Applications of Computer Science - project in evolutionary algorithms

By:

Noga Mendler

Mor Hakmon

Table of Contents

introduction

2048 is a single player sliding- tile-puzzle video game designed by an Italian web developer Gabriele Cirulli which was published on GitHub. The Nineteen-year-old web developer created the game in a single weekend as a test to see if he could program a game from scratch. 2048 is played on a plain 4×4 grid, with numbered tiles that slide when a player moves them using one of four motions: up, down, right or left. The game begins with two tiles already in the grid, having a value of either 2 or 4, and another such tile appears in a random empty space after each turn. 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 with the same value collide while moving, they will merge into one tile with a total value of the previous two tiles. The resulting tile will not merge with another tile again in the same turn.

The user's score starts at zero point, and increases whenever two tiles combine, by the value of the new tile. In case the entire 4x4 grid becomes occupied with "numbered" tiles, and there are no more valid moves left to make, the game will end with a loss.

The objective of the game is to slide and merge identical numbered tiles, with the goal of reaching a tile with a value of 2048, to reach said goal the player has to use varied strategies such as keeping the largest tiles in a specific corner and to fill a specified row or column with the largest numbers in monotonic increasing order. 2048 gained widespread popularity shortly after its release and was described by the wall street journal as "almost like Candy Crush for math geeks".

Figure 1 - 2048 game board

evolutionary algorithm

An evolutionary algorithm is an AI-based computer application that solves problems by employing processes that mimic the behaviors of living things. The algorithm will start with a random input to the problem it tries to solve.

In each generation while using mechanisms that are typically associated with biological evolution, such as crossover (reproduction) and mutation it tries to optimize the solution.

The algorithm evaluates the solutions performance, selecting the best-fitting ones, and then generating a new population, with the goal being creating an algorithm which output the best fitted action.

The mathematical nature of 2048 has made the game of interest to AI researchers. In our project we will try to create an evolutionary algorithm that will be able to estimate the best next move based on the specific state of the board at any given time.

The problem

Since its release, many researchers have explored the different aspects of 2048. It's has been proven that the problem of solving the game (reaching tile with 2048) is a NP-hardness[1] and P-SPACE-hardness (for example by Reduction from 3SAT) which means that it is impossible to have an algorithm that will correctly choose the best possible move every single turn while taking polynomial time. The stochastic aspect of the problem is what makes it particularly interesting – there is no way to predict.

exactly whether the next tile will be 4 or 2 (90% probability of 2 and 10% probability of 4) and what spot it will land on. Different approaches were used in order to create an efficient algorithm that will in most likelihood choose the best possible.

move are including applying Monte-Carlo Tree Search[2] - heuristic search algorithm for some kinds of decision processes combining with neural networks or

[1] Threes!, Fives, 1024!, and 2048 are Hard by Stefan Langerman, Yushi Uno - https://doi.org/10.48550/arXiv.1505.04274

[2] Playing 2048 with AI - pt. 3 - https://p-mckenzie.github.io/2020/04/23/2048-part-3/

Expectimax search algorithm. In our attempt we will use Neuroevolotion- artificial intelligence that uses evolutionary algorithms to generate artificial neural networks. After each turn, we will give the ANN score – based on the tile number it created and whether its decision fits to different requirements we defined in advanced, as part of the process of trying to determine who are the best agents of each generation. We will use the evolutionary mechanisms mentioned above to create new and improved generation that will eventually generate us agents that consistently choose the best possible move.

The solution

In our project to create an algorithm for playing 2048 we used combinations of

evolutionary algorithms and neural networks. Neural networks, emulating the structure and operation of the human brain, comprise interconnected nodes termed neurons, arranged in layers. Each neuron receives inputs, processes them via a sum, multiplying and activation function, and yields an output. The inter-neuron connections possess adjustable weights that adapt during the learning phase, enabling the network to "discover" insights from data.

We chose to use neural networks due to their good ability to generalize complex functions and due to their high popularity in these days. We believed that using neural networks would allow us to create the optimal agent.

Neuroevolution is a method that combines neural networks and evolutionary algorithms to optimize neural network architectures and parameters.
First, a population of neural networks is created (In our game and code called agent). These agents start with predefined structures (3 layers).
Each neural network in the population is evaluated on a specific task or problem. This involve tasks like playing a game, or implementing semi – strategies defined by us, and shall be elaborated in the next topic. The performance of each network is measured using a fitness function that quantifies how well it performs the task. Networks with higher fitness scores are selected to "reproduce" and create children

for the next generation. They create children through techniques like mutation and crossover.

Child networks replace some of the less successful parent networks in the population. This ensures that the population continues to evolve over successive generations. The process is repeated for multiple generations, with each generation ideally leading to improvements in the overall performance of the population.

The code in the class "Main" initializes a population of predefined agents, each embodying a neural network and proceeds to execute the evolutionary algorithm over pre-defined generations. During each iteration, the fitness of every agent is assessed, and the most proficient agents, with the highest fitness score, are chosen to reproduce children via mutation, crossover. The rest of the new generation is produced via random initialization. This iterative process aims to gradually improve how well the neural networks play the game.

Neural networks acquire knowledge from data through a process known as training. Throughout trials of varying mutation, crossover, and the number of generations or agents, we try to determine the optimal settings.

The program

main.py:

main is a top-level hierarchy file that determines the parameters of the evolution algorithm. It creates the flow of the algorithm.

1. Defining the generation size and number of generations.
2. Calculates each generation’s fitness score.
3. It sorts the agents based on their fitness scores in descending order.
4. According to the fitness score:
    a. Operating mutation.
    b. Operating cross over.
    c. Generating new random agents instead of the fitted ones.

In addition, we added a segment of wanb so we can track the performance of the algorithm.

Agent.py:

Defines a simple neural network class called Agent along with two functions, mutate_layer and crossover_layer, for mutation and crossover operations on the neural network layers.

Additionally, a fitness function is defined which evaluates the performance of the neural network based on its average performance at twenty games of 2048 (calls to the function game which take care of the game logic.

mutate_layer: This function takes weight and biases of the agents and a probability as input. It iterates through each element in the weight's matrix and biases array. For each element, with a probability of "prob", it replaces the current value with a new random value.

crossover_layer: This function performs crossover between two sets of weights and biases with a given probability. For each parameter, with a probability of prob, it will take the weight and bias of agent a or b.

Agent class:

1. **__init__:** Initializes the neural network with random weights and biases. It defines fully connected three layers: input layer, a hidden layer, and an output layer.
2. **run**: this method takes as an input a matrix, flattens it to 1-D vector, and passes it through the neural network layers. The output is a list of tuples, each containing a neuron score and its index, sorted in descending order.
3. **mutate**: Applies the mutation operation to the neural network layers with a given probability.
4. **crossover**: Applies the crossover operation between the current neural network and another neural network with a given probability.

5. **fitness:** Evaluates the average score of the neural network over 20 games played.

Game logic.py

GameLogic is based on 2048 Game in Python - GeeksforGeeks[3] which responsible for changing the state of the board based on the agent action. It contains essential functions for managing the game state:

1. **start_game**: Initializes the game/grid at the start.
2. **add_new_2/4**: Adds a new 2 or 4 to the grid at any random empty cell.
3. **get_current_state**: Retrieves the current state of the game and determines if the player won or lost.
4. **compress**: Compresses the grid after every step, both before and after merging cells.
5. **merge**: Merges the cells in the matrix after compression.
6. **move_left**, **move_right**, **move_up**, **move_down**: Updates the matrix according to the agent moves/swipes.

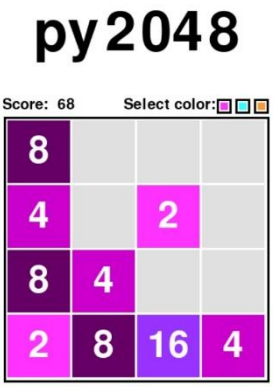
Game.py:

Gets an agent as a parameter and simulates its game duration from start to end. After each turn our evolution system changes the score of the agent based on a set of rules we defined.

[3] 2048 Game in Python - https://www.geeksforgeeks.org/2048-game-in-python/

Graphic.py:

Graphic representation of a game played by an agent. The visual aspect is based on a code from Andrea Vuce repository on github[4], we altered the original code so that instead of getting input from the user it will load agent that we saved in advance (example_agent.pickle) and each time the user press the space bar the agent makes a decision of what he think the best move will be and we can see this move take place in real time.



The experiment

Definitions:

Game score - the average of the highest tile values that the agents manage to get to over a period of ten different games.

Fitness function- rewarding system determined to achieve specific behavioral patterns.

Fitness score-the average of 10 outputs of the fitness function that an agent got over the course of 10 different games.

Stage A – hyper parameters

We started off our experiments by using the raw values in the board as the neural network's input. But after a few unsuccessful runs we decided to apply log function over the date to normalize it and reduce the impact of high value on the output.

At first, we wanted to determine what the values of the generation size and number of generations would be. We started off with 100 generations, then increased to 200 generations and the last trail for this hyper-parameter was a run with 400 generations.

[4] py2048 - https://github.com/AndreaVuce/py2048/tree/master by AndreaVuce

We saw that the game score of the best agent from each generation stayed almost the same in each of the trails mentioned above

In addition the best game score from a generation to generation was very "noisy" as can be seen in the graphs, so we decided to move forward with 200 generations in order to achieve better efficiency when it comes to running time.

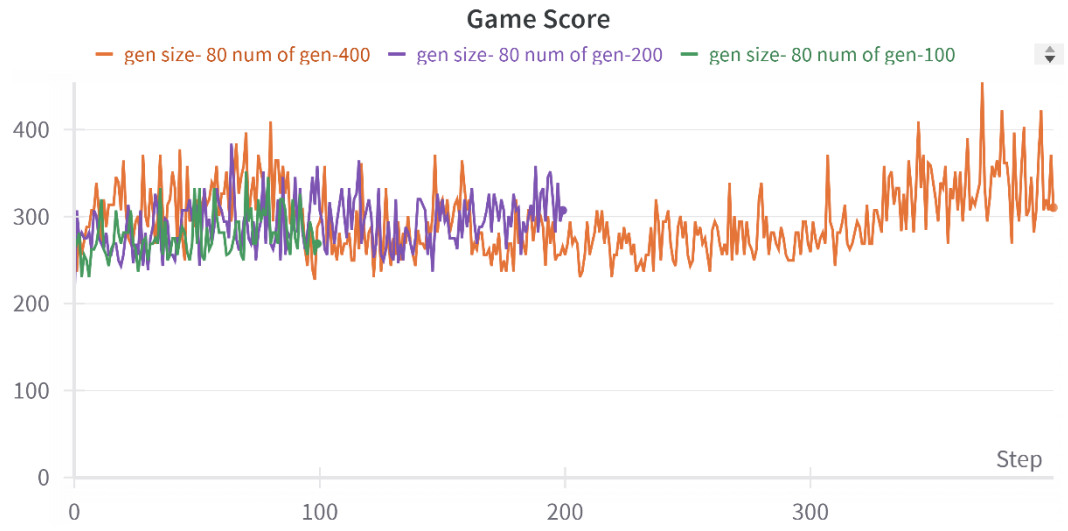


Figure 2- almost equal game score for different number of generations

The next parameter we checked was the number of agents per generation. We discovered that this parameter has a larger impact over the results. After some trials we have concluded that the best setting for the algorithm is 200 generations and 320 agents in a generation.

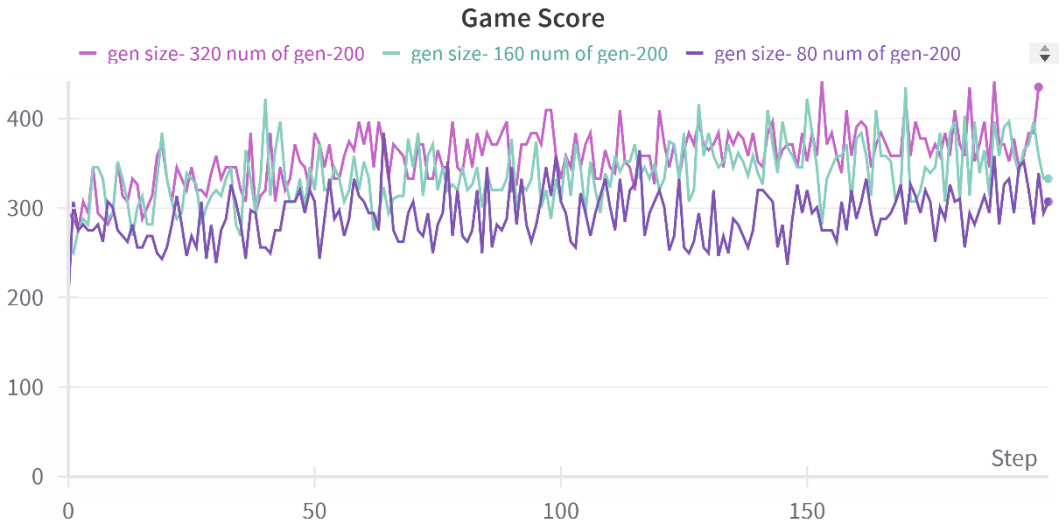


Figure 3- As we can see 320 agents in a generation yields the best game score on avarege

Stage B – Fitness function

The next stage was to try to achieve better score by using different fitness functions to "help" our agents that used "smarter" strategies.

First strategy that we have tried to reward was keeping the highest tile in one of the corners of the board, as we know when the highest tile isn't at the corner it is much harder to collapse equal tiles together.

We have chose to give extra 10 points in each turn the highest tile was at the corner and to deduct 10 points elsewise. At the end we summed up the score that the agent has achieve during the game with the real game score.

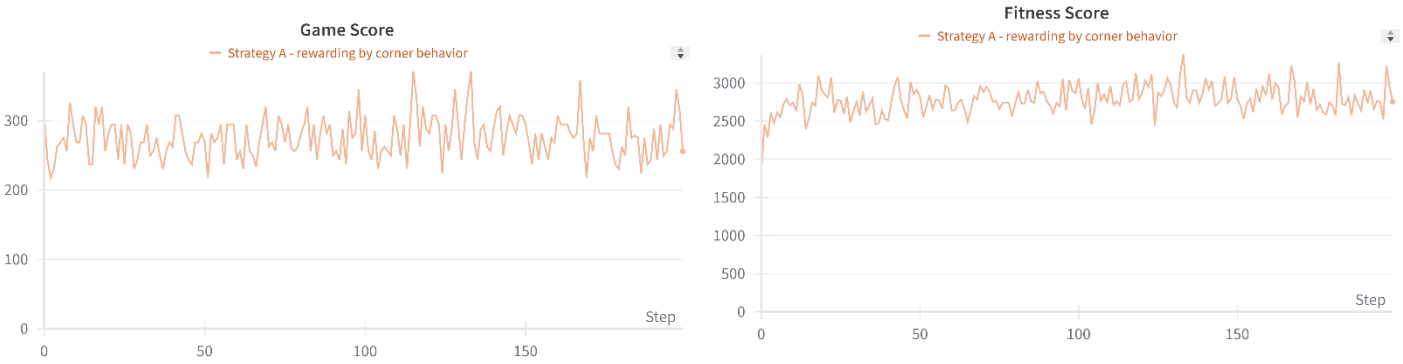


Figure-4 Game score and fitness score as function of generation using strategy A

After seeing no improvement, we decide to give more significance to the actual result on the tile rather than to the strategy (ratio of 1:5 between fitness score and game score when summing them up at the end) we kept this ratio for the next strategy as well.

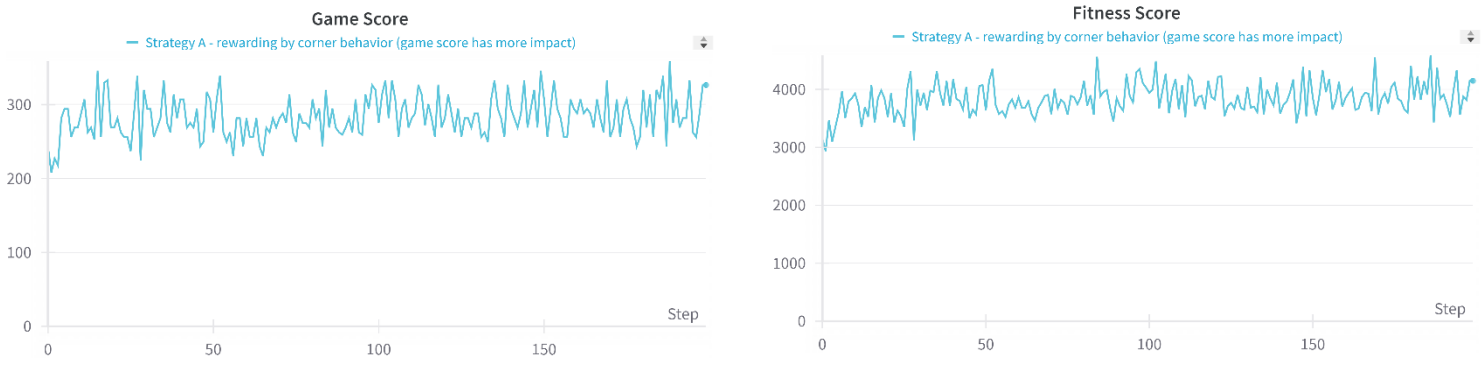


Figure5 - Game score and fitness score as function of generation using strategy A with the new ratio

The second strategy that we have tried is to give extra 20 points if the agent kept the highest tile in one of the corners of the board and in addition make sure is numbers are organized in monotonic order (for example: creating a row with 256 at the edge followed by 128, 64 and 32 in the same row or column as the highest tile), in case the agent move the highest tile from the corner we deduct 10 points as before.

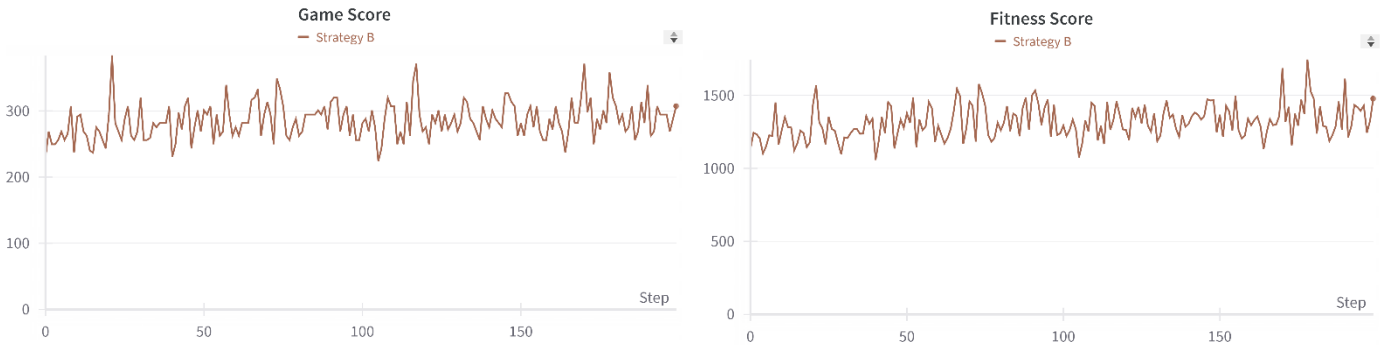


Figure6 - Game score and fitness score as function of generation using strategy B

Stage C- back to hyper parameters (mutation & crossover)

After seeing that our fitness function wasn't enough for us to reach a good outcome we went back and changed our crossover and mutate parameters. In the previous runs the probability of a certain agent getting mutated was 0.05 – we decided to increase that probability to 0.1 to prefer exploration over exploitation.

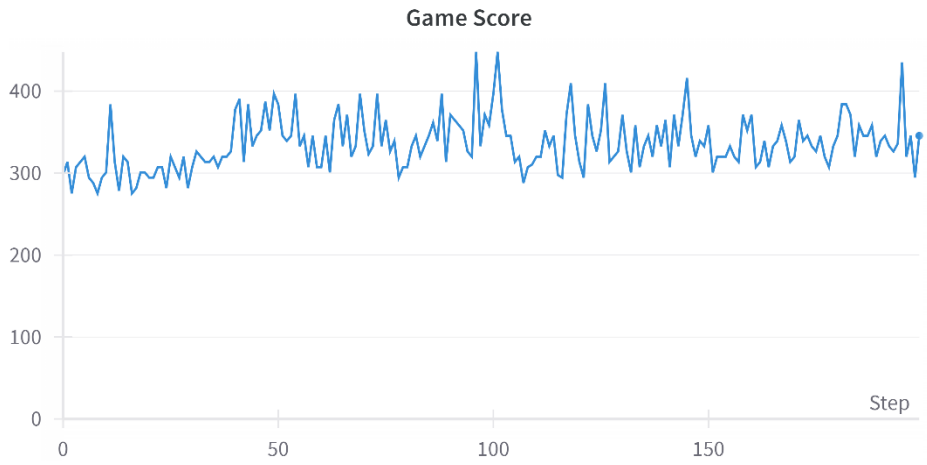


Figure 7- game score as function of generation in stage C

For our next run we increased the probability for mutation to 0.3 and we also decreased the cross over parameter which was 0.5 in all the previous runs to 0.3 and preferred the better agent when cross-overing them.

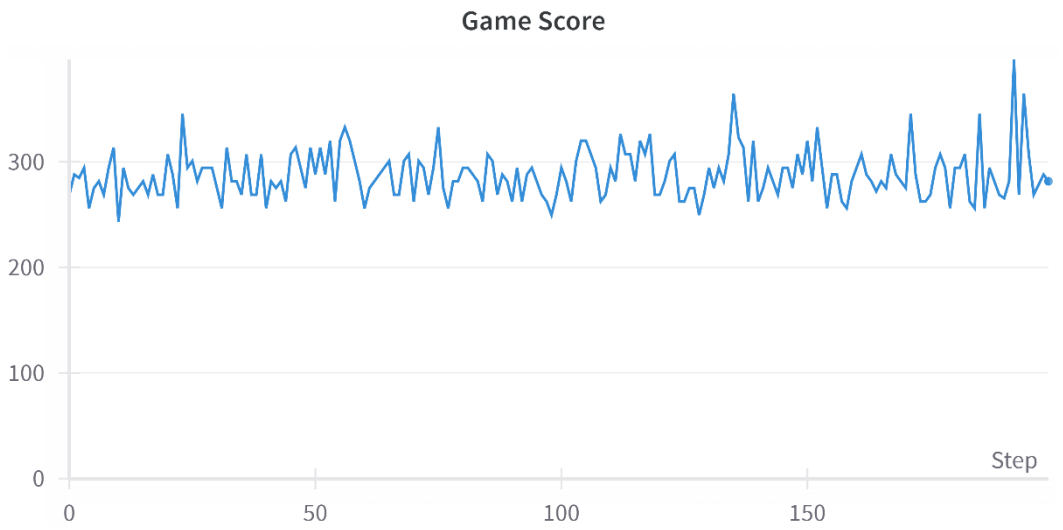


Figure 8- game score as function of generation with crossover parameter of 0.3

Stage D – Last attempt

As we saw we couldn't achieve better results using our predefined strategy aimed scoring system. Therefore, we decided to try another trail using a bigger generation population, in attempt to see whether it will help to achieve better results or whether we got to saturation.

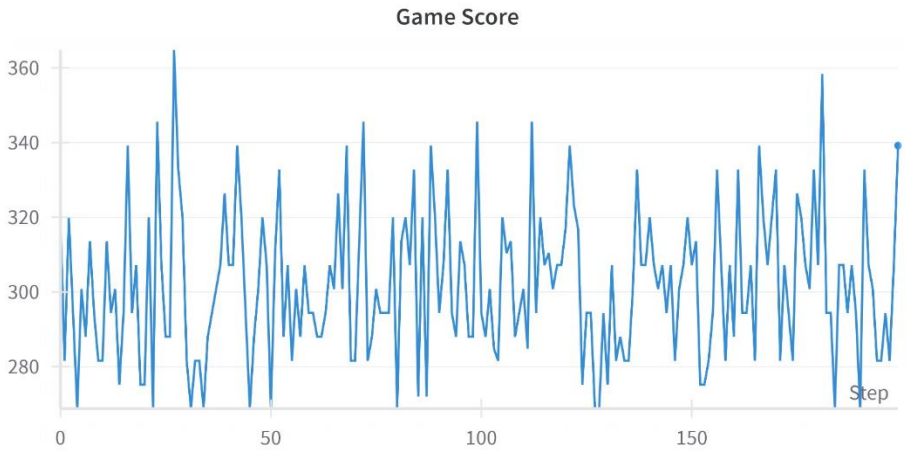


Figure 9 - generation size of 640 agents.

As can be seen in figure number 9, we used 640 agents in a generation and only evaluated them using the game score they manage to achieve. But we didn’t get better results than the 320 agents in a generation meaning that we got to saturation.

Conclusions

Generation size and number of generations

We conclude that the hyper parameter generation size has the biggest impact on our results in the current network structure. We run varies experiments with different numbers of generations and found out that without enough agents in each generation we won't be able to reach good results and achieve learning process, this is why we end up going from 80 agents in each generation at the beginning to 640 agents at the end.

Another perspective that might had effect over the test was that we initialized the agents' weights (at the neural network) using random normal distribution and therefore we might started searching far from the optimal solution.

On the other end we found out that the number of generations hold lower impact (going from 100 generations to 400 made almost no difference) so we decided to work on 200 generations each time for more efficient running time.

Fitness function

Several factors contribute to the challenges inherent in this task. Firstly, the game 2048 presents a vast and intricate searching space, making it challenging to explore effectively. Our agents had a lot of "experience" with dealing with small numbers, but it takes time to reach larger values which makes the learning process of the agents from a certain point of the game less efficient and biased to lower values. We tried to overcome this issue by creating fitness function that gives rewords based on the ratio between different tiles, such as the rewarding sorting the tiles in monotonic order on the board or based on the location of the tiles.

However our fitness function alone wasn't enough in order to deal with this issue, and we believe that this problem might requires a larger neural network with more

parameters and different architecture.

Another suggestion we have for future work might be to include differential scoring system that reword "good moves" as function of the number of turns we played, i.e. as the game proceed we shall give more points for following our strategy.