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

Ruben Mayer-Hirshfeld
Department of Computer Science
Stanford University
Stanford, CA 94305
rmayer99@stanford.edu

Jayden Navarro
Department of Computer Science
Stanford University
Stanford, CA 94305
jaynavar@stanford.edu

Magdy Saleh
Department of Computer Science
Stanford University
Stanford, CA 94305
mksaleh@stanford.edu

Abstract

Since 2048 went "viral" in 2014, many researchers have come up with AI agents that play the game effectively. Given the game's easy to simulate mechanics and its large state-space, the most effective of these agents play the game using Temporal Difference Learning - an approach that combines a search of the game space with linear generalization to find the best moves in the game space. Our work compares the effectiveness of an online, exploration based approach (Expectimax Tree Search) and an offline approach with generic learning features (Q-Learning with function approximation) over an expanded game space. We implemented these two algorithms, achieving an improvement over the average score achieved by a Q-Learning agent in the literature. By comparing the performance of these two algorithms over 2048 boards of size 4×4, 5×5 and 6×6, we show that Expectimax is more effective at generalizing the game space than Q-Learning, and that this advantage increases as the game space grows.

1 Introduction

2048 is a non-deterministic puzzle game that went viral in 2014. While the mechanics of the game are simple to learn, playing the game can be quite challenging. As a result, there have been a wide range of attempts to create an AI agent that plays the game effectively. The game can be modeled as a Markov Decision Process (MDP); however, the the game's state space is very large ($O(2^{20})$ for the traditional 4×4 board), which makes finding an optimal policy an interesting search problem.

The most effective 2048 agents published thus far have consisted of variations of Temporal Difference (TD) Learning - an On-Policy Reinforcement Learning-based approach that includes exploration during the search process. In addition, some implementations of Expectimax - an exploration-based approach with manually tuned heuristics has also yielded respectable results. From the literature, it is clear that exploration-free approaches such as Q-Learning with Function Approximation have not been as effective for solving the search problem. However, there is little information about how the effectiveness of exploration-free approaches to solving 2048 compare to exploration-based approaches as the state space grows.

In this paper, we compared the difference in effectiveness between exploration-based and Reinforcement Learning-based approaches to solving 2048 over an expanding game space. In particular, we compared the effectiveness of Q-Learning with function approximation with the effectiveness of Expectimax with manually tuned heuristics. We chose these two algorithms because they represent two distinctly different approaches to generalizing over a large state space. Expectimax relies on a high level of domain knowledge that includes a finely tuned heuristic function, as well as forward-looking simulation of the game, whereas Q-Learning relies exclusively on generic features that do not model any strategy or game play.

Our results showed that the Expectimax agent was significantly more effective than the Q-Learning agent, and that this dominance increased as the state space grew larger.

The paper is structured as follows: Section 2 places our contribution in context of the relevant literature. Section 3 delves into the game engine, as well as the infrastructure that we used in our implementation. Sections 4 and 5 go over our Q-Learning and Expectimax implementations. Section 6 displays our results, section 7 presents a discussion of our findings and error analysis and finally, section 8 concludes and proposes areas of future work.

2 Literature Review

The foundational work that introduced reinforcement learning for game playing was first published by Samuel, who created a Checkers AI Agent, and then by Tesauro, who created an expert Backgamon agent by using TD learning [1] [2]. Since then, Reinforcement learning techniques have been used to solve a wide variety of games, including Tetris, Connect 4, Atari and Go [3], [4], [5], [6].

Temporal Difference Learning

The use of TD Learning for 2048 was first introduced by Szubert and Jaskowski, who used n-tuple network features based upon the work by Bledsoe and Browning to reach an average score of 100,178 [7] [8]. Jaskowski later built on this work to achieve the highest 2048 score to our knowledge by implementing a Delayed Temporal Coherence Learning algorithm that made use of "multi-stage function approximator with weight promotion, carousel shaping, and redundant encoding," to reach an average score of 609,104 [9].

Expectimax Tree Search

The Expectimax Tree Search algorithm has also seen a high level of popularity for creating a 2048 agent. Wu et al. implemented a combination of TD-Learning with an Expectimax search implementation with depth 5 to reach an average score of 443,526 [10]. In addition, there are several open source Expectimax-based 2048 agents online. To our knowledge, the best of these, created by Xiao et al reaches an average score of 442,419 by making use of a sophisticated heuristic function, along with a bitboard game representation and variable-depth search [11].

Q-Learning

There is limited research on Q-Learning implementations for 2048. This can be explained by the fact that when game mechanics are known, TD-Learning almost always performs better than Q-Learning. To our knowledge, the best Q-Learning agent was published by Szubert and Jaskowski, achieving an average score of 20,504 [7].

Expanding game space

To our knowledge, no published research has implemented an AI-Agent for 2048-like games over an expanded game space, or compared the performance of Q-Learning with Expectimax over such a game space.

3 Game Engine and Infrastructure

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. Child classes of `Player` were created for the baselines (Greedy, Corners, Random), Q-Learning, and Expectimax.

In order to collect statistics about the performance of our agents, for each run we store the runtime (seconds), score, maximum tile, and number of moves. After an iteration of runs we output the max, min, median, and stdev of each of these categories, along with a histogram of the maximum tiles.

Infrastructure and data collection

After the initial implementation of the game engine, we needed to verify that it was bug-free before building out the rest of our agents on top of it, so we implemented a unit test that performed black box testing of the functionality of the exposed interfaces. After testing the game engine and writing the baseline agents, we added a sanity test that verified each agent performed within a reasonable upper and lower bound for each of the statistics (score, maximum tile, number of moves).

To prevent regressions, we setup Travis CI to run all of our unit tests on every commit. This ensured that if we broke the game engine or degraded the performance of either of our smart players, we'd be notified and could look into the issue. On multiple occasions this proved useful, as a drastic improvement to our heuristic caused the scores for Expectimax to break the test alerting us to the sudden boost in performance, and in another instance Expectimax had a bug that was causing it get stuck in an "UP" loop which the test caught.

To collect data for Expectimax we tuned the hyperparameters (depth and probability cutoff) to get a single run within a certain runtime and then run a set number of iterations to collect data. To automate this we wrote a wrapper script that would mirror the program output to stdout and to a data file so we could preserve the data while still viewing the progress.

For Q-Learning it was much more difficult, as we had to control both the training of the weights and also the inference data collection. To solve this issue we developed a system called `sequences` which allowed us to specify a set of trials to perform in a yaml file, and then we could run our test harness with a given sequence and collect data in this manner. This allowed us to train for a certain number of iterations, then collect pure inference data, and then continue this pattern in a loop. Similar to the Expectimax data collection, we wrote a wrapper script to allow us to easily gather data.

Due to the Q-Learning weights size, which could use as much as 100GB of RAM for large board sizes, it was unfeasible to use our own machines or even a shared machine like Rice. We opted to use Google Cloud Platform (GCP), and ran a 55 vCPU 350 GB RAM virtual machine to gather our Q-Learning data, and a 16 vCPU 100 GB RAM virtual machine to collect our Expectimax data. After trials were completed, all data was checked into the git repository for further analysis.

4 Q-Learning

We have implemented Q-learning with linear approximation to create a game agent. Our implementation had 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. 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.

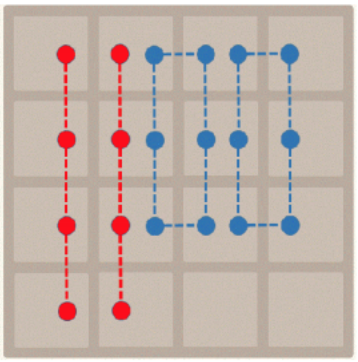

Figure 1: Q-Learning Features [10]

Features

We implemented two types of $N-Tuple\ Network$ features based on the work by Wu, Yeh, Liang Chang, and Chiang [10]. 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 feature 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 track all the possible placements and orientations of vertical lines, horizontal lines, 4×6 rectangles and 6×4 rectangles. These features are effective because they encapsulate the most important aspects of the game (the arrangement and position of all adjacent tiles).

The feature space for our 4×4 board is in the order of $10^{7.6}$ [1], it is in the order of $10^{8.0}$ for our 5×5 board [2] and it is in the order of $10^{8.8}$ for our 6×6 board[3]. The main factors that lead to an increased feature space include a larger maximum tile value, as well as a larger number of total features.

Learning equation

We added an adjustment to the traditional Q-Learning equation, specifically to the weight vector update. The goal of our modified 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. On each state, action reward and successor state tuple (s, a, r, s') we performed an update according to the equation:

$$\mathbf{w} \leftarrow -\sqrt{\eta[\hat{Q}_{opt}(s,a;\mathbf{w}) - (r + \gamma \hat{V}_{opt}(s'))]}(\phi(s,a) \cdot \mathbf{n})$$

Where $\mathbf{n}$ is a vector with values:

$$\frac{1}{\sqrt{freq_{\phi(s,a,i)}}}$$

Where i represents each value in $\phi(s,a)$, and $freq_{\phi(s,a,i)}$ represents the total number of times that the $i'th$ feature of $\phi(s,a)$ has appeared in an update. Thus, the first time that a feature appears, the

[1] $\log(12^6 \times 12)$ where the first value represents the total number of tile values (where the max tile value is 2^{12}) the exponent represents the size of the largest feature and the second value represents the total number of size 6 features

[2] $\log(13^6 \times 24)$ using the same logic as the 4×4 board

[3] $\log(15^6 \times 52)$ using the same logic as the 4×4 board

update has a big impact, but as this feature appears more often, the impact of the update is reduced specifically on that feature. Since our modification already reduced the impact of updates as the algorithm performed more iterations, we used a value of one for our step size η. We also used a value of one for our discount factor γ because the agent's performance is evaluated based on its overall score.

5 Expectimax

Model and algorithm

For the sake of this work we used a limited depth form of the Expectimax algorithm, where we alternate between the agent's turn and a random move of tile spawning. The expectation step used for the tile spawn is weighted since for any position there is a 90% chance the tile spawned is a two and a 10% chance that it is a four.

$$V_{em}(s,d) = \begin{cases} -100 & \text{IsEnd(s)} \\ \max_{a \in \text{Actions}} V_{em}(\text{Succ(s, a)}, d) & \text{Player(s) = agent} \\ \sum_{a \in \text{Actions}} \text{Prob(Succ(s, a))} * V_{em}(\text{Succ(s, a)}, d-1) & \text{Player(s) = random} \\ \text{heuristic(s)} & d = 0 \text{ or Prob(s)} < \text{Cutoff} \end{cases}$$

We use the above recurrence for the implementation of the algorithm, where s is the state, represented as the current board and the score, a is an action taken from state s and d is the search depth. Prob(s) describes the probability of reaching a given state s. We also give a negative score of 100 at the end of the game to discourage the algorithm from reaching the end of the game as we want it to continue playing for as long as possible.

The depth limited version of the algorithm is used to make the performance possible as otherwise the state space is too big to be fully explored at each move. We also prune states that are unlikely to be reached. The reason these approximate optimizations are used is because the state space of the algorithm is $O((4 \cdot (2 \cdot n^2))^d) \rightarrow O(8^d n^{2d})$, since at every depth d a player can make four possible moves, and you can spawn a two or a four tile at each of the $O(n^2)$ open locations. Thus, for a depth of 8 search on a 4×4 board, the number of search paths is $O(10^{17})$ (note that this double counts many board configurations, necessitating the use of Dynamic Programming).

Heuristic

Arguably one of the most important aspects of the algorithm is the heuristic used to evaluate the boards when the depth is zero or probability cutoff is reached. We look at multiple factors that traditionally feature in a good strategy to play the game and we design a heuristic to reward such strategies. However, there is a balance that needs to be struck between encouraging a strong strategy and explicitly forcing the heuristic to only reward that strategy, since in the latter case that could be replaced by a rule based algorithm. Our heuristic encourages four main things:

- Maximum number of unoccupied tiles
- Highest value tile is in a corner
- Monotonicity of rows and columns
- Number of adjacent tiles that can be merged

We understand that there is a bit of a trade off between maximizing the number of adjacent tiles and the number of unoccupied tiles, but the idea is that at least one will be maximized between both, allowing for a better outcome of the game.

Implementation

We initially implemented Expectimax in Python, however for depths larger than two it would take multiple seconds to perform a single move, leading to a single game taking hours to complete.

A vast performance improvement was required, so we opted to rewrite Expectimax in C++ from scratch, and interface with our existing infrastructure using Python ctypes. The Python Expectimax player became a wrapper that, on every `getMove` call, passed the Python board state into the C++ code and called out to the `getMove` C++ function, returning back the computed move.

After converting to C++ using a two-dimensional int vector to represent the board state, we decided to optimize further by representing the board as a flat char array of tile exponent numbers (e.g. rather than storing 256 we store 8). We provided two interfaces to the board: one that interacts directly with the raw exponent numbers and another that converts it into and out of the base-2 representation using `log2` and `pow`. This made it easy to transition to the new board, and we could gradually convert everything into using the raw board representation when appropriate for performance gains.

We implemented memoization (Dynamic Programming) in C++ using a class consisting of board state, player (`USER` or `TILE_SPAWN`), and depth as the key, which lead to an exponential performance improvement (note: we had used memoization in our Python implementation as well).

Not all states are equally likely for this game. There is a 90% probability of spawning a 2 tile during a random move, compared to 10% for a 4 tile, so a probability cutoff was used to prune parts of the search tree where the probability of reaching those states was below a certain threshold. While this is inherently an approximation it meant that we could substantially speed up performance, allowing us to search higher depths.

6 Results

Board Size	Expectimax Depth	Expectimax Probability Cutoff	Q-Learning Iterations
4×4	9	1×10^{-6}	500,000
5×5	7	5×10^{-5}	100,000
6×6	6	1×10^{-4}	5,000

Table 1: Parameters used in data collection

Board Size	EM: Avg. Score	QL: Avg. Score	EM/QL Ratio	EM: Med. Max Tile	QL: Med. Max Tile
4×4	41,727	24,102	1.7x	2048	1024
5×5	436,393	42,612	10.2x	16384	2048
6×6	5,995,388	753,978	8.0x	262144	32768

Table 2: Properties of different board sizes

We ran a side by side comparison of the performance of our Expectimax algorithm and our Q-Learning algorithm over a 4×4 sized board, a 5×5 sized board and a 6×6 sized board. We trained the agent for the 4×4 board for 500,000 iterations based on comparable experiments in past literature, and tuned the parameters for the rest of our experiments so that they would have a comparable run time (around 24 hours) [7]. The specific parameters that we used in each experiment can be seen in Table 1. It is important to note that for every board, the performance of our Q-Learning algorithm performance came very close to convergence, as can be seen in Figure 8.

For the 4×4 board, our Q-Learning agent improved upon the existing literature, attaining an average score of 24,102, in comparison to an average score of 20,504 achieved by Szubert and Jaskowski's Q-Learning implementation. Our Expectimax algorithm, however, under-performed the state of the art, attaining an average score of 41,727 in comparison to 442,419 in Xiao et al's implementation (this gap in performance is explained in the following section) [11].

Despite the shortcomings of our Expectimax algorithm, it still outperformed the Q-Learning algorithm on every board size. Furthermore, Expectimax's performance over Q-Learning increased as the board size got bigger, with Expectimax getting a score that was 1.7 times greater in the 4×4 board, but was 10.2 and 8.0 times greater for the subsequent two board sizes (see Table 2).

4 × 4 comparison

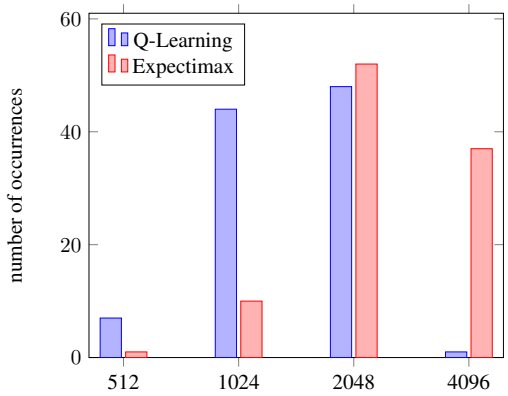

Figure 2: 4×4 maximum tile comparison

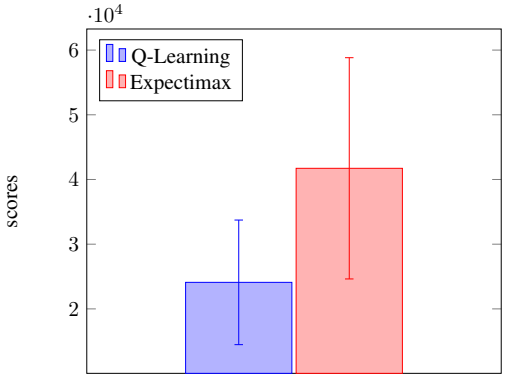

Figure 3: 4×4 score comparison

On the 4×4 board size, Q-Learning and Expectimax both had a most frequent maximum tile of 2048, however Expectimax was able to reach the 4096 tile a large number of times while Q-Learning only reached it once. In terms of the scores, Expectimax on average yielded 1.7 times the scores of Q-Learning (see Table 2).

5 × 5 comparison

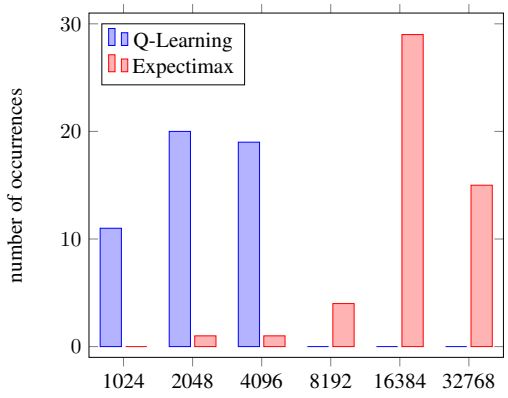

Figure 4: 5×5 maximum tile comparison

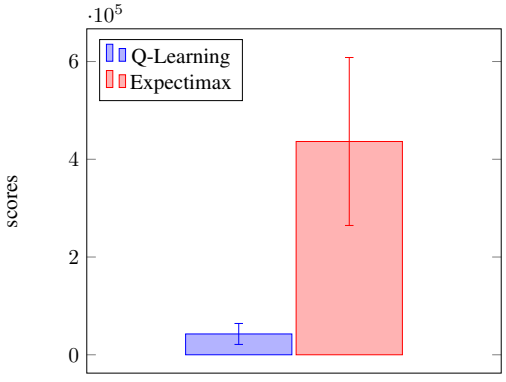

Figure 5: 5×5 score comparison

On the 5×5 board size, Q-Learning and Expectimax only overlapped on the smaller tiles, where Expectimax had a few outlier runs that yielded low maximum tiles. Besides these outliers, Expectimax yielded much larger maximum tiles. In terms of the scores, Expectimax on average yielded 10.2 times the scores of Q-Learning (see Table 2).

6 × 6 comparison

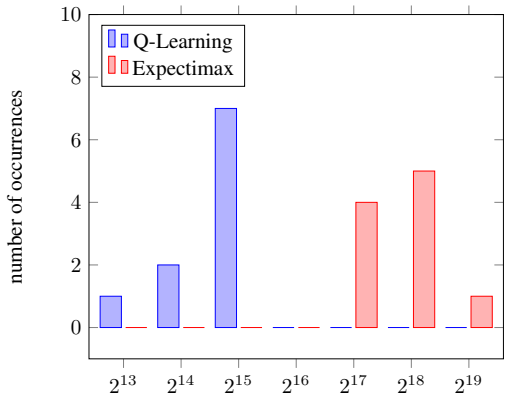

Figure 6: 6 × 6 maximum tile comparison

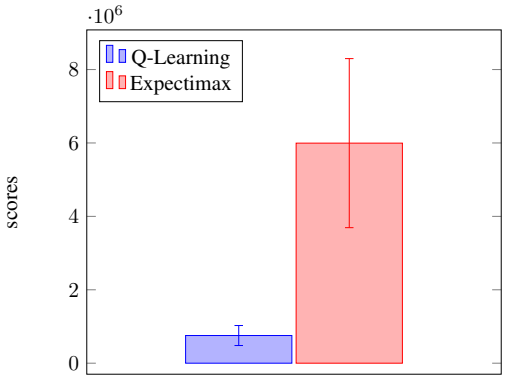

Figure 7: 6 × 6 score comparison

On the 6 × 6 board size, Q-Learning and Expectimax had no overlap, including having a tile separating the two players that was reached by neither (2^{16}), as Q-Learning never performed well enough and Expectimax never performed poor enough to reach it. In terms of the scores, Expectimax on average yielded 8.0 times the scores of Q-Learning (see Table 2).

Q-Learning scores over training iterations

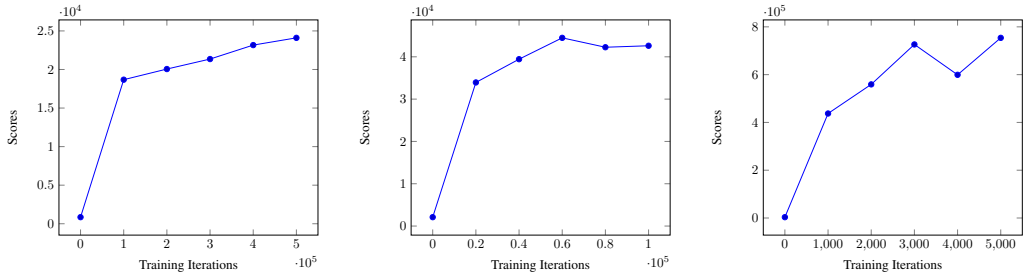

Figure 8: From left to right: 4 × 4, 5 × 5, and 6 × 6 scores over training iterations, respectively

From these charts, it is clear that most of the Q-Learning training gains take place during the first few iterations, and then the gains decrease exponentially. Note that the drop in average score as iterations increase seen in the 5 × 5 board and the 6 × 6 board are caused by random variation in the specific games (we only performed 50, and 10 trials for each data point respectively due to time constraints).

Expectimax with varying hyper-parameters

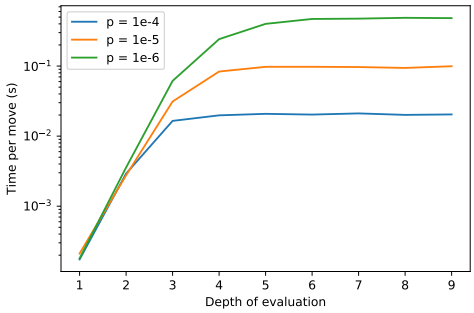

Figure 9: Time per move of the Expectimax algorithm vs. depths based on different pruning cutoff probabilities on the 4×4 square

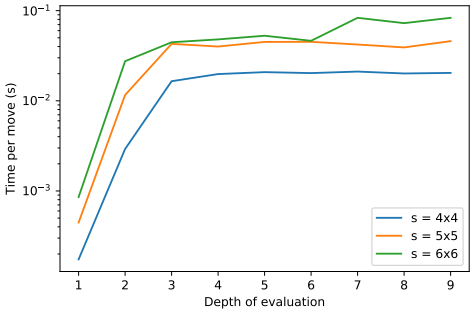

Figure 10: Time per move of the Expectimax algorithm vs depths based on different board sizes given a 10^{-4} cutoff probability

One of our aims was to look to understand how the Expectimax algorithm depends on different hyper-parameters. In Figure 9 it is clear that at initial depth, the pruning probability does not impact the time per move at all, since no pruning is occurring at such shallow searches. We then find that the curves diverge and plateau at different times, which is due to the fact that at higher depths almost all the paths are being pruned and so the cutoff probability is the limiting factor. We find that for a depth 6 and on-wards, having a cutoff probability of around 10^{-6} leads to moves taking almost a second each. It is also interesting to note that the difference in time per move for different board sizes is fairly constant across different depths as seen in Figure 10.

7 Discussion

Our results clearly show that, given 24 hours to run, an Expectimax Tree Search 2048 agent outperforms a Q-Learning agent, and that this advantage increases as the state space grows. This implies that a finely tuned heuristic function, based on specific knowledge of the game generalizes more effectively over 2048 boards than the generic features that power Q-Learning. One important caveat, however, is that Q-Learning saw a greater increase in performance for the 6×6 board in comparison to the 5×5 board relative to the increase in performance of Expectimax. Due to the limitations described below, it is unclear if this difference is reflective of any meaningful patterns, or just a product of the experimental factors.

Limitations

One key limitation to our work involves the low performance of our Expectimax algorithm in comparison to the state of the art. This gap can be explained by a key difference in the implementation - Xiao et al's implementation precomputes the heuristic function for all possible rows and columns before running, whereas our implementation computes these values on the fly. As a result, Xiao et al's agent can calculate a heuristic score for every single possible future move at each state, whereas our agent prunes out low-probability states to decrease runtime. This gap in performance, however, does not affect the validity of our findings - since the Expectimax algorithm outperformed Q-Learning despite its shortcomings, we can only assume that a better Expectimax implementation would be even more dominant.

Another limitation in our work involves time and computational constraints that prevented our Q-Learning agent from reaching full convergence. Our findings clearly show that Q-Learning's gains from more training iterations decreases exponentially, thus making it very unlikely that the gains from more training would have led to Q-Learning overtaking Expectimax.

A third limitation to our work involves the hyper-parameters that we used in our Expectimax implementation. We tuned these hyper-parameters to reach our desired runtime, using heuristic

experimentation to select a trade-off between the depth and probability cutoff parameters. However, this heuristic experimentation was not rigorous enough to confirm that we selected the optimal trade-off. Similar to our first limitation, this limitation does not put our overall conclusions in doubt because a better selection of hyper-parameters would have resulted in Expectimax being more dominant, not less.

While none of these limitations casts significant doubt on our overall findings, they do make it difficult to quantify the advantage of Expectimax over Q-Learning with a high degree of confidence.

8 Conclusion

We compared the performance of Expectimax Tree Search and Q-Learning with a function approximation over an expanding 2048 game space. Our results showed that the Expectimax agent performed better than the Q-Learning agent, and that this dominance increased as the state space grew. This shows that the game-specific knowledge that powers the Expectimax heuristic generalizes more effectively over a larger game space than the generic features that power the Q-Learning algorithm.

While our overall conclusions are defensible, a few limitations such as the poor performance of our Expectimax algorithm compared to the literature, as well as time and computational constraints prevent us from precisely quantifying the advantage of Expectimax over Q-Learning. An interesting area for future research would involve creating a precise, quantitative comparison of the two algorithms. In addition, it would be interesting to compare the performance of the two algorithms over more than three data points (for instance, by adding boards with rectangular dimensions such as 3×4). In this way, it would be possible to rigorously infer a trend in the relative performance of the two algorithms.

References

[1] A. L. Samuel. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development*, 44(1.2):206–226, 7 2000.

[2] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–69, 3 1995.

[3] Bruno Scherrer, Victor Gabillon, Mohammad Ghavamzadeh, and Matthieu Geist. Approximate Modified Policy Iteration. 5 2012.

[4] Markus Thill, Patrick Koch, and Wolfgang Konen. Reinforcement Learning with N-tuples on the Game Connect-4. pages 184–194. Springer, Berlin, Heidelberg, 2012.

[5] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 7 2012.

[6] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 1 2016.

[7] Marcin Szubert and Wojciech Jaskowski Jaskowski. Temporal Difference Learning of N-Tuple Networks for the Game 2048. Technical report, 2014.

[8] Paweł Liskowski, Wojciech Jaskowski Jaskowski, and Krzysztof Krawiec. IEEE TRANSACTIONS ON COMPUTATIONAL INTELLIGENCE AND AI IN GAMES Learning to Play Othello with Deep Neural Networks. Technical report.

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

[10] 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.

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