TDL Value Function Approximations in *2048*

May 14, 2020

Owen Siljander

silja006@umn.edu

University of Minnesota

Abstract

2048 is a puzzle game released in 2014 by Gabriele Cirulli that quickly gained popularity, reaching thousands of years of total playtime by players [8]. The game is particularly attractive as a testing environment for agent design due to it being a discrete time stochastic environment with a substantial state space. Many classical search approaches have been studied under this environment, as well as reinforcement learning agents, with substantial success over human players [8]. Explicit value functions (e.g. table lookup) for temporal difference learning (TDL) are not equipped to handle a state space as large as the one in *2048,* and so this paper investigates linear functions of features and neural networks as value function approximations for TDL.

The results of this study are mostly inconclusive in part due to computing and time constraints. The linear function weight vectors in experiments would often diverge extremely quickly (in less than 10 learning steps) which is perhaps caused by the poor convergence of linear approximaters in general [5][3]. Experiments with four three-layered non-linear neural networks showed that it occasionally performed better than a simple random agent. It is impossible to make a claim in confidence about the performance in general due to the insufficient number of games the networks trained for and evaluated with. The time it takes for this variant to make a decision is comparable to classical approaches at around 20s. There is large room for improvements and experimentation in the application of both these value function approximaters in *2048*.

1 Introduction

1.1 Environment

The game *2048* consists of a 4×4 grid starting out with two tiles in random locations with each tile having a value of 2 or 4 (ex. Figure 1.1). The player has the available moves of *up*, *down*, *left*, or

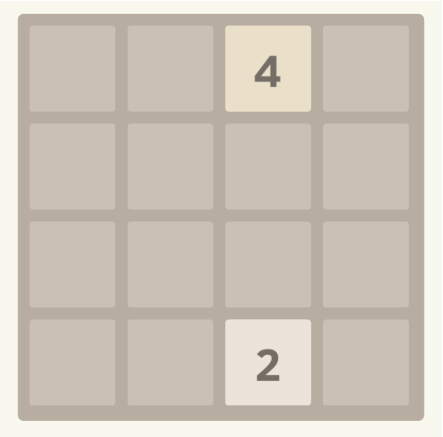

Figure 1.1: Example starting state for *2048*

right. Each of these actions will slide the tiles in the appropriate direction. If a tile collides with or moves into another tile of the same value, they combine into a single tile with a value equal to their sum. A tile which has been produced through such a combination cannot combine with another tile within the same move; that is to say, a player could not combine a row of 2s into a single tile with a value of 8, it would instead produce two tiles of value 4.[1] If a move would not combine or move any tiles on the grid, it is not a legal move. Each time the agent takes an action, a new tile is generated in a random location having a value of 2 with probability 0.9 and a value of 4 with probability 0.1. The game ends when there is no legal move (i.e. cannot move or combine tiles). For every move, the agent receives a reward equal to the value of all tiles which have been combined (e.g. combining two 2 tiles and two 4 tiles results in a reward of 12). The game is considered to be won when the agent has produced a tile with a value of 2048, though the agent can keep playing to achieve higher scores.

The reason this environment is challenging is not only due to the stochastic nature of the game, but also due to the large state space. Considering each tile's value is a power of two, and the maximum possible tile value (in practice) is 2^{17}, there are roughly $18^{16} \approx 10^{20}$ possible states (18 to include the zero/empty tile) [6].[2] This state space increases greatly when you consider state-action pairs. This large state space has made sampling algorithms, such as Monte-Carlo tree search (MCTS), particularly attractive solutions.

1.2 Requisite Knowledge

This paper focuses on the application of temporal difference learning (TDL) in *2048* where the environment is modeled as a Markovian decision process (MDP). Temporal difference learning is a particular class of model-free reinforcement learning algorithms that use a value function, which calculates the expected utility of a state, as a means for decision making. By determining the value

[1] Some research fails to state this property of the game.

[2] [6] calculates the number of states to be $16^{18} = 4.7 \times 10^{21}$ but I believe this calculation to be erroneous.

of potential next states, an agent can use the value function to choose the best action to take. TDL works by continually adjusting its estimated value function, $V^\pi(s)$, under some policy π. Generally, TDL is denoted by TD(λ) where λ denotes the number of states it looks ahead. One of the simplest forms is where $\lambda = 0$, this actually denotes a one step look-ahead. In this case, the value function is updated as shown in Equation 1.1.

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha(r + \gamma V^\pi(s') - V^\pi(s)) \tag{1.1}$$

As for notation, $\alpha \in [0, 1]$ is some learning rate, $\gamma \in [0, 1]$ is the discount factor, s' is the state following an action from s under the policy π, and r is the state reward. The value $r + \gamma V^\pi(s')$ is commonly called the target and denotes the temporal difference between the current estimate and the updated estimate. The value function additionally satisfies Bellman's equation (Equation 1.2).

$$V^\pi(s) = R(s) + \gamma \sum_{s'} P(s'|s, \pi(s))V^\pi(s') \tag{1.2}$$

Bellman's equation indicates that the value for a particular state under some policy π is given by the state reward plus the discounted expected value of all possible subsequent states.

This paper uses the formal definition of an MDP as found in [6], an abridged version is repeated here for convenience and completeness. An MDP is a tuple $\langle S, A, R, P \rangle$ where

- S is a set of possible states
- A is a set of actions, $A(s)$ denotes the possible actions at state s
- $R : S \times A \to \mathbb{R}$ is the reward function that maps a state-action pair to a real number
- $P : S \times A \times S \to \mathbb{R}$ is a stochastic transition function where $P(s, a, s')$ denotes the probability of transitioning from state s to s' with an action a.

Because TDL is a model-free algorithm, it isn't necessary for it to have a priori knowledge of the transition function or the reward function. In some cases, it may be possible to express the reward function explicitly, e.g. as a lookup table. In the case of *2048*, the state space makes this solution infeasible. Instead, TDL can be modified to use an approximation of the value function. In some particular cases, it may be more appropriate to enable decision making through a function of state-action pairs (as opposed to just states); this is an extension of TDL known as Q-learning. Q-learning uses a function $Q(s, a) : S \times A \to \mathbb{R}$ instead of simply a state value function. A further discussion on Q-learning isn't necessary as an understanding of the approach in this study only requires knowledge of TDL.

2 Related Work

More classical approaches have been taken to create agents for this game, namely Minimax, Expectimax, and Monte-Carlo tree search (MCTS). For instance, [4] found that averaged depth-limited search (ADLS) and MCTS can exceed average scores of 50,0000 and 120,000 respectively. However, they found that in particular, these agents worked quite well for maintaining high average scores but failed to produce large individual scores. Specifically, an increase in the search limit for ADLS would reduce the number of low-scores, but would also reduce the number of high-scores.

[6] used temporal difference learning (TDL) with an n-tuple network (the n-tuple network allows approximation of the state value function much like parameterized linear functions) to create an agent for this game that performed exceedingly well. In over 97% of plays, the agent reached the 2048

tile and had an average score of 100,178 and a maximum of 261,526. The additional benefit of this technique is that it can make decisions far quicker than the more classical approaches. In particular, [6] found that both Expectimax and Minimax performed exceedingly slower. The implementation of Expectimax is capable of 6.6 moves per second compared to 330,000 moves per second by TDL. Nonetheless, TDL still suffers from the same issue as the agents in [4], it struggles to reach large scores as pointed out in [8] and [7]. [8] proposes an improvement upon the implementation of TDL by [6]; this extension is called multistage temporal difference learning (MS-TD). The authors equate it to hierarchical reinforcement learning methods. In essence, MS-TD breaks up the training process into three distinct stages, each containing their own weights. The intuition here is that each stage presents a stronger (and different) challenge and is not as forgiving with the agent's actions as it would be in the early stages. The intuition indeed turned out to be correct, this extension increased the rate at which the agent achieved higher scores. Further improvements were made upon this to reinforce it with expectimax to yield even higher rates of reaching high-valued tiles.

Some studies have been done using deep reinforcement learning as well, such as in [2]. However, these tend to pale in a performance comparison to TDL methods.

3 Motivation and Objective

2048 can be modeled as a Markovian Decision Process (MDP); this is a framework to facilitate decision making in a (discrete) time stochastic environment. The motivation to select TDL for this study is largely arbitrary but is also due to the fact that it is a model-free reinforcement learning solution to decision making in such an environment. This allows the agent to not need a priori knowledge of the transition and reward function.

While there is a fair amount of research on reinforcement learning and TDL for *2048*, very few of these studies with TDL appear to discuss ways in which to combat the large state space. Although these techniques may be sub-optimal compared to the results discussed in the related work section, it can be quite illuminating to discover how these variants perform and perhaps why. TDL is not normally equipped to handle such large state spaces and so this paper focuses on particular technique known as value function approximation. Of the various techniques for this, this paper will consider linear combinations of features and neural networks. The main objective for studying these techniques is to discover the efficacy of value function approximations other than the n-tuple networks used in [6].

4 Methods

While it is possible to use plain TDL, this paper uses an extension from [6] that enables TDL to behave quite like Q-learning. Q-learning is a variant upon TDL that considers a state-action value function, as opposed to just a state value function in TDL. This should increase the accuracy in which the value function approximations can determine the expected utility of states. For both variants of the value function approximations, there is a distinct approximater for each action, denoted V_a for the respective action $a \in A$. This extension comes from [6] and is included in this study to reduce the number of variables which may cause the results to differ, allowing a more direct comparison of results.

4.1 Linear Functions of Features

In direct similarity to [6], we update the value function approximation according to Algorithm 1.

ALGORITHM 1: Update for Value Function Approximation

`Learn-Evaluation`(s, a, r, s', s'')
$v_{next} \leftarrow \arg\max_{a' \in A(s'')} V_{a'}(s'')$
$V_a(s) \leftarrow V_a(s) + \alpha(r + v_{next} - V_a(s))$

The state succeeding an action is separated into two distinct states: s' and s''. s' denotes the state after an action has taken place but before a tile is randomly generated. s'' denotes the state after an action and a random tile has been generated. This design comes from [6], though it is not utilized here.

The parameterized linear functions take form as

$$V_a(s) = \sum_i w_{ai}^T f_i(s) = \boldsymbol{w}_a^T \Phi(s) \tag{4.1}$$

where $\boldsymbol{w}_a$ is a weight vector for the value function approximater for action a. The feature vector, $\Phi(s) = \langle f_1(s), f_2(s), \ldots, f_n(s) \rangle$, is simply the value of each game tile (e.g. $f_1(s)$ would be the value of the first tile). The motivation for selecting this is that it should be sufficient to characterize states according to common heuristics. A higher weight could be given to high value tiles being held in the corner and in adjacent tiles. That is, keeping large valued tiles outside of a corner or small valued tiles in a corner, would contribute little and would be biased against by the value function approximation. This is a common heuristic employed to improve search based agents.

It's important to note that while the value function satisfies Bellman's equation (Equation 1.2), approximating the value function in this way is not guaranteed to satisfy it [3]. This can lead to issues of converging results; this becomes apparent in the results section. There are various mitigations for this issue but including them would make the scope of this paper too broad.

4.2 Neural Networks

Just like in the previous section, four neural networks are used, one for each possible action. The activation chosen was ReLU (also called the rectifier), which is defined to be

$$f(x) = \max(0, x) \tag{4.2}$$

where x is the input to the neuron. ReLU has the advantage of faster computation over other activation functions such as $\tanh(x)$; this is the primary reason for its choice. The optimizer is stochastic gradient descent and each network has three layers: a 16 neuron input layer, a middle layer with 4 neurons, and an output layer with one neuron. The choice for the number of neurons in the middle layer comes from two factors: symmetry of the game board, and a the heuristic mentioned in the previous section. A common strategy for *2048* is to keep the highest valued tile in a corner. The symmetry of the game board means it does not matter which corner this high valued tile is in. For this reason, four neurons in the middle layer should pick up on an organization of tile values within the four four-tiled corners of the game board. Most of these decisions are based upon intuition and there are techniques for designing neural networks (e.g. as in [1]), but this falls outside the scope of this study. The particular tools used to implement and train these neural networks are models from Keras and GPU accelerated learning through CUDA. Keras is a high-level Python library for artificial intelligence applications that abstracts out the necessary, but tedious, computations needed to train neural networks (i.e. abstracts away needing to compute back propagation). CUDA is a developmental toolkit created by Nvidia that enables the exploitation of the GPU to speed up the training of models; Keras is able to take advantage of this to speed up the training process.

5 Approach

This section is largely empty due to inconclusive results (reasons for which are given in the next section). The game environment was built completely from scratch to hold control over performance. The primary metrics for evaluation were going to be milliseconds per game, minimum score, maximum score, average score, and standard deviation of average. These are common metrics in literature. Only three out of these five metrics held any meaning due to the lack of volume of data: milliseconds per game, minimum score, and maximum score. Thus, the results section will focus primarily on why there was a data scarcity and why the linear approximaters failed to converge.

6 Results

Results from the experiment were largely inconclusive, and as such, there is very little data worth being presented here. The information collected from the linear function approximater yielded inconclusive results due to the nature of the data. The results for neural networks are mostly inconclusive due to the insufficient number of training examples (the reasoning for insufficient training is discussed later). The reasons for such failures and inconclusiveness is nonetheless still discussed and various avenues of potential improvement are presented.

6.1 Linear Functions of Features

The linear combination of features performed quite poorly and diverged quickly. Multiple trials were conducted, both with the weight vectors initialized randomly and initialized to zero. In all trials, the weight vectors associated with the *left* and *up* moves quickly approached the maximum integer value and the other two weight vectors either stayed at zero or were some incoherent collection of numbers. Interestingly, this type of weighting roughly coincides with the common strategy of keeping the high value tile in a corner; which is achievable by repeatedly moving *up* and *left* and occasionally *right* or *down* as needed. There is not too much to be claimed about this particular result though it suggests a more clever feature combination may reveal such a strategy is learned by TDL. The reason for inconclusiveness is that it is not guaranteed that the linear combination of features is even able to approximate the true value function since linear approximations cannot be guaranteed to satisfy the Bellman equation [3]. This characteristic of linear approximations such as in Q-learning is also pointed out in [5]. The main reason for this is that if we consider that the value function takes the form of $\boldsymbol{w}^T\Phi(s)$, it is entirely possible that the set of basis functions chosen for the agent do not span the optimal solution (i.e. a dimension mismatch); thus, it's impossible for the estimation to approximate it. Unfortunately, this discovery was made after the experiment was completed and time did not permit exploration of solutions for this. Both [5] and [3] provide solutions that make convergence a more likely prospect (the former being the superior solution of the two). Thus, nothing of import can really be ascertained from the results of this experiment. In theory, the linear function should have no trouble adjusting itself to conform to common heuristics and it would be interesting to discover why it didn't.

6.2 Neural Networks

For the neural networks, these performed quite poorly as well but not much can be claimed since the number of games used in training was less than a thousand. ReLU was chosen as the activation function for the neural networks to introduce non-linearity into the function approximation (in

addition to the reasons given in section 4.2). This was largely motivated by the results for the linear combination of features being quite poor. If it was indeed the case that the value function was not linearly approximable, a non-linear approximation should not be impeded by such characteristics. Training the neural networks adequately would have taken on the order of weeks and was thus not feasible given the time constraints (assuming training for $\approx 100,000+$ games). However, testing the neural networks decision making speeds proved somewhat illuminating. In comparison to [6], which found a game could be completed in 23ms, the neural networks often took 20+s to finish a single game (a more accurate comparison should include the scores achieved in the time frame but such data isn't available). Unfortunately, the hardware differences are unknown and thus it is impossible to make a claim about the efficacy of neural networks either. Nonetheless, even such a small training sample still enabled the agent to reach scores over $3,000$, an improvement over random agents which have been shown to never reach the 256 tile (a score of around $2,000$) [2].[3]

7 Conclusion and Future Work

The data from experiments was very minimal and not particularly meaningful due to lack of volume. Not much can be claimed about the results from the linear functions, despite the experiments running successfully. More investigation is needed into the convergence of linear approximations to make a claim about their viability as a value function approximator for *2048*.

The same holds mostly true for the implementation of neural networks as function approximaters, though it is difficult to make claims about the efficacy of this solution in comparison to other results due to potential hardware differences. One semi-conclusive fact is that a neural networks decision making speed is comparable to that of classical approaches. However, more testing would be needed to declare confidence in this result. The neural networks as an approximater nonetheless still achieved a higher maximum score than a random agent. This particularly motivates interest into non-linear functions for approximating the value function.

For future improvements, it would be important to take a closer look at linear function approximaters since the results here suggest something of interest, though it is not concrete. Perhaps improvements to the stability of convergence for linear function approximaters as discussed in [5] could be implemented and analyzed to discover particularly why this implementation failed. It would also be interesting to explore hierarchical solutions similar to [8] (see section 2) that use a linear function, assuming the linear function approximaters were discovered to work. Admittedly, more advanced techniques could've been chosen to increase the probability of convergence of the functions, but it initially didn't seem likely that they would fail to converge; hence, improvements upon convergence were not initially investigated.

There are many avenues for experimentation and improvements with neural networks that time did not permit. Perhaps the most obvious among these is fine-tuning the layers of the network and choosing the activation function. It's also possible to use just a single network with an additional input neuron containing the action.

[3] Of course, given enough trials, a random agent would eventually reach such a tile but it's exceedingly rare.

References

[1] Nirmal K Bose and Amulya K Garga. Neural network design using voronoi diagrams. *IEEE Transactions on Neural Networks*, 4(5):778–787, 1993.

[2] Antoine Dedieu and Jonathan Amar. Deep reinforcement learning for 2048. In *Conference on Neural Information Processing Systems (NIPS), 31st, Long Beach, CA, USA*, 2017.

[3] J Zico Kolter and Andrew Y Ng. Regularization and feature selection in least-squares temporal difference learning. In *Proceedings of the 26th annual international conference on machine learning*, pages 521–528, 2009.

[4] Philip Rodgers and John Levine. An investigation into 2048 ai strategies. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–2. IEEE, 2014.

[5] Richard S Sutton, Hamid Reza Maei, Doina Precup, Shalabh Bhatnagar, David Silver, Csaba Szepesvári, and Eric Wiewiora. Fast gradient-descent methods for temporal-difference learning with linear function approximation. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 993–1000, 2009.

[6] Marcin Szubert and Wojciech Jaśkowski. Temporal difference learning of n-tuple networks for the game 2048. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8. IEEE, 2014.

[7] I-Chen Wu, Kun-Hao Yeh, Chao-Chin Liang, Chia-Chuan Chang, and Han Chiang. Multi-stage temporal difference learning for 2048. In *International Conference on Technologies and Applications of Artificial Intelligence*, pages 366–378. Springer, 2014.

[8] Kun-Hao Yeh, I-Chen Wu, Chu-Hsuan Hsueh, Chia-Chuan Chang, Chao-Chin Liang, and Han Chiang. Multistage temporal difference learning for 2048-like games. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(4):369–380, 2016.