Deep Q-Learning in 2048 and Variations

Memphis Lau, Tyler Lee, Nathanael Nam, Jonah Jung

Abstract

Our objective is to better understand reinforcement learning (specifically deep Q-learning), assessing its adaptability and capacity to formulate strategies for optimal solutions. To accomplish this, we study the model in the context of 2048, a seemingly simple game that has a surprisingly high amount of complexity and a high ceiling for performance due to its randomness and there being no "ending" to the game. For our model we chose deep Q-learning, a complex reinforcement learning model that creates abstract representations of input states, which is well-suited for 2048, since its extremely high amount of possible states are difficult to explicitly record. To better understand the versatility of this model, we introduce modifications to the game, and train the same model on these new games and compare the performance (learning convergence rate, improvement vs random guessing, etc). Ultimately, we can gauge whether or not our model works only specifically for certain situations, or if it can adapt well to different environments.

Introduction

2048 is a single-player game that gained massive popularity in recent years. The game is played on a 4x4 grid and begins with two tiles with the number 2 placed randomly in the grid. Swiping in any direction moves all tiles in that chosen direction, and when two tiles of the same value collide, they combine into a tile with the sum value. The player's total score gets increased by this sum value. In addition, at every swipe, a new tile appears randomly in any empty spot, with a 90% chance of being a 2 tile and a 10% chance of being a 4 tile. The objective of the game, as given by the name, is to achieve the 2048 (2^{11}) tile before the board fills up with no legal moves.

We aim to build a deep Q-learning model that can play the game of 2048 better than humans can by recognizing patterns and calculating optimal actions at each state. After training a model to do so, we want to observe how the model adapts to different game environments. Thus, we will create modifications to the game, train the model on thousands of games, and compare the performances. We intend to understand how randomness, complexity, and obstacles affect the reinforcement learning model and draw conclusions about its generalization.

Background

Reinforcement learning in the game 2048 has been well studied, since the game has relatively simple mechanics, a small board size (the weights of any network do not have to be as high-dimensional compared to a game like chess or Mario), and Markov properties. Many different types of architectures and methods have been applied to the game, the best being N-tuple networks trained with reinforcement learning, specifically multi-stage temporal difference [1]. Other strong attempts at beating the game include using Monte Carlo Tree search, expectimax search, stochastic MuZero, and supervised learning using other pretrained models [2].

In the realm of reinforcement learning "players", the best models have used advanced temporal difference learning, which is beyond the scope of this course. With regards to Q-learning, because the game has a large number of states, deep Q-learning works well. Mnih et al. first presented a learning algorithm based on deep Q-learning with convolutional neural networks, which was then applied to Atari games, outperforming humans in many games [3]. This method, explained more in following sections, has been thoroughly studied with the game 2048. However, none have been able to achieve the 2048 tile as reliably as other methods mentioned above [4]. There have also been multiple github projects online that test deep Q-learning on the game, all with unique training/implementation/architecture [5]. There are projects with mini-batch gradient descent, convolutional layers with different sized filters, epsilon-greedy policies, log2 normalization, etc. We combined methods from multiple sources to write our code [6].

Our project aims to test deep Q-learning's adaptation on variations of the game. To our knowledge (searching through Google and Google Scholar), there have been no published papers or attempts to do any research with 2048 variations.

Theory

Deep Q-learning is a fundamental concept in reinforcement learning, notably applied to gaming scenarios. In the game 2048, Deep Q-learning is encoded to empower the model in decision-making, assessing in-game moves, and refining its comprehension of state-action pairs. Updating these pairs approximates the optimal action-value function, or Q-function, representing expected cumulative rewards. Q-values, derived from the Q-function, estimate the total expected rewards an agent can achieve from a given state, action, and policy. Temporal difference learning updates weights and Q-values using the Bellman equation, incorporating rewards and estimated Q-values of subsequent states. To expedite implementation, experience replay is employed, enabling learning from stored and replayed game states in small batches to prevent dataset distribution skewing. However, in high-dimensional state spaces, optimizing replay mechanisms or exploring prioritized experience replay becomes crucial to address limitations in replay memory's coverage of game diversity.

Batch gradient descent is integrated into the Deep Q-learning algorithm, enhancing stability by using different batches for learning. The approach smoothens convergence towards the optimum, balancing gradient averaging over diverse samples. Parameters are updated through gradient descent in each batch, minimizing mean squared error between predicted and target Q-values. Both the predicted and target Q-values are approximated using convolutional neural networks, which processes complex information, recognizing spatial patterns in the 2048 game board and mapping visual inputs to actions. Convolutional and fully connected layers capture patterns, relationships, and global dependencies. Training across epochs optimizes hidden layer parameters, producing a probability distribution at the output layer for optimal moves.

The epsilon-greedy strategy is deployed to manage exploration-exploitation tradeoffs in reinforcement learning. Epsilon serves as a crucial factor controlling an agent's behavior, initialized at a higher value and progressively decreasing during training. A higher epsilon encourages random action selection for enhanced environmental understanding, while a lower epsilon prioritizes exploitation of learned policies and Q-values. Applied in 2048 training, the epsilon-greedy strategy facilitates exposure to diverse game states, fostering a more robust policy adaptable beyond training samples. As the agent gains experience, the strategy dynamically shifts from exploration-heavy to exploitation-focused behavior, aligning with improved game knowledge.

Methods

As mentioned, we implemented Deep Q Learning in the game 2048. We start by creating a simulation of the game. The first necessary component of the game is to create the game board. This is done in Python with a list of lists representing a matrix. At first, the board is initialized with zeros to create a blank board. The next component of the game is the function that adds random tile values to the board for each game step. The function first searches if there are empty spaces on the board where the new value can go, it searches for the zeros in the matrix. If there are empty spaces, then the function will calculate a random probability. If this random probability is less than or equal to 0.9, it will randomly choose an empty space and enter a value of two. In the latter case that it is greater than 0.9, a four will be entered into the matrix. The next function of the game is one that checks whether or not the game has ended or can be continued. For the actual mechanics of the game, we first define a few helper functions: a flip function, a transpose function, and a slide function. The slide function keeps track of the score, as well. The helper functions are combined to create the functions for moving up, down, left, or right.

With the game logic correctly designed, we move on to the deep Q-learning model. This type of reinforcement learning uses a neural network to approximate the optimal Q value. In our model, we use a convolutional neural network to do this. Although any type of neural network can theoretically be used, we decided to use a convolutional neural network to analyze patterns in small windows of the board. In our model, we define a convolutional neural network with two convolutional layers, each layer using a ReLU activation function. There is no max pooling because we did not want to minimize the already small dimensions of the 4x4 grid. The convolution layers then lead to a fully connected layer that then outputs to four nodes. Each of these nodes indicates the model's probabilities of potential moves during a given state. The max value of these four nodes is the optimal action for a given state which is used to make the next move in the game.

The implementation of the model starts with a loop of M iterations. M describes how many times we want to train our mode; each M represents a full game played out. In a single instance of the game being played out, the model checks if the game is still playable. If it is, then we calculate the maximum Q value for the state to find the optimal move. Because we want our model to take into consideration both exploration and exploitation, we use an epsilon-greedy method. In our exploration stage, we take a random move and gather information. In the exploitation stage, we take the maximum Q value from the convolutional neural network and apply it to our game board. Eventually, as more games are played out, we slowly decrease the value of epsilon so that we do more exploitation than exploration now that we have observed a large part of the game. Concurrently, we run backpropagation. The backpropagation is calculated using the mean squared loss between the predicted Q value and the target Q value. The target Q values are calculated through a replay function where we essentially memorize the previous states and actions taken, and we compare them to those. After the M iterations, the model is finished training.

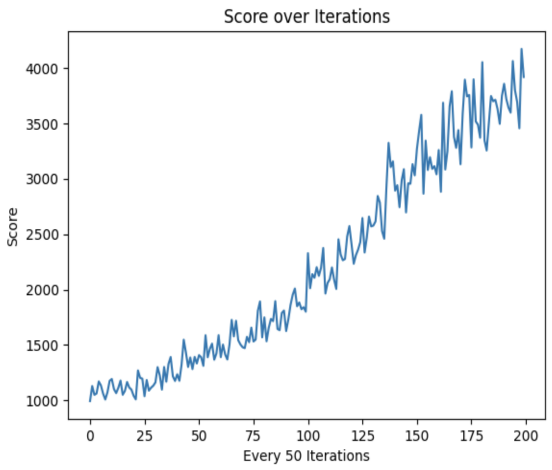
We repeat this same process for 3 variations of the game 2048. The first change is utilizing a 5x5 grid, instead of the original 4x4 board. With this adjustment, the function to create a new game slightly alters, and the dimensions of the convolutional neural networks are changed. The input layer is now a 5x5x25 tensor, thus the convolutional layers directly after also increase in dimension. Our code's reproducibility and use of variables made this change simple. The next variation is a 4x4 game with no randomness. As opposed to the original game rules in which a 2-tile spawns with 90% probability (4-tile otherwise) in a random empty spot, we designed a game board that generates 2-tiles always and places them with the same rules every swipe. If the board were labeled 1 through 16 starting from the top left corner and going left, the new 2-tile would spawn in the empty square with the smallest number. With a non-random game design, we hoped that the model would be able to converge towards an optimal solution. The last variation is a board that initializes with a 1-tile. By the rules of the game, the 1-tile cannot combine with any other tile, since every other tile will be greater than 1. The 1-tile acts as a block, restricting connections between similar tiles and greatly increasing the chance of losing the game.

Results

Original Game

After 10,000 games played, our model learns the game of 2048 relatively well. As we can see from the plot below, the average score continues to increase as the model is trained more. This graph was generated by taking the average score of every 50 games played. As a reminder, the score is the sum of every tile created throughout the duration of the game. The variation and spikes in the plot are the result of the randomness of the game. The relative performance of the model is measured by its comparison to an agent that randomly picks one of the directions for every move. After running 500 games with this baseline agent, the average score is 919. After 5,000 iterations, the model is, on average, 2.22 times better than the baseline agent. After 10,000 iterations, the model is 4 times better than just playing random moves. The maximum score achieved is 10,508, with a board that included a 1024 tile.

Number of Episodes	**Average score (25 before and after)**	**How much better than baseline**
1000	1066.56	1.159345
2000	1201.84	1.306393
3000	1518.56	1.650666
4000	1625.52	1.766931
5000	2046.56	2.224599
6000	2594.48	2.820185
7000	2923.44	3.177763
8000	3243.28	3.525427
9000	3379.12	3.673084
10000	3697.92	4.019618



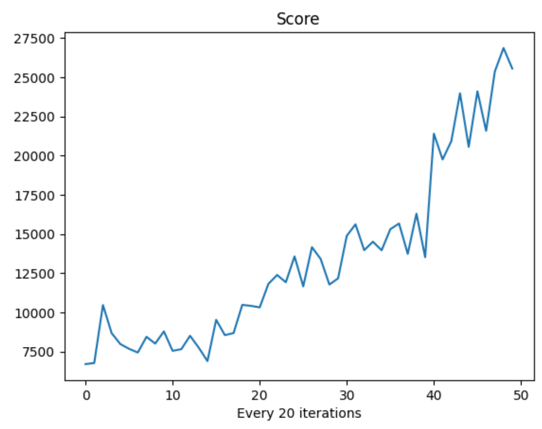
Plotting the ratios versus the number of episodes follows almost a linear pattern. Fitting a linear regression model to the data gives us the model:

Ratio of Average score to Baseline = 0.0003425 * *Number of Episodes* + 0.649

Of course, there is no guarantee that this pattern will remain linear over hundreds of thousands of iterations. However, under the assumption that it does, our model would be almost 35 times better than the baseline agent after 100,000 episodes and 69 times better after 200,000 episodes. This provides hope that our model would be able to beat the game and reach the 2048 tile if we had the space and time to train the model more.

5x5 Board

Now, we test the model when the game is played on a 5x5 board. As explained in Methods, the convolutional neural network is slightly larger. Because of the larger board and thus longer gameplay time, we only had the resources to train the model through 1000 games. However, through just those 1000 games, the model shows promise of being very effective in learning the game. After running 500 games with a baseline agent, the average score is 6159. After 1,000 iterations, the model is, on average, 3.46 times better than the baseline agent. In comparison, with a regular 4x4 board, the model after 1000 iterations is only 1.16 times better.



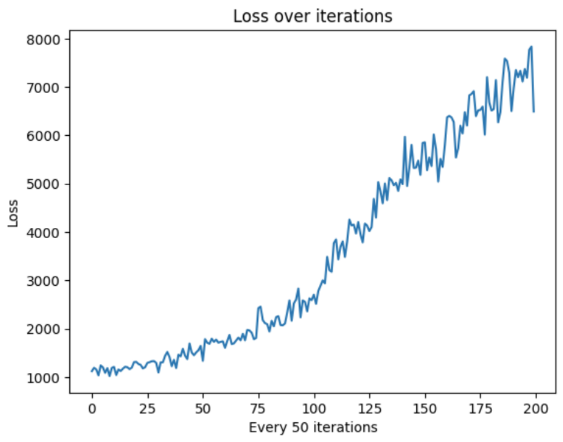
How much better the model is compared to the baseline agent versus the number of episodes follows almost a linear pattern. By similar processes, we achieve the formula:

Ratio of Average score to Baseline = 0.0029178 * *Number of Episodes* + 0.617

Again, there is no guarantee that this pattern will remain linear over thousands of iterations. Under the assumption that it does, our model would be almost 30 times better than the baseline agent after 10,000 episodes and 59 times better after 20,000 episodes.

No Randomness Game

Our interpretation of the game with no randomness means that after every swipe, a 2 tile spawns always. It spawns in the first empty spot going from the top left corner to the right (if the board was labeled [1,2,3,4],[5,6,7,8]...). As expected, the plot for the average scores is much less jagged than previously, as we have removed a lot of randomness from the game. After 10,000 games played, our model learns this new game better than it did for the original 2048 mechanics. After running 500 games with a baseline agent, the average score is 1072. After 5,000 iterations, the model is, on average, 2.3 times better than the baseline agent. After 8,000 iterations, the model is almost 6 times better than just playing random moves. It does not make significant improvement over the next 2,000 iterations.



Plotting the ratios versus the number of episodes does not follow a linear pattern as closely as the original game. The ratio also plateaus after 8,000 iterations, so it would be difficult to model.

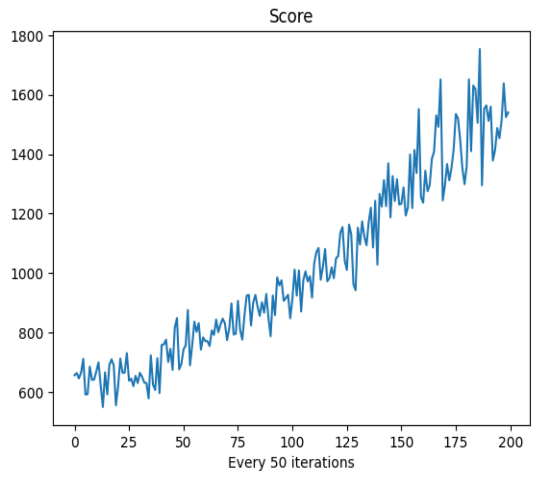
Block Introduced

Now, we test the model when the game is played with a tile that cannot combine with any other tile. In the game mechanics, we began every game with a tile with a value of 1. In this variation of the game, the model does not effectively learn. We see large spikes in the plot of scores over iterations. After running 500 games with a baseline agent, the average score is 578.2. After 5,000 iterations, the model is, on average, only 1,5 times better than the baseline agent. After 10,000 iterations, the model only performs 2.4 times better.

Plotting the ratios versus the number of episodes follows almost a linear pattern. Fitting a linear regression model to the data gives us the model:

Ratio of Average score to Baseline = 0.00016198 * *Number of Episodes* + 0.8746

With the assumption that this linear pattern holds, our model would be 17 times better than the baseline agent after 100,000 episodes and 33 times better after 200,000 episodes.



Analysis

Our model performed significantly better in the original game of 2048 when compared to simply random guessing. The score steadily increased at a relatively constant rate over the course of its iterations. We are highly certain that our model would have continued to improve with an increase in episodes, but due to time and computation constraints, we decided to train the model on only 10,000 games. By the end of our 10,000 iterations, our model plays up to 4 times better (in terms of score) when compared to simple random movement. It is important to note that a score to the factor of 4 means more than just that the model is 4 times better than random movement. As the game goes on longer, it becomes increasingly difficult to avoid losing the game, as there are larger tiles on the board that cannot be combined as easily. The model's ability to survive long enough to quadruple the baseline score is very impressive. The Q-learning model, through a relatively small amount of training, was able to recognize successful patterns through approximating the target and predicted Q-value at each state. Breaking into the model and exploring exactly how it understands 2048 game patterns would be a very interesting next step for this project.

In the 5x5 modification, the model achieves better results than with the original 4x4 board. Of course, direct comparison is difficult, given that the increased board size leads to higher complexity and a possible amount of states. Because of the larger neural network, as well as longer gameplay time, we chose to decrease the amount of iterations to 1,000 in response to our limitations in resources. Despite this, we still observed significant improvements over these 1,000 games. The model scored 3.46 times better than an agent that played the board with random movements, which is superior to the model for the original game, performing 1.16 times better after 1,000 iterations. The same result can be seen when comparing the slopes of the lines of best fit modeling the scores versus episodes. The success of our model on a 5x5 board is most likely due to the convolutional neural network having more inputs and more dimensions. More data and information to capture allows the model to perform better. In addition, it is more difficult to lose the game (fill up all 25 games), so the model has more time to learn which moves have positive rewards.

In our game with no randomness, following intuition, the model plays relatively better than that of the original game. At the end of the training, the model is 6 times better than a random movement player. However, the model starts to decline in its improvement near the end. It reaches its peak performance at around 8,000 iterations, after which the improvements seem stagnant/negligible. This is a noticeable difference when compared to the performance of our baseline game, which steadily increased throughout the course of its iterations. One possible explanation for this is that the removal of randomness allowed the model to converge faster, meaning that it reaches its peak performance at a faster rate thus causing the decline in performance increases to start faster. Perhaps, our model finds its version of an optimal solution after around 8,000 games.

The least efficient model we created was that for the game 2048 with a block introduced. After running 10,000 iterations, the model only performs 2.4 times better than random guessing, which is significantly lower than the other models. This could be because the network does not learn how to accurately assess the potential reward/punishment of a block with a value of 1. Perhaps, instead of viewing the block as an obstacle, the model treats it as half the value of a 2-tile, when in reality, it has no value. By introducing this block, we limit the environment's possible states and the agent's possible moves, which limits its potential to increase performance.

Conclusion

Through these results, we see that the model's performance has high variance depending on its application. This supports the claim that our model of deep Q-learning isn't extremely adaptable to any environment, as it depends on the certain aspects of the environment to be able to perform well. In the context of 2048, the performance depends on the level of complexity, level of randomness, and ceiling of performance. By adding more complexity to the game (as in the case of our 5x5 model), the model has much more potential to increase its performance which allows it to improve at a much faster rate. As for randomness, by taking out the randomness and allowing the agent to learn where the next tile will spawn allows the model to better fit to the game, which also increases the performance. Introducing the block limits the agent's possible actions, which decreases its potential for performance. Therefore, though the model does perform better than a baseline agent for all of the game variations, the rate at which the model increases is not consistent throughout, which may argue against the versatility of deep Q-learning.

Though the performance of our deep Q-learning is better than random guessing, as well as the average human performance, there may still exist better models for this situation. Some examples include temporal difference learning, N-tuple networks, and Monte Carlo tree search. Temporal difference learning's advantage comes from being able to learn continuously (it doesn't need to wait for the episode to finish to learn, unlike other methods), increasing efficiency and addressing our limitations in resources in the case of deep Q-learning [7]. As for N-tuple networks, its advantage comes from being able to well recognize patterns in the game, which may allow it to develop more effective strategies that make use of patterns that appear throughout the gameplay. Lastly, Monte Carlo tree search excels in decision-making and finding the best actions, which is crucial to high performance [8].

Further research based on the findings of this paper could be on exploring how different CNN architectures, specifically exploring different dimensions and number of layers, could affect performance. With a game with a small grid like 2048, there can be interesting findings regarding the limits of complexity for the model. Another topic of discussion is unrolling the DQN architecture and understanding what patterns our model recognizes and how it finds optimal actions at each state.

Data Availability

All of our data was produced by our own simulations so it is not proprietary and does not belong to anyone.

Contribution Statement

Nathanael coded the game logic to create 2048 and wrote the code for the model including creating the game data and running it through the neural network. Weakly involved in the literature review. Memphis developed and coded variations of the game, performed the literature review, and analyzed results from the different tests. He was not as involved in understanding the theoretical aspects of the model. Tyler created presentation slides, conducted data research on our chosen model and participated in the analysis of results and making the conclusion. Weakly involved in code analysis and review. Jonah researched the theory behind the different machine learning methods used and how they work together. Involved in literature review, creating plots and visualizations, and organization. He was weakly involved in analysis code and review.

References

[1] Szubert, M. and Jaskowski, W: *Temporal Difference Learning of N-Tuple Networks for the Game 2048*, available at https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=6932907
[2] Guei, Hung: *On Reinforcement Learning for the Game of 2048*, available at https://arxiv.org/ftp/arxiv/papers/2212/2212.11087.pdf
[3] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. iedmiller, "Playing Atari with Deep Reinforcement Learning,"
[4] J. Downey. "Evolving Neural Networks to Play 2048." May 2, 2014). [Online Video]. Available: https://www.youtube.com/watch?v=jsVnuw5Bv0s
[5] https://github.com/topics/2048-ai
[6] https://github.com/navjindervirdee/2048-deep-reinforcement-learning
[7] https://www.autoblocks.ai/glossary/temporal-difference-learning
[8] https://builtin.com/machine-learning/monte-carlo-tree-search