Final Assignment - 2048 Game

Reinforcement Learning COMPSCI 9170B

Submitted By:
Aesha Alpeshkumar Gabhawala
251203021

Table of Contents

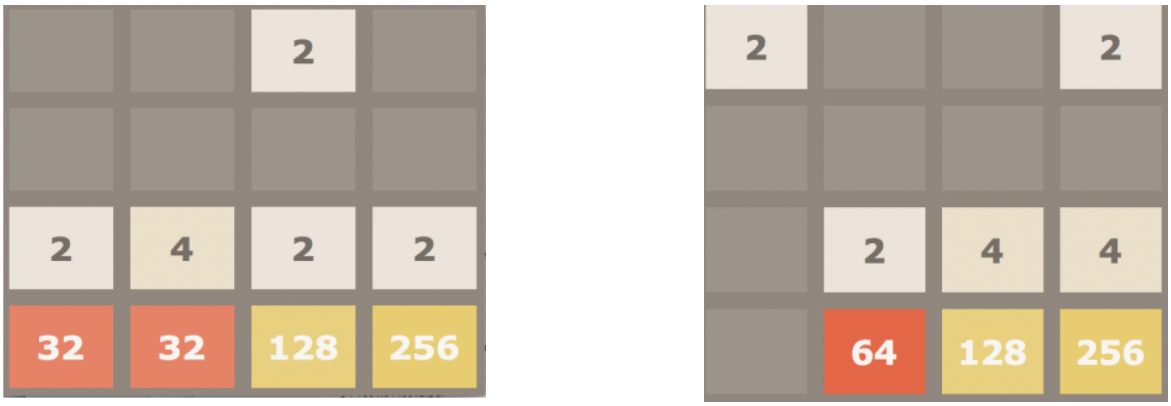
1. Introduction

1.1 Reinforcement Learning

Reinforcement Learning (RL) is a type of learning mechanism in which an agent learns by completing tasks that are rewarded. It's employed in situations where there's no single right way to tackle a problem (Kaundinya, et al., 2018). Reinforcement learning has been frequently used in recent years to play a variety of strategy games containing uncertainty, such as backgammon, card games, and go. In 2016, AlphaGo consistently defeated world champion Jie Ke in the game of Go, proving that reinforcement learning may achieve superhuman competence in strategy games without prior human experience (Li & Peng, 2021). This latest achievement is seen as a significant advancement in artificial intelligence because the system must explore over a vast state space before making a decision (Amar & Dedieu, 2017).

1.2 2048 Game

I chose to work on 2048, a single-player sliding block puzzle game played on a 4x4 grid with each cell a power of 2, which was released in March 2014. The goal of the game is to get a score of 2048 by shifting the grid in any of the four directions (up, down, right, left) and merging two consecutive cells with the same value (Amar & Dedieu, 2017). After attaining the goal, though, the game can be continued by generating tiles with greater numbers. The goal of this project is to create a self-learning agent that can play the game and attain the 2048 goal using Reinforcement Learning. Because of its simplicity and ease of testing Reinforcement Learning agents, 2048 has been a playground for AI researchers since its release. Despite the fact that several studies have been conducted on this game, they all differ in terms of the RL model utilized and the reward function used to train the agent (Anonymous, 2020)



a) Before the move b) After the move

Fig. 1: Illustration of the dynamics of 2048. The grid is shifted right, and a 2 randomly appears on the grid

1.3 Literature Survey

Amar & Dedieu (2017) use a policy network to learn the best strategy for playing 2048. The policy network is a two-layer convolutional neural network using ReLU as the activation function. The model takes the current state of the game as input and outputs a policy vector of length 4, with each item representing the likelihood of picking a policy that will result in the highest score at the current state. The study also utilizes ε-greedy as a decreasing function of the number of games played to stimulate the investigation of new strategies. The model, on the other hand, does not help win the game: it frequently reaches 1024 and occasionally reaches 2048.

Szubert, et al. (2014) used temporal difference learning (TDL) to learn the best policy by modelling the 2048 game as a Markov decision process (MDP). The model outperforms humans, reaching 2048 at a rate of 97%. However, the model struggles to reach larger tiles (never reaching 32768 or more), thus Yeh, et al. 2015 improves the model by combining multi-stage temporal difference learning with 3-ply expectimax search, and the new model achieves 32768 31.75%of the time.

Google Deepmind is the most well-known player in this industry. They employed deep Q learning techniques to create an agent capable of playing seven Atari 2600 games. In three games, their agent was able to outperform human experts. Their project was notable for being able to train an agent despite having extremely high dimensional input (pixels) and no knowledge of the game. (Mnih, et al.)

1.4 Background

In this project, I chose to implement Deep Q-Learning Networks (DQN) and study the influence of reward function choices on agent performance.

MDP:

Markov decision processes (MDPs) provide a framework for modelling problems in which an agent must conduct a series of actions in a given environment. Actions alter the environment's state and often have both immediate and delayed consequences that can be measured as rewards for the agent.

Formally, an MDP is a discrete time stochastic control process, and can be defined as <S, A, R, P>, where:

• S is a set of possible states of the environment,

• A is a set of actions and $A(s) \subseteq A$ denotes a set of actions available in state $s \in S$,

• $R : S \times A \rightarrow R$ is a reward function, where R(s, a) provides the reward for making action a in state s,

• $P : S \times A \times S \rightarrow [0, 1]$ is a stochastic transition function, where P(s, a, s'') denotes the probability of transition to state s'' in the result of taking action a in state s.

The goal of an agent placed in an MDP environment is to learn a decision-making policy $\pi : S \rightarrow A$ that maximizes the predicted cumulative reward. The game 2048 is naturally expressed as an MDP, with states representing board locations and actions representing valid movements. The game rules clearly define the reward function as well as the transition function. The random tile generation technique, which calculates the probabilities P(s, a, s"), makes the transition function nondeterministic. (Szubert, et al., 2014)

Q Learning

The state value function $V^{\pi} : S \rightarrow R$ estimates the expected cumulative reward that will be received by the agent if it uses policy π to make actions starting from a given state $s \in S$. In the context of playing games, the state value function predicts how many points the agent will get from the given state till the end of the game. The agent leverages experience from interactions with the environment to learn the state value function. (Szubert, et al., 2014)

In Q-learning, the agent learns the action value function $Q\pi : S \times A \rightarrow R$, which estimates the utility of a given state-action pair, i.e., the expected sum of future rewards after making action $a \in A(s)$ in state $s \in S$. Q-learning is an off-policy algorithm as the optimal policy is (max Q(s,a)) and the behavioral policy is explorative. Upon observing a transition (s, a, r, s''), it updates the action value estimates as follows:

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

where $\alpha \in [0, 1]$ is the learning rate which determines to what extent newly acquired information overrides old information. (Szubert, et al., 2014)

Deep Q-Networks

The game 2048 has about $(4 \times 4)^{18} \approx 4.7 \times 10^{21}$ states because the maximum value of a single tile[3] is $2^{17} = 131072$. Therefore, it is computationally impossible to implement an explicit value table. To overcome this problem, in deep Q-learning, the Q-values can be approximated using machine learning models like a neural network, which is how Google DeepMind's algorithm works (Szubert, et al., 2014). The state is given as an input, and the output is the Q-value of all potential actions. Fig 2 illustrates the difference between Q-learning and deep Q-learning.

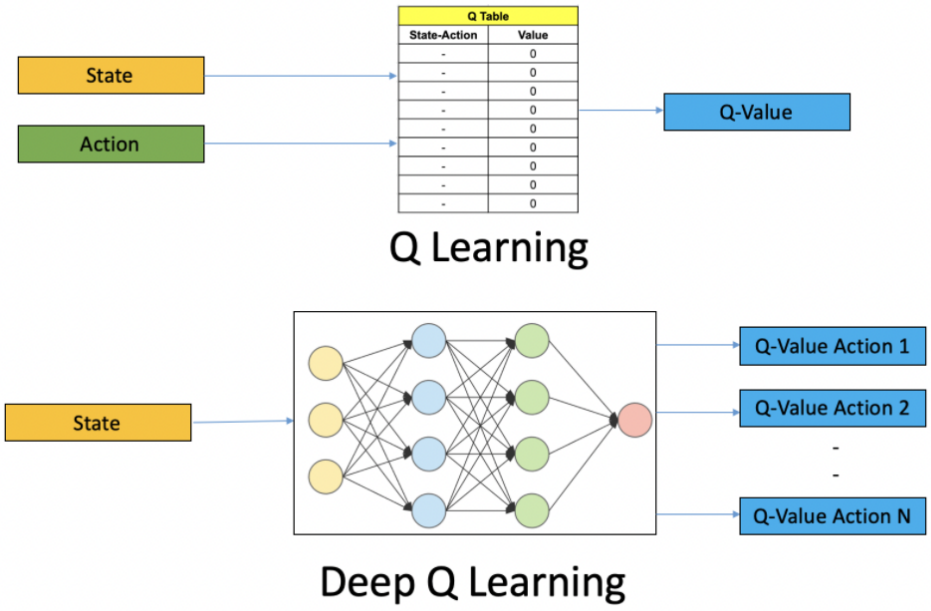


Fig. 2: Comparison between Q-learning and Deep Q-learning
(Choudhary, 2019)

In deep Q-learning networks (DQNs), all the past experience is stored in memory and the next action is determined by the maximum output of the Q-network. The loss function here is mean squared error of the predicted Q-value and the target Q-value – Q*. This is basically a regression problem but we do not know the target or actual value here as we are dealing with a reinforcement learning problem. The section in green in the Q-value update equation derived from the Bellman equation given below represents the target. As R is the unbiased true reward, the network will update its gradient using backpropagation and converge.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

(Choudhary, 2019)

2. Methodology

2.1 The Environment

I have trained my model on a customized version of the open-source-gym-2048 environment which is based on the following:

• **State space**: All observations are 4 x 4 numpy arrays representing the grid. The array is 0 for empty locations and numbered 2, 4, 8, ... wherever the tiles are placed.

• **Action space**: There are four actions defined by integers (LEFT = 0, UP = 1, RIGHT = 2, DOWN = 3)

• **Reward**: By default, reward is the total score achieved by combining all possible tiles for a specific action. The score earned by combining two tiles is just the sum of their values. Other custom rewards, which are shown below, were developed by overriding the methods of the original environment in an inherited class.

It is a simple environment (stochastic and completely observed) that's ideal for a DQN agent. The basic goal is to create a network architecture that has a reward mechanism that produces the best results.

I evaluated the agent's performance based on the score curves and the distribution of the max tiles using three distinct reward functions based on 1500 training episodes.

• **Reward 1- Score**: This function is the basic reward function given by the environment. The total score for a specific action acquired by merging all possible tiles. The score earned by combining two tiles is just the sum of their values. The merging of high-value tiles is encouraged by this reward.

• **Reward 2 - Total merges and maximum tile**: I separated the two objectives in this reward function to encourage the development of high-value tiles and the merging of tiles that aren't necessarily high-value. It is defined as the number of merges multiplied by the base 2 logarithm of the maximum value tile for a given state transition. This metric promotes not just the merging of two tiles, but also longer-term methods for generating additional powers of two, which is also near to the game's purpose.

• **Reward 3 - Number of empty tiles**: This reward function works on the idea that the more empty squares in the grid, the further one can progress in the game.

(Anonymous, 2020)

2.2 The Agent

I have experimented with architecture based on a Deep Q-Network with convolutional layers for this game. A vanilla DQN architecture with replay memory (technique used to produce uncorrelated batches of input data) and convolution layers was used as illustrated in Fig 3:

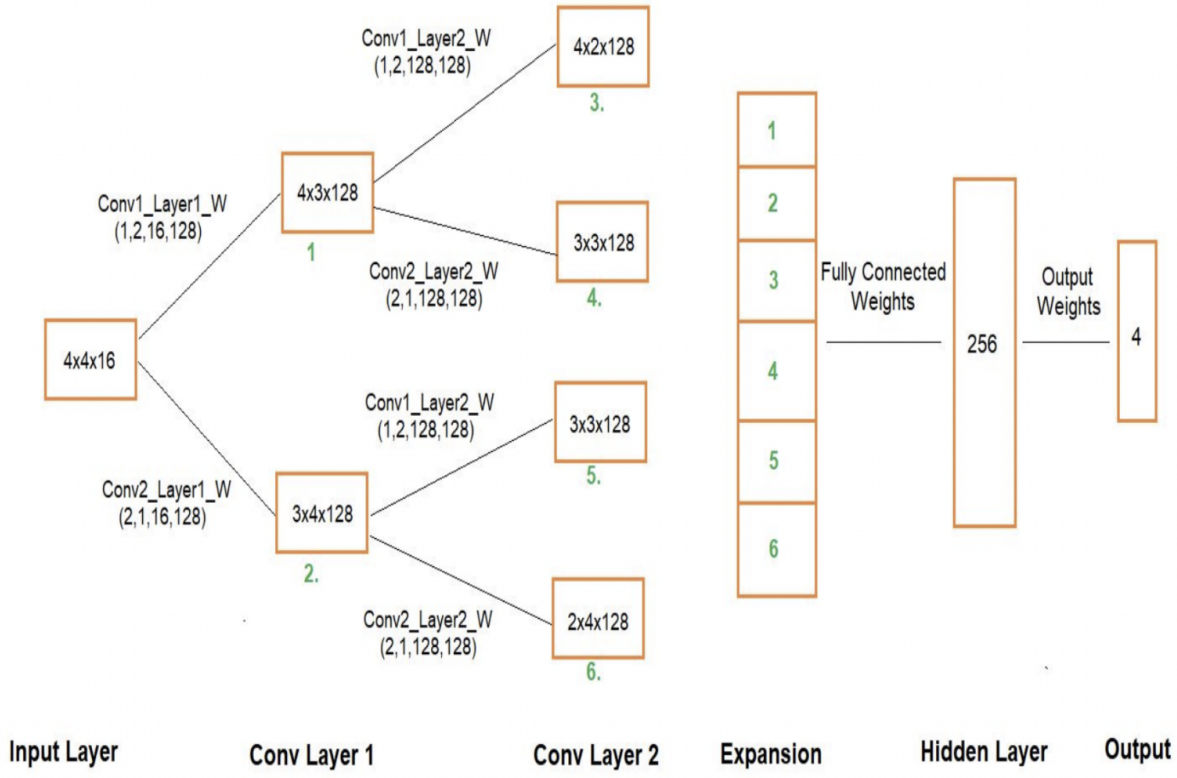


Fig 3: Architecture of the Deep Q-Network

DQN with MSE loss works by minimizing the following loss function:

$$L_i = \mathbf{E}_{s,a,s',r}(r + \gamma.max_{a'}Q(s', a') - Q(s, a))^2$$

which computes the target Q one-step ahead and compares it to the actual one. This model works best as it understands the grid and builds the target policy using 2D grid features. It also handles the exploration-exploitation trade-off well which is important in this game given that exploration is critical in 2048. Double DQN, an upgrade on the vanilla DQN, was used which entails storing two copies of the network, one for greedy action selection (Online network) and the other for value estimation (Target network). It improves the performances by lowering the bias over Q value estimations.

An ε-greedy policy with decaying ε is used. Low values of ε are employed as random moves can have a significant impact on 2048 game performance.

The parameters used in the training of this DQN with an ADAM optimizer are given in Table 1.

Parameter	Value
Learning rate α	10^{-4}
γ	0.99
Initial ε	0.1
Final ε	10^{-4}
Exploration step	10^{-4}
Replay memory storage	Last 50000 moves
Batch length	16

Table 1: Values of the parameters of the DQN

3. Experimental Results

3.1 Training

Fig 4, Fig 5 and Fig 6 represent the moving averages of the three rewards functions - score, total merges and maximum tile, and number of empty tiles respectively, with a window size of 25.

Fig 4: Averaged reward 1 (Score) w.r.t the number of games

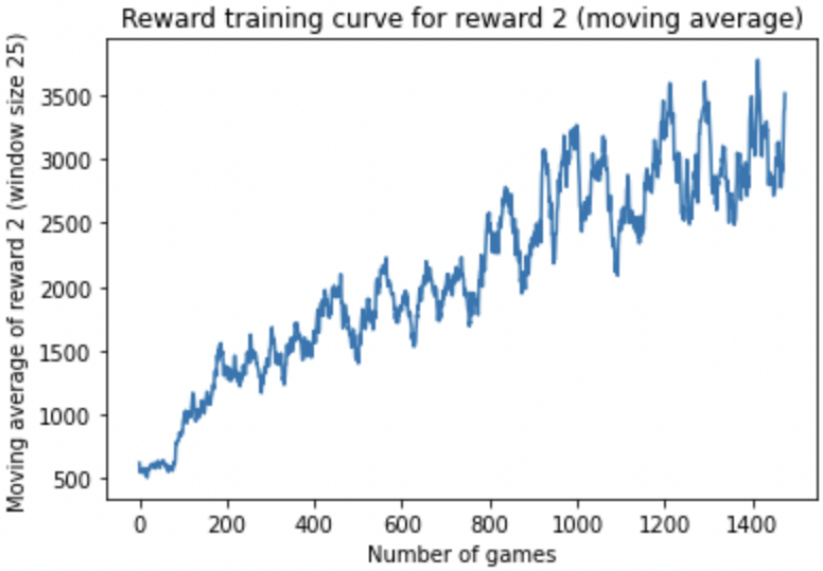


Fig 5: Averaged reward 2 (total merges and maximum tile) w.r.t the number of games

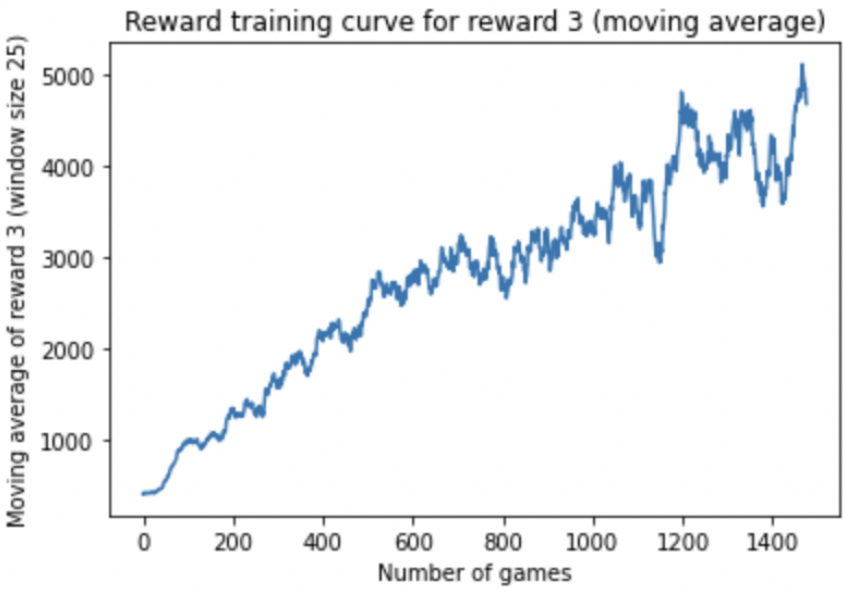


Fig 6: Averaged reward 3 (number of empty tiles) w.r.t the number of games

As evident from the graphs, reward 3 increases a little quickly at the beginning and reaches a constant value with less fluctuations than the other rewards. Fig 7 represents the moving averages (window size of 25) of maximum tile reached at each game during the training.

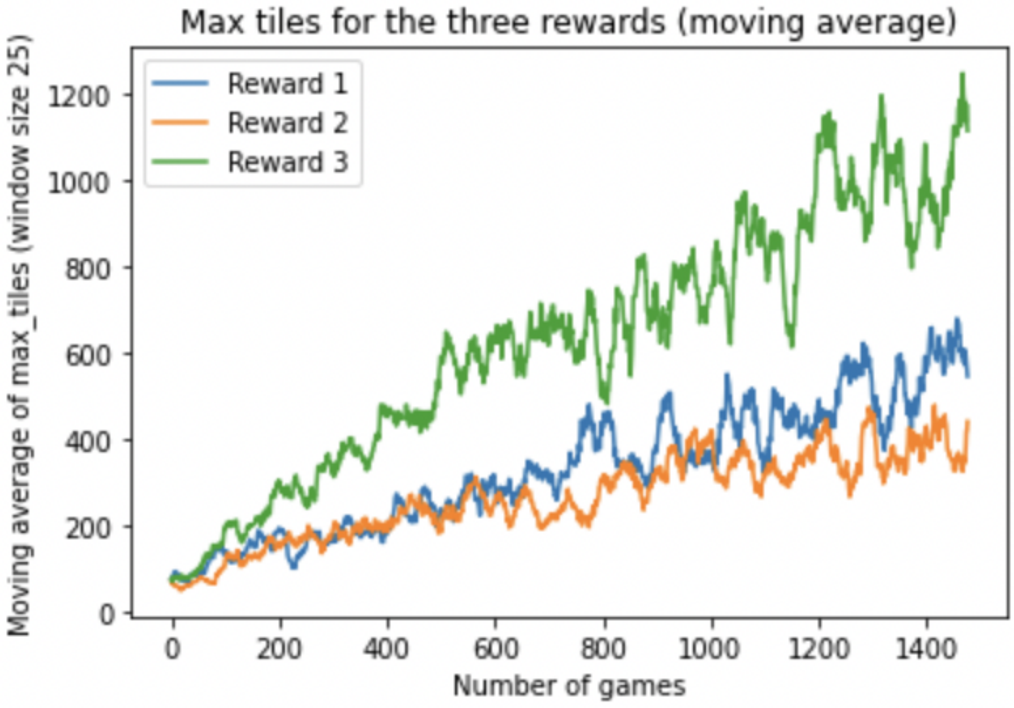


Fig 7: Averaged of maximum tiles reached for each game

3.2 Testing

After training for 1500 episodes, the performance of the three agents was tested by playing 1000 games and recording the number of occurrences of each maximum tile and the results are given in Table 2.

Max Tile	Reward 1 - Score	Reward 2 - Merges and Max tiles	Reward 3 - Empty tiles
32	2	4	1
64	11	34	1
128	46	177	5
256	166	310	29
512	448	390	147
1024	315	85	492
2048	12	0	325
4096	0	0	0

Table 2: Occurrences of each maximum tile for the 3 rewards for 1000 games

Reward 3 outperforms the other rewards and reaches 2048 3.25% v.s 0.12% and 0%. It also reaches 1024 more frequently. Reward 1 gives a better performance than reward 2 and reaches 2048 0.12% (compared to 0) and 1024 3.15% times (compared to 0.85%).

4. Conclusion

It can be concluded that the DQN agent using the number of empty tiles as a reward function performs reasonably well, since it achieves the objective 2048 in 3.25 percent of the games after only 1500 episodes of training.

The agent with reward 3 outperforms the agents with the other two rewards - score and maximum tile. The reason for this is that the agent attempts to empty the grid as much as possible, allowing it to stay in the game for longer. It's also worth noting that counting the number of empty tiles without considering the contents of the tiles yields better results. It can also be observed that agents with other rewards rarely pick the action of merging low-value tiles, which eventually results in a full grid. Agent with reward 3 gives better results as the agent picks actions that do not immediately boost the score (i.e. reward 1) but allow for greater space for substantial merging.

Furthermore, agent 1 outperforms agent 2 due to the definition of reward 2, which causes agent 2 to overlook merges of high-value tiles when the resulting tile is already existing in the grid.

Finally, each of these agents has the flaw of failing to conserve strategic positions: for example, a traditional strategy for this game is to arrange high-value tiles in the corners, which does not occur during the games played by the agents.

References

Amar J & Dedieu A., "Deep Reinforcement Learning for 2048." (2017)
http://www.mit.edu/~amarj/files/2048.pdf

Ankit Choudhary. A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python (April, 2019)
https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

Anonymous Authors. Reinforcement Learning Project - 2048 Game (2020)
https://github.com/achraf-azize/2048-DQN

Nneoneo. (Mar 22, 2017). AI for the 2048 game [Online] https://github.com/nneonneo/2048-ai

Shilun Li, Veronica Peng. Playing 2048 With Reinforcement Learning (October, 2021)
https://arxiv.org/pdf/2110.10374v1.pdf

Szubert, Marcin and Wojciech Jaskowski. "Temporal Difference Learning of N-Tuple Networks for the Game 2048." IEEE, (2014)
http://www.cs.put.poznan.pl/wjaskowski/pub/papers/Szubert2014_2048.pdf

Varun Kaundinya, Shubham Jain, Sumanth Saligram, C. K. Vanamala*, Avinash B. Game Playing Agent for 2048 using Deep Reinforcement Learning (2018)
https://pdfs.semanticscholar.org/6f96/a3907eb44852dce15fc12d6931e8237dc2e2.pdf

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. “Playing Atari with Deep Reinforcement Learning”. In Deep Learning, Neural Information Processing Systems Workshop, 2013
https://arxiv.org/pdf/1312.5602.pdf

Yeh, Kun-Hao, et al. "Multi-Stage Temporal Difference Learning for 2048-like Games." TCIAIG, (2015) https://arxiv.org/pdf/1606.07374.pdf

Appendix

GitHub Link:
https://github.com/aesha10/Reinforcement-Learning-2048

Given below are the graphs of maximum tile obtained during training for the three rewards.

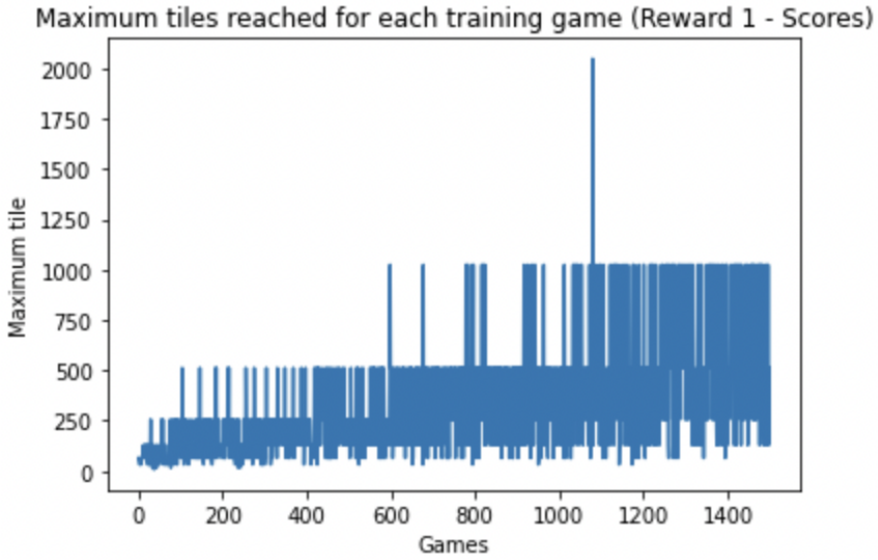


Fig 8: Maximum tiles for Reward 1 (Score)

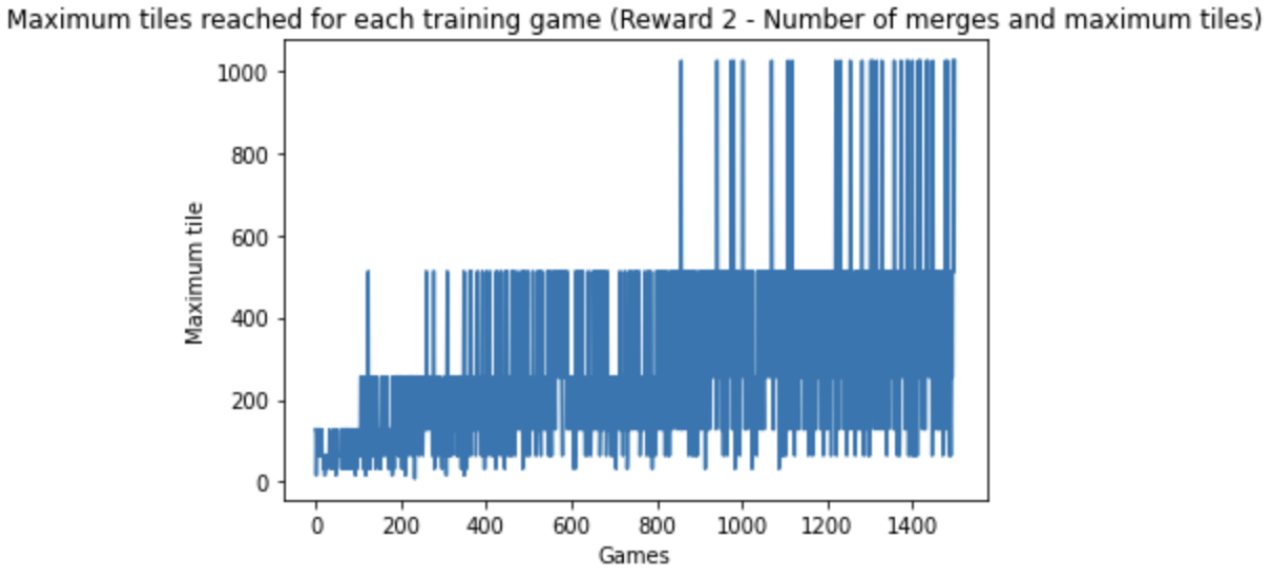


Fig 9: Maximum tiles for Reward 2 (Number of merge and maximum tiles)

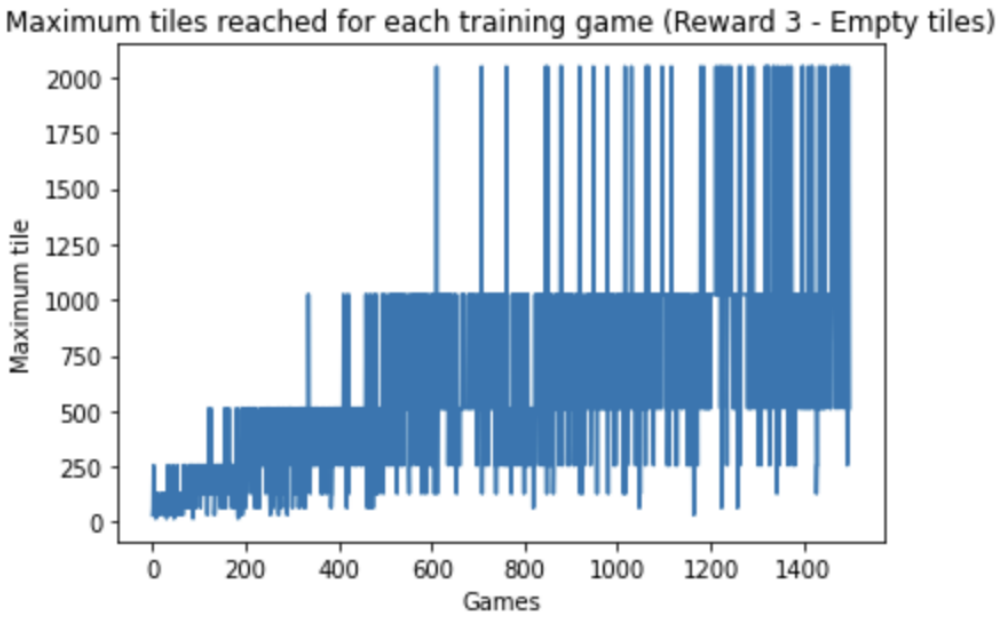


Fig 10: Maximum tiles for Reward 3 (Number of empty tiles)