Reinforcement Learning Agent to Play 2048

Alex Eringis AERINGIS@SEAS.UPENN.EDU
Porter Killian PKILLIAN@SEAS.UPENN.EDU
Joseph LaForgia JLAF@SEAS.UPENN.EDU

Abstract

Our project aims to create a reinforcement learning agent to master the game 2048. We create a convolutional neural network that learns a policy from a look ahead agent as a baseline. We then implement a policy gradient to approximate the optimal policy. From there we plan on using the policy network to estimate the value function via regression to allow our agent to master 2048.

1. Introduction

1.1. Motivation and Related Work

We aim to grow our understanding of Reinforcement Learning by implementing and experimenting with a variety of methods that attempt to master 2048. 2048 is a game we have all played and appreciated, developing varying degrees of mastery since its genesis a few years ago, and we enjoy working on Reinforcement Learning algorithms to do the same.

In the past there have been many attempts to create a 2048 player using reinforcement learning of varying success (Kondo & Matsuzaki, 2019). Thus we believe that by using deep reinforcement learning and can create a model that plays the game nearly perfectly. The fact that there are only four possible moves a user can make: up, down, left, or right, means that we should be able to model a very clear policy of moves and rewards given states of the game. A normal person can likely consistently get to 2048 or beyond after they become experienced with the game, but we aim to create an agent to be able to get close to the highest possible score.

We chose to apply policy gradient to this problem, implementing two classic algorithms. Policy gradient based algorithms have been show to converge to an optimal policy (Fu & Hsu, 2016). We believed that the complexity of the game and the length of play it takes to receive rewards would lend itself to focusing on the policy instead of the value of the various boards. We explore two algorithms and how we tuned them to try to maximize success, and then discuss what held us back, what we learned, and what we would have done differently given more time.

University of Pennsylvania, CIS 419/519 Course Project.

1.2. Gameplay

2048 is a single player puzzle game where the user slides numbered tiles on a 4 by 4 grid in an attempt to reach the number 2048, or the highest tile possible. Tiles are all powers of 2 and tiles of the same number can be combined to create a single tile of their sum if they bump into each other during a slide. After each slide, a new tile of value 2 or 4 is added to the board randomly, and you lose the game if the board fills up completely and no slide would remove a tile. A game that achieves the 2048 tile normally takes about 2000 moves. The score achieved is the sum of all new tiles created throughout the game, e.g. merging two four blocks would add 8 to the score. We choose to evaluate the quality of a play through as the largest tile achieved, as that is inline with how people play.

1.3. Board Configurations

Where m is the number of moves, there is at least 4^m board configurations as each move yields a different board, There is actually many more than this because the board spawns in a new tile with each move randomly. Since the largest tile achievable is 2^{16} assuming only 2 spawns, so there are 16 different blocks and so an upper bound is $16^{16} = 4^{32}$ board configurations. Additionally seeing if any specific tile can be reached given a starting position is in NP-Hard (Langerman & Uno, 2018). This throws out any methods that involve learning value or policy functions directly, therefore we turn to function approximation, via Neural Nets.

2. Methods

After reviewing our class materials, we set off to design a Policy CNN starting with a random policy. This proved to be quite slow due to the many board configurations as described above, and this agent rarely achieved even the 2^7

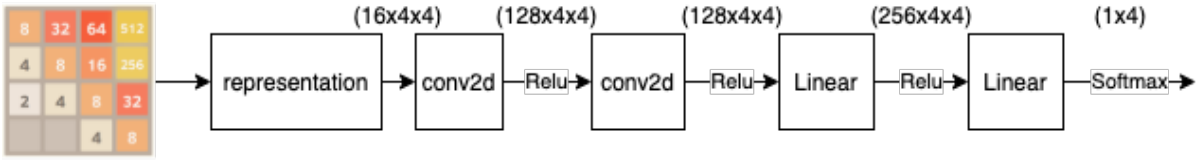

Figure 1. Our Supervised Learning CNN

tile.

2.1. State Representation

We start off using Selenium Webdriver to allow our algorithms to play the game on it's website. This was rather slow leading us to create a representation of the game using python. We modeled the game as a 4x4 numpy array exactly how it is in it's played version, with values of 2^x. We later chose to represent the 2048 board as a $4x4$ numpy matrix with values being log_2 of the value in the real game to further speed up computations in our neural nets. Lastly, we moved to represent the game board as a $16x4x4$ numpy array, with the ith value of the first dimension being 1 iff the board contains 2^i at that first dimension's latter two dimensions, and 0 otherwise. This representation allowed our CNN to have more channels for convolutions and also made all the values in the array 0 or 1, increasing the speed and accuracy of our CNN.

2.2. Supervised Learning

We then decided to provide our models with "expert" data by recording our own games. We played a bunch of games and used Selenium webdriver to record every board configuration and our corresponding move. We then fed this data to a CNN to train up a base-level policy network.

The structure of our CNN is shown above in figure 1. It is two layers of convolutions and two fully connected linear layers, each with 2x2 kernels and zero padding. Layers are activated by Relu with the exception of the last layer, which uses the softmax activation, so as to return probabilities of each of our actions. We chose the softmax activation to make designing our policy gradient easier. We use the negative log-likelihood for our loss function and the Adam optimizer.

2.3. Data Creation using BoardTree

We realized we did not have enough data to properly train this policy network, so we turned to design a simple game player that could achieve decent results to produce much more data. This agent uses a recursive board tree structure such that each node has four children, each being a board corresponding to the board one move away from its parent board. The children boards always assume a bad spawn resulting from the move to achieve that board, that is they spawn a new 2 or 4 that is near its larger cells, penalizing taking risky moves. An example of a risky move is one that opens up the edge or corner of the board that holds the board's bigger values. This worst-case spawn was accomplished by spawning a new value into the lowest available cell. At each move, the agent looks 5 levels down its tree, and chooses the action that has the largest score summed down that branch.

This agent achieved pretty strong results, the distribution of the max tile achieved is recorded in figure 2.

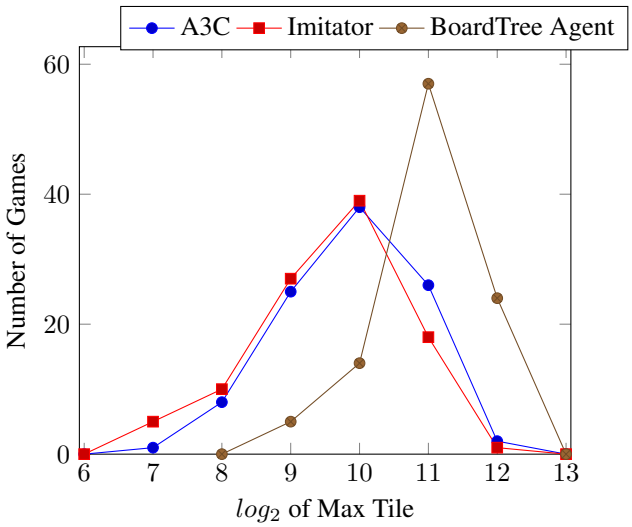

Figure 2. Max Tile Distribution Achieved by BoardTree, imitator, and A3C trained agents

2.4. Supervised Learning Revisited

We used our BoardTree agent to record a lot of "good" moves that we could use to train our supervised learning policy network. We recorded every single move and corresponding board to use as data points. We trained up the network from figure 1 on this data for 40 epochs. The policy did not yield as good scores as the BoardTree agent, but it played much faster (100 games using the BoardTree took over 5 hours), giving us hope that with some policy gradient and then a development of a value network, we would be able to develop an ideal player. The distribution of the

max tile achieved from playing 100 games following this policy network are recorded in figure 2, compared to the BoardTree agent.

2.5. Policy Gradient: REINFORCE

We set out to try and improve upon our baseline policy network using the REINFORCE algorithm (Sutton et al., 2000). This was an introductory policy gradient algorithm that allowed us to get a handle on the basic of policy gradient implementations but it did not take us much further than that. The REINFORCE algorithm has success when applied to simple networks for simple problems, but 2048 is a game with complicated structure and strategies that could not be improved upon by REINFORCE. After many runs of this algorithm and fine-tuning of parameters, the loss calculated by REINFORCE only diverged, taking our policy with it. We do not spend much time on REINFORCE in this paper because we see this implementation as our first step in understanding policy gradients and not really as a step in getting closer to beating our game, as this algorithm made no progress toward that end.

2.6. Policy Gradient: A3C

We moved to apply a more complex policy gradient algorithm to our network, namely A3C. A3C is a form of deep reinforcement learning that uses asynchronous gradient descent for optimization of deep neural network controllers(Mnih et al., 2016). This implementation required us to delve into torch multiprocessing and other new topics to build a functioning program. We followed the algorithm description from the original A3C paper very closely, following their advantage estimate: $R_t - b_t$ where R_t is the accumulated discounted rewards and b_t is the current value estimate for each of those states. The loss function for our policy network was thus this advantage times the log of the probabilities our our selected actions. We calculated the advantage in batches of size 64. As part of the Actor-Critic frame work, we also had to train up a value network. This network had the same structure as our policy network, except it had only one output node instead of three, and used a L_2 loss function. We also followed one last piece of advice from this paper, adding the entropy of the output of the policy network to our loss function to encourage exploration. With this implementation was saw pretty marginal gains, if any, in our policy network. The algorithm was still very prone to diverging, and rewards would minutely improve in its time before divergence. We were able to experiment with this algorithm on machines with 4, 8, and 16 cores using Torch Multiprocessing. All were laptops and so we were limited by the computational power, but we did see our best results from the 16-core machine.

We moved to update our advantage function using the Generalized Advantage Estimate: $GAE = r_t + \gamma b_{t+1} - b_t$, we calculated and discounted this advantage in batches of 64 (Schulman et al., 2015). Similarly our loss function was this advantage times the log of the probabilities our our selected actions. We also added an Adam optimizer with shared parameters between threads. These two alterations to A3C helped our policy learn more stably, allowing a more noticeable growth in rewards and subsequently the max tile achieved in games, as shown in figure 2.

2.7. Reward Functions

Through this process, we experimented with a few reward functions:

- $R_1 = merges(s^1, s)$
- $$R_2 = \begin{cases} .1 & \text{if } n \geq 12 \\ .01 & \text{if } n = 11 \\ .001 & \text{if } n = 10 \\ .0001 & \text{if } n = 9 \\ 0 & \text{otherwise} \end{cases}$$
- $$R_3 = \begin{cases} 1 & \text{if } n = 11 \\ 0 & \text{otherwise} \end{cases}$$

  where n is log_2 of the largest tile
- R_4: incremental trainings using R_2 style

R_1 simply awards each move for the number of tiles it combined. This was our first intuition as we thought the goal of the game was to merge tiles to get larger tiles. While this is true, this reward function actually shortened the length of the games over time from the baseline policy. We attribute this to the fact that a player that aims to merge as many tiles as possible will lose early because they are playing too greedily. 2048 requires significant foresight, this reward would emphasize strategies that would focus on combining any tiles possible as soon as possible, leading to a board full of smaller tiles, ending the game.

R_2 applies an exponential award for the max tile on the board. This reward function was significantly better than the previous, because it rewarded good games instead of what might be good moves, strengthening the policy network slower.

R_3 was a similar approach to R_2, except only reward games that reached the winning tile, and subsequently ending them. R_3 had similar-or-worse success to R_2. We speculate that this was because games that marginally made it to the 2048 tile would have their value be the same as those that made it to 2048 easily and potentially would have

made it to the 4096 tile. This is problematic because policies that bring a game to achieve barely 2048 would do that with very low percentage, but these policies would be strengthened just as much as legitimately good policies, so we found R_2 worked better.

R_4 Another method we tried was to apply different reward functions over different runs, saving the model each run and using it in the next run. We would start by using R_2 and then save the best model achieved. Then we would run on that model using R_2 without the reward for the max tile being 9, and save the best model. We would repeat this process until we were only rewarding for the max tile being 11 or higher. The idea was that we would shape our policy more slowly, trying to eliminate the worst parts of the policy incrementally. Unfortunately, this method still had the model diverge by the second or third run. In the future we would like to attempt such a method on a less complex game that still had the type of hierarchical structure as 2048.

After significant testing, we found our best reward function was a combination of R_2 and R_3. Namely, we rewarded only for the 2048 tile but allowed the game to continue until it lost, getting a small reward for each move it lasted after achieving the 2048 tile. This proved to focus our policy on winning the game while still emphasizing policies that win the game better than other winning policies.

3. Conclusion

Unfortunately, we were not able to train an agent that could beat the game consistently like our board tree look ahead agent could. Despite this, we learned a lot about deeper neural networks and policy gradient implementations and pytorch best practices. We were also able to learn a lot about this problem and the pros and cons of our methods and decisions along the way.

3.1. Randomness

Most literature runs A3C on games with less randomness. 2048 is a game for which a good strategy focuses significantly on the randomness of the tile spawns. A good player makes moves to actually limit the randomness of the game by leaving less unoccupied blocks. This level of strategy proved to be very challenging to develop in our policy network. To experiment with this, we eliminated the randomness of the spawn by always making the new tile spawn in the highest-right-most block available. This succeeded, allowing our agent to continue to improve further, achieving the $2^{11} = 2048$ tile over 30% of the time, compared to the 26% of the same A3C run on the actual random game.

3.2. Length of Play

Compounding the problems with randomness in the game of 2048, the games are also very long. A winning game will require at least 2000 moves, each is highly based off randomness making it challenging to learn a policy. The long games combined with the high randomness throughout makes on-policy models very unstable, which is why we ran into so much divergence in all of our policy gradient trials. The complexity of the game made it very challenging to have a network of adequate complexity and enough time to train a network before diverging, making this problem very challenging to solve.

3.3. Machine Limitations

A3C gets its power from many workers being able to run in parallel, each gradually updating the global brain. As Google Colab only offers CPUs with 2 cores or a single GPU, and our machines were much slower with 4 or 8 cores, we lacked the computational power to be able to test our methods efficiently. This combined with the strict due date of the project did not give us enough time to test everything we wanted to, as we would have to run A3C for a few hours before knowing id it diverged or not each time we wanted to try something new. Had we had better foresight into this problem, we would have chosen a non asynchronous method that would've been able to capture the power of Google Colab more effectively for our runs, such as DQN.

3.4. On-Policy Approaches

On-policy approaches such as the Actor-Critic framework and Reinforce often have the problem of diverging loss functions and local convergence, especially for more complex and challenging games such as 2048. With our new appreciation of 2048 and reinforcement learning techniques, we realized it would've been better to go with an offline approach to try and solve this problem. We also could have avoided the machine limitations more effectively because strong off-policy approaches do not require the multiprocessing that A3C does for strength, so our project could have remained on Google Colab.

Acknowledgments

Dinesh Jayaraman provided us with guidance on where to take this project throughout during office hours. Rahul Rajasekaran helped us to optimize our design to try and incrementally improve our results during various meetings. Ramaman Sekar helped us develop our insight into the algorithms we were exploring in office hours.

References

Fu, Justin and Hsu, Irving. Model-based reinforcement learning for playing atari games. Technical report, Technical Report, Stanford University, 2016.

Kondo, Naoki and Matsuzaki, Kiminori. Playing game 2048 with deep convolutional neural networks trained by supervised learning. *Journal of Information Processing*, 27:340–347, 2019.

Langerman, Stefan and Uno, Yushi. Threes!, fives, 1024!, and 2048 are hard. *Theoretical Computer Science*, 748: 17–27, 2018.

Mnih, Volodymyr, Badia, Adrià Puigdomènech, Mirza, Mehdi, Graves, Alex, Lillicrap, Timothy P., Harley, Tim, Silver, David, and Kavukcuoglu, Koray. Asynchronous methods for deep reinforcement learning, 2016.

Schulman, John, Moritz, Philipp, Levine, Sergey, Jordan, Michael, and Abbeel, Pieter. High-dimensional continuous control using generalized advantage estimation, 2015.

Sutton, Richard S, McAllester, David A, Singh, Satinder P, and Mansour, Yishay. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pp. 1057–1063, 2000.