Reinforcement Learning Project - 2048 Game

Anonymous Authors

***Abstract*—2048 is a single-player sliding block puzzle game released on March 2014. The game's objective is to slide numbered tiles on a grid to combine them to create a tile with the number 2048. However, one can continue to play the game after reaching the goal, creating tiles with larger numbers. The purpose of this paper is to provide a self-learning agent capable of playing the game and reaching at least the 2048 goal through Reinforcement Learning. In particular, we will look into Deep Q-Networks (DQN) and its improvements which seems to be the most efficient way for solving this problem.**

I. Introduction

Since its release, 2048 has been a playground for AI researchers due to its simplicity and ease to test Reinforcement Learning (RL) agents. Even though many studies have been carried for this game, they all differ by the choice of the RL model as well as the reward function used for training the agent. After a literature review, we focused on DQN agents and studied the influence of reward function choices on the agent performance. Our code is available online[1] and runs on Python 3.

Game 2048 is played on a 4x4 grid. The objective of the original Game 2048 is to reach a 2048 tile by moving and merging the tiles on the board according to the rules below. In an initial state, two tiles are placed randomly with probability 0.9 or 4 with probability 0.1. The player selects a direction (up, right, down, or left), and then all the tiles will move in the selected direction. When two tiles of the same number collide, they merge and create a tile with the sum value. After each move, a new tile appears randomly at an empty cell with number 2 (probability 0.9) or 4 (probability 0.1). The game ends when the grid is full and no more movement is possible.

The game was so popular, that players were looking for performance and thus achieved higher scores and get bigger tiles beyond 2048, such as 4096, 8192... on their grid.

II. Background and Related Work

Research in RL has been very active and finding the right model for a specific environment can be a difficult task. Literature is full of agents trained to solve 2048, all based on DQN which proved to be the most efficient choice [1], and using various custom reward functions. Besides, most models encountered are trained over a large number of games (approx. 20000 games). Even though they yield good results, we wanted to focus on a much smaller training time (3000 games), and investigated the best learning model within this time constraint. Finally, it is worth notice that tree search algorithms - that is without any training time - seem to work well for solving this game. For instance, a simple Monte Carlo tree search with 100 runs per move (approx. 500 milliseconds) can achieve a decent 80% success rate for the 2048 tile (50% for 4096), and up to 100% with 10000 runs per move [2]. However, it does virtually never reach the 8192 tile, which can be reached by deep networks sufficiently trained.

III. The Environment

As of most RL models tested on games, we made use of the OpenAI Gym interface which provides good standardized game environments. In particular, we trained our model on a customized version of the open-source gym-2048 environment which is based on the following:

- State space: All observations are 4 x 4 numpy arrays representing the grid. The array is 0 for empty locations and numbered 2, 4, 8, ... wherever the tiles are placed.
- Action space: There are four actions defined by integers (LEFT = 0, UP = 1, RIGHT = 2, DOWN = 3)
- By default, reward is defined as the total score obtained by merging any potential tiles for a given action. Score obtained by merging two tiles is simply the sum of values of those two tiles. However, we created other custom rewards, presented below, by overriding the methods of the original environment in an inherited class.

Hence, it is a fairly simple environment (stochastic and fully observed) which is well suited for a DQN agent. The main interest is to design the network architecture with a reward function that achieve best results.

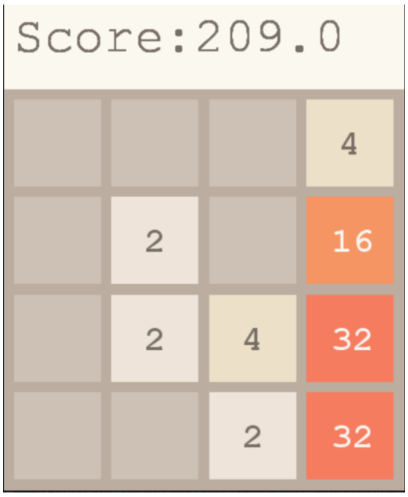

Fig. 1: Example of fancy environment rendering.

Part of the optimization of the algorithm was the choice of the reward function. We were then able to test the performance of the agent with 3 different reward functions

[1] Our Code: https://github.com/achrafvit1213/2048-master.

based on the same number of training episodes: 3000. By analyzing the score curves and the distribution of the max tiles over 3000 sessions, we could effectively observe the impact of the reward on the performance and choose the function that allowed the agent to be the most efficient.

- **Reward 1: Score**
  This function is the basic reward function given by the environment. the total score obtained by merging any potential tiles for a given action. Score obtained by merging two tiles is simply the sum of values of those two tiles. This reward encourages the merge of high value tiles.

- **Reward 2: Total merges and maximum tile**
  In order to encourage the creation of high value tiles and the merge of tiles that are not necessarily of high value, we dissociated the two objectives in this reward function. It is defined, for a given state transition, as the number of merges added to the base 2 logarithm of the maximum value tile. This metric encourages not only the merging of two tiles but also longer term strategies to create new powers of two, which is also close to the goal of this game.

- **Reward 3: Number of empty tiles**
  This reward function is very different from the first two and describes another way of looking at the game. It is based on the principle that the more empty squares there are in the grid, the more one can advance in the game. Indeed, the game ends when the grid is full and no more movement is possible.

IV. The Agent

The agent trained for this game is based on a Deep Q-Network with convolutional layers. It is particularly suitable to use a deep reinforcement learning procedure because of the size of the state space. We started from a vanilla DQN with replay memory (technique used to produce uncorrelated batches of input data) and first added convolutional layers, as illustrated by the figure.

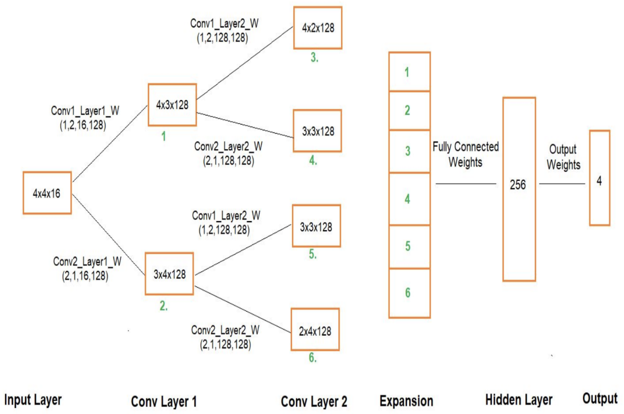

Fig. 2: Architecture of the Deep Q-Network [4]

As a reminder, DQN with MSE loss works by minimizing the following loss function [3]:

$$L_i = \mathbf{E}_{s,a,s',r}(r + \gamma.max_{a'}Q(s',a') - Q(s,a))^2$$

which consists in computing the one-step ahead target Q value and confronting it to the actual one. We believe that this model is the best approach as it uses 2D grid features through convolutions for understanding the grid and building the target policy, and it handles well the exploration-exploitation trade-off for this particular game where exploration is essential. In fact, the main point is that 2048 suffers from local comfort zones. Thus, sometimes the agent needs to take negative actions to move since the direction that it wanted first is not valid. For that matter, methods based on policy gradients would understandably not perform as well because they tend to focus on localizing behavior. Besides, it is worth notice that we used an improvement of the vanilla DQN known as Double DQN. It consists in keeping tow copies of the network, one used for greedy selecting the action (Online Network) and the other for estimating the Q value (Target Network). It is known to improve performances by reducing the bias over Q values estimations.

For the policy, we use an ϵ-greedy approach with a decaying ϵ. We choose low values of ϵ because, in the context of this game, random moves can have a huge effect on the outcome of the game.

Below are the main parameters used in the training of this DQN (with an ADAM optimizer).

TABLE I: Values of the main parameters of the DQN

Parameter	Value
Learning rate	10^{-4}
γ	0.99
Initial ϵ	0.1
Final ϵ	10^{-4}
Exploration step	10^{-4}
Replay Memory storage	Last 50 000 moves
Batch length	16

Then, we tried another well-known improvement of DQN architectures which is Dueling Double DQN[5]. In the latter, Q is computed as follows :

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a=1}^{|\mathcal{A}|} A(s,a)$$

Instead of learning Q, we use two separate heads to compute V and A. It is known to improve performances because it updates value function V which is general over all actions, and thus training iterations can have a larger impact. However, after trying it with our model (reward 2), we've seen that it does not perform as well as the simple Double DQN (it probably needs more than 3000 epochs to train correctly).

V. Results and Discussion

In this section, we present and discuss the results on the training and testing of our agents, and compare them for the 3 reward functions defined in section 2. We start with the Deep Q-Network.

Training

We obtain the following graphs for the evolution of the reward during training, for each of the 3 rewards. The graphs represent moving averages of these rewards, with a moving window of 25. This averaging compensates for the obvious statistical fluctuations of the reward from game to game.

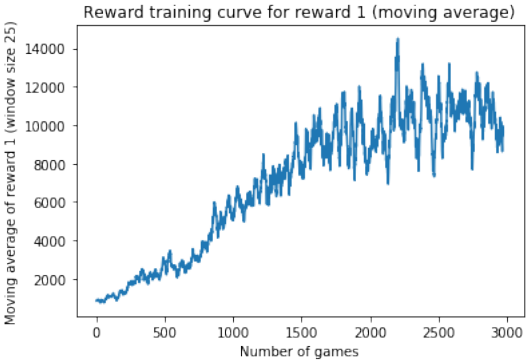

Fig. 3: Averaged reward 1 w.r.t the number of games.

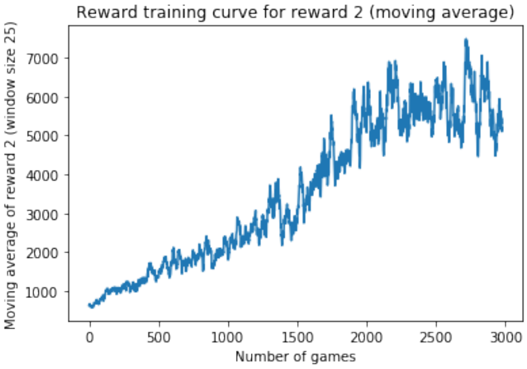

Fig. 4: Averaged reward 2 w.r.t the number of games.

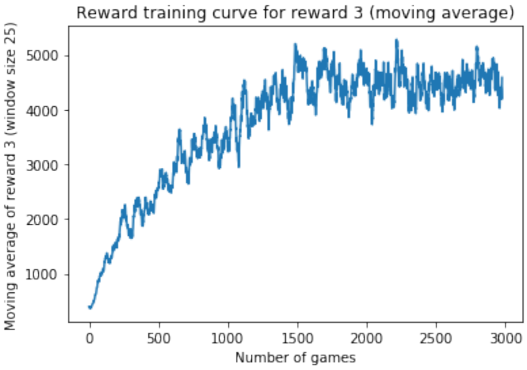

Fig. 5: Averaged reward 3 w.r.t the number of games.

Due to the different scoring methods for each reward, these results cannot be compared in terms of reward value. However, we can note that the reward 3 increases very quickly at the beginning of the training, and reaches a constant value with less fluctuation than the other rewards. The other rewards train less quickly.

In order to properly compare the 3 agents, we can observe the maximum tiles reached at each game during the training. Once again, we smooth the curves using a moving average strategy with a window of 25.

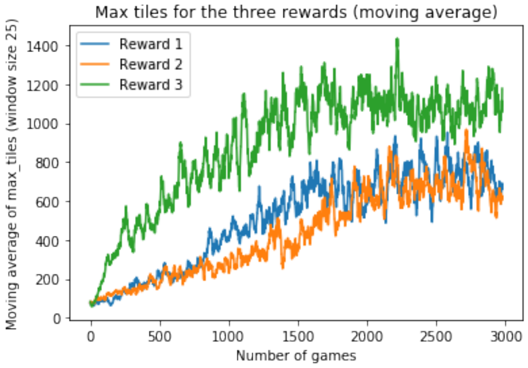

Fig. 6: Averaged maximum tiles reached for each reward.

It appears clearly that reward 3 performs much better than the other rewards, from the beginning of the training to the end. At the middle of the training, the reward 1 outperforms the reward 2, then the latter ends up having more or less similar average maximum tiles for the 1000 last games.

Testing

After having trained our agents, we test their performance by playing 1000 games, and recording the number of occurrences of each maximum tile. This is our main result.

TABLE II: Occurrences of each maximum tile for the 3 rewards (1000 runs)

Max tile	Reward 1	Reward 2	Reward 3
32	0	0	1
64	3	9	2
128	29	33	10
256	130	144	39
512	339	409	194
1024	454	381	525
2048	45	24	227
4096	0	0	2

It seems clear that the agent with reward 3 largely outperforms the others, with a rate of reached 2048 approximately equal to 23% (v.s 2 and 4%). It also reaches 1024 more frequently. It is interesting to note that agent 3 manages to reach 4096 twice, for 1000 games. Additionally, agent 1 has an overall better performance than agent 2 with twice as many 2048s reached and more 1024s.

Dueling Double DQN

This method does not yield great results, probably due to the fact that the number of epochs (3000) is not high enough to properly train the networks.

Discussion

Based on a more precise assessment of the performance of the agents, through a precise analysis of their game-plays and relationship to reward (by observing games), we can provide a discussion on our results.

From the previous results, we clearly conclude that the reward 3 is the best one in terms of performance. It seems that basing the reward on the number of empty tiles enables to get better results: the agent tries to empty the grid as much as possible which enables to last longer in the game. It is worthy of interest that simply counting the number of empty tiles instead of taking into account the contents of the tiles gives better results. Actually, by observing the other agents playing several games, we note that they tend to not choose the action of merging low value tiles which results in a full grid. Therefore, this explains why the reward 3 could have a better performance: the agent sometimes chooses actions that do not improve the score (i.e reward 1) immediately, but enable to get more space for significant merges.

Moreover, agent 1 has a better performance than agent 2 probably because, due to the definition of reward 2, agent 2 tends to ignore merges of big value tiles when the resulting tile is already present in the grid.

Finally, a weak point of every one of these agents is that they tend to not conserve strategical positions: for instance, a classical strategy for this game is to place high value tiles on the corners, which does not happen during the games played by the agents.

VI. Conclusion and Future Work

To conclude, we were able to create a relatively powerful agent, since it reaches the goal of the game on average in 25% of the games and does less well than 512 for only 5% of the games.

We trained this agent using a double deep Q learning network and several types of reward function during 5 hours (3000 sessions) for each function.

The one that gets the best results corresponds to an original reward function that we developed and that is not found in the literature: the more empty cells in the grid, the more the agent is rewarded.

There are mainly 3 axes of improvement.

- Perform simulations by testing more DQN hyper-parameters. Find the best tuning.
- Improve the reward function and combine the one to be created, in particular by rewarding the creation of empty cells as the max tile is bigger and bigger, in short by using a characteristic of the state of the game (the max tile) in the calculation of the reward.
- To train the agent on more game, we were unfortunately limited by the computing power of our computers and time.

VII. Appendix

Here are the graphs of maximum tiles obtained during training for each reward.

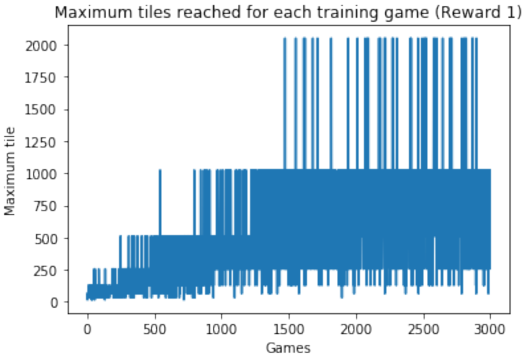

Fig. 7: Maximum tiles for reward 1.

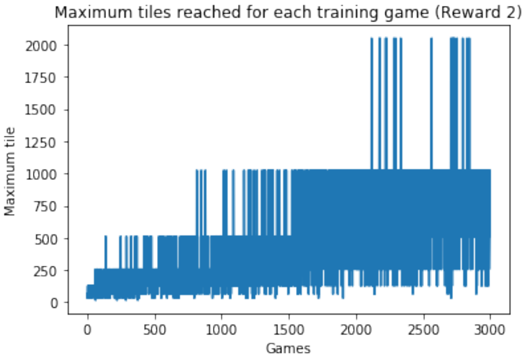

Fig. 8: Maximum tiles for reward 2.

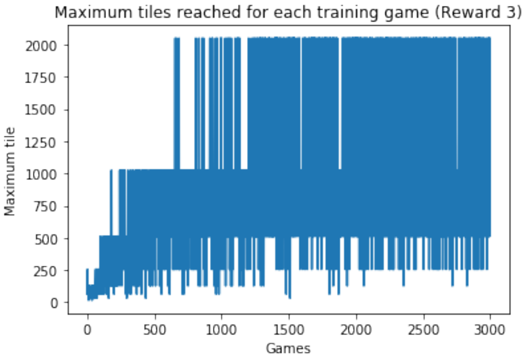

Fig. 9: Maximum tiles for reward 3.

REFERENCES

[1] O. Vinyals et al. StarCraft II: A New Challenge for Reinforcement Learning. https://github.com/daviddwlee84/ReinforcementLearning2048, 2017.

[2] O. Vinyals et al. StarCraft II: A New Challenge for Reinforcement Learning. https://github.com/ronzil/2048-AI, 2017.

[3] Jonathan Hui - RL — DQN Deep Q-network https://medium.com/@jonathan_hui/rl-dqn-deep-q-network-e207751f7ae4 , 2018.

[4] https://github.com/navjindervirdee/2048-deep-reinforcement-learning/blob/master/Architecture/Architecture.JPG, 2018.

[5] Gouxiangchen - Dueling-DQN-pytorch https://github.com/gouxiangchen/dueling-DQN-pytorch/blob/master/dqn.py, 2019.