DLP-Lab3

ILE 310553043 You-Pin, Chen

April 1, 2022

1 Introduction

In this assignment, we are asked to fill five TODO regions in 2048 sample code. 2048 [2][3] is a puzzle game with 16 tiles which the only action are up, down, left and right. The objective of the game is to slide numbered tiles on a grid to combine them to create a tile with the number 2048; however, one can continue to play the game after reaching the goal, creating tiles with larger numbers. Therefore, the work attempt to implement reinforcement learning to get the higher score. Reinforcement learning is about learning the optimal behavior in an environment to obtain maximum reward. This optimal behavior is learned through interactions with the environment and observations of how it responds, similar to children exploring the world around them and learning the actions that help them achieve a goal. We chose model-free as type of RL and Temporal-Difference Learning (TD Learning) as learning method with before state policy. In this report, you will see the explanation of question of implementation detail and our experiment results. The code is available in this github [link].

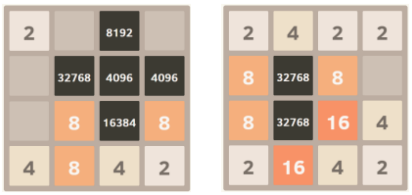
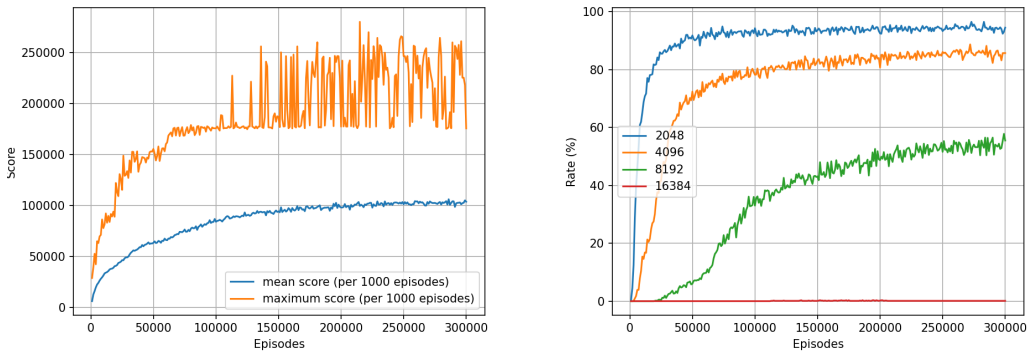


Figure 1: The first 65536 using RL from CGI Lab supervised by Prof. I-Chen, Wu[4].

2 Results

2.1 Best Result Plotting



(a) Maximum/Mean Score curve.

(b) Reaching Rate curve.

Figure 2: Our best result with accuracy 96.7% on reaching rate 2048.

3 Related Work

3.1 n-tuple network

In 2048, the game contains 16 tiles and the goal is to reach 2048 in any of it. The number of probability is 12^{16} which are considerable enough to crash the system. To fix this situation, n-tuple network aka RAM-based neural network is proposed which we define n tiles as feature. Instead of whole tiles, now we only need to consider n tiles and the number of states decrease to 12^n which n is much smaller than 16. Also considering that the feature could be similar, we assemble all isomorphisms as a pattern. By multiplying the pattern by the weight table, the output is so called *board*.

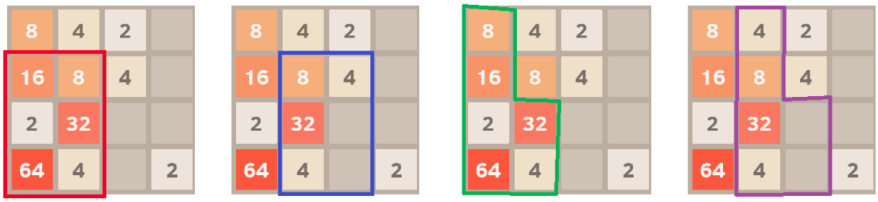


Figure 3: Different types of features with n tiles.

3.2 Mechanism of TD(0)

TD(0) is a simplest temporal-difference learning algorithm so called *TD learning*. Comparing with TD(1) also known as Monte-Carlo Learning, TD(0) just consider about the next one step. Fig.6 shows the flow of TD-learning. The equation is defined in Eq.3.2.1 which we follow this function to update value. In Eq.3.2.2 TD-target and TD-error are defined to update $V(S_t)$ and process TD-backup in the end of episode respectively. α is learning rate and we can ignore γ in this assignment.

$$V(S_t) = V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)) \tag{3.2.1}$$

which

$$\begin{aligned} TD\ target &= R_{t+1} + \gamma V(S_{t+1}) \\ TD\ error &= TD\ target - V(S_t) \end{aligned} \tag{3.2.2}$$

3.3 TD-backup diagram of V(after-state)

In Eq.3.2.1, we understood the mechanism of TD(0), there are 2 types of TD-backup: state and after-state. Fig.4 display the diagram of after-state, instead of updating the value before action, we refresh the board after movement called $V(S')$. The following equation will turn into Eq.3.3.1.

$$V(S') = V(S') + \alpha(R_{next} + \gamma V(S'_{next}) - V(S')) \tag{3.3.1}$$

3.4 Action selection of V(after-state)

After-state policy update the state while finishing the movement. As Fig.4 exhibits, the state will turn to S' after action. In 2048, we are allow to move up, down, left and right which will get different reward. Our goal is to select best movement that will return the maximum value in the result of action reward add enviroment reward $V(S')$.

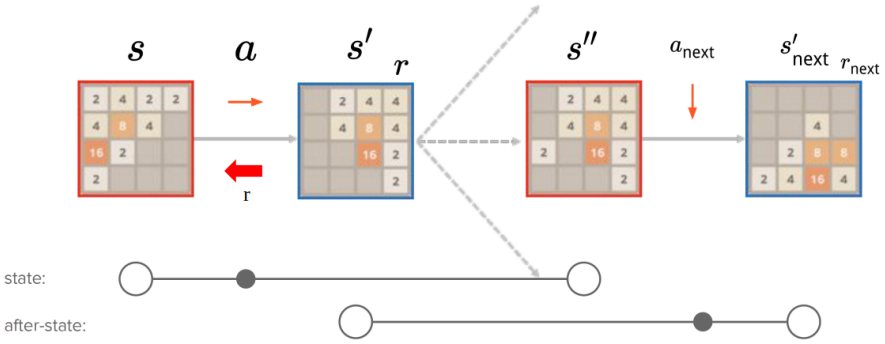


Figure 4: TD-backup diagram: state and after-state.

function EVALUATE(s, a)
  $s^{'}$, $r \leftarrow$ COMPUTE AFTERSTATE(s, a)
  return $r + V(s^{'})$

function LEARN EVALUATION$(s, a, r, s^{'}, s^{''})$
$a_{next} \leftarrow EVALUATE(s^{''}, a^{'})$
$s^{'}_{next}, r_{next} \leftarrow COMPUTE\ AFTERSTATE(s^{''}, a_{next})$
$V(s^{'}) \leftarrow V(s^{'}) + \alpha(r_{next} + V(s^{'}_{next}) - V(s^{'}))$

Figure 5: Pseudo code of after-state.

3.5 TD-backup diagram of V(state)

Different with after-state, the value we desire to update is before action, which is $V(S)$ in Fig.4. The equation is in Eq.3.5.1 that we estimate the value after the environment give us a reward. By subtracting $V(S)$ from environment reward, we add action reward as TD-error and multiply the learning rate as the adjustment range.

$$V(S) = V(S) + \alpha(R + \gamma V(S'') - V(S)) \tag{3.5.1}$$

3.6 Action selection of V(state)

In Fig.4, when S' state turn into S'' state, there will be many possible since each empty tile will have a opportunity to create 2 or 4. To evaluate the value of $V(S'')$, we must calculate the expected value which means we have to simulate every situation if 2 or 4 will be in what position. In this assignment, the probability of the appearance of 2 and 4 are 90% and 10% respectively.

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

function LEARN EVALUATION$(s, a, r, s^{'}, s^{''})$
$V(s) \leftarrow V(s) + \alpha(r + V(s'') - V(s))$

Figure 6: Pseudo code of before-state.

4 Implementation

4.1 Estimate

In the previous section, we had mentioned that to get the total reward, we need to evaluate the environment value. Therefore, this function estimates the value of given board. We had also mentioned that each pattern will have more than one isomorphic tuple, so we used the value of isomorphic tuple as a index to find the weight and added them up.

```
/**
 * estimate the value of a given board
 */
virtual float estimate(const board& b) const {
   float value = 0;
   for (int i = 0; i < iso_last; i++) {
      size_t index = indexof(isomorphic[i], b);
      value += operator[](index);
   }
   return value;
}
```

4.2 Update

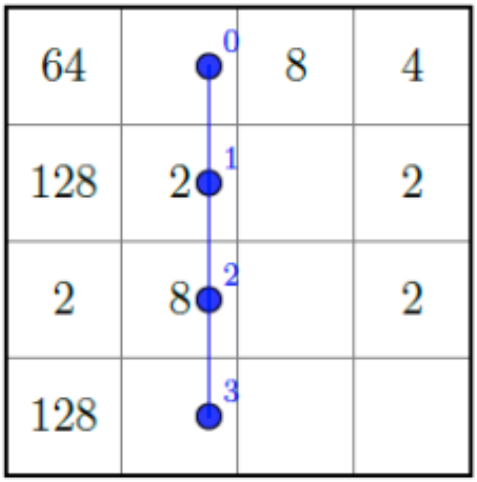
After we estimate the expected value, we will update the value of a given board which we had mentioned in Eq.3.5.1. Same as section 4.1, there will be more than one isomorphic tuple we have to consider.

```
/**
 * update the value of a given board, and return its updated value
 */
virtual float update(const board& b, float u) {
   float u_split = u / iso_last;
   float value = 0;
   for (int i = 0; i < iso_last; i++) {
      size_t index = indexof(isomorphic[i], b);
      operator[](index) += u_split;
      value += operator[](index);
   }
   return value;
}
```

4.3 Index of

Return the index which used to map into weight table. The input parameter *patt* is the pattern we defined in the beginning and b is the present board; i in for loop is the number of tuple which is same as n in n-tuple. By bitwise operator OR, we are allowed to concatenate value in each tile. For example, in Fig.7, the pattern is $\langle 1, 5, 9, 13\rangle$ which the value is $\langle 0, 2, 8, 0\rangle$. We run each index step by step that first comes to $patt[1]$ and variable *index* is updated from 0 to 0000. Then, the value of $patt[5]$ is 2 which now updated *index* to 0010 0000. Therefore, in this example, the final result will be 0000 1000 0010 0000.

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



0123	weight
0000	3.04
0001	−3.90
0002	−2.14
⋮	⋮
0010	5.89
⋮	⋮
0130	**-2.01**
⋮	⋮

Figure 7: Encode the pattern to map into weight table.

4.4 Select best move

This section implements the equation in Eq.3.5.1 which to process TD-backup. Since there are many states in $V(S'')$, we should concern each possible. First, we had to consider about all movement which is up, down, left and right. Second, during for loop, $move \rightarrow after\_state()$ is the state after movement which also represents $V(S')$ in Fig.4. We checked if there is an empty tile in the present board and simulate the value (which means 2 or 4) in each of it. The final result will be the expected value of this step, then we set the value in this movement and compared if this one had maximum value which we said was the best move.

```
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)) {
         int value = 0;
         int empty_tile = 0;

         for(int i = 0; i < 16; i++) {
            if(move->after_state().at(i) == 0) {
               board tmp = move->after_state();
               tmp.set(i, 1); value += 0.9 * estimate(tmp);
               tmp.set(i, 2); value += 0.1 * estimate(tmp);
               empty_tile++;
            }
         }

         value = value / empty_tile;
         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;
}
```

4.5 Update episode

After we selected the best move, we will save it in the *path* which is the sequence of states in a episode. Since while loop is broken down, it means that we finished a game. Meanwhile, we can process the learn evaluation part. The last entry in *path* is the final state which we can called *path.back*() function to get it. The move we got here is the state after movement, so if we want to estimate $V(S)$, we need to called $move \rightarrow before\_state()$ to reverse the state. Variable *error* represents TD-error that we use it update our network.

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

5 Discussion

5.1 Different learning rate

We applied four 6-tuple features as our network with learning rate α=0.1. Although in *update* function in *learning* object had already split the rate into each feature, which we divided learning rate by the number of feature, that is said to be the best learning rate for training, the loss stopped in the middle of whole learning. We suspected that if our gradient stopped in the local minimum, so we attempt to decrease our rate again to 0.025. The result exhibits in Table 1, each of them was with $300k$ episodes, the reaching rate decreased obviously in every part. We kept training the case with $lr = 0.025$ after $300k$ episodes and the result shown in Table 2. The accuracy improved to 87.3%. Since we did not have enough, we believe that the accuracy will improve if we keep moving on.

Reaching rate	lr=0.1	lr=0.025
256	100%	99.7%
512	99.6%	99.1%
1024	98.7%	95.1%
2048	90.5%	81.9%
4096	76.5%	51.1%
8192	34.5%	2.3%
Mean	85468.3	50415.1
Maximum	191196	143792

Table 1: Performance Comparison

Episodes	300k	560k
2048	81.9%	87.3%

Table 2: Performance Comparison in different episodes.

5.2 Multi n-tuple networks

5.2.1 4 x 6-tuple networks

In last section, we mentioned that we applied four 6-tuple networks as our model. The combination is proposed in [4] which the author changed 4-tuples which are proposed in [3] to 6-tuples in knife-shaped. The new 6-tuples cover all the original 4-tuples while still not covering the 6-tuples in 2×3. Fig.12 illustrates the result of the pattern, we achieved 90.5% in 2048 reaching rate after $300k$ episodes.

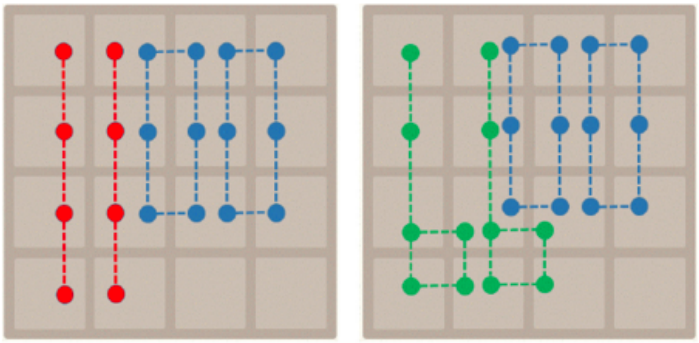


Figure 8: (a) Tuples used in [3] and (b) tuples used in [4].

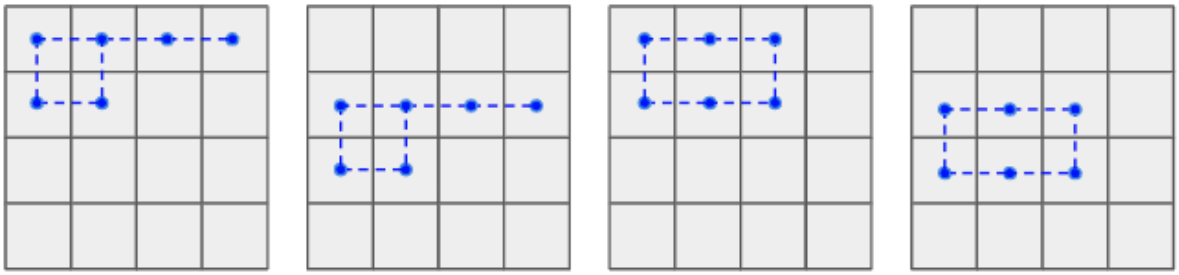
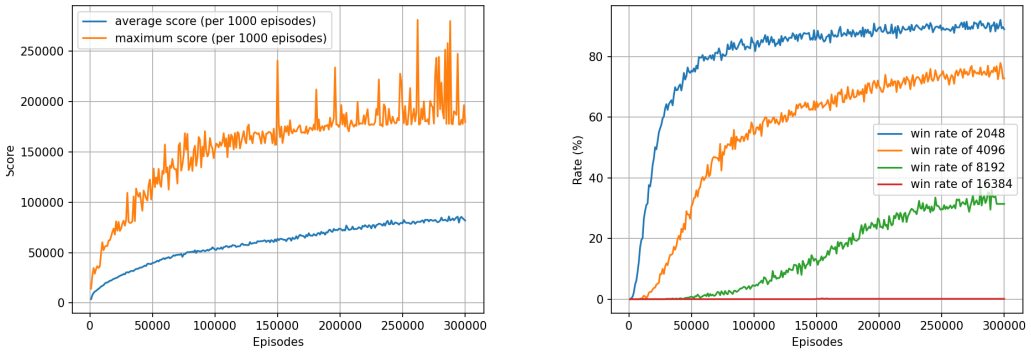


Figure 9: Pattern we used in the experiment which is isomerism in Fig.8 (b).



(a) Maximum/Mean Score curve.

(b) Reaching Rate curve.

Figure 10: Our result with 4x6-tuples, $\alpha = 0.1$, $300k$ episodes.

5.2.2 Combine different shape of feature

Since in last experiment, the inference of the weights just match the baseline. We rethought about the situation that if not change 4-tuples to 6-tuples instead of combination. Therefore, we proposed the new pattern in Fig.11 which added two other features $\langle 0, 1, 5, 6, 7\rangle$ and $\langle 4, 5, 9, 10, 11\rangle$. By this eight features, we reached the new best result which reaching rate in 2048 increased to 96.5% and maximum score achieved 279864 after $300k$ episodes of learning.

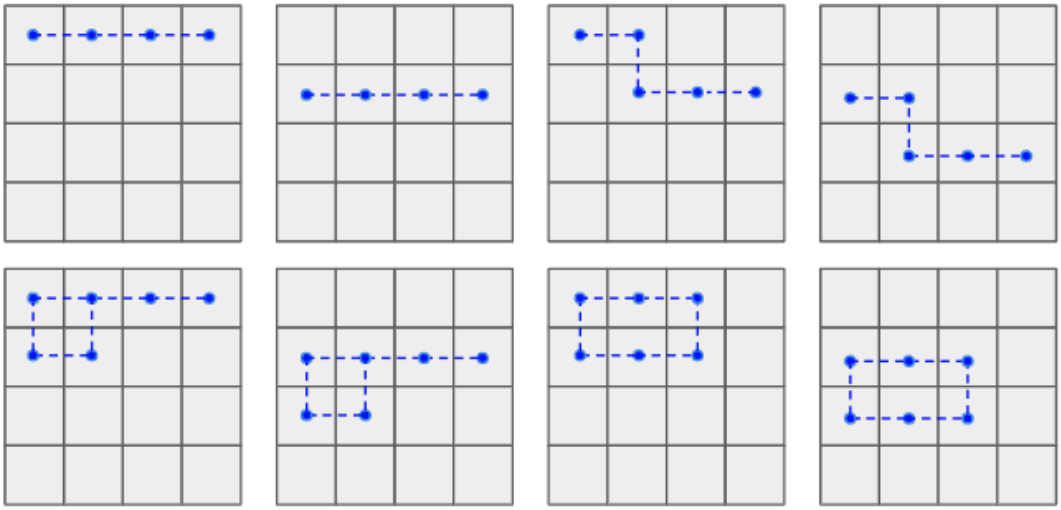
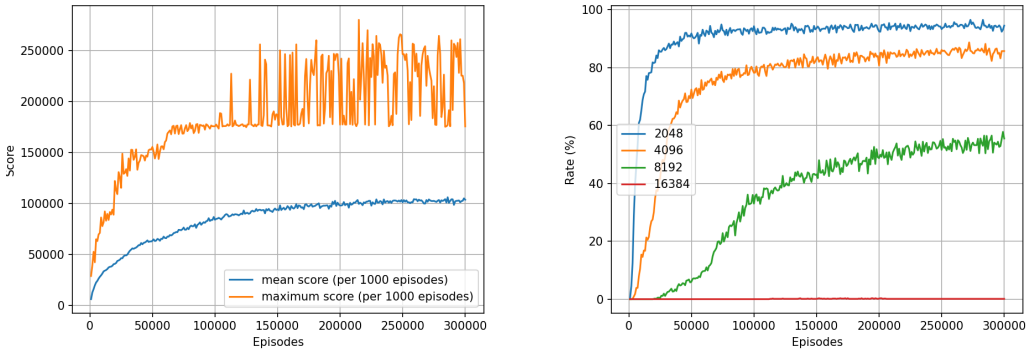


Figure 11: Four different type of feature.



(a) Maximum/Mean Score curve.

(b) Reaching Rate curve.

Figure 12: Our result with multi tuples, $\alpha = 0.1$, $300k$ episodes.

5.2.3 Challenge of 7-tuple network

Still, we kept trying to push our accuracy up by changing 6-tuples to 7-tuples which is proposed in [1] that is published in IEEE, 2021. The author is same in [4] whom is also the teacher for this course. They got best performance by using eight 7-tuples. However, our implementation failed because of the limitation of hardware since the 2048 programs use CPU to execute and We do not have enough memory for the execution (6-tuple need $64MB$ per feature which 7-tuple increase to $1G$ exponentially per feature).

References

[1] Hung Guei, Lung-Pin Chen, and I-Chen Wu. Optimistic temporal difference learning for 2048. *IEEE Transactions on Games*, 2021.

[2] Kazuto Oka and Kiminori Matsuzaki. Systematic selection of n-tuple networks for 2048. In *International Conference on Computers and Games*, pages 81–92. Springer, 2016.

[3] Marcin Szubert and Wojciech Jaśkowski. Temporal difference learning of n-tuple networks for the game 2048. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[4] Kun-Hao Yeh, I-Chen Wu, Chu-Hsuan Hsueh, Chia-Chuan Chang, Chao-Chin Liang, and Han Chiang. Multistage temporal difference learning for 2048-like games. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(4):369–380, 2016.