Reinforcement Learning for 2048

Michael Baluja

March 15, 2021

Abstract

Reinforcement learning (RL) is utilized in many applications in order to solve problems not easily undertaken by supervised or unsupervised machine learning methods. We utilize reinforcement learning with neural network-based nonlinear function approximation in this paper to play the game 2048 using various reinforcement learning algorithms, and discuss the results after training models and algorithms. We report both the merits and deficits present in the use of the methods implemented.

1 Introduction

Reinforcement learning is utilized in many applications in order to solve problems not easily undertaken by supervised or unsupervised machine learning methods. While supervised machine learning is focused on classification and regression tasks, and unsupervised learning focuses on uncovering structure, reinforcement learning focuses on how to act in environments in order to maximize a scalar reward value. Due to this versatility, reinforcement learning has been used in many instances to learn to play board games and video games. In particular, the use of reinforcement learning has been used in gameplay in feats such as AlphaGo defeating human professional players [2]. In a similar, but less impressive scope, we look to utilize reinforcement learing in this paper to learn how to play the game 2048.

In the game 2048, the game board is a 4x4 grid with tiles that each contain a number $2^k, k \geq 1$. Players choose directions to shift the numbers on the board. If two like-numbered tiles are shifted into each other, then the tiles combine to produce a new tile of their sum. If a valid move is made, that is, if any tiles shift position or combine due to a player action, then a new tile containing either a 2 or a 4 is added randomly to an open position on the game board.

For this game, the end goal is to create the largest possible numbered-tile before the game board is filled with pieces that are unable to be moved or combined. In this sense, the player's reward is directly proportional to the size of the tiles that have been created; creating a tile of sum z rewards the player with z points.

Playing the game consists of moving left, right, up, and down, which respectively shifts all tiles in those directions. We look to understand if our reinforcement learning implementation can manipulate these actions in our simulated environment to achieve human-level agent performance.

2 Related Literature

The most prominent literature in relation to our work was completed by Dedieu et al., of MIT [1]. While the authors of this paper were unable to reach a tile sum of 2048, they did reach tile sums of 1024 on multiple occasions. Although this is a vital step towards "solving" the game, we choose different implementation in hopes to reach a different, and potentially better, outcome. In this regard, the authors of this original paper utilize Deep Reinforcement Learning with Monte Carlo Tree Search in order to create value estimates. In contrast, we utilize both Temporal Difference and Monte Carlo Control methods with neural network function approximation. This method helps to show how the generalization of function approximation is vital for large state-space problems such as this. Additionally, this method allows us to utilize both bootstrapping and sample-trajectory learning.

3 Methodology

In this paper, we utilize function approximation control methods in order to approximate solutions and values to our problem. As we don't know the true value function to help approximate our estimates, we utilize two different methods to create approximations of the true value function. Namely, we implement function approximation with Gradient Monte Carlo and Semi-Gradient SARSA(0), both utilizing a custom artificial neural network as their value function, as recommended.

3.1 RL Methods

3.1.1 Gradient Monte Carlo (MC)

Monte Carlo Control is an attractive algorithm for the problem we face due to the ability to form updates based on actual trajectories, and focus on states that are actually reached in the game. However, the downfall of the tabular method lies in the size of the state space for our problem. For a 4x4 gameboard with tiles that can have a number in $\{0,\ 2^k, k \geq 1\}$, simply allowing our game to only be played until 2048 will command a state-space of roughly 11^{16} states, and our implementation allows higher numbers as well, further ballooning the state-space. For our implementation, the state space contains roughly 16^{16} states, which would not allow us to even attempt utilizing tabular reinforcement learning without some form of adaptation. For this reason, we utilize Monte Carlo Control with function approximation in order to find solutions without worrying about the size of the state-space. To do this, we implement the Gradient Monte Carlo Algorithm presented on page 202 of the course textbook, and added an ϵ-greedy policy that selects actions based on our function-based action-value function [4]. While this modification is not true Monte Carlo Control, the change is made in order to avoid potential errors from creating a policy network.

3.1.2 Semi-Gradient SARSA(0) (SARSA)

With similar reasoning to the aforenoted strengths of Gradient Monte Carlo, we also implement Semi-Gradient SARSA. This approach differs in that we allow bootstrapping, and thus must use a semi-gradient approach when updating our network weights. This method is attractive for convergence reasons, as it does not require playing out the entire trajectory before updating the network. We use a similar action selection approach as mentioned earlier in order to implement this algorithm. We further modify this approach by integrating a replay buffer with a batch size of 128, which allows us to better train our network. This works by passing in a random sample of experiences to update the network on, instead of continually passing in highly correlated experiences [3]. Due to this modification, our network should theoretically perform better.

In addition to the modified ϵ-greedy action selection based on our action-value function, we slightly modify the pseudo-code given in the text by making use of built-in gradient descent and loss calculations. However, we note that these modifications should not change the underlying mechanics of our algorithms.

0	0	0	0
0	0	0	0
0	0	0	0
2	2	0	0

Table 1: Starting State

0	0	0	0
0	2	0	0
0	0	0	2
128	128	8	4

Table 2: Learning State

3.2 Setup Specifications

3.2.1 Environment and States

The environment for our 2048 implementation consists of a 4x4 grid of numbered tiles. To start each episode, we return the state seen in Table 1. Although our function approximation approach allows us to solve larger state-space problems with fewer worries, we still choose this static starting state in order to aid convergence. Upon successful actions, a new tile numbered either 2 or 4 occurs equiprobably at any 0-numbered tile location on the gameboard. The returned state consists of the modified 4x4 grid. If an unsuccessful action is taken, the environment ignores this, and the player makes another action. This was done to ensure that our player does not get trapped in some part of the state space until a nongreedy action is selected. Due to this, the agent will visit a state after every successful action (which will tend towards being every action after enough episodes).

3.2.2 Actions

The agent/player is capable of moving up, down, left, and right. These actions respectively move and combine all applicable tiles in that direction. If a valid action is taken, a new tile appears as described above. We utilize an ϵ-greedy action selection based on our value function, with varying time-based decaying ϵ to allow exploration at first, but still ensure a high level of exploitation in the long run.

3.2.3 Rewards

Rewards are given based on the sum of combined tiles, as is done in the original game. For example,

combining a 2 and 2 as well as a 4 and 4 will return a reward of 12, (2 + 2 + 4 + 4). If no tiles are combined, no reward is given. In this sense, our agent receives a reward after every action, but will not necessarily receive a non-zero reward after every action.

3.3 Neural Network

We implement a simple feed-forward network to act as our value function. The network, ValueNet, has two hidden layers, which will allow some level of interaction between the tile locations on our gameboard, while only requiring a small number of weights to adjust. The input size of our network is 1024 (16 tiles * 16 one-hot powers of two * 4 actions), with the output being a single value. Further, we utilize the L1 loss function to imitate the standard update rule we have learned throughout the course, and utilize PyTorch's SGD Optimizer to handle our gradient descent calculations, with momentum of 0.2.

4 Results

We run both our MC and SARSA algorithms over 1000 training cycles and 500 cycles, respectively. While this difference in training length hinders us from directly comparing the results, the effect from our replay buffer on SARSA will allow the network to train faster, as it updates on more samples comparatively. Due to this, some level of comparison is fair. Both algorithms are ran with $\alpha = 0.0001$, $\epsilon_0 = 0.8$ (with a 0.9 decay, 0.01 minimum threshold), and momentum of 0.2. The maximum tile number is recorded after every episode, and the value of our learning state (as shown in Table 2) is recorded after every ten iterations to show the rate at which our network is learning. The results of both of these are seen respectively in Tables 1, 2 and 3, 4.

Analyzing the results of Tables 1 and 2, we note that our Monte Carlo implementation appears to be outperforming the SARSA implementation with higher frequency of 256 pieces, though the higher rewards do not come until after the 500th episode. Due to this, we are unable to draw any conclusions on the different rate at which these algorithms learn. While computational runtime issues prevented longer SARSA trials, we believe that additional runtime would allow us to better see the learning capabilities of the SARSA implementation.

We first note for Tables 3 and 4 that the episode is taken mod 10. Analyzing the results of Tables 3 and 4, we see that there does appear to be a difference in our algorithms in terms of learning capacity. While

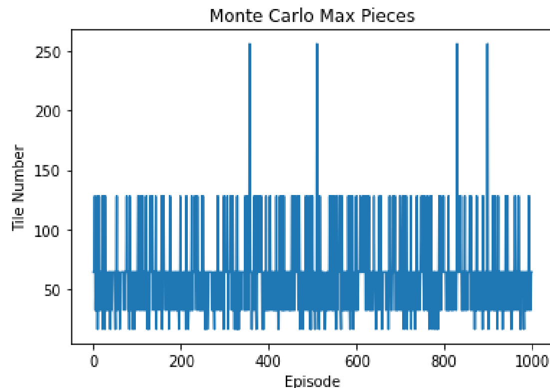

Figure 1

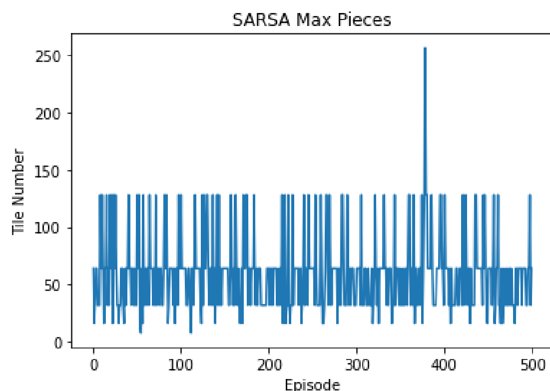

Figure 2

the SARSA algorithm utilizes a replay buffer to train from multiple uncorrelated states at once, the Monte Carlo algorithm trains an entire episode of states. Although these states are correlated, the non-singular batch length allows us to better converge to values. In this regard, it appears that the correlated nature of the states actually tends to aid learning.

5 Discussion

5.1 Implementation Ideas

For this project, we note multiple different techniques used to attempt to create working algorithms. First in this respect is the rate parameters, namely the α (alpha), ϵ, (epsilon) and decay, and γ (gamma) parameters used for updating our value estimates (used as the neural network learning rate), deciding on actions to take, and discounting rewards, respectively. We initially utilize different α values between 0.0001 to 0.8, but did not find any benefit from values over

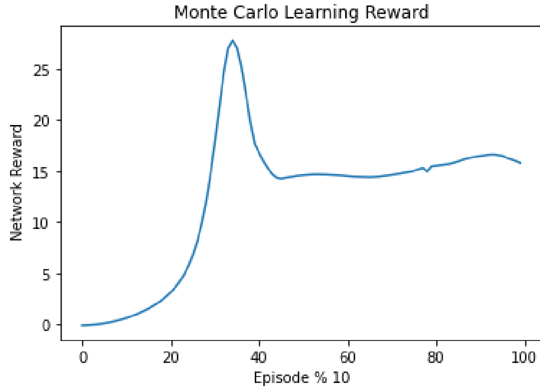

Figure 3

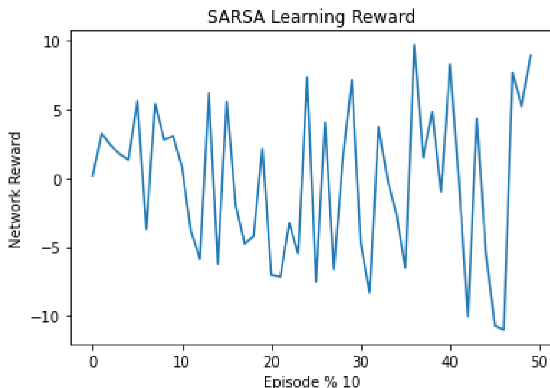

Figure 4

0.2. While a larger α value may have been beneficial for converging with low-frequency state action pairs, we ultimately decided to utilize a smaller α value in order to avoid jumping around too much when performing gradient descent, as the returned states are not deterministic. We similarly utilize a handful of ϵ values and decided on a decaying epsilon value to encourage exploration at first when the network is not as useful, but to slowly encourage exploitation more when the network has been trained. We additionally decided to keep γ constant at a value of 1, as the setup of the game appears to perform well for undiscounted learning. Small changes in momentum were also made, but we heuristically decide on a small momentum value of 0.2.

Additionally, we have tried training with adjusted reward signals. In addition to our tile sum-based reward, we implemented and tested negative rewards for each invalid move, with both constant rewards of -1 as well as variable rewards based on the size of greatest tile number. Both of these methods were abandoned in favor of removing the ability to count invalid moves.

5.2 Merits

In terms of strengths, we highlight the environment utilized for this project, and note that it has performed incredibly well and without error for all noted states and actions. While it can be difficult to reverse-engineer a game, this project does so properly, and additionally makes note of the relevant states, actions, rewards, and other gameplay variables. Overall, this project has been a great learning opportunity, and a great opportunity to showcase the knowledge of reinforcement learning that has been accumulated throughout this course.

For the algorithms, a major strength is the ability for our Monte Carlo implementation to learn to some degree. Additionally, we believe that the SARSA-based implementation for this game has a strong foundation for being able to learn to play 2048 further, but we simply did not have enough time to train the networks in depth and properly tune the network hyperparameters for the task at hand.

5.3 Deficiencies

First and foremost, the overshadowing deficiency at hand is with regard to the efficacy of the implementations used in this paper; the algorithms utilized did not successfully solve the task we sought out to solve. As multiple different algorithms (Gradient Monte Carlo/Monte Carlo Control, Semi-Gradient

TD(0), Semi-Gradient SARSA with and without Replay Buffer) were tested, we believe that the issue in this paper is with regard to the neural network implemented for the non-linear value function approximation. While the network architecture and implementation is relatively standard, we believe that this issue is caused by the computational expense required to properly train our model with proper parameters. For instance, we recognize that a small ϵ value is beneficial, but lowering the value also increases training time, as we now have to run more examples through our network to get the best action from it. In this regard, additional computation time would likely serve this project well.

6 Contributions

All implementation code has been independently written, with small bits loosely based on online guides (such as the neural network code) [4], [5].

7 Additional Information

Project code repository is available here:
`https://github.com/michaelbaluja/rl-2048`
Project video review is available here:
`https://youtu.be/mGs0tmc8yNE`

References

[1] A. Dedieu, "Deep Reinforcement Learning for 2048", MIT. Available: http://www.mit.edu/people/adedieu/pdf/2048.pdf [Accessed 8 March 2021].

[2] D. Silver et al., "Mastering the game of Go with deep neural networks and tree search", Nature, vol. 529, no. 7587, pp. 484-489, 2016. Available: 10.1038/nature16961 [Accessed 8 March 2021].

[3] "Reinforcement Learning (DQN) Tutorial", PyTorch, 2020. [Online]. Available: https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html

[4] R. S. Sutton and A. Barto, Reinforcement learning: an introduction. Cambridge, MA: The MIT Press, 2018.

[5] "yunjey/pytorch-tutorial", GitHub, 2021. [Online]. Available: https://github.com/yunjey/pytorch-tutorial/blob/master/tutorials/01-basics/feedforward_neural_network/main.py. [Accessed: 15- Mar- 2021].