Deep Q-learning results

when playing 2048

# Context

For my reinforcement learning (RL) challenge, I decided to use Deep Q-learning to train a model how to play the game 2048.

*2048 is played on a plain 4×4 grid, with numbered tiles that slide when a player moves them using the four arrow keys. Every turn, a new tile randomly appears in an empty spot on the board with a value of either 2 or 4. Tiles slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles of the same number collide while moving, they will merge into a tile with the total value of the two tiles that collided. The objective of the game 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.*

* [Wikipedia](https://en.wikipedia.org/wiki/2048_(video_game))

# Setup

I based my code on Juan Gallostra Acín’s [python implementation](https://github.com/juangallostra/2048) of the 2048 game. I then modified the code so I can retrieve the data I needed, added score and remade the control logic so the game could be played by an AI, instead of a human. I also implemented a Deep Q-learning (DQN) model from the Keras library.

The hyperparameters I could tune were:

* The size of the network (number of layers and number of nodes per layer)
* The size of the action memory that model will use (haw many steps in the past does it remember)
* The learning rate of the network
* The number of train steps

The network had 16 input parameters corresponding to each game cell and 4 output parameters corresponding to each possible action (up, right, down, left).

# Results

The best results I achieved were with a network with four hidden layers, the first with 256 neurons, the next two with 64 neurons and the last one with 32 neurons. I used a memory limit of 10, thus remembering only the last 10 moves made and a learning rate of 0.05. The network was trained for 300000 steps and in this time, it managed to play 110 games with an average tile value per game of 187. The tile distribution that it achieved was:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tile value | 32 | 64 | 128 | 256 | 512 | 1024 |
| Percentage | 4% | 18% | 36% | 34% | 8% | 0% |

You can also take a look at the graph of the results from each episode bellow:

Chart, histogram

Description automatically generated

I also simulated 1000 runs taking random moves as a benchmark, and the results were the following:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Tile value | 32 | 64 | 128 | 256 | 512 | 1024 |
| Percentage | 7% | 37% | 49% | 7% | 0% | 0% |

There is a very clear improvement when using the DQN and honestly, I am very satisfied with the results, although we couldn’t achieve a 2048 tile. Furthermore, one of the other networks managed to achieve a 1024 tile once, which showed that the strategy the AI chose was decent.

Application, calendar

Description automatically generatedCalendar

Description automatically generatedAs we can see bellow, the said strategy was pretty simple – stack the high value tiles in one of the corners and surround them with higher and higher tiles the closer you get to them. A lot of people start to use that strategy when they initially start playing the game.

# Recommendations

If I had more time for this challenge, I would’ve implemented a few features that would most likely improve the performance of the network. These features are:

* Applying one-hot encoding to the values in the tiles and expanding the input layer so it has neurons for each tile (16) and each possible value in that tile up to 2048 (11 unique values), resulting in 176 input neurons.
* Adding 4 neurons in the input layer, indicating the possible moves the player can take.
* Changing the reward from using the score to using a factor of the maximum tile value