Reinforcement Learning in 2048

Matthew Chung
Harvard University
matthewchung@college.harvard.edu

Abstract

We want to train a Reinforcement Learning agent using Deep Q Learning to achieve the winning 2048 tile in the game of 2048. After determining the state and action spaces, we experiment with a few different model architectures and compare their performances after training. We also attempt various methods of rewards and evaluate the differences in play style by the agent, as well as the overall results.

1 Hypothesis/Question

The original goal was to train a reinforcement learning agent to play the game of 2048. 2048 is a popular single-player puzzle game that is played on a 4x4 grid. The objective of the game is to combine tiles with the same number to create a tile with the value of 2048. The game ends when the grid is full and no more moves can be made.

At the start of the game, the grid contains two tiles with a value of 2 or 4. The player can move the tiles in four directions - up, down, left, or right. When the player makes a move, all the tiles on the grid slide in the chosen direction until they hit the edge of the grid or another tile. If two tiles with the same value collide, they merge into a single tile with double the value.

After each move, a new tile with a value of 2 or 4 is added to the grid in a random empty position. The player can continue making moves until the grid is full and no more moves can be made, at which point the game ends.

The score in 2048 is calculated by adding up the value of all the tiles on the grid. The player can achieve higher scores by creating larger tiles, such as a 1024 tile or a 2048 tile, and by creating chains of merges with multiple tiles.

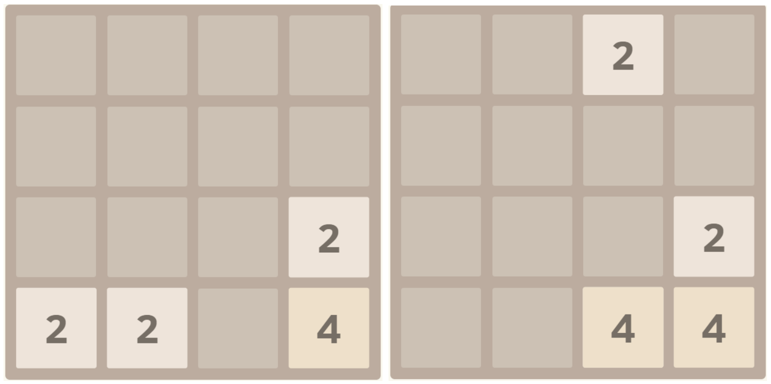

Figure 1: Game state prior to a move right (left) and after the move (right)

Despite being a simple game, the most variable part of training the reinforcement learning agent would most likely come from the reward. Figuring what exactly would lead the agent to form an optimal strategy to beat the game was the main part I wanted to experiment with in this since there are multiple ways to go about it. The most simple way would be to allow the agent's only reward to be the score of the game, since ultimately, increasing the score is done through combining tiles, which is what we want in order to reach the 2048 tile. However, there are other strategies in the game that a human would use normally, and evaluating the effectiveness of training the agent in this manner would be something interesting to compare to the strategies it comes up with on its own.

2 Brief Literature Review

When figuring out which algorithm to use for the game of 2048, I wanted to choose one that hadn't been widely used already with high success rates but wasn't already shown to be suboptimal. After reading through Dedieu and Amar's paper[1] on using Probabilistic Policy Networks for reinforcement learning in 2048, it did not seem like there was much success at all from their methods.

Probabilistic policy networks are typically used in reinforcement learning problems where the agent must learn a stochastic policy that balances exploration and exploitation. In some problems, it may be desirable for the agent to take suboptimal actions with some probability to explore the environment and learn more about it. In these cases, a probabilistic policy can be useful because it allows the agent to sample actions from the policy distribution and explore the environment.

In this instance, it seems that for the game of 2048, it may have some use since a lot of the times, there isn't always one specific move that will work every time when given a certain state space. There will be a chain of moves required to advance in the game in the most beneficial but safest way possible, as there is certainly a tradeoff between finding new tiles to add together while trying not to fill up the board too fast. In some instances, it may be better to perform a move seen as non-beneficial in that moment, but it could very well lead to possible moves which allow for more progression than previously.

However, it might not be a good idea to use probabilistic policy networks to train an RL agent to play 2048 because the action space in 2048 is discrete and relatively small. The game of 2048 has only four possible actions - up, down, left, and right - and the agent can determine the best action to take by computing the Q-values of each action in a given state and choosing the action with the highest Q-value. In this case, a deterministic policy function that maps each state to a single action is sufficient to achieve good performance in the game.

Using a probabilistic policy function to solve 2048 could add unnecessary complexity to the problem and may not lead to significant improvements in the agent's performance. A probabilistic policy function is more appropriate when the action space is continuous and the agent needs to sample actions from a probability distribution to explore the environment effectively. Thus, I decided to go with Deep Q-Learning.

Based on reading the reinforcement learning paper in 2048 by Hung Guei supervised by Dr. I-Chen Wu[2], it seemed that Temporal Difference Learning was extremely effective when it came to 2048, as when combined with n-tuple networks, it has been shown to consistently beat the game. One level above that would be Multistage Temporal Difference Learning, which was shown to be capable of achieving even the 65536 tile, which is extremely impressive. However, Guei stated that Temporal Difference Learning significantly outperformed Q-learning, so I wanted to see how Q-learning performed for myself.

Something else interesting that I saw in Dedieu and Amar's paper[1] was their reward that they used. Initially, they thought of the idea of the reward simply being 1 if the agent achieved the 2048 tile and 0 if it didn't, but clearly this reward was too simple even for 2048. A time-homogeneous reward was considered where the largest tile at the end would be the reward, but this doesn't take into account that the longer the game goes, the more likely of a higher score, larger tiles, and better chances at the 2048 tile. Thus, they decided to go with the sum of the tiles on the board as the reward. However, just making a move that doesn't merge any tiles will spawn another tile on the board, effectively giving a reward for a move that may be detrimental to the game state, so I decided to go with the increase in score after an action as the starting reward.

3 Methods/Steps

Here I will describe all the various things that I tried to see how the agent would respond, as well as all the important necessities for the game itself. All of this corresponds to just the implementation of the game played through matrices, but I will explain later on how I tested it visually to see exactly what was going on. All code written by myself except for parts of the model architecture, which was assisted by Binxu Wang[3].

3.1 State space

Originally, I had the state space set up as a 4 x 4 matrix where each element of the matrix represents a tile space on the board. The tile space can either be empty of filled, and its filled values are powers of 2 starting from 2 to infinity. Of course, it is not actually possible to get that high of a tile in the game, so I capped the highest value at 2^{15} since its the highest tile you can actually achieve in the game with every tile lined up in increasing order, despite the chances of reaching this state being pretty much impossible (incredible amounts of luck would be needed in regards to where the new tiles spawn on the board).

However, after discussing with Binxu, it seemed like a better idea to go with a 4 x 4 x 16 state representation, where we still have the same 4 x 4 board originally but instead of assigning a value to each space on the board, we now have a boolean-like indicator where each of the 16 layers represents on the powers of 2, and the merging of 2 elements in the same layer combines them into one element in the layer above.

The action space is quite simple, as it is a discrete space with 4 moves: up, right, down, left. However, there are certain moves in certain game states that will not progress the game, and I'll mention how I dealt with these later on.

To implement this in code, I used the gym library to create an environment for the game. Besides the state space and action space already mentioned previously, I had to implement 2 main functions for my 2048 environment: step and reset.

To implement step, I needed to take in the current 2048 game board state and perform the action given on it, which was essentially just moving all the tiles in the direction of the move, combining tiles wherever necessary. For this, an implementation of the merging of tiles was necessary to figure out the next state of the board. The next part was to figure out the reward after the action is complete, and I'll dive much deeper into the reward in a later section.

Next up was reset, which needs to clear the current game and start a new one. This was quite simple, as all that I really needed to do was to create a new board of 0s, set that to the current board, add two starting tiles randomly on the board, and reset the score to 0. The last step would be turning the 4 x 4 matrix back into its 4 x 4 x 16 representation to stay consistent with the state space we defined earlier.

3.2 Model

Since 2048 is a game with a large state space, there are many possible states that the game board can be in. Deep Q-Learning can be effective in such environments because it uses neural networks to approximate the Q-values of state-action pairs, allowing it to generalize across similar states. In order to maximize the score, we must include considerations such as merging tiles, creating chains of merges, and avoiding getting stuck. Deep Q-Learning is a model-free algorithm, meaning it does not require any prior knowledge about the environment, and can learn complex strategies by trial-and-error.

Using the stable baselines3 library, I was able to train my custom gym environment using Deep Q-Learning. I decided to try 3 different convolutional neural network architectures to compare the results against each other. The first type was a convolutional neural network with 2 2D convolutional layers where the game board turns into an "image" and each tile space on the board is a pixel, The second type was a convolutional neural network with 2 3D convolutional layers where we stick with the 3D representation of the board described in the state space. The last type was a convolutional neural network that is a merge of the 2D and 3D ones.

4 Results

Here will be where all the training and testing is discussed. The reward breakdown will also be broken down to show some interesting insights as to how the reinforcement learning agent's strategies differed when altering the reward parameters

4.1 Training

I trained each type for 30 million steps each and here are the trajectories below.

Cyan: 3D + 2D | Green: 3D | Purple: 2D

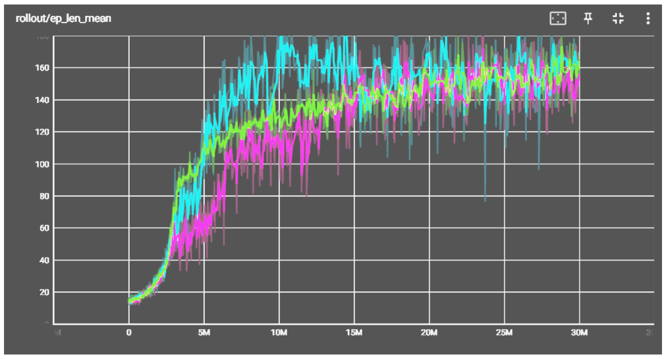

Figure 2: Average episode lengths over the training period

Cyan: 3D + 2D | Green: 3D | Purple: 2D

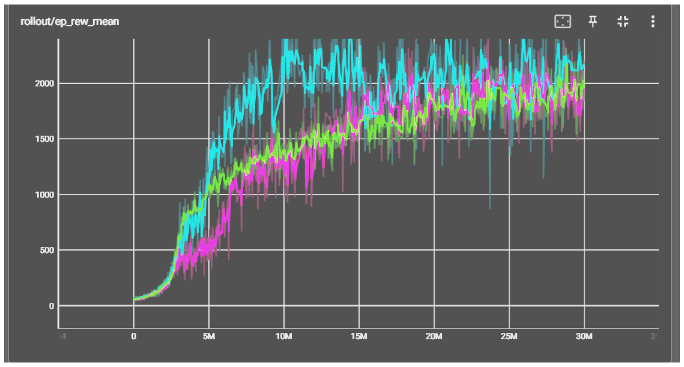

Figure 3: Average total rewards over the training period

Something that is quite apparent in both of these graphs is that all 3 types have a spike in both average episode length and average reward at similar points during training. It may possibly mean that the agent discovered a strategy that worked much better than how it had previously been doing. What's interesting to note is also that all 3 trajectories finish at about the same place after 30 million steps of training. When episode length finishes as the same spot, that means that the average number of steps per game is about the same for all 3. When average total reward finishes at the same spot, that means that the rewards are extremely similar.

However, the variance in the 3D and 2D merged Convolutional Neural Network seems to be much higher than the other two, and this may actually be the reason why it turns out that it outperforms the other two quite significantly, since it means that the 3D and 2D version was capable of reaching higher highs, thus potentially getting to significantly higher scores than the other two.

4.2 Testing

After finishing training, I tested each type for 10000 episodes and plotted the score distributions.

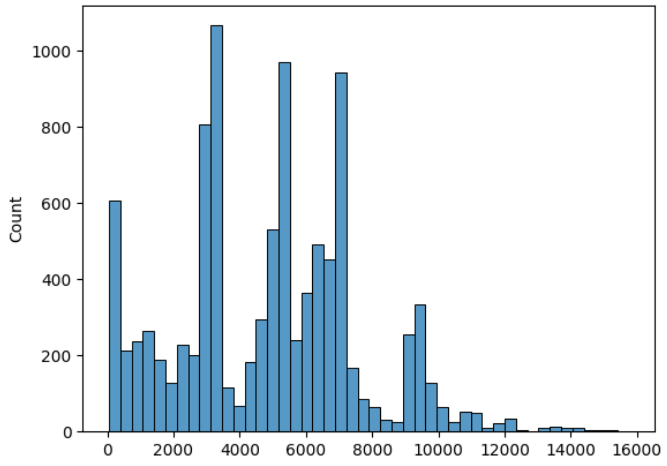

Figure 4: Score distribution for 2D over 10000 games

The score distribution for our 2D version already turns out to do better than what could be done with random moves according to Dedieu and Amar[1]. Random moves could never achieve the 256 tile for them, so these scores are a significant improvement upon that. While I was able to reach the 256 tile on some random move games, they were very rare. A score of around 2000 will consistently mean there's a 256 tile on the board, and we can see the majority of games went past that. The best performances from this architecture was the 1024 tile occurring for about 10% of the games. The majority of games where able to get to the 512 tile, so this result was decent but not where we wanted to be. One main observation from the way the agent played was that it understood to place the largest tile in one of the corners of the board, and I will elaborate more on the strategies in a later section.

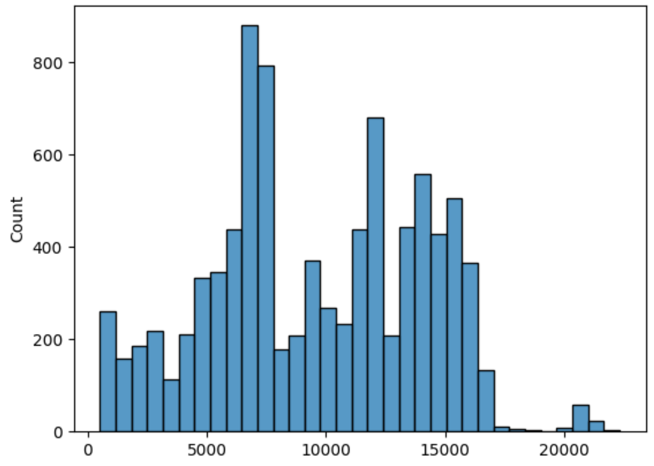

Figure 5: Score distribution for 3D over 10000 games

The 3D version's score distribution clearly improved upon the 2D version's scores since we can clearly see games where the score ended with over 20000. The 20000 barrier indicates that the 2048 tile has been reached, thereby beating the game. This is a huge step up, as we were unable to reach the 2048 tile at all in the 2D version. While getting the 2048 tile as seen in the score distribution was rare, it was often enough to not be considered a one-off fluke due to luck or things of that nature, and was able to achieve it 3% of the time. This time, the 1024 tile was achieved about 40% of the time, which is way higher than the previous version. Something to note is that in this version, the agent was capable of fixing its own mistakes when moving the largest tile out of the corner by turning whichever tile was currently in the corner into the largest tile by repeated attempts at merging all possible tiles into it.

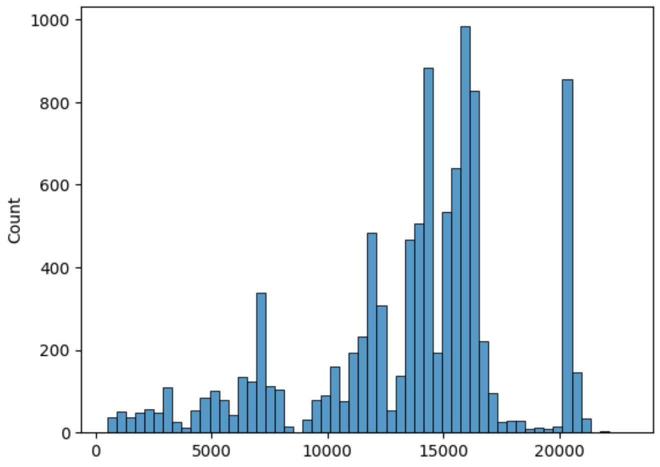

Figure 6: Score distribution for 3D + 2D over 10000 games

The final version was able to far surpass both of its predecessors and achieve the 2048 tile at a 20% rate. In fact, it had a 90% rate of getting the 1024 tile, which means it was able to get close to winning on many occasions. When watching the agent play, there were many games that it would be able to line up the 1024 tile, 512 tile, 256 tile, and so on but towards the last few steps before merging all the tiles, the 1024 tile would be moved out of place, resulting in a loss before the big merge. This version's agent was capable of moving the largest tile out of position and permute the board to move it back into position at times, as opposed to trying to turn another tile into the largest.

4.3 Visualization

The utilization of webscraping helped significantly in watching the testing of the agent play the game. The selenium library was used to open the 2048 game website and let the agent play repeatedly on it. This was done by modifying the environment for when I set a boolean that noted whether I wanted the web version to be used to true, all operations would be transitioned onto the web, so there was no longer a need to generate new tiles on my own, but some modifications were done to press the retry button on the screen when a game was finished. This was an amazing tool to watch the agent play in real time and understand the process behind the moves.

4.4 Reward and Strategies

Up until now, the results have all been garnered with the simple reward of how much the score increased from the last move. Here we will further explore the other possible parameters for reward.

There's an argument to be made for whether it's best to let the agent learn almost completely hands-free with just the score and let it figure out a strategy on its own, or whether it's better to nudge the agent in certain directions to play more in a certain way. Letting the agent learn on its own could potentially allow it to play the game superhumanly whereas nudging it may stunt its potential by playing like a human does and become unable to surpass that.

The most optimal way for humans to play the game has been to put the largest tile in the corner. This has been a well-known strategy for a long time, and it's one that I use myself religiously when playing the game. The thought process behind it is that keeping it in the corner reduces the possibility of it moving around, and thus it becomes sort of an anchor point where all your moves revolve around merging tiles into that largest tile. This logically makes the most sense since the less all your larger tiles move around, the easier it is to line multiple of them up for a chain of merges to create the largest tile possible, since we know that each move will move all the tiles on the board.

We can see that when training just on the reward of increase in score, the agent learns by itself that this is the most optimal way for it to play, as with all 3 architectures it has done this for every game it has played. However, they will go about maintaining this condition in different ways and with different priority levels. There seems to be a correlation between the amount of importance placed on the corner being the largest tile and total score/highest tile. The 2D version places the least emphasis on it, where the largest tile will frequently move out of the corner and will not always be moved back immediately after in the case a new tile does not spawn in the largest tile's former place. This most definitely caused the agent to lose games quickly since it cannot always get lucky enough to move the largest tile back to wear it originally was.

The 3D version had similar issues of moving it out of the corner, although not as frequently, but it still occurred quite a lot. However, it had learned a way to fix its mistake by treating the new corner tile as the largest tile, where it would focus all of its attention to merging as many tiles into it as possible to turn it into the largest tile. This may be due to the mistakes of moving the largest tile out of the corner earlier on into the game, so fixing the new tile with this method is easier as there are less tiles needed to merge. However, once the 1024 tile is achieved and is moved out of the corner, the only way to replace the new corner tile as the largest is to turn it into a 1024 tile as well, and at that point, the game will be won. Unfortunately, many times, the 1024 tile is not the only large tile on the board and will be accompanied by some 128, 256, and even 512 tiles that have already been positioned next to the 1024 tile. If the 1024 tile is moved, then it now becomes a roadblock to merging tiles with the new corner tile, decreasing the chances of winning significantly due to clogging up the flow of the board. This may have contributed to the struggles of achieving the 2048 tile, as running into this issue would be difficult to solve with this method of fixing.

The 3D + 2D version saw a new method of fixing the out of place largest tile. At smaller tile values, it would sometimes just use the 3D version's method of fixing the board since it could work much more easily, but other times, it would rearrange tiles to move the largest tile back into the corner. Of course, it didn't work every single time, since bad luck sometimes spawns a tile in a position where the largest tile can no longer be moved back, and in this case it would resort to the 3D version's method again. However, I found this method of permuting the board to be extremely fascinating, as this is something I think the vast majority of humans would not attempt to do since the unpredictability of tiles spawning would deter such a method as it requires adaptability, whereas the 3D version's method of rebuilding the largest tile from scratch in the corner is much more stable and consistent to work with. When looking at some people's expectimax AIs[4] play 2048, even though their scores significantly outperformed my reinforcement learning agent's scores, those AIs employed a similar method of shifting the largest tile around to get it back into a corner. However, their AIs were capable of moving the largest tiles into different corners, which was something my reinforcement learning agent was not quite capable of.

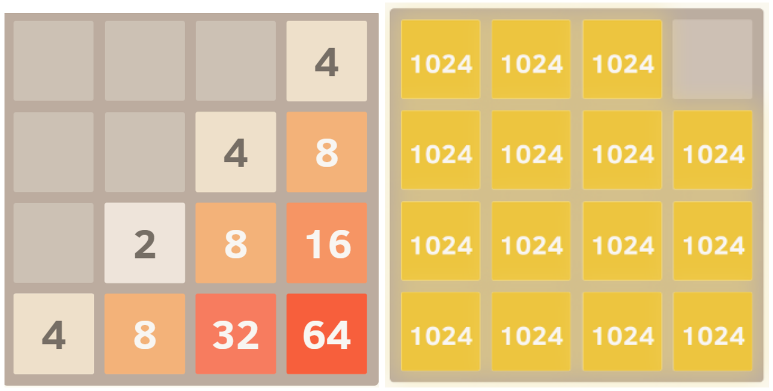

Figure 7: A monotonic board (left) and a smooth board (right)

After this one main tactic of cornering the largest tile that was already learned naturally, I wanted to see if other heuristics might improve the agent's performance. Some of the main ones I found from nneonneo and ovolve on stack exchange[4] included rewarding open spaces, monotonicity, and smoothness. Open spaces refers to having less tiles on the board as a whole, which would mean that there are more tiles merged and less straggler tiles just out there unable to be merged. Monotonicity refers to having adjacent tiles being one power down and smoothness refers to adjacent tiles being the same as seen in figure 7. However, each of these things turned out to have their own issues that I noticed when watching the agent playing the game after training. Rewarding for having less tiles was problematic since early on, the game would reward significantly as there are generally less tiles. In certain situations later on, there would be better moves that could be made where more tiles were needed to merge many tiles together, but the agent would attempt to have less tiles immediately due to the nature of the reward, sort of like a greedy algorithm. The issue with monotonicity was that it was too general for the board.

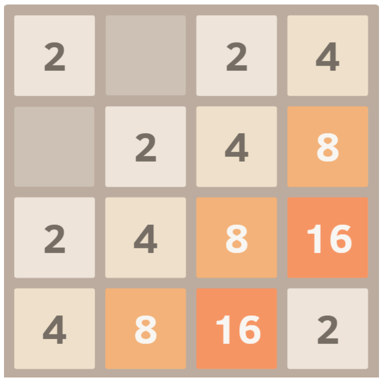

Figure 8: A generally monotonic board that is detrimental

Figure 8 shows a board that would be rewarded greatly for its monotonic properties, but this type of board would lead to losses quickly. The agent was unable to recognize that there were certain tiles out of place at times since the rest of the board would in a way band-aid those problematic tiles and overshadow them. In many cases, this would lead to quick losses since tiles that spawn in the middle of formations would go undetected and destroy the board from the inside. The problem with smoothness was that it was too unrealistic to accomplish anything. The amount of game states where there are many of the same tile next to each other is just too rare. Having one or two tiles next to other same tiles wouldn't be enough to offset the amount of penalty incurred from not being in a state like that. However, toning down the impact of it or removing the penalty side of it altogether would pretty much make it a non-factor as it just didn't come up often enough to truly have a large effect.

Overall, it would take some more experimenting and research to find an optimal reward to train on, as I don't believe that a simple reward like score, despite being effective, is the most optimal reward to use.

5 Github Repository

The code can be found at

`https://github.com/matthewc423/2048-RL`

There is also a video of the reinforcement learning agent playing 2048 and beating the game on the github titled "2048win.mp4". I will also copy the google drive link to the video as well at

`https://drive.google.com/file/d/1EHnPZ9t1pVk0MtxHxjzSsKKYSnm1MAyf/view?usp=sharing`

The reinforcement learning agent in the video has been trained on the 3D2D architecture.

References

[1] Dedieu, Antoine & Amar, Jonathan. (2017) *Deep Reinforcement Learning for 2048*. MIT.

[2] Guei, Hung. (2023) *On Reinforcement Learning for the Game of 2048*. National Yang Ming Chiao Tung University.

[3] Wang, Binxu. (2022) *Training Agents to play 2048 with 3D CNN*. `https://github.com/Animadversio/py2048`

[4] nneonneo & ovolve. (2014) *What is the optimal algorithm for the game 2048?*. `https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048`