Game Playing Agent for 2048 using Deep Reinforcement Learning

Varun Kaundinya, Shubham Jain, Sumanth Saligram, C. K. Vanamala*, Avinash B

NIE, Mysuru

DOI: https://doi.org/10.21467/proceedings.1.57
* Corresponding author email: ckvanamala@nie.ac.in

Abstract

Reinforcement learning is used in applications where there is no correct approach to solve the problem. Teaching computers to play games is a complex learning problem that mostly depends on the game complexity and representational complexity, which has recently seen increased attention towards this field. This paper presents an approach using concepts of reinforcement learning to master the game of 2048. The approach is based on Q learning and SARSA (State-Action-Reward-State-Action) which are the most popular algorithms in the field of reinforcement learning. The design involves the use of neural networks as the function approximation method. Like most deep Q learning models, the heart of any reinforcement learning agent is the reward function. In this paper reward functions are designed to train the model to learn the best playing moves. But in 2048 there are 4 random components, that is, initial configuration of the game, addition of random tiles after very move, exploration of the agent and unavailability of moves. This paper attempts to provide an approach to solve the issue of inherent randomness in 2048. This approach is based on prioritized experience replay where the model trains to learn best game-playing strategy from the experiences collected. With this approach we achieved maximum tile value of 512.

Index Terms- Reinforcement learning, Q-learning, Experience Replay, 2048.

1 INTRODUCTION

Reinforcement Learning is a type of learning mechanism where the agent learns by performing tasks which are quantified by rewards. It is used in applications where there is no correct approach to solve the problem. For example, in self-driving cars it would be unrealistic to program it based on rules instead it should learn to perform best possible actions under uncertain circumstances which it learns by getting rewards on performing desirable actions.

The goal of this project is to build an agent learn to play 2048. The game is played on a 4 × 4 board as shown in Fig 1. The tiles are either in power of 2 or blank. It starts with a random configuration of two tiles consisting of 2s or 4s. Four actions are available: left, right, up and down. But not all actions are possible in every game situation and a random tile of 2 or 4 is

 ISBN: 978-81-936820-0-5

added after each move. Score is accumulated by merging tiles of same powers by performing actions. The goal of the game is to achieve a tile of 2048.

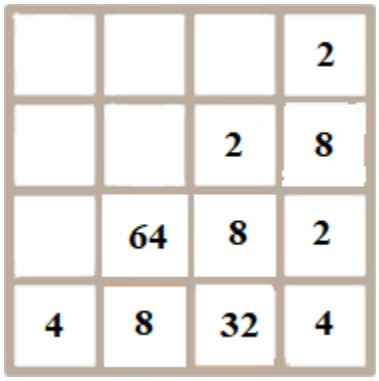

Fig 1: Basic structure of 2048

The difficulties in this game are initial configurations of the game are random in nature, addition of random tiles after very move, explorative nature of the agent during initial phase, large state space and unavailability of moves in certain situations. Hence, building a successful agent for this game is very difficult.

2 LITERATURE SURVEY

The most prominent work in this field is done by Google Deepmind [1]. They used deep Q learning methods to build an agent that was successful in playing 7 of the Atari 2600 games. Their agent was able to surpass human expert-level in three games. The salient feature of their project was that they were able to train an agent despite extremely high dimensional input (pixels) and no information about the game. There was a successful attempt at learning the game 2048 by [2]. They used expectimax optimization and they were able achieve tiles of 8192 on all most all runs of the game. Their model simply performs maximization over all possible moves, followed by expectation over all possible tile spawns (weighted by the probability of the tiles, i.e. 10% for a 4 and 90% for a 2). This is an alternative approach which is not based on reinforcement learning but has produced very good results. Another attempt at learning to play 2048 was done by [3]. They used Monte-Carlo Tree search strategy to encourage optimal decision making during explorative phase , but this turned out be very expensive and did not produce successful results.

3 METHOD

In this section we describe the various algorithms and parameters used in building our agent.

3.1 State Representation

This project uses the coded version of the game [4]. Here the game states are represented in the form of matrix with each element representing an exponent of 2 as shown in Fig 2. Each element is equivalent to a tile in the 4×4 grid of the game. This design choice was made to be computationally efficient. The matrix is flattened into a 1-D array which is fed as an input to

ISBN: 978-81-936820-0-5
Proceedings DOI: 10.21467/proceedings.1

Series: AIJR Proceedings

the neural network.

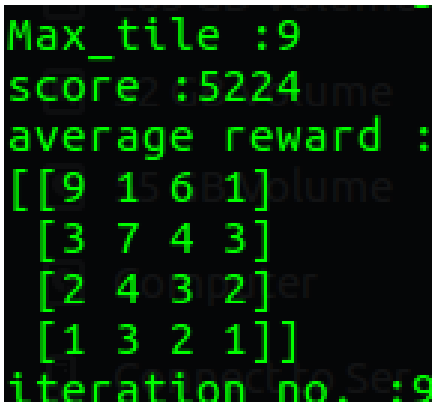

Fig 2: State representation of 2048 in implementation

3.1 Algorithms Implemented

Q learning and SARSA are the fundamental algorithms in the field of reinforcement learning which works by maximizing the expected value of the total payoff.

In Q learning, the agent tries to learn the optimal policy from its history of playing the same game previously. In Q learning, the history is the experiences from the previous runs which is stored in the format (S_0, A_0, R_0, S_1) where S_0 indicates the state in which the agent was in and A_0 represents the action that was taken and R_0 is the reward it had received for taking the action, while S_1 is the state reached due to A_0. Q learning is an off policy algorithm, this is because Q-learning is learnt via an optimal policy (max Q(s,a)), but its behavior policy is not same as the optimal policy, since the behavior policy may take actions that are explorative that is why Q learning is an off policy algorithm. SARSA is learnt via a non-optimal policy (Q(s,a)) and its behavior policy is also non optimal and takes actions that are explorative that is why SARSA is called an on-policy algorithm. SARSA stores experiences in the format (S_0, A_0, R_0, S_1, A_1) which is the indicative of the origin of the algorithm's name.

These algorithms direct the agent to learn an optimal policy by making it learn a Q function which is used for optimal action selection. These algorithms don't actually specify what the agent must do. The agent can make two types of actions: exploitation and exploration. In exploitation the agent utilizes the knowledge of current state to take an action that maximizes the Q value. While in exploration the agent doesn't necessarily takes an action that maximizes the Q, rather it takes an action other than the one it thinks is the best. This is necessary to build a better estimate of the optimal policy. The way exploration and exploitation is implemented is by having an agent make a lot of explorative moves in the beginning that is when the agent has no idea how to play the game and the unexplored state space is very large. This leads to random actions. After much iteration the agent converges to which actions/states are better and therefore it would exploit more and build a better estimate of the optimal policy. The key strategies to implement this are using ε-greedy method and softmax action selection.

In ε-greedy, ε times the agent explores and 1-ε the agent exploits. This gives programmer control over the behavior of the agent. But there are two problems with a *ε*-greedy strategy: one is that the actions other than best one are treated equally and a random pick is done on them. But it would be preferable to choose one of the better moves from the remaining actions (other than the best), instead of having a random pick. As better moves lead better rewards and better game situations and the other is that it adds another random component to an inherently random game of 2048. One way to do that is to select action with a probability depending on the Q value. Hence we use soft-max action selection. In our implementation we use multinomial distribution to sample the action.

3.2 Experience Replay

Experience replay is a data structure in which previous moves performed by reinforcement learning agent are stored. The experience replay is built via simultaneously appending actions performed onto an nd-array, list or even hash maps (based on efficiency). The presence of experience replay is what makes reinforcement learning work, that is, it helps de-correlate experiences experienced by the agent. From this the list of experiences are randomly sampled out and subjected to batch learning, by doing this reinforcement learning agent learns from de-correlated experiences and chances of over-fitting are reduced.

For example: In self-driving cars, suppose the agent in its training phase was trained to drive on straight roads 85% of times and turns 15 % of times, if this was used as a dataset to make the agent learn, then the agent will always be skewed towards learning to drive on straight road, and will also over-fit on patterns of straight road, due to data being biased, but by randomly sampling and de-correlation helps us avoid such issues.

Another advantage of experience replay is, agents get to train on rare experiences more often than in normal case of online learning due to random sampling. Experience Replay is analogous to a dataset in supervised learning but different in the sense that it is not fixed and is constantly updated. We used 3 approaches to implement experience replay:

In the first approach we stored the actions in the experience replay without differentiating on which action belonged to which game. We also randomly sampled experience from the experience replay during learn phase.

In the second approach we stored actions belonging to a particular game run (start to finish) in a separate data structure which was then appended to experience replay, during learning phase we sampled games randomly and then sequentially appended their actions to build batches which are input to the neural network.

In the third approach we stored actions belonging to a particular game run in a separate data structure which was only appended to the experience replay based on a condition , That is say a threshold score , This would mean network gets trained on games that have score better than the threshold, rest of the procedure is same as in 2 . This idea is called prioritized experience replay.

ISBN: 978-81-936820-0-5
Proceedings DOI: 10.21467/proceedings.1

Series: AIJR Proceedings

3.3 Reward Function

Heart of reinforcement learning systems is reward function. Reward function quantifies how desirable an action taken by the agent. Design of reward function should be in such a way that it must be easy for the agent to pick up good game playing strategies, an ideal reward function takes an agent making desirable moves towards the goal rather than away from it. Reward functions must be designed keeping in mind the tradeoff between intent that is what the agent must do and incentive what the agent actually does.

Example: For example a goal of a robotic arm is to place blocks one over another, if say it was rewarded a point for placing one block over another, here the intent is that we want the agent to build a stack, while in essence agent can easily place the block and then remove it and then again place the block in a loop, technically agent is getting rewarded for this but this is not desirable at all.

In our implementation of the game, we found from personal experience from playing the game that placing the maximum tile in corners always leads to higher scores, having more number of moves and free tiles available at any point is more desirable, monotonous decreases of powers in row and column of the max tile and in order is also desirable and moves that lead more number of merges in turn leads to higher scores and free tiles are better.

Learning rate is 0.009, gamma is 0.9 and algorithm used is Q-learning for all our models described in this section. The first reward function gave a consistent increase in reward based on the value of the max tile being in the corner. Along with this if the agent was able to maintain monotonicity in the particular row or column, it was further rewarded.

The second reward function used prioritized experience replay in which we stored actions of games which has scored more than a threshold set. The rewarding scheme remained the same as previous.

In third reward function, the elements of experience replay were the whole set of actions that the agent took in a game instead of individual actions. We randomly sampled the games and batched up the actions. This was used for training the neural network. The rewarding scheme was the score.

4 RESULTS AND ANALYSIS

4.1 First Reward Function

All the reward functions were trained on 35,000 games. The agent partly received the intention of this function as mentioned above. It was able to follow the policy to make tiles of 256. This was due to the present of large number of free tiles. In most cases, the game got terminated very quickly from this state due to the unavailability of free tiles and random tile addition at undesirable positions as seen in Fig 3. To increase the lifespan of the game and making the agent understand the strategy better, prioritized experience replay was used in the next reward function as mentioned in the previous section.

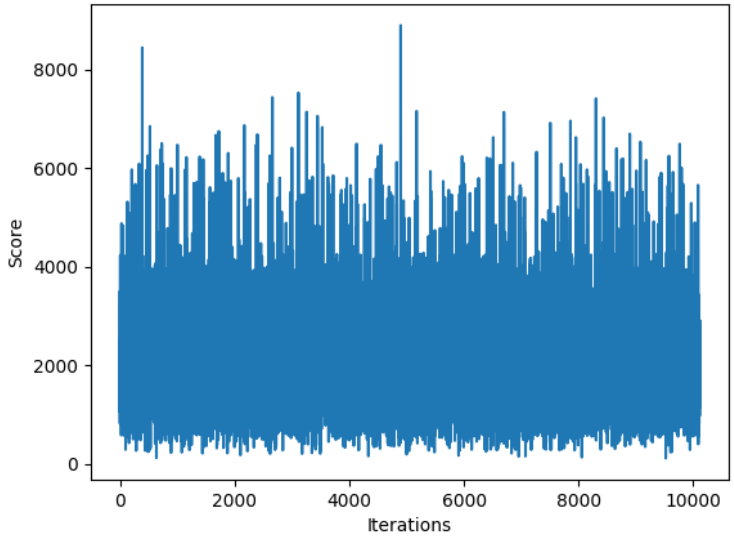

Fig 3: Graph for reward function 1

4.2 Second Reward Function

By this method we achieved underwhelming results compared to the previous method as shown in the Fig 4. On analysis, we found that the poor performance of this method was due to the sampling of the actions randomly from the experience replay. Even though good game moves were stored in the experience replay but due to the de-correlation of the strategy that was used during the game the network was not able to learn any optimal policy.

This had an effect on previous method as well. We also concluded that the reward function was complicated for the agent to learn the policy when it reached deeper into the game (reaching 256).

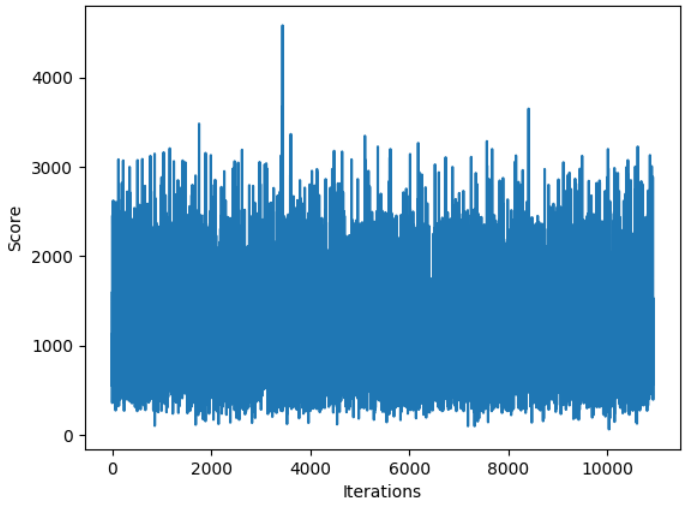

Fig 4: Graph for reward function 2

ISBN: 978-81-936820-0-5
Proceedings DOI: 10.21467/proceedings.1

Series: AIJR Proceedings

4.3 Third Reward Function

To overcome the problems in the previous method we used game score as the reward function and bundled actions into games which were then appended to experience replay memory. This would retain the in-game policy. We achieved our best results using this method. We increased the lifespan of the game which is shown in the Fig 5.

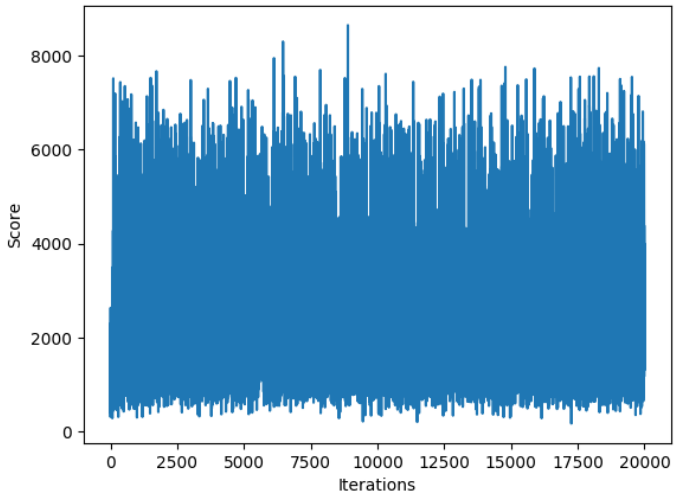

Fig 5: Graph for reward function 3

4.4 Other Experiments

Above reward functions were applied on SARSA algorithm. The results were similar. We also implemented more complex reward functions which consisted of snake-shaped patterns, number of free tiles, number of merges, distance between the same valued tiles. These results were not very successful as the agent would get one reward for many factors which the agent couldn't always comprehend.

5 CONCLUSION

Although our initial goal was to master the game 2048, which we weren't able to achieve, all of our models achieved max-tile of 512.We reflect on what lead to such outcomes and why 2048 was not mastered. Our agent learned to make the tiles of 128 and 256 very easily, during this phase the agent played the game optimally by having largest tile in the corner and it also had monotonous decrease in the tile power in both row and column of the max-tile. We believe the reason the agent mastered this strategy was due to availability of large number of free tiles due to which random move done at this stage is recoverable in terms of strategy learnt, but as the agent progressed to make tiles of higher value like 1024 and 512 it struggled. This is because there were very few free tiles available and number of moves available was also

very small, due to which even one explorative move at this stage or placement of a random tile in an undesirable position ruined the game strategy which leads to early game termination. This is an interesting project, due to its inherent randomness fitting it into reinforcement learning methods is difficult, this requires more research into optimal reward function which would make the agent learn the game strategy and also learn expectation over randomness.

6 FUTURE SCOPE

Here are few things which we believe might help get better results, than what we have achieved. Clipping the rewards between -1 to +1 [1], storing experiences in exponential manner based on game scores, that is storing actions that lead to better scores based on total score achieved, so idea is to set certain limits on score proportional to number of iteration such that agent would only store all experiences better than the limit set. It is actually found that as we go deeper into the game, the number of available moves decrease but also number of optimal moves comes down (only 2 moves in most cases) and this is dependent on where max-tile is placed. So designing a reward function such that it takes into account the reduced optimal moves as we go deeper into the game (512 and above) can help.

One alternative is usage of dueling Q-architecture instead of classic Q-learning. Also, there can be reduction in the overestimation of the classic Q-learning algorithm. Doubling Q-learning algorithm can be used.

References

[1] 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*

[2] Nneoneo. (Mar 22, 2017). *AI for the 2048 game* [Online] https://github.com/nneonneo/2048-ai. [Accessed : 8th March 2018]

[3] Antoine Dedieu and Jonathan Amar. "Deep Reinforcement Learning for 2048".

[4] George Wiese. (July 10, 2016). *2048 Reinforcement Learning.* [Online] https://github.com/georgwiese/2048-rl [Accessed : 8th March 2018]

[5] Richard Sutton and Andrew Barto. Reinforcement Learning: An Introduction. MIT Press, 1988.

[6] T. Schaul, J. Quan, Antonoglou, D. Silver, "Prioritized Experience Replay". *ICLR 2016.* http://arxiv.org/abs/1511.05952

[7] Andrej Karpathy. (May 31, 2016) *Reinforcement Learning: Pong from Pixels*. [Online] http://karpathy.github.io/2016/05/31/rl [Accessed : 15th March]

[8] Arthur Juliani. (August 25, 2016). *Simple Reinforcement Learning with Tensorflow (10 Parts)* [Online] https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-0-q-learning-with-tables-and-neural-networks-d195264329d0

[9] Kevin Chen, Deep Reinforcement Learning for Flappy Bird, CS 229 Machine-Learning Final Projects, Autumn 2015

[10] David Silver. (2015) *"UCL Course on RL"* [Online] http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html

ISBN: 978-81-936820-0-5
Proceedings DOI: 10.21467/proceedings.1

Series: AIJR Proceedings