Reinforcement Learning Methods for 2048

Vincent GEFFROY, Xiyao LI, Guillaume RIQUET

***Abstract*—In this paper, we present two different Reinforcement Learning approaches to solve the 2048 game. We define the performance of our agents by the final score and the maximum value obtained. When we get 2048, the problem is considered to be solved. To solve the problem by RL algorithm, we try a Policy neural network and a Pure Monte Carlo Game Search with different game playing strategies. Then we present the results obtained with our agents and discuss the shortcomings of our research and the possible improvement directions.**

Keywords: Reinforcement Learning, Neural Network, Pure Monte Carlo Game Search

I. INTRODUCTION

Reinforcement Learning is now widely used to control how agents make optimal decisions when facing uncertainty. In this project we will show how reinforcement learning can be used to solve the game 2048, which introduce an extra random factor during game playing.

We first implement a **Policy Network** with Reinforcement Learning which controls the agents' actions in 2048. We use the front-propagation algorithm for the game playing and the back-propagation for the learning process. Since this method did not give a satisfying result, we tried a completely different approach, the **Pure Monte Carlo Game Search**. We explore several game playing strategies during the reinforcement phase, and some of them have outstanding performance in 2048 game playing.

In the following sections, we will present the environment settings of the game, the detailed implementation, the results achieved, a comparison of each method and possible improvement directions. The code of our implementation can be found here. [1]

II. BACKGROUND AND RELATED WORK

2048 is a puzzle video game designed in March 2014. After the game's first release, it took people's interest immediately due to its simple rules. Some discussions on whether the problem could be solved by Artificial Intelligence quickly appear.

The first published AI algorithm for 2048 appeared on *StackOverFlow* in March 2014 [1]. The core algorithm used by the authors is the **Minimax with Alpha-beta pruning**, which is also commonly used in information-symmetric game AI such as chess. The approach considers the game as adversarial. The AI does traditional depth-limited MiniMax searching, assuming the opponent will place the worst possible tile. So the AI makes the decision on the worst possible case, which is an event that may be highly unlikely to happen. So the algorithm is clearly sub-optimal. The authors later give some improvements on the different parameters in the algorithm. It has a probability of more than 90% of getting 2048, but the probability of getting 4096 or even 8192 is not high.

Methods for solving the 2048 problem using Reinforcement learning have emerged as well. Reinforcement learning is an area of machine learning inspired by behaviourist psychology, concerned with how agents take actions in an environment to maximise some notion of cumulative reward. For example, in this link, the author uses a two-layer **Convolutional Neural Network** to train the agent. The success rate of 1000 games for getting a 2048 cell is 10%, and a 4096 cell is 0.05%.

Inspired by the RL algorithms that we find on Internet, we would also like to try some approaches we have seen in the course INF581 and combine some with the currently existing methods. In the next session, we present the implementation of our agents and the reason for our choices.

III. THE ENVIRONMENT

2048 is a sliding block puzzle game with a 4×4 grid where the value of each cell is a power of 2. The player can choose four actions: up, down, left and right. When one action is performed, all cells move in the chosen direction. Any two adjacent cells with the same value along this direction merge to form one single cell with the sum of the two cells. In each step, a cell will appear randomly in an unoccupied cell. The players succeed when they create a 2048 cell. And they fail when there is no more possible move in the grid.

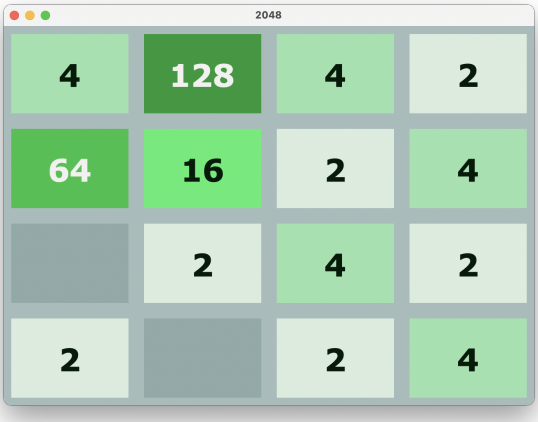

Fig. 1: 2048 Game Interface

The game has a 4×4 grid, so we represent the state by a list of size 16. Each case contains a value of the cell in the grid.

As a starting point, we will use MikhailLenko's code [3] that implement a playable version of 2048 and a simple AI,

[1] https://github.com/Lixiyao-meow/Reinforcement-Learning-for-2048.git

which we will modify and improve. We had also programmed a playable version for the deep learning model.

We choose a very intuitive scoring method. The game starts with score of 0. Every time two cells merge, the score increase by the value of the new cell. For example, if two cells of 4 merge, the score increase by 8. The score of a game is its final score. The reward of each step is defined by the increased value between the last step and the current step. We do not use the sum of all cells because the score always increases by 2 at each step.

2048 can essentially be described as an information-symmetric two-player game model. The player moves in one of four directions, and the computer fills in a cell with 2. Here *information symmetry* means that both players have the same information about the game at any given moment, and the move strategy relies only on reasoning about the next step. The main challenge of this game comes from its random component. We cannot correctly predict where the new tile will be placed. So it is impossible to have an algorithm solve the game correctly every time.

IV. The Agents

A. Policy Network with Reinforcement Learning

Firstly, inspired by Jonathan Amar and Antoine Dedi's article [4], we chose a Policy Neural Network taking the state as input and sending action as output. We realise the game playing with the front-propagation and learning process with the back-propagation by taking into account the reward of each action.

Our Policy Neural Network has three layers. The two hidden layers are of size 200 and 100. Therefore our Neural Network is encoded over three weight matrices W1, W2 and W3. The output of the current state is a vector of size 4, representing four possible moves: up, down, left, right. The architecture of our neural network is in the following.

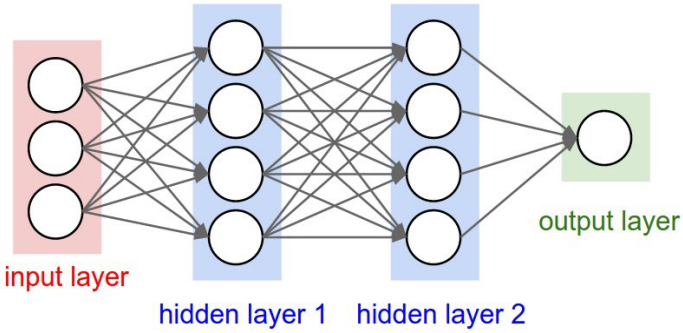

Fig. 2: Neural Network Structure

1) ***Front-propagation with Policy Network****:* We aim to find the most appropriate policy π^* which maximises the reward obtained in the following steps. The neural network takes the game's state as input. And the output of the current state $\mathbf{p}$ is a vector of size 4, representing the probability of the four possible actions. We apply a *RELU* activation function after the two hidden layers and a *Softmax* transformation before the output to get the probability of a multiple-label problem. The forward propagation algorithm is as follows.

1) Define the first hidden layer $h_1 = \max(\mathbf{W_1^T x}, \mathbf{0})$
2) Define the first hidden layer $h_2 = \max(\mathbf{W_2^T x}, \mathbf{0})$
3) $p = S(\mathbf{W_3^T x})$ with $S(h_i) := \frac{\exp \mathbf{h_i}}{\sum_k \exp \mathbf{h_k}} \in [0, 1]$

Then the action is made by a probabilistic approach. We choose a move randomly according to its corresponding probability in $\mathbf{p}$. At each iteration, we initialise the game and play N_{batch} steps. We store all states, actions made, outputs and vectors in hidden layers to update the weights used in the back-propagation algorithm. It is essential to notice that we do not save the probability of unfeasible actions. Since the algorithm cannot know that the action is impossible, it will still encourage the action chosen.

2) ***Back-propagation****:* The most critical question in the learning phase is how to define the reward. Here we use the score obtained in one step as a reward, so the agent will focus on the short-term reward in its learning process. Then we update the weights in the neural network according to the back-propagation algorithm by a gradient descent method.

More precisely, we associate a step score r, which is the score get during one step, to the action a. If the reward is positive, we can encourage the action. And for negative rewards, we do the opposite. Consequently, the back-propagation algorithm is as follow.

1) Define the reward vector $\mathbf{V} = \mathbf{R} * (\mathbf{A} - \mathbf{P})$
2) Define $d\mathbf{W_3} = \mathbf{V^T H_2}$
3) Define $d\mathbf{h_2} = \mathbf{V W_3}$ and $d\mathbf{W_2} = d\mathbf{h_2}^T \mathbf{H_1}$
4) Define $d\mathbf{h_1} = d\mathbf{h_2 W_2}$ and $d\mathbf{W_1} = d\mathbf{h_1}^T \mathbf{X}$

where $\mathbf{X}$ the states, $\mathbf{R}$ the rewards, $\mathbf{P}$ the probabilities and $\mathbf{A}$ the actions taken in N_{batch} games.

3) ***Performance****:* After a long learning process, when we play 2048 with this agent, the largest Maximum Value obtained at the end game is 512, and the most often Maximum Value obtained is 128. This performance is equal to the one of a random play. Then we also try other reward methods. We used the current score of each round to maximise the length of one game. We also try the maximum value achieved by the game to encourage the agent to focus on the long-term reward. However, after long training, the performance does not show a recognisable improvement. Hence we move to another method that we present below.

B. Pure Monte Carlo Game Search

Due to the unsatisfying performance of the neural network, we try another method, the Pure Monte Carlo Game Search (PMCGS). We start with a 2048 board, and the objective is to find the best possible move. The idea is to obtain a score for each possible action (left, up, right, down) and then choose the action with the best score.

The Monte Carlo term indicates that the score of a move is computed by playing multiple steps after the first move

and then taking the average of the score of each step. This averaging allows limiting the impact of the randomness of the game.

At the end of PMCGS, we end up with the move that, on average, gives the best game score with the chosen method of playing. So in PMCGS, we need at first some already defined methods to play the game. In the following sections, we will present different approach to play the game.

*1) **Random Approach**:* The idea is to play the game randomly. We choose a random possible move for each state. Therefore, this straightforward implementation allows playing lots of rounds at high speed.

It will give a lot of bad results and some good results. The interest of the method is that it does not suppose anything, which allows obtaining unexpected long term rewards in some rare games.

The biggest con of this method is that we need to receive lots of bad results to obtain one good one. The question is then: do the benefits of the low cost of this method outweigh the price of computation of a more complex approach?

*2) **Greedy Approach**:* In this approach, we play most greedily by choosing the move that will give the best immediate cost. This method gives much fewer bad results than the random ones. However, it is a short-sighted algorithm and does not involve long-term rewards.

*3) **PMCGS + Random Approach**:* Here is an improved version of the PMCGS approach mentioned above. For the N first moves ($N \approx 4$), we choose the move that gives the best score for another mini-PMCGS: the score is an average of the sum of N random next move's score.

Therefore, we use N low-cost PMCGS to compute the first N moves. Then we continue with a random game. This combination gives a better beginning but at a much higher cost.

*4) **UCS + Random Approach**:* Instead of using a PMCGS to compute the first N moves, we can also use the Uniform Cost Search (UCS) algorithm to find the best N moves.

Finding the N absolute best moves is impossible because the number 2 pops up randomly on the board. However, we change the main idea here. We just computed the N best moves by assuming the 2 pops up in the same cell. Then since we play multiple games, we can consider different ways the 2 pops up.

Then we try to implement the UCS in our scenario. Usually, UCS is an algorithm that allows finding the most probable path in a probability tree. However, we do not have a probability tree but a scoring tree here (each move gives a score). In particular, the final score is the sum of the scores of the edges and not the product of the probability of the edges.

We found a way to apply UCS to our score tree but first lets recall what the UCS algorithm does with probability tree: if we note $(p_i)_{1\leq i\leq N}$ the series of the probability of a branch of length N in the tree then the UCS algorithm find the branch $(p_i)_{1\leq i\leq N}$ such that $\prod p_i$ is maximum.

In fact we noticed that the UCS algorithm, we can be applied to every problem written as such:

$$\text{Maximise} f_N((x_i)_{1\leq i\leq N})$$
with (f_n) a family of function such that:
$$\forall n, \forall (x_i)_{1\leq i\leq N}, \forall X, f_{n+1}(((x_i)_{1\leq i\leq n}, X)) \leq f_n((x_i)_{1\leq i\leq n})$$

In the case of the probability tree, (f_n) is the product function. The property is respected because $0 \leq x_i \leq 1$.

To apply UCS to our score tree, we therefore need a family of function (f_n) that respects this property. However, this family of function also needs to respect this other property:

$$\forall n, \forall (x_i)_{1\leq i\leq n}, \forall (y_i)_{1\leq i\leq n} :$$
$$f_n((x_i)_{1\leq i\leq n}) \leq f_n((y_i)_{1\leq i\leq n}) \implies \sum x_i \leq \sum y_i$$

Indeed, we need to be sure that finding a maximum for f_n means finding a maximum for the the sum of the scores.

We find that

$$\forall n, f_n((x_i)_{1\leq i\leq n}) = \prod \exp(\frac{x_i + 2048}{4096} - 1)$$

respects both properties. So we apply this function to the UCS algorithm and find the best N first moves.

We apply this function to the UCS algorithm and find the best N first moves.

V. Results and Discussion

A. Policy Neural Network

Unfortunately, the agent using Policy Neural Network dose not get a good performance. The largest Maximum Value obtained at the end game is 512, and the most often Maximum Value obtained is 128.

Even if we try other reward methods, the result remains unsatisfying. We try the current score of each round as reward to maximise the length of one game. We also try the maximum value achieved by the game to encourage the agent to focus on the long-term reward. However, after several hours of training on a PC, the performance does not show a recognisable improvement. The main issues we see are the following.

It is challenging to adjust the weight in hidden layers, the learning rate and the reward score. When the weight W1, W2, W3 or the learning rate is too big, the algorithm converges too quickly, and the agent makes decisions with quasi-certainty. If we reduce the learning rate, the learning phase seems too slow.

While running the Policy Network algorithm, we notice that starting from a random policy network results in a huge cost by increasing the learning time. It explores too much before understanding the strategies of the game. One solution would

be to implement simple methods to help the agent play better at the beginning, i.e. repeat the left-up action for the first 20 moves.

Overall, we find that the neural network is not the best Reinforcement Learning method to solve 2048. There are faster and more appropriate solutions to the 2048 problem.

B. *Pure Monte Carlo Game Search*

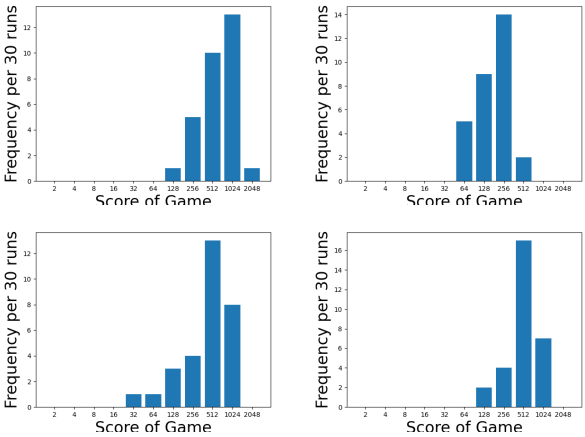

Fig. 3: Maximum Value reached using 0.1s for computing one move, from left to right: random games, greedy games, PMCGS+random games, UCS+random games

Firstly, we test these four methods (random, greedy, tree-search, UCS) on 30 rounds by scoring them with the maximum value obtained or 2048 if the agent wins the game. To balance the computing cost of each method, we give the algorithm only 0.1 seconds to decide the best move. We show the results in the figure above. (Remark: a game takes approximately 1000 steps, so roughly 1 min 40 with our settings. Since we test our model on 30 games, we can not afford to give the algorithm too much time to think.)

The performance of these four strategies can be resumed by the table below.

Strategy	Maximum Value	Most Often Value
Random	2048	1024
Greedy	512	256
PMCGS + Random	1024	512
UCS + Random	1024	512

TABLE I: Performance with 0.1s computation time for each step

First, we observe that only the purely random method win the game. It is also the method with the best average score of 729,6 and an average log2score of 9.37. The greedy algorithm is the worst that does not even reach 1024. The PMCGS is worse than the random one and can have a very poor performance like 32.

The UCS method draws our attention as it is the most stable one: it reaches 512 half the time. However, it does not win the game and has an average score of 571,7 and an average log2score of 8.9, which is lower than the random one.

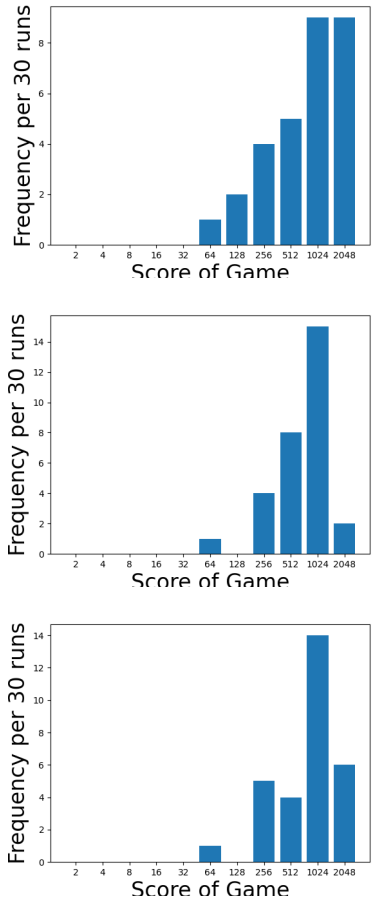

Fig. 4: Maximum Value reached using 0.3s for computing one move, from left to right: random games, PMCGS+random games, UCS+random games

Then we increase the computing power. We observe that these three methods have a good perforation and all of them win the game. The PMCGS and the UCS algorithm gain more than the random one. UCS and PMCGS are relatively equivalent because of their regularity. But UCS is a little bit better with an average score of 1000.5 than PMCGS with an average score of 821.2. The performance of these four strategies can be resumed by the table below.

Strategy	Maximum Value	Most Often Value
Random	2048	2048 / 1024
PMCGS + Random	2048	1024
UCS + Random	2048	1024

TABLE II: Performance with 0.3s computation time for each step

Now we compare the UCS and the random method. It is now much harder to pick the best move through the random one while it dominates on the 0.1-second test. Although the

random method has the highest win rate, UCS give us more chance to reach at least 1024. If we compare their average score, the random method wins with 1051.7 against UCS with 1000.5. But UCS has a better log2 average (9.6 against 9.5).

After comparing these two tests, we can say that the purely random method is good for its low-cost computational aspect. But when we leave more time for each step, the UCS algorithm becomes more relatively efficient.

VI. Conclusion

Due to the complexity of possible states, our first model spends too much time before understanding the game's strategies. We observe that the Neural Network is high-cost on computing time and is not suitable for solving the 2048 problem.

Then we try another method, the Pure Monte Carlo Game Search combined with different game playing strategies. We implemented four strategies: Random Approach, Greedy Approach, PMCGS with Random Approach and UCS with Random Approach. We test them at the same computational power to compare their performance.

We conclude that Random Approach is the best way to solve 2048 if we do not have many resources on the computational aspect. But when we leave more time for each step, the UCS algorithm becomes more relatively efficient. We get the 2048 cell from three strategies, so the Pure Monte Carlo Game Search can be summarised as successful.

References

[1] What is the optimal algorithm for the game 2048? https://stackoverflow.com/questions/22342854/what-is-the-optimal-algorithm-for-the-game-2048

[2] Navjinder Virdee. 2048 deep reinforcement learning. https://github.com/navjindervirdee/2048-deep-reinforcement-learning

[3] Mikhai lLenko's code for 2048 base game and first AI. https://github.com/kiteco/python-youtube-code/tree/master/AI-plays-2048

[4] Jonathan Amar and Antoine Dedieu. Deep Reinforcement Learning for 2048. *Operations Research Center, Massachusetts Insitute of Technology*, 2017.