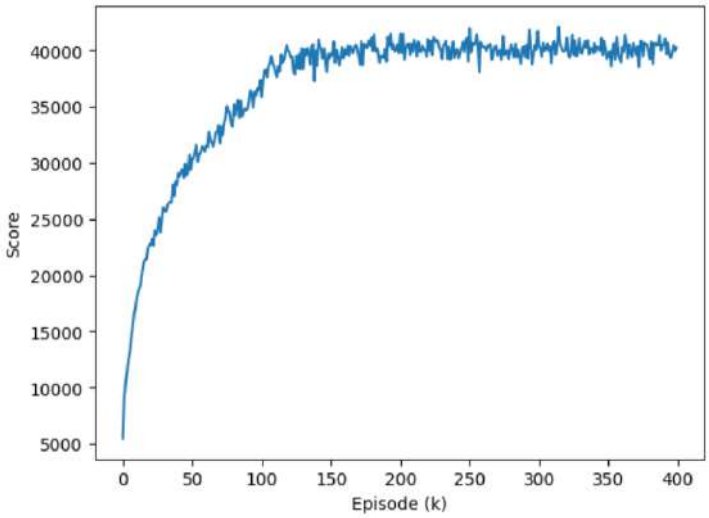
RL HW1

110705013 沈昱宏

- Plot – 400k



I set alpha = 0.1 for the first 200k, alpha = 0.01 for the subsequent 100k, and alpha = 0.001 for the last 100k.

- Implementation & usage of n-tuple network

There are 16 blocks on the board, and each block consist of 16 bossible values. We can't directly compute using the state of whole board due to memory limit, so we have to use n-tuple network to solve the problem. I used 4*6-tuple networks with iso = 8 (the same as the sample code) (iso is for making more blocks considered with the same feature by rotating or mirroring the feature.

```
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 }));
```

The patterns are then stored into class "learning" using a vector of pointer of "feature" class (the content is a derived class "pattern"). In the array of vector <int> named isomorphic under the class "pattern", all the pattern (including its isomorphic) are stored using its index and will be used to (1) search for index, (2) update the weights stored in its base class "feature" (3) estimate the best choice.

The following codes are the implementation of the 3 functions I mentioned. Other explanation can be found in the comment.

```
size_t indexof(const std::vector<int>& patt, const board& b) const {
    // TODO
    // Given the pattern and the board, find the index of the pattern
    // e.g. when 0x4,0x5,0x6 are on the board in place {0,1,2} and the
    // pattern is {0,1,2}, output 0x456
    // For all the element index in pattern, left shift & add.
    size_t temp = 0;
    for(int i = 0; i < patt.size(); i++){
        temp = temp << 4;
        temp += b.at(patt[i]);
    }
    return temp;
    // end TODO
}
```

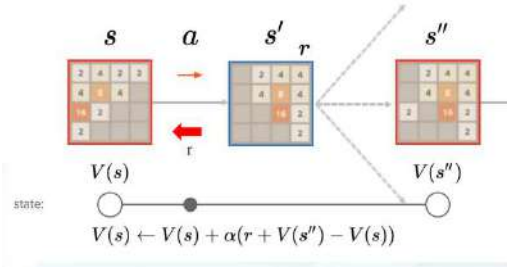
```
virtual float estimate(const board& b) const {
    // TODO
    // sum weights of all the index computed by indexof()
    float output_value = 0;
    for(int i = 0 ; i < iso_last ; i++){
        size_t index = indexof(isomorphic[i], b);
        output_value += weight[index];
    }
    return output_value;

    // end TODO
}
```

```
virtual float update(const board& b, float u) {
    // TODO
    // input u is the update value. I divide u into amount of wieghts that need
    // updating (though it can also be adjusted by modifying alpha)
    // Add the value to the weights that need updating.
    float adjusteach = u / iso_last;
    float output_value = 0;
    for(int i = 0; i < iso_last; i++){
        size_t temp = indexof(isomorphic[i], b);
        weight[temp] += adjusteach;
        output_value += weight[temp];
    }
    return output_value;
    // end TODO
}
```

- TD(0)

  In TD(0), you only have to consider two consecutive steps. The thing you want to learn is the value of V(s). By definition, we want the value of the next state V(s'') equal to the value of the current state V(s) plus the reward r, so we update the V(s) with the TDerror (the difference that we want to update) and alpha.



- Implementation – action selection

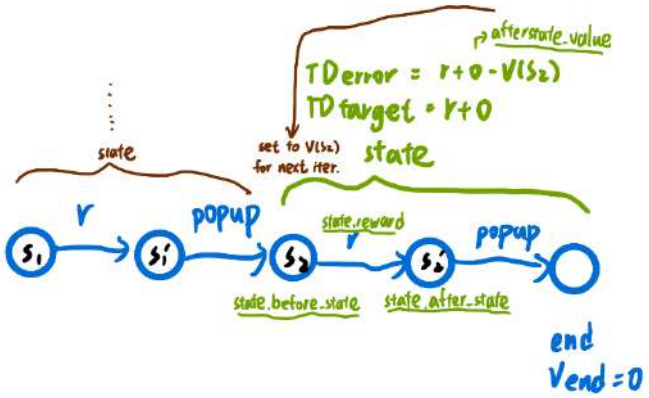
```
state select_best_move(const board& b) const {
    state after[4] = { 0, 1, 2, 3 }; // up, right, down, left
    state* best = after;
    for (state* move = after; move != after + 4; move++) {
        if (move->assign(b)) {
            // TODO
            board temp = move->after_state();
            temp.popup();
            move->set_value(move->reward() + estimate(temp));
            // end TODO
            if (move->value() > best->value())
                best = move;
        }
        else {
            move->set_value(-std::numeric_limits<float>::max());
        }
        debug << "test " << *move;
    }
    return *best;
}
```

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

In the code, I iterate through all the possible moves and select the one that has the biggest value (the value is update by calculating $\Sigma_{s'' \in S''} P(s, a, s'') V(s'')$ + reward. The psuedo code is given by TA for estimating actions. However, there isn't a function that can find $\Sigma_{s'' \in S''} P(s, a, s'') V(s'')$, so I add pop up a tile directly when estimating the value of the future state. When the agent face the same situation multiple times, the mean of the pop up tile will gradually become $\Sigma_{s'' \in S''} P(s, a, s'') V(s'')$, which is the desired value.

- TD backup

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



I update the path from the ending state to the first step. I use a variable “afterstate_value” to save the value of the before_state on the last updated step, which will further be used to compute TD target and TD error.