Using Reinforcement Learning to Teach an Agent to Play 2048

Alexandra Walker
University of Colorado, Colorado Springs
1420 Austin Bluffs Pkwy
Colorado Springs, CO 80918
Awalke17@uccs.edu

Abstract

This paper will propose a test for Reinforcement Learning by implementing the Markov method and experimenting with Q-learning and Monte-Carlo Search tree algorithms on 2048. Reinforcement Learning with the Markov process requires a reward policy. 2048 has several different strategies and scoring systems inherently, so testing each method with different reward systems will be interesting. This process plans on reporting results based on learning and search algorithms combined with reward policies and recording the best win or lose state for each.

Introduction

The game 2048 is deceptively simple; by using your keyboard, mouse, or finger, the player moves numbers on a grid to slide in the direction chosen. Human players will discover some strategies that make it easier to win. The agent will be given the chance to develop complex strategies by playing 2048. The goal of the agent will be to win 2048. The goal of this paper is to discover what rewards and algorithms optimize the agent's learning.
The reward set will experiment with rewarding behaviors like keeping high numbers in a corner versus rewarding the combining of numbers or the number of empty spaces on the board. The reward also rewarded a combination of behaviors and actions.

This paper will explore two learning algorithms along with the reward system proposed. Monte Carlo Search Tree and Q-learning. Nathaniel Hoeanwan and Wu (2020), (Kaundinya et al. 2018), and (Wang, Emmerich, and Plaat 2018) and (Chentsov and Belyaev 2020) all explore this idea with varying results based on their different approaches. By combining results and making adjustments to the reward system to see if improved results can be achieved,

Reinforcement learning is described in Sutton and Barto (2020) as "Learning what to do—how to map situations to actions—so as to maximize a numerical reward signal." This reward signal is a reward sent by the environment to the agent. This reward is then summed up and evaluated using different methods. Over the course of a single episode of learning, an agent can be rewarded several times or a single time. How the user chooses to define and distribute the reward can directly affect how the agent performs and the optimal policy created.

The Markov decision process will be implemented and used to develop the agent's policy based on its environment and rewards. Due to the nature of 2048, every state contains a summary of all moves before it. For example, if you have a 16 tile on the board, then at some point in the game the player combined 2 eights, 4 fours, and somewhere between 2 and 8 twos. Due to the random nature of the game, 8 twos is not guaranteed. It's unlikely every single 4 on the board was created by the player. Therefore, 2048 will have the Markov property.

The project focuses on implementing the agent and the environment in Python using a game class for the environment and an agent class. The agent will have the set of actions Up, Down, Left, Right to choose from. As humans have access to the entire environment at all times, the agent will as well.

Background

Playing video games with AI has been around for about as long as video games have been around. Playing games with AI has been around longer than video games. Since then new techniques have been experimented with and tested and revised. General Game Playing, a technique that describes one Agent being taught how to play several games, is a well-discussed and researched field. Atari games are often used as a baseline example of these kinds of AIs(Mnih et al. 2013). Reinforcement learning and deep reinforcement learning are the newest innovations in the field often beating human players.

Vocabulary

In this paper, there are several similar concepts, so defining each concept is useful.

- A **spawned number** is a value placed on the board randomly; this number can be a 2 or a 4.
- A **combined number** is the value created when two numbers are combined.

- The **highest number** is the number on the board that is greater or equal to all other values placed.

2048

2048 is a 1-player game that was released in 2014 and created by Gabriele Cirulli. The game is played by using arrow keys to choose a direction. The game slides all the numbers on the board in that direction. If two numbers are the same, then the game will combine them, increasing the score. For example, if 2 fours are combined, then the game increases the score by 8. 2048 will only have numbers on the board consisting of powers of two, also known as 2, 4, 8 . . . 1024, 2048. By combining two 1024s, the player will win and then be prompted if they want to continue. If the player does choose to continue the game, then enters an endless mode where there is no win condition and the challenge changes to simply trying to get the highest score possible. Due to the size of the board, the highest possible tile in 2048 is 131,072, as there is no more space to reach a higher value, and it is considered the ultimate win state.
2048 can only be lost when the player leaves no more room for a new number to spawn and the player cannot make any more moves. 2048 can be seen as a Markov Game (Littman 1994). Markov Games are games that can follow a strict pattern of learning. The Agent will perform be given a state, perform an action, and be given a reward. 2048 follows this process down to the letter allowing for MDP to be implemented easily. The state as discussed before is the board itself, however, as the board has a large state space simplifying the states is best. The actions are of course the four cardinal directions, and the rewards are given by the game after each direction.

Strategies

2048 has a few easy strategies that without make the game more difficult. The first strategy can be called three directions. The player picks three directions they will use mainly, and the fourth will only be used when in a stuck state. When all three moves do not allow for the game to be progressed in any way. The second is to simply keep all the numbers on one wall, with the highest value in one corner and each subsequent highest value moving along that wall in descending order.

Literature review

Li and Peng (2021) described their rewarding system as directly using the scoring system from the game. Their paper chose to focus more on the search algorithm for each AI design. The baseline was performing random actions, where the AI reached the highest number of 128 only 48% of the time. Their best-performing algorithm, the Beam Search, has the highest number of 2048, the win states, at 28.5% of the time and even reaching 4096 0.4% of the time.

Nathaniel Hoeanwan and Wu (2020) focused on testing the evaluation function used to determine the best agent plays. They implemented a Monte Carlo tree search to find the maximum-valued play to make. Each test prioritized a different scoring method: taking the largest sum on the board, taking the largest tile or number on the board, or taking the largest of the total score. They also chose to change the number of roll-outs for the Monte Carlo Tree Search and experimented with how well the agent played with 50, 100, 250, and 500 roll-outs.
Their results showed that taking the largest sum of the board with 500 roll-outs had the best results. Managing to make 2048 43% of the time with an average sum of the final board being 2444.58 and the average final score being 19107.28. The runner-up method uses the largest total score as the evaluation method. This method had similar results.

Chentsov and Belyaev (2020) re-imagined the Monte Carlo search tree algorithm to optimize for computer games. They chose to include a parameter in the evaluation function. This parameter would change the probability of the MCT choosing one solution over another. This parameter would be determined by a few properties:

- Location of agents and obstacles
- Previous actions

They concluded that this change would work better for games that don't depend on player location.
Kaundinya et al. (2018) attempted to create an AI to play 2048. Focusing on Q-learning and SARSA algorithms to solve the game. Their reward function was an attempt at finding what was most beneficial for the agent. They attempted three different functions:

- Rewarding based on strategy,
- Rewarding based on their experience replay for a game over a threshold, and
- Using the whole game for their experience replay and training the neural network. Also using the score as a basic reward.

All three of their attempts were deemed unsuccessful, as their AI was never able to reach 2048. However, it's useful to understand where they failed. Their recommendation was to keep rewards between -1 and 1 based on the score. They also recommended using Q-architecture instead of Q-learning.

Wang, Emmerich, and Plaat (2018) explores the field of General Game Playing (GGP). Focusing on Q-learning and Q-learning with Monte Carlo Lookahead. The authors found that Q-learning does converge for real games like Hex, Tic-Tac-Toe, and Connect Four however, the convergence is slow. When exploring Q-learning with Monte Carlo Lookahead the convergence was significantly faster, especially in the early game.

Approach

By implementing 2048 as an RL agent utilizing the Markov decision process as compiled by Van Otterlo(2009). The agent can learn from each game played, trying to achieve a better score with each attempt. This paper will explore changing the reward system between the score and the highest tile. Then compare how the agent does with MCST

and without. These results will be compared to q-learning. Littman(1994) expands on the process of agents playing games to include multi-agent games. In these games, agent 1 will be in a state, and make an action and the second agent will respond with their own action. Some papers have described 2048 to be a 2 agent game where the environment is a mindless random agent and the player is the second player. The results of Littman show that an 'intelligent' agent significantly out performs the random agent. This is an interesting addition to the 2048 problem, however as the second player is random this paper will treat the problem as a single agent.

Environment State Design

2048 has a very large state space, with about 281 trillion combinations (Sijtsma, Jansen, and Vaandrager 2020) so simplifying down the state space was vital. This results in an interesting issue with the policy. If the policy can only train a state by reaching that state and for that state a specific action, the likelihood of being able to train a substantial number of states well is unlikely. Simplifying the state to only moves that result in a reward, with the move that resulted in a simpler reward evaluation. While this method may have some drawbacks, it will provide enough information for the agent to learn and make intelligent decisions. The algorithm to find the states is defined in Algorithm 1. A Python dictionary is returned in the format (row, column): ([list of directions], reward). For example, given a board like

```
2    2     8     4
_    4     2     16
2    4     32    2
4    16    64    16
```

the resulting state will be (0, 0): (['right', 'down'], 2), (1, 1): (['down','left'], 4),(2,0):(["up'], 2). This state is a simplistic representation of all the moves that will result in a reward. This state shows that in row 0 column 0 if the agent plays right a reward of 2 will be gained. Additionally, in row 1 column 1 if the agent plays down a reward of 4 will be given. By representing the state like this, the agent will more easily see what moves result in a greater reward. The number of states will not be decreased with this method, but the representation of a state is significantly simpler. Also solving the issue of a lack of training data. Now any two in the same location with a move available to it will add to the training data.

Baselines

To compare the improvement of the agent a baseline was used. The first baseline test was a simple random-direction algorithm. The agent will randomly choose from up, down, left, and right. Each direction has a random chance of being chosen, 25%.

The next baseline test was simply forcing the agent to prefer one wall and forgo one direction, unless necessary. As shown in algorithm 2 the agent will first choose between left, right, and down. If the move was invalid then the agent must keep choosing moves until a valid move is found. While inefficient this method maintains the purity of no intelligence and keeps all choices completely random.

Algorithm 1: Defining state information

$board = deepcopy(ActiveBoard)$
$state = \{\}$ ▷ A python dictionary
for each row and column in board **do**
  $origin = board[row][col]$
  if original is nonempty **then**
    $state[row, col] = ([], original)$
    for i in [1,2,3] **do**
      if row $\pm$ 1 is inside board and row - i ¿= 0 **then**
        $check =$
        $isTileEq(origin, board[row \pm 1][col])$
        $iszero(0, board[row \pm 1][col])$
        if check is true **then**
          $states[row, col][0].append(direction)$
        else if zero is true **then**
          $states[row, col][0].append(Oposite_dir)$
        else if neither zero or check is true **then**
          break
        end if
      end if
      if col $\pm$ 1 is in board **then**
        $check =$
        $isTileEqu(origin, board[row][col \pm 1])$
        $iszero(0, board[row \pm 1][col])$
        if check is true **then**
          $states[row, col][0].append(direction)$
        else if zero is true **then**
          $states[row, col][0].append(Oposite_dir)$
        else if neither zero or check is true **then**
          break
        end if
      end if
    end for
  end if
end for

Algorithm 2: Prefer down

Require: Gameboard, player
$choice = player.pick([left, right, down])$
if choice is valid move **then**
  continue . . .
else
  while Choice not valid **do**
    $choice = player.pick([left, right, down])$
  end while
end if

Monte Carlo RL algorithm

The Algorithm used to update the qvalues and update the policy is: Algorithm 3. To actually make the choice of which state to use and what direction to go was made by a separate function. The states were ordered by which move was the highest value used the policy to decide whether or not to choose that option.

Algorithm 3: Monte Carlo

```
Require: DR = discountRate
  for episode in AllEpisodes do
    for state in episode do
      tileLoc in state
      Reward in AllReward +1
      N(state, a) += 1
      q(s,a)+ = DR(1/N(s,a) * (R − q(s,a)))
    end for
  end for
  for tile in qvalues do
    for dir in tile do
      if dir == 0 then directionSum += 1
      elsedirectionSum = dir
      end if
    end for
    for i in [0,1,2,3] do
      if tile[i] == 0 then policy(s,a, dir) = 1/direction-
Sum
      elsepolicy(s,a,dir) = tile[i]/directionSum
      end if
    end for
  end for
```

QLearning Algorithm

The algorithm used for the qLearning algorithm is: Algorithm 4 (Watkins and Dayan 1992). The discount rate used was .98 and the Learning rate was .85. The move function first organized each state by highest value the used the qvalues to determine whether or not to choose that state.

Algorithm 4: QLearning

```
CurrentLocation = StateChosenLocation
oppositeAction is opposite to the action taken.
NextLocation = State after action
q(s,a)+ = LR(R + DF * max(q(NextLocation)) −
q(NextLocation, Naction))
```

Reward Policies

The reward policies are the main part of this paper, testing which reward would prove to be more strategic. As in other papers the reward to the algorithm tends to be the score gained when merging, the final score at the end of an episode, the highest tile on the board at the end of a episode or the final sum on the board. This paper plans on testing whether or not counting the number of blank spaces on the board is a valid option.

Algorithm 5: Empty Spaces

```
reward = 0
for tile in Board do
    if tile != 0 then
        reward += 1
    end if
end for
if Board[3][3] == HighestTile then reward += 16
end if
```

Algorithm 5 is later referred to by 'Empty'. This Algorithm counts the number of blank spaces on the board and rewards 1 for each. This also rewards having the highest tile in the bottom right corner.

Algorithm 6: Empty Spaces

```
reward = 0
for state in CurrentStates do
    if M thenove is in stateValidSctions reward += score-
Delta * (1+row)*(1+col)
    end if
end for
if Board[3][3] == HighestTile then reward += Highest-
TileValue
end if
```

Algorithm 6 is late referred to by 'Score'. This algorithm simply rewards combining the highest value merge and the merge closest to the bottom left hand corner. This also encourages having the highest tile in the bottom right hand corner.

Both algorithms encourage moving down as was proven by the 'Prefer Down' baseline algorithm to be a successful strategy.

Policy Methodology

Once the state information can be properly stored and evaluated making the policy is the next step. The policy will be some array describing the best action in any state. For example, give a state where a pair of twos and a pair of 32s are available the program should choose to merge the 32s over the twos. There are some difficulties and challenges with this methodology. 1) How to encourage one state's policy option over another. 2) How to represent the policy for 2048 since it can have multiple states at once or no state. 3) When given a policy and a new episode of training how to best update that policy. Once a state is given the goal of getting the highest tile in a corner and playing with a preferred wall is best. So by adding a weight to the state, either by looking at score or location or both, the policy can determine what is the best state to prioritize. Next, keeping track of the moves that best achieve the 'Preferred down' goal or get a higher score will be the main choice of the policy. Of course, the issue of exploitation versus exploration will need to be assessed, and tweaking weights and goals will be one way of managing those issues.

Evaluation

The agent will be evaluated based on the percentage it successfully creates a 2048 tile. Other evaluation methods, like average score per game and number of moves made, will also factor into the evaluation of the agent. With Python being a notoriously slow language, as it's interpreted, it doesn't make sense to evaluate the agent based on the time played, so this paper will omit that point. Although that data may be logged and reported.

Results

The two methods are run 10,000 times then results are saved and averaged for an easy-to-read format. Each algorithm takes about 2 minutes to run with saving to a file adding another 2. The total time for running the algorithm and saving the data is about 4 minutes. The other algorithms discussed QLearning and Monte Carlo RL are only run with 1000 episodes due to each episode taking about 20 or more hours to run the full 10,000.

Baselines

Average Results of Random and Prefer Down Methods		
	Random	Prefer Down
Highest Tile	96.9592	101.776
Valid Moves	84.1473	93.9959
Invalid Moves	17.7921	18.0114
Total	101.9394	112.0073

Table 1: After 10,000 games the average highest tile achieved and the average number of valid, invalid, and total moves.

The Maximum of Random and Prefer Down Methods		
	Random	Prefer Down
Highest Tile	256	512
Valid Moves	206	318
Invalid Moves	68	74
Total	261	392

Table 2: After 10,000 games the maximum highest tile achieved and the average number of valid, invalid, and total moves ever.

The Minimum of Random and Prefer Down Methods		
	Random	Prefer Down
Highest Tile	8	8
Valid Moves	20	20
Invalid Moves	0	1
Total	24	23

Table 3: After 10,000 games the minimum highest tile achieved and the average number of valid, invalid, and total moves ever.

Table 1 represents the average number of valid moves, invalid moves, and total moves across all 10,000 games. Here Prefer Down increases the average number of valid moves and total moves Improving the length of the game, however, the number of invalid moves does increase slightly.

Table 2 represents the maximum of each category across all games. This best shows how the two methods compare even when they play at their best. The highest tile in Table 2 is slightly misleading as Random is capable of achieving 512 however in the final run it did not capture that. Otherwise Prefer Down manages to get runs over 100 moves longer than Random and only increases the invalid moves slightly.

Table 3 shows how the two methods compare when they play their worst. Here it's about the same across the board. Figure 1 gives the best representation of the differences between the two methods. Random barely manages 500 256 tiles, while Prefer Down manages to get about 20 512 tiles. This shows how implementing the one strategy of sticking to one wall or choosing only 3 of the four directions affects the game play. This information will inform future decisions on how to set up the policy.

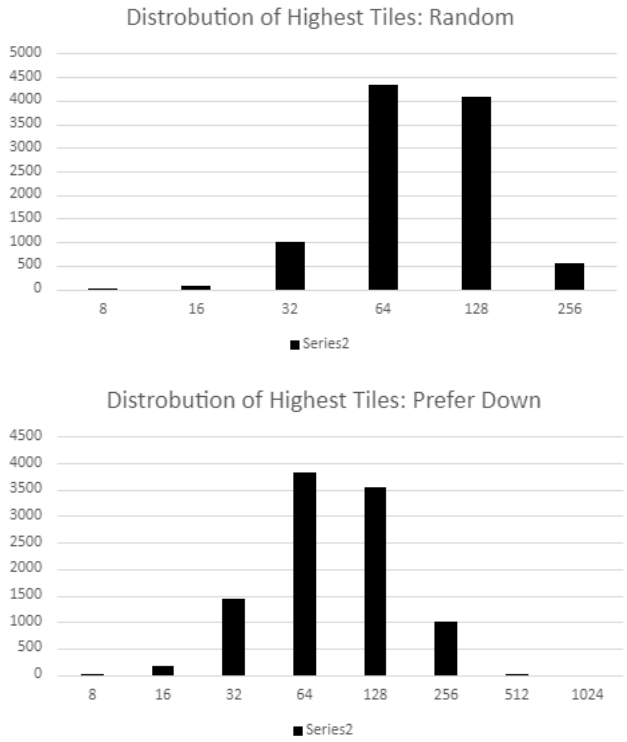

Figure 1: Top: The distribution of the highest tiles achieved before death for the Random method. Bottom: The same distribution for the Prefer Down method.

RL results

Table 4 and Table 5 and Table 6 show the results for the RL algorithms and their combinations of reward and with or without a tree.

Table 4 shows the highest tile achieved with each algorithm. The Monte Carlo algorithms are shown first then the Q-Learning algorithms are next. 'Tree' refers to whether or not the algorithm was preformed with a Monte Carlo Search Tree. 'Score' and 'Empty' refers to witch algorithm was used to reward the q-values.

Type	Highest Tile	Ave Highest Tile
MC: Tree and Score	256	92.111
MC: Score	256	75.1
MC: Tree and Empty	256	90.544
MC: Empty	256	77.472
Q: Tree and Score	256	92.92
Q: Score	256	76.108
Q: Tree and Empty	256	90.704
Q: Empty	256	74.24

Table 4: The Highest Tile achieved and the average tile achieved over the course of 1000 games

While there is not an difference in the highest tile achieved, the Ave Highest Tile does show more information. If the Algorithm was not preformed with a search tree the average highest tile was 20 less then when preformed with a tree. This is most likely a result of the nature of 2048. In the game a human player will see that the best play or win is a few moves a way and with a specific combination of moves that best score can be achieved. While an Agent can only see the next move unless a search tree is included.

Take for example:

```
2   2    8    4
_   4    2    16
2   4    32   2
4   16   64   16
```

A player may notice that by playing the sequence {down, down, left, down, right/left} a 32 can be made in the bottom left corner assuming 'good luck' with the placement of the random tiles during play. An Agent may struggle to find that sequence and execute it.

Type	% Left	% Right	% Down	% Up
MC: Tree and Score	29	28	25	17
MC: Score	30	29	26	18
MC: Tree and Empty	28	27	25	16
MC: Empty	31	29	27	19
Q: Tree and Score	29	28	25	17
Q: Score	20	26	32	6
Q: Tree and Empty	18	23	28	4
Q: Empty	22	26	32	5

Table 5: The percentage that each direction was chosen for 1000 games of each algorithm

Table 5 shows that Q-learning was best able to achieve the 'Prefer down' stratigy. It should be noted here that the Prefer Down algorithm went up less then 1% of the time. Perhaps if Q-learning is given more the 1000 episodes to train that the percentage of 'Up' may decrease further. Otherwise all algorithms did manage to prefer down if only slightly.

Type	# 64	# 128	# 256
MC: Tree and Score	431	449	16
MC: Score	526	251	7
MC: Tree and Empty	438	426	18
MC: Empty	549	281	9
Q: Tree and Score	447	443	20
Q: Score	522	274	7
Q: Tree and Empty	472	403	23
Q: Empty	532	246	9

Table 6: for 1000 Games of each type how many 64,128 and 256 were achieved as the final highest tile.

Table 6 gives a better idea of how each algorithm is truly preforming. Q-Learning out preformed Monte Carlo. This also shows that the Empty reward routinely does better then the Score reward. In future work a larger set (10,000 games) will confirm this data.

Conclusion

The Game 2048 is a strategy game with a extremely large data set. These algorithms were not able to 'learn' human strategies in this paper however they showed that Q-Learning with a search tree, and rewarding the algorithm based on the number of empty spaces may be a better algorithm then simply rewarding based on the score given.

Future Work

Exploring more rewards like only highest in the corner and snaking subsequent highest. Trying the Double Q-Learning as it improves on the overestimation problem of single Q-Learning (Hasselt 2010) or self correcting q-learning as it claims to have an even better estimation then DQN (Zhu and Rigotti 2021) algorithms could prove to have even better results. Finlay, an improved search algorithm or a different use of the search algorithm would also show better results.

Time Line

2/25/24 Have a working environment and an agent capable of making a baseline(random moves)
3/9/24 Have RL with Markov process
3/16/24 Q-learning
3/18/24 Midterm report done
4/13/24 MCTS working
4/27/24 Optimizations
5/1/24 Final report done

References

Chentsov, D. A.; and Belyaev, S. A. 2020. Monte Carlo Tree Search Modification for Computer Games. In *2020 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus)*, 252–255.

Hasselt, H. 2010. Double Q-learning. *Advances in neural information processing systems*, 23.

Kaundinya, V.; Jain, S.; Saligram, S.; Vanamala, C. K.; and B, A. 2018. Game Playing Agent for 2048 using Deep Re-

inforcement Learning. *National Conference on Image Processing, Computing, Communication, Networking and Data Analytics*.

Li, S.; and Peng, V. 2021. Playing 2048 With Reinforcement Learning. *CoRR*, abs/2110.10374.

Littman, M. L. 1994. Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, 157–163. Elsevier.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Graves, A.; Antonoglou, I.; Wierstra, D.; and Riedmiller, M. 2013. Playing Atari with Deep Reinforcement Learning. arXiv:1312.5602.

Nathaniel Hoeanwan, S. T.; and Wu, K. 2020. What's in a Game: Solving 2048 with Reinforcement Learning. *Stanford University*.

Sijtsma, J.; Jansen, N. H.; and Vaandrager, F. W. 2020. *Creating a formal model of the game 2048*. Ph.D. thesis, Ph. D. dissertation, Radboud Univ., Nijmegen, The Netherlands.

Sutton, R. S.; and Barto, A. G. 2020. *Reinforcement Learning an Introduction: Second Edition*. Westchest Publishing Services.

Van Otterlo, M. 2009. Markov decision processes: Concepts and algorithms. *Course on 'Learning and Reasoning*.

Wang, H.; Emmerich, M.; and Plaat, A. 2018. Monte Carlo Q-learning for General Game Playing. *CoRR*, abs/1802.05944.

Watkins, C. J.; and Dayan, P. 1992. Q-learning. *Machine learning*, 8: 279–292.

Zhu, R.; and Rigotti, M. 2021. Self-correcting q-learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, 11185–11192.