Solving 2048 with RL

Yevhen (Jake) Horban

Abstract

This paper overviews state of the art methods developed between 2015 and 2021 to solve 2048 with reinforcement learning. There are quite a few papers using deep neural networks but they do not achieve results that nearly as good.

Problem

Description

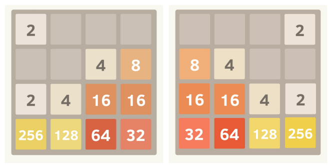
This paper attempts to find a reinforcement learning method that obtains the highest score in a video game 2048. In this game an agent slides the tiles in one of four directions until there are no more legal moves left. The hardest part of the game and gap between scores occurs when an agent is about to complete a new large tile with the chain of smaller tiles. This causes a *survival phenomenon* because if one succeeds in completing a high valued tile the board opens up and an agent can continue playing for many more steps.

State

The game begins with two tiles randomly placed on a 4×4 grid. $p(\text{2-tile}) = \frac{9}{10}$ and $p(\text{4-tile}) = \frac{1}{10}$. The value of the tiles can be modelled by $v = 2^n$ where $n \in \{0, 1, \ldots 17\}$. The empty tile is represented by $v = 2^0 = 1$. The highest possible tile theoretically is $v = 2^{17} = 131072$ because there is no more space for the smaller tiles that would allow to collect the next large tile.

$|S| = 18^{4\times 4} \approx 1.2 \cdot 10^{20}$ which is about 2.8 times the number of all possible $3\times3\times3$ Rubik's cube configurations. However, some of these states are unreachable because it is impossible to have a fully empty board or a board full of 2-tiles.

There is an opportunity to make certain states equivalent to reduce dimensionality similar to the game of Tic-Tac-Toe. For example two states bellow are equivalent and the optimal moves are mirror reflections. Same logic applies to the vertical and diagonal reflections leading us to an 8-fold dimensinality reduction.



Action

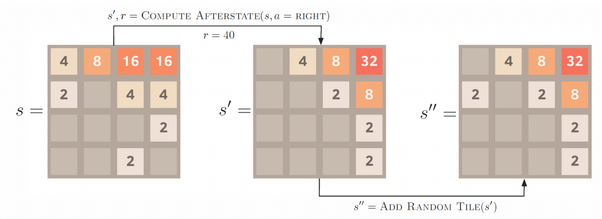

$s', r =$ COMPUTE AFTERSTATE$(s, a =$ RIGHT$)$
$r = 40$
$s =$
$s' =$
$s'' =$
$s'' =$ ADD RANDOM TILE(s')

Figure 2: A two-step state transition occurring after taking the action $a =$ RIGHT in the state s.

There are only for actions that correspond with sliding all tiles left, right, up or down: $A = \{L, R, U, D\}$. During the sliding of the tiles all same-value pairs closest to the direction of the slide get merged. A valid move is if the board changes after the move. After a valid move another additional tile spawns in a randomly chosen empty cell. The game ends when the board runs out of empty tiles and there are no legal moves to make.

An important distinction has to be made in the state transition process. There are two parts:

1. a deterministic step that only slides and merges tiles in the direction of the action, an *after-state*
2. a stochastic step that spawns a new tile, a *next state*

One can create a list of all possible transitions and their probabilities which means that this game can be modelled as a Markov Decision Process.

Reward

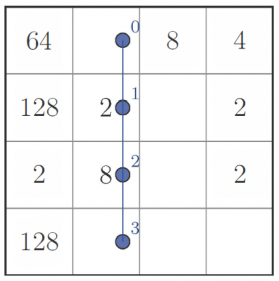
For this project we are going to use the reward that is provided by the environment. The reward is equal to the sum of all numbers of the merged tiles. For example, if an action results in a merge of two 2-tiles and two 8-tiles, reward $= (2 + 2) + (8 + 8) = 20$

Simulator

I used gym implementation of 2048 and added a few modifications in this pull request. I had to fix bugs, adjust the state to be represented as powers of 2, added the render function and useful helper functions.

Methods

n-Tuple Networks



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

Figure 7: A straight 4-tuple on a *2048* board. According to the values in its lookup table, for the given board state it returns −2.01, since the empty square is encoded as 0, the square containing 2 as 1, and the square containing 8 as 3.

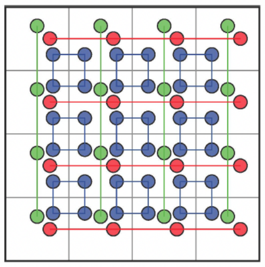


Figure 8: The n-tuple network consisting of all possible horizontal and vertical straight 4-tuples (red and green, respectively), and all possible 2×2 square tuples (blue).

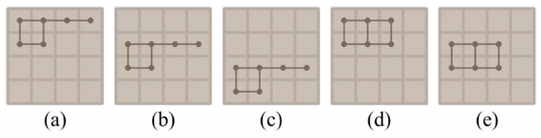


Fig. 2. (a), (b), (d), and (e) 4×6-tuple network proposed by Yeh *et al.* [3]. (a)–(e) 5×6-tuple network used by Jaśkowski [5].

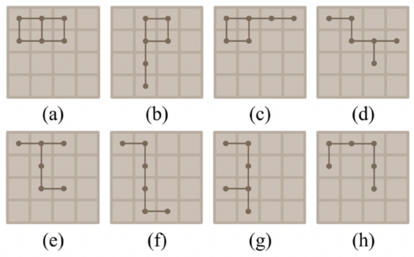


Fig. 3. (a)–(h) 8×6-tuple network proposed by Matsuzaki [8].

$$V(s) = \sum_{i=1}^{m} LUT_i[\phi_i(s)] \tag{1}$$

Estimating the state value on the whole state is impractical so instead a N-Tuple Network function approximator is applied. Based on the cells of the n-tuple we construct a a set of feature weights and stored and updated it in a look up table. The state value estimation is then calculated by retrieving the feature vector and summing the weights.

I have followed the example of M. Szubert and W. Jaskowski with constructing a 4-tuple with the maximum tile value of $2^{15} = 32678$ to limit the number of weights in the network. Similar to Figure 8, I have used 4 horizontal and vertical tuples, and 9 square tuples which resulted in $17 \times 15^4 = 860,625$ weights.

Multi-stage TD Learning

```
1:  function PLAY GAME
2:      $score \leftarrow 0$
3:      $s \leftarrow$ INITIALIZE GAME STATE
4:      while $\neg$IS TERMINAL STATE$(s)$ do
5:          $a \leftarrow \arg\max_{a' \in A(s)}$ EVALUATE$(s, a')$
6:          $r, s', s'' \leftarrow$ MAKE MOVE$(s, a)$
7:          if LEARNING ENABLED then
8:              LEARN EVALUATION$(s, a, r, s', s'')$
9:          $score \leftarrow score + r$
10:         $s \leftarrow s''$
11:     return $score$
12:
13: function MAKE MOVE$(s, a)$
14:     $s', r \leftarrow$ COMPUTE AFTERSTATE$(s, a)$
15:     $s'' \leftarrow$ ADD RANDOM TILE$(s')$
16:     return $(r, s', s'')$
```

Figure 3: A pseudocode of a game engine with moves selected according to the evaluation function. If learning is enabled, the evaluation function is adjusted after each move.

```
1: function EVALUATE$(s, a)$
2:     return $V_a(s)$
3:
4: function LEARN EVALUATION$(s, a, r, s', s'')$
5:     $v_{next} \leftarrow \max_{a' \in A(s'')} V_{a'}(s'')$
6:     $V_a(s) \leftarrow V_a(s) + \alpha(r + v_{next} - V_a(s))$
```

Figure 4: The *action evaluation function* and Q-LEARNING.

TD learning is intrinsically to train and learn to obtain average (or expected) scores as high as possible based on adjusting state values from the subsequent evaluations. However, research has shown that it preforms poorly on this task. Instead, we can use multi-stage TD learning, where the learning process is divided into multiple stages, each of which has its own learning agent and sub-goal, e.g., reaching 8192-tiles or 16384-tiles.

Optimistic Initialization

Some works have shown that exploration techniques, such as ϵ-greedy and softmax were not successful for this environment because it is not likely that an agent selects an unexplored state or action when revisiting it. Instead, we want to use Optimistic Initialization, so that large initial state value are reduced once the corresponding states are visited until all the actions are sufficiently explored.

TC Learning

$$\delta_t = r_{t+1} + V(s'_{t+1}) - V(s'_t) \tag{2}$$

TC learning is a TD variant with adaptive learning rates. Instead of adjusting the learning rate directly, this method introduces new parameters weights β, E and A.

$$\theta_i \leftarrow \theta_i + \alpha\beta_i\delta_t \tag{3}$$

$$\beta_i = \begin{cases} |E_i|/A_i \text{ , if } A_i \neq 0 \\ 1 \text{ , otherwise} \end{cases} \tag{4}$$

β_i represents the coherence of θ_i, and is calculated from two parameters E_i and A_i for each weight. Both E_i and A_i are initialized with 0 and adjusted by

$$E_i \leftarrow E_i + \delta_t \text{ and } A_i \leftarrow A_i + |\delta_t| \tag{5}$$

It is an effective method for this problem because it reduces weight adjustments to automatically accelerate network convergence, however it introduces a triple memory overhead.

Expectimax Search

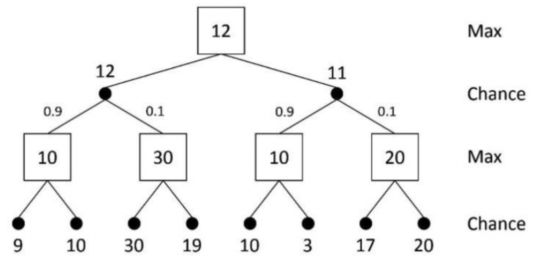


Fig. 2. An expectimax search tree.

Like in most games, the search tree looks as the max node and the expected value of its children. Features used for Expectimax Search are monotonicity of a board, number of empty tiles, number of mergeable tiles, but there can be more.

To speed up the search we should use the Transposition Table to avoid searching redundant states. Common implementation is a Zorbist hashing algorithm based on doing an exclusive or operation on the 16 keys of the grid cells. As described in the state section, we can also include board reflections to reduce the number of leaves needed to search.

The board is considered to have high *monotonicity* if the values are either non-increasing along all rows or non-decreasing along all rows and if the values are non-increasing along all columns or non-increasing along all columns, with the highest value being in one of the four corners. Algorithm 2 describes a simplified version of the process used to score a single game state's monotonicity. Rotating 90 degrees clockwise is to transform all coordinates such that such that

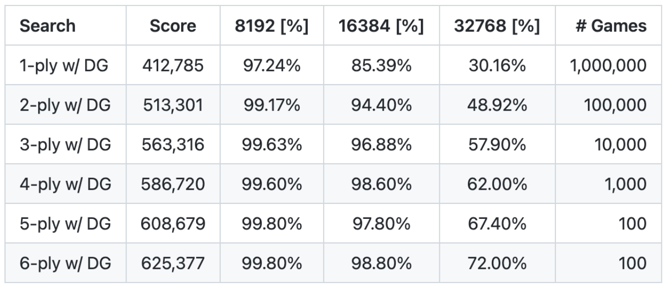
$$G_r[3-y][x] \leftarrow G[x][y] \tag{6}$$

for all x and y. In the code, the board is only rotated once, with two corners being checked per rotation (the second corner is checked by substituting the $\geq$ on lines 7 and 14 with $\leq$).

Algorithm 2 Scoring a game's monotonicity

```
Input: $G$ - a game
Output: The game's monotonicity score
 1: procedure ScoreMonotonicity($G$)
 2:     $best \leftarrow -1$
 3:     for $i \leftarrow 1, 4$ do
 4:         $current \leftarrow 0$
 5:         for $row \leftarrow 0, 3$ do
 6:             for $col \leftarrow 0, 2$ do
 7:                 if $G[row][col] \geq G[row][col+1]$ then
 8:                     $current \leftarrow current + 1$
 9:                 end if
10:             end for
11:         end for
12:         for $col \leftarrow 0, 3$ do
13:             for $row \leftarrow 0, 2$ do
14:                 if $G[row][col] \geq G[row+1][col]$ then
15:                     $current \leftarrow current + 1$
16:                 end if
17:             end for
18:         end for
19:         if $current > best$ then
20:             $best \leftarrow current$
21:         end if
22:         Rotate the board 90 degrees clockwise
23:     end for
24:     return $best$
25: end procedure
```

Results

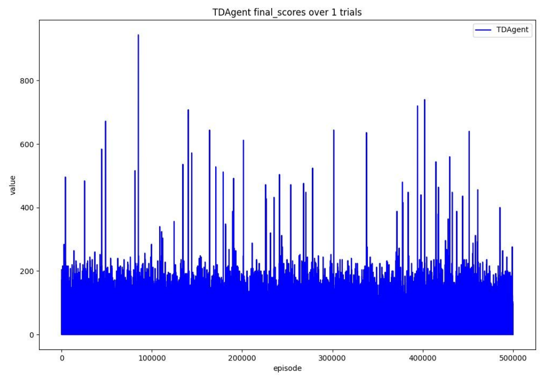


Search	Score	8192 [%]	16384 [%]	32768 [%]	# Games
1-ply w/ DG	412,785	97.24%	85.39%	30.16%	1,000,000
2-ply w/ DG	513,301	99.17%	94.40%	48.92%	100,000
3-ply w/ DG	563,316	99.63%	96.88%	57.90%	10,000
4-ply w/ DG	586,720	99.60%	98.60%	62.00%	1,000
5-ply w/ DG	608,679	99.80%	97.80%	67.40%	100
6-ply w/ DG	625,377	99.80%	98.80%	72.00%	100

In addition, for sufficiently large tests, 65536-tiles are reached at a rate of 0.02%.

The results above are from https://github.com/moporgic/TDL2048, a state of the art optimistic TD+TC (OTD+TC) learning for training and tile-downgrading (DG) expectimax search for testing.

As described in the multi-stage TD learning section, avarage total score or number of steps in the game are not necesarily the best metric for evaluating the algorithms. Instead we want to look at the rate at which the high-valued tiles are reached. One can notice that if we increase the depth of the search tree (n-ply) then we get consistently better results, but it becomes more computationally expensive to run simulations so the number of games in the table is smaller.



My results in the graph above show that my agent is not improving over 500K trials, which suggests that I have a bug in my implementation.

Future work

So far, I only implemented a regular TD Learning algorithm with 17 4-tuple networks. Firstly, I want to find a bug that prevents my agent from learning and add additional learning methods described in this paper. MSTD might be the next easiest method to implement.

With a more successful implementation I want to:

1. check if the algorithm uses the corner for the highest tile and cascades the smaller tiles in a ladder pattern
2. check if the algorithm risks to bring the highest tile back into the corner in case it was forced out of it
3. how often the algorithm looses the game as the result of a forced move - for example when you are collecting a high valued tile in the bottom left and need to make an ”up” action that ruins the tile chain

References

Guei, H.; Chen, L.-P.; and Wu, I.-C. 2022. Optimistic Temporal Difference Learning for 2048. *IEEE Transactions on Games*, 14(3): 478–487.

Kohler, I.; Migler, T.; and Khosmood, F. 2019. Composition of basic heuristics for the game 2048.

Szubert, M.; and Jaskowski, W. 2014. Temporal difference learning of N-tuple networks for the game 2048.

Yeh, K.-H.; Wu, I.-C.; Hsueh, C.-H.; Chang, C.-C.; Liang, C.-C.; and Chiang, H. 2016. Multi-Stage Temporal Difference Learning for 2048-like Games.

Appendix

Code https://github.com/h0rban/2048-rl
Lecture slides https://docs.google.com/presentation/d/1lhgJfJ3f-rJLfIHltV4ai3gLgtf5U6HSHPXfUvBGsnk/edit?usp=sharing