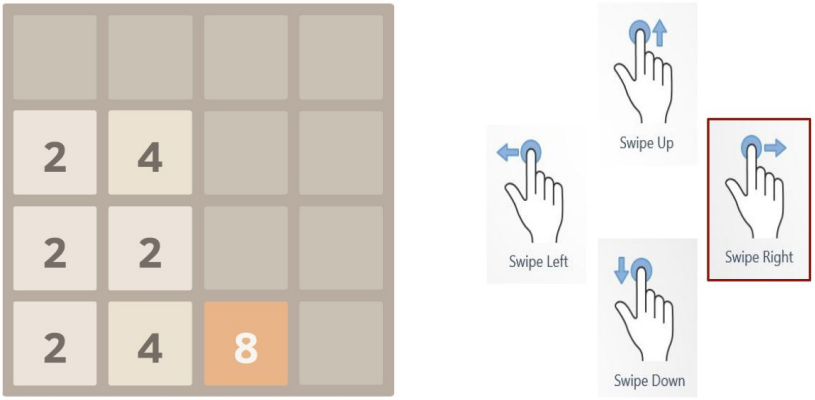
An RL Agent That Can Actually Play 2048

Jyoti Prakash Maheshwari
Prakhar Agrawal

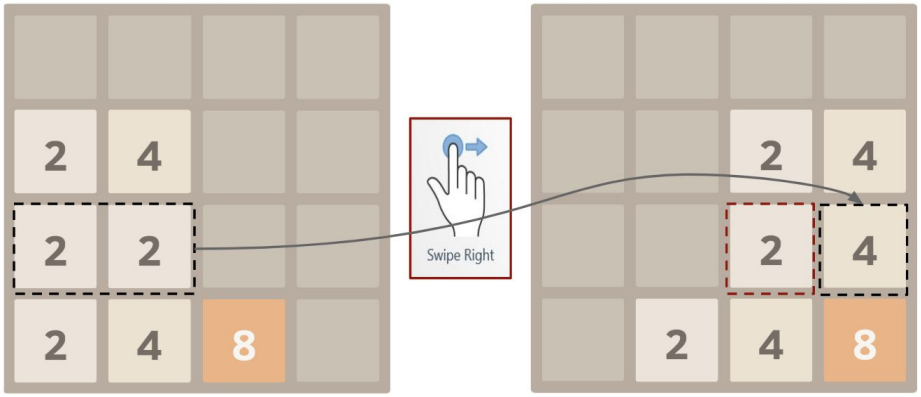
CONTENTS

1. HOW 2048 WORKS

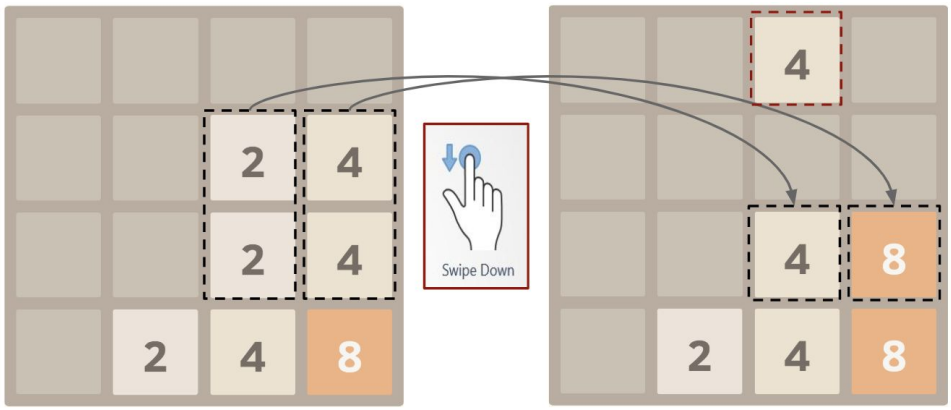
The game board is a 4*4 grid with all values being integer powers of 2. The user has a maximum of 4 available actions for each turn, i.e., swiping left, up, down, or right.



Let's say we choose to swipe right. What happens next is that all the tiles in the grid are shifted to their right, as much as is possible. Also, if in the course of doing that, a number collides with itself, the two merge by adding up. Lastly, you also see a new number show up at random, in one of the empty grid cells, the number always being either 2 or 4.



Similarly, if you were to swipe down right now, all the tiles in the grid will be shifted as far down as is possible in this grid. Merges will also be made, if applicable, and you'll also see a new number show up like earlier.



The game goes on until you run out of moves, or when the biggest tile on the board is 2048. A lot of games allow you to keep playing after you get 2048, and we'll also assume the same for the game our agent will be playing.

2. ENVIRONMENT SUMMARY

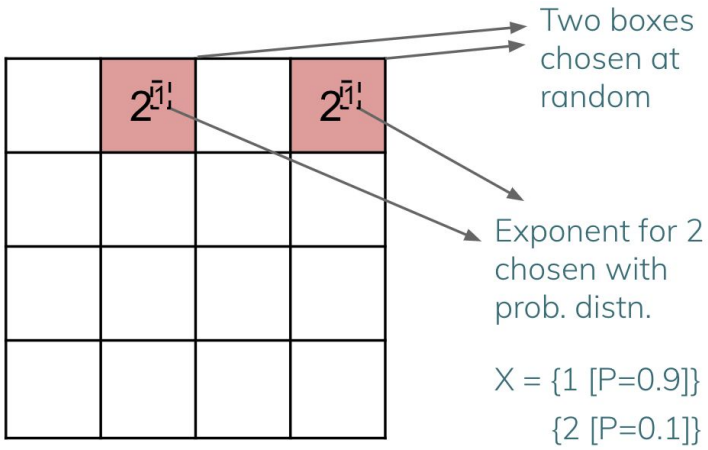
The RL environment for this problem was set up as following:

- ENVIRONMENT = Game Board
- AGENT = Trained RL Agent
- STATES = Numbers on Game Board
- ACTIONS = Left, Up, Right, Down
- REWARDS = Total Value of Merged Numbers

Let's delve deeper into how the environment was actually implemented in python.

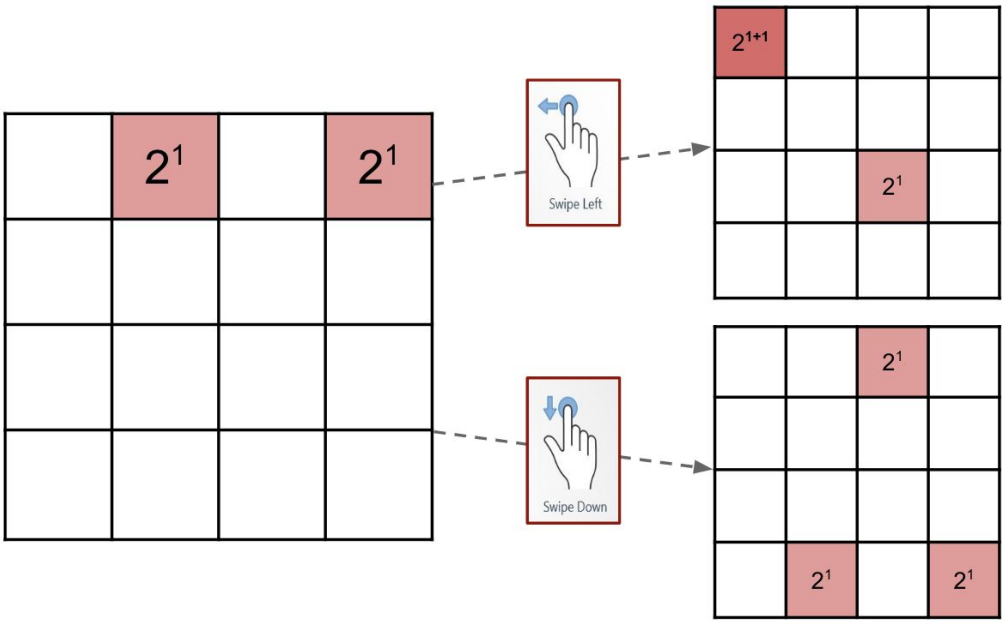
IMPLEMENTATION DETAILS:

Board Initialization



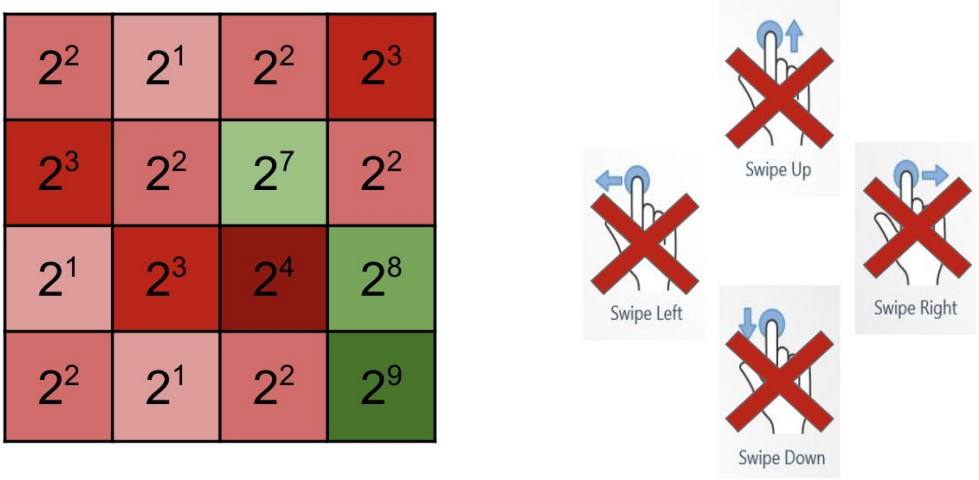
To initialize the board, we choose two tiles at random to fill. For these two tiles, we choose x to be 1 with prob. 0.9 and 2 with prob. 0.1, and then, 2^x becomes the value of these tiles.

Executing Actions



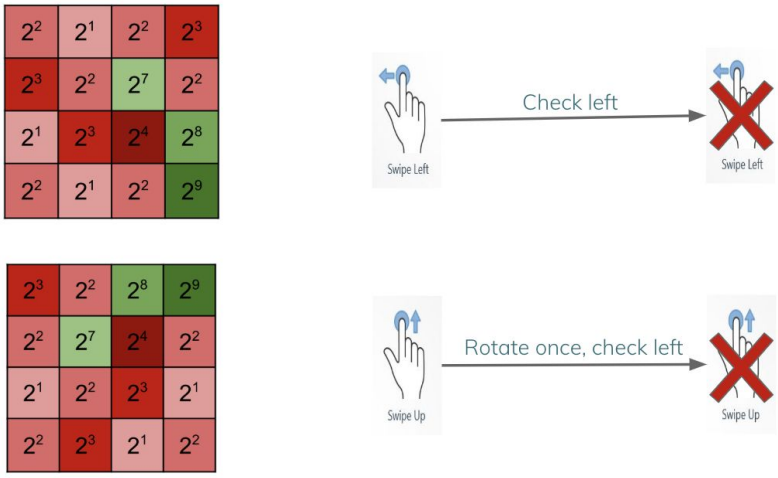
Next, depending on the user's actions, we update the value of x for each tile in the grid. In case of merges, we add x values, as has happened in the above case. Also, we will generate a random x value for one of the empty cells again, in a matter similar to board initialization, with the only difference being that this time, we only generate numbers for 1 tile instead of 2.

Detecting Game Termination



The game terminates when you reach a situation where you run out of available moves. For example, in the above case, you can't swipe left, up, down, or right. Hence, the game has terminated.

However, instead of checking for all four actions here, we've simply done the following:

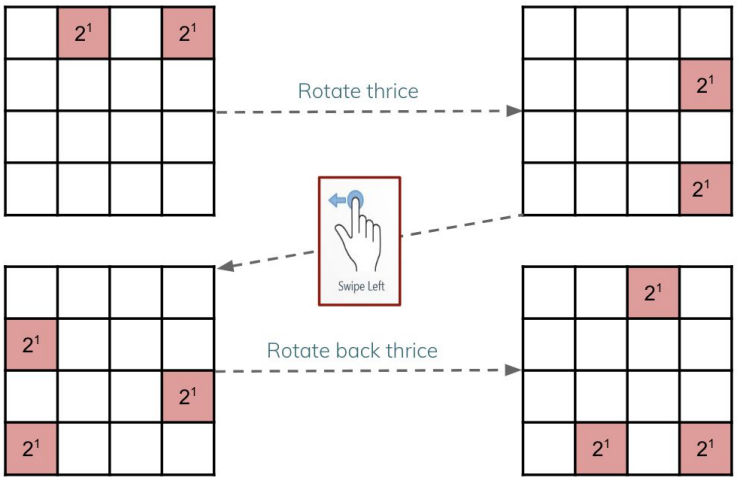


1. Check if the left action is available or not
2. Rotate the matrix once and then see if the left action is available or not, this checks whether the up action is available or not
3. Rotate once again and check for left, this checks for right swipe
4. Rotate one more time and check for left, which helps us check if the down swipe available or not

This helps us keep the code modular. Also, we only need to understand the logic for checking one type of action (left swipe), which makes the function easier to write.

Similarly, we can execute all actions using just the code for left swipe. For example, let's say we want to execute a swipe down. The steps would be:

1. Rotate thrice by 90 degrees
2. Execute left swipe
3. Rotate back thrice by 90 degrees



Using a similar logic, we can execute all possible actions.

3. LEARNING APPROACHES THAT SUCKED

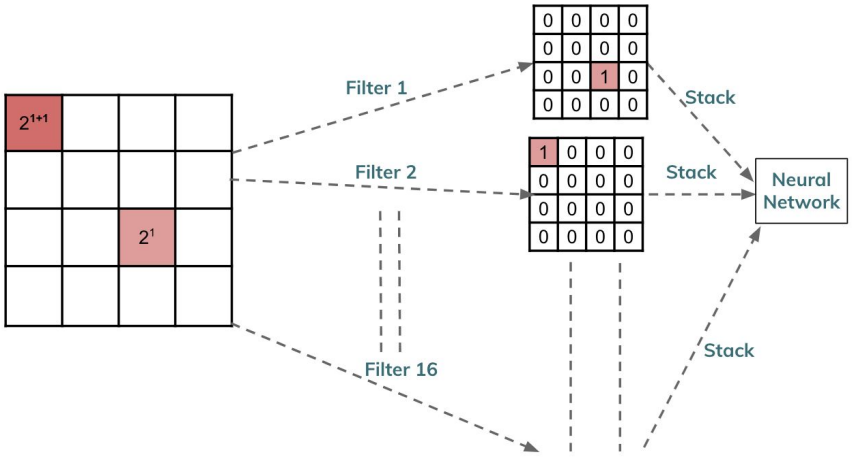
The following approaches (except for human level) were tried, but none of them look like they're actually doing well.

Approach	Mean Score	Max Score
Random	1093	2736
Q-Learning	1181	3324
DDQN	1205	3530
Human Level	14321	20214

We think Q-Learning is not doing well, because of the sheer number of state-action pairs in the environment. This hurts the ability of the model to learn a simple, generalisable policy.

Also, DDQN, even though it addresses the above problem, by using a value function approximation, doesn't do very well either. It suffers from myopia, in the sense that it knows the best action in the short term, but doesn't have a clue about its value in the long term. This being a strategy game, long term visibility is actually important.

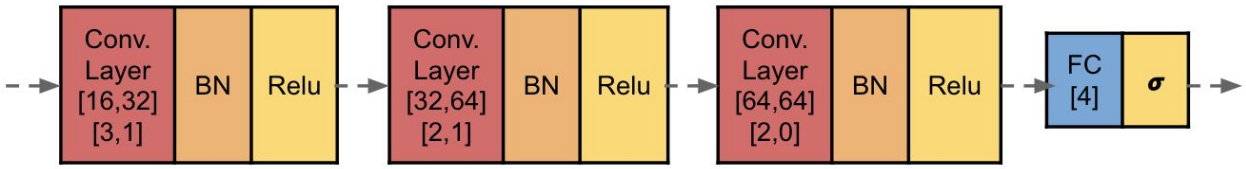
Nevertheless, let's discuss some of the strategies we used in our implementation of the DDQN:



Instead of just feeding a matrix of numbers blindly into the network, we process the matrix as shown above, before doing so. Basically, we use a series of filters that work this way:

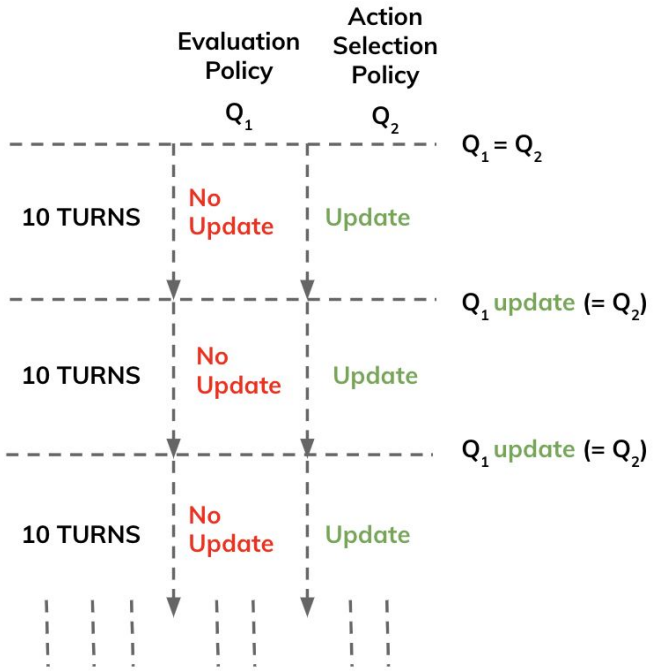
1. Filter 1 converts the cell to 1, if it is equal to 2^1, and converts it to 0 if it's not
2. Filter 2 converts the cell to 1, if it is equal to 2^2, and to 0 if it's not
3. And so on, till filter 16

We take all matrices that we get from these filters, stack them all up, and then feed this tensor to the following network:



This network, comprised of 3 convolutional layers, finally outputs 4 probabilities, one for each possible action in the game.

Also, for the double Q-learning part, we opted for a fixed-q learning strategy, instead of the random selection strategy taught in class. The way this strategy works is that we maintain an evaluation policy (Q_1) and an action selection policy (Q_2). The action selection policy is updated for say, 10 turns in a row, while the evaluation policy is kept fixed. After that, we update the evaluation policy, Q_1, based on what Q_2 is. We keep doing this for as many iterations as are necessary until a stopping condition is reached.

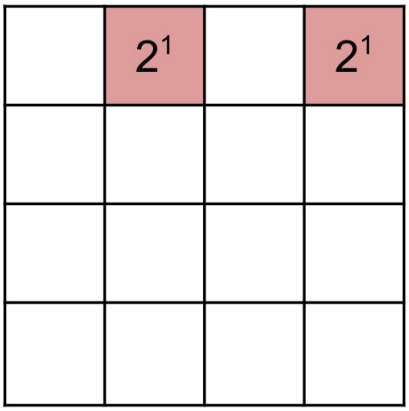


Even though all of these techniques might sound novel and effective, (at least I hope they do), the DDQN model didn't do very well. It suffers from all of the problems we mentioned earlier in this section.

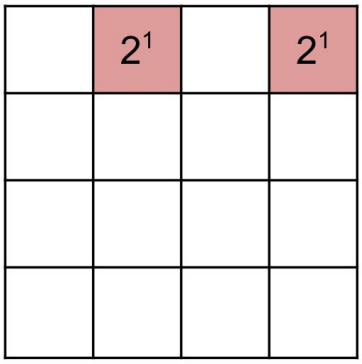
4. LEARNING APPROACH(ES) THAT DIDN'T (SUCK)

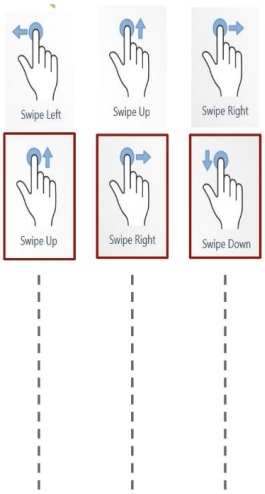
The technique that finally worked for us, might actually sound too simplistic - We used a Monte Carlo Tree Search. Here's how it works:

We check all possible moves and their associated value. Our next move would be the one with the highest value.



We could also opt for a more complicated approach - instead of checking just the next move, we can check the next, say, three moves. Basically, check all possible combinations of moves for the next three turns, and then select the next move based on their values.





This approach works very well, and it's really not hard to guess why - at every point of time, we decide our future actions based on long term strategy - that is bound to have higher payoffs.

To further illustrate and prove our point, we present the mean and max scores for each strategy again, this time with Monte Carlo Approach's performance included as well.

Approach	Mean Score	Max Score
Random	1093	2736
Q-Learning	1181	3324
DDQN	1205	3530
MC (1-step)	1811	6192
MC (2-step)	7648	16132
MC (3-step)	8609	16248
Human Level	14321	20214

While 1 step MC doesn't really do very well, our multi-step MC approaches are able to exceed human level performance.