Different Reinforcement Learning Approaches to the Game 2048

Ajinkya Bhosale, Harshkumar Patel, Rohit Shetty
Department of Computer Science
University of Toronto

Abstract— **This research paper aims to compare different Reinforcement Learning Algorithms in their performance learning and playing the game 2048. We have implemented the algorithms state-action-reward-state-action (SARSA), Deep Q-Learning (DQN) and Monte Carlo Tree Search (MCTS). We compare the results of these algorithms, provide potential reasoning why one algorithm is better compared to another and also talking about situation wheres these algorithms may perform better. Moreover, we try to extend MCTS to weighted MTCS to potentially factor in human intuition to get better gameplay performance.**

I. Introduction

The mobile game 2048 is a non-determinsitic game where on a 4x4 grid, there are tiles which spawn. The player has the option of moving up, down, left or right. All the tiles in the game are powers of 2, (2, 4, 8, 16, etc.). When you make a move and two of the same tiles collide into eachother, the combine to create the next power of two. The non-determinisitc aspect of this game is that each time a player makes a move, a new tile is added to the grid with a probability of being 2 or 4.

While many individuals have created models to maximize the score in 2048, to our knowledge, we have not found any research done that really compares the different models and gives some reasoning about the strengths and weaknesses of models in the contexts of games like 2048.

This paper looks specifically at three reinforcement learning algorithms: State-Action-Reward-State-Action (SARSA), Deep Q-Learning (DQN) and Monte Carlo Tree Search (MCTS) and moreover tries to extend MCTS to incorporate a strategy humans use when playing 2048.

II. Literature Survey

A. Background

State-Action-Reward-State-Action (SARSA) is an on-policy reinforcement learning algorithm used for solving Markov decision processes. While SARSA may seem similar to DQN, the difference is SARSA learns a policy and updates its value estimation based on the actions actually taken rather than the greedy approach, which always looks for the maximum. Here is how SARSA is implemented in the context of 2048:

- State Representation and Action Space: Each state of the game is represented by the current tiles and their layout and the grid and the action space is the four possible moves the player can make.
- Q-Table Initialization: SARSA uses a Q-Table to store and update the valye of state-action pairs with the table being initially filled with arbitrary values representing the expected reward for taking a specific action on a given state.
- Epsilon-Greedy Policy: SARSA utilizes an epsilon-greedy policy for action selection. This balanced exploration (trying new things) and exploitation (picking actions based on what was learned). There is a notion of epsilon decay implemented so the algorithm focuses more on exploration at the beginning and exploitation at the end.
- Learning from Sequences: The core of SARSA is learning from sequences of state-action pairs. After performing an action, the agent observes the reward and the next state. It then selects the next action (again, following the epsilon-greedy policy) and updates the Q-table based on these observations. The update formula is:
  $Q(s,a) \leftarrow Q(s,a) + \alpha \times (r + \gamma \times Q(s',a') - Q(s,a))$
  where s and s' are current and next states, a and a' are current and next actions, r is the reward, α is the learning rate and γ is the discount factor.
- Goal and Evaluation: The goal of the SARSA agent in 2048 is to maximize the score, ideally reaching the 2048 tile. The agent's performance is evaluated based on the highest tile achieved and the total score over multiple episodes.

Deep Q-Learning (DQN) is a variant of Q-learning that combines reinforcement learning with deep neural networks. The main goal is to approximate the Q-function, which represents the expected cumulative future rewards for taking a particular action in a given state. Here's an overview of how Deep Q-Learning works in our instance of 2048:

- Neural Network Representation: A neural network is used to approximate the Q-values, where the input is a preprocessed state of the 2048 game. The neural network has layers for processing the input and estimating Q-values for each possible action.
- Epsilon-Greedy Policy: The agent follows an epsilon-greedy policy for action selection. With a high probability $(1-\varepsilon)$, it chooses the action with the highest estimated Q-value according to the current policy. With a low probability ε, it explores by choosing a random action.

- Experience Replay: The agent stores experiences (state, action, reward, next state) in a replay memory. During learning, it randomly samples batches of experiences from this memory to break correlations and improve stability.
- Q-Network Update: The agent updates its Q-network periodically using a batch of experiences. It computes the Q-values for the current state and the next state using the neural network. The loss is calculated based on the temporal difference between predicted Q-values and target Q-values. Backpropagation is used to update the neural network weights, aiming to minimize this loss.
- Training Loop: The agent plays the 2048 game for multiple epochs, interacting with the environment and updating its Q-network. The training loop continues until 300 epochs are completed.

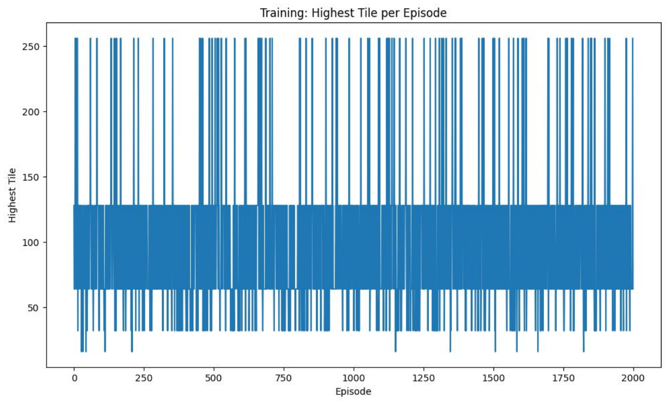
Monte-Carlo Tree Search (MCTS) is an algorithm for stochastic selective search that relies on random simulations. It's commonly employed in artificial intelligence for games like chess, Go, and various other strategic or tactical games. Starting from a game tree with just a root node representing the current game state,the process involves four key steps, progressively expanding a game tree:

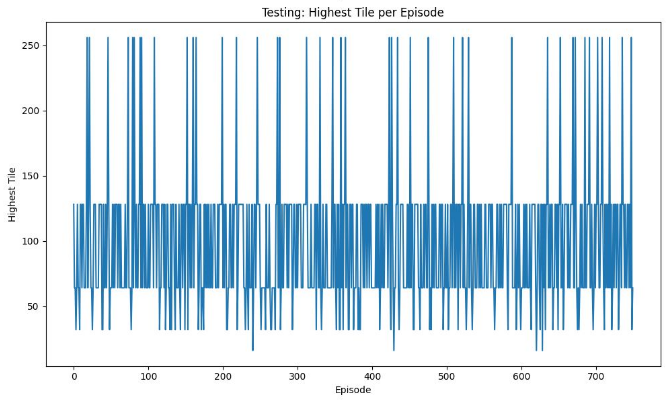
- **Selection**: The algorithm begins at the root and navigates down the tree to a leaf. The selection process involves a balancing between exploring new moves and exploiting moves that have led to favorable outcomes in the past.
- **Expansion**: Once a node is selected, the algorithm expands the tree by considering possible moves from the current state. For 2048, this means considering moves such as up, down, left, and right, and creating child nodes representing the possible outcomes of these moves.
- **Simulation**: The algorithm conducts a simulation or rollout from the selected child node by making random moves until reaching a terminal state.
- **Backpropagation**: The results of the simulation are backpropagated up the tree, updating the statistics of the nodes visited during the selection and expansion phases.

B. Related Work & Past Studies

As demonstrated earlier, there were three distinct reinforcement learning (RL) algorithms that captured our interest, each demonstrating varying degrees of success in terms of achieving a high score in 2048.

For SARSA, if we look at the old paper[1], they did not get great results. The result of their algorithm was quite inconsistent and did not show any overall growth as the algorithm tried to learn the game. They did claim however that if the algorithm ran for longer, it would do better and stabilize. Using similar hyperparameter settings as they used, we ran SARSA on our own game and these are the results.

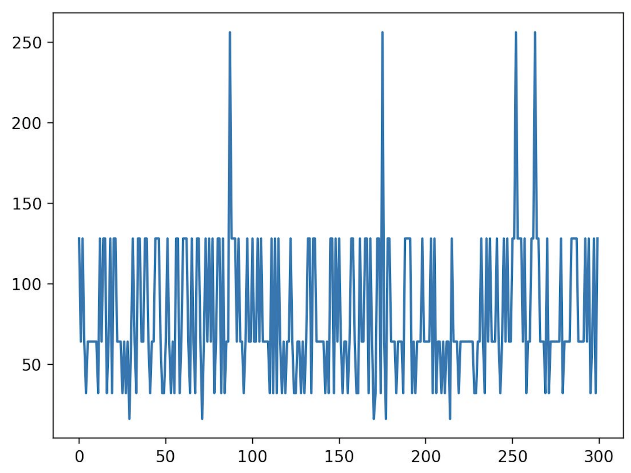




It is clear even with more iterations, the algorithm does not show much growth. There is still inconsistency in terms of the highest tile it reaches for each game it plays. A potential reason for this may be due to the randomness of tile spawning. Machine Learning algorithms generally perform better when there is not an element of randomness for each turn. Moreover because of the randomness of tile spawning, it can be conjectured that the algorithm was not able to come up with a strong strategy and each time it would explore a new strategy, it would not necessarily be applicable to another iteration of playing the game.

Limitations: Not good with a lot of randomness
Strengths: Computationally cheap

For DQN, the agent learns to play the 2048 game by interacting with the environment, storing experiences, and updating its Q-network using a combination of epsilon-greedy exploration and experience replay. The use of a target network helps stabilize the learning process. The results were fairly unsuccessfully. Have a look at the graph obtained looking at max score vs epoch.



There are just a few scores of 256 achieved and we find a general trend of scores in the 32 to 128 range with a high degree of fluctuation. Here are some reasons possible reasons:

- The rewards in 2048 are often delayed, as merging tiles and achieving higher values takes time. Q-learning may struggle to attribute rewards accurately to specific actions, leading to delayed learning.
- Q-learning might find it challenging to generalize learning from one game configuration to another, especially when the agent encounters different patterns and structures in the grid.
- Rewards could perhaps be placed on merge type scores instead of following the suggested basic rewards for increasing score.

MCTS In one of the recent studies [2], they leverage the Monte Carlo Tree Search algorithm to develop a 2048-playing agent, comparing different evaluation policies. The results emphasize the effectiveness of using the sum of tiles on the final board for achieving a higher percentage of 2048 tiles, a larger average sum, and a higher average merge score. This research contributes to the understanding of AI agents in complex games, highlighting the significance of evaluation functions in decision-making processes. Although the studies showed great results, claiming the agent consistently reaching a score of 2048, it came at a high computational expense. Their best score was achieved when the bot required around 250 rollouts per possible move to find an optimal move, given a game state, which took their experiment to run for 2 days and 8 hours.

To be able to benchmark this approach to our proposed algorithm shown later on, we implemented and tested their MCST method on our 2048 game instance to keep consistency. Overall, the MSCT algorithm performed best out of the three aforementioned RL algorithms with the following results averaged over 30 trials each:

Where it took approx. 57min to conduct 30 rollouts/simulation and approx. 1hr 52min for 60 rollouts/simulation.

Evaluation Function	No. Rollouts	Avg Sum	Avg Merge Score
Sum Tiles	30	1427.4	7308
Sum Tiles	60	1581.54	9324
Largest Tile	30	1358.44	6306
Largest Tile	60	1468.01	7968

TABLE I
RESULTS OF BASELINE MSCT ON 2048 OVER 30 TRIALS

From our experiments and analysing the results of MSCT we devised the following limitations and advantages of usign MSCT on 2048

Limitations:

- **Computationally Intensive**–since we have to do 4*(No. rollouts) random simulations of a game to choose the most optimal move given a game
- **Difficulty in Handling Long-Term Planning**–since MCTS tends to focus more on short-term gains, it struggles to effectively evaluate moves that have delayed consequences.

Advantages:

- **Effective in Large State Spaces**–since 2048 has a large state space, MCTS can handle such situations efficiently as it explores and focuses on promising branches, avoiding the need to explicitly visit every state.
- **Adaptability to Uncertainty**–MCTS in general is well-suited for games with complex and uncertain environments, making it applicable to 2048 where the randomness of tile placements introduces uncertainty.

III. ALGORITHM EXTENSION AND RESULTS

Comparing the algorithms we tested on 2048, it is clear that although MCTS was computationally expensive, it produced the best results, which is why it will be the focus as the base algorithm to be used to extension.

After actually playing 2048, it was clear that a basic general strategy to increase score was to keep some sort of organization on the grid. To put differently, cluster the values so the larger ones are towards a corner with the values getting smaller as the tiles are further away from that corner on the grind.

A. Attempt 1

Our naive first attempt at clustering involved updating our evaluation metrics to now be the sum of the top-left 3x3 grid. This attempt was motivated by the observation that, as humans, there is a tendency to strategically position the highest blocks at the corners while playing this game. Recognizing this common human behavior, we hypothesized that the sum of the top-left 3x3 grid could effectively capture the essence of cluster significance in the context of the game.

Evaluation Function	No. Rollouts	Avg Sum	Avg Merge Score
Sum Tiles	30	328.4	2282.4
Sum Tiles	60	528.4	3977.28

TABLE II
RESULTS OF L-CLUSTER MSCT ON 2048 OVER 30 TRIALS

The results were worse than the baseline solutions, indicating that relying solely on the sum of the top-left 3x3 grid as an evaluation metric may have limitations compared to considering the sum of the entire board in the context of the game as:

1. **Limited View of the Board:**
   Restricting the evaluation to the top-left 3x3 grid provides only a partial view of the game board. It may not capture the full complexity and interactions happening across the entire board.
2. **Neglecting Key Areas:**
   Important developments and high-value tiles may occur outside the top-left 3x3 grid. Ignoring the rest of the board might lead to overlooking critical patterns or clusters that contribute significantly to the overall game strategy.

B. Attempt 2

Our second attempt involved layering the weights and not focusing on a specific corner. The main flaw with the original algorithm is that while we focused on clustering in 1 corner, that did not necessarily mean that there was organization within the cluster. This second attempt focuses to layer the clustering to focus the largest tile towards one corner while smaller tiles will be adjacent to that corner.

This was implemented by adding a board heuristic where we would go through the board and add a weight (multiply by a factor) to encourage the algorithm to push towards keeping the largest tile in the corner and then keeping the next largest adjacent with a smaller weight.

Evaluation Function	No. Rollouts	Avg Sum	Avg Merge Score
Sum Tiles	30	479.6	2392.7
Sum Tiles	60	593.2	4162.4

TABLE III
RESULTS OF L-CLUSTER MSCT ON 2048 OVER 30 TRIALS

While this implementation of layering improved the performance compared to one big cluster, it still has worse results compared to the base algorithm.

IV. CONCLUSIONS

In this paper, we implemented and analyzed three RL algorithms used previously on 2048 and devices a augmentation of MCST that addressed the limitations and challenges we notices for the algorithms.

Some key factors to point out is that basic MCTS had the best performance, which is likely due to the fact that it considers a lot more different possibilities and just the breath of possibilities alone gives it more options to choose from. Now the other two algorithms had their limitations and in general did not perform too well. This is likely due to the random nature of the game and due to the Q-table not being able to recognize and pick up patterns.

While weighted MTCS was not necessarily an improvement, we think that there could be a smarter way of balancing the weights or reducing cluster, but do to time constraints we were not able to consider many other options.

REFERENCES

[1] Baluja, M. (2021). Reinforcement Learning for 2048. https://www.michaelbaluja.com/assets/files/drl-2048.pdf[Accessed: 29 - Nov - 2023]
[2] Goenawan, N., Tao, S., amp; Wu, K. (2021). What's in a game: Solving 2048 with reinforcement learning. https://web.stanford.edu/class/aa228/reports/2020/final41.pdf. [Accessed: 01 - Dec - 2023]
[3] 15418.courses.cs.cmu.edu. (2018). The Barnes-Hut Algorithm : 15-418 Spring 2013. [Online]. Available: http://15418.courses.cs.cmu.edu/spring2013/article/18. [Accessed: 07- Jan- 2018].