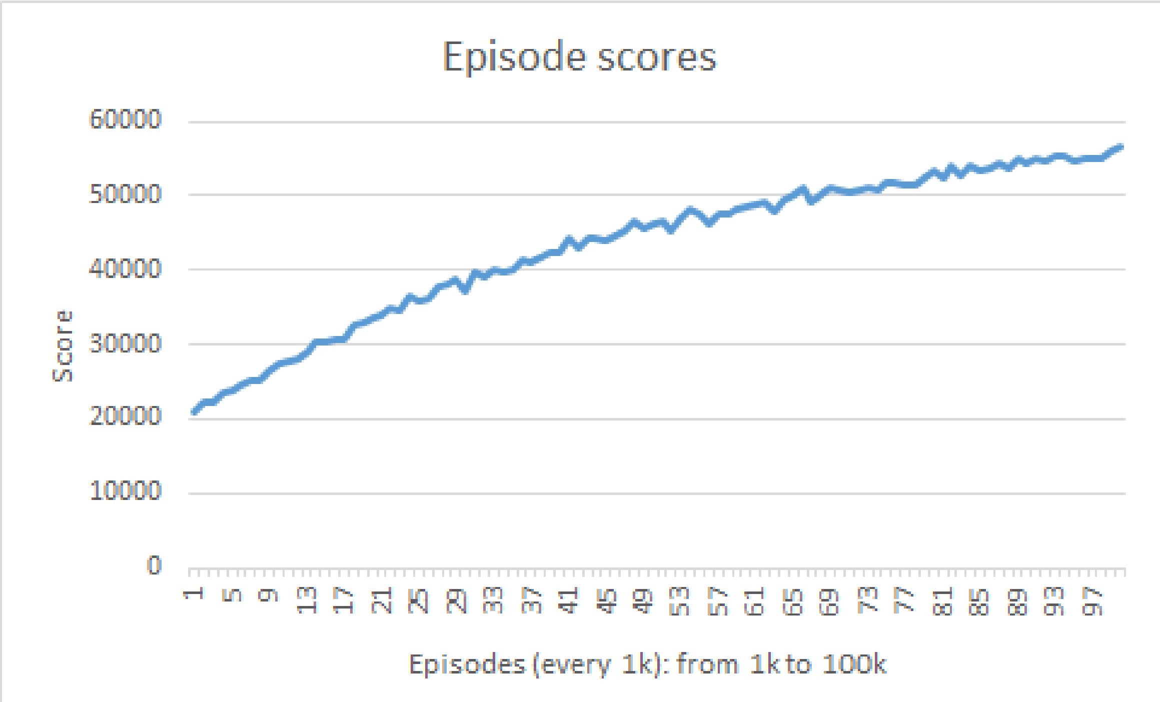
Deep Learning and Practice

Lab2: Temporal Difference Learning

H092093 林佳縈

- **A plot shows episode scores of at least 100,000 training episodes**



▲ Episode scores of 100,000 training episodes

To plot the episode scores chart, I took the mean value of every 1000 episodes as a unit. Therefore, there are totally 100 points on the chart up above which indicates the average value of every 1k episodes from episode 1 to episode 100,000.

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

According to TA's introduction on n-tuples, the reason we use n-tuple networks is because it is impossible to record all the state's and the value respectively.

The board of 2048 is 4*4, which means that there would be 2^{64} possibilities, which requires too much memory. So if we use 4 kinds of 6-tuple network, it can be 4 * 15^6 * 4, which can greatly reduce the memory usage.

In the given sample code, a weight table is defined in the feature class and is initialized to later on will be used to get the state value. Notably, 8 isomorphisms of a pattern share the same weight table.

- **Explain the mechanism of TD(0)**

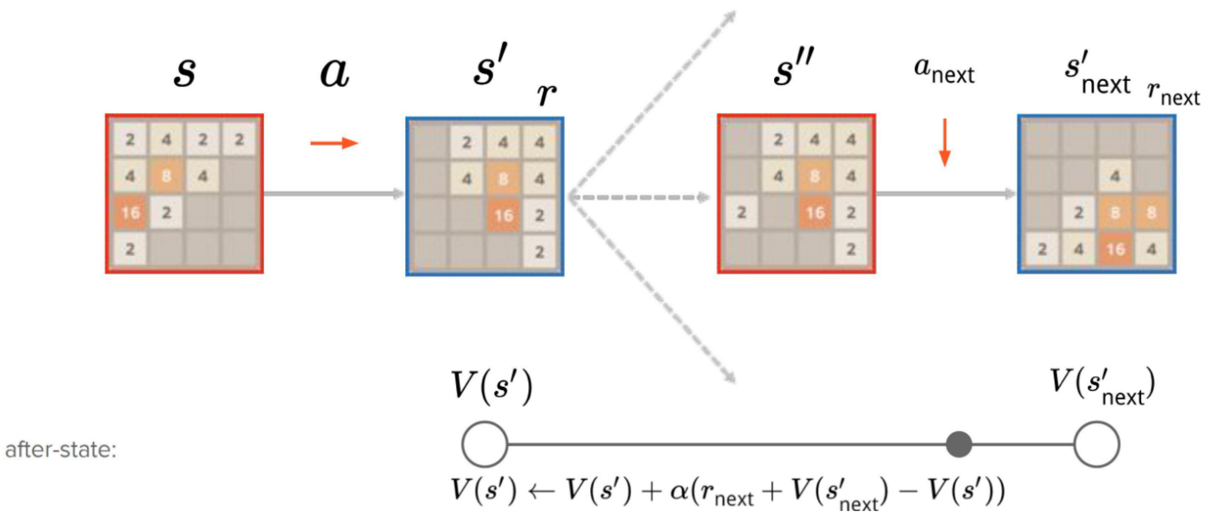
The mechanism of TD(0) is to update the state value by reward and value function ***every time step, meaning don't have to wait until the episode is over, it can directly use the next state value to do an update.*** Compared to Monte-Carlo Backup(another model-free learning method), which has to wait until an episode is over to get the return value to update state value:

$$\text{MC update: } V(S_t) \text{ <- } V(S_t) + a(G_t - V(S_t))$$

TD backup update value every single step:

$$\text{TD backup: } V(S_t)\text{<- } V(S_t) + a(R_{t+1} + V(S_{t+1}) - V(S_t))$$

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



While we are doing TD-learning, after an episode is over, we have to do the updating process: from terminal to initial state, we update every estimated state value by another estimated value on the next time step, which is a kind of bootstrapping method. There are total 2 different updating methods: (before-)state, after-state TD-backup, classified by which state value is targeted to be updated:

In the after-state version, the value of **after-state** is updated with the value of **after-state** at the next time step.

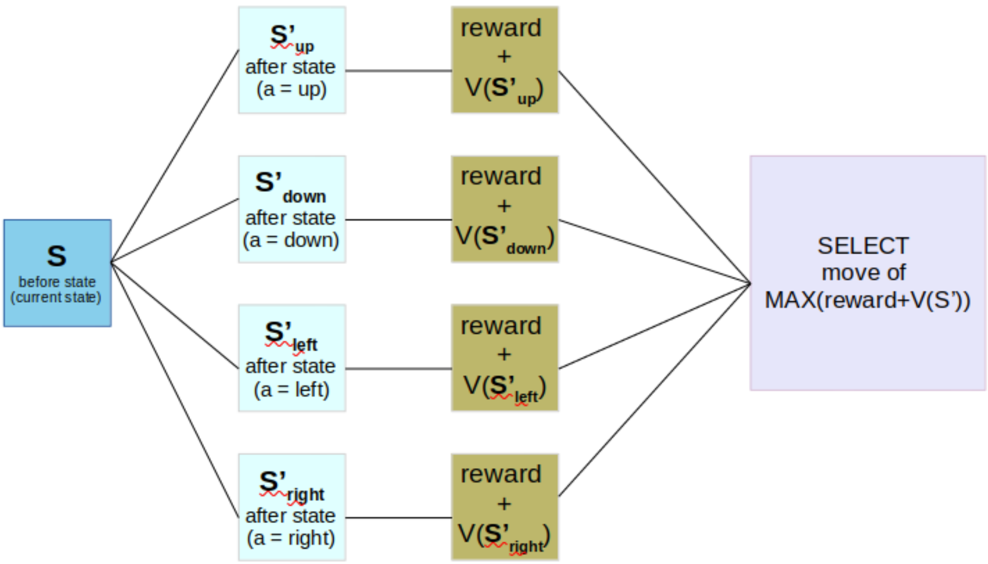
This is how it is backuped:

$$td\_error: (reward_{next} + V(s'_{next})) - V(s')$$

$$V(s') = V(s') + alpha * td\_error$$

where alpha is the learning rate (step-size), V(s') is the current estimated after state's value and $V(s'_{next})$ is the estimated next after-state's value. So in the above diagram the value of the blue box S' will be updated based on the value of the blue of S'_{next} on the next time step.

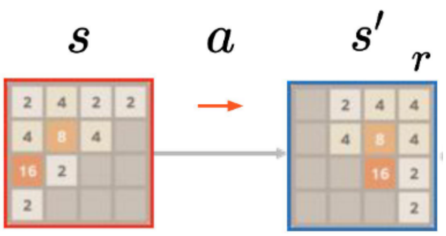
- **Explain the action selection of V(after-state) in a diagram**



▲ Diagram 1: after-state version action selection process

Diagram 1 shows briefly the workflow of the after-state version in selecting the best move. Here is the explanation:

Let's start with the following example:



1. We are at the current (before)-state S. In order to choose the best next move, we have to consider the following four cases: moving up/down/left/right.

2. Then, we take the action, moving up/down/left/right which makes the current (before)-state transfer to the current after-state s', that's to say, we'll have four corresponding after states: S'_{up} / S'_{down} / S'_{left} / S'_{right}

3. Thus, we calculate the corresponding value of reward + value of current_after_state V(s'), so we'll have:

$$value_1 = reward + V(S'_{up})$$

$$value_2 = reward + V(S'_{down})$$

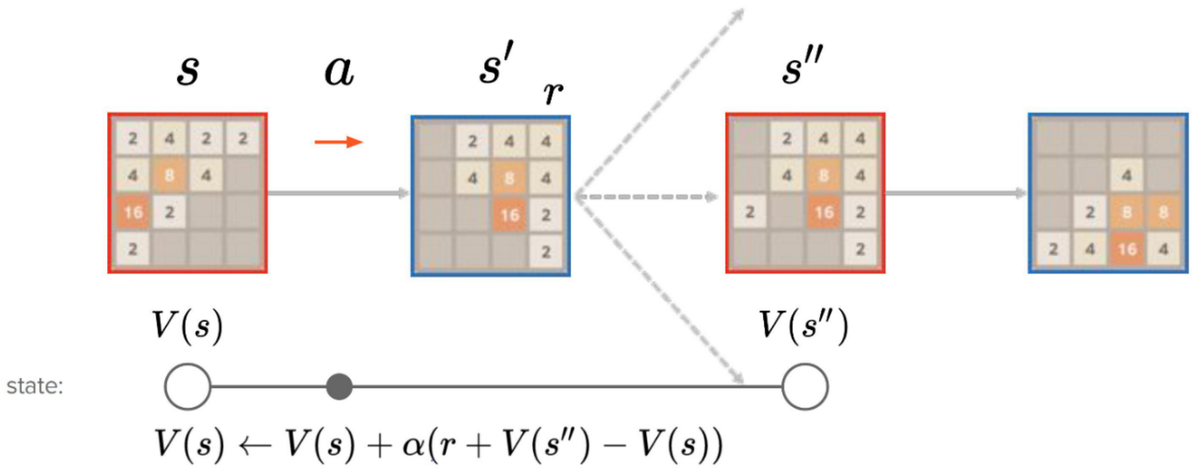
$$value_3 = reward + V(S'_{left})$$

$$value_4 = reward + V(S'_{right})$$

4. Finally, we'll choose the move which forms the maximum value among the four.

$$\textbf{\textit{best move}} = \textbf{\textit{argmax}}_{move = \{up,\, down,\, left,\, right\}}(\textbf{\textit{value}})$$

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



▲ Diagram 2: State version TD backup diagram

Similar to TD-backup diagram of V(after-state), there are only slight differences when it comes to state-version td-backup. The significant difference is the object that is updated. In the state version, the value of **before-state** is updated with the value of **before-state** at the next time step.

This is how it is backuped:

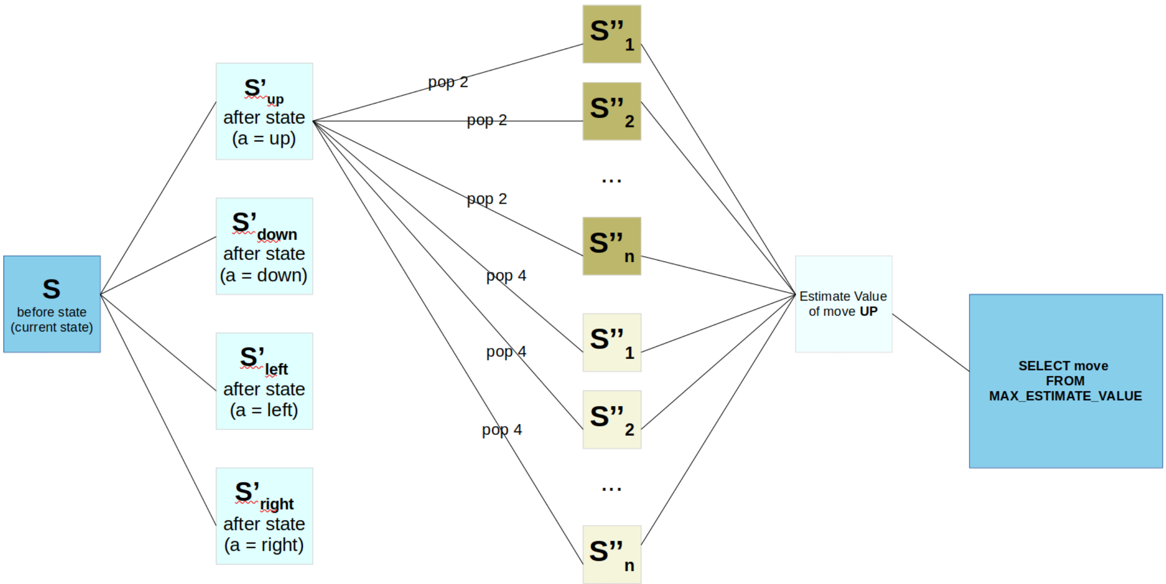
$$\textbf{\textit{td\_error: (reward + V(s''))- V(s)}}$$

$V(s) = V(s) + alpha * td\_error$

where alpha is the learning rate (step-size), V(s) is the current estimated state's value and V(s'') is the estimated next state's value. Therefore, we can tell that the difference between TD-backup diagrams of V(state) and V(after-state) is how it is backuped:

State version's td error to calculate the difference between *reward + next_**before_state**_estimated_value* and *current_**before_state**_estimated value*, while after-state version will calculate the difference between *next_state_reward + next_**after_state**_estimated_value* and *current_**after_state**_value.*

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

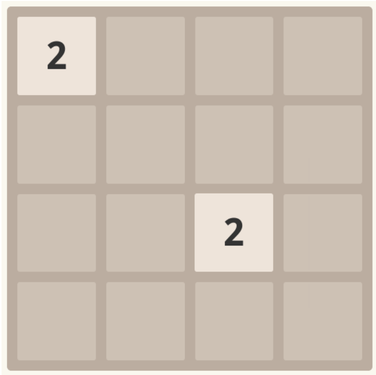


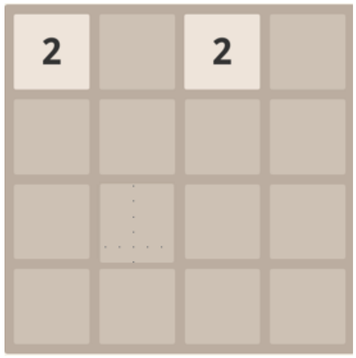
▲ Diagram 3: state version action selection process

The diagram up above shows briefly how state version in selecting the best move works and here is the explanation:

Let's take the following board for example:

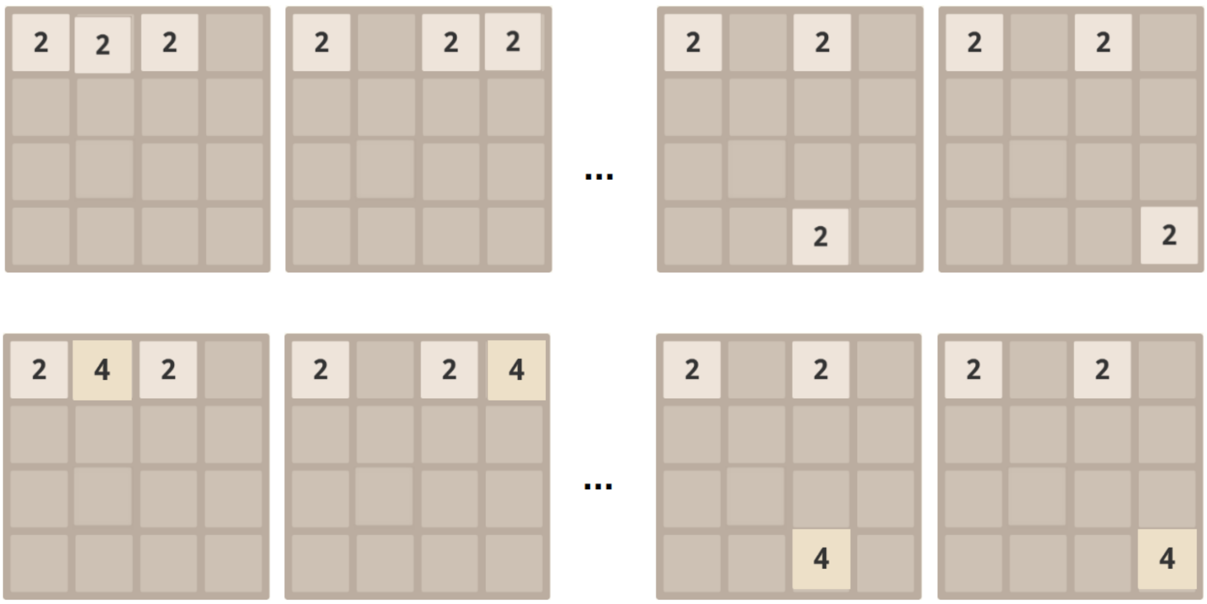
1. Board on the left hand side is where we are now. It is the before state S, which is our current state. In order to determine which move we are going to take, we have to test all move possibilities: moving up/down/left/right, and calculate the corresponding estimation value to decide the best move. For example, we test moving up first.





▲ Current state : **S** Before-state ▲ Moving Up: **S'**
after-state

2. After we take action of moving up, in after state S', there will be 2*14 possible after-state boards, because there may either pop 2 or 4 in any blanks in the S' board, which leads to the next state S''. Therefore, we have to enumerate every kind of possibility of next state S'', like in **Diagram 3** the brown cubes indicate, here list the possibilities:



▲ Possibilities of all next-state S'' : after-state S'+ (pop2 or pop4)

After all the next-states are listed, we then calculate the corresponding estimation value according to this formula:

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

so we'll have

pop 2: 0.9*estimate(next_state_board)

pop 4: 0.1*estimate(next_state_board)[1]

Therefore, the estimation value of making this move will be:

estimation_value_of_the_move = reward + (pop2 + pop4) / numbers_of_blank

By calculating the estimation value of making each move (up/down/left/right), we can then select the best move by choosing the move which leads to the maximum estimation value:

$$\textbf{best move} = \textbf{argmax}_{action = \{up,\ down,\ left,\ right\}}\ \textbf{(estimation\_value)}$$

- **Describe your implementation in detail**

To train to play the 2048 game this time, there are a total of 5 TODOs within the given sample code. To describe my implementation in detail, I'll explain what I've done separately and sequentially with algorithm-like descriptions.

I implemented TODO1, 2, 3 based on the sample github, so in the following description I'll focus more on TODO 4 and 5.

TODO 1: estimate

To estimate the given board's value, we have to look up the weight table and calculate the given board's patterns and its isomorphism value.

```
for (i = 0 to last_isomorphism):
    //look up the weight table
    value += indexof(isomorphic[i], b)'s value
```

TODO 2: udpate

In this part, we have to update 8 isomorphic feature values of specific patterns and return the sum of the updated value to make it the state's new value.

TODO 3: indexof

[1] According to 2048's rule, the probability of popping 2 is 0.9 and popping 4 is 0.1.

Since I use the default setting of 6-tuple network with each tile represented by 4 bit, so getting index will be:

```
index |= b.at(pattern[i]) << (4*i)
```

TODO 4: select_best_move

Here is the pseudo code how I implement the select_best_move part:

```
for ( move = up, down, left, right):

    all_possible_board = move.after_state()

    for (i=0 to 15):
        if all_possible_board.at(i) == 0:
            blank+=1
            all_possible_board.set(i, 1)
            pop_two += 0.9 * estimate(all_possible_board)
            all_possible_board.set(i, 2)
            pop_four += 0.1 * estimate(all_possible_board)
            all_possible_board.set(i, 0)

        sum = (pop_two + pop_four)/blank
        move->set_value(reward + sum)

        if move->value() > best->value():
            best = move
```

The goal of the step is to pick the move with the best estimation value by considering the four cases: moving up/down/left/right by iterating through the for loop.

After picking a move:

First, record the after-state-board to all_possible_board:

```
all_possible_board = move.after_state()
```

Second: **`for(i=0 to 15)`** enumerate every kind of next state possibilities based on current move by iterating through the board and popping up 2 or 4 at any blank.

About how to find a blank: if the board's tile is 0, it means it's a blank at that place.

If it is a blank, then pop either 2 or 4 at that place:

Case 1: pop 2 on all_possible_board, then calculate the estimation value by 0.9* board_after_pop_2's_value:

```
all_possible_board.set(i, 1)

pop_two += 0.9 * estimate(all_possible_board)
```

Case 2: pop 4 on all_possible_board, then calculate the estimation value by 0.1* board_after_pop_4's_value

```
all_possible_board.set(i, 2)

pop_four += 0.1 * estimate(all_possible_board)
```

After calculation, the blank with 2 or 4 popped must be set back to 0.

```
all_possible_board.set(i, 0)
```

Value is set based on this formula: Value = r + Sigma(P(s,a,s'')V(S'')), where Sigma(P(s,a,s'')V(s'')) is the sum here

```
sum = (pop_two + pop_four)/blank

move->set_value(reward + sum)
```

if current move brings bigger value, then it is better:

```
if move->value() > best->value():

      best = move
```

TODO 5: update_episode

Here is the pseudo code how I implement the update_episode part:

```
next_state_value = 0;
cur_value = 0;

for (from terminal_state back to initial_state) {
     state& move = last_state
     cur_value = estimate(move.before_state());
     error = reward + next_state_value - cur_value;
     next_state_value = update(move.before_state(), a*error);
}
```

The most important thing in update_episode i think is to calculate the error.

According to the given formula, the update formula is:

$$V(S) = V(S) + a(r + V(S'') - V(S))$$

So my thought is to iterate through all the moves that have been done and update the move's before_state value thus to set it as the next_state_value for next iteration, which will be:

if there are total 4 moves:

$$[s1, s1'] \rightarrow [s2, s2'] \rightarrow [s3, s3'] \rightarrow [s4, s4']$$

the update will be:

$$V(s3) \leftarrow V(s3) + a(s3.reward + V(s4) - V(s3))$$

$$V(s2) \leftarrow V(s2) + a(s2.reward + V(s3) - V(s2))$$

$$V(s1) \leftarrow V(s1) + a(s1.reward + V(s2) - V(s1))$$

The V(s3) in updating V(s2) is derived from the updated V(s3), and V(s2), V(s1) likewise.

Since the terminal board's value is 0, next_state_value is initialized to 0 at first. Then iterate from state before terminal state till initial state, do update. To get the value, I simply use estimate: estimate the current move's before_state. Then calculate the error according to the given formula and thus set the updated value as the next_state_value, so that in the next iteration it can be used.