Recommendation system to obtain the next best movement in 2048 game

Carlos Domínguez Becerril and Aitor González Marfil

Abstract

2048 is often played on a gray 4×4 grid, with numbered tiles that slide when a player moves them using the four arrow keys. Every turn, a new tile will randomly appear in an empty spot on the board with a value of 2. 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 resulting tile cannot merge with another tile again in the same move. The intend of this project is to build a neural network using neuroevolution to add a component that improves the game experience, in our case, we implement a recommendation system to obtain the next best movement that can be done in the game to make the experience for the user easier.

1 Description of the problem

The goal of this project is to insert a neural network component as part of a computer game and show that this added component improves or enhances the game experience in anyway. In our case we are going to build neural network in order to give a hint to the user of which one is the next recommended movement to beat the 2048 game.

1.1 Description of the game

The game table consists in a grid of 4x4 where each cell contains a value or not. The player has 4 possible movements:

- Up $\rightarrow$ Move all the numbers to the top.
- Down $\rightarrow$ Move all the numbers to the bottom.
- Left $\rightarrow$ Move all the numbers to the left.
- Right $\rightarrow$ Move all the numbers to the right.

When the player applies any of these movements the grid is updated, therefore, every value inside moves according to the selected direction until it arrives the limit of the grid or hits a cell with a different value. If it hits a cell that contains the same value, they will merge generating a new value with the sum of each cell, for example, 2+2=4, 16+16=32. The values are powers of 2, being the lowest one 2 and the highest one 2048. After each movement it appears a new value of 2 inside a random empty cell. With this mechanics, the goal of the game is to obtain a cell with the 2048 value. The number of moves needed to finish the game is not fixed but according to [1] the minimum number of turns required to win the game, assuming we play a perfect game is in the range of 519 and 1032.

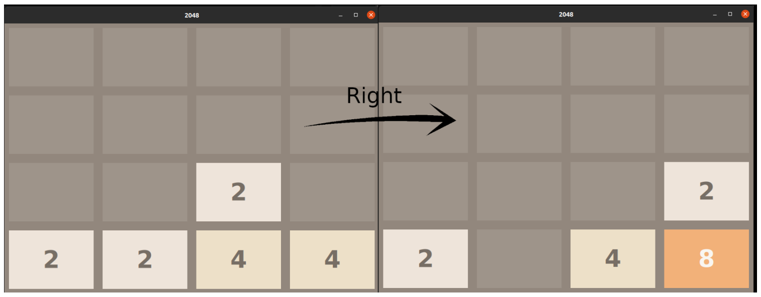

Figure 1: Example of two merges.

On this image, we can see an example of two merges at the same time when we apply a move to the right (the two 4s merge making an 8 and two 2s merge making a 4). Moreover, we can see how a new value appears at bottom-left side of the grid. If our next decision is to apply a down movement the grid will be exactly the same making no changes and avoiding the creation of a new random value.

2 Description of our approach

We organize the implementation of the project in the following steps:

1. Game implementation.
2. Neural network design.
3. Neural network training.
4. Neural network validation.

2.1 Game implementation

We used an implemented version of the game by Yangshun Tay, available on GitHub [2] under the MIT license.

The game is divided in three files:

1. ***Logic.py***: Includes the logic of the game that allows to perform the different movements, tiles merging, creation of new tiles...
2. ***Constants.py***: Includes the constants that the GUI is going to use. It includes: colors, valid keys, fonts, grid size, window size... The available keys to play the game are the following according to the direction:
   - **Up:** ”w” or ”k”.
   - **Down:** ”s” or ”j”.
   - **Left:** ”a” or ”h”.
   - **Right:** ”d” or ”l”.
3. ***Puzzle.py***: Includes the GUI of the game developed using Tkinter Python library. We create two versions of this file:
   - The first version removes the GUI part leaving only the core of the game and some extra functions. This one is used to train the neural network.
   - The second version includes extra functions to make the neural network work with the game and an extra window with two buttons. The first button activate / deactivate the hint system and the second button activate / deactivate the ability to play the game autonomously.

2.2 Neural network design

The neural network design is straightforward. The first layer (input layer) consists of 16 neurons, one for each value in the grid, then we add a hidden layer with 8 neurons (half of the input) and finally the output layer consists of 4 neurons (one for each direction that we can perform). To the output layer we apply the sigmoid function so we can get values between 0 and 1 and then we apply softmax to this output so the sum of our output equals to 1. This last step is done so we can take the index of the neuron with the highest value as our direction, therefore, it can be seen as the confidence level for a movement. Finally, to ensure that we perform a valid movement we make an element wise multiplication with the valid movements (a vector of 0s and 1s where 0 is an invalid movement and 1 a valid movement).

Indexes according to each movement:

- 0: up
- 1: left
- 2: down
- 3: right

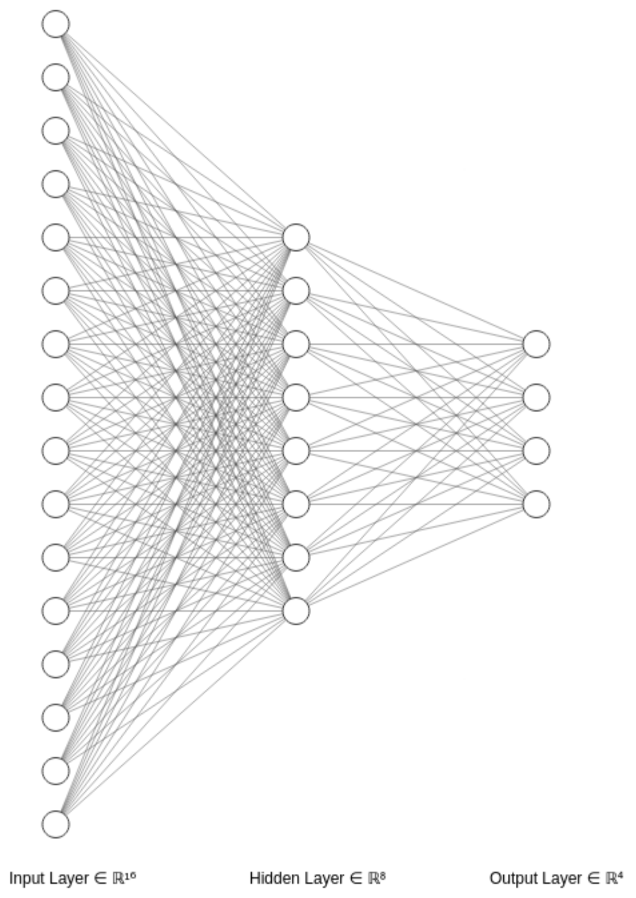
Input Layer $\in \mathbb{R}^{16}$
Hidden Layer $\in \mathbb{R}^{8}$
Output Layer $\in \mathbb{R}^{4}$

Figure 2: Neural network architecture (fully connected).

The neural network is built using the ”neat” Python library with the default configuration file and only modifying the input layer variable to 16, the number of hidden layers variable to 8 and the output layer variable to 4. We use neat so we can apply neuroevolution and learn the right connections and weights.

2.3 Neural network training

The training of the neural network is done using neuroevolution and therefore the most important part of it is the fitness function to be used. We have decided to implement two fitness function that tries to maximize the fitness of each genome over 500 generations and then compare the results in order to select the best one.

2.3.1 Fitness function 1

This fitness function takes into account the number of steps performed when playing the game since performing more steps will mean that is closer to 2048.

We reward the following points according to the three different statuses:

- not over: For each step we give to the fitness function one point.
- win: (5000 - number_of_steps) * 2. Winning the game gives 5000 points but we decrease it by the number of steps. Using this we encourage the neural network to win in the smallest number of steps while following the actual path.
- lose: number_of_steps / 2: Losing the game gives half of the points (number of steps). Using this we encourage the network to follow a different path.

2.3.2 Fitness function 2

This fitness function combines our ideas with the proposal done by Ovolve user on Stack Overflow [3], we can see his implementation in the GitHub repository [3].

The fitness function has multiple parts:

- **Smoothness:** Measures how smooth the grid is (as if the values of the pieces were interpreted as elevations). Sums of the pairwise difference between neighboring tiles (in log space, so it represents the number of merges that need to happen before they can merge). Note that the pieces can be distant. Smoothness rewards more if similar blocks are near to each other to merge them later. This helps to merge multiple blocks at the same time using one movement.

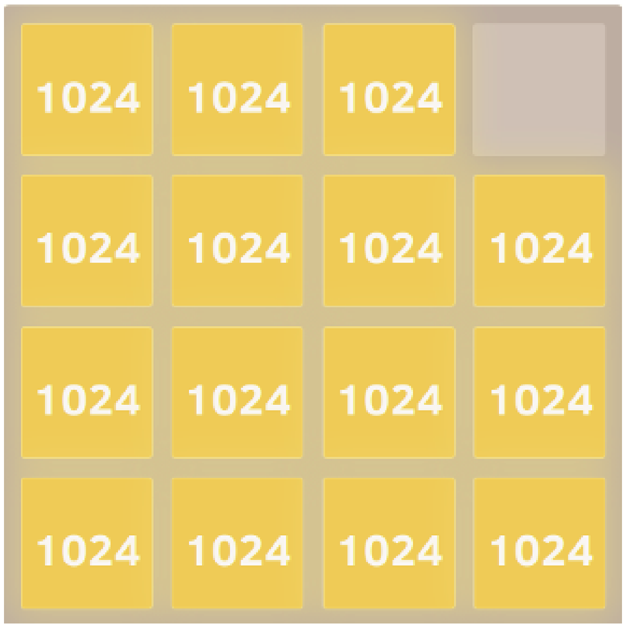

Figure 3: Example of a perfectly smooth grid.

- **Monotonicity:** Measures how monotonic the grid is. This means the values of the tiles are strictly increasing or decreasing in both the left/right and up/down directions. Monotonicity rewards more if the blocks are in increasing or decreasing order along the grid. This helps to create a chain effect and merge a lot of blocks in a small number of movements in order to obtain a bigger block. Moreover, we reward the fact that the biggest blocks are usually in one of the corners and therefore we force the game to add new blocks in the opposite side in order to perform this chain effect.

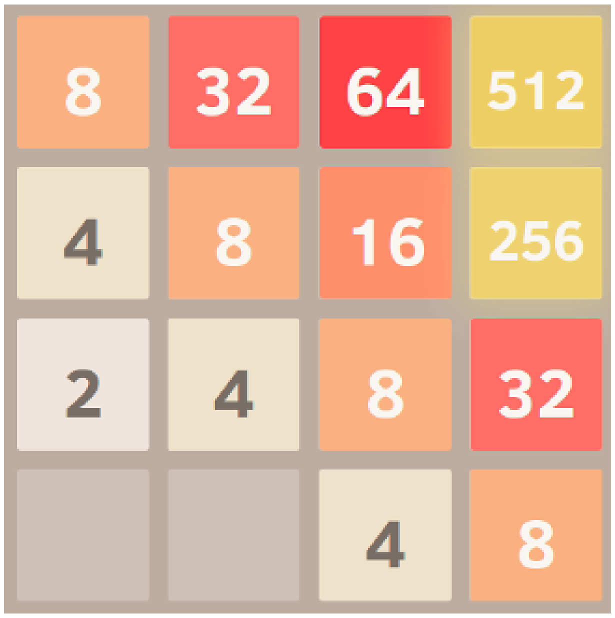

Figure 4: Example of a perfectly monotonic grid.

- **Empty Cells:** Measure the number of empty cells on the grid. It rewards more when we have empty cells because it allows players to add more blocks. Viewing this from the opposite side we are rewarding less points when we have the grid full of numbers because this means that the player is close to lose the game.
- **Biggest block:** Is the value of the biggest block obtained. The bigger the block is, closer we are to obtain the 2048 block and win the game, in that sense, the reward is bigger.
- **Sum of all blocks:** Is the sum of all the blocks. The higher the sum is, closer it is to obtain the 2048 block and therefore the reward is bigger.

The final results of the fitness function is the sum of all the parts multiplied each one by a weight in order to give them more / less importance according to what we want to obtain. In our case we use the default values by Ovolve.

The weights for each part are the following:

- Smoothness: 0.1.
- Monotonicity: 1.0.
- Empty cells: 2.7.
- Biggest block: 1.0.
- Sum of all the blocks: 0.03.

We reward the following points according to the three different statuses:

- not over: The fitness function mention before.
- win: The fitness function mention before plus 500 in order to encourage to follow the actual path.
- lose: The fitness function before minus 400 in order to encourage to search for an alternative path.

2.4 Neural network validation

For validation we make the neural network play 1000 games and record the value of the highest number tile of the grid after finishing each game to measure how close we have been from the objective and the average number of steps performed to arrive to that result.

3 Implementation

The implementation consists in the following steps: Game implementation, neural network training, neural network validation and game testing. All these steps were implemented in Python. We use NEAT for evolving the neural network and Tkinter for the game graphics. The implementation of how it works can be found in the Python Jupyter Notebook ”P35_Notebook_Dominguez_Gonzalez(fitness1).ipynb”. We also include the notebook ”P35_Notebook_Dominguez_Gonzalez(fitness2).ipynb” with the execution of the second fitness function. The pretrained genomes can be found under the name ”winner_genome_fitness1.neat” and ”winner_genome_fitness2.neat” corresponding respectively to each of the fitness functions.

The implementation of the 2048 game ”logic.py”, ”constants.py” and ”puzzle.py” are included in separated files in order to not overload the notebook. We include an extra empty folder call ”genomes” for storing the genomes if the notebook is executed.

Warning: The notebook can take more than 1 hours using the first fitness function and more than 2 hours using the second fitness function.

4 Results

The results obtained are the following according to the fitness function.

4.1 Fitness function 1

The number of times that each tile appears after finishing the game can be found in the following table:

Tile value	Number of times
64	26
128	175
256	489
512	305
1024	5
2048	0

Table 1: Maximum score obtained in 1000 games using the first fitness function

The average number of steps needed to finish the game is 369.

We can see that the model achieves 256 tile 49% of the times, 512 tile 30% and 1024 tile 0.5% of the times. There are few exceptions where it obtains worse results. Nevertheless, it obtains 256, 512 and 1024 tiles nearly 80.5% of the times, therefore, we can consider that the neural network is pretty stable even if the results are lower than expected.

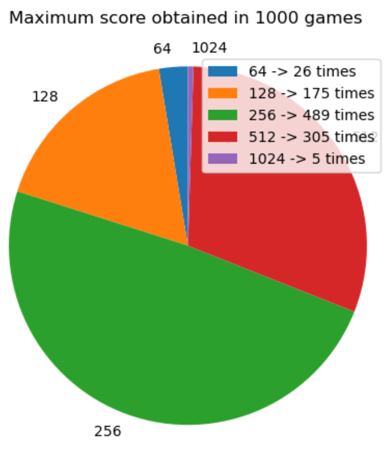

Figure 5: Pie chart showing how many times each tile appears (fitness 1)

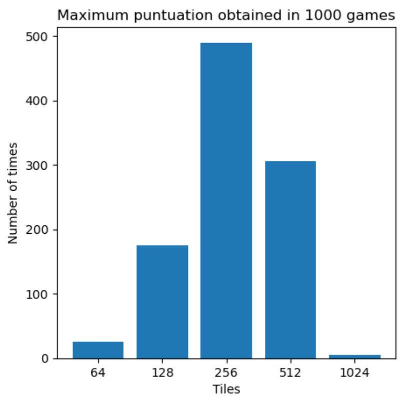

Figure 6: Bar chart showing how many times each tile appears (fitness 1)

4.2 Fitness function 2

The number of times that each tile appears after finishing the game can be found in the following table:

Tile value	Number of times
32	1
64	16
128	153
256	428
512	354
1024	48
2048	0

Table 2: Maximum score obtained in 1000 games using the second fitness function

The average number of steps needed to finish the game is 399.

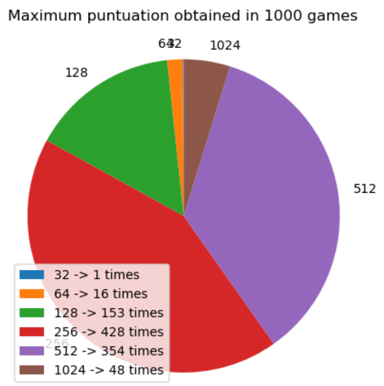

Figure 7: Pie chart showing how many times each tile appears (fitness 2)

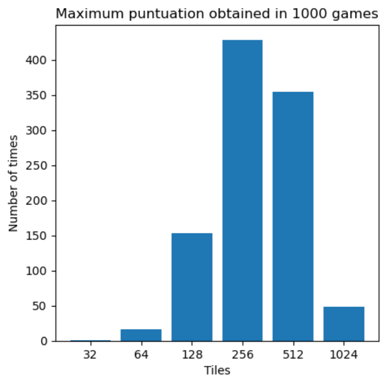

Figure 8: Bar chart showing how many times each tile appears (fitness 2)

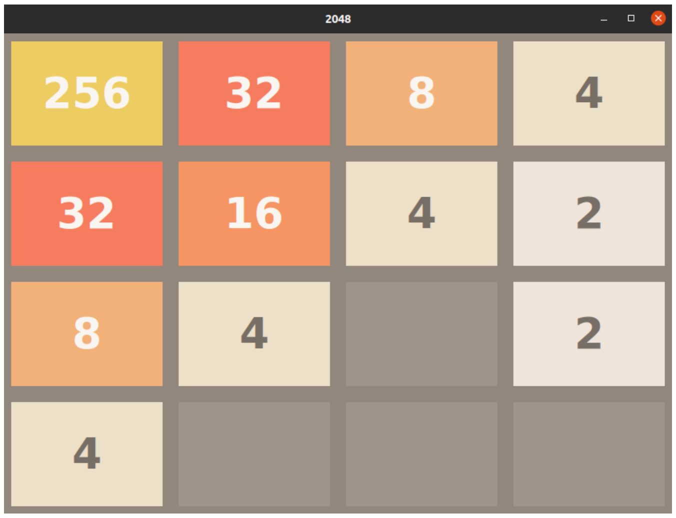

Figure 9: Example using the second fitness function.

We can see that the model achieves 256 tile 43% of the times, 512 tile 35% and 1024 tile 5% of the times. There are few exceptions where it obtains worse results. Nevertheless, it obtains 256, 512 and 1024 tiles nearly 85% of the times, therefore, we can consider that the neural network is pretty stable even if the results are lower than expected.

Both models perform similar but the second one is capable to obtain better results than the first one. Nevertheless, in both cases the results are lower than expected because the neural network is not able to beat the game, even though in some rare cases it could.

5 Conclusions

Comparing both neural networks we realize that the first one needs to learn by itself patterns that the second one includes in the fitness function. Considering that we are using a simple neural network, its learning is limited and cannot perform well, therefore, due to the second one being more guided thanks to the fitness function it performs better.

In general, both fitness function can be considered good, especially the second one that performs according to the fitness purpose (see figure 9). Nevertheless, neither the first one nor the second one is able to get the objective value of 2048, mainly because after a visual check on the second one we realize that a wrong movement could add a new tile in a position where it is surrounded by bigger tiles making it lose due to the impossibility to make a merge.

This game can be considered difficult for humans and therefore a simple neural network cannot perform well either. The results could be improved using deep neural networks like convolutional neural networks in order to take into account the neighbor values when making a move and recurrent neural networks in order to remember previous movements and act accordingly. The results of using a deep neural network can be found in [4].

6 Questions

1. What class of problems can be solved with the neural network?
2. What is the network architecture?
3. What is the rationale behind the conception of the neural network?
4. How is inference implemented?
5. What are the learning methods used to learn the network?

6.1 What class of problems can be solved with the neural network?

The problem we can solve using a neural network is to select the next movement with a evolutionary algorithm (neuroevolution). The task we are solving makes a recommendation of which is the best movement according to one state of the game. This can be done using a ranking system to rank the movements according to the confidence level (which one is better than the other) and selecting the best one.

6.2 What is the network architecture?

The neural network architecture is explain in section 2.2 Neural network design.

6.3 What is the rationale behind the conception of the NN?

The neural network architecture is explain in section 2.2 Neural network design. We have 16 input neuron layer each one corresponding to one value in the grid and 4 output neuron layer each corresponding to one movement.

There is not a fixed architecture due to using neuroevolution, this means that every generation we will have different connections and different weights.

Nowadays, there are no ways to know which one is the perfect movement according to one state in the board. Obtaining the best movement using traditional algorithms is computationally expensive due to adding more stochasticity every movement and therefore, in order to solve this problem efficiently, a neural network can be trained to obtain a good approximation.

6.4 How is inference implemented?

The inference is implemented making a complete forward pass through the network, where the input is the grid values and the output the expected movement. A better explanation can be found in section 2.2 Neural network design.

6.5 What are the learning methods used to learn the network?

The learning method used to learn the neural network is an evolutionary algorithm (neuroevolution).

References

[1] Bhargavi Goel. Mathematical Analysis of 2048, The Game (2017). Paper.

[2] Yangshun Tay. 2048 game (2013). GitHub.

[3] Ovolve. 2048 game AI (2013). GitHub, StackOverflow.

[4] Navjinder Virdee. 2048 using a deep neural network (2017). GitHub.