What's in a Game: Solving 2048 with Reinforcement Learning

Nathaniel Goenawan
Stanford University
450 Jane Stanford Way
nathgoh@stanford.edu

Simon Tao
Stanford University
450 Jane Stanford Way
stao18@stanford.edu

Katherine Wu
Stanford University
450 Jane Stanford Way
kjwu00@stanford.edu

Abstract

Game AI agents are a common sub-field of artificial intelligence research. One of the most well known agents is Alpha Go Zero which achieved superhuman performance in the game Go without the aid of human supervision. In this project, we implemented an agent which uses the Monte Carlo Tree Search algorithm to play the game 2048. We compared three different value functions: the merge score of the game (the sum of all merged tiles throughout the game), the largest tile on the final board of the game (with ties broken by the sum of the tiles on the final board), and the sum of all of the tiles on the final board of the game. We evaluated agents using the percentage of games it achieves the 2048 tile in, the average sum of the final board for the games it plays, and the average merge score for the games it plays. We found that using the sum of the tile on the final board of the game as the evaluation policy allows us to obtain the largest percentage of games where the tile 2048 is achieved, largest average sum of the final board, and largest average merge score.

1. Introduction

2048 is a single-player game played on a 4x4 grid. In each turn, two events occur: 1) the user must decide how to slide the tiles, and 2) if the grid changes after the tiles slide, a random tile with the number 2 or 4 spawns in a free space on the grid. When the player presses an arrow button, the tiles slide as far as possible to the corresponding side of the board until it hits either the side of the board or another tile. If the tile hits a tile of the same value, they combine into one tile with double the original value. This combined tile cannot combine with other tiles in the same move. A merge score is kept track based on the value of a combined tile and will increase with every combined tile that is made throughout the duration of the game.

Overall, the objectives of the game are to maximize the tile with the largest number on it (or achieve 2048), maximize the sum of the tiles on the final board, and maximize the merge score. The game ends when all free spaces are used up and no two tiles can be merged. In our project, our goal is to develop a bot which is able to play 2048 well through reinforcement learning.

The source of randomness in this game is the random placement of a new tile on a available space on the board after each turn and whether the tile value is a 2 or a 4. The bot would need to take into account this uncertainty and play with a strategy to maximize the largest tile achieved, the sum of the tiles on the final board, and the merge score.

2. Related Work

Several computer players have been developed for 2048 over the past years. Early players employed a depth-limited tree search algorithms like minimax or expectimax alongside an evaluation function [2]. Later, more advanced approaches came around like that one employed by Szubert and Jaśkowski who utilized N-tuple networks as the evaluation functions and apply a reinforcement learning method to adjust the weights of the N-tuple networks [4].

Our work for this paper involves implementing a Monte-Carlo Tree Search. MCTS is an online, heuristic search algorithm that finds the optimal decision by running simulations on the given state and simply choosing the action that maximize the estimate of the action-value $Q(s, a)$ [1]. This method of reinforcement learning has achieved relative success in game AIs. Most notably, MCTS has been used in creating Google's AlphaGo and AlphaGo Zero. In a similar approach to AlphaGo Zero, where its neural networks learns by generating its own training data through simulations of playing Go, we will play simulations of 2048 to determine the most optimal direction move for our bot to play [3].

3. Methods

We are able to model the game 2048 as a Markov decision process. 2048 has discrete state and action spaces. Every valid state is a four by four grid and each cell on the board is either zero or a power of two. The board is initially empty except for a tile that can have value 2 or 4. The ac-

tion space is up, down, left, and right. Choosing a direction will push all the tiles in that direction, and any two tiles that have the same number and are adjacent to each other in the direction chosen, will be combined into their sum. As mentioned in the introduction, the game ends when the board is full and no more moves can be made, meaning that no more merges can be made in any direction.

In order to do so, we created a 2048 game class. We keep track of the game board using a two-dimensional array along with the merge score of the game. We implemented functions to check whether or not a game is over, complete a given move given a board, get the sum of the tiles on the board, get the largest tile on the board, and get the merge score of the game.

3.1. Evaluation Methods

When evaluating an agent, we run 100 trials (games). We consider three different factors to mimic what people may want to achieve when playing 2048.

Our first factor is the percentage of games which achieved a maximum tile number that is equal to or more than 2048: given that the name of the game 2048 is 2048, many people consider achieving a 2048 tile as winning the game. Our second factor is the sum of the squares on the final board, as players may evaluate their final board as a whole and this incorporates all of the information in the final state of the board. Our third factor is the average merge score for the 100 games, where the merge score is the sum of the values for all squares that were merged together throughout the game (we use a discount factor of 1).

An agent which achieves a higher 2048 square percentage, a higher average sum of final tiles, and a higher average merge score is considered better performing.

3.2. Baseline

Our baseline model, which we call **Random**, is a uniform random policy that randomly chooses between up, down, left, and right.

3.3. Monte Carlo Tree Search

We choose to use an online planning approach to solve 2048, as 2048 has an extremely large state space: each of the 16 cells on the board can take on any either zero or any power of two. If we consider the limited state space where the largest number is less than or equal to 2048, the state space is $16^{12} \approx 2.815 \cdot 10^{14}$.

More specifically, we chose to create a Monte Carlo Tree Search solver, which we call **MCTS**. Usually, when performing MCTS, we would take the action that maximizes the upper confidence bound defined by $Q(s,a) + c\sqrt{\frac{\log N(s)}{N(s,a)}}$. Here, $Q(s,a)$ is the action-value function of taking action a from state s, which we will define as the state-action pair (s,a). $N(s)$ is the number of times we have reached a particular state s, and $N(s,a)$ is the number of times we have chosen to take action a from state s. Finally, c is an exploration parameter that scales the value of unexplored actions, and higher values of c are assigned to actions that we haven't tried as frequently. However, for the game of 2048, we can perform several modifications to this update function. Firstly, we can notice that since 2048 has a very large state space, it is extremely unlikely that we revisit a state space that we have previously seen before. This is supported by the fact that, since we generate a new tile every time a move is performed and we constantly get larger tiles as numbers are merged, the total sum on the board keeps increasing, so we can't revisit a previous state with a smaller sum of tiles on the board. In our algorithm, at each state, we perform an equal amount of exploration for each action, randomizing the rollouts. Consequently, we can see that the ratio of $N(s)$ to $N(s,a)$ would stay constant. Secondly, every time we make a move, we are performing an unexplored action from an unexplored state. Thus, adding the same exploration reward for every action we take is equivalent to not adding the exploration reward across all actions. As a result, at every step, we take the action that solely maximizes the action-value function $Q(s,a)$, which we describe in more detail in section 3.3.1.

Thus, to play a game using the MCTS, we do an iteration of choosing and making a move while the game has not ended (the board has empty squares or at least two squares can still be merged). For each of the four available moves, we first make a deep copy of the board and complete the chosen move on the copy of the board. If making the move does not change the board, we disregard that move. If the move does change the board, we then perform many iterations of rollout. For each rollout, we continue to choose random moves until the game is complete. We choose the action which has the highest average value for the rollouts given the chosen evaluation method. Algorithms 1 and 2 provide the pseudo-code for this approach.

3.3.1 MCTS Value Functions

When playing the game, we considered three different types of value functions used to determine which action to take at each step in the game.

The first value function we used was the merge score at the end of the game, which we also discuss when evaluating the performance of an agent.

The second value function we used was the largest board sum at the end of the game, as this metric captures information about every number on the board at the end of the game. As many 2048 players are not aware of the merge score the 2048 provides, we felt that this more closely represents the judgement of a human 2048 player at the end of

Evaluation Policy	N.O. Rollouts	N.O. Trials
Merge Score	50	100
Merge Score	100	100
Merge Score	250	100
Merge Score	500	100
Total Sum	50	100
Total Sum	100	100
Total Sum	250	100
Total Sum	500	100
Largest Number, Total Sum	50	100
Largest Number, Total Sum	100	100
Largest Number, Total Sum	250	100
Largest Number, Total Sum	500	100

Table 1. *Monte Carlo Tree Search Experiments*

Algorithm 1: MCTS agent

while *a move can still be made* **do**
  bestMove := None;
  bestValue := None;
  for *move* $\in (up, down, left, right)$ **do**
    moveSumValue := 0;
    make a copy of the game;
    make the move on the copy of the board;
    if *copy of board is identical to the original board* **then**
      continue;
    end
    for *i* $\in [1, ..., Number\ of\ Roll\ Outs]$ **do**
      moveSumValue += random(copy of the game);
    end
    if *sumValue* $>$ *bestValue* **then**
      bestValue := moveSumValue;
      bestMove := move;
    end
  end
  make the best move on the board;
end

Algorithm 2: Complete a random run given a starting 2048 board.

make a copy of the board passed in;
while *a move can still be made* **do**
  choose a random move;
  make the random move on the copy of the board;
end
return the value of the copy of the board for the appropriate method of evaluation;

the game.

The third value function we used was the largest tile at the end of the game, with ties being broken with the sum of the tiles on the board at the end of the game. This uses the idea that one metric some 2048 players use is the highest number (aligning with one of our evaluation metrics, the 2048 square percentage), and gives a way to break ties.

3.3.2 MCTS Experiments

We were interested in comparing the performance of our MCTS when we use of our three different value functions. We were also interested in determining how the number of rollouts affects the performance of the agent, and choose to test agents which had 50 rollouts, 100 rollouts, 250 rollouts, and 500 rollouts per action for each move. We tested an agent for each combination of value function and number of rollouts for a total of 12 Monte Carlo agents, each of which was tested 100 times, as summarized in Table 1.

3.4. Code

If you are interested in more details for our method, code for our implementations can be found at https://github.com/nathgoh/CS238-2048.git.

4. Results

4.1. Random Policy

When evaluating our models, we first ran experiments on the random policy. We simulated 100 full games to find the average largest tile, sum of the final board, and merge score. The random policy never reached the 2048 tile, had a really low average sum for the final board of 262.8, and had a really low average merge sum of only 942.72. Table

Evaluation Function	2048%	Avg Sum of Final Board	Avg Merge Score	Run Time
Random	0	262.8	942.72	$<$ 1 sec

Table 2. *Results with the random policy baseline model. Tested using computer B.*

Evaluation Function	N.O. Rollouts	2048%	Avg Sum of Final Board	Avg Merge Score	Run Time
Largest Merge Sum	50	13	1729.42	12647.12	4 hrs 8 min 24 sec
Largest Merge Sum	100	27	2001.1	15205.16	9 hrs 27 min 38 sec
Largest Merge Sum	250	34	2261.44	17516.6	26 hrs 27 min 38 sec
Largest Merge Sum	500	42	2393.54	18740.4	2 days 8 hrs 32 min 52 sec

Table 3. *Results with the Largest Merge Score Valuation Function. Tested using computer A.*

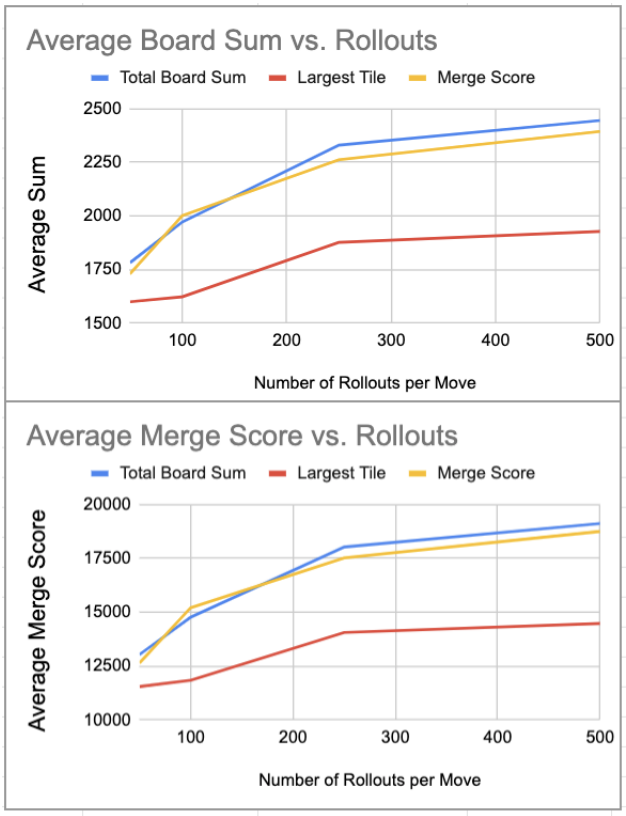

Figure 1. Average merge score and average board sum occurrences over rollouts for the three value functions.

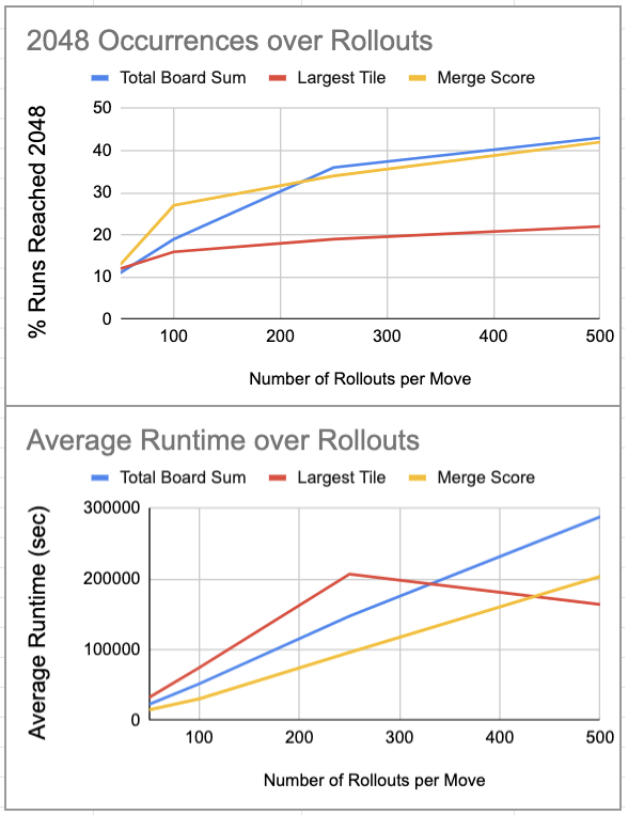

Figure 2. 2048 occurrences and average runtime over rollouts for the three value functions. We ran the experiments for different value functions of separate computers, with the exception of the largest tile value function with 500 rollouts which was run using the computer used for merge score.

2 shows our results for the random policy.

4.2. Monte Carlo Tree Search

We compared three different value functions: the merge score of the game (calculated by taking the sum of all merged tiles throughout the game), the sum of all of the tiles on the final board of the game, and the largest tile on the final board of the game (with ties broken by the sum of the tiles on the final board). We also tested different number of rollouts (50, 100, 250, and 500 rollouts). We ran 100 trials for each combination of value function and number of rollouts.

Of the agents we tested, the best performing agent used

Evaluation Function	N.O. Rollouts	2048%	Avg Sum of Final Board	Avg Merge Score	Run Time
Largest Sum of Board	50	11	1781.86	13037.76	6 hrs 17 min 43 sec
Largest Sum of Board	100	19	1971.22	14763.96	14 hrs 21 min 36 sec
Largest Sum of Board	250	36	2329.18	18022.88	1 day 16 hrs 57 min 48 sec
Largest Sum of Board	500	43	2444.58	19107.28	3 days 8 hrs 3 min 17 sec

Table 4. *Results with Sum of Tiles on Final Board. Tested using computer B.*

Evaluation Function	N.O. Rollouts	2048%	Avg Sum of Final Board	Avg Merge Score	Run Time
Largest Tile	50	12	1599.42	11552.04	8 hrs 59 min 27 sec
Largest Tile	100	16	1622.72	11839.96	20 hrs 39 min 30 sec
Largest Tile	250	19	1876.66	14054.12	2 days 9 hrs 31 min 1 sec
Largest Tile	500	22	1927.36	14471.96	1 day 21 hrs 35 min 54 sec

Table 5. *Results with the Largest Tile Valuation Function. Tested using computer C except for 500 rollouts, which was tested using computer A.*

the sum of all of the squares on the final board as the value function and had 500 rollouts, achieving the 2048 tile in 43% percent of its games, an average sum of 2444.58 on the final board of its games, and an average merge score of 19107.28 in its games. The agent which used the merge score of the game as the value function and had 500 rollouts performed marginally worse, achieving the 2048 tile in 42% percent of its games, an average sum of 2393.54 on the final board of its games, and an average merge score of 18740.4 in its games.

We found that the MCTS agents which used these value functions with any number of rollouts we tested (at least 50 rollouts) had much better performance in all three categories (percentage of games where the tile 2048 is achieved, average sum of the final board, and average merge score) when compared to the random policy, as shown in Tables 2, 3, 4 and 5.

The results of our experiments comparing the three value functions using different numbers of rollouts can be visualized in Figures 1 and 2 and are documented in Tables 3 (results using the largest merge score as the value function), 4 (results using the sum of tiles on final board as the value function), and 5 (results using the largest tile on the final board as the value function).

4.2.1 Comparison of Number of Rollouts

For all three value functions, increasing the number of rollouts also increased the performance of the agent across all three categories: percentage of games where the tile 2048 is achieved, average sum of the final board, and average merge score (as demonstrated in Figure 1 and the top plot in Figure 2). This makes sense, as the more rollouts we do, the more information we are able to gather on each action, so doing more rollouts allows us to make a more informed decision.

However, the rollouts exhibit decreasing marginal utility (as demonstrated in Figure 1 and the top plot in Figure 2). This makes sense, as when we have many rollouts, the new information provided by an additional rollout increases the total amount of information by a smaller percentage, since we have a larger amount of total information.

The runtime scales relatively linearly with the amount of rollouts we perform (when the same computer is used), as shown in the bottom plot for Figure 2. This makes sense as the number of games which we play at each move increases linearly with the number of rollouts. However, since the more rollouts we play, the better performing the agent tends to be, when we increase the number of rollouts, we also increase the average number of moves in the game (as the agent is able to progress further in the game before the board is full and no two squares can be merged). Thus, the runtime scales a little more quickly than linearly with the amount of rollouts.

4.2.2 Comparison of Value Functions

The first value function we evaluated calculates the sum of every merge that occurs. This is the same as the traditional scoring method for 2048. The results of this value function are shown in table 3. this value function, when used with 500 rollouts, reaching the tile 2048 or better 42% of the time, an average final board sum of 2393.54, and an average merge score of 18740. As expected, agents with this value function perform well it comes to getting optimizing the merge score for the game. Yet at the same time, agents with this value function also performed well in terms of reaching reaching a tile with 2048 and maximizing the sum of the total tiles left on the board at the end.

Our second value function is the sum of all the tiles on the board when the game ends. We sum the resulting tiles for each rollout we perform, and choose the action associated with the highest average for the sum of tiles on the final board. Table 4 shows the results for the various numbers of rollouts for this value function. At 500 rollouts, we reached the tile 2048 or better 43% of the time, an average merge sum of 19107.28, and an average board sum of 2444.58. This valuation function performed slightly better than the merge score function when 500 rollouts were used, although this may be also in part due to better random luck: they performed very similarly when comparing agents using the same number of rollouts.

For our final value function, we use the value of the largest tile generated by each run of the game. We choose the move with the maximum average largest tile. As described, we evaluate our agent using the percentage of games it achieves the 2048 tile, so it is reasonable to prefer states that reach higher tiles. Table 5 shows the results from this value function. With this value function and 500 rollouts, we reached 2048 only 22% of the time, with an average merge sum of 14471.96 and an average board sum of 1927.36. Overall, the agent which used the largest tile generated as the value function performed the worst out of our three value functions. Since we get larger tiles through this process of merging, we would want to encourage the process. However, in this valuation function, we only care about the highest number achieved on the board at the end of the game, so there is less reward for getting more merges throughout the states. In the end, this method is shown to be inferior than the other two models.

5. Conclusion

We created an agent which plays 2048 using the Monte Carlo Tree Search algorithm, comparing three different value functions: the merge score of the game (calculated by taking the sum of all merged tiles throughout the game), the sum of all of the tiles on the final board of the game, and the largest tile on the final board of the game (with ties broken by the sum of the tiles on the final board). We found that the MCTS agents which used these value functions had much better performance in all three categories (percentage of games where the tile 2048 is achieved, average sum of the final board, and average merge score) when compared to the random policy.

From our experiments for the Monte Carlo Tree Search algorithm, we can see that the most optimal value function to use to reach the target of 2048, get the largest sum of the tiles on the final board, and get the largest merge score, is the sum of the squares. However, it is also notable that the merge score valuation function performs nearly as well as the largest tile function. The largest tile value function, which only takes into account the largest tile on final board state, performed the worst.

Finally, we note that increasing the number of rollouts improves the average final board sum, average merge score, and percent of games which achieve 2048. However, the number of rollouts is relatively linearly correlated with the runtime and there are decreasing marginal returns.

6. Future Work

In our project, we chose to analyze three valuation functions to run with MCTS for a computer to beat the game of 2048. Currently, following the MCTS algorithm, we perform a randomized action for each rollout step. Since we know that the goal of 2048 is to either reach the largest tile, largest sum of tiles on the final board, and largest merge score, we can add in heuristics that, at every rollout move, favor moves that either merge two tiles together, or create a new largest tile. One additional strategy people tend to use is to keep the largest tile in a corner, so we can also add heuristics to simulate this game-play strategy. Of course, MCTS is only one of the many functions that can be used to simulate the game. Some other algorithms that can be considered include expecti-minimax, and using Convolutional Neural Networks.

Furthermore, many optimizations can be made to our algorithm to make it run faster. For example, on an average computer, our algorithm took more than 2 days to run 100 trials of 500 rollouts. A computer equipped with faster processors would allow us to test agents which have more rollouts for each move, which may allow us to further determine whether or not there would be decreasing marginal utility for additional rollouts past a certain point.

7. Contributions

Nathaniel helped build the 2048 game class and implemented the baseline algorithm.

Simon added merge score calculations to the 2048 game and allowed the Monte Carlo Tree Search algorithm to use different value functions.

Katherine helped build the 2048 game class and implemented the structure for the Monte Carlo Tree Search algorithm.

References

[1] M. Kochenderfer, T. Wheeler, and K. Wray. Algorithms for decision making. 2020.

[2] P. Rodgers and J. Levine. An investigation into 2048 ai strategies. pages 1–2, 2014.

[3] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, and et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[4] M. Szubert and W. Jaśkowski. Temporal difference learning of n-tuple networks for the game 2048. pages 1–8, 2014.