Best Move Using FFNN For 2048 Image

Alexandru Micsoniu
A01039572
amicsoniu@gmail.com

Garik Smith-Manchip
A00900749
gariksm2@gmail.com

Abstract

This paper introduces a single-layer feed-forward neural network (FFNN) that is trained as a multi-layer FFNN to predict the next best possible move using the most important weighted parameters. Furthermore, there will be functionality that processes an image, utilizes Stochastic Gradient Descent (SGD) to classify it as either a 2048 board or not, and transforms it into a game state for the FFNN. The SGD classifier performs image classification and board state transformation with 100% accuracy. The FFNN model showed vast improvements in achieving 2048 in all experiments as well as providing the best possible move.

1 Introduction

1.1 Problem Overview

Someone playing 2048 may not know what move to make that will progress them further towards achieving a successful 2048 tile. This model will take in an image classified as a 2048 board and use a single-layer FFNN that is trained as a multi-layer FFNN to predict the next best possible move using the most important weighted parameters.

1.2 Scope and Limitations

The following items limit the scope and state the limitations of the project.

1. Generate a variable number of 2048 board states for training neural network.
2. Generate images of 2048 boards that will be used for training image classifier.
3. Generating Board images must be done using a 4k monitor.
4. AI will stop game simulations after 2048 is achieved or no possible move exists.
5. AI will use a multi-layer FFNN for training and single-layer FFNN for simulations.
6. Image Classifier will use SGD from sklearn tool kit.
7. Image Classifier will be binary (board or not_board).
8. Simulation will take an image and produce the best possible move to make.
9. Neural Network will train 250 states at a time, up to 2,500 states.

1.3 Manual

Action	Run Command
Generate States	python generate_boards.py -n [number of states] -t [random, new, mix] -o [statefile]
Generate State Images	python generate_board_images.py -d [statefile]
Image Training	./run.sh
Neural Network Training	./run.sh
Experiment Training	./run.sh
Run Simulation	./run.sh
Training Photos Download Link	rb.gy/9jnldw

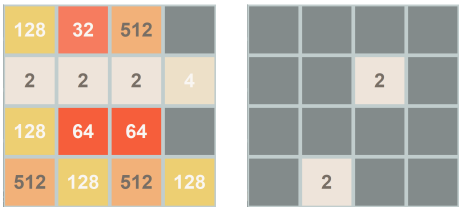
1.4 Hardware Specifications

Processor	Intel(R) Core(TM) i7-7500U CPU @ 2.70GHz 2.90 GHz
RAM	16.0 GB
OS	Windows 10 Home 64-bit
Resolution	3840 x 2160 (needed for board image generator)

2 Preliminaries

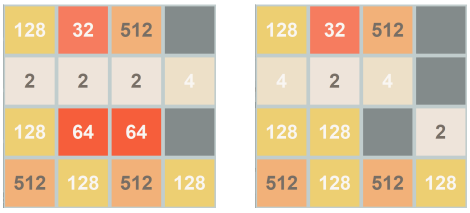
2.1 Game Presentation

2048 is a 4 x 4 tile game where you can move up, down, left, right, to merge like-valued tiles into larger valued tiles in an attempt to get 2048. The possible testing board states are represented as random or new.



Random State New State

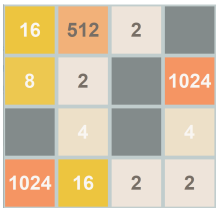
Choosing a direction will slide the tiles together creating a new board state. After a slide takes place, a new tile will spawn with a 90% chance to be a 2 and 10% chance to be a 4.



Input State State After Slide Left

2.2 Game State Structure

The input data structure takes in an image of a 2048 board and separates the board into a two dimensional vector of values. This vector will be converted so that each value is represented in its exponent form where 0's are read as 0.

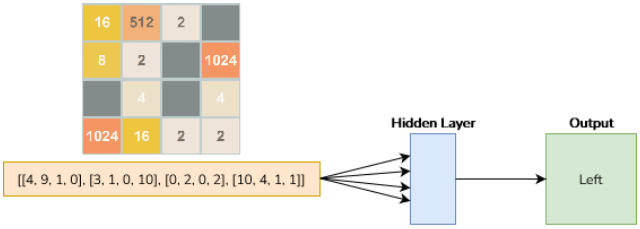


Step 1

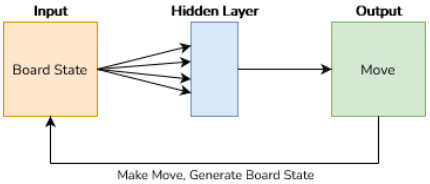
Step 2: [[16, 512, 2, 0], [8, 2, 0, 1024], [0 4, 0, 4], [1024, 16, 2, 2]]
Step 3: [[4, 9, 1, 0], [3, 1, 0, 10], [0, 2, 0, 2], [10, 4, 1, 1]]

2.3 Neural Network Model

The model will take in a board state and output the best possible move based on analysis of the trained feed-forward neural network (FFNN). This will function as a single layer neural network.



During training, the model will be a multi-layer feed-forward neural network that makes the best possible move then progresses the board state forward to a terminal state through either achieving 2048 or by finding a locked board state where no more moves can be made.



2.4 Image Classifier Model Selection

The decision to use Stochastic Gradient Descent (SGD) was made. It appealed in a number of ways, such as its efficiency and its ease of implementation that allows for a great deal of tuning. Furthermore, decisions made in this project allows the bypass of some of SGD's weaknesses, such as its sensitivity to feature scaling which is reduced via Hog Transformation. The SGD image classifier is implemented with the help of the scikit-learn API.

3 Design

3.1 Parameters and Weights

The parameters represent four advantageous board states. When deciding between the four possible moves, the board is analyzed based on the following questions and saved as a binary representation or 1 if answered as yes, or 0 is answered no.

Highest Value	Does the board state have the highest value tile compared to the others?
Home Row	In relation to the highest value, is that tile's row in ascending or descending order?
Home Column	In relation to the highest value, is that tile's column in ascending or descending order?
Home Diagonal	In relation to the highest value, is that tile's diagonal, to the opposite corner, in ascending or descending order?

If successful end states are achieved and certain parameters are visible as a 1 then they will be deemed more important and have their weight increased. If they are present in a losing state then their value will decrease.

The four model weights will be set to 0.25 to start and will progress with the adjustment formula value = value +/- (value × adjustment) where the adjustment descends from 1% to 0.5% in 0.05% increments for training. After that, the weights get normalized to sum to 1.

Initially, we believed that the above parameters would be weighted mostly evenly with a slight lean towards either favouring the home row or home column as the most important parameter. This would either make all parameters hyper parameter's or lead to the favoured home row/column being the sole hyper-parameter needed for adjustments.

During the experiments, we learned that the home diagonal was the leaned upon parameter of choice when training our model while the highest value parameter became almost non-existent.

3.2 Adjustment Considerations

Before proper training can take place, we must know two things, the variation of states to train on and the adjustment of the parameter weights when an end state is reached.

Total = 20	Random States		New States	
	Success Rates	Weights	Success Rates	Weights
Baseline	0.15	-	0.00	-
Front Weighted	0.35	0.27, 0.13, 0.50, 0.34	0.00	0.13, 0.57, 0.17, 0.17
0.001	0.25	0.25, 0.25, 0.25, 0.25	0.00	0.25, 0.25, 0.25, 0.25
0.01	0.30	0.21, 0.26, 0.26, 0.27	0.00	0.25, 0.25, 0.25, 0.25
0.10	0.25	0.04, 0.25, 0.25, 0.46	0.00	0.20, 0.30, 0.25, 0.25
0.20	0.25	0.00, 0.19, 0.19, 0.62	0.00	0.16, 0.36, 0.24, 0.24
0.50	0.25	0.00, 0.03, 0.03, 0.94	0.00	0.06, 0.56, 0.19, 0.19

The selected adjustment factor will start at 0.01 because it shares the qualities needed for separating the weights properly while having a high success rate. The new boards failed to achieve a success state because they had no bias towards what success was. On average, random states took 3.5 seconds to complete with an average of 35 moves and new states took 1,300.8 seconds with an average of 361. The baseline for a completed successful game should average 938 moves for a new board.

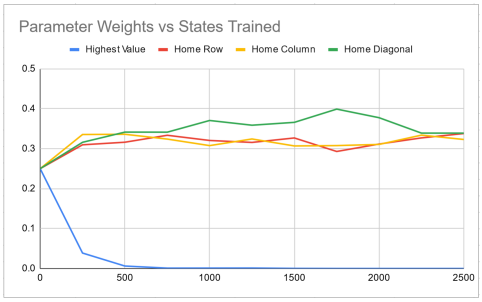
Random moves found success in 15% of random boards and 0% of new boards.This was due to there being no initial bias in the parameter weights meaning that it couldn't start finding wins early to swing the adjustments in favour of completing a game, in the 20 test states

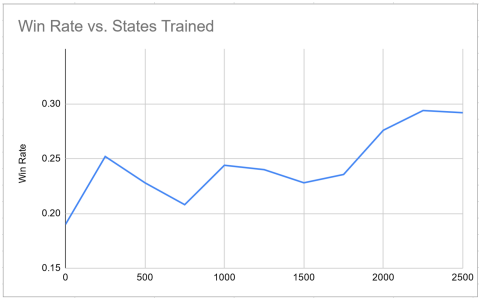
3.3 Neural Network Training

The model was trained on 2,500 states where 95% are random and 5% are new.

	Minimum	Baseline	Oracle	Hypothesis	Results
Average Moves	0.0	11.0	200.0	62.0	15.04
Average Win %	0.0%	19.0%	100%	20%	29.2%
Average Game Time	0.1 sec.	60.0 sec.	0.1 sec.	62.0 sec.	14.05 sec.

The model completed games much faster than expected because about 20% of board states were generated without any possible moves.



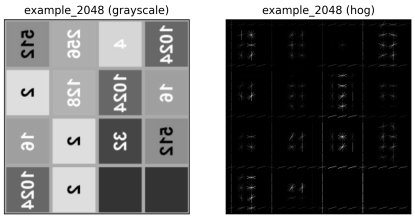


Originally we assumed the weight of the highest value parameter would have the largest weight. Turns out it is the least important parameter as the others average about 33% of the weight each, with a alight leaning towards the home diagonal which was a complete change from the hypothesis that this would be the least important parameter out of the four.

The win % was well above what we expected at 29.2% meaning the model was able to find successful states of board that would have been hard for a human to successfully achieve 2048 in.

3.4 Image Classification Training

Images are loaded, resized, and written as arrays to a dictionary together with labels and metadata. Once this data is read and split into training and testing, the array of images are transformed from RGB to grayscale, and a process known as Hog Transformation is applied. Hog Transformation allows for feature reduction, thus lowering the complexity of the images without sacrificing the variation within images. The example below shows the effect a Hog Transformation has on a 2048 board:



Despite the hog image appearing as gibberish to the human eye, a computer can still use the information in it to make reliable comparisons and accurate classifications. The final transformation is Standard Scalar, which allows for the standardization of a dataset. Finally, we fit the training data to the SGD classifier, and test it.

3.5 Image Classification Testing

The Stochastic Gradient Descent (SGD) image classification model was trained on 1713 2048 boards, 750 chess boards, 245 go boards, and 718 random images. These categories of images are split into two folders: a folder called 2048_boards containing the 2048 board images, and a folder called not_2048_boards containing all other images. The training and testing split is 0.90 and 0.10 respectively. These numbers are based on a 0% minimum, 95% baseline, 99% hypothesis, and 100% oracle.

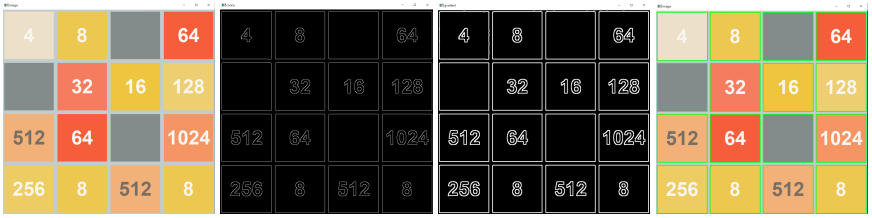
	Precision	Recall	F1-Score	Support
Boards	1.00	1.00	1.00	179
Not Boards	1.00	1.00	1.00	164
Accuracy			1.00	343
Macro	1.00	1.00	1.00	343
Weighted	1.00	1.00	1.00	343

The scores are excellent. They demonstrate that the SGD classifier is fully capable of highly accurate image classification with its current implementation and sample sizes. Furthermore, these results are obtained in an average computation time (over 10 trials) of 166 seconds, leading to 13 images processed per second.

4 Simulation and Experiments

4.1 2048 Board Scanner

This feature allows the program to take an image of a 2048 board, and successfully convert it to a game state that can be taken by the FFNN to determine the best next move. OpenCV is the tool that was extensively used to complete this functionality. Images of 2048 boards are loaded, sorted, and their file names saved to a list. Each image goes through 3 transformations: turned into a Canny image, a gradient is added, and finally contours.



As we can see, when an image is Canny-ed, all color is removed, and the borders of each tile are

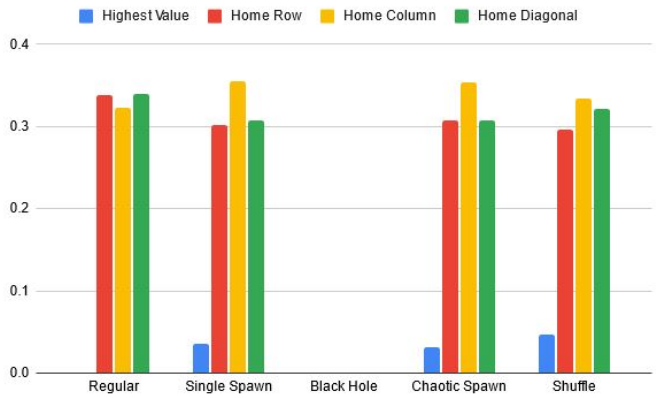
clear squares. The gradient added to the Canny image fills in gaps and again make the tile borders more pronounced. Finally, the program finds the contours (the tile borders) of the gradient image, superimposes those contours on the normal image, and extracts each tile as its own image. Each of those 16 tile images are then compared to a list of keys by subtracting the average RGB value of a tile image by the key, and finding the closest value to zero. The key that produced the closest value to zero is selected as the tile image's value. These values are then added to a list, their log base 2 values are calculated, the list is reformatted, and if these images were generated from a game board states file, compared for correctness. As for this program's accuracy, assuming the 2048 game board images are generated using generate_board_images.py, accuracy is at 100% constantly.

4.2 Re-Training Model

We will now compare the regular trained model against a set of experiments, retrained with the same 250 states with a 1% adjustment to avoid over-fitting. These experiment features will occur after every move. Single Spawn only spawns 2's on the board. Black Hole will designate one location as a void space that consumes the existing tile. Chaotic Spawn will fill all empty spaces by spawning a random valid tile from 2 - 1024. Shuffle will randomly shuffle the tiles.

	Average Win %	Average Moves	Average Game Time
Baseline	0.19	11.00	60.00 sec
Normal	0.29	15.04	14.05 sec
Single Spawn	0.22	17.32	16.96 sec
Black Hole	0.09	-	49.12 sec
Chaotic Spawn	0.20	3.02	1.93 sec
Shuffle	0.30	20.25	24.41

The black hole experiment was not able to successfully complete the experiment while the others shows at least a marginal improvement from the original baseline of a normal board.



The experiment results mimic the original hypothesis of the normal trained model to play regular 2048 with the home column coming out as the most important parameter, but not by much. The highest value also showed similar features of being not nearly as useful as needed. This parameter is still useful to complete games and achieve 2048 so that a locked terminal states does not occur.

4.3 Experiment Analysis

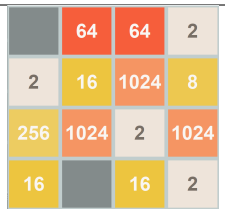
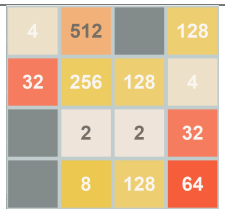
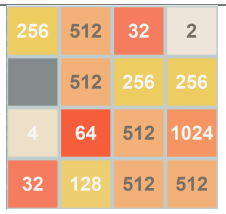
The first recognizable issue was that the black hole experiment was not able to complete the training. This was because it only successfully found the end game in 10% of training states where 22 of the 25 completed states were wins. The issues occurred when the majority of the games could not end properly due to an infinite loop of important tiles being consumed by the black hole. In the end, the black hole only succeeded in 8.8% of board states, making it the worst of the experiment results.

The single spawn was surprisingly worse off then the normal run. This could be either due to the parameters not being adjusted enough to fit the experiment or because spawning a random tile of either 2 or 4 achieves a better board state then spawning 2's at 100%.

Chaotic spawn, as expected, produced poor results. By filling the board with random valid tiles after every move meant that the board state generated from the previous move did not reflect the final board state outputted from the FFNN hidden layer. A quick understanding is that none of the hyper-parameters, except for possibly the highest value would serve a purpose in trying to complete this challenge. All of the victories occurred when a 1024 tiles spawned beside either a new matching tile or a previous matching tile to instantly complete the 2048.

Finally, the most interesting experiment was the 30% success rate from the shuffle experiment. The average moves jumped up from the previous tests as well. We hypothesized that this experiment would fail similarly to chaotic spawn because of how random the board states become. The true results showed that shuffling a board state that contained the properly distributed tile values ended up simplifying a position that adhered to non of the parameters to satisfy at least one of them after the shuffle occurred. The fallout from this means that the best neural network maybe be to have a mutation factor that shuffles the or mutates positions to try and reflect possible future positions more-so than just simulating a predicted game from the starting board states.

4.4 Simulation Analysis

	State 1	State 2	State 3
Images			
Best Move	Left	Down	right

The first result shows the model moving left because it deemed the only valuable parameter was to further the highest value in the position.

The second simulation moves down because it is trying to section off the highest value (512) to eventually push it into the top-left corner in an attempt to create a home row/column.

The last simulation moves right. A human could argue to move up or down as well, to combine the 512 blocks but the model knew that moving right would create a descending home column on the right side of the board.

States 2 and 3 are possible successful board states while state 1 will always end in a terminal locked state of no possible moves.

5 Literature Review

5.1 Image Classification

In anticipation of managing a large volume of images that would need to trained for image classification, the paper titled "Towards Good Practice in Large-Scale Learning for Image Classification" (IM paper) was the bedrock of this paper's implementation of image classification. Specifically, there were two key takeaways from the IM paper: its endorsement of stochastic training, specifically Stochastic Gradient Descent (SGD), and the classification strategy of one-vs-rest (OvR).

While the IM paper is geared towards experiments that involve millions of images being processed with a computer that has a highly respectable 16 CPU cores, relative to this paper's thousands of images and weaker computers, the recommendation to perform learning with SGD still applies perfectly. Moreover, the IM paper mentions how SGD "has recently gained popularity in image classification". This is important for this paper since as novices in image classification, a popular classifier means there are many resources to learn from and help with troubleshooting.

The OvR strategy splits a multi-class classification into one binary classification. In an early implementation of this paper's image classification, 2048 boards were one class among chess boards, go boards, and other random images. Requiring the classifier to learn all these classes not only cost

computation time, but also accuracy. Upon implementing the OvR strategy where the only two classes became 2048_boards and not_2048_boards, computation time decreased while accuracy reached the reported 100%. Finally, the nature of this paper's image classification omitted one of OvR's weaknesses. In a larger project, the OvR strategy requires a model to be created for each class. I.e., one model for 2048_boards and not_2048_boards, a model of chess_boards and not_chess_boards, and so on. However, since the scope of this paper only worries about 2048 boards, we did not have to concern ourselves with this OvR weakness.

5.2 Neural Network Design

The CNN paper was the backbone for focusing on solving the main problem of returning the best move by how to properly show examples and return normalized scores of the four moves considered by the hidden layer of the FNN. The paper found that increasing the number of hidden layers showed significant improvements in decisiveness for what move the model selected. This is something we would have added if time and increased hardware permitted. These tests would push to see how many layers should be added until results were negligible from the previous layer selection.

From the deep reinforced learning paper we learned that a basic MCTS or Minimax algorithm would yield successful results but only from new game states. It would perform poorly on randomly generated game states. The experiments they conducted with MCTS confirmed this hypothesis. From section 4.2 we gained knowledge of the four most important parameters for our model in having the largest number and a descending home row, column, and diagonal to a single corner of the board.

The ENN paper helped curtail the testing attributes needed for training the model and running the experiments. Table 1 recommended 5,000 generations, but we noticed no sizable performance increase when using only 2,500 states. They used a population size of 200 during their experiments which we upped to 250 to match a single round of training. The huge focus of this paper was to use mutations to find the best path towards the best board state. This is something we went against since it was not properly simulating games and failed while testing this functionality with random states. This paper proved an increase in success score as the complexity or the neural network was increased. If time permitted, we would have expanded our parameters to include states where like-tiles were touching, trapped smaller values were avoided, and where the total value of the board was the highest.

6 Conclusion

6.1 Problems

In regards to the 2048 game board image to game state functionality, a great deal of work went towards the implementation of complex functions utilizing OpenCV. Although it functions flawlessly, a solution where a 2048 board image is carefully sliced into 16 tiles and then evaluated would have taken far less time and produced an equal outcome.

Final problems occurred during training because of the weakness of our test machine. We could only train 250 states at a time and had to manually piece together and adjust the training process. It ended up succeeding but was made to be a tedious process.

6.2 Closing Thoughts

The problems solved from this project were complex and provided a great learning experience. This was a surprise when starting the project due to our perspective that the game 2048 is simple. However, the challenges in building our own version of 2048 so that the FFNN and SGD classifier can manipulate and train on data quickly humbled us. Not to mention the journey in learning how to correctly implement a FFNN, SGD classifier, and OpenCV. This was the first project we worked on that included image manipulation. Thus, nearly each step of the implementation process had mandatory troubleshooting and reevaluation. Creating a learning model to find the best possible move for 2048 took much more work than implementing an algorithm such as Maximax or Monte-Carlo. The results showed that the initial parameters we thought were equals turned out to heavily devalue achieving the highest value in place of developing a more structured board state.

7 Other

7.1 References

1. Code Bullet 2048 - www.youtube.com/watch?v=1g1HCYTX3Rg
2. 2048 Math - jdlm.info/articles/2017/08/05/markov-chain-2048.html
3. 2048 Algorithms - www.baeldung.com/cs/2048-algorithm
4. Convolutional Neural Networks - Found in Papers folder
5. Deep Reinforcement Learning - Found in Papers folder
6. Evolving Neural Networks - Found in Papers folder
7. Go Boards - https://tomasm.cz/imago
8. Chess Boards - https://osf.io/xf3ka/

7.2 Appendix

1. FEB 09th | Proposal Feedback | Need simple baseline implementations for image classification. Classify images as binary vector. Use FFNN to train and run model. Want good literature review.
2. MAR 13th | Progress Feedback | Use FFNN and not Minimax / MCTS.
3. MAR 29th | Inquiry | Advised to read in individual tiles and not whole board with OpenCV and contours.
4. APR 04th | Inquiry | Use binary classifier for image recognition. Train with as many images as possible.