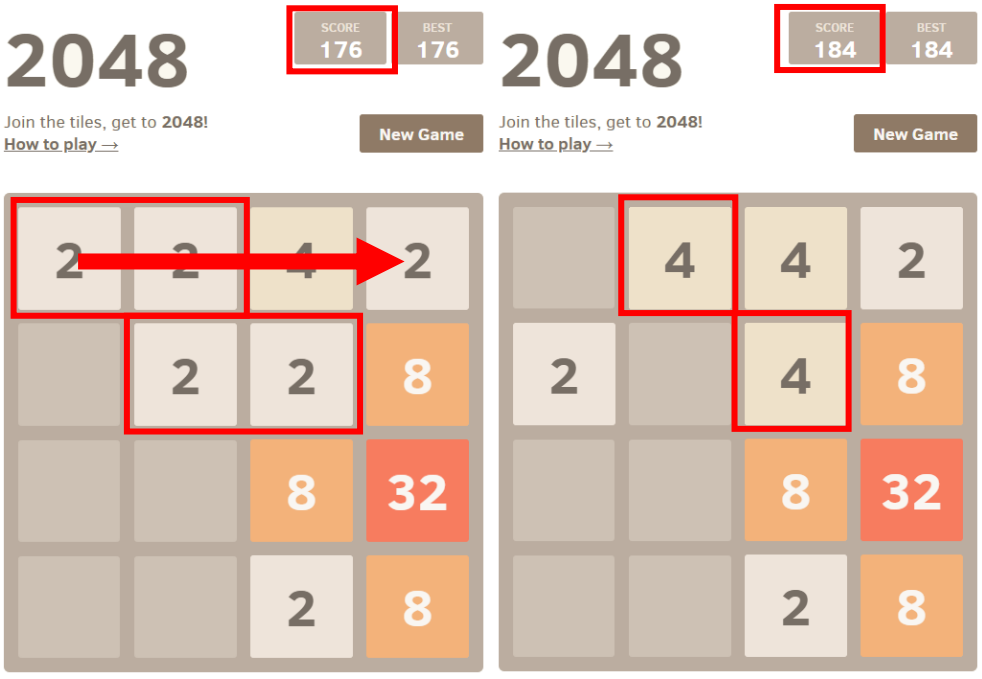
Reinforcement Learning – Project 4 – 2048 game

Celso T. do Cabo, Rushabh Kheni and Venkatesh R. Mullur

PROJECT REPORT

INTRODUCTION

2048 is a nondeterministic puzzle game which the objective is based on adding same numbers that are multiple of two until reaching 2048. For that, four actions are possible in the game, the numbers can move either, left, right, up or down, equal numbers will be added, and for each action either a 2 or 4 will appear at a random location. The game will be lost if there are no possible action, and the score will be counted as the number generated at each action.



(a) (b)

Figure 1: 2048 game first state (a) and after action to right (b)

In Fig. 1 it is shown an initial state of the game (a) and after the action of moving to the right, it is shown (b) that the equal numbers were added – two locations where number twos were next to each other – and finally the score is updated adding 8 to the previous value since two number four were generated. In addition, it is possible to see that a new number two is added on the second line in the first column for the next round.

METHODOLOGY

In order to apply a reinforcement learning algorithm in the project, it was needed to better understand how the game works in the perspective of the class. For that, in this section the game will be described as a reinforcement learning problem, whereas how the environment can be represented, how to describe the possible actions in an algorithm and how the reward can be calculated, then, based on the state space of the problem, it is proposed two algorithms to achieve a high score in the game. Firstly, it is proposed to use a simple Deep Q-learning algorithm, and later, due to the performance of the algorithm, it is proposed to apply the Double Deep Q-learning method.

Problem description

The game can be described based on a few key features that are needed for the application of a reinforcement learning algorithm, among those features, it is crucial to understand how our environment works, the size of our state space in order to verify if it is possible to apply a deterministic solver, in cases where the state space is smaller enough to be computed. How the reward will be computed, and the possible action space are also important to understand the problem. Following below, are how each one of those features were defined in the project:

1) Environment: - We have used gym-2048 python package [1] for our game environment which renders the game in the grid format.
2) State Space: - The total number of possible states is 2^{48}. There are 16 tiles in the grid and each tile can hold a value from "none", 2, 4, 8 and so on until 2048 i.e., each tile can hold 12 different values. Hence the total number of states are $16^{12} = 2^{48}$ states. This is only if we consider the maximum tile value to be 2048, but in reality, the maximum value of the tile can reach up to 131,072 which makes the state space exponentially larger.
3) State Representation: - The input to our model is an array of length 16. We have taken the 4x4 grid and flattened it out to make an array of length 16 which is the input to our model.
4) Action space: - There are four actions defined by integers.
    - Left: 0
    - Up: 1
    - Right: 2
    - Down: 3
5) Rewards: - Reward is the total score obtained by merging any potential tiles for a given action. The score obtained by merging two tiles is simply the sum of values of those two tiles.

Deep Q Learning

Deep Q-learning (DQN) is a reinforcement technique applied to problems where the state space is too large to be easily represented, which was initially introduced by [2], differently from the traditional Q-learning algorithm, where a matrix with possible states and actions are compiled and the Q-value is later calculated based on the updates from the actions taken. In addition, other works

where DQN was applied into the 2048 game were done [3]. The DQN algorithm will calculate the Q matrix based on a neural network, whereas the state will be used as input for the system and the neural network will calculate the Q-values for each possible action. The neural network structure used for this project can be seen in Fig. 2 below.

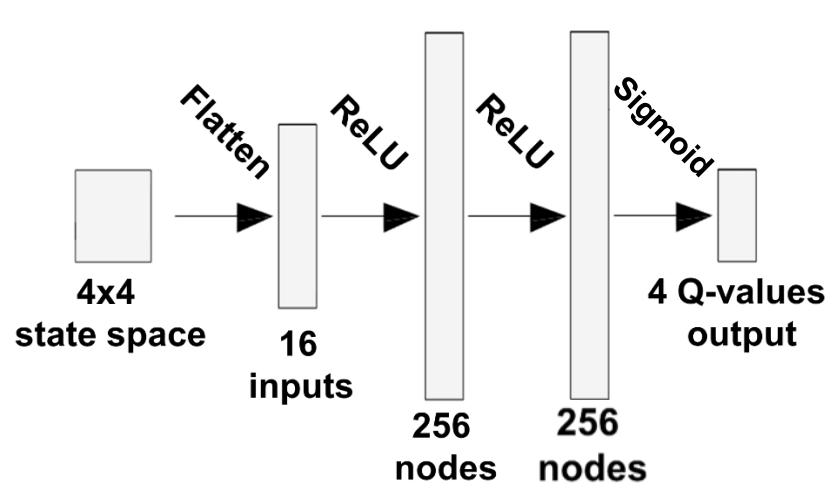


Figure 2: Neural network structure for DQN

For this project, since the input for the neural network is a 4x4 matrix, initially the input is flattened to convert into a vector with the 16 inputs for the neural network, the activation functions used between the layers were the ReLU and the neural network had three hidden layers with 256 nodes each, only the final activation function that was used a Sigmoid. Finally, the output is the Q-values for each of the four actions. Based on the rewards obtained from the actions, the weights of the network will be updated. How the Q-value is updated can be seen in Equation (1).

$$Q_t(a) = \begin{cases} R_t + \gamma \times \text{argmax}(Q_{t+1}) \\ R_t, \ \ if\ end\ of\ episode \end{cases} \tag{1}$$

Where, $Q_t(a)$ is the updated Q-value for the action a, R_t is the reward of the episode and γ is the factor used to discount the importance of the next step in the calculation of the Q-value of the current state. Initially, to explore different states and actions, an ε-greedy policy is applied to the algorithm, which is described in Equation (2).

$$Action(t) = \begin{cases} \max\big(Q_t(a)\big), with\ probability\ 1-\epsilon \\ random\ action, with\ probability\ \epsilon \end{cases} \tag{2}$$

The hyperparameters used in the training loop are described in Table 1. The ε-greedy policy was applied from 1.0 where all actions are random to 0.025 where only 2.5% of the actions are random

during the 20,000 first episodes of the game. After those episodes, the actions followed a 2.5% rate of random actions until convergence.

Table 1: Hyperparameters used to train the DQL

Hyperparameter	Value
Batch size	32
Learning rate	5e-5
Gamma (γ)	0.99
Epsilon (ε)	1.0 – 0.025
Number of episodes	50,000+

In addition to the hyperparameters, the loss function was also changed, while [2] recommends the use of the L_1 smooth loss function for different game environments, it was noted that the mean square error (MSE) loss function had a better performance while training the network, which is described in Equation (3).

$$\ell(x, y) = \frac{1}{N}\sum_{i=1}^{N}(x_i - y_i)^2 \tag{3}$$

Where, x_i is the predicted Q-value from the neural network and y_i is the target value, obtained by Equation (1) and $\ell(x, y)$ the total loss.

Double Deep Q Learning

In addition to the traditional DQN algorithm, it was also implemented a Double Deep Q-learning (DDQN) to observe if the new model would obtain a better reward or converge faster than the original DQN. This variation of DQN introduced by [4], uses not only the online network to update the Q-values, but also the target network. If the Equation (1) is rewritten as in Equation (4).

$$Q_t(a) = R_t + \gamma \times \mathrm{Q}(\mathrm{S_{t+1}}, \mathrm{argmax}\big(Q(S_{t+1}, a; \theta_t)\big); \theta_t) \tag{4}$$

Showing that the $Q_t(a)$ is calculated based on a neural network (the online neural network) and it is a function of its weights θ_t, then, how the Q-value estimated can be rewritten based on both the online and target network, as shown in Equation (5).

$$Q_t(a) = R_t + \gamma \times \mathrm{Q}(\mathrm{S_{t+1}}, \mathrm{argmax}\big(Q(S_{t+1}, a; \theta_t)\big); \theta'_t) \tag{5}$$

Where, θ'_t are the weights from the target neural network. In other words, while the traditional DQN uses only one of the neural networks to upload its Q-values, the Double DQN will use both the target and online network while calculating the Q-values, accelerating the process of convergence and increasing the reward obtained after training. For the project, the Double DQN

was used with the same hyperparameters and conditions described in the previous section from the DQN, so that the methods could be compared.

RESULTS

After completing the model with both the DQN and double DQN, the training metrics can be seen in the figures below. Both methods were trained for at least 50,000 episodes, and as mentioned previously with an ε-greedy policy for the first 20,000 episodes, starting with totally random actions to a probability of 2.5% of random actions. In Fig. 3 it is possible to see the improvement of the reward over the number of episodes. To reduce the variation, the reward is counted based on the last 30 episodes. In the beginning of the training loop, 20 episodes are generated randomly so that there is an enough number of frames for the training batch. As it can be seen in Fig. 3, the reward stays approximately around 1,000, which is similar to the average of rewards obtained by random actions, however, after 20,000 episodes, it is possible to see that the reward starts increasing. However, after 65,000 iterations, the value starts to converge and the variance increases. Since the simulation was already running for around 24 hours it was decided to use this model as a metric for the DQN.

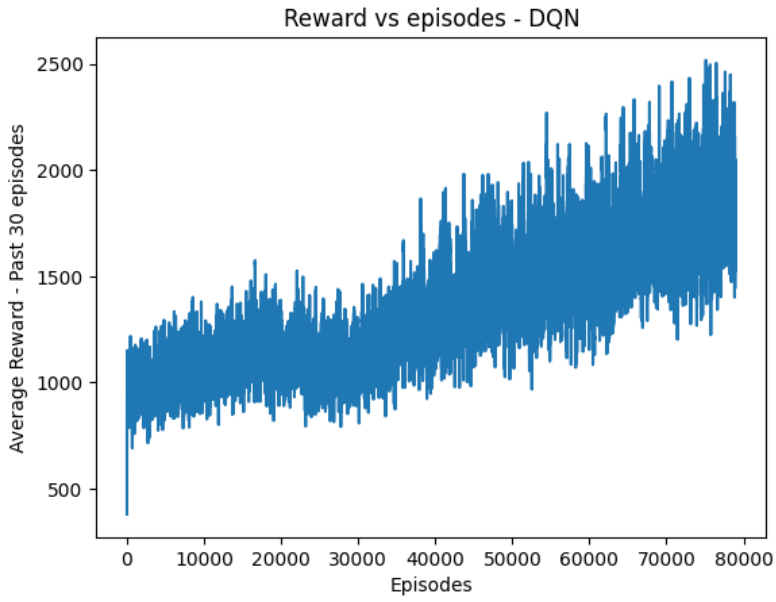


Figure 3: Training reward for the DQN model

After training the DQN model, the changes needed to generate the double DQN was done, the training reward versus the number of episodes for the double DQN can be seen in Fig. 4. Similarly to the previous model, since the ε-greedy policy is applied to the first 20,000 iterations, thus the reward is still low, however, it is possible to see that as soon as the number of random actions is reduced the reward starts improving. After 20,000 episodes, it has a sharp increase for some iterations, and then it starts increasing steadily. It is possible to see that 50,000 iterations its convergence starts to decrease, and it was decided to finish the training. In addition, in Fig. 4 it

was also plotted the maximum value obtained in each episode. It is possible to observe that in the beginning, the maximum value obtained is 256, however, after 20,000 iterations the number of 512 obtained starts to increase, and by the end of the training, the number of 512 is equivalent to the number of other values.

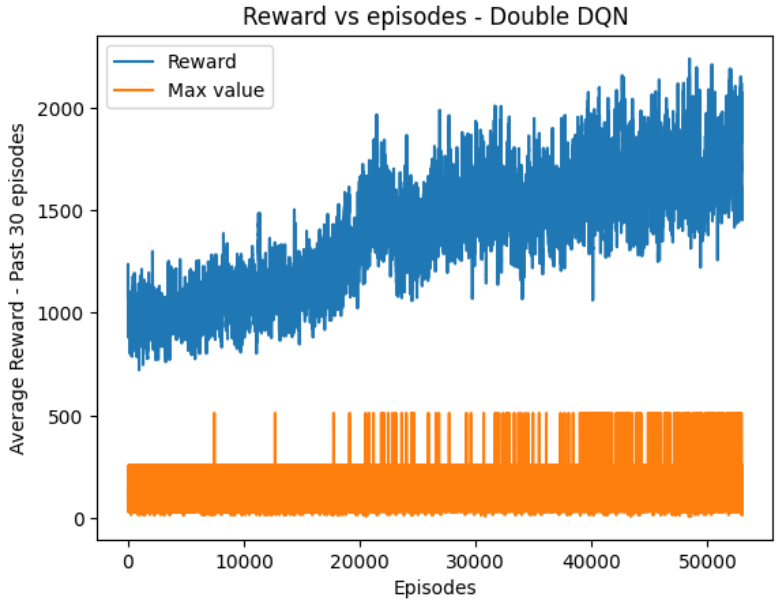


Figure 4: Training reward for the double DQN model

After training the model, a statistical run with 1,000 episodes were run for each model and only using random actions, the result can be seen in Table 2. It is possible to notice that although the Double DQN algorithm had a good improvement over time, since the time available to train the model was limited, it the final test did not outperform the traditional DQN. In addition, the DQN algorithm alone could not get any state values above 512, which also happened with the double DQN model.

Table 2: Probability of maximum value for DQN, double DQN and random actions.

State	Probability DQN	Double DQN	Random Action
512	5.7%	1.3%	>0.1%
256	44.5%	33.8%	4.2%
128	83.0%	79.2%	45.7%
64	97.2%	98.3%	91.5%
32	100%	100%	99.1%
16	100%	100%	100%

However, although those results were not as expected, it is possible to see an improvement compared to just random actions, both in the reward plots as in Table 2. While random actions have a reward of about 1,000, both models had scores around 2,000. In addition, it is possible to see that in the random action scenario, the chance of getting a 512 is practically zero, and even to achieve a 256 is much smaller than the models trained. The same results can be seen for the other values, whereas there are still some simulations that the maximum value obtained from the random action was only 16.

CONCLUSIONS AND FUTURE WORK

Although it was possible to obtain an increase of the reward compared to random actions, the final results obtained from both the traditional DQN and double DQN were below of the expected, the project was a good approach to understanding how different algorithms works and on how to tune them depending on the problem given. Some of the trainings showed a promising result, however, however, the time needed to the model convergence was higher than expected, turning them impractical.

For future work, different projects were done in reinforcement learning with the game 2048. Some of them used either time difference (TD) techniques allied with N-tuple networks [5-7]. Applying such methodology poses to be very effective, all papers shown that the results obtained were much higher than 2048, sometimes going to the maximum limit that the game supports.

All the codes used are also available at the GitHub repository from the Team 4 [8].

REFERENCES

[1] S. Kapoor. "gym-2048 0.2.6." https://pypi.org/project/gym-2048/ (accessed 11/08/2022.

[2] V. Mnih *et al.*, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602,* 2013.

[3] A. Dedieu and J. Amar, "Deep reinforcement learning for 2048," in *Conference on Neural Information Processing Systems (NIPS), 31st, Long Beach, CA, USA*, 2017.

[4] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, 2016, vol. 30, no. 1.

[5] N. Kondo and K. Matsuzaki, "Playing game 2048 with deep convolutional neural networks trained by supervised learning," *Journal of Information Processing,* vol. 27, pp. 340-347, 2019.

[6] A. Mehta, "Reinforcement Learning For Constraint Satisfaction Game Agents (15-Puzzle, Minesweeper, 2048, and Sudoku)," *arXiv preprint arXiv:2102.06019,* 2021.

[7] M. Szubert and W. Jaśkowski, "Temporal difference learning of n-tuple networks for the game 2048," in *2014 IEEE Conference on Computational Intelligence and Games*, 2014: IEEE, pp. 1-8.

[8] *Reinforcement Learning - Project 4, 2048 game*. (2022). [Online]. Available: https://github.com/CelsoTC/Project4_Team4.git