# Game Agents For 2048

**Srinithi Ramesh**
Northeastern University, Boston, MA
ramesh.sr@husky.neu.edu

**Abstract**

2048 is a stochastic puzzle game released in the year 2014 by Gabriele Cirulli. The game has gained momentum soon after its release and is played even today, 4 years after its release. The motivation for the paper is to create agents for the game without incorporating any human expertise. Two variants of Temporal Difference Learning were employed to acquire action value (Q-Learning) and afterstate value (TD-Afterstate) for evaluating moves. The board was represented using an n-tuple network which was successful in similar games like Othello, Connect-4 and Tetris. The agents were trained on a 3 x 3 board for 200 games to identify the best training parameters and using these parameters, the agents were trained on a 4 x 4 board for 1,500 games. Since TD-Afterstate performed better, the agent was trained using it for 10,000 games. The agent scored 4356 before the board was filled but it has more room for development.

## Introduction

The game 2048 is a highly addictive single player game. The game is simple and consists of a 4 x 4 board with each square being empty or containing a tile of value n where n is in powers of two. The game starts with two numbered tiles placed initially at random locations. Figure 1 is one possible initial state of the game 2048. The player at each turn can move in four directions (Left, Right, up and Down). When two tiles of the same number slide over each other, they combine to form a new tile with value twice that of the merging tiles. Moving the tile up will cause the two tiles to merge and form a 4-tile at the top right square. Each move also introduces a tile at a random location on the board with a 0.9 probability of it being a 2 and 0.1 probability of it being a 4. The objective of the game is to continuously merge tiles to form the 2048 tile. The game can be continued even after obtaining the 2048 tile. The game ends when there are no more legal moves. A move is legal if it causes at least one tile to slide. In Figure 1, all moves except moving to the right are legal moves. The score is calculated by adding the value of the tiles merged. The game is symmetrical and the mathematics behind the game has been analyzed thoroughly (Goel 2017) which can be exploited by programmers to create an agent.

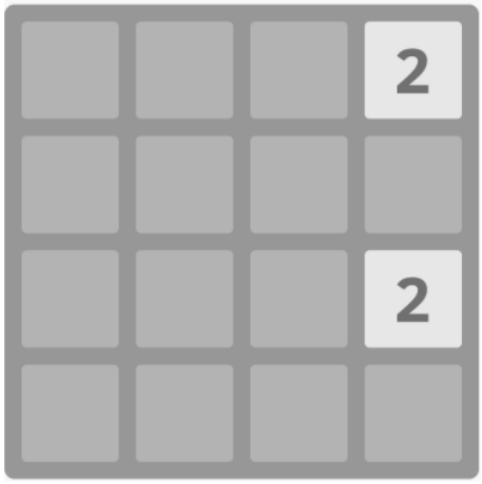


*Figure 1:Sample initial board for 2048 game*

Only the current state and the action taken in this state defines the next state of the board. This makes the idea of having the reward dependent only on the present state an ideal choice. As the history of the game has no impact on the future states, the transitions caused by the action will be Markovian. Due to these characteristics, treating the game as a Markov Decision Process and solving it using Reinforcement Learning will be the typical approach. In this paper, two variants of the Temporal Difference Learning algorithms are implemented, and their performance are compared. One variant calculates the action values (Q learning) while the other calculates the afterstate values (TD Afterstate). Since the state space is huge, a 4-tuple function approximator is used to reduce resource utilization. These topics will be explained in more detail in the later sections.

## Background

Temporal Difference Learning (TDL) is a type of model-free Reinforcement Learning which involves updating the estimates based on the rewards obtained by sampling the environment. An optimal policy will determine which action to take at each state to obtain maximum reward. The

TD algorithms have the following update rule to adjust the value function:

$$V(s) \leftarrow V(s) + \alpha\big(r + \mathrm{V}(\mathrm{s}'') - V(s)\big)$$

That is at each state, it tries to minimize the difference between the actual value, $r + \mathrm{V}(\mathrm{s}'')$, where $r$ is the actual reward and $V(s'')$ is the one-step ahead prediction, and the current value V(s). The effect of the update is determined by the learning rate, α. TDL methods compute a value function to facilitate decision making. In the context of game playing, the state value function is used to predict how many points the agent will obtain from a given state till the end of the game. The state value function is learnt from experiences with the environment. The agent takes an action, $a$, on the environment and observes the state transition $\mathrm{s} \rightarrow \mathrm{s}''$. It then updates the state value function based on the reward, $r$, it receives. Two variations of the TD(0) have been implemented: Q-Learning and TD-Afterstate.

The goal of Q-learning is to evaluate the state-action pairs to determine the action which gives the maximum reward from a given state. Q-learning is used to systematically update the utility of the action value pair using the formula:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left(r + \max_{\forall a' \in A(s'')} Q(s'',a') - Q(s,a)\right)$$

At each move, two transitions occur on the board. One is the sliding and merging of tiles (leading to the afterstate) and the other is the addition of a tile on the board at a random empty square. As shown in Figure 2, the initial game state, $s$, and an action, $a$, causes a transition to the afterstate, $s'$, receiving a reward, $r$. Addition of the random tile causes a transition from afterstate $s'$ to state $s''$. It can be observed that the first step is deterministic while the second is random. It is possible to exploit this knowledge of the game, to train an agent to update the afterstate values and decisions can be made based on the afterstate value function and this is the TD-Afterstate learning.

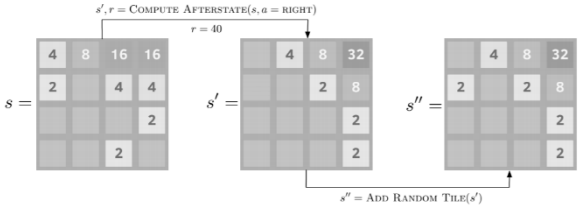


*Figure 2: Two step transition after taking action a=RIGHT. Image from Szubert and Jaskowski, Temporal Difference Learning of N-Tuple Networks for the Game 2048, IEEE Conference on Computational Intelligence and Games,2014*

This method is tailored for the 2048 game and updates the afterstate value function based on the error between successive afterstates. Therefore, for updating the afterstates, it calculates the next action, $a_{next}$, the action which the agent takes at the state, $s''$ causing a transition to the next afterstate, $s'_{next}$, to get the maximum reward possible, $r_{next}$. The update rule is given by,

$$V(s') \leftarrow V(s') + \alpha\big(r_{next} + \mathrm{V}(\mathrm{s}'_{\mathrm{next}}) - V(s')\big)$$

For any TD learning method, exploration is an important component. The agent has to explore even if it knows that the action is suboptimal based on the current value function. This is important since the exploration allows the agent to visit previously unvisited state and increases the probability of the agent to get an optimal solution. A simple exploration policy, epsilon-greedy, was implemented for this purpose. According to the policy, with a probability of epsilon, the agent will choose to explore other actions with equal probability including the optimal action.

Simple TD algorithms save the action value and the afterstate value in a look-up table. But for the 2048 game the size of such a table is very large. Each square on the board can hold 17 possible tiles (2 to 131072) (Goel 2017)**.** Hence each square can have 18 different values (17 possible tiles and empty square (No tile)). Hence the total number of states is $16^{18} = 4.7 \times 10^{21}$. It is computationally infeasible to have an explicit lookup table and hence some sort of function approximation is needed to capture the state of the board.

The success of n-tuple networks in other similar games like Connect-4 (Thill, Koch and Konen 2012) has inspired the use of them as a function approximator in the game 2048 (Szubert and Jaskowski 2014). The board is represented by a predetermined sequence of squares and each tuple is associated with a weight. The weights are updated after each move and a linear combination of the weights is used as the value function.

## Related Work

The most common approach to solve the 2048 puzzle game is the use of the Expectimax algorithm (Chowdhury and Dhamodaran 2014). The Expectimax algorithm is a variation of the minimax tree algorithm for agents which play games involving chance elements. The placement of the random tile in the board is the chance element in 2048. Expectimax algorithms have been used to create 2048 agents which can perform very well but the algorithm involves the construction of the Expectimax tree which contains all the possible states of the game. As seen in the previous section, the number of states in the game is huge and hence use of Expectimax algorithm is memory intensive. Moreover, the agent needs to be set with a lot of knowledge and heuristics about the game so that it can perform better. The agent doesn't learn from the inputs but rather walks through the tree to retrieve the optimal action. Monte Carlo Tree Search (MCTS) is another approach used

to train agents and like Expectimax is computationally expensive (Rodgers and Levine 2014). For optimization, both the trees need to be pruned and pruning causes the agent to perform poorly.

Since tree approaches were computationally expensive, some researchers performed reinforcement learning to train the agents. Many used deep reinforcement learning to get action policies but none of the approaches were able to create an agent capable of winning the game (Dedieu and Amar 2017). Agents which have reached even 1024 tile used some human knowledge like placing the largest tile in the corner.

Due to the success of n-tuple networks as a function approximation for the games Othello (Jaskowski 2014) and Connect-4, it was chosen to represent the state of the 2048 game. Temporal Difference Learning using N-tuple networks is a viable option to train an agent to play the 2048 game without specifying any human expertise.

## Project Description

For both TD-Afterstate and Q-Learning, a function approximator is required to store the weights. The function approximator implemented is an n-tuple network. The board is divided into a set of predefined tuple states and their values are stored in a lookup table. For the 2048 game a 4-tuple network was chosen i.e., four squares represent a tuple. Though it is not required, it is often that the tuples are chosen to hold sequential squares. The board was split into 17 tuples by grouping four horizontal 4-tuples, four vertical 4-tuples and 9 square 4-tuples as shown in Figure 3. The total number of weights stored by each lookup table is $18^4 = 104974$. For the 4-tuple network the total number of weights is given by 17 x 104974 = 1784592.

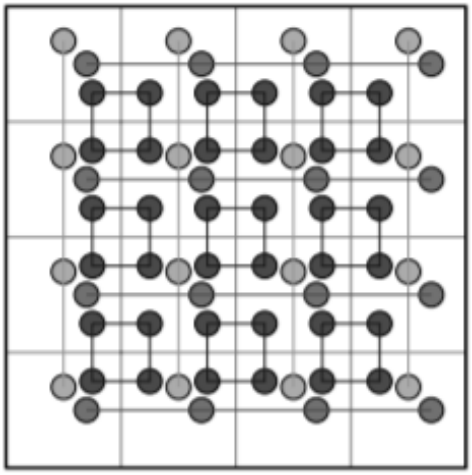

*Figure 3: 2048 board represented by seventeen 4-tuples. Image from Szubert and Jaskowski, Temporal Difference Learning of N-Tuple Networks for the Game 2048, IEEE Conference on Computational Intelligence and Games,2014*

To store the values in the lookup table, it is useful to note one property of the game. The tiles formed are always in powers of 2. The 2's exponent value of a tile is calculated and in case there are no tiles 0 is used, then this value is cross-referenced with the lookup table by using it as an index. For example, a tuple having tiles 2, 4, empty, 2 is represented by the indices 1, 2, 0, 1. Each of the 17 tuples are associated with a lookup table. The value function is represented as a linear function of the weights corresponding to each tuple where $LUT_i$ corresponds to the lookup table of the $i^{th}$ tuple and the $LUT_i[index]$ will return the value of the $i^{th}$ tuple for the given state. The value function at any state, $s$, is given by:

$$f(s) = \sum_{i=1}^{m} f_{i(s)} = \sum_{i=1}^{m} LUT_i \left[ index \left( s_{loc_{i_1}} \dots s_{loc_{i_{ni}}} \right) \right]$$

During learning, the weights are updated in the lookup table where the change in the $LUT$ weight is given by:

$$\nabla LUT_i \left[ index \left( s_{loc_{i1}} .. s_{loc_{i_{ni}}} \right) \right] = \alpha \left( f(s') - f(s) \right)$$

As stated in the previous sections, two algorithms are implemented to train the agents, Q-Learning and TD-Afterstate. Both the algorithms have the same general sequence given in figure 4. Only the evaluate and the learn function changes for the two approaches. A game will consist of the following steps:

```
1: begin
2:    score ← 0
3:    state ← initialize game
4:    ACTIONS ← {UP, DOWN, RIGHT, LEFT}
5:    while not gameOver(state) do
6:       nextAction ← epsilonGreedy(state)
7:       totalScore, s', s'' ← play (state, nextAction)
8:       reward = totalScore - score
9:       learn (state, nextAction, reward, s', s'')
10:      score ← score + reward
11:      state ← s''
12:   end while
13: end
```

*Figure 4: Pseudocode for general game 2048 game*

As stated in the previous sections, an epsilon greedy strategy is employed to discover unexplored states by taking suboptimal actions. Figure 5 represents the epsilon greedy policy. It chooses a suboptimal action ε times by randomly choosing a random action. For $(1 - \varepsilon)$ times, the agent chooses all the actions which have the maximum state value and selects randomly from that list. This is done so that when many actions have the same state value, the selection is random among them.

More formal details

```
1: begin
2:    greedyChoice ← random value between 0 and 1
3:    if greedyChoice < ε then
4:       nextAction ← getRandomAction(ACTIONS)
5:       return nextAction
6:    else
7:       i = 0
8:       for a in ACTION do
9:          actionValues[i] ← evaluate (state, a)
10:         i ← i + 1
11:      end for
12:      maxValue ← max(actionValues)
13:      maxActions ← action ∀ actionValues=maxValue
14:      return random(maxActions)
15:   end if
16: end
```

*Figure 5: Pseudocode for epsilon greedy policy*

The first algorithm discussed is Q-Learning. The lookup table is used to store the weights for each action state pair, meaning, four 4-tuple networks ($V_{UP}$, $V_{DOWN}$, $V_{LEFT}$, $V_{RIGHT}$) were used, one for each possible action. The evaluate function compares the values of each action independently. While learning the Q-Learning agent updates the weights in the corresponding lookup table. The evaluate and learn functions for the agent is:

```
1: function evaluate (state, action)
2:    return V_action(state)
```

```
1: function learn (state, action, reward, s', s'')
2:    V_next ← max_{a'∈A(s'')} V_a'(state'')
3:    error = reward + V_next − V_action(state)
4:    V_action(state) ← V_action(state) + α (error)
```

*Figure 6: Q-Learning evaluate and learn function*

Since a separate n-tuple network is maintained for each action, the evaluate function involves just looking up the value corresponding to the state. For the learn function, the maximum value from the final state, $V_{next}$, is calculated and then the error is calculated by adding $V_{next}$ to the reward and subtracting the result from the already existing action value. The action value is updated according to the equation in line 4 of learn function. The weights are updated as given initially in this section.

The second algorithm used to train the agent is TD-Afterstate. It exploits the deterministic step during each action and uses it determine the optimal policy. The learn and evaluate function is given in Figure 7.

```
1: function evaluate (state, action)
2:    s', r ← computeAfterstate(s, a)
3:    return r + V(s')
```

```
1: function learn (state, action, reward, s', s'')
2:    a_next ← max_{a'∈A(s'')} evaluate(s'', a')
3:    s'_next, r_next ← computeAfterstate(s'', a_next)
4:    V(s') ← V(s') + α(r_next + V(s'_next) − V(s'))
```

*Figure 7: TD-Afterstate evaluate and learn functions*

In the evaluate function, we find the afterstate and the reward by performing the given action on the current state. The value of the state is given by the sum of the reward observed and the value at the afterstate. During update, the learn function finds the action which produces the highest valued afterstate from the state ($s''$). This afterstate value and reward observed is used to update the value at the current afterstate using the formula in line 4 of learn function.

## Experiments

The training was conducted on a 3 x 3 board to fix the model parameters which are then used to train the agent to play on the 4 x 4 board. The tuples mentioned in the previous section was updated to match a 3 x 3 board: 3 horizontal tuples, 3 vertical tuples and 4 square tuples, 10 tuples in total. Since the maximum tile which can be obtained in the 3 x 3 board is 512, each square can have 10 possible values. Hence, the total number of weights is 10 x ($10^3$) = 10000.

The training on the 3 x 3 board consisted of letting the agent play 200 games to adjust the weights. The smaller board also allowed to test if the agent was learning correctly before training on a larger board. The agent was trained using both Q-Learning and TD-Afterstate and was tested for various learning rates and epsilon values and the best parameters were chosen based on the score obtained by the agent.

The results obtained by training the agents on a 3 x 3 board by the two algorithms are given below. The maximum scores obtained, which was used to select the parameters, are listed in Table 1 and 2.

| alpha (α) | epsilon (ε) | Max Score |
|---|---|---|
| 0.001 | 0.01 | 932 |
| 0.005 | 0.01 | 1324 |
| 0.01 | 0.01 | 856 |
| 0.005 | 0.01 | 1324 |
| 0.005 | 0.05 | 1256 |
| 0.005 | 0.001 | 1386 |

*Table 1: Q-Learning agent performance for varying α, ε*

| alpha (α) | epsilon (ε) | Max Score |
|---|---|---|
| 0.001 | 0.001 | 1124 |
| 0.005 | 0.001 | 1288 |
| 0.01 | 0.001 | 1496 |
| 0.01 | 0.001 | 1496 |
| 0.01 | 0.005 | 1524 |
| 0.01 | 0.01 | 1428 |

*Table 2: TD-Afterstate agent performance for varying α, ε*

Both the Q-learning agent and the TD-Afterstate agent was able to form the 64-Tile almost every time on the 3 x 3 board and formed the 128-Tile half the time. From the observations on the results from the 3 x 3 board, it can be noted that the Q-Learning agent performs best for a learning rate of 0.005 while the TD-Afterstate performs well for the learning rate of 0.01. The epsilon greedy parameter, ε, does not affect the performance of both the agents. The inherently random nature of the game makes exploration unnecessary since the game itself provides sufficiently diverse experience.

During training it was observed that TD-Afterstate performs much faster than Q-Learning and provided better results faster. This is because the TD-Afterstate exploits the deterministic afterstate to calculate the function values.

After fixing the game parameters, the agent was trained over 1500 games using Q-Learning. The agent was able to obtain a maximum score of 1592 and form the 256-Tile

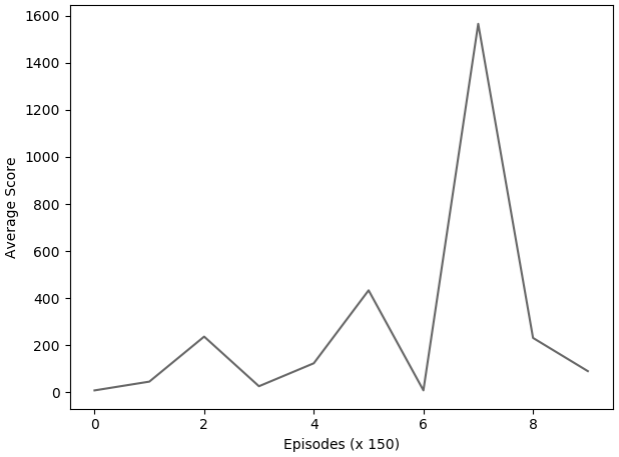


*Figure 8: Performance of Q-Learning over 1500 games*

occasionally and the 128-Tile almost every time. Figure 8 shows the graph by plotting the average score for every 150 episodes. As can be seen from the graph, the score of the agent keeps increasing as the agent plays more games.

The results from training the agent using TD-Afterstate with 1500 games is listed in Figure 9. It can be noted that the TD-Learning algorithm performs better than the Q-Learning agent and learns more quickly. The agent scores a maximum of 1854 before losing. It formed the 256-Tile more frequently.

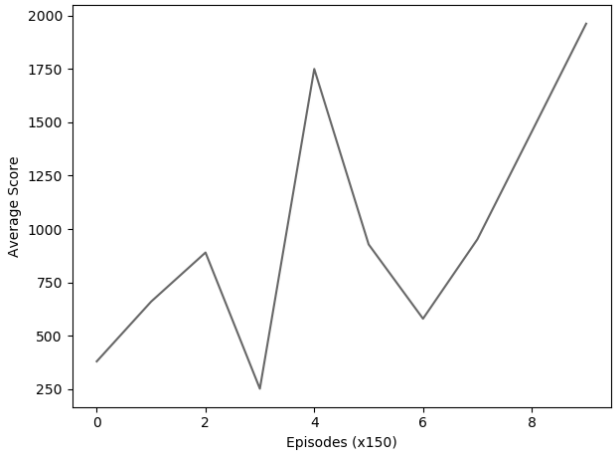


*Figure 9: Performance of TD-Afterstate over 1500 games*

Due to the better performance of the TD-Learning Algorithm, it was chosen to be trained for 10,000 games and the result is present in Figure 10. The agent formed the 256-Tile every time and formed the 512-Tile frequently. The agent scored a maximum of 4356 before losing.

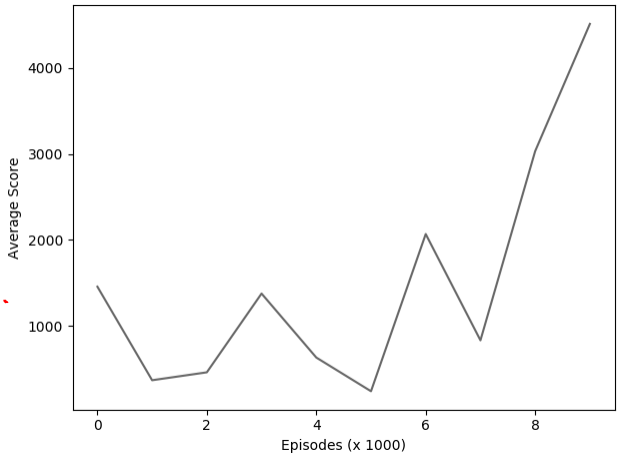


*Figure 10: Performance of TD-Afterstate in 10000 games*

The increased training episodes allow us to notice the learning curve of the agent. We can also observe that the score sometimes dips lower. This could be due to the random nature inherent in the game which does not allow the agents to converge over the small training episodes. For example, it is possible for the random tile to appear in an unusual location which causes the game to terminate in an otherwise favorable board.

## Conclusion

Using n-tuple TD-Afterstate algorithm, the agent was able to get 256-Tile every time it plays the game. Even though Q-Learning is not as much effective as the TD-Afterstate algorithm, it was observed that the agent was still able to obtain 256-Tile most of the times. However, considering the training time, the maximum score an agent can score and how consistent are the results, TD-Afterstate seems to the ideal choice for the game 2048. Although the results obtained in this project are much less than other similar agents, the TD-Afterstate agent showed promising improvement during training. The results obtained from this project illustrates that there still are a lot of unexplored states that are present and training the agents for more number of episodes will allow them to traverse deeper through the state-space. This will eventually improve the performance of the agents, thus allowing them to obtain higher order of tiles.

## References


Goel, B. 2017. Mathematical Analysis of 2048, The Game. *Advances in Applied Mathematical Analysis, ISSN 0973-5313 Volume 12, Number 1 (2017), pp. 1-7, Research India Publications.*

Szubert, M., Jaskowski, W. 2014. Temporal Difference Learning of N-Tuple Networks for the Game 2048. *IEEE Conference on Computational Intelligence and Games, Dortmund, Germany.*

Thill, M.; Koch, P.; and Konen, W. 2012. Reinforcement Learning with N-Tuples on the Game Connect-4. *Department of Computer Science, Cologne University of Applied Sciences, Germany.*

Chowdhury, G., Dhamodaran, V. 2014. 2048 Using Expectimax. *Department of Computer Science, University of Massachusetts Lowell.*

Rodgers, P., Levine, J. 2014. An investigation into 2048 AI strategies. *IEEE Conference on Computational Intelligence and Games, Dortmund, Germany.*

Dedieu, A., Amar, J. 2017. Deep Reinforcement Learning for 2048. *31st conference on Neural Information Processing Systems, Long Beach, CA.*

Jaskowski, W. 2014. Systematic N-Tuple Networks for Position Evaluation: Exceeding 90% in the Othello League. *ICGA Journal 37(2), pp. 85-96.*