Comparing AI Agents in 2048: A Performance and Strategy Analysis

Computer Science 480: Artificial Intelligence

Professor Kirk Duran

June 2, 2025

Content created, written, and displayed by:

Lucas Summers (lsumme01@calpoly.edu)
Megan Fung (mfung06@calpoly.edu)
Braeden Alonge (balonge@calpoly.edu)
Nathan Lim (nlim10@calpoly.edu)

1. Abstract

This project aims to develop an AI agent capable of achieving the highest possible score in the single player puzzle game 2048. We will experiment with a variety of AI techniques, including Expectimax search, Monte Carlo Tree Search (MCTS), Reinforcement Learning, and Hybrid methods that combine multiple techniques into one. Our bot will dynamically decide optimal moves at each game/board state by predicting future outcomes and maximizing reward. We aim to develop models that are able to consistently beat the game (reach the 2048 tile) and get as close to the 131,072 tile as possible, the theoretical limit of a 4x4 board. Through this project, we hope to explore how AI planning and decision-making algorithms perform under uncertain, partially observable environments.

---

2. Motivation

Games like 2048 offer an ideal environment for experimenting with AI techniques through having to balance strategy, planning, and randomness. Building an AI that can consistently achieve high scores in 2048 is a challenging way to apply search algorithms, heuristic designs, and reinforcement learning techniques, all while revealing how each different AI algorithm compares with one another. Success in this project would demonstrate practical mastery of game playing AI techniques, and deepen our understanding of how artificial agents can make decisions with incomplete information. It also provides a demonstration of how different models compare in the context of 2048 and games alike, potentially helping developers to choose appropriate AI algorithms and techniques to implement in their own environment, as well as which to avoid.

---

3. Background

Previous work on 2048 AI agents has employed several strategies, including (see bibliography for full list of referenced work):

- **Expectimax Search**: A probabilistic extension of Minimax that models random tile spawns and maximizes the expected utility of a move. Matt Overlan's 2014 bot proved that a depth-3 search, backed by corner-max and empty-cell heuristics, can go as far as to touch the 4096 tile.
- **Monte Carlo Tree Search (MCTS)**: A popular approach in stochastic games that balances exploration and exploitation through simulated rollouts, using a statistical framework to guide decision-making. It is noted for its adaptability and relatively low reliance on handcrafted

heuristics. Basic MCTS is shown in research to fail to top Expectimax strategies because random rollouts tend to be too noisy and variant, but Watanabe & Matsuzaki's 2022 CNN-guided version improved upon that variance and was able to compete with scores from Expectimax.

- **Reinforcement Learning (RL)**: allows agents to learn optimal strategies from scratch through self play. Models like Deep Q Networks and policy gradient methods have been applied to 2048 with promising results, though they require significant training time and computational resources. Yulin Zhou's AlphaGo-Zero style pipeline was able to eventually hit the theoretical max tile of 131,072, but this was only possible after weeks of massive computation time. Hung Guei's DQN was able to compete with Expectimax after around 100,000 episodes with fine-tuned reward shaping.
- **Hybrid Approaches**: combines the structural strengths of tree search with the flexibility of learned policies and heuristics. These models often use neural networks in the rollout phases of MCTS or as evaluators in Expectimax.

Challenges faced by these models include the computational cost of deep searches, the difficulty of crafting effective heuristic evaluation functions, and the delicate balance between exploration and exploitation in RL settings.

---

4. Problem Statement & Impact

Despite the popularity of the 2048 puzzle game, AI agents that can reliably attain high scores remain to be a complex challenge because 2048 is a game of stochastic nature and a largely vast search space. Our main goal is to design and compare multiple AI-based strategies, including Expectimax, Greedy, MCTS, Reinforcement Learning (RL), and hybrid approaches, to determine which is the most effective under limited computational budgets. This research contributes to the understanding of decision-making in stochastic environments with applications extending to broader AI fields such as game theory, planning under uncertainty, and reinforcement learning-based agents.

---

5. Relevance

This project is highly relevant to the core topics covered in class. It directly applies various AI techniques discussed in class, including search-based algorithms, heuristic-based decision-making, as

well as reinforcement learning. Specifically, the Expectimax and Monte Carlo Tree Search agents represent advanced applications of heuristic and probabilistic search strategies. The project also engages with stochastic and adversarial planning, as the game of 2048 involves randomness through tile spawning and complex state evaluations. Additionally, the reinforcement learning component allows us to explore agent training through reward over time, aligning with class topics such as Q-learning and policy optimization. By developing and evaluating these agents in a real world environment, the project reinforces theoretical concepts and illustrates how AI can make decisions in uncertain and partially observable domains.

---

6. Methods

6.1 Setup

We wanted to create a modular design for both the agents and heuristics to be able to easily plug in and test different agents and configurations. Using the same visual aesthetic as the original online game, we integrated our several AI algorithms into a local 2048 applet (shown below). This applet provides a number of visual features, including game attributes (current score, best score, highest numbers, etc.), agent statistics, and an interactive visual of the moves that the algorithm makes. Most importantly, the applet includes the ability to modify which algorithm to run as well as the ability to stop the algorithm, change it, and begin a new game. The user can also stop the AI agent from running and play the game normally at any point.

6.2 Heuristics

We implemented numerous board heuristics to effectively evaluate the current game state beyond just the score. This is because 2048 games can have thousands of moves before terminating, and so both Expectimax and MCTS will almost never reach a terminal state in a reasonable time. Thus, we need heuristics that can evaluate a game state based on how conducive the board is to a high scoring game. With weights, we then combine these heuristics into one composite heuristic to be used for evaluation. Each factor captures a desirable strategy that is used by expert human players. For example, keeping the highest tile in one of the corners allows for easier stacking and prevents premature merges.

The heuristics are as follows:

- **Empty Tile**: Rewards boards with more empty spaces. Having more empty tiles provides greater flexibility for future moves and reduces the likelihood of premature game-over. To reward having access to crucial positions, it also weights each empty tile by its position on the board, where empty tiles on the corners are more desirable than empty tiles on the edges, which are more desirable than empty tiles in the center.
- **Monotonicity**: Encourages tile ordering in increasing or decreasing sequences along rows and columns. Monotonic boards are easier to merge and expand. The heuristic simply scores each

row/column based on how ordered the tiles are, allowing for partial ordering and penalizing large breaks in ordering.

- **Smoothness**: Measures how similar adjacent tiles are in value, which enables cascades of multiple merges. Boards where neighboring tiles:
    - Differ in large amounts → *Penalized*
    - Gradual value transitions → *Rewarded*
- **Max Value**: Rewards boards with higher maximum tile values, promoting long term growth toward the 2048+ tiles. The score is proportional to the logarithm (base 2) of the max tile.
- **Corner Max**: Encourages strategic placement of the highest tile in a corner. This is a common expert tactic that keeps high value tiles stable and stackable. The heuristic gives:
    - If max tile is in a corner → *Larger reward*
    - If max tile is on an edge → *Smaller reward*
    - If max tile in the center → *Penalty*
- **Merge Potential**: Evaluates how many tile pairs on the board are immediately mergeable, either horizontally or vertically. It encourages boards with multiple merge possibilities, favoring configurations that maximize scoring per move.
- **Quality/Stability:** A more complex strategy that measures how stable, or resistant to quality loss, the board is. It essentially checks how protected the max tile is, or how many sides are exposed to empty cells, as well as rewards having the second-highest tile near the max tile. This captures the concept of avoiding "trap" positions: board configurations might look good now, but are vulnerable to future tile spawns that create an obstacle and force many suboptimal moves. Well-protected configurations with adjacent large tiles are able to absorb these disruptive tile spawns.

Heuristic	**Weight**
Empty tile	3.0
Monotonicity	3.0
Smoothness	1.5
Max value	1.0
Corner max	5.0
Merge potential	2.5

Quality/Stability	2.0

We had originally planned to use a reinforcement learning based approach to iteratively tune the heuristic weights based on performance. However, due to long run times for both MCTS and Expectimax, this became unreasonable. Thus, we hardcoded weights based on our research on effective strategies and opinions on what strategies should be prioritized.

6.3 Random

As a baseline for evaluating performance, the random agent selects moves randomly at each game state. This agent is a crucial component of our agent evaluation as it serves as a useful lower bound against our other agents: Expectimax, MCTS, RL, and hybrid. Its random decision-making behavior helps highlight the importance of strategic planning and game state evaluation in stochastic environments like 2048.

6.4 Greedy

This agent is also a baseline, but with some added sophistication above the random agent. The greedy agent evaluates all available moves and selects the one that yields the highest short term gain. Its evaluation score combines two key factors in a weighted combination:

- the log base 2 of the *resulting maximum tile* on the board (weight = 1.0)
- the *score increase* from the move (weight = 0.1)

Unlike deeper search based agents, this agent performs a single step look ahead, making it fast but also very shortsighted.

6.5 MCTS

The MCTS agent makes decisions by simulating many possible future game trajectories within a fixed time budget ("thinking time"). It follows the classical four phase MCTS process: Selection, Expansion, Simulation, and Backpropagation, using the UCB1 formula to balance exploration and exploitation during tree traversal.

- **Selection**: Agent walks down the search tree by selecting child nodes with the highest UCB1 score until it reaches a leaf node (standard MCTS exploration rate of $C = \sqrt{2}$ used).
- **Expansion**: If the leaf node selected is not terminal and has unvisited children, a new node is added for an unexplored move. The node whose move has the least quality loss (difference between evaluation score before and after the move) is preferred first.

- **Simulations**: Rollouts are then carried out from the expanded node to a fixed depth or until a terminal state is reached. The rollout policy can be either random (with a fixed depth of 50 moves), Expectimax based (see section 6.8), or RL based (see section 6.9). The final state, whether terminal or not, is then scored using a composite heuristic function (see section 6.2).
- **Backpropagation**: The result is back-propagated up the tree. After the time budget is exhausted, the agent chooses the move from the root node that has the most visits, which is the standard MCTS selection policy.

NOTE: **The MCTS main loop** is provided in the appendix (see section 13.2).

6.6 Expectimax

Expectimax's strength lies in its balance of deterministic planning and probabilistic modeling, which makes it a great baseline for performance in stochastic games like 2048. The Expectimax agent simulates future game states by exploring possible player moves and random tile placements, incorporating strategic optimizations. The agent models the 2048 game as a stochastic decision tree with two alternating node types:

- **Max nodes** (the player's turn): Chooses the move that maximizes quality loss first, then maximizes expected value.
    - NOTE: quality loss is simply the expected value before the move subtracted from the expected value after the move
- **Chance nodes** (random tile spawns): Average over strategically selected spawn locations
    - 90% chance a new 2-tile spawns
    - 10% chance a new 4-tile spawns

Our original Expectimax agent, which used a fixed depth of 3, performed decently in early benchmarks, consistently reaching the 1024 tile in controlled tests. However, due to the exponential computational cost with each additional layer of depth, the algorithm requires more time each move to explore a deeper game tree and produce a more informed decision. In order to combat this, we implemented several optimizations to manage the exponential branching factor as well as allow the agent to make better decisions at shallower depths.

We implemented a recursive Expectimax algorithm that performs progressive deepening, where at each iteration the agent searches to a greater depth until the allotted time budget ("thinking time") is exceeded. At each move, the agent examines legal directions (left, right, up, or down), simulates the resulting game state, and recursively estimates the expected score and potential quality degradation

through simulation of spawned tiles. We prune tile spawn locations by only testing empty tiles adjacent to existing tiles, not all empty tiles. This reduces the branching factor drastically, speeding up the time it takes to generate a move. Also, unlike traditional Expectimax that solely maximizes expected value, our enhanced implementation employs a quality-loss-aware selection process, where moves are chosen first to minimize potential degradation in board quality (worst case), with expected value serving as a tiebreaker. In 2048, it's important to maintain a quality board layout, as bad layouts have a cascading effect that can be hard to correct at a certain point. Thus, this conservative approach dramatically improves consistency in reaching high-value tiles by attempting to preserve a good board structure for better future moves.

NOTE: **The fixed-depth Expectimax snippet** is provided in the appendix (see section 13.1).

6.7 Reinforcement Learning

The reinforcement learning agent uses a DQN (deep Q network) to learn optimal move policies from gameplay experience. The 4 by 4 game board is flattened into a 16 element vector, with tile values log scaled to compress the input range. Empty cells remain as 0. The log-scale compression helps to keep inputs in a narrow numeric range that correspond to the “level” of the tile rather than the raw value. This vector is fed into a neural network that outputs Q values (expected future rewards) for each of the four possible actions (up, down, left, and right).

During training, the agent uses **ε greedy exploration**, choosing random actions with decaying probability to balance exploration and exploitation. Game transitions are stored in a replay buffer, and the network is trained using batches sampled from this buffer to minimize the difference between predicted and target Q values. A target network is used to stabilize learning.

Training was done with multiple models trained between 25,000 and 50,000 episodes. Every environment step yields a shaped reward based on our previously defined heuristics, and the agent evaluates itself every 500 episodes and completes 100 games with ε=0 to evaluate its progress in learning.

A learning rate manager was also implemented that monitors the evaluations to lower the learning rate if the average score doesn’t improve for three consecutive evaluations in a row, helping to maintain fast early learning and more accurate fine-tuning as training continues.

In addition to using the previous defined heuristics as part of the overall reward function (empty tile, monotonicity, smoothness, max tile, corner max, and merge potential), we also implemented a part of

the reward function to account for snake-order, where values decrease along an S-shaped path on the board and do not oscillate within rows or columns, which is a generally-optimal approach to playing 2048. The reward function also has a loss penalty for when the game terminates in a loss.

The RL agent operates with a **single step look ahead**, relying on the Q network to estimate long term reward. While slower to converge and more complex than heuristic based agents, the RL approach learns from experience and has the potential to discover novel strategies beyond those encoded by heuristics.

NOTE: **The core DQN training loop** is provided in the appendix (see section 13.3).

6.8 Hybrid– MCTS with Expectimax Rollouts

Our hybrid agent combines MCTS with Expectimax to create a powerful decision-making strategy that leverages the strengths of both approaches. While MCTS offers efficient exploration of the game tree using statistical sampling, Expectimax provides a deeper, more structured look ahead during simulations. This hybrid approach allows for more accurate evaluations without fully expanding the tree.

The hybrid agent follows the standard MCTS framework:

- **Selection**: Traverses the tree using the UCB1 formula to balance exploration and exploitation.
- **Expansion**: Add a new child node for a previously unexplored move. Again, moves with the least quality loss are preferred.
- **Simulation:** Instead of simulating random moves, the hybrid agent invokes the Expectimax algorithm at a fixed shallow depth (depth = 3) to make sure we don't exceed the time budget.
- **Backpropagation**: Propagate the Expectimax derived reward up the tree.

This integration allows MCTS to make rollout evaluations that are smarter than purely random simulations Some advantages are:

- **Deeper insight**: Expectimax provides a look ahead that accounts for stochastic tile placements, giving more realistic outcome estimations during rollouts.
- **Improved accuracy**: Expectimax based rollouts reduce variance in value estimation compared to random simulations.
- **Remain within time budget**: The agent can still operate within its time budget given the shallow depth of the Expectimax agent.

7. Study Design

7.1 Game Environment

Our implementation of the 2048 environment replicates the classic game logic but is designed specifically to support **AI agent interaction and experimentation**. Basic game environment is composed of the Board class and Game2048 class:

- The **Board** class handles the core tile mechanics. It maintains a 4 by 4 grid and supports movement, tile merging, and spawning of new tiles. It also computes valid moves, checks for terminal states, and provides tile statistics.
- The **Game2048** class tracks score, number of moves made, and whether the game has ended. It also supports state copying, which is a crucial function for our AI agents to simulate future actions without altering real game state.

To support **live testing and visualization**, we've implemented a Flask base server that interacts with the 2048 game environment through API calls and allows users to run agents, view gameplay in real time, and interact with the game through an applet (see section 6.1). Agents can be selected, swapped, paused, or restarted mid-game. Each AI agent interacts with the environment through:

1. Getting available moves from the current game state (Game2048 class)
2. Simulating moves using copies of the game state, retrieving state features such as score, max tile, empty cell count, etc.
3. Making an **informed decision** accordingly, changing the game state, which is reflected in the applet

7.2 Agent Performance Analysis

To rigorously evaluate and compare the performance of our 2048 AI agents, we developed a framework to simulate games and collect and visualize the results. Our analysis framework leverages multi-level parallelization to efficiently conduct hundreds of game simulations. It implements a two-tier parallel processing architecture:

- **Process-level:** Distributes game batches across all available CPU cores, maximizing hardware utilization.
- **Batch processing:** Each process manages configurable batches of games to balance memory usage and parallelization overhead.
- **Thread-level:** Within each batch, the system dynamically allocates threads to run each game. If the number of threads is equal to the batch size, all games in the batch run in parallel. If the number of threads is 1, games run sequentially.

Results for each agent are collected progressively in a JSON format for analysis and visualization later (see appendix section 13.4 and 13.5 for example). This allows processes to store data as soon as games complete, allowing for real-time progress monitoring as well as early termination. The structure also allows results to be easily combined from multiple simulation runs. For each game an agent completes, we record the final score, max tile reached, number of moves made, and the time the game took to run. We use this data to then compute the following agent evaluation criteria:

- Agent name and configuration used
- Minimum, maximum, and average scores
- Win rate (percentage of games reaching the 2048+ tile)
- Absolute and median max tiles reached
- Efficiency (final score divided by average number of moves in a game)
- Move distribution (percentage of left, right, up, and down moves)

Visualization tools will then be used to compare performance metrics, decision behavior, and computation time (see section 8 for results).

7.3 Experimental Design

We will compare the following agent configurations:

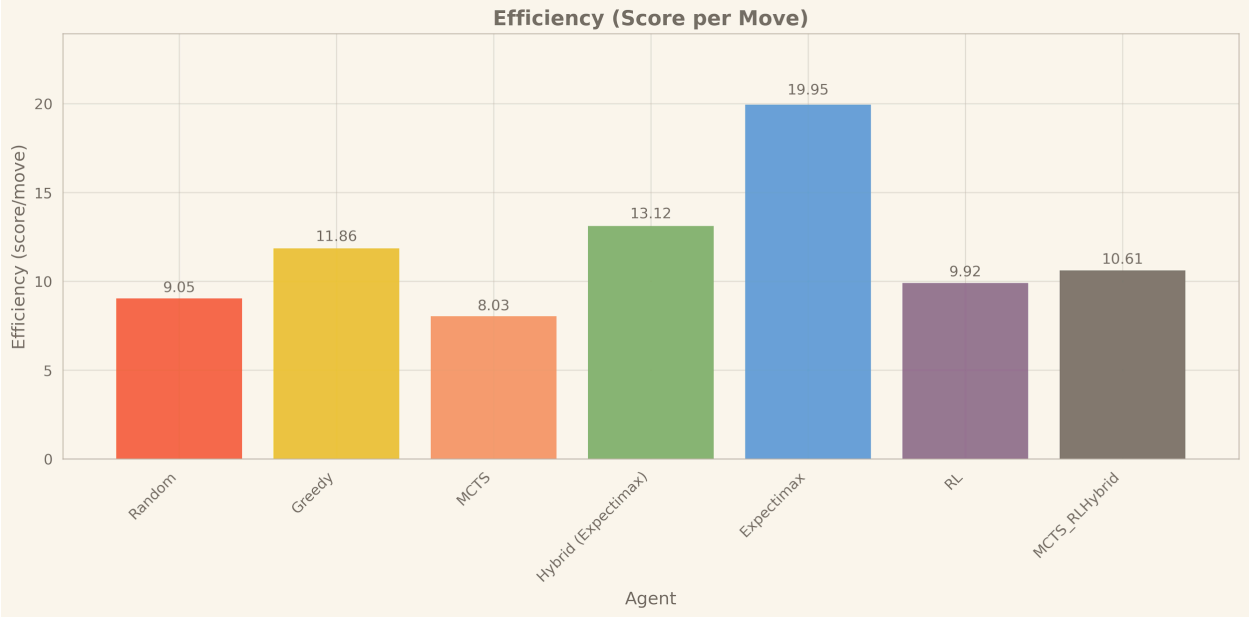
- Random Agent (baseline)
- Greedy Agent (baseline)
    - Tile weight = 1.0
    - Score weight = 0.1
- Expectimax Agent
    - Thinking time = 1.0 s
- MCTS Agent (w/ random rollout)
    - Thinking time = 1.0 s
    - Exploration weight = 1.414
- Reinforcement Learning Agent
- Hybrid (MCTS + Expectimax rollout)
    - Thinking time = 1.0 s

Each agent will play 100 games with a fixed random seed for reproducibility. Standardizing the time budget, or “thinking time”, for the search-based algorithms allows us to create an even playing field to be able to accurately compare performance. Note that the other agents do not need a time budget because they produce outputs almost immediately.

8. Results

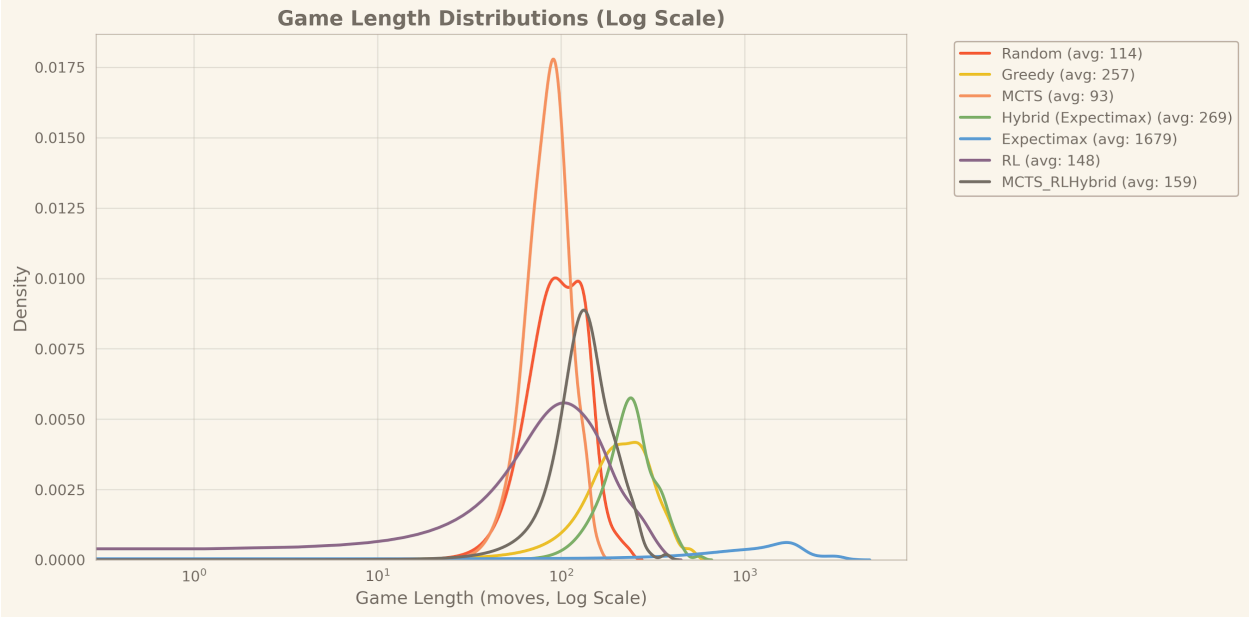
To evaluate the relative performance of our agents, we conducted large scale simulations and collected data across several key metrics:

- Efficiency (score per move)
- Game Length Distribution
- Max Tile Distribution
- Movement Behavior
- Score Distribution
- Win Rate

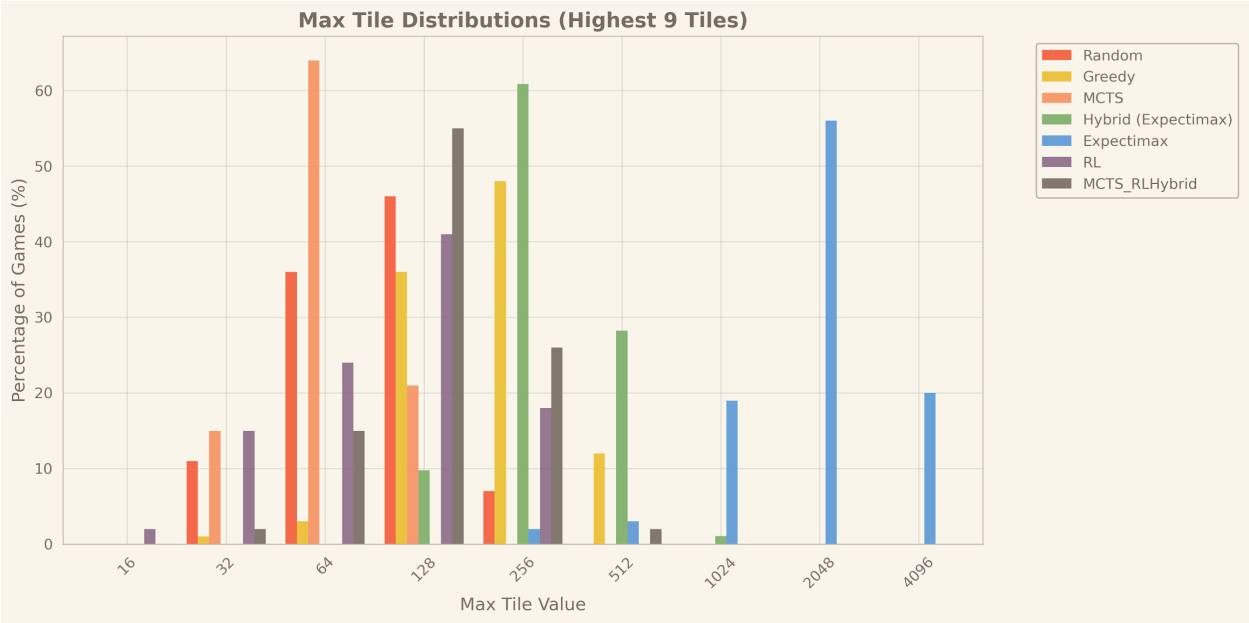


The **efficiency plot** measures average score per move. The results allow us to rank the effectiveness of each agent's decision-making process. Demonstrating productivity in long term planning with heuristics, **Expectimax scores 19.95 per move**. Coming in at second, **MCTS with Expectimax rollouts** scores 13.12 per move, suggesting that probabilistic sampling paired with smarter simulations can boost efficiency. At third, **Greedy scores 11.86 per move**, on average, highlighting that fast decision-making and short term planning can accomplish mid-tier efficiency, but optimal efficiency requires foresight. Ranking fourth in efficiency, our **MCTS with RL rollouts agent (10.61)** outperforms our **standalone RL agent (9.92)**, suggesting that while learned policies offer some guidance, they fall short of delivering high-yielding decisions without the structured balance of

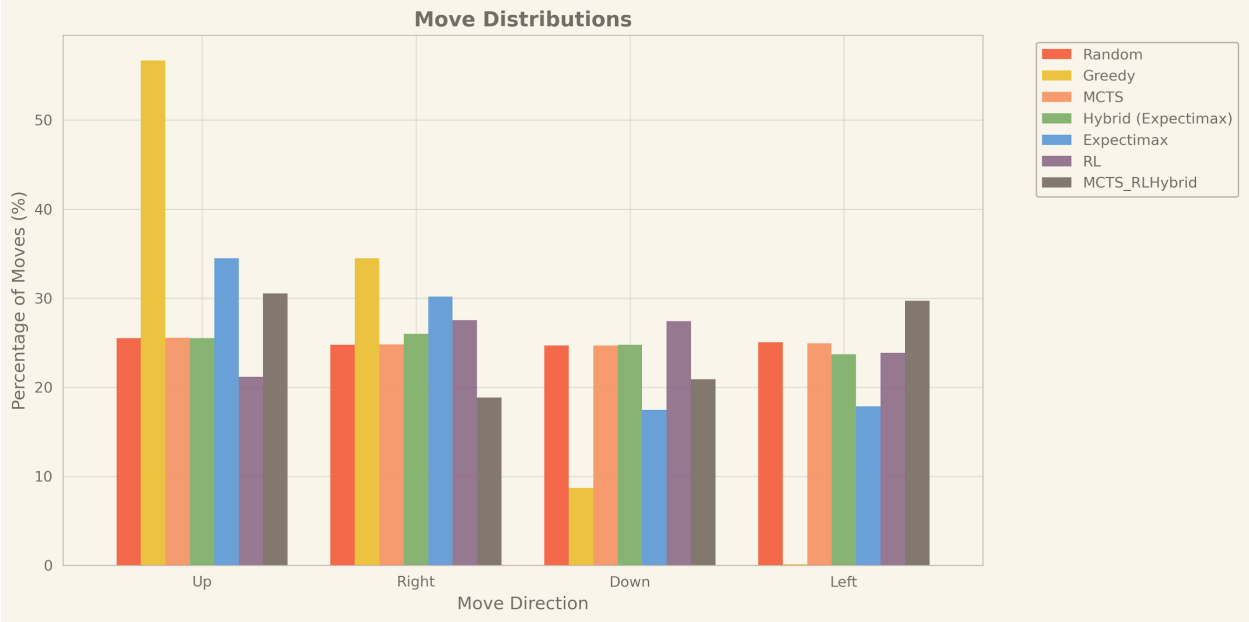
exploration and exploitation in MCTS– learned policies mirror basic strategies but lacks advanced planning and state evaluation. At last, are our **Random and MCTS agents, scoring 9.05 and 8.03**, respectively, per move. This ranking highlights the importance of structured planning strategies and informed rollout policies for better move efficiency.



The **game length** plot measures the number of moves before the agent loses and the game terminates. As captured in our efficiency plot, **Expectimax**, again, leads by a large margin, averaging 1679 moves per game. Our **MCTS with Expectimax rollouts** comes in second, averaging 269 moves per game, followed by **Greedy** with 257, **MCTS with RL rollouts** with 159, and **RL** averaging 148 moves per game. At the bottom of the ranking, we have **Random** who dies quickly after an average of 114 moves, and **MCTS** who consistently averages only 93 moves per game. This plot reinforces that intelligent planning not only leads to higher scores per move but also keeps the board alive and productive for extended periods.



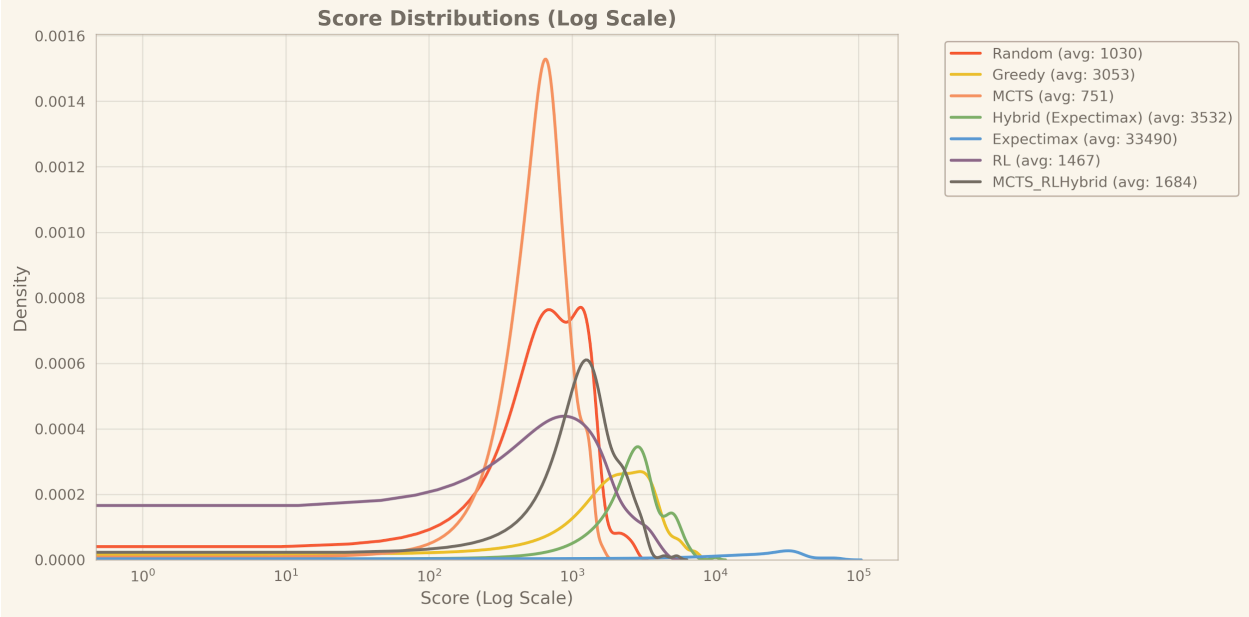
The **max tile distribution** distinguishes the agent based on the highest tile each was able to achieve. As reflected in previous plots, **Expectimax** ranks highest in this metric by far, with most of its distribution in the 2048-4096 region of the plot. The rest of the plot is collected on the left with our **MCTS with Expectimax rollouts** and **Greedy** agents peaking at the 256 and 512 tiles, our **RL** and **MCTS with RL rollouts** only reaching the 128 and 256 tiles, and our **MCTS** and **Random** agents heavily clustered around the 64 and 128 tiles. This emphasizes that Expectimax progresses the board further than any other agent.



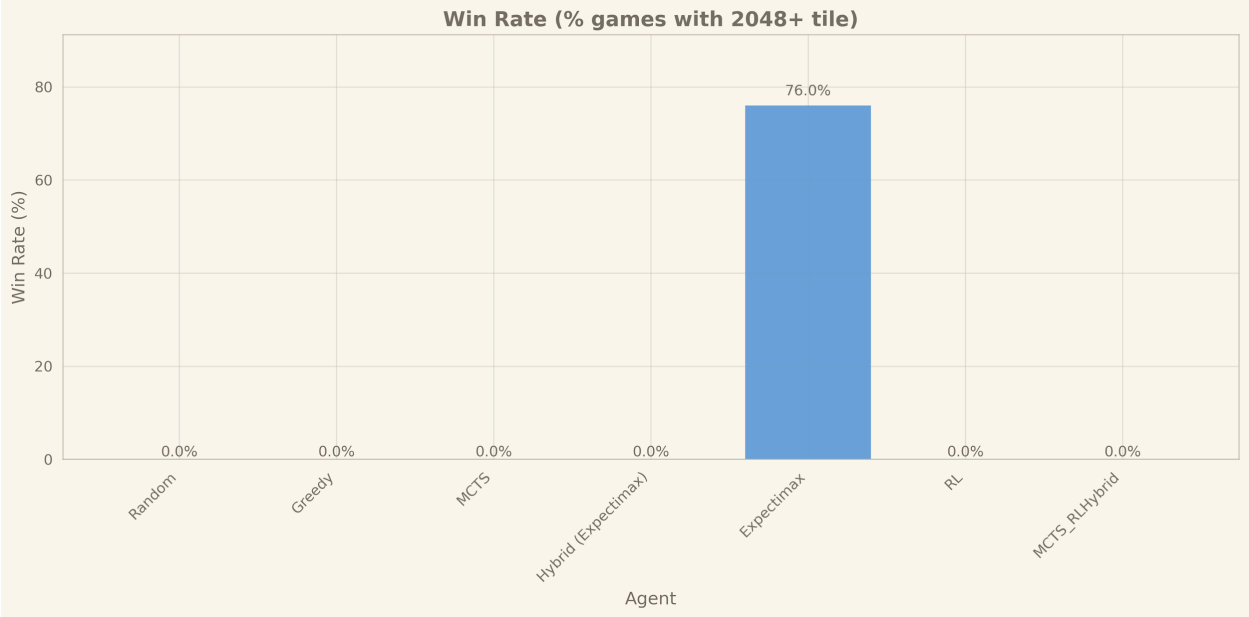
The **move distribution** plot captures the frequency at which each of the four actions: up, right, down, and left, are selected by each agent. Using this distribution, we can analyze the approaches each agent takes to manage board space and strategize for future game board states.

- As illustrated, **Expectimax** has a relatively even distribution between going up and right, while going down and left are not as frequently executed. This reflects a consistent directional bias to organize high value tiles in the top right corner.
- The **Greedy** agent's distribution of move direction is extremely skewed, as it chooses to go up almost 60% of the time, go right about 35% of the time, go down less than 10% of the time, and almost never choose to go left. This suggests that the Greedy agent prioritizes immediate tile merges and maximizing empty space, explaining why it performs well in early game states and does not adapt well to more mature game states.
- **MCTS with Expectimax rollouts and MCTS** agents both have nearly uniform distributions, with all actions having an equal likelihood of being chosen. This highlights a lack of directional strategy and control over the board space, explaining the agent's mid-tier performance.
- The **MCTS with RL rollouts** agent slightly favors going up and left more than going down and right, while the **RL** agent favors going right, down, and left. These directions potentially suggest a learned max tile corner based strategy, but not with high precision.
- The **Random** agent is a perfectly uniform move distribution, indicating no strategic preference and an unstructured approach.

All in all, this plot exhibits that strong directional strategies are essential for effective board management in stochastic game environments like 2048. Agents that lack directional bias perform poorly, while structured and purposed move selection yields high performance in the 2048 environment.



The **score distribution**, shown on a log scale, plots instances of peak performance and consistency over final scores for each agent. Our **Expectimax** agent has a right skewed tail, consistently achieving high scores, averaging 33490 per game and occasionally reaching the 40k+ range. Our **MCTS with Expectimax rollouts** and **Greedy** agents, average significantly lower, averaging 3500 and 3000, respectively, per game. Our **MCTS with RL rollouts** and **standalone RL** agents average 1684 and 1467, respectively, with wider distributions– highlighting more variability and less reliability in achieving high scores. Scoring 1030 and 751, on average per game, we have our **Random** and **MCTS** agents, showing the narrowest and lowest distribution. These results continue to align with the insights collected in earlier metrics, confirming Expectimax's dominance in strong performance and the limited effectiveness of MCTS and Random.



The **win rate plot** measures the percentage of games in which the agent was able to achieve at least the 2048 tile, presenting distinction in capability between agents. As displayed, **Expectimax** is the only agent that consistently achieves getting at least the 2048 tile, with a win rate of 76% across games. **Every other agent fails to reach the 2048 tile**, validating Expectimax's superiority in both strategic depth and consistent execution.

Collectively, these results illustrate a consistent **hierarchy of agent performance**:

1. **Expectimax**, ranking first across every metric. Its deep, deterministic search allows it to plan several moves ahead with high accuracy, making it effective in navigating the stochastic environment of 2048.
2. **Expectimax + MCTS Hybrid**, ranking second overall, showing that integrating Expectimax rollouts into MCTS significantly improves performance.
3. **Greedy**, offering a lightweight and fast alternative that performs ok short term but lacks long term strategic depth.
4. **RL and MCTS + RL**, although performing better than our random agent, struggles with consistency and often fails to reach high value tiles.
5. **Random,** performing about as well as one would expect with completely random moves.
6. **MCTS w/ random rollouts,** which performs similarly to—if not worse than—our random agent. With random rollouts, it's evident that MCTS is no better than random because of the incredibly noisy and variant environment.

9. Discussion

Our final analyses show that our Expectimax agent was the most effective agent, but we learned that creating AI agents for something as seemingly simple as a game of 2048 is far more complex than we expected. Training the reinforcement learning agent was the first major obstacle that we faced. The many thousands of episodes needed for effective training of just one model could take days, especially because we did not have access to any NVIDIA GPUs to use CUDA. We also had to explore different methods of shaping the reward with our different heuristic terms, which meant restarting training multiple times.

Expectimax was another one of the biggest issues in our project. Expectimax takes an incredibly long time to run, and so running a large number of games to evaluate different versions of it took a long time, especially when we played around with different values for thinking time. As was the case with training RL, every tweak to heuristic weights or the structure of our evaluation function required us to do reruns of thousands of games.

Because of our limited computation power and time constraints, we were forced to be very selective in our testing and what we wanted to experiment with. The very large variance in game outcomes for the game of 2048 (as it is both stochastic and scores are heavily left skewed) made large sample sizes important; however, since there is a very large difference in computational costs between the different agents, it was nearly impossible to run as many games for our Expectimax agent, for example, compared to our Greedy or Random agents.

Our Greedy agent was actually an unexpectedly effective agent, which was surprising to us because of its simplicity. When analyzing its play style, it seemed like it was able to play better than our more complex agents that were perhaps overcomplicating their decision-making process in certain scenarios. Perhaps straightforward heuristics can sometimes align better when the environment is simple and be competitive with more complicated agents.

It's clear that the performance of our MCTS and RL agents were relatively weak, especially when compared to our Greedy agent. We think that this is because of the highly stochastic nature of 2048. MCTS relies heavily on the quality of rollout policies. While using Expectimax rollouts seemed to greatly increase the performance of our hybrid MCTS, RL rollouts seemed to be less effective and the stochastic nature of 2048 meant that variance in the rollouts was incredibly high. This led to

inefficiencies in the algorithm (especially in terms of exploration of different trajectories). For the RL agent, not only did computation power limit the amount of training time that we could afford to provide the models, but RL training in 2048 is uniquely challenging because of sparse and delayed rewards. As the max tile increases, merging tiles becomes increasingly difficult and takes place over many moves. This causes the RL to tend to get stuck with strategies that maximize short-term score gains but fail to plan for long-term merges that are essential to prolonging the game and preventing gridlock of tiles and a loss.

Overall, our biggest challenge was trying to fairly evaluate our different agents in an environment where luck plays a significant role in scores that are also highly variable by nature. Large sample sizes were important, but hard to achieve because of limited computational resources and time. As a result, we had to learn to accept some level of uncertainty in our results.

---

10. Conclusion

In this project, we explored multiple AI methodologies to build high performing agents for the 2048 game. Each approach, whether it's search-based, learning-based, or hybrid, offers unique strengths. Our comparative analysis provides insight into how these agents perform under uncertain environments and how effective planning, evaluation, and learning must be harmonized to reach optimal strategies in real time decision-making.

Our Expectimax agent demonstrated the highest performance amongst all the AI agents we developed and was the only one to effectively reach the 2048 tile (and the only one to reach further to the 4096 tile). Our next highest performing agent was our MCTS + Expectimax hybrid, but surprisingly, our third-highest performing agent was our Greedy agent and used a simple single-step heuristic that outperformed our other MCTS and RL agents. Our results reveal that elaborate algorithms and complex models do not necessarily guarantee high performance in heavily stochastic environments like 2048.

Our analysis also highlights the importance of domain-specific heuristics (like empty-tile count, monotonicity, corner-max, smoothness, etc.) in learning agents that act in delayed reward environments. Using pure raw score as a heuristic would have led to far worse agents. We also discovered that computational budgets not only heavily impact the performance, but also the process of iterative improvement and evaluation of AI agents.

In its entirety, our project contributes a unified evaluation platform, a suite of heuristic functions that can be generalized across different 2048 AI agents, and a quantitative comparison that explains the strengths and weaknesses of each AI technique in solving 2048. While none of the AI agents we explored are perfect or perform as well as we had hoped, we are confident that our project works to shine a light on heuristic learning in a heavily stochastic environment.

---

11. Future Work

Given additional time and computational resources, we would:

- Train deeper and more complex reinforcement learning models (e.g., using Double DQN or Dueling Networks).
- Explore more complex rollout strategies for MCTS.
- Apply caching or more pruning methods to allow search-based algorithms to reach larger depths in a reasonable time
- Integrate AlphaZero-style self-play reinforcement learning, combining MCTS with learned policies and value networks.
- Expand the analysis to larger board sizes (e.g., 5x5 or 6x6) and compare scalability.
- Evaluate computational efficiency more rigorously by measuring time per move under different hardware configurations.
- Test adding, removing, and modifying existing heuristics and implement a grid search to attain the best hyperparameters for each individual AI algorithm.

Instead of hand-designing heuristics for Expectimax and MCTS, we would also investigate learning heuristic functions directly through supervised learning, possibly using imitation learning from expert plays. We would also benchmark agent performance with varying computational time budgets (e.g., 0.1s, 1s per move) to measure practical efficiency and responsiveness. Lastly, we would add more visibility and interpretability in our algorithms. In search-based methods like Expectimax and MCTS, adding explanations and clarity to understand why an agent favors certain moves could drastically increase the transparency of the model. We would experiment with the efficacy of visual heatmaps of tile importance or action values that could potentially add insight to the decision process of the agent.

---

12. Bibliography

1. **Matt Overlan, "Building a 2048 AI with Expectimax,"** GitHub, 2014. https://github.com/nneonneo/2048-ai

2. **Guei, Hung. On Reinforcement Learning for the Game of 2048. Diss. National Yang Ming Chiao Tung University, 2023.** https://arxiv.org/abs/2212.11087

3. **P. Rodgers and J. Levine, "An investigation into 2048 AI strategies," 2014 IEEE Conference on Computational Intelligence and Games, Dortmund, Germany, 2014, pp. 1-2, doi: 10.1109/CIG.2014.6932920.** https://ieeexplore.ieee.org/document/6932920

4. **Watanabe, Shota and Kiminori Matsuzaki. "Enhancement of CNN-based 2048 Player with Monte-Carlo Tree Search." 2022 International Conference on Technologies and Applications of Artificial Intelligence (TAAI) (2022): 48-53.** https://www.semanticscholar.org/paper/Enhancement-of-CNN-based-2048-Player-with-Tree-Watanabe-Matsuzaki/7ea90a71c41240386da55bed2a1530e13b330de4

5. **Zhou, Yulin. "From AlphaGo Zero to 2048." (2019).** https://www.semanticscholar.org/paper/From-AlphaGo-Zero-to-2048-Zhou/b9cc9a861531ede494a365e42b7ba788d638eaec

13. Appendix

13.1 Expectimax Fixed-depth

```
    def _expectimax(self, game, depth, is_max_player, start_time):
        """Standard expectimax with pruning optimizations."""
        if time.time() - start_time >= self.thinking_time:
            return self.heuristic.evaluate(game.board.grid)

        if depth == 0 or game.is_game_over():
            return self.heuristic.evaluate(game.board.grid)

        self.stats["max_depth_reached"] =
max(self.stats["max_depth_reached"], depth)

        if is_max_player:
            best_value = float('-inf')
            for move in game.get_available_moves():
                next_state = game.copy()
                next_state.step(move)
                value = self._expectimax(next_state, depth-1, False,
start_time)
                best_value = max(best_value, value)
            return best_value
        else:
            # Use strategic empty cells instead of all empty cells
            empty_cells = self._get_strategic_empty_cells(game.board.grid)
            if not empty_cells:
                return self.heuristic.evaluate(game.board.grid)

            expected_value = 0
            for (i, j) in empty_cells:
                for tile_value, probability in [(2, 0.9), (4, 0.1)]:
                    next_game = game.copy()
                    next_game.board.grid[i,j] = tile_value
                    value = self._expectimax(next_game, depth-1, True,
start_time)
                    expected_value += value * probability

            return expected_value / len(empty_cells)
```

13.2 MCTS Main Loop

```
    def get_move(self, game):
        """Returns the best move using MCTS."""

        # Reset stats each move
        self.stats = {
            "search_iterations": 0,
            "max_depth_reached": 0,
            "avg_reward": 0.0
        }

        available_moves = game.get_available_moves()
        if not available_moves:
            return -1
        if len(available_moves) == 1:
            return available_moves[0]

        root = MCTSNode(game.copy())

        start_time = time.time()
        iterations = 0
        total_reward = 0
        while time.time() - start_time < self.thinking_time:
            node = self._select(root)

            if not node.is_terminal and node.visits > 0:
                node = self._expand(node)

            reward = self._simulate(node)

            self._backpropagate(node, reward)

            iterations += 1
            total_reward += reward

        self.stats["search_iterations"] = iterations
        if iterations > 0:
            self.stats["avg_reward"] = total_reward / iterations

        return self._best_move(root)
```

13.3 DQN Core Training Loop

```
   def _learn(self):
       if len(self.replay_buffer) < self.batch_size:
           return None
       self._steps_done += 1

       batch =
Transition(*zip(*self.replay_buffer.sample(self.batch_size)))

       state_batch = torch.as_tensor(np.stack(batch.state),
device=self.device)
       action_batch = torch.as_tensor(batch.action, device=self.device,
dtype=torch.int64).unsqueeze(1)
       reward_batch = torch.as_tensor(batch.reward, device=self.device,
dtype=torch.float32)

       non_final_mask = torch.tensor([s is not None for s in
batch.next_state], device=self.device, dtype=torch.bool)
       non_final_next_states = torch.as_tensor(
           np.stack([s for s in batch.next_state if s is not None]),
device=self.device
       ) if non_final_mask.any() else torch.empty((0, 16),
device=self.device)

       # Q(s,a)
       state_action_values = self.policy_net(state_batch).gather(1,
action_batch).squeeze(1)

       # V(s')
       next_state_values = torch.zeros(self.batch_size,
device=self.device)
       if non_final_mask.any():
           with torch.no_grad():
               next_state_values[non_final_mask] =
self.target_net(non_final_next_states).max(1)[0]

       expected = reward_batch + self.gamma * next_state_values
```

```
        loss = self.criterion(state_action_values, expected)
        self.optimizer.zero_grad()
        loss.backward()
        nn.utils.clip_grad_norm_(self.policy_net.parameters(), 1.0)
        self.optimizer.step()
        return loss.item()
```

13.4 Agent Performance Results Example


```
{
 "Random": {
   "agent_name": "Random",
   "agent_config": {
     "seed": null
   },
   "scores": [...],
   "max_tiles": [...],
   "moves_per_game": [...],
   "game_durations": [...],
   "move_distribution": {
     "0": 2930,
     "1": 2957,
     "2": 2903,
     "3": 2922
   },
   "win_rate": 0.0,
   "avg_score": 1087.76,
   "avg_moves": 117.12,
   "avg_game_duration": 0.050838973522186276,
   "median_max_tile": 64.0,
   "absolute_max_tile": 256,
   "efficiency": 9.287568306010929
 },
 "MCTS_Random": {
   "agent_name": "MCTS_Random",
   "agent_config": {
     "thinking_time": 4,
     "exploration_weight": 1.414,
     "rollout_type": "random",
     "name": "MCTS_Random"
```

```
  },
  "scores": [...],
  "max_tiles": [...],
  "moves_per_game": [...],
  "game_durations": [...],
  "move_distribution": {
    "0": 2169,
    "1": 2221,
    "2": 2252,
    "3": 2177
  },
  "win_rate": 0.0,
  "avg_score": 699.16,
  "avg_moves": 88.19,
  "avg_game_duration": 367.957152338028,
  "median_max_tile": 64.0,
  "absolute_max_tile": 128,
  "efficiency": 7.927882979929697
},
```


13.5 Expectimax Best Runs– (70 games, 2 sec. thinking time)


```
{
 "Expectimax": {
   "agent_name": "Expectimax",
   "agent_config": {
     "thinking_time": 2.0,
     "name": "Expectimax"
   },
"max_tiles": [
 2048,
 1024,
 2048,
 2048,
 2048,
```

```
 2048,
 4096,
 2048,
 4096,
 2048,
 1024,
 512,
 512,
 2048,
 4096,
 4096,
 4096,
 4096,
...
"max_tiles": [
 2048,
 1024,
 2048,
 2048,
 2048,
 2048,
 4096,
 2048,
 4096,
 2048,
 1024,
 512,
 512,
```

```
 2048,
 4096,
 4096,
 4096,
 4096,
...
"game_durations": [
    ...
    5475.145347118378,
    6254.584916353226,
    3557.7819056510925,
    5535.340355396271,
    1933.1485092639923,
    2133.667058467865,
    1306.1288812160492,
    3654.74183177948,
    1976.7009329795837,
    562.4114689826965,
    923.4919168949127,
    3241.4202609062195,
    4905.514191150665,
    1657.98992395401,
    929.5799765586853,
    6654.788226366043,
    4113.299973726273,
    6595.922682285309
  ],
  "move_distribution": {
```

```
    "0": 40853,
    "1": 38260,
    "2": 18637,
    "3": 16886
  },
  "win_rate": 67.14285714285714,
  "absolute_max_tile": 4096,
  "min_score": 3484,
  "max_score": 76116,
  "avg_score": 33003.08571428571,
  "avg_moves": 1637.6571428571428,
  "avg_game_duration": 3279.944664185388,
  "median_max_tile": 2048.0,
  "efficiency": 20.152622212917407
 }
}
```