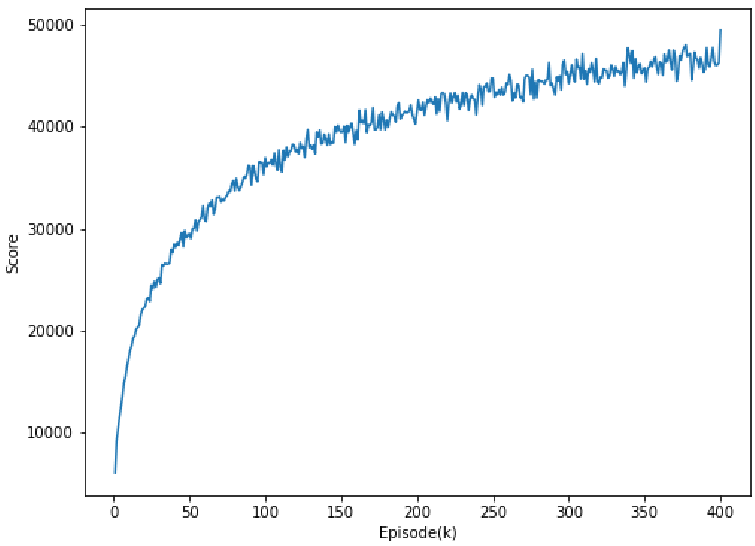
RL lab1

Name: 陳以瑄 Student ID:109705001

- **Plot of scores**



Training episodes: 400k, Learning rate: 0.1

- **Implementation and the usage of *n*-tuple network**

The reason for using n-tuples is that if we were to directly record all the estimates, since each cell can have one of 12 possible values [empty, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048], and there are 16 cells in total, we would need to store a total of 12^{16} estimates, which is not feasible due to memory constraints.

However, if we use an n-tuple network, with k features each composed of n cells, we only need to store 12^n * k estimates. The features I use are as follows:

```
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 }));
```

Therefore, I only need to store a maximum of 12^6 * 4 estimates, which is significantly smaller than 12^{16}.

Additionally, since the feature in different isomorphisms will update the same estimates, the n-tuple network can also solve the issue where the same board position may result in different estimates after being flipped or rotated.

The following is the code I implemented.

(1) Find out the index of the pattern

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

The 'patt' we passed in is the indices of the cells where the feature is located, but we need to update the estimates based on the values stored in these cells. For example, if the value in cell 0 is 0x2, the value in cell 1 is 0x4, and the value in cell 2 is 0x1, then feature {0, 1, 2} needs to update the estimated value of index 0x241. Therefore, we can find the values in the cells using b.at(), then left-shift them and add them together to obtain the index we need.

(2) Estimate the value of a given board

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

Since one kind of feature has 'iso_last' different isomorphisms, when estimating the value of this feature, we need to sun up the estimates of each isomorphism.

(3) Update the value of a given board

```
virtual float update(const board& b, float u) {
    // TODO
    size_t index;
    float u8 =u/8;
    float V=0;
    for (int i = 0; i< iso_last; i++){
        index = indexof(isomorphic[i], b);
        V+= weight[index];
        weight[index]+=u8;
    }
    return V + u;
}
```

Since one kind of feature has 'iso_last' different isomorphisms, when updating, each isomorphism needs to update its corresponding index estimates, and each only needs to update one-'iso_last' of the values. Lastly, return the sum of original estimates plus the value that needs to be updated.

- **Mechanism of TD(0)**

  In TD(0), the '0' means we update the value estimates based on the difference between the current state and the next state. For example, when we take action a in state s, and then undergo a random popup, we receive an immediate reward r and transition to state s''. So we update the value estimates of current state V(s) based on the difference between V(s) and r + V(s''). Since we want a smaller variance, we will set a learning rateαto determine how much should be updated. Therefore the update rule is $V(s) \longleftarrow V(s)+\alpha\cdot[r + V(s'') - V(s)]$.

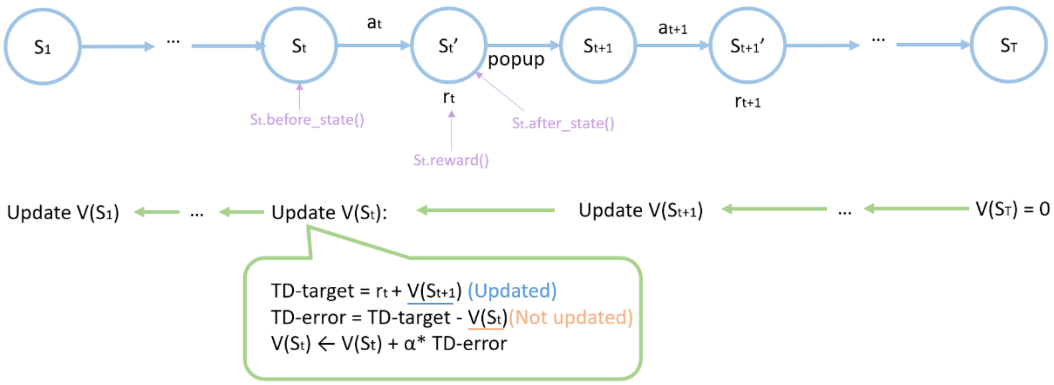
- **Action selection and TD backup diagram**

  (1) Action selection

```
701 state select_best_move(const board& b) const {
702     state after[4] = { 0, 1, 2, 3 }; // up, right, down, left
703     state* best = after;
704     for (state* move = after; move != after + 4; move++) {
705         if (move->assign(b)) {
706             // TODO
707             //estimate return  = R_t+1 + sum P(s,a,s')*V(S_t+1)
708             board af_s = move->after_state();
709             af_s.popup();
710             move->set_value(move->reward() + estimate(af_s));
711 
712             if (move->value() > best->value())
713                 best = move;
714         } else {
715             move->set_value(-std::numeric_limits<float>::max());
716         }
717         debug << "test " << *move;
718     }
719     return *best;
720 }
```

  When selecting the action, we will choose the one that has the highest estimated value, $r + E[V(s'')] = r + \Sigma_{s'' \in S''} P(s, a, s'')V(S'')$. Since we use V(state) rather than V(after-state), we need to take the random popup into account when calculating the expected value (lines 708 to 710). As we know, when repeating a random trial multiple times under the same conditions, the relative frequency of a specific event will converge to a specific value as the number of trials increases. Therefore, after training multiple times, the sample P(s, a, s'') will converge to the population P(s, a, s''), and the expected value we count will become the desired one.

(2) TD backup diagram



Update the estimated value V(s) based on the rule I mentioned before from the end state to the first state. The following is the code I implemented:

```
void update_episode(std::vector<state>& path, float alpha = 0.1) const {
    // TODO
    //V(s)<-V(s)+alpha(r + V(s'')-V(s))
    float V_t1 = 0;
    state& t1 = path.back();
    for(;path.size();path.pop_back()){
        state& t = path.back();
        float TD_target = t.reward()+V_t1;
        float TD_error = TD_target-estimate(t.before_state());
        V_t1 = update(t.before_state(),alpha* TD_error);
    }
}
```

Since we need to use the updated $V(s_{t+1})$ to update $V(s_t)$, I store the updated $V(s_{t+1})$ in ‘V_t1’.