Progress Report

AI Agent for 2048: A comparison between Brute Force Search and Q-Learning over an expanded game space

Ruben Mayer (rmayer99), Magdy Saleh (mksaleh), Jayden Navarro (jaynavar)

Summary

The goal of our paper is to create an AI agent that plays the game 2048 effectively, and to compare the effectiveness of two popular techniques for solving the game over an expanded game space. Specifically, we will be comparing the effectiveness of a fixed-depth Brute Force search (implemented using an Expectimax Algorithm) with the effectiveness of a Reinforcement Learning approach (implemented using Q-Learning with Linear Approximation). Our goal is to compare the performance of these two techniques over a game space ranging from a 3×3 board to a 6×6 board.

So far, we have implemented a preliminary version of both of our algorithms, and tested them in a 4×4 sized board (the classic size of the 2048 game). Both of our algorithms currently perform better than the random baseline, with our Q-Learning player beating the Greedy and Corner baselines as well. However, neither are fully optimized yet, and there is much room for improvement. For our next milestone, we will be focusing on optimizing our algorithms and comparing their performance over an expanding game space.

Game Engine Model

To model the 2048 game we defined a class to represent the game state called `Model` and a base class to represent the agents called `Player`. The game board is represented as an `NxN` two dimensional list which contains the integer values in the positions of the visual board representation and `None` for the empty positions. The `Model` class exposes methods to get the current board state and score, execute a move (`UP, DOWN, LEFT, RIGHT`), check if the game is over, and reset the game state.

The `Player` base class contains a run method which loops while the game is not over, retrieving the current game state and passing it into the method which retrieves a move implemented by the child class, and then executing the move. In order to simulate multiple runs, an outer loop is used over the number of iterations desired, using the reset model method after each run to setup the model for a new simulation.

Q-Learning With Linear Approximation

We have implemented a preliminary version of Q-learning with linear approximation to solve the game. The model has the following characteristics:

Model

In our Q-Learning model, each state is defined by the placement of tiles on the board. The potential actions are always `UP, DOWN, LEFT, RIGHT` with one exception: If an action does not lead to a change in state, it is

removed from the list of potential actions for that state. Finally, we defined the reward as the points scored after any given action. If an action leads to a lost game, then the reward is equal to $-1 \times$ TotalScore.

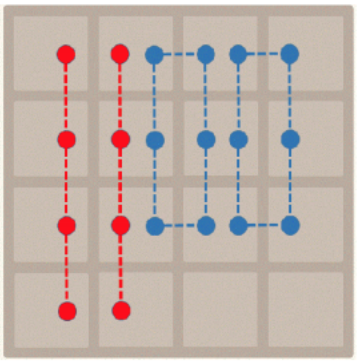

Figure 1: Q-Learning Features [2]

Features

We have implemented two types of $N-TupleNetwork$ features based on the work by Wu, Yeh, Liang Chang, and Chiang [2]. The first feature tracks the values of vertical and horizontal straight-line arrangement of the tiles on the board (a tuple network of size 4, see red lines in *Figure 1*), and the second one tracks the values of tiles forming 3×2 rectangles on the board (a tuple network of size 6, see blue lines in *Figure 1*). These features are effective because they encapsulate the most important aspects of the game (the arrangement and position of all adjacent tiles), but they are small enough such that their state space is manageable ($O(10^6)$ and $O(10^4)$).

Adjustment of Update Value

We added an adjustment to the weights update. The goal of the function is to reduce the impact of update values that are too large, and to progressively reduce the impact of each update on a per-feature basis. Specifically, instead of updating our weights according to:

$$weights[feature] = weights[feature] \times \eta UpdateValue$$

We are using the following update function:

$$weights[feature] = weights[feature] \times \frac{\sqrt{UpdateValue}}{\sqrt{1 + i_{feature}}}$$

where $i_{feature}$ represents the total number of previous updates on the feature.

Additional Notes

Since there is some natural randomness to the 2048 game, it was not necessary to incorporate any exploration to our algorithm. So far, we have only tested the algorithm on a 4×4 board, and it performs significantly better than our baselines, but not as well as the state of the art results from the literature (the current results are discussed in a subsequent section).

Expectimax

Model

For our second agent, we are approaching this problem by modeling it as a two player game with a probabilistic opponent. Here the agent is the player taking his turn swiping and the opponent is the random tile generated between every turn. We know that a random tile will be generated in any of the unoccupied squares with a 90% probability of it being a tile of value 2 and a 10% probability of value 4.

Algorithm

Our Expectimax algorithm works as follows: At every move the algorithm is given the current state of the board. At first, we generate the board for all possible next moves (for the agent) and then we recurse on each of these boards, then we model all the possible next moves (random tile spawning) on each of these boards. If any of the generated boards represents an end to a game we return the score. Otherwise if we reach max depth we run the current board through a heuristic function to estimate the strength of that position. Since the opponent is random we use the expectation of all the scores in the opponents turn and the maximum of all the possible moves in the agent's turn.

Heuristic

The current heuristic function we are using is still very simple. We aim to maximize the number of unoccupied tiles in the grid and ensure that the player has their highest valued tile in the top right corner (this a widely used strategy in the game). We do this by giving out a per-free tile reward as well as a big reward if the player has the highest tile in the corner.

Initial Results

We ran one thousand simulations for our three baseline players and our two "smart" players on a 4x4 grid. The Q-Learning algorithm was trained over 10,000 iterations, and the Expectimax algorithm used a depth of 1. We produced the following results:

Table 1: Score and Max Tile statistics for each agent on the 4x4 grid (1K Simulations)

	Max Score	Avg. Score	Stdev Score	Max Max Tile	Avg. Max Tile	Stdev Max Tile
Greedy	11424	3143	1530	1024	242	129
Corner	7128	2508	1219	512	196	98
Random	3316	1110	530	256	106	52
$Q - Learning$	50968	14160	6458	4096	763	403
Expectimax	5008	1241	676	512	115	63

As shown by the data above, the Q-Learning algorithm was able to achieve at least one winning game (reaching 2048), and even made it to the next higher tile (4096). The other values were also much larger than the baseline players across all statistics, showing a significant improvement over the naive approaches.

The Expectimax algorithm could only be operated at a depth of 1 due to time constraints and optimization issues, so we could not show its real potential at this stage. An interesting thing to note is that the Expectimax algorithm with depth 1 should perform slightly better than the naive approaches as it evaluates a heuristic on all possible next states, as opposed to just the current state. This implies that there could be

an issue with the implementation, or the heuristic is not ideal. We plan on addressing these issues between now and the final report (see *Next Steps* section below for more details).

While both of our "smart" algorithms are performing better than the random baseline, there is still a lot of room for improvement (the state of the art models can earn up to 600,000 points on average [2]).

Next Steps

There are three main areas where we have more work to do:

Optimize Q-Learning

We plan to improve the performance of our Q-Learning algorithm by implementing more features, such as $L-TupleNetworks$ [2], and by improving the run time of the algorithm so that we can run more iterations.

Optimize Expectimax

We plan on improving the heuristic function that we are using for our Expectimax algorithm. Specifically, we will add the following to our current function:

- The monotonicity of each row and column, which measures whether tiles are arranged in increasing magnitude or not (increasing magnitude is better).
- The total amount of "Collapsible Tiles," which measures the total number of tiles that can be collapsed into bigger tiles at any given state.

In addition, we plan to optimize the time performance of the algorithm so that we can run it using a larger depth.

Implement comparison over increasing state spaces

Our final step will be to implement a comparison of the Expectimax and Q-Learning algorithms over different sized boards, and draw a conclusion about which performs better as the state space increases.

References

[1] Nneonneo. 2048 AI. *GitHub*.

[2] 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, 2017.

[3] Rahul Mehta. 2048 IS (PSPACE) HARD, BUT SOMETIMES EASY. Technical report, 2014.

[4] Wojciech Jaśkowski. Mastering 2048 with Delayed Temporal Coherence Learning, Multi-Stage Weight Promotion, Redundant Encoding and Carousel Shaping. 2016.