Finding the Optimal Strategy of 2048 using Q-Learning

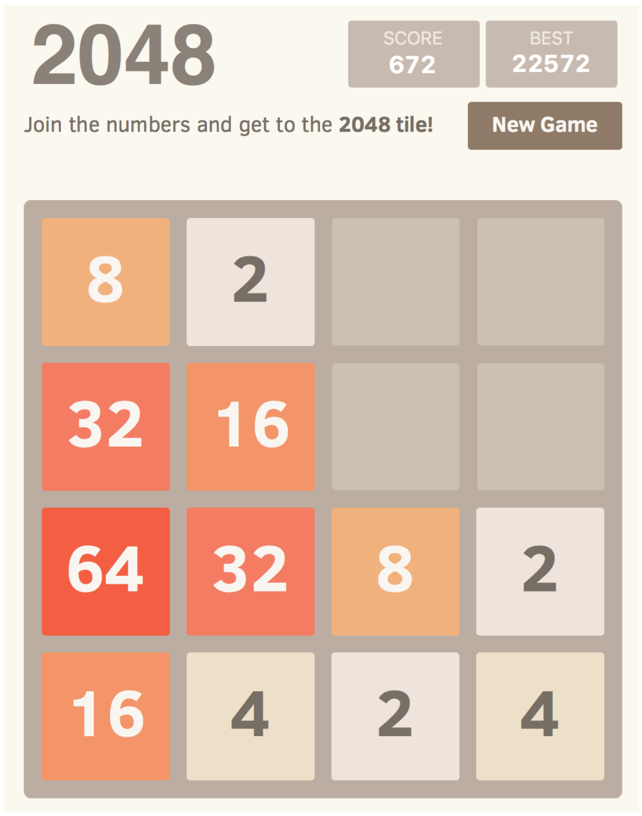

Figure 1: Possible state of 2048 Game

Using the Python packages pyautogui and PIL we investigate the game play of 2048. Gameplay is actuated by pressing one of the arrow keys which shifts the cells seen in Figure 1 in the given direction. An example of the procedure is seen below in Figure 2.

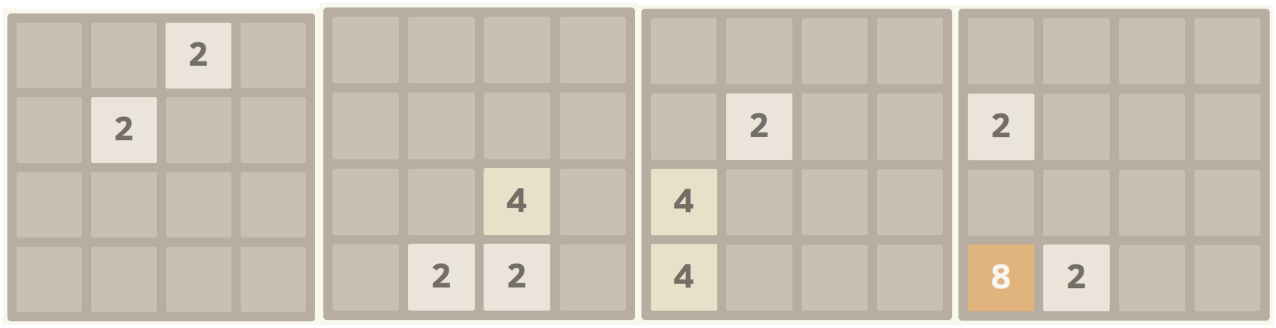

Figure 2: 2048 Game progression with moves "Down, Left, Down"

We extract the numbers by finding the pixel color of each square. This gives a 4x4 matrix as follows (corresponding to Figure 1):

$$\begin{bmatrix} 8 & 2 & 0 & 0 \\ 32 & 16 & 0 & 0 \\ 64 & 32 & 8 & 2 \\ 16 & 4 & 2 & 4 \end{bmatrix}$$

The difficulties that arise when attempting to write a learning algorithm for 2048 stem from the stochastic nature of the board. This means that we have a stochastic state space of indeterminate size. Some estimates have shown that it takes upwards of 10,000 moves to reach a tile of value 2048 or higher. This requires that we create a functional which takes any given state and action and returns its long term reward. From which point we could select the appropriate action. While this seems feasible it requires significant computational power as well as a complex interpretation of the state matrix regarding its geometry and the exact effect of an action. Which I realized takes much more time than I had available.

But, the ideal strategy of game play can be easily summarized into a Weight matrix. This weight matrix gives us the locations on the board where higher numbers should be as to maximize the long term reward. Throughout my research I found that almost every implementation uses some sort of weight matrix to place larger numbers into corresponding patterns, when possible that is.

After playing the game a few times one may notice that we prefer the largest number to be located in a corner, but what of the second largest and the third largest? The interesting result is that the weight matrix is not symmetric. An example of a weight matrix found by Yiyuan Lee is seen below.

$$\begin{bmatrix} 0.135759 & 0.121925 & 0.102812 & 0.099937 \\ 0.0997992 & 0.0888405 & 0.076711 & 0.0724143 \\ 0.060654 & 0.0562579 & 0.037116 & 0.0161889 \\ 0.0125498 & 0.00992495 & 0.00575871 & 0.00335193 \end{bmatrix}$$

Notice how the largest weight is found in the top right corner with progressively smaller weights found along columns. Let us now develop a Q-Leaning scheme to recover this form.

Instead of working with the entirely of the state space of 2048 we limit our algorithm to the first 13 moves. This allows us to compute the Q-Values with respect to the next state by setting our state to be the move number. This gives us a Q-Value matrix of 13 by 4 due to the four possible actions.

Using the most basic implementation of Q-Learning where we determine the values by a linear difference and a reward function. To accurately represent game play we define a specific reward function as a function of the state, action and weight matrix. At the start the weight matrix is normalized in the sense that every position has equal weight, namely $\frac{1}{16}$. The action space is discretized from strings of ('left', 'up', 'right', 'down') to $[[-1,0],[0,1],[1,0],[0,-1]]$. From this representation we can mathematically write our reward function as follows for each respective action. Where the $\circ$ operator is the Hadamard product of matrices or element wise multiplication.

'left'	'up'	'right'	'down'
$(W \circ S)\,W[:,0]$	$(W \circ S)\,W[0,:]$	$(W \circ S)\,W[:,3]$	$(W \circ S)\,W[3,:]$

Using this reward function we can run our code for multiple iterations and compute the respective Q-Values this tells us which moves are on average more likely to yield a higher reward. We take the maximum along the rows of the Q-Val matrix and transform it into a matrix of zeros and ones.

$$\begin{array}{ccc}
\text{Q-Values} & & \text{Moves} \\
\hline
\begin{bmatrix} 0.0267 & 0.0137 & 0.0222 & 0.0424 \\ 0.0358 & 0.05751 & 0.0145 & 0.0609 \\ \vdots & \vdots & \vdots & \vdots \\ 0.1398 & 0.1727 & 0.1412 & 0.1196 \\ 0.1116 & 0.1206 & 0.1437 & 0.1035 \end{bmatrix} & \Longrightarrow & \begin{bmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ \vdots & \vdots & \vdots & \vdots \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}
\end{array}$$

From this point we adjust the weight matrix according to which action was taken most often and least often. Taking the sum along columns we can find which action is the most profitable. We adjust the weight matrix by increasing the row or column corresponding to the given move. Meaning the top row if the the 'up' action was taken most often or the left most column if the 'left' was taken most often. We increase it by a factor of $\frac{1}{64}$ and the row/column nearest to it by a factor of $\frac{1}{128}$. For the least used action we subtract the same values as mentioned.

$$\begin{array}{ccc}
\text{Before} & & \text{After} \\
\hline
\begin{bmatrix} \frac{1}{16} & \frac{1}{16} & \frac{1}{16} & \frac{1}{16} \\ \frac{1}{16} & \frac{1}{16} & \frac{1}{16} & \frac{1}{16} \\ \frac{1}{16} & \frac{1}{16} & \frac{1}{16} & \frac{1}{16} \\ \frac{1}{16} & \frac{1}{16} & \frac{1}{16} & \frac{1}{16} \end{bmatrix} & \Longrightarrow & \begin{bmatrix} \frac{65}{1024} & \frac{65}{1024} & \frac{129}{2049} & \frac{1}{16} \\ \frac{129}{2048} & \frac{129}{2048} & \frac{1}{16} & \frac{127}{2049} \\ \frac{1}{16} & \frac{1}{16} & \frac{127}{2049} & \frac{63}{1024} \\ \frac{1}{16} & \frac{1}{16} & \frac{127}{2049} & \frac{63}{1024} \end{bmatrix}
\end{array}$$

The process described above maintains the total sum of the weights at 1, as such the result is slightly different from the weight matrix given above. But the geometric relationship of the weights is recovered.

$$\begin{bmatrix} 0.0879 & 0.0698 & 0.0380 & 0.0244 \\ 0.0933 & 0.0751 & 0.0434 & 0.0298 \\ 0.0996 & 0.0815 & 0.0498 & 0.0361 \\ 0.1005 & 0.0825 & 0.0507 & 0.0371 \end{bmatrix}$$

This asymmetric and non-intuitive result is what we interpret as the optimal strategy of the game 2048. Using this strategy and an efficient interpretation of the actual board state along with an optimization algorithm, Expectimax for example, I believe will lead to the maximal score possible, $32 \cdot 2048$.