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 final report of the term project, we re-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 2048 in 60% of the episodes when run with a depth of 3. 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.

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

Figure 1. Screenshot of the 2048 environment

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, re-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 A. At the current state of the project, we support a maximum depth of 3 since deep-copying the board is a costly operation and requires optimization. Hence, the environment crashes if we provide a depth greater than 3. 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 re-implement the Monte Carlo Tree Search (MCTS) for the game 2048, we benefited from UCB (Upper Confidence Bound) values to balance the exploration and exploitation during the search. We utilized 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.

The MCTS algorithm was used to select the best action at each step. The step function was called with the current environment state and the number of simulations to run. This function returned the action that had the highest UCB value after running the specified number of simulations. After selecting the action that results in the highest UCB value, the environment was updated by executing that action, and the process continues until the episode terminates.

Lastly, for the Double Deep Q-Network, we followed a similar approach to the course TA's re-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 = 512, and buffer size = 10000.

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

IV. RESULTS

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

A. *Min-Max Tree Search Algorithm*

The Min-Max Tree Search algorithm was tested for tree depth values 2 and 3. The results of these algorithms are shown over 10 episodes. The depth 2 run time was 1975.11s for 10 episodes which means it had an average run time of 197.5. The depth 3 run time on the other hand was indeed slower with 4302.78s in total over 10 episodes and a runtime of 430.27s on average.

The success rate when the algorithm is run for ten episodes with tree depth 2 is 0% for reaching the 2048 tile and 50% for the 1024 tile. The average score is 10594.4, and the maximum score is 20644.

The success rate when the algorithm is run for ten episodes with tree depth 3 is 60% for reaching the 2048 tile and 100% for the 1024 tile. The average score is 24827.2, and the maximum score is 37724. The graphs that present the results for both depths are as follows:

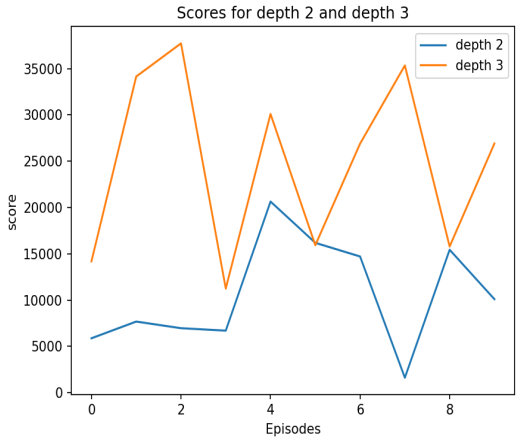

Figure 2. Score vs. episodes for depths 2 and 3

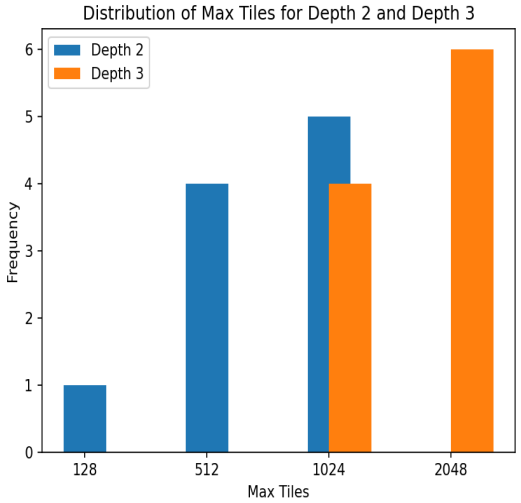

Figure 3. Max tile distributions for depths 2 and 3

B. *Monte Carlo Tree Search Algorithm*

The Monte Carlo Tree Search algorithm was tested for tree depths 100 and 150. When the goal is 2048 (meaning the game ends when the 2048 tile is reached), the total runtime for 10 episodes for depth 100 is 12 minutes 37 seconds. When the depth is increased to 150, the total runtime becomes 18 minutes 31 seconds. 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 80% for reaching the 2048 tile and 100% for the 1024 tile. Moreover, the success rate when the algorithm is run for ten episodes with a tree depth of 150 is 100% for reaching the 2048 tile and 100% for the 1024 tile. The graph that presents the results is as follows:

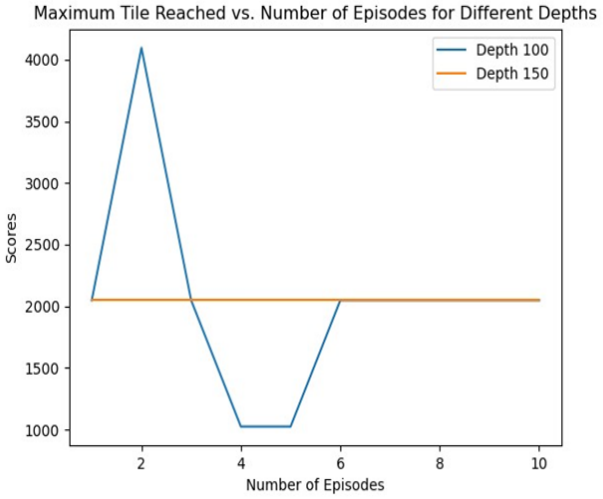

Figure 4. Maximum Tile Reached vs. Episode (depth=100, 150)

When the depth is 100, the average score is 24250. when the depth is increased to 150, the average score becomes 27080. As observed, increasing the depth of the MCTS results in better performance. The graph that presents the results is as follows:

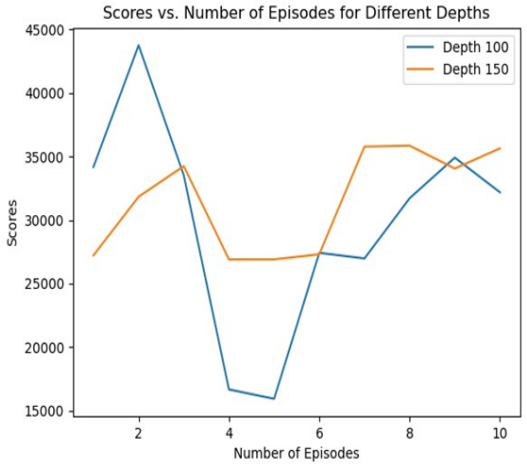

Figure 5. Scores vs. Episode (depth=100, 150)

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 1E-4, 5E+4, and 1E-5. Results are shown over 1000 episodes for this algorithm. The default parameters were set to be 1E-4 for the learning rate, 0.99 for Gamma, 1E+5 for the buffer size, 1 for Epsilon, 0.995 for Epsilon Decay and 0.01 for Minimum Epsilon.

The success rate when the algorithm is run for one thousand episodes with learning rate 1E-4 is 5.7%, 2.5E-4 is 7.2% and 5E-4 is 6% for reaching 256. This algorithm did not reach any bigger tiles, such as 512. The average scores for learning rates are 979, 984, and, 958, respectively.

Fig. 6, shows the scores over the episodes graph for each learning rate. Looking at the graph and average results, we can say that learning rate 2.5E-4 performs slightly better than other learning rates in reaching the 248 tiles as well as having the highest score. Additionally, the max tiles achieved in each episode with the best-performing learning rate, 2.5E-4, are shown in Fig. 7.

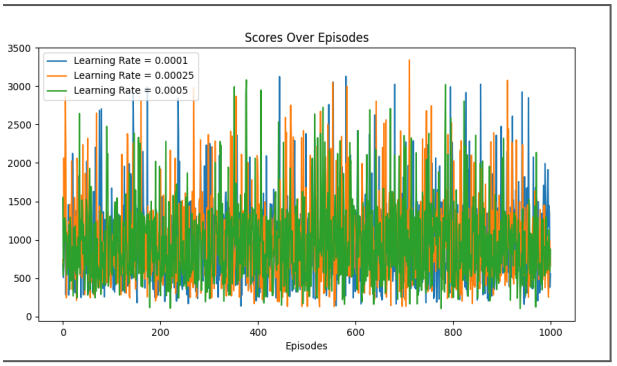

Figure 6. Score vs. Episode (with different learning rates)

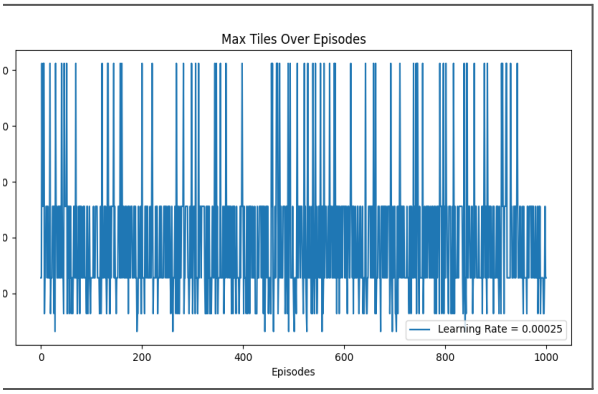

Figure 7. Max Tile vs. Episode (learning rate=2.5E-4)

The success rate when the algorithm is run for one thousand episodes with gamma 0.1 is 5.7%, gamma 0.5 is 4.3% and gamma 0.8 is 6.3% for reaching the 256. This algorithm did not reach any bigger tiles such as 512. The average scores are 928, 924 ,and 963, respectively. Fig. 8, shows the scores over the episodes graph for each gamma value. Looking at the graph and average results, we can say that gamma value 0.8 performs slightly better than other values.

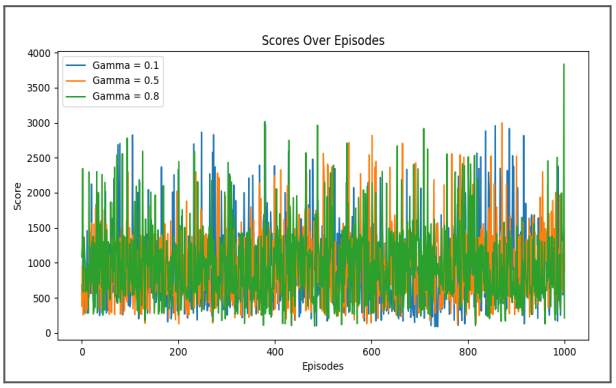

Figure 8. Score vs. Episode (with different gamma values)

The max tile reached by the algorithm with the best performing gamma value, 0.8 for each episode can be seen below in Fig. 9.

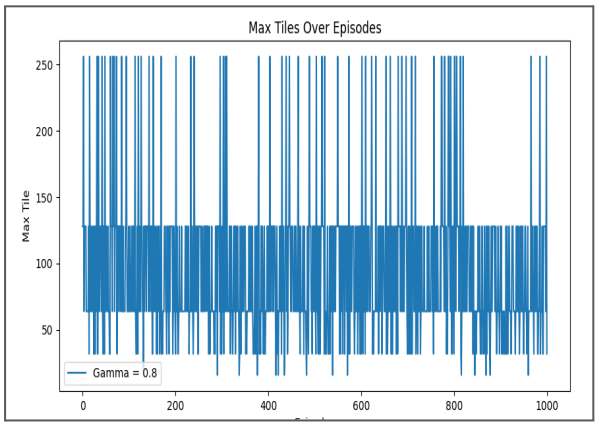

Figure 9. Max Tile vs. Episode (gamma value=0.8)

The success rate when the algorithm is run for one thousand episodes with buffer size 1E+4 is 5.7%, buffer size 5E+4 is 6.7%, and buffer size 1E-5 is 5.5% for reaching the 256 tile. This algorithm did not reach any bigger tiles. The average scores are 979, 989 and 976 respectively. Fig. 10, shows the scores over episodes graph for each buffer size value. Looking at the graph and average results, we can say that buffer size 5E+4 performs slightly better than other values.

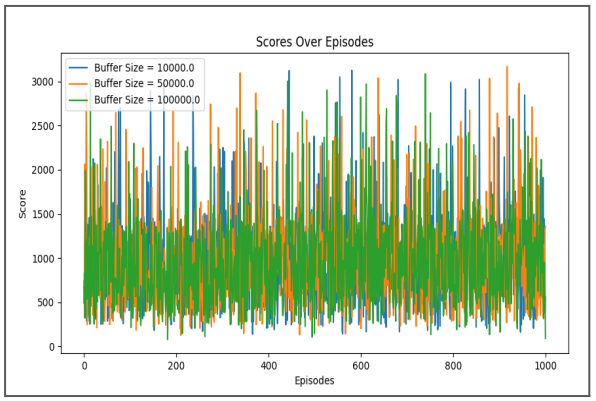

Figure 10. Score vs. Episode (with different buffer sizes)

Additionally, the max tile reached by the algorithm with the best performing buffer size, 5E+4 for each episode can be seen below in Figure 11.

According to our observations, best performance happens for the following parameters: buffer size=50000, learning rate=0.00025, discount rate=0.99. Even with this configuration, average score by episode is 989.64 and 5.7% of the time it can achieve 256, which is its best achieved maximum tile. These results show that DDQN does not come near to the previously mentioned tree-based algorithms in terms of performance.

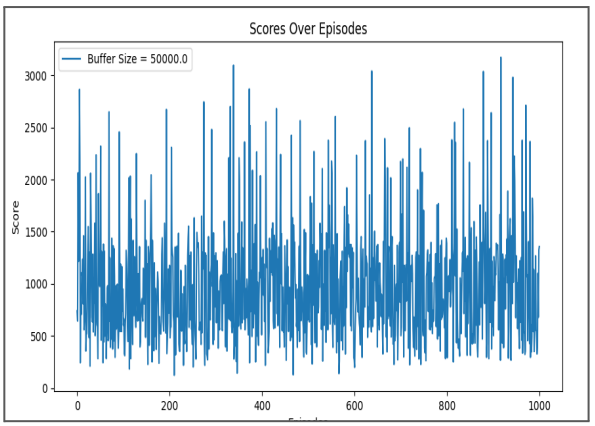

Figure 11. Max Tile vs. Episode (buffer size 5E+4)

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 2 and 3 in reaching the 2048 tile (0%, 60%) suggest that the algorithm can make reasonably good decisions in the short term. When depth 2 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 3 allows the algorithm to explore more potential moves and future states. Even though the success rate for reaching tile 2048 is not guaranteed, the 60% success rate indicates that it explores more potential moves since, for depth 2, 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 depth 3 scores are higher than the depth 2 scores in Fig. 2. Maximum tile distribution is also prone to higher scores when the results of Fig. 3 are compared.

Another aspect to consider is the run times of these algorithms. It was found that the depth 3 re-implementation took almost 4 times the time the depth 2 re-implementation took. One of the reasons for this is that when the depth is increased, the algorithm looks farther ahead, which means more computation and memory will be used. Another reason for the increase in run time is that when the depth is shorter, the game ends more quickly, causing

the overall run time of depth 3 to seem longer than it is.

The min-max algorithm has a decent success rate however, it can be improved. It was seen that the algorithm gave better results with increased depth. However, despite pruning, the re-implementation freezes after depth 3, so in order to further improve this re-implementation, the algorithm has to be re-implemented in a more efficient and performant environment like being written in C.

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. 5 that the average performance scores are higher than the Min-Max tree search algorithm's performance. Also, as seen in Fig. 4, it can reach the 2048 tile with a 100% success rate and 1024 with a 100% success rate again. 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. 4 and 5, 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.

However, getting better performance with increased depth comes with a negative attribute: time. As mentioned before, when the depth is 100, the running time is 12 minutes 37 seconds, but when the depth is increased to 150, the total time becomes 18 minutes 31 seconds. Increasing the depth means a longer run since as the depth increases, the tree becomes larger and more complex. The increased number of nodes directly contributes to longer running times.

Additionally, MCTS has a shorter running time compared to Min-Max Tree Search. This is due to the fact that MCTS explores promising branches rather than every branch, unlike the Min-Max Tree Search algorithm.

Since it reaches the 2048 tile with a success rate of 100%, 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. After the progress report, the number of episodes has been increased to 1000 from 100, resulting in approximately one million steps running for 17 minutes. Hyperparameters have also been explored further to reach better results.

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%, labeling this algorithm as the least successful between the three algorithms re-implemented.

As can be seen from Fig. 6, when the learning rate is set to be 2.5E-4 average score higher compared to 1E-4 and 5E-4. The success rate for 256 reached is 7.2,% and the highest also with a learning rate 2.5E-4.

Fig. 8 shows the scores reached with Gamma 0.8, 0.5 and 0.1. It is clear that when the discount factor is 0.8, the average score and the success rate for 256 reached is higher than other values, 968 and 6.3% respectively.

Lastly, the effect of buffer size on DDQN is explored, and the results can be seen in Fig. 10. The highest average score 989 and the highest success rate for reaching 256 are reached with buffer size 5E+4. Larger replay buffer sizes can store more experiences, resulting in better score and success rate, but require more memory.

The algorithm can reach at most 256 tiles, performing worse than Min-Max Tree Search and MCTS. Some underlying reasons for this could be the complexity of the game, the neural network architecture, hyperparameters, training duration and the rewarding system. Since the game involves a significant amount of strategic depth, it is understandable that MCTS and Min-Max Tree Search perform better than DDQN due to their lookahead capabilities. DDQN, being a value-based method,

might fail to plan strategically, and it is more suitable for control/intent action games such as Pong.

VI. IMPLICATIONS & POTENTIAL NEXT STEPS

It was found that Min-Max Tree Search can be improved by further increasing the depth. However, with the current implementation, increasing the depth is impossible due to performance constraints. The program simply does not run with a depth of 4 or more, and the program freezes. A C implementation could make the algorithms even more optimized and performant, allowing even more depth and increasing the success rate in the process.

Further tuning the parameters of the DDQN (like learning rate, discount factor, and the policy network's architecture) can potentially yield even better performance in 2048. However, our efforts show that even with parameter tuning, DDQN is not suitable for playing 2048 since it lacks foresight and strategic planning abilities compared to the tree-based algorithms. So, "solving" the game 2048 in a formal way, might not be possible with using DDQN.

For DDQN, dynamic hyperparameter adaptation can also result in better performance. Implementing adaptive strategies for dynamic adjustment of hyperparameters during training could be explored. Techniques that allow the algorithm to autonomously adapt its learning rate, gamma values, or other parameters based on its performance could enhance its adaptability to evolving game dynamics.

If we ought to develop an AI model to play the game 2048 indefinitely (i.e., without losing), we should use an MCTS-based approach. MCTS' foresight capabilities, with its dynamic exploration and exploitation of the game tree, provide an excellent approach to this challenge. Its ability to simulate multiple game scenarios and predict their outcomes allows for advanced strategic planning, which is very important in a game like 2048, where long-term strategy is critical. By constantly evaluating the potential future states of the game board, MCTS can make informed decisions that maximize the chances of success and avoid traps that could lead to a loss.

VII. CONCLUSION

We tried re-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 2048 60% percent of the time when depth is increased to 3. However, there is no guarantee of reaching the goal. We can increase the depth even more by optimizing the algorithm to be able to handle such load.

The MCTS algorithm reached 2048 100% of the time and 1024 at every episode when the depth was set to 150. When the depth was 100, it again reached 1024 tiles with a 100% success rate, and 2048 tiles with an 80% success rate, which is still better compared to the Min-Max Tree Search algorithm. Hence, the MCTS 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. It was thought that hyperparameters like the learning rates, gamma values, and buffer sizes may have played a role in this algorithm failure and required tuning.

Thus, a comprehensive hyperparameter tuning process was initiated to investigate the impact of key parameters on DDQN's performance. The learning rate, gamma values, and buffer sizes were identified as potential factors influencing the algorithm's ability to navigate the complex state space effectively.

We can claim that DDQN performs worse than both Min-Max Tree Search and MCTS. There might be several underlying reasons for that. For instance, 2048 involves a significant amount of strategic depth. MCTS and Min-Max Tree Search may perform better due to their lookahead capabilities. Also, the DDQN algorithm, being a value-based method, might fail to plan strategically. It is much easier for DDQN to learn to play control/instant action games (e.g., Pong). Additionally, Deep Q-learning methods can face challenges in balancing exploration and exploitation. The algorithm might struggle to explore the vast state space of the game, which can result in missing out on optimal policies.

In that sense, other threats that might contribute to the insufficient performance of DDQN can include the Neural Network Architecture that we used, hyperparameter tuning, training duration, and rewarding system. Although we explored the capabilities of the DDQN algorithm for solving the game 2048 a lot, there could always remain some threats to validity.

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 and had a shorter runtime since it expands promising branches and is more suitable for stochastic games.

VIII. WORKLOAD DISTRIBUTION

- Ceren Akyar: MCTS experiments, plotting & inferring the results
- Deniz Mert Dilaverler: Implementation of Min-Max Tree Search and its experiments
- Berk Çakar: Implementation of MCTS, DDQN algorithms
- Elifsena Öz: DDQN experiments, plotting & inferring the results
- İpek Öztaş: DDQN experiments, plotting & inferring the results

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