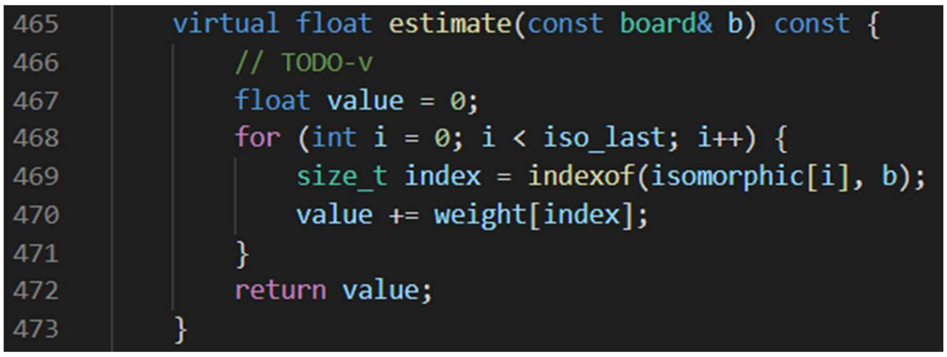
6. **Explain the TD-backup diagram of V(state).**

   The diagram shows that during the training, it will update the value of before-state. With TD target equals to "current state's action's reward" plus "the estimated value of next before-state".

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

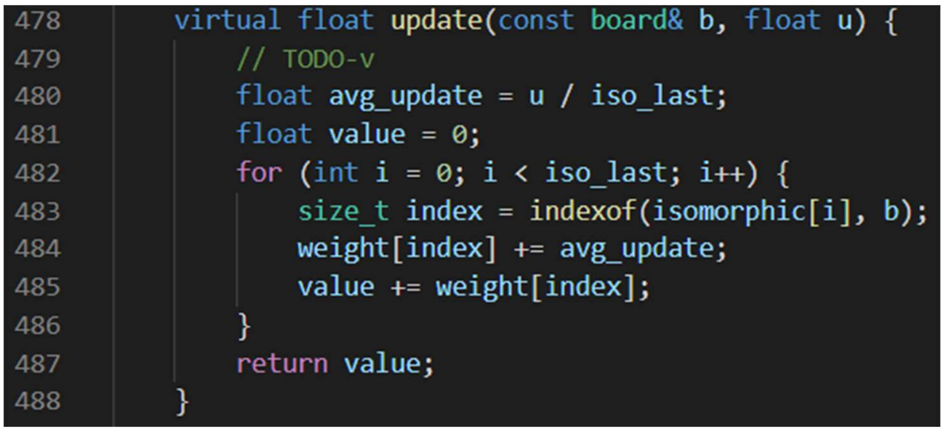
It’s to select the action that has the greatest value of “reward after performing that action” plus “the expect estimated value of next before-state after performing that action”.

8. Describe your implementation in detail.



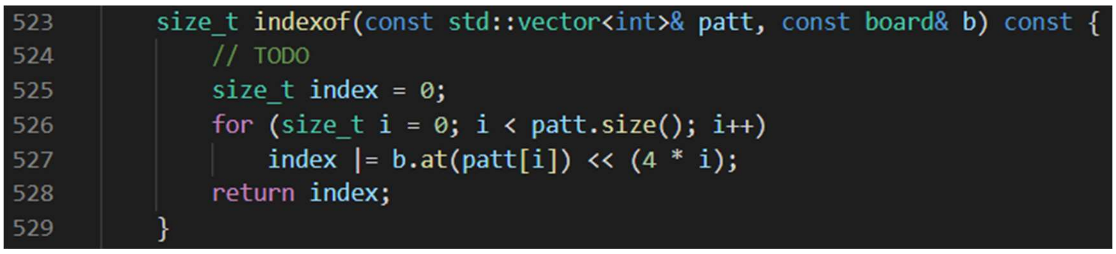
```
    virtual float estimate(const board& b) const {
        // TODO-v
        float value = 0;
        for (int i = 0; i < iso_last; i++) {
            size_t index = indexof(isomorphic[i], b);
            value += weight[index];
        }
        return value;
    }
```

In the “estimate” function, since the board can rotate and reflect, the return value would be the sum of these feature values. In each isomorphic, I would use the “indexof” function to convert the feature on the given board into an integer index. Then, get the weight of that index and add it to the final value.



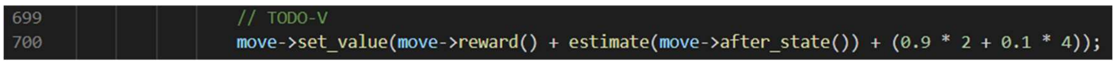
```
    virtual float update(const board& b, float u) {
        // TODO-v
        float avg_update = u / iso_last;
        float value = 0;
        for (int i = 0; i < iso_last; i++) {
            size_t index = indexof(isomorphic[i], b);
            weight[index] += avg_update;
            value += weight[index];
        }
        return value;
    }
```

In the “update” function, since the estimated value comes from various isomorphic, we need to average the updating value to these isomorphic features. In each isomorphic, I would use the “indexof” function to convert the feature on the given board into an integer index. Then update these values to the weight.



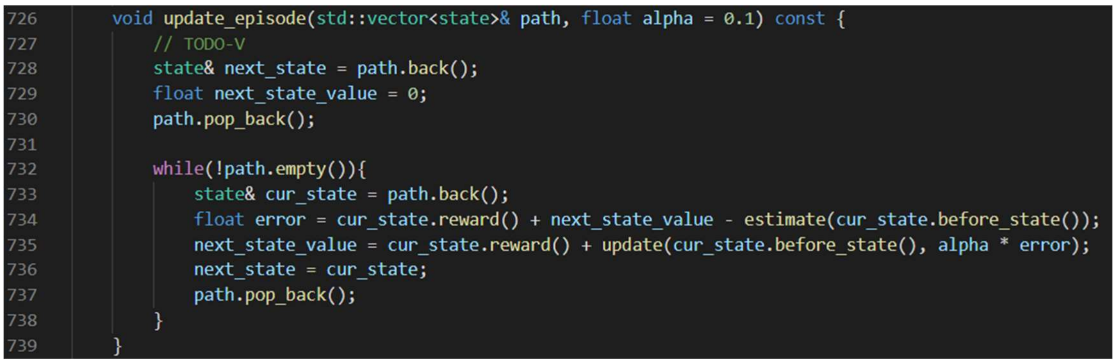
```
    size_t indexof(const std::vector<int>& patt, const board& b) const {
        // TODO
        size_t index = 0;
        for (size_t i = 0; i < patt.size(); i++)
            index |= b.at(patt[i]) << (4 * i);
        return index;
    }
```

From the constructor of class “pattern”, we know that it allocates a floating array with length 2^(n*4) to weight. And in the index, every 4 bits’ data represent the value of that position. Hence, in the “indexof” function, I follow the way how it represents the index to construct it. For each pattern in the n-tuple, I would use the "at" function to get its data, and put it in the corresponding position of "index" integer.



```
                    // TODO-V
                    move->set_value(move->reward() + estimate(move->after_state()) + (0.9 * 2 + 0.1 * 4));
```

In the pseudocode of **function** EVALUATE(s, a) of TD learning in Spec, we need to evaluate "$r + \Sigma_{s'' \in S''} P(s, a, s'')V(s'')$". Since V(***s''***) is the state after the popup operation on after-state ***s'***. Besides, the popup operation would popup 2 with 90% probability, popup 4 with 10% probability. Hence, the $\Sigma_{s'' \in S''} P(s, a, s'')V(s'')$ term would be 0.9 * (V(***s'***) + 2) + 0.1 * (V(***s'***) + 4) = V(***s'***) + (0.9*2+0.1*4).



```
    void update_episode(std::vector<state>& path, float alpha = 0.1) const {
        // TODO-V
        state& next_state = path.back();
        float next_state_value = 0;
        path.pop_back();

        while(!path.empty()){
            state& cur_state = path.back();
            float error = cur_state.reward() + next_state_value - estimate(cur_state.before_state());
            next_state_value = cur_state.reward() + update(cur_state.before_state(), alpha * error);
            next_state = cur_state;
            path.pop_back();
        }
    }
```

In the "update_episode" function, we need to update the value from the last state of the path to the first one. According to the formula of before-state TD learning,

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

we will update the before-state value V(s) in each time steps. The "r" in the formula is current state's reward, " V(s'') " is the next before-state's estimated value, " V(s) " is current before-state's estimated value. We'll use these information to calculate the TD error and update the before-state's value.

9. Other discussions or improvements.

When I was training the model, I found that sometimes its win rate would start decreasing after about 200,000 episodes. I thought that this n-tuple network may be overfitting at that time. Hence, I added some code to let its learning rate α to be half every 100,000 episodes. After that, this n-tuple network's win rate would not decrease any more, even training in millions of episodes.

I tried different n-tuple networks in this lab. First of all, I didn't change the n-tuple in the sample code. Then, I found that no matter how long did the network trained, its 2048 win rate always can't exceed 75%. Hence, I decided to modify the n-tuple network, adding 2 different patterns, they are {0,1,2,4,5,8} and {0,1,2,4,8}. After that, my n-tuple network grew faster than the original networks. Besides, its 2048 win rate could reach 85% after 200,000 episodes.