STAT1840 Project Milestone
Winning at 2048 Using Reinforcement Learning

Eliot Atlani
Master Student
Harvard University
Cambridge, MA 02138
`eliotatlani@g.harvard.edu`

Mathilde Cros
Master Student
Harvard University
Cambridge, MA 02138
`mathildecros@g.harvard.edu`

Abstract

This project investigates the application of reinforcement learning techniques to the game of 2048. Implemented in Python using object-oriented programming principles, the study explores a progression of baseline models, ranging from simple heuristics to more advanced methods such as Monte Carlo Tree Search (MCTS) and policy gradients, gradually increasing model complexity. Additionally, we collected gameplay data to develop an expert agent, which is utilized to enhance both the MCTS and policy gradient approaches. This milestone focuses on evaluating and refining these models to establish a strong foundation for achieving high performance in 2048.

1 Problem statement

1.1 Context

Progressing in the project, we have now finished implementing the random action policy and heuristic-based models to construct our baseline. We have also implemented a first working version of expectimax optimization [1], the policy gradient and the monte carlo tree search methods as described in the project proposal. However, there remains space for improvement and further interpretability of our results that we will complete for the final report. We have implemented all the algorithms fully from scratch, with the only packages being used are itertools, random, numpy, torch and tkinter (for the game play).

We also have now decided to use **bit manipulation** in our state space to optimize computations, reducing cost and enhancing speed.

The game presents an interesting challenge for reinforcement learning due to its stochastic nature, high-dimensional state space, and dynamic board configuration. By applying RL techniques, we aim to discover effective strategies for achieving high tiles consistently and understanding the mechanics of decision-making in uncertain environments. We furthermore wish to explore for our implemented algorithms how model capacity affects performance and get different technical interpretation insights from our different algorithms' results.

Our implementation of the project can be found in the GitHub repository RL-for-2048. We based our work on that of Anders Qiu's GitHub Repository [2] on top of which we implemented our own approach. We decided to proceed in the following way instead of a fully built in version / library in order to have our own version implemented of the game and thus avoid any constraints. In such a way, our implemented version is also less computationally heavy due to better optimisation of the implementation logic of the game.

2 Simple baseline models

2.1 Random Action Policy

As a baseline for comparison, we implement a policy that selects actions randomly at each time step. This approach provides a straightforward reference point for evaluating the effectiveness of more sophisticated strategies. To ensure a robust estimation of the average performance under this policy, we conduct 100 independent simulations, calculating the score across these runs.

Table 1: Random action policy simulation result

Mean score	Best score reached
1194.28	3136

As shown in Table 1, the random action policy achieves an average score of 1194.28 across 100 simulations.

2.2 Heuristic-Based Model

A heuristic-based policy provides a simple yet effective approach to decision-making. In this method, the evaluation of the grid is guided by predefined criteria designed to encourage favorable outcomes and discourage unfavorable ones. At each state, the model evaluates all possible actions by simulating their effects and scoring the resulting grid based on the heuristic. The action yielding the highest heuristic score is then selected.

As before, we ran 100 simulations of each heuristic.

Table 2: Heuristic policy simulation results

Heuristic Name	Description	Mean Score	Best Score
Snake Chain	Encourages snake-like patterns.	2217	6968
Max Tile Corner	Positions the highest tile in a corner.	1104	3484
Monotonicity	Rewards consistent row/column values.	3078	6636
Merge Tile	Prioritizes merging tiles.	2848	6436
Empty Tiles	Favors more empty tiles.	2736	6688
Weighted Combination	Balances multiple heuristics.	3695	7328

From the results in Table 2, it is evident that heuristics contribute to improving scores compared to random action policies. However, they are not sufficient to consistently achieve optimal results.

These heuristics provide valuable insights into which strategies yield better outcomes and how to prioritize certain actions over others.

Additionally, performance can be further improved by increasing the depth of simulation. Instead of evaluating only the immediate effect of an action, the policy can simulate 3 to 6 future steps for each action, significantly enhancing decision quality and final scores, which leads us to the next part.

3 Expert Agent

To develop our Policy Gradient or MCTS model, an expert agent will be used. The role of the expert is to provide confident action and help our model to "learn" properly. To create such an agent, we'll use the work of *Robert Xiao* [1]. In his work, Robert achievied to create a model capable to reach 16384 at almost every game. He uses *expectimax optimization* where the AI simply performs maximization over all possible moves, followed by expectation over all possible tile spawns.

The powerful model will be used to collect data to train our Expert AgenTo train our expert agent, we developed a CNN-based policy network using a dataset generated by Robert Xiao's expectimax model (with some data augmentation), which maps board states to expert actions. The network processes 4x4 grid inputs and predicts the best action (up, down, left, or right). To handle class imbalance, we applied weighted cross-entropy loss during training.

The dataset was split into training and validation sets, and the model was trained over 100 epochs, with performance monitored through accuracy and loss metrics. Once trained, the policy network was saved and will be used to enhance our MCTS and Policy Gradient models by providing informed actions based on expert data.

4 Policy gradient

We then implemented the Policy Gradient algorithm to directly learn the policy without explicitly learning a value function, which is convenient for high-dimensional and stochastic environments like 2048, as it allows for optimizing non-deterministic policies.

We started by implementing a neural network architecture of fully connected layers with ReLU activation functions to learn complex representations of the game states and with a final softmax layer to ensure that the outputs represent valid probabilities. The flexibility of the network's design is achieved through configurable parameters, such as the number of layers and hidden units, which can be adjusted for experimentation. We then preprocessed the grid state before being passed to the neural network by flattening it into a 1D array and normalizing it by taking the base-2 logarithm of each tile value (to handle the exponential growth of tile values) and scaling it between 0 and 1. Empty tiles are represented as zero.

As before, the rewards are defined as the sum of the tile values merged during each action and we discount these rewards to prioritize immediate rewards while still considering future gains. Using the Policy Gradient theorem, we compute the gradient of the objective function (negative log probability of the chosen action scaled by the discounted reward) and optimize the network using the Adam optimizer.

Finally, we implemented a grid search over the network's hidden layer configurations and learning rates. Hidden sizes ranged from simple architectures (e.g., [128]) to complex, deeper networks (e.g., [1024, 512, 256, 128]). Learning rates varied between 0.01, 0.001, and 0.0001. Each configuration was tested over multiple runs, and the average score was recorded to identify the best-performing combination, over 100 simulations again, that we reported in the following table.

Table 3: Policy gradient simulation result

Mean score	Best score reached
1262	4634

From Table 3, this algorithm barely outperforms our Random Action baseline model but not the Heuristic-Based ones and never reaches 2048, even after fine tuning all hyper parameters mentioned above.

5 Monte Carlo Tree Search Model

For this milestone, a basic Monte Carlo Tree Search (MCTS) algorithm was implemented. The algorithm explores possible game states by simulating actions through four steps: selection, expansion, simulation, and backpropagation. At each step, it balances exploration and exploitation using the Upper Confidence Bound for Trees (UCT). Simulations use random moves to evaluate potential outcomes, with either the grid score or a combined heuristic to evaluate the game states. Finally, the action with the highest average score is selected, providing a simple yet effective strategy for decision-making.

MCTS is well-suited for 2048 because it handles the game's stochastic nature, with new tiles appearing randomly, by exploring multiple action sequences. It balances exploring new possibilities and exploiting promising moves, making it effective for complex decision-making.

To compare our fist model to our baseline, we'll run as well 100 simulations with the following hyperparameters: $iterations = 50$, $simulation\_depth = 50$ and $exploration\_weight = 1$.

Table 4: Monte carlo tree search simulation result

Mean score	Best score reached
8141	16096

From Table 4, our basic model outperform our baseline, but it's still far to obtain a 2048 ($\sim$ 22000 in total score). It's good to note that the time complexity of this algorithm is much higher than the previous exposed one.

6 Work Left / Next Steps

We still have a few steps to complete this project.

We will try to experiment further with nuanced experiments for Policy Gradient. For example, we plan to experiment with the size of the actor and critic networks in our models. These networks are a key part of how the agent learns. By increasing the number of layers or nodes in these networks, we can test how much improvement we see in performance. At some point, the performance will stop improving as the networks get larger, and we want to figure out where that point is. This will help us understand the balance between making the model more powerful and keeping it efficient.

For the Monte Carlo Tree Search, we will finetune the parameters and limit us to a minimum of 3 moves per second for the finetuning. We will also integrate the expert agent into our MCTS model to improve its performance. By guiding simulations with informed actions instead of random moves, the expert agent will provide better state evaluations, leading to more accurate backpropagation and action selection.

Lastly, we want to dig deeper into understanding why some algorithms work better than others in certain situations. For example, we'll look at how the randomness in the game, like where new tiles appear, affects the agent's decisions and overall performance. We also plan to analyze how quickly each algorithm learns and whether they remain consistent across multiple games. Exploring things like how reward structures impact learning will help us gain more insight into the agent's behavior and strategies. We'll also look at how reliable each model is over multiple games.

References

[1] Robert Xiao 2048-ai github repository `https://github.com/nneonneo/2048-ai`.

[2] Anders Qiu's 2048 Code Github Repository `https://github.com/andersqiu/python-tk-2048`.