CS 461 – Artificial Intelligence – Intermediate Report – Group 12
2048 with Min-Max Tree Search, MCTS, and DDQN

Berk Çakar - 22003021, Ceren Akyar - 22003158, Deniz Mert Dilaverler - 22003530,
Elifsena Öz - 22002245, İpek Öztaş - 22003250

***Abstract*— For this intermediate report of the term project, we implemented, tested and analyzed three algorithms Min-Max Tree Search, Monte Carlo Tree Search (MCTS), and Double Deep Q-Network (DDQN) on the 2048 game. Metrics of average score achieved, highest score achieved, and highest tile reached were collected on our tests. It was found that the MCTS algorithm performed the best consistently reaching 1024 and reaching 2048 easily. Min-Max tree search also performed decently, reaching 1024 in 70% of the episodes when run with a depth of 5. However, the DDQN algorithm performed poorly, barely reaching higher than the tile 256.**

I. INTRODUCTION

The game 2048 was introduced by Gabriele Cirulli in 2014 [1]. The game is played on a 4x4 grid, with numbered tiles that slide when a player moves them using the four arrow keys. The game's objective is to slide numbered tiles on a grid to combine them to create a tile with the number 2048. The game is won when a tile with a value of 2048 appears on the board. After reaching the 2048 tile, players can continue to play (beyond the 2048 tile) to reach higher scores. The game is lost when the player cannot make a move (i.e., the board is full, and no adjacent tiles have the same value).

This project implements and tests three algorithms for playing 2048: Min-Max Tree Search, Monte Carlo Tree Search [2], and Double Deep Q-Network (DDQN). Our principal goal is to analyze these three algorithms for solving the game of 2048. The algorithms are evaluated and compared according to their max and average final scores and their percentage to reach 512, 1024, and 2048 after 100 tries. Hence, our project's significance extends the literature by comparing different AI techniques and a replication package that other researchers can use. Also, the deep reinforcement algorithms only achieved the 2048 tile 7% of the time, which will be discussed in detail in the Related Work section. Hence, our further intention is to improve the 7% success rate.

II. RELATED WORK

Several studies in the literature use AI approaches to play 2048. The most prominent one comes from a doctoral thesis [3]. There, the author tries techniques such as Monte Carlo Tree Search, Min-Max Tree Search, Temporal Difference Learning, Expectimax Search, and Deep Reinforcement Learning. However, a clear comparison between these techniques is not provided. Similarly, the replication package of this study is not available. Another work related to our project is a paper that utilizes variants of Deep Q-Learning to play 2048, as well as Minesweeper and Sudoku. Similar to the previous one, no replication package is available, and the emphasis on 2048 is less than the other two games [4]. Another study also used Deep Reinforcement Learning to play 2048, but they only achieved the 2048 tile 7% of the time [5]. In conclusion, despite the previous works, the findings suggest a continued interest in leveraging AI for 2048, with room for further investigation and improvement in achieving a more consistent success rate.

III. METHODOLOGY

To realize a 2048 environment with reward, state, and action spaces, we used the OpenAI Gym library and its wrapper for 2048[1]. A screenshot of the game is shown in Fig. 1. The reward is only given when the player moves the tiles and combines two tiles with the same value. In such cases, the reward is the sum of the two tiles. For example, if the player combines two tiles with the value of 16, the reward is 32. The state space is the board itself, which is a 4x4 matrix. This 4x4 contains the values of the tiles on the board in log2 form, where 0 represents an empty tile. For example, if a tile has a value of 16, its value in the state space is 4. The action space is the four directions that the tiles can move: up, down, left, and right. Unless otherwise given as an argument to the 2048 environment, the game is won when a tile with a value of 2048 appears on the board. The game ends if there is no possible move left.

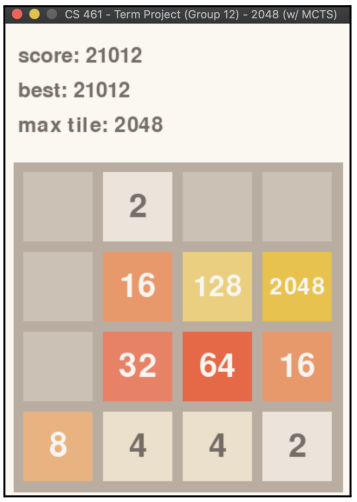

Figure 1. Screenshot of the 2048 environment

[1] https://github.com/helpingstar/gym-game2048

For the Min-Max Tree Search, we used the pseudocode provided in the lecture slides. The algorithm is as follows:

```
def value(state):
    if state is terminal:
        return utility(state)
    if state is max:
        return max-value(state)
    if state is min:
        return min-value(state)
def max-value(state):
    v = -inf
    for each successor of state:
        v = max(v, value(successor))
    return v
def min-value(state):
    v = inf
    for each successor of state:
        v = min(v, value(successor))
    return v
```

However, implementing this algorithm was not straightforward. For constructing the tree, we deep-copied the current state of the board and applied the action to the copied board. Then, we added the new board to the tree. We repeated this process until we reached the maximum depth. At the current state of the project, we support a maximum depth of 8 since deep-copying the board is a costly operation and requires optimization. Hence, the environment crashes if we provide a depth greater than 8. After constructing the tree, we applied the Min-Max algorithm, as shown in the pseudocode. The algorithm returns the best action to take, which is the action that leads to the highest score.

To implement the Monte Carlo Tree Search, we benefited from an extension of the OpenAI Gym library, which integrates the Monte Carlo Tree Search algorithm to OpenAI Gym environments[2]. The library utilizes deep-copying and simulation operations as we did in the Min-Max Tree Search. As for the depth of the tree, we set it to 100 by default. However, it can be changed as an argument to the environment.

Lastly, for the Double Deep Q-Network, we followed a similar approach to the course TA's implementation of the Deep Q-Network. The only difference is that we used a Double Deep Q-Network instead of a Deep Q-Network. The Double Deep Q-Network is a variant that uses two neural networks. The first neural network is called the online network, and the second is called the target network. The online network is updated every iteration, while the target network is updated every 1000 iterations. The neural networks consist of three fully connected layers with 128 neurons each. The input layer has 16 neurons, which is the size of the state space. The output layer has four neurons, which is the size of the action space. The activation function of the first two layers is ReLU, and the activation function of the last layer is linear. Unless otherwise stated, the default hyperparameters used for experiments are as follows: learning rate = 1E-4, discount factor (gamma) = 0.99, epsilon = 1, and epsilon decay = 0.995, epsilon min = 0.01, batch size = 128, and buffer size = 3E+5.

The hyperparameters that have been explored for each algorithm to get the highest success rates and final scores.

IV. RESULTS

In this section, the implemented algorithms' performance results are presented.

A. Min-Max Tree Search Algorithm

The Min-Max Tree Search algorithm was tested for tree depth values 4 and 5. Currently, since the run-time for this algorithm takes a long time, the results are shown over 10 episodes rather than 100.

The success rate when the algorithm is run for ten episodes with tree depth 4 is 0% for reaching the 2048 tile and 40% for the 1024 tile. The average score is 8042.0, and the maximum score is 14920. The graphs that present the results are as follows:

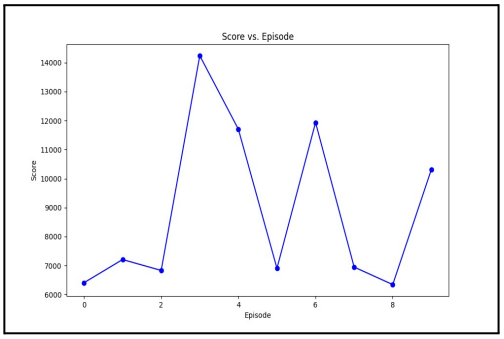

Figure 2. Score vs. Episode (depth=4)

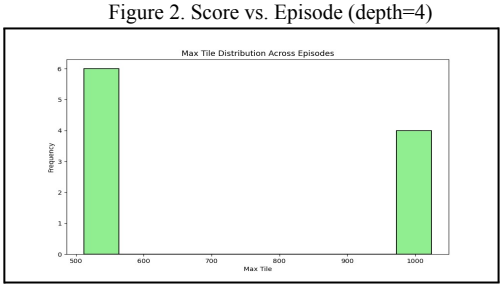

Figure 3. Max Tile Distribution (depth=4)

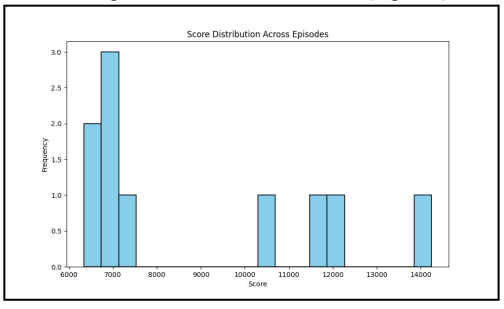

Figure 4. Score Distribution Across Episodes (depth=4)

The success rate when the algorithm is run for ten episodes with tree depth 5 is 10% for reaching the 2048 tile and 70% for the 1024 tile. The average score is 8886.8, and the maximum score is 15808. The graphs that present the results are as follows:

[2] https://github.com/PatrickKorus/mcts-general

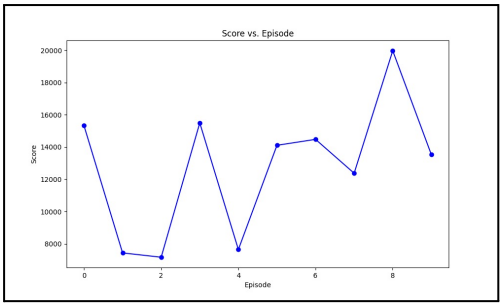

Figure 5. Score vs. Episode (depth=5)

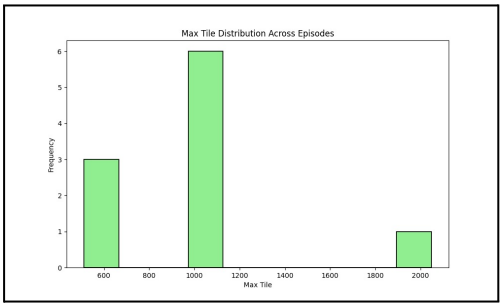

Figure 6. Max Tile Distribution (depth=5)

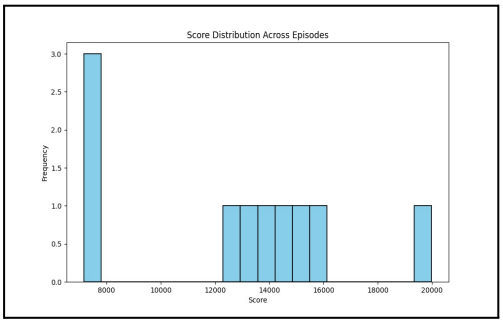

Figure 7. Score Distribution Across Episodes (depth=5)

B. Monte Carlo Tree Search Algorithm

The Monte Carlo Tree Search algorithm was tested for tree depth 100. Currently, since the run-time for this algorithm takes a long time, the results are shown over 10 episodes rather than 100.

The success rate when the algorithm is run for ten episodes with a tree depth of 100 is 10% for reaching the 2048 tile and 90% for the 1024 tile. The average score is 14390. The graphs that present the results are as follows:

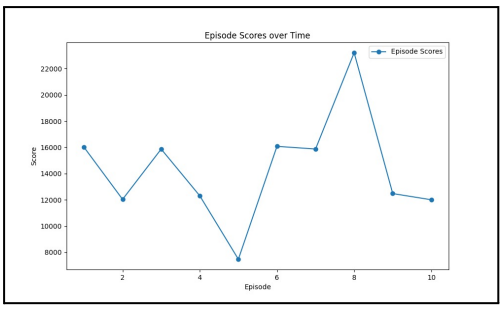

Figure 8. Score vs. Episode (depth=100)

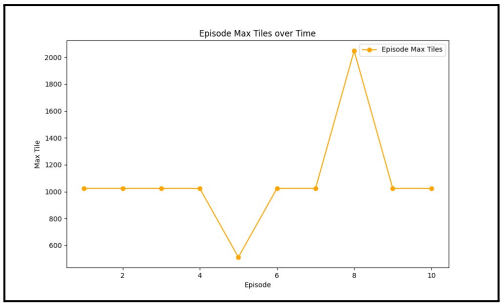

Figure 9. Max Tile vs. Episode (depth=100)

The success rate when the algorithm is run for ten episodes with a tree depth of 150 is 70% for reaching the 2048 tile and 100% for the 1024 tile. The average score is 25486. As observed, increasing the depth of the MCTS results in better performance. The graphs that present the results are as follows:

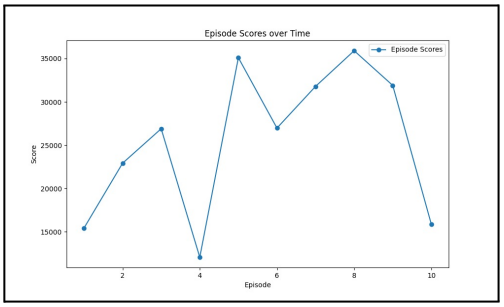

Figure 10. Score vs. Episode (depth=150)

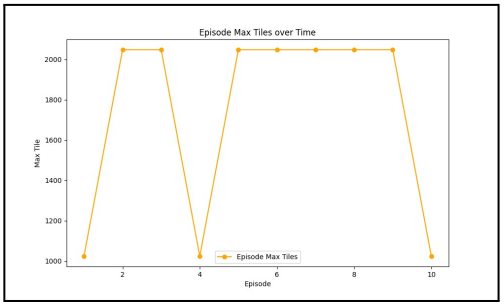

Figure 11. Max Tile vs. Episode (depth=150)

C. Double Deep Q-Network (DDQN) Algorithm

The DDQN Algorithm was tested for learning rates 1E-4, 2.5E-4, 5E-4; gamma 0.1, 0.5, 0.8; and buffer size 1.5E+5, 6E+5. Results are shown over 100 episodes for this algorithm.

The success rate when the algorithm is run for one hundred episodes with a learning rate 1E-4 is 5% for reaching 256. This algorithm did not reach any bigger tiles, such as 512. The average score is 993. The graphs that present the results are as follows:

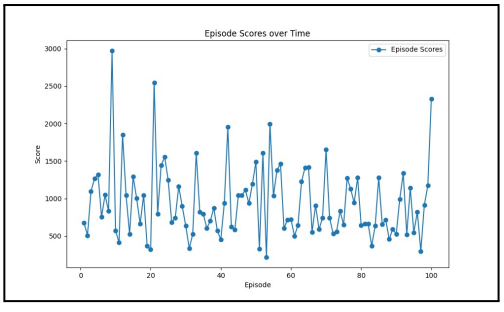

Figure 12. Score vs. Episode (learning rate=1E-4)

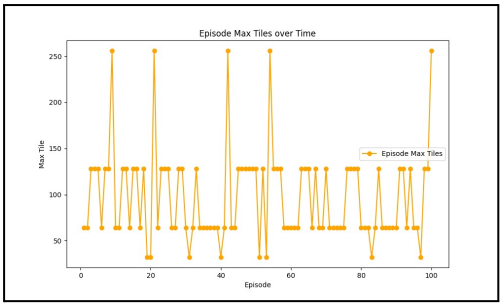

Figure 13. Max Tile vs. Episode (learning rate=1E-4)

The success rate when the algorithm is run for one hundred episodes with a learning rate of 2.5E-4 is 5% for reaching the 256. This algorithm did not reach any bigger tiles, such as 512. The average score is 943. The graphs that present the results are as follows:

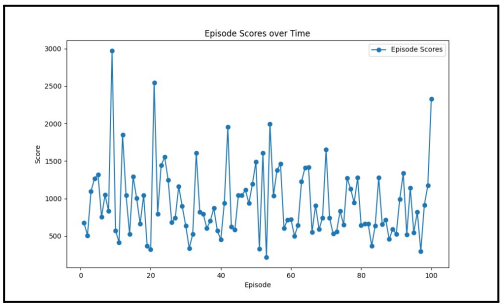

Figure 14. Score vs. Episode (learning rate=2.5E-4)

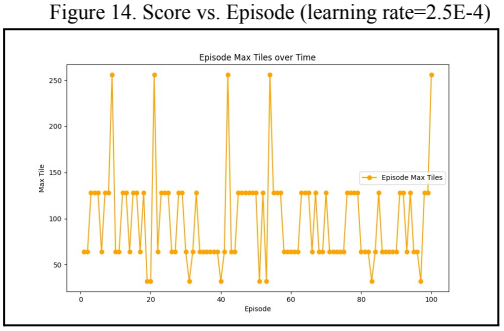

Figure 15. Max Tile vs. Episode (learning rate=1E-4)

The success rate when the algorithm is run for one hundred episodes with learning rate 5E-4 is 4% for reaching the 256. This algorithm did not reach any bigger tiles such as 512. The average score is 967. The graphs that present the results are as follows:

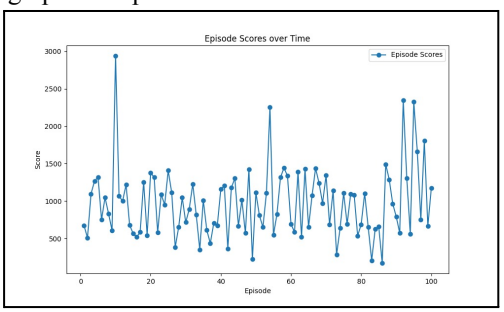

Figure 16. Score vs. Episode (learning rate=5E-4)

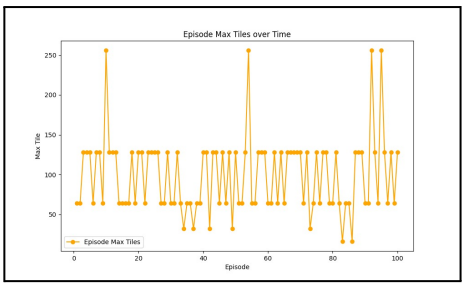

Figure 17. Max Tile vs. Episode (learning rate=5E-4)

The success rate when the algorithm is run for one hundred episodes with gamma 0.8 is 4% for reaching the 256. This algorithm did not reach any bigger tiles such as 512. The average score is 1020. The graphs that present the results are as follows:

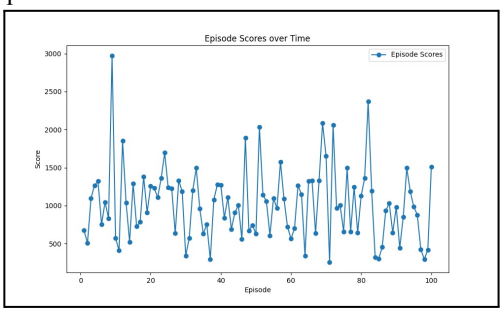

Figure 18. Score vs. Episode (gamma=0.8)

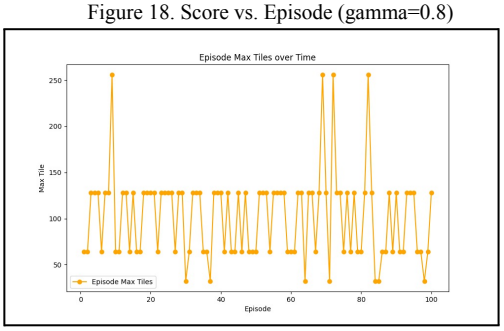

Figure 19. Max Tile vs. Episode (gamma=0.8)

The success rate when the algorithm is run for one hundred episodes with gamma 0.5 is 4% for reaching the 256. This algorithm did not reach any bigger tiles such as 512. The average score is 926. The graphs that present the results are as follows:

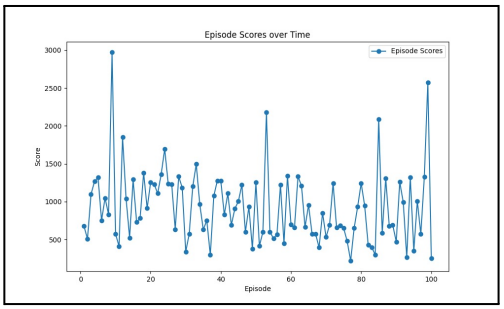

Figure 20. Score vs. Episode (gamma=0.5)

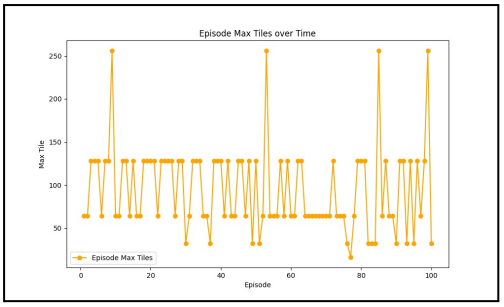

Figure 21. Max Tile vs. Episode (gamma=0.5)

The success rate when the algorithm is run for one hundred episodes with gamma 0.1 is 8% for reaching the 256. This algorithm did not reach any bigger tiles such as 512. The average score is 1069. The graphs that present the results are as follows:

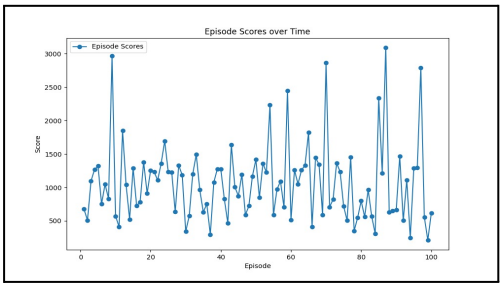

Figure 22. Score vs. Episode (gamma=0.1)

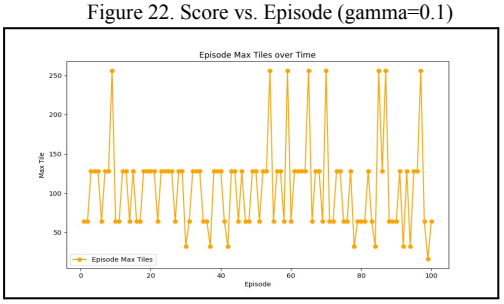

Figure 23. Max Tile vs. Episode (gamma=0.1)

The success rate when the algorithm is run for one hundred episodes with buffer size 1.5E+5 is 6% for reaching the 256. This algorithm did not reach any bigger tiles such as 512. The average score is 993. The graphs that present the results are as follows:

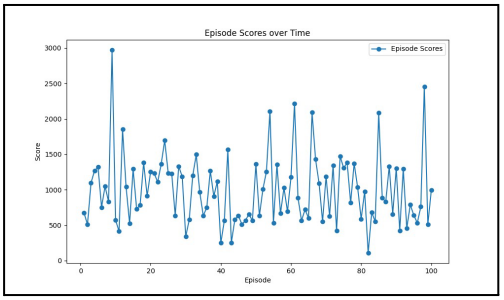

Figure 24. Score vs. Episode (buffer size=1.5E+5)

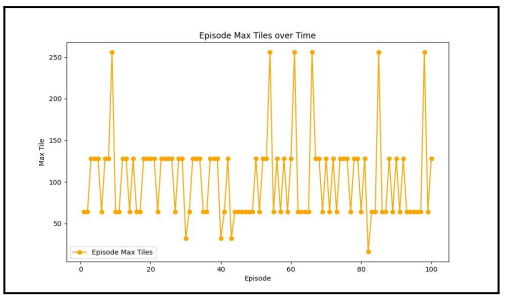

Figure 25. Max Tile vs. Episode (buffer size=1.5E+5)

The success rate when the algorithm is run for one hundred episodes with buffer size 6E+5 is 6% for reaching the 256. This algorithm did not reach any bigger tiles such as 512. The average score is 993. The graphs that present the results are as follows:

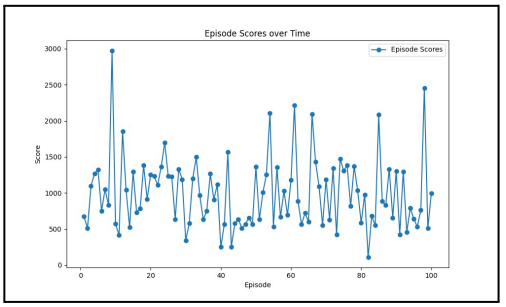

Figure 26. Score vs. Episode (buffer size=6.5E+5)

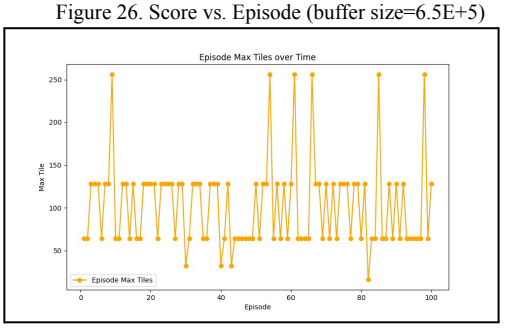

Figure 27. Max Tile vs. Episode (buffer size=6.5E+5)

V. DISCUSSION

This section discusses three algorithms and their results separately and in comparison. Also, potential next steps for each algorithm are provided at the end of each sub-section.

A. Min-Max Tree Search Algorithm

The success rates for depths 4 and 5 in reaching the 1024 tile (40%, 70%) suggest that the algorithm can make reasonably good decisions in the short term. When depth four is considered, the failure to reach the 2048 tile may be due to the limited lookahead provided by the depth. The algorithm might be unable to foresee the optimal moves since the search depth is restricted. The increase in the depth of the search tree to 5 allows the algorithm to explore more potential moves and future states. Even though the success rate for reaching tile 2048 is relatively low, the 10% rate indicates that it explores more potential moves since, for depth 4, the success rate was 0%.

When the depth of the search tree is increased, it allows for a more thorough exploration of possible

moves and outcomes. Hence, the scores in Fig. 5 are higher than the scores in Fig. 2. Also, the distribution of the scores is prone to higher scores when the depth is five rather than four, as can be seen from Fig. 4 and Fig. 7. Maximum tile distribution is also prone to higher scores when Fig. 3 and Fig. 6 are compared.

The low success rate for the 2048 tile may indicate that, even with increased lookahead, the algorithm needs to work on navigating the complex decision space effectively. Hence, as a potential next step, alpha-beta pruning will be implemented to make the search more efficient and less time-consuming. Also, with the decreased run-time, experiments will be done with different tree depths to find the optimal balance between computational efficiency and lookahead.

B. Monte Carlo Tree Search Algorithm

The Monte Carlo Tree Search algorithm explores promising branches of the search tree. It searches towards the nodes that seem more likely to lead to successful outcomes. Hence, this algorithm is more adaptable to computational resources than the Min-Max Tree Search algorithm. It can be seen from Fig. 8 and Fig. 10 that the average performance scores are higher than the Min-Max tree search algorithm's performance. Also, as seen in Fig. 9 and Fig. 11, it can reach the 2048 tile with a 70% success rate and 1024 with a 100% success rate. Hence, the Monte Carlo Tree Search algorithm is more suitable for games with uncertainty or stochastic elements, which is the category of 2048, while Min-Max is designed for deterministic games.

From Fig. 8 and 10, it can be seen that when the depth increases, the algorithm's overall performance also increases. Also, the max tile it reaches is prone towards a higher score. This is because the explored tree depth is increased; therefore, a more thorough exploration of possible moves and outcomes was allowed.

Since it reaches the 2048 tile with a success rate of 70%, the algorithm is mostly complete if no further feedback is received.

C. Double Deep Q-Network (DDQN) Algorithm

The DDQN Algorithm reduces the overestimation of Q-values by decomposing action selection and action evaluation. Currently the success rates for reaching 1024 and 2048 tiles are 0% with this algorithm. The success rates for reaching 256 are between 4% to 8%.

As can be seen from Fig. 13, Fig. 15, and Fig. 17, when the learning rate is set to be 5E-4 success rate for 256 reached is smaller compared to 1E-4 and 2.5E-4. The average score is the highest with 1E-4 which is used as the default value during experimenting with other parameters.

Fig. 19, Fig. 21, and Fig. 23 show the maximum tiles reached with Gamma 0.8, 0.5 and 0.1. It is clear that when the discount factor is 0.1, the success rate for 256 tiles is 8%, which is the highest success rate reached with DDQN in 100 episodes. This result matches the expectations for DDQN results since the lower discount factor means that, the newer episodes have more influence on the Q values.

Lastly, the effect of buffer size on DDQN is explored, and the results can be seen in Fig. 25 and Fig. 27. Change in buffer sizes did not result in any difference in 256 success rate or average scores. Larger replay buffer sizes can store more experiences but require more memory.

Since the algorithm can reach at most 256 tiles, the reward function will be optimized as the next step, and parameter tuning will be continued.

VI. CONCLUSION

We tried implementing a Monte Carlo Tree Search (MCTS), a Min-Max Tree Search, and a Double Deep Q-Network (DDQN) to achieve high results in 2048. Our object was to quantify the success rates of these algorithms with metrics like highest tile, highest score, and average score.

The Min-Max Tree Search Algorithm showed promise with achieving 1024 70% percent of the time when depth is increased to 5 however, with no success in reaching the tile 2048. We plan on implementing alpha-beta pruning to increase the depth, increasing the algorithm's success rate.

The MCTS algorithm reached 2048 70% of the time and 1024 at every episode when the depth was set to 150. This algorithm shows great promise in stochastic games, outclassing its counterparts.

Unlike MCTS, DDQN struggled against this problem. It could only reach 256 4-8% of the time and could not reach higher than 512. The hyperparameters like the learning rates, gamma values, and buffer sizes may have played a role in this algorithm failure and require tuning until the final report to increase its performance.

In conclusion, MCTS outperformed the other two algorithms, and the DDQN algorithms had the worst results. While both of them are tree searches, MCTS outperformed Min-Max Tree Search.

REFERENCES

[1] "2048 (video game)," Wikipedia. Oct. 20, 2023. Accessed: Nov. 26, 2023. [Online]. Available: https://en.wikipedia.org/w/index.php?title=2048_(video_game)&oldid=1185818281

[2] L. H. Chan, "Playing 2048 with Deep Q-Learning (with pytorch implementation)," Medium, Oct. 20, 2023. [Online]. Available: https://medium.com/@qwert12500/playing-2048-with-deep-q-learning-with-pytorch-implementation-4313291efe61. [Accessed: Oct. 20, 2023].

[3] H. Guei, *On Reinforcement Learning for the Game of 2048*, Jan. 2023. doi: https://doi.org/10.48550/arXiv.2212.11087

[4] A. Mehta, "Reinforcement Learning For Constraint Satisfaction Game Agents (15-Puzzle, Minesweeper, 2048, and Sudoku)." arXiv, Feb. 09, 2021. Accessed: Oct. 20, 2023. [Online]. Available: http://arxiv.org/abs/2102.06019

[5] A. Goga, *Reinforcement learning in 2048 game*, Oct. 2017. Accessed: Oct. 20, 2023. [Online]. Available: http://cogsci.fmph.uniba.sk/~farkas/theses/adrian.goga.bak18.pdf