Teaching a Neural Network to Play 2048 (+ cat)

Angela Qian
Certified PHcker .[?]
qian220@purdue.edu

Abstract

I decided to teach a neural network to play 2048, despite knowing very little about how to actually do that. This work represents a comprehensive summer-long case study employing the experimental methodology colloquially known as "fuck around and find out," formalized here as an iterative process of unstructured empirical exploration punctuated by intermittent bursts of ideas that appeared to me in my dreams. Despite training on a barely adequate dataset and a general disregard for best practices in machine learning, the project yielded a partially functional model that occasionally achieves non-embarrassing results. Future work will focus on replacing my uneducated, half-baked ideas with something vaguely resembling standard practice, as well as examining the effects of reading at least one relevant paper before implementation.

1 TL;DR: I made an AI that plays 2048

I've been mildly (okay, *extremely*) obsessed with 2048 since I was around ten years old. Funny tiles with big numbers itches my brain really good. Anyways, I'm studying computer science in college right now, and lately I've gotten interested in machine learning. So I figured, why not combine the two?

Figure 1: Author's artistic interpretation of this project[1] as a horse. Unclear if author has ever seen a horse.

2 Oops! I don't know what I'm doing

I got all hyped up about making a 2048 bot, but was quickly brought back to the crushing reality that: I'm a stupid little undergrad with a smooth little brain and do not know much about machine learning.

I started looking into some machine learning techniques, and the one that stood out to me most was imitation learning — basically, monkey see, monkey do. The reason? I honestly just wanted to avoid the headache of coming up with a way to "quantify" or "rank" how good a move is. With imitation learning, the model is given a bunch of state-action pairs (boards and their corresponding moves), and learns to predict the next action from a given state.

3 What would I do? Let's make the bot guess

Thanks to doing some undergrad research assistant stuff, I know that usually training these types of models requires *massive* amounts of data. As in, hundreds or even thousands of games. So of course, my first instinct is to search online for some pre-existing datasets I can use.

---

[1] This project can be found and played at https://angelazqian.github.io/2048-AI. Hopefully by the time you read this paper, my models will be slightly more competent than they were when I wrote this.

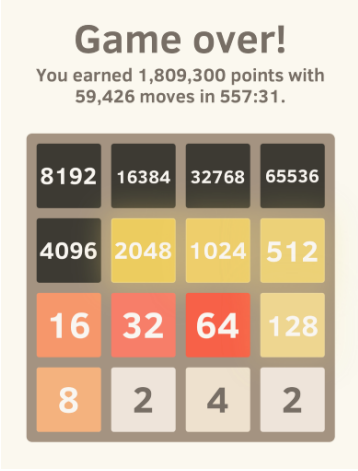

Figure 2: My highscore from last summer

I *did* manage to find some datasets of 2048 games online, but after digging into the stats, it turns out those players don't perform nearly as well as I do. For reference, Figure 2 shows my highscore from last summer.

Not only did their games tend to end much earlier than mine, looking through their gameplay, a lot of their moves were less than strategic. And as the saying goes — if you want something done right, do it yourself.

I ended up writing a small Python script that worked as a keylogger. When run, it opens 2048 in my browser, and for every move I make, the script saves the board state along with the move into a JSON file. This also allowed me to undo a move if I slipped up, so I didn't end up logging "bad moves" into the dataset. After collecting a staggering eight games (ok, not a lot lol but in my defense each game lasts anywhere from half an hour to four hours and i didn't have much free time since i was employed full-time over the summer), it's time to start training!

4 Forcing my model to see The Horrors

When looking into how to do imitation learning, the first method I came across was using Multilayer Perceptrons (MLPs).[2] Essentially, it's a type of neural network that processes multiple inputs, and returns a single output. This seemed like a good solution, as I could use the 16 grids of the game board as the input, then have it return the direction to move the tile in. I train it on

[2]MLP is also the acronym for "*My Little Pony*," which is rather fitting considering this is an entry in SIG**HORSE**. I thought this was hilarious so I have given my model a ponysona. See Figure 1.

the 8 games I have, load the model into the game and… yeah it sucks. Half the moves it was making were things I would never do. Back to the drawing board.

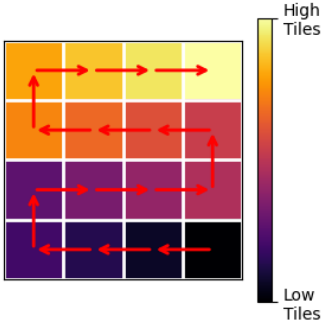

Figure 3: My preferred orientation

Maybe the problem lies with my data? When I play 2048, I tend to shove my big tiles in the top-right corner, favoring the upper edge, as shown in Figure 3. This would be reflected in the dataset, since all the collected games would follow the same pattern.

Since the model would have only learned to play well in the same orientation as me, when it loses that structure, it struggles to recover. That's also a problem if someone wanted to try playing the game themself, then swap in the bot mid-way — their tile alignment may be different than mine. Even natural gameplay sometimes shifts the board's orientation over time. I duplicated my data to represent all 8 orientations ($4 \times 90°$ rotations, then $2 \times$ for each mirror), as shown in Figure 4, and there is a *slight* improvement, but it's still comically bad.

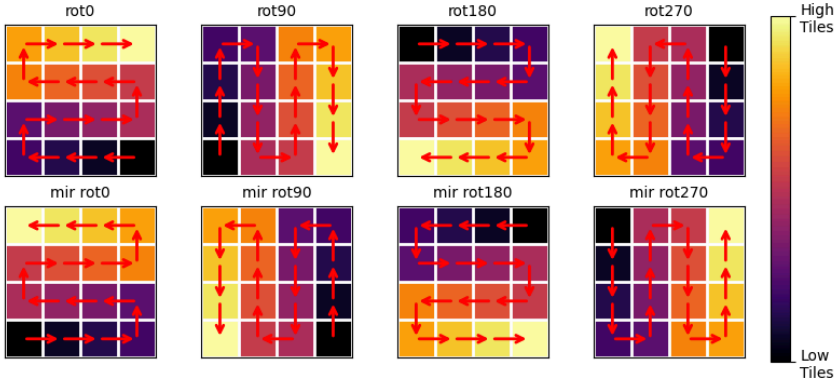

Figure 4: All 8 possible orientations of the board

After some thought, it occurred to me that treating each block in the grid as an independent parameter doesn't communicate any positional information, which is extremely important in 2048. To address this, instead of treating

the grid as 16 independent parameters, I started treating the grid as an image, essentially implementing a rudimentary form of computer vision. I accomplished this by switching from using MLPs to using a Convolutional Neural Network (CNN), which takes in a matrix input and uses convolutional layers to produce a single output. Much better results! But still not nearly as good as I had hoped.

5 Finetuning, but I am Woefully Uneducated

At this point I wanted to try finetuning my model, which means taking an existing trained model and continuing to train it so it becomes better adapted for the task. I looked into some common finetuning techniques, and the one that made the most sense to me was reinforcement learning through self-play, since 2048 is a single player game where you can objectively tell how good a game was through the final score.

In reinforcement learning, each full playthrough of the game is called an episode, and after each episode the model updates based on a reward function, which is basically a formula that tells the model what counts as "good". To encourage the model to be constantly improving instead of settling for an okay-ish score, I defined the reward function as the difference between the latest episode's score and the average score of past episodes.

And… it gets worse. What.[3]

I'll put this on the back burner for now and return to this later.

6 Making my model stop being Evil

The main issue about my model at this point is that it dies a lot early game, but if it somehow survives past a certain point, it starts performing well,

[3]In the writing of this journal entry, I found out that my mistake was baking a moving baseline into the reward function, which makes reward non-stationary. What I was previously using as the reward function was actually something called the advantage (essentially how much better an action was than what the model usually expects). However, that should be handled inside the learning algorithm, not included in the reward itself. What I should have used here was a Deep Q-Network (DQN), which is designed to estimate long-term value for actions in each game state and updates the model more reliably.

which I think is because of rotation noise. This is because early on in the game, the model seems to execute moves from various rotations, as if it can't decide which one to follow, leading me to remove the early gameplay from all rotations. I also noticed that when the board gains a large tile in a corner, it becomes ambiguous to the model as to which orientation it should follow. Considering that one of the main rule-of-thumb's when playing 2048 is that you should pick a direction, label it as "evil" and **avoid it at all costs**, this becomes a bit of a problem. For example, in Figure 5, if the largest tile is placed in the top-right corner, it *could* follow the orientation that favors the upper edge, designates `down` to be the "evil move", and mostly play moves `up`, `right`, and `left`. However, it could *also* follow the orientation that favors the right edge, designates `left` as the "evil move", and mostly play moves `right`, `down`, and `up`. Following this logic, all 4 directions may seem like reasonable moves to the model, which is *very bad*.

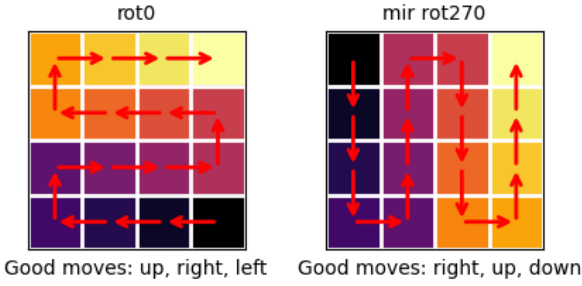

Figure 5: Ambiguity leads to all four directions being possible "good moves"

When playing 2048, it is good to choose one orientation and stick with it. However, sometimes a mistake happens, and you are forced to switch orientations in order to recover. The most common type of orientation change that happens mid-game is when you keep the same "evil move" and continue to favor the same edge, but switch to the other corner on the edge to keep the largest tiles. As an example, in Figure 6, the "evil move" continues to be `down` and the favored edge continues to be `up`, but the corner used to store the largest tile switches from the top right corner to being the top left.

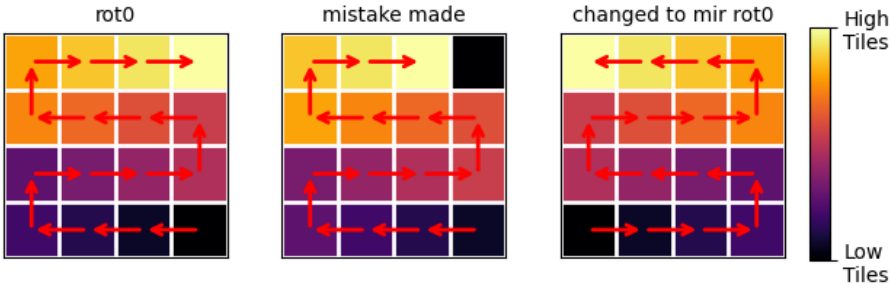

Figure 6: A common way of saving a game after a blunder is to switch orientations

To get rid of the orientation ambiguity for the model while still allowing it the flexibility to recover from blunders, I removed half of the rotations (the ones involving 90° and 270° rotations) from the training dataset. Just for good measure, I also removed all instances of when I was forced to do the "evil move" from all of the rotations. When I trained my model again on this new filtered data, I got much better results.

7 Cat

You're probably wondering where the cat comes in. There was "cat" in the title, you flipped to see what it was about, and instead got the deranged ramblings of some loser with an unhealthy obsession with 2048.

On July 18th around 10pm, my neighbor knocked on my door to tell me she heard what sounded like kitten meowing noises coming from my car. When I went to check, I could hear this little creature *wailing* from inside the car engine. I popped open the hood of my car, hoping to scoop him out, but the noise startled him, and he bolted into the surrounding bushes.

I regularly feed the neighborhood stray cats, and I know that all the strays in the area have been spayed or neutered, so the kitten had likely been separated from his mother. I couldn't bring myself to just leave him to the elements, so I sat on my porch with a bowl of Churu and unsalted chicken broth to try to lure him back out. He kept crying from the bushes and would occasionally dart

under other cars on the street, but he still wouldn't come near me. By 6am, I was cold, exhausted, and realizing this approach wasn't going to work.

The neighbor who first alerted me has experience trapping and rehabilitating stray cats. She's currently caring for an older cat and didn't want to risk exposing him to anything the kitten might carry, but she kindly lent me one of her humane cage traps. I put the Churu-broth bowl inside, set the trap, then went inside to rest for a bit.

When I checked a few hours later, the food was gone and I realized **he was too small to trigger the trap**. To fix that, I placed a 3 lb dumbbell on the pressure plate, refreshed the food, and waited. About an hour later — twenty-six hours after first hearing him — I finally caught him.

Figure 7: Tiny critter caught in a cartoonish cage trap

The next day, I brought him to a vet to make sure he was okay. They estimated he was about 8 weeks old and weighed only 1.59 pounds — on the low side for his age. He also had infections in both eyes and both ears, and was absolutely covered in fleas.

We started him on medication right away, and I kept him quarantined from my other cat while he recovered. After a few weeks of treatment, the vet gave us

the all-clear, and we finally introduced him to my resident cat, and thankfully, they hit it off. Have some pictures of them together.

Figure 8: Their first meeting!

Figure 10: They play with each other in a game almost like “cat and mouse.” It’s very entertaining to watch them chase each other at full speed.

Figure 9: My other cat started carrying him around the house by the scruff

Figure 11: The kitten has been imitating my big cat, including napping poses

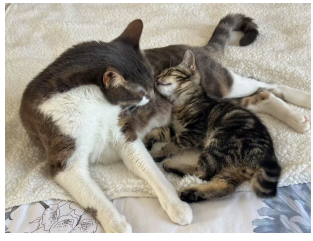

Figure 12: They like to sleep on each other all the time, using each other as pillows

Absolutely adorable. I love them both so much.

What was I doing before this again?

Oh. Right. 2048. Anyways, let’s get back to it!

8 The model got too locked in

At this point, I had managed to collect about 25 games for training — much more than the 8 I started with, but still a tiny dataset compared to what most machine learning models thrive on. Around then, I started to suspect that my model was overfitting — in other words, it was getting too good at memorizing the training data instead of actually generalizing to new games. This is not ideal, because it means the model performs well on board states it has already "seen" during training, but struggles or completely fails when faced with new situations. To combat this, I added a dropout layer, which is a layer that will randomly "turn off" some neurons so the model becomes more robust and is less dependent on specific neurons. There's a lot of improvement!

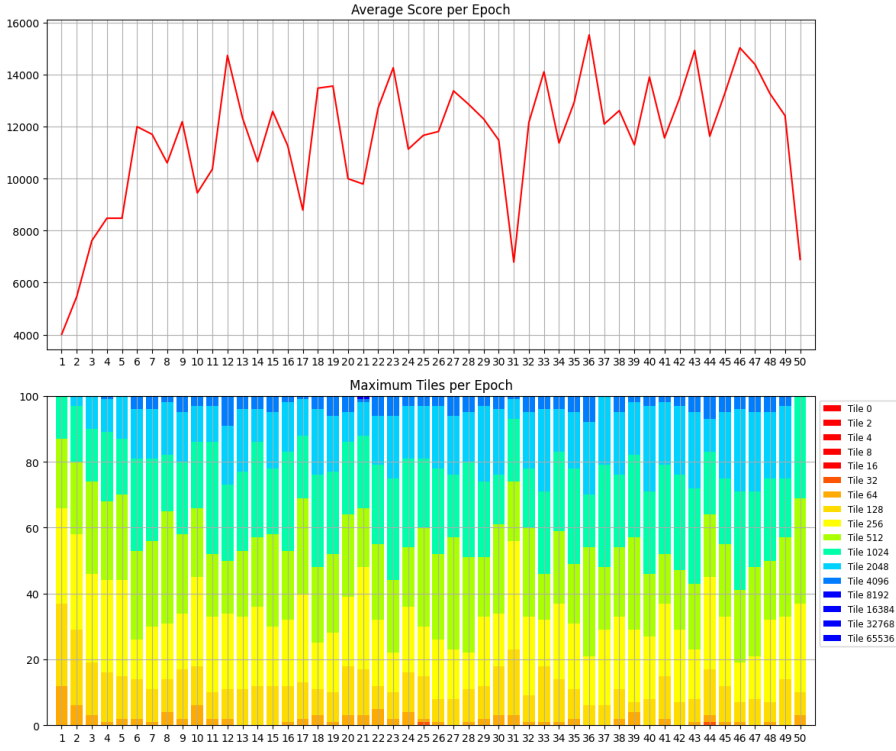

Figure 13: Performance of the model as epochs increase

However, I still felt that my model is overfitting. To monitor this, after every epoch (one full pass through the training dataset), I have the model play 100 games and record the average score as well as the distribution of the highest tiles reached. This way, I could track performance over time and identify when the model was actually improving versus just memorizing moves. At the end of training, I stored the weights from the epoch with the best average score, essentially picking the optimal epoch rather than blindly keeping the last one.

8.1 Brief interlude for the important graph that requires an entire subsection to explain properly

I've been told that the bottom graph in Figure 13 is a bit difficult to understand, so I'll try to break it down. Each bar shows the distribution of the highest tile reached during games played at that epoch. The proportion of a bar that's a given color corresponds to the proportion of games where the corresponding tile was the maximum. For example, at Epoch 36, around 7% of games ended with 128 as the max tile, while approximately 30% of games reached 2048 or higher.

Another way to read the graph is as a kind of "failure rate": the label on each bar tells you the percentage of games that failed to reach a certain tile. So for Epoch 36, the model fails to reach 512 about 21% of the time, and fails to reach 4096 92% of the time.

Here's the pseudocode for how the graph is generated, hopefully this helps if my explanation wasn't clear enough:

```
for each epoch:
    tile_distributions = {0, 0, 0,...}
    for each game in epoch:
        get max_tile created in game
        tile_distributions[max_tile] += 1
    for i in range(17):
        tile_value = 2^i
        show segment of length tile_distributions[tile_value],
            with color corresponding to tile_value
    show bar with these segments
```

8.2 Back to the main content, where I re-attempt finetuning

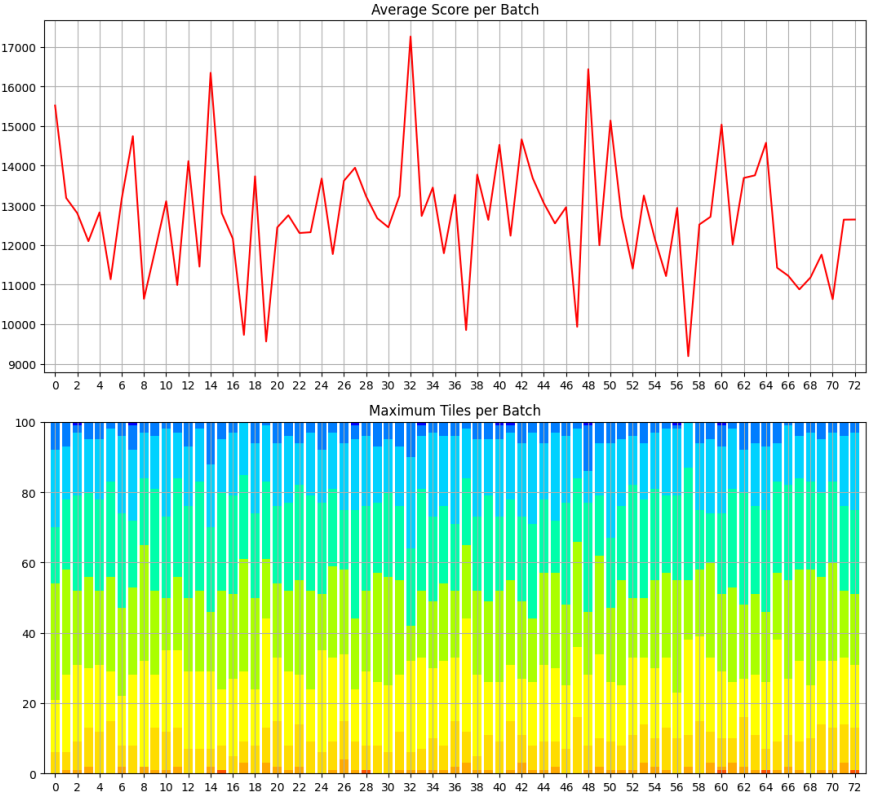

Figure 14: Performance of the model as batches increase

With the finetuning that I had attempted earlier, I noticed that the average scores reached by the model would dip a decent amount before they improved, then they would start worsening again.

Because of this, I essentially did the same thing, where after each batch (a small group of training samples, in this case 25 episodes, processed before updating the model's weights), I test it for 100 games, then track the average scores.

If no improvement is seen after 40 batches, I reload from the last best model, then continue from there. If it reloads too many times in a row (8), I stop and end the training.

Turns out, this actually does lead to some improvement!

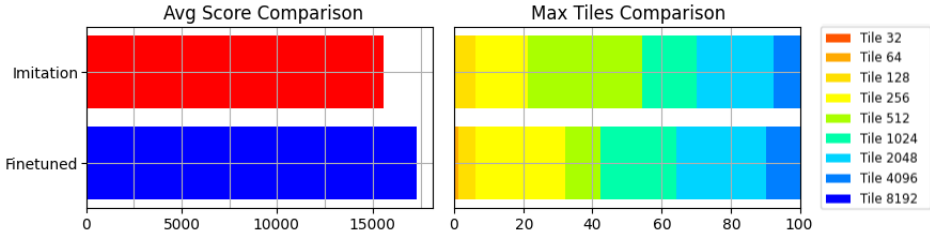

Figure 15: Comparison between pure imitation and finetuned imitation

9 The end?

My model is still far from perfect, and to be honest I'm still not completely satisfied with its performance. The finetuned version only reaches 2048 around 35% of the time, and since it was trained with imitation learning, it struggles to recover once it makes a mistake. It also does badly if you drop it mid-game where a human had been playing with a 90° or 270° rotation, since I excluded those orientations from my training data. Looking ahead, I'd also like to experiment with models trained entirely through reinforcement learning without any of my own gameplay data, and eventually implement a DQN[4] into my model.

But alas, summer has come to an end, and with it, the end of my free time — and the end of my journal entry. Those are projects for another day.

This is my first ever journal entry, and thus, I have no idea how to end this. Bye bye, thanks for reading, etc. The end!!!

[4] I explained what a DQN is in a previous footnote. Are you not reading my footnotes?! My feelings are hurt.