Reinforcement Learning on 2048 Game

150050060 - Teja Madaka
150050091 - Bhavan Turaka
150050081 - Mani Shankar
150050095 - Akhil Orsu

.

Abstract

The goal of the project was to research the possibilities of creating a reinforcement learning based agent, that could learn to play the 2048 board game. This game introduces a difficult task due to its factor of randomness. We designed and tested a base model of the agent using of a deep feed forward neural network and two types of input encoding in a number of experiments. We implemented the agent using the Keras library with TensorFlow backend in Python language. For visualization of its performance we used Matplotlib library.

1. Introduction

Reinforcement learning algorithms aims at finding the optimal policy of the environment encoded as an MDP and thereby maximizing the long-term reward besides learning about the environment which in notion can be described as tradeoff between exploitation vs exploration in an MDP. 2048 (Gabriele, 2014) is one such popular game which gained popularity in RL field because of its medium solving complexity unlike Atari games where we have only win or lose situations. Currently algorithms like expectiminimax, ϵ - greedy learning with Convolutional Neural Network decision algorithm , RL algorithms like value iteration with discrete states and ϵ - greedy learning algorithms with monte carlo exploration (Yun et al., 2016) are found majorly to be implemented to solve this game. We intend to solve this task through the Q-Learning RL algorithms and evaluate the performance of it. One point to note in this approach is encoding the game as an MDP for implementing this algorithm results in high blow-up in the number of states(as high as 2^{64}) and therefore learning the Q-Table is quite a hard task. We then investigate how we can improve the performance in learning . For this we use the implementation of Q-Learning as Deep Neural Networks as one such potential method.

In this project , we study and implement these approaches to solve this task. Based on variety in implementation of DNNs this is of two types Q-networks , Policy Neural Net. Q-network is the NN version of estimating the Q-value for the state-action pair in an MDP whereas the Policy Net is a hybrid version wherein policy/action is determined for the state.

2. Research done in 2048 game

There were many attempts at solving the game by an artificial intelligence. (Szubert & Jaskowski, 2014) used Temporal Difference (TD) learning together with N-tuple network to achieve a winning rate of 97%. (Oka & Matsuzaki, 2016) extended previous TD approaches and discovered that the positions of N-tuples significantly influence the performance of the algorithm. (Dedieu & Amar, 2017) implemented a deep Policy Network, but their agent was not able to achieve the 2048-tile a single time.

3. Problem Description

3.1. Game overview

2048 is a single-player puzzle game developed by Gabriele Cirulli[1] during a single weekend . 2048 is played on a board of size 4×4 where the player in each turn slides the board in one of the four directions in order to merge tiles of equal values and create tiles of higher values. The game rules are as follows given in (Adrian, 2017): Initially the board has 2 tiles on two positions of the board selected randomly. The value of the tiles will be 2 with probability 0.9 and 4 with probability of 0.1. A player in each step performs an action from the set of actions $\{$Up, Down, Left, Right$\}$. For every action, the tiles will move in the direction of action till they hit the edge of the board or any other tile. When a tile hit another tile of same value, they merge to form a tile whose value is some of their values. After every action, a random tile will spawn in one of the empty cells on the board with uniform probability. The value of this tile can be 2 or 4 with probabilities 0.9 and 0.1 respectively. The game ends when there is no valid action. A valid action is an action, which when performed, should leave at least one empty cell for the new tile to spawn after merging. The goal of the game is to reach a tile whose value is 2048.

3.2. Game environment

We used a 4x4 matrix whose values are the exponents of the values of the tiles in 2 powers to represent the board while playing. The empty tile will have a value 0.

4. Our Algorithms

The game is modelled as an MDP wherein the current game state is encoded as 256 bit vector (16 bits for the value held in each cell as the max value possible is 2^{16}). As the total number of states explodes to 16^{16} (2^{64}) states, storing Q values in a table isn't a great idea. So we used neural networks to predict Q values or to predict the best action. We worked on two neural networks and selects the one which works better. The first one takes (state, action) pair as input and gives Q(state,action) as output which we called Q-Neural Net. The second neural network takes a state as the input and gives Q(state,action) for all actions taken from that state which we called as Policy Neural Net. In both Neural Nets epsilon-greedy exploration is used.

4.1. Catastrophic Interference and Experience Replay

Neural networks can only learn multiple tasks when trained on them jointly. When tasks arrive sequentially, they lose performance on previously learnt tasks. This is called Catastrophic Forgetting(Interference) and is an important challenge to overcome in order to enable systems to learn continuously. An important assumption for successful gradient-based learning is to observe iid samples from the joint distribution of all tasks to be learnt. Since sequential learning systems violate this assumption, catastrophic forgetting is inevitable. So a direct approach would be to store previously seen samples and replay them along with new samples in appropriate proportions to restore the iid sampling assumption. (Nitin et al., 2018)

To perform experience replay, we store the agent's experiences $e^t(s^t,a^t,r^t,s^{t+1})$ at each time-step t in a data set $D^t=\{e^1,...,e^t\}$. During learning, we apply Q-learning updates, on samples (or mini-batches) of experience (s,a,r,s') $\in$ U(D) (Subset), drawn uniformly at random from the pool of stored samples. This experience replay is relevant for this situation because while playing the game the tasks do come in sequentially and this results in Catastrophic Interference as mentioned above. For our Problem we used replay memory of size 50,000 steps and the size of mini-batch which is random sampled is of size 1000.

4.2. Reward Function

The reward function we used is $4\times$ the score obtained by action taken minus sum of the scores obtained for every possible action from that state.

$$R(s,a,s') = 4 \times (Score(s,a)) - \sum_{a'} Score(s,a')$$

[1]the original game can be played at https://gabrielecirulli.github.io/2048/

4.3. Q- Neural Net

We train the neural network using Q-learning algorithm with input encoded as 192 x 4 = 768 (Since aim of the game is to reach 2048 which is 2^{12} , each configuration of the board can be represented by using 16 * 12 = 192 bit vector) bit vector where each quartile represents a specific action as in 1st quartile represents action Up, 2nd quartile represents action Right, 3rd quartile represents action Down and 4th quartile represents action Left. To get a Q(s,a) value the input to neural network is of the form where the quartile representing action a contains encoded form of state s and the remaining quartiles are filled with zeroes. This normalizes the input and makes finding patterns easier for the Neural Net although takes longer time to train. This Neural network contains 2 hidden layers with 1 hidden layer containing 200 nodes and the other hidden layer containing 20 hidden nodes, the output of NN contains 1 node which gives Q(s,a) value. Activation functions for all nodes except output layer are Relu for the output nodes the activation function is linear. Mean-square error function with adam Optimizer is used for back-propagation. For a state-action-reward-nextstate step taken by the agent, Q learning algorithm uses the below formula to update the Q(state,action). We also used the for Q value training.

Q(state,action) = (reward + gamma $\times max(Q(state, action')\forall$ action' $\in$ actions)

After filling up the replay memory of size 50000 steps, the NN is updated by taking randomly 1000 steps and doing the batch training of size 1000 and number of epochs equals to 1. Here replay memory is a Queue, after the maximum size is reached for every further step taken by the agent ,a tuple is deleted and this new (s,a,r,s') tuple is added into queue. Since replay memory size doesn't change for actions from this step, the neural network training can be started.

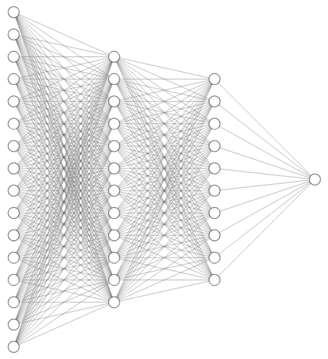

Figure 1. Scaled Neural Network visualization of Q-NN mode

4.4. Policy Neural Net

We train the neural network using Q-learning algorithm with input encoded as 192 (As mentioned above in Q-neural net each state can be encoded as 192 bit vector). This normalizes the input and makes finding patterns easier. The output of Neural Network contains 4 nodes each representing Q(s,a) for 4 actions for the input state s. This neural network contains 3 hidden layers with first hidden layer having 100 nodes and second hidden layer containing 50 nodes and the third hidden layer containing 15 nodes. Activation Functions for all nodes except output layer are Relu for the output nodes the activation function is linear. Mean-square error function with adam Optimizer is used for backpropagation. For a state-action-reward-nextstate step taken by the agent, we updated the Q values using the formula:

Q(state,action) = reward + max(Q(nextstate))/(number of all possible next-states) $\forall$ possible next states

This is chosen instead of generic Q-learning because the 2048 is itself a stochastic game(the generation of new 2s and 4s is probabilistic) so instead of max Q value of next state , finding the probabilistic max Q sum of all possible next states averages out this stochasticity. After filling up the replay memory of size 50000 steps, the NN is updated by taking randomly 1000 steps and doing the batch training of size 1000 and number of epochs equals to 1. Here replay memory is a Queue, after the maximum size is reached for every further step taken by the agent ,a tuple is deleted and this new (s,a,r,s') tuple is added into queue. Since replay memory size doesn't change for actions from this step, the neural network training can be started.

In the above Neural network after calculating new Q(state,action) for updating Neural Network instead of updating Q value corresponding to the action action , Q(state,a) for all actions are updated with actions which are not equal to action are kept the same as the previous values. This way we are reinforcing Q(state,a) is correct for a != action. This might result in bad results. Instead if we use masking in neural networks (Ecoffet, 2017) this problem can be overcomed. We can mask all the outputs except the output for the action taken, while updating neural network though we didn't implement this.

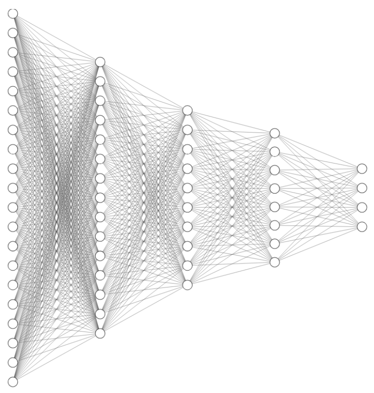

Figure 2. Scaled Neural Network visualization of Policy-NN mode

5. Results and Observations

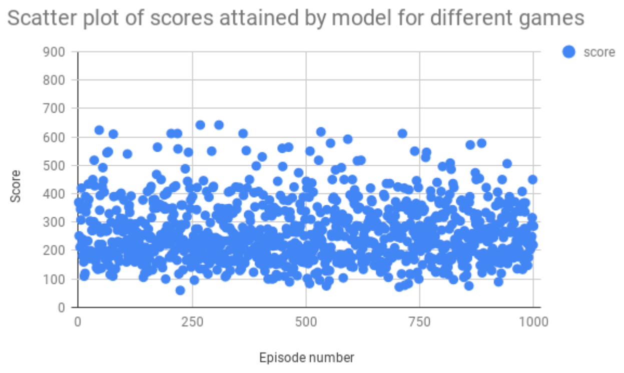

Figure 3. scatter plot obtained by model

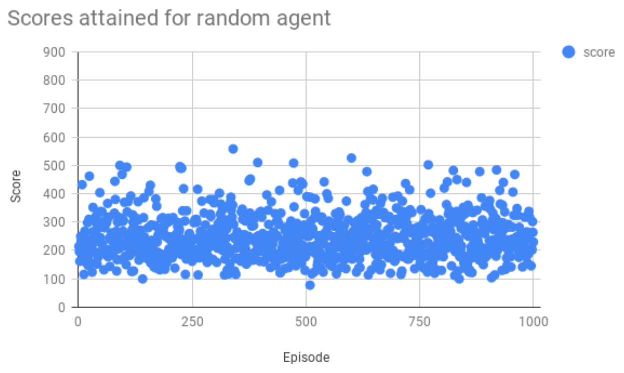

Figure 4. scatter plot obtained by random agent

Maximum score (sum of all tiles at the end of the game) attained by our model is more than 500 for a good number of games (3.5%) where as score is greater than 600 for 1% of the games out of 1000 games. Maximum score (sum of all tiles at the end of the game) attained by the random model is more than 500 for only 0.5% of the games whereas score is greater than 600 for 0% of the games.

Table 1. Permormance Comparion of Model with Random agent with scores

Score	>400	>500	>600
Our Model	10.6%	3.5%	1.0%
Random agent	4.66%	0.6%	0.0%%

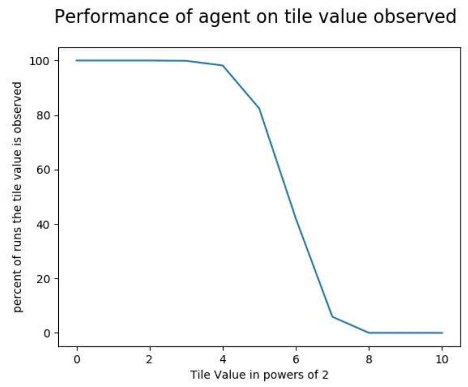

Figure 5. The percentage of games for which tet given value is occured on the board atleast once (Percentage vs log)

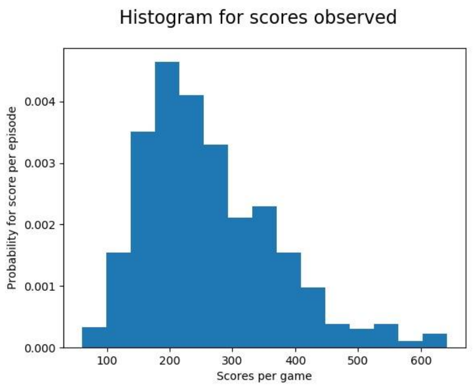

Figure 6. Probability with which a score is attained among all games played

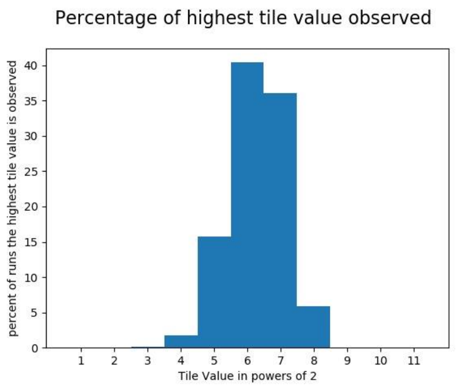

Figure 7. Number of episodes for which highest tile value is observed vs log(highest tile value)

6. Conclusion

We designed and implemented agents based on the Deep Q Network RL algorithm, Batch Q Network algorithm with experience replay to prevent Catastrophic Interference . We expected our agents to perform better ,but owing to the mis-fitting of Deep NN we couldn't achieve our expected goals though it was a significant achievement to the learning aspect of the agent. These Neural Nets couldn't capture the exact function approximation for Q-Function as there is stochasticity involved in the game. Humans can discover the right moves easily that lead to winning. One thing we had observed in this game is it's a harder challenge to meet for Deep Neural Network based reinforcement learning algorithms, due to variety of rewards and possibility of various moves thereby more randomness in estimating next state for state-action pair. Thus we think these NN based Q-learning algorithms game requires an advanced level of study and implementation for better learning of the agent. As seen in implementation of deep Policy Network for this game by (Dedieu & Amar, 2017), it is indeed difficult to get 2048 using policy networks.

References

Adrian. http://www.dcs.fmph.uniba.sk/bakalarky/registracia/getfile.php/main.pdf?id=375fid=740type=application%2Fpdf. Technical report, 2017.

Dedieu and Amar. https://stuff.mit.edu/people/adedieu/pdf/2048.pdf . Technical report, 2017.

Ecoffet, Adrien Lucas. https://becominghuman.ai/lets-build-an-atari-ai-part-1-dqn-df57e8ff3b26 . Technical report, 2017.

Gabriele. 2048(video game) - wikipedia , 2014.

Nitin, Umang, and Liu. https://arxiv.org/pdf/1710.10368.pdf . Technical report, 2018.

Oka, Kazuto and Matsuzaki, Kiminori. Systematic selection of n-tuple networks for game 2048. pp. 81–92. Springer International Publishing., 2016.

Szubert and Jaskowski. http://www.cs.put.poznan.pl/mszubert/pub/szubert2014cig.pdf. Technical report, 2014.

Yun, Wenqi, and Yicheng. http://cs229.stanford.edu/proj2016/report/NieHouAn-AIPlays2048-report.pdf. Technical report, 2016.