A Comparative Study of Deep Reinforcement Learning and Expectimax Search for the Game 2048

Soham Konar
Stanford University
skonar@stanford.edu

Danny S. Park
Stanford University
danspark@stanford.edu

Andrew Wu
Stanford University
acwu02@stanford.edu

Abstract

The game 2048 presents a challenging environment for artificial intelligence agents due to its large state space and stochastic nature. This paper presents a comparative study of two distinct approaches to solving 2048: a model-free Deep Q-Network (DQN) and a model-based Expectimax search algorithm. We implement and evaluate a standard DQN, a variant using Prioritized Experience Replay (PER), and an Expectimax agent at various search depths. Our quantitative results show that the Expectimax agent, particularly at depth 4 achieving a mean score of 458.2, significantly outperforms the DQN variants, with performance scaling with search depth until constrained by computational limits. We analyze the performance of each agent, discuss the challenges encountered, including hyperparameter sensitivity in DQNs, and propose directions for future work, such as exploring hybrid agent architectures.

One-Page Extended Abstract

Motivation: The game 2048, while governed by simple rules, poses a significant challenge for artificial intelligence due to its high-branching-factor, stochastic nature, and large state space. This work is motivated by the desire to compare two distinct AI paradigms in this complex environment: model-free deep reinforcement learning (DQN) and model-based classical search (Expectimax). We aim to understand their relative performance, computational trade-offs, and the impact of specific enhancements like Prioritized Experience Replay (PER).

Method: We implemented and evaluated three primary agents. The first is a **Vanilla Deep Q-Network (DQN)**, which uses a convolutional neural network to approximate the action-value function and is trained with a standard experience replay buffer. The second is a **DQN with Prioritized Experience Replay (PER)**, an enhancement that samples transitions from the replay buffer based on their TD error, allowing the agent to learn from more "surprising" events. The third is a depth-limited **Expectimax agent**, which uses a handcrafted heuristic function to evaluate board states. This heuristic considers factors like tile monotonicity, smoothness, the number of empty cells, and the value of the maximum tile. The Expectimax agent explores the game tree to a specified depth, averaging over the outcomes of random tile spawns (the "chance" nodes) and selecting the move that maximizes its expected score (the "max" nodes).

Implementation and Results: Our agents were evaluated based on mean score, median score, and the maximum tile achieved. The Expectimax agent demonstrated significantly stronger performance than both DQN variants, with its mean score increasing with search depth up to depth 4 (mean score: 458.2). At depth 5, performance degraded, likely due to the extreme computational cost (over 13 hours for 20 episodes), which limited effective exploration. The Vanilla DQN (mean score: 93.52) unexpectedly outperformed the PER-enhanced DQN (mean score: 81.44). This suggests that our DQN implementations are highly sensitive to hyperparameters and may require more extensive tuning or longer training periods to realize the benefits of PER.

Discussion and Conclusion: Our findings indicate that for the game 2048, a well-designed, model-based search agent like Expectimax can outperform learning-based DQN agents, provided sufficient computational resources for an adequate search depth. The performance of DQN agents is heavily dependent on hyperparameter tuning, and techniques like PER do not guarantee improvement without careful optimization. The study highlights a clear trade-off: Expectimax offers strong performance at the cost of high per-move computation, while DQNs have a lower per-move cost but require extensive offline training and are sensitive to their configuration. Future work could explore hybrid models that use a learned value function from a DQN as a heuristic for a tree search, potentially combining the strengths of both approaches.

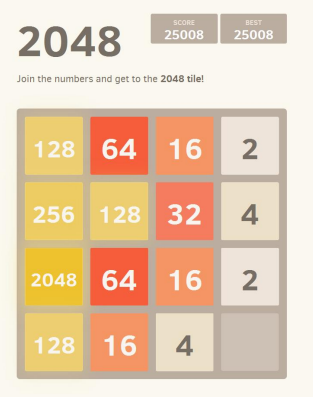

Figure 1. Example 2048 board.

1. Introduction

The game 2048, played on a 4x4 grid, is a seemingly simple puzzle with deep strategic complexity. The objective is to slide numbered tiles in one of four cardinal directions, merging tiles of the same value to create a tile with the value 2048. After each move, a new tile (either a '2' or a '4') appears in a random empty cell. While simple for humans to learn, mastering the game is a significant challenge for artificial intelligence due to its large state space (approximately 10^{16} states [5]), stochastic tile spawns, and the sparse, delayed nature of rewards, which can make credit assignment difficult and slow down convergence for learning agents like DQN.

This project aims to explore and compare the effectiveness of two fundamentally different AI paradigms in the context of 2048: model-free reinforcement learning and classical model-based search. Our primary research questions are:

1. How does the performance of a Deep Q-Network (DQN) agent compare to a traditional Expectimax search agent on the game 2048?

2. Can advanced techniques like Prioritized Experience Replay (PER) significantly improve the performance and learning efficiency of a DQN agent in this environment?

3. What are the trade-offs between the learning-based approach (DQN) and the search-based approach (Expectimax) in terms of performance, computational cost, and strategic behavior?

To answer these questions, we implement a Vanilla DQN, a DQN enhanced with PER, and an Expectimax agent with a carefully designed heuristic function. We evaluate these agents based on their game scores and highest achieved tiles, providing a quantitative comparison and a discussion of their respective strengths and weaknesses.

2. Related Work

Significant research has been conducted on applying reinforcement learning (RL) to 2048. Goga experimented with various DQN architectures and reward functions, finding that a modified, non-negative reward function combined with normalized encodings yielded the best results, though learning still plateaued [1]. Szubert and Jaskowski utilized temporal difference (TD) learning with n-tuple networks to represent value functions, achieving a win rate of over 97% and noting that the game's stochasticity made explicit exploration unnecessary [4]. More recently, Guei proposed an optimistic TD learning framework that achieved state-of-the-art (SOTA) performance with fewer network parameters, demonstrating high average scores and a 72% rate of reaching the 32768-tile [2]. Our work distinguishes itself by providing a direct, side-by-side comparison of a modern RL technique (DQN with PER) against a strong classical search algorithm (Expectimax), focusing on the practical trade-offs between these approaches. Classical search methods, including variants of minimax and Monte Carlo Tree Search, have also been extensively applied to 2048, often leveraging sophisticated heuristics and pruning techniques to manage the game's complexity.

3. Experimental Setup

3.1. Environment

All experiments are conducted using the open-source Python task environment *Gymnasium-2048* (gymnasium_2048 / TwentyFortyEight-v0). The action space is discrete with four actions (up, down, left, right), and the observation space is a 4x4 grid of integer tile values, represented as a Box(low=0, high=2^{17}, shape=(4, 4), dtype=int32).

3.2. Baselines

We compare our agent extensions against two main baselines:

- **Random Agent**: An agent that selects a legal move uniformly at random at each step. This serves as a conceptual lower bound for intelligent play.

- **Vanilla DQN**: Our primary baseline is a Deep Q-Network with a standard experience replay buffer, trained for 10,000 episodes. This allows us to directly measure the impact of our main extension, Prioritized Experience Replay.

3.3. Evaluation Metrics

To evaluate agent performance, we use the following metrics, averaged over 20 evaluation episodes for all agents:

- **Mean Score**: The average final score achieved across all evaluation episodes.

- **Median Score**: The median final score, which is less sensitive to outlier performances.
- **Max Tile Achieved**: The highest tile value (e.g., 1024, 2048) reached by the agent during evaluation.

4. Methods

4.1. Deep Q-Network (DQN)

Our baseline model is a Deep Q-Network (DQN) architecture. The network consists of two convolutional layers, each with a kernel size of two and followed by a ReLU nonlinearity, and two fully-connected linear layers. The model is trained using an epsilon-greedy policy to balance exploration and exploitation. For optimization, we use the Adam optimizer with an initial learning rate of 5×10^{-5}. The network weights are updated using the Q-learning rule:

$$Q(s,a) \leftarrow Q(s,a) + \alpha \left(r + \gamma \max_{a'} Q(s',a') - Q(s,a) \right) \tag{1}$$

where s is the current state, a is the action taken, r is the reward received, s' is the next state, a' is the action in the next state, α is the learning rate, and γ is the discount factor.

4.1.1 Learning Rate Hyperparameter Finetuning

As part of our investigation, we experimented with different learning rates to observe their effect on agent performance. The results for a learning rate of 5×10^{-4} are presented in our Results section to highlight hyperparameter sensitivity.

4.1.2 Prioritized Experience Replay (PER)

Our Deep Q-Network (DQN) agent, implemented in **dqn.py**, utilizes Prioritized Experience Replay (PER) [3] to improve learning efficiency and performance. Unlike standard experience replay which samples transitions uniformly, PER gives priority to transitions that the agent can learn most from, typically those with a high temporal difference (TD) error. This focused sampling accelerates learning by replaying important transitions more frequently.

The core of our PER implementation resides in the *PrioritizedReplayBuffer* class within **dqn.py**. This class uses a *SumTree* data structure, a specialized binary tree that enables efficient O(log N) updates of priorities and sampling of experiences. When a new transition (composed of state, action, reward, next state, and done flag) is added to the buffer using the *push* method, it is initially assigned the current maximum priority recorded in the tree. This ensures that all new experiences are considered for sampling at least once with high importance until their actual TD error is computed.

The priority p_i for a given transition i is determined by its absolute TD error δ_i. Specifically, $p_i = (|\delta_i| + \epsilon)^\alpha$, where α (set to 0.6 in our *DQNAgent*) is a hyperparameter controlling the degree of prioritization ($\alpha = 0$ reverts to uniform random sampling). The small constant ϵ (1e-5 in our *DQNAgent*) is added to ensure that transitions with zero TD error still have a non-zero probability of being sampled. After a batch of transitions is processed and their TD errors are computed, the *update_priorities* method is called to refresh their respective priorities in the *SumTree*. The *max_priority* attribute of the buffer is dynamically updated to track the highest priority seen so far.

Sampling from the *PrioritizedReplayBuffer* is handled by the *sample* method. It draws a batch of transitions where each transition's probability of being selected is proportional to its stored priority p_i. To correct for the bias introduced by this non-uniform sampling strategy, Importance Sampling (IS) weights are computed for each sampled transition. The IS weight w_i is given by $w_i = (N \cdot P(i))^{-\beta}$, normalized by $1/\max_j(w_j)$, where N is the total number of items in the replay buffer and $P(i)$ is the probability of sampling transition i. The hyperparameter β (annealing from *beta_start*=0.4 to 1.0 over *beta_frames*=250,000, as managed by *_beta_by_frame*) controls the amount of correction. These IS weights are then used to scale the loss for each transition in the batch during the DQN's learning update (when *use_per* is true, *nn.SmoothL1Loss* is used with its *reduction* parameter set to 'none' to facilitate this). This ensures that the parameter updates remain unbiased despite the prioritized sampling. The *DQNAgent* class seamlessly integrates this PER mechanism when its *use_per* parameter is set to true.

4.2. Expectimax Search

The file **expectimax.py** implements a depth-limited expectimax search that chooses the next move in 2048 by alternating between "max" nodes (our moves) and "chance" nodes (random tile spawns). It begins in *expectimax_search*, which masks out illegal moves, simulates each legal slide via *simulate_move*, and accumulates the immediate reward plus the expected future value returned from *chance_node_value*. If a slide doesn't change the board, it still explores that branch by descending into the chance node, penalizing unproductive moves implicitly by having their follow-on evaluations unchanged.

At the player's decision nodes, *max_node_value* examines each legal direction, caches results by hashing the board and depth, and returns the highest reward-plus-chance value. It stops recursing when the depth limit is reached or when no moves remain, at which point it calls *evaluate_board*. In similar fashion, *chance_node_value* enumerates all empty cells, spawns a "2" tile with 90% probability and a "4" tile with 10%, divides by the number of empty cells, and sums those weighted values. Memoiza-

tion in both functions keeps repeated subtrees from being recomputed.

Terminal states are detected by *is_terminal*, which simply checks that no legal moves exist. When the recursion bottoms out or the board is terminal, *evaluate_board* computes a weighted heuristic: it adds the sum of tile values, the count of empty cells, a monotonicity score (how consistently increasing the tiles are along rows and columns), a smoothness penalty (differences between neighbors), and the exponent of the largest tile. Helper functions *calculate_monotonicity* and *calculate_smoothness* quantify those spatial patterns so that the heuristic pushes toward empty, ordered, and clustered boards. For example, monotonicity can be calculated by summing the differences between adjacent tiles in increasing sequences along rows and columns, and smoothness by penalizing differences between neighboring tiles. While our implementation uses a weighted sum of these and other features (empty cells, max tile value), conceptual formulas can be represented as:

$$S_{\text{mono}}(B) = \sum_{r \in \text{rows}} \sum_{i=1}^{3} \mathbb{I}(B_{r,i} \leq B_{r,i+1}) \tag{2}$$

$$+ \sum_{c \in \text{cols}} \sum_{j=1}^{3} \mathbb{I}(B_{j,c} \leq B_{j+1,c}) \tag{3}$$

$$S_{\text{smooth}}(B) = - \sum_{\text{adjacent pairs } (t_1, t_2)} |\log_2 t_1 - \log_2 t_2| \tag{4}$$

where B is the board state and $\mathbb{I}(\cdot)$ is the indicator function. These specific formulas are illustrative of the concepts.

Finally, when run as a script, the module demonstrates itself on sample, empty, and full boards by printing the tile grid, the best action at a given search depth, and the heuristic score. This organization cleanly separates game logic (*legal_move_mask*/*simulate_move*) from search control flow, memoization, and evaluation heuristics.

5. Results

In this section, we present the performance of our implemented agents and the outcomes of our hyperparameter tuning experiments. We evaluated agents based on mean score, median score, mean episode length, and the highest tile achieved.

5.1. DQN Agent Performance

We conducted several experiments to evaluate and improve our Deep Q-Network (DQN) agent. The baseline "Vanilla DQN" uses a standard experience replay buffer. We then implemented "Prioritized Experience Replay (PER)" to assess its impact. Finally, we performed "Hyperparameter Finetuning," focusing on the learning rate (LR); the results presented here are for a sample learning rate of 5×10^{-4}. As Table 1 shows, the Vanilla DQN attains a mean score of 93.52, whereas the PER variant actually underperforms with a mean score of 81.44. The learning rate finetuning experiment resulted in the lowest performance. These results highlight the sensitivity of DQN performance to both architectural choices like PER and fundamental hyperparameters like the learning rate.

Table 1. Performance of DQN-based agents. "Best Tile" indicates the highest tile value achieved.

Experiment	Mean Score	Median Score	Best Tile
Vanilla DQN	93.52	76.0	2^8 (256)
PER	81.44	60.0	2^7 (128)
LR Finetuning	54.80	44.0	2^5 (32)

To understand these performance differences, we analyzed the training dynamics. Figure 2 plots the TD error curve for the Vanilla DQN. The curve shows a relatively smooth decrease in TD error, which begins to plateau around 4,000 episodes, suggesting convergence of the value estimates. This stable learning trajectory likely contributes to its better performance compared to the other DQN variants.

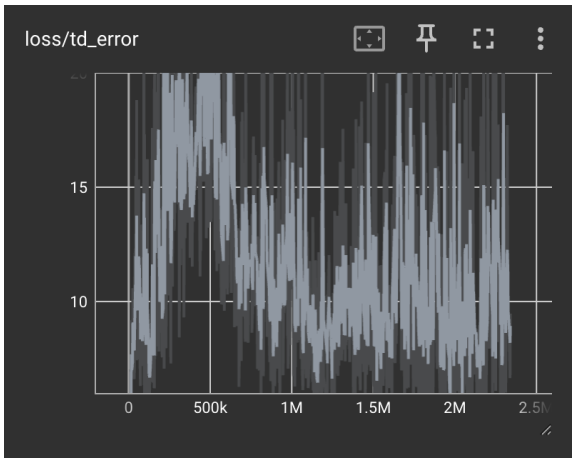

Figure 2. TD error for the Vanilla DQN decreases sharply and then plateaus around 4,000 episodes, indicating convergence in value estimates and correlating with its relatively stable performance.

In contrast, Figure 3 depicts the TD error for the DQN with PER. This curve exhibits larger oscillations, particularly around the 2,000-episode mark, and a less consistent decline compared to the Vanilla DQN. Such instability might suggest that the prioritization mechanism, perhaps due to the chosen α and β parameters, led to less stable learning, which could explain its lower final scores.

Finally, Figure 4 shows the TD error for the DQN with a higher learning rate (5×10^{-4}). The error remains con-

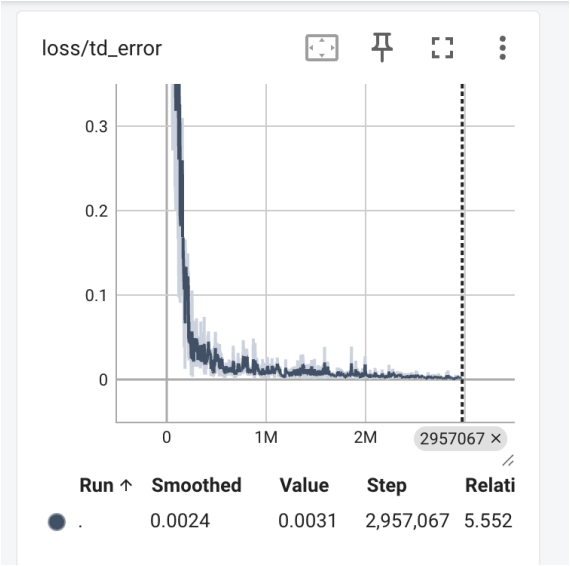

Figure 3. The TD error curve for DQN with PER shows notable oscillations, especially around 2,000 episodes, suggesting learning instability that may have contributed to its lower scores compared to Vanilla DQN.

sistently high throughout training and shows erratic behavior without a clear downward trend. This indicates that the learning process failed to converge effectively, directly explaining its significantly lower performance.

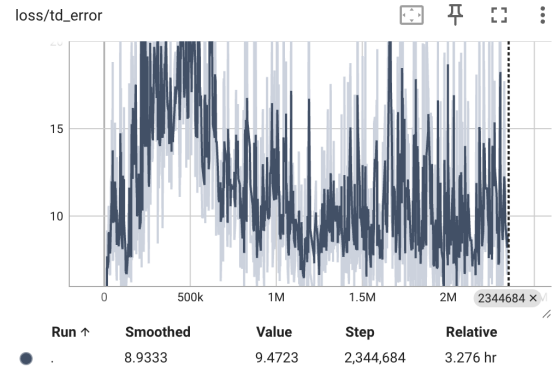

Figure 4. The TD error for the DQN with a learning rate of 5×10^{-4} remains high and erratic, indicating a failure to converge and explaining its poor performance.

5.2. Expectimax Agent Performance

The Expectimax agent was evaluated across different search depths, from 1 to 5. Each depth was tested for 20 episodes. Table 2 details the Expectimax agent's performance. As the table illustrates, mean scores improve with search depth up to depth 4 (458.2), after which depth 5 shows a notable decrease (356.0). We observed a general trend of improving scores with increased depth up to depth 4. Performance notably decreased at depth 5. This is likely due to the exponential increase in computation time with depth; for instance, evaluating 20 episodes at depth 5 took over 13 hours, compared to approximately 1.6 hours for depth 4. Such long computation times might lead to practical limitations in exploring the game space effectively for very deep searches. The 'Max Tile Achieved' is derived from the *max_tile_exponent* provided in the logs.

Table 2. Performance of Expectimax agent at different search depths (20 episodes each).

Depth	Mean Score	Median Score	Max Tile Achieved
1	91.0	74.0	2^5 (32)
2	297.0	192.0	2^7 (128)
3	412.8	324.0	2^7 (128)
4	458.2	364.0	2^7 (128)
5	356.0	182.0	2^8 (256)

6. Discussion

Our experiments provide insights into the performance of different agents on the 2048 game.

For the DQN-based approaches, the Vanilla DQN served as a baseline. Interestingly, the introduction of Prioritized Experience Replay (PER) did not yield an improvement in mean or median scores over the Vanilla DQN in our setup (81.44 vs. 93.52 mean score). This outcome is somewhat counter-intuitive, as PER is generally expected to accelerate learning and improve performance by focusing on more significant transitions. This might suggest that the PER parameters (α, β, ϵ) were not optimally tuned for this specific task, or that the benefits of PER might only become apparent with a much larger number of training episodes or a different network architecture. The hyperparameter fine-tuning experiment, which tested a learning rate of 5×10^{-4} (shown as "LR Finetuning"), resulted in lower performance compared to both Vanilla DQN and PER. This highlights the sensitivity of DQN agents to hyperparameter choices and underscores the need for more extensive tuning to find an optimal configuration. For PER specifically, we suspect our chosen $\alpha = 0.6$ may have overweighted rarer or noisy transitions, leading to instability. Future work should systematically sweep PER hyperparameters, for instance $\alpha \in \{0.4, 0.5, 0.7, 0.8\}$ and different β annealing schedules (e.g., slower annealing to 1.0), to determine if a more conservative prioritization scheme can narrow the performance gap or surpass the Vanilla DQN.

The Expectimax agent demonstrated a clear trend of improved performance with increasing search depth, up to depth 4, which achieved the highest mean score (458.2). However, performance notably declined at depth 5 (mean score 356.0), despite achieving the highest single tile ($2^8 =$

256) among all Expectimax runs. This decline is strongly correlated with the exponential increase in computation time. Evaluating 20 episodes at depth 5 took over 13 hours, compared to approximately 1.6 hours for depth 4. Such prolonged computation times for deeper searches can become a practical bottleneck. The choice of heuristic function for Expectimax is also critical, and while our current heuristic (based on monotonicity, smoothness, free tiles, and max tile) performs reasonably, its optimality is an area for further investigation. The performance drop at depth 5, despite longer computation, suggests that the sheer time taken per move (over 13 hours for 20 episodes) may have led to practical limitations in how effectively the search could be conducted, possibly due to implicit pruning or an inability to explore sufficient breadth at deeper nodes within reasonable timeframes. The term *max_tile_exponent* in the logs was used to derive the 'Max Tile Achieved'.

Contextualizing our best Expectimax performance, the mean score of 458.2 at depth 4 is respectable but lower than some highly optimized search-based agents reported in the literature. For example, Tiwari et al. [5] reported scores over 600 with a beam-search hybrid. This difference could be attributed to our heuristic's relative simplicity, which, for instance, does not explicitly prioritize forming smooth paths along the grid edges or corners, a common strategy in high-performing 2048 agents.

Comparing the two classes of agents, the Expectimax agent, particularly at depths 3 and 4, significantly outperformed all our DQN implementations in terms of average scores. This is not entirely surprising, as tree search methods can often achieve strong performance in games with well-defined state transitions and effective heuristics, by looking ahead multiple steps. DQN agents, on the other hand, learn a generalized policy from experience, which can be powerful but often requires substantial data and careful tuning.

Several challenges were encountered during implementation. For instance, debugging the SumTree logic for Prioritized Experience Replay required careful attention to indexing and priority updates. Similarly, ensuring the correctness and efficiency of the caching mechanism in the Expectimax agent was crucial for deeper searches.

The comparison between DQN and Expectimax also offers broader insights into the trade-offs between model-free and model-based approaches in stochastic environments. While model-free agents like DQN promise generality, they can be sample-inefficient and highly sensitive to hyperparameters. Model-based methods like Expectimax, when a reasonable model or heuristic is available, can achieve strong performance with less data but may struggle with scalability or require significant domain knowledge for heuristic engineering.

Limitations of this study include the relatively limited scope of hyperparameter tuning for the DQN agents and the fixed number of evaluation episodes (20) for the Expectimax agent, which might not capture the full variance in performance. Furthermore, the exploration strategy for DQN and the specific implementation details of PER could be further refined.

7. Conclusion

In this project, we implemented and evaluated several agents for the game 2048, including Deep Q-Network (DQN) variants and an Expectimax search agent. Our results indicate that the Expectimax agent, when configured with sufficient search depth (specifically depths 3 and 4), achieved superior performance compared to the DQN agents in terms of average game scores. The Expectimax agent's performance generally scaled with search depth, but was ultimately constrained by a significant increase in computational cost at higher depths (e.g., depth 5).

Among the DQN approaches, the Vanilla DQN unexpectedly outperformed our implementation of Prioritized Experience Replay (PER) and a DQN with a modified learning rate (LR Finetuning). This suggests that further tuning of hyperparameters, network architecture, and PER-specific parameters is necessary to unlock the full potential of these learning-based methods for 2048. The project highlights the challenges in applying reinforcement learning to 2048 and underscores the effectiveness of traditional search methods when coupled with good heuristics and sufficient computational budget.

Additionally, there are several promising avenues for future work to build upon the findings of this project.

For the DQN agents, a more comprehensive hyperparameter optimization is warranted. This includes exploring a wider range of learning rates, different network architectures (e.g., deeper or wider layers, convolutional layers to better capture spatial patterns on the board), adjustments to the exploration-exploitation strategy (e.g., ϵ-decay schedule), and fine-tuning PER parameters (α, β). Investigating other advanced DQN extensions, such as Double DQN or Dueling DQN, could also lead to performance improvements. Longer training durations, potentially leveraging more computational resources, would allow the agents more experience to learn effective policies.

Regarding the Expectimax agent, further development could focus on designing more sophisticated heuristic functions. While the current heuristic considers several board features, incorporating machine learning to learn a heuristic, or using more complex handcrafted features, might enhance its evaluation capabilities. Optimizing the search algorithm itself, perhaps through more aggressive pruning techniques suitable for Expectimax or adaptive search depth control, could also improve its efficiency.

Exploring other types of agents, such as Monte Carlo

Tree Search (MCTS), which has shown success in similar games, or policy gradient methods, could provide alternative approaches to tackling 2048. A more extensive evaluation framework, involving a larger number of episodes and comparison against established 2048 AI benchmarks, would provide a more robust assessment of agent performance.

Finally, hybrid approaches that combine the strengths of learning and search could be particularly effective. For instance, a DQN could be used to learn a value function that serves as a heuristic for a shallower tree search, potentially offering a better balance between performance and computational cost.

8. Team Contributions

Soham Konar: Implemented and ran experiments for Expectimax. Wrote the report. Helped with the poster.

Danny Park: Implemented and ran experiments for Vanilla DQN and LR Finetuning. Did research on existing methods. Wrote the report. Created the poster.

Andrew Wu: Implemented and ran experiments for PER. Did research on existing methods. Wrote the report. Helped with the poster.

Here is the GitHub repository link for our codebase: https://github.com/sohamkonar/CS224R_Final_Project

References

[1] A. Goga. Reinforcement learning in 2048 game. *Bachelor thesis of Faculty of Mathematics, Physics and Informatics, Comenius University in Bratislava*, 2018. 2

[2] H. Guei. *On Reinforcement Learning for the Game of 2048*. PhD thesis, National Yang Ming Chiao Tung University, 2023. 2

[3] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015. 3

[4] M. Szubert and W. 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. 2

[5] V. Tiwari. From pixels to plans: Cracking 2048 with reinforcement learning and beam search. Medium post, Mar. 2025. Accessed 2025-06-09. 2, 6