Reinforcement Learning for Connect Four

E. Alderton
Stanford University, Stanford, California, 94305, USA

E. Wopat
Stanford University, Stanford, California, 94305, USA

J. Koffman
Stanford University, Stanford, California, 94305, USA

This paper presents a reinforcement learning approach to the classic two-player slot and disc game Connect Four. We survey two reinforcement learning techniques—Sarsa and Q-learning— to train an agent to play Connect Four using an optimal strategy for the game. We used TensorFlow, an end-to-end open source machine learning platform to create and test deep learning models. We studied how varying exploration rate and rewards models affects performance by both algorithms. Ultimately, we found did not find a significant difference between the two algorithms, as both recorded strikingly similar win percentages against themselves and against the opposing algorithm. We also used Docker to containerize our application and source code. This allowed us to run our tests and TensorFlow software in a safe containerized approach that did not interfere with our own machines.

I. INTRODUCTION

Connect Four is a well-known two-player strategy game. Each player has their own colored discs and players alternate in selecting a slot on the board to drop the disc in with the goal of getting four of their colored discs in a row, either horizontal, vertical or diagonal. The game's roots stem from Tic-Tac-Toe, a pen and paper game with the goal of getting three in a row, as opposed to four [1]. The game first became popular under the Milton Bradley Company after it was first coined Connect Four in 1974.

Connect Four is a compelling game as although it appears simple and straightforward, there is significant opportunity to use strategy to increase and even guarantee one's likelihood of winning. For example, targeting specific slots over others is important. The slots in the middle of the board are more valuable than the ones on the edges because there is a higher chance of creating four in a row. The board and rules of the game enables a large variety of possible final boards with players able to take numerous different actions to reach them. We aimed to use reinforcement learning to find the optimal policies for the Connect Four Markov Decision Process.

II. CONNECT FOUR RULES

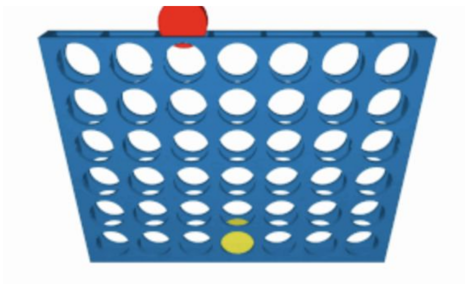

Figure 1: Connect Four board [2]

Connect Four is played on a vertical board with six rows and seven columns which totals in 42 playable positions on the board (See Figure 1). At the top vertical edge of the board, for each column there is slit where the pieces are slotted into. Once the piece has been slotted into a column it falls down the column to the lowest row or it lands the row above the piece that was last played in that column. It is a two player game and each player has twenty one coin-like pieces in a different color to the other player (player one may have all red pieces and player two all black) (See Figure 2).

The game begins by randomly deciding whether player one or player two will take the first turn. After the first player takes their turn, turns are then taken alternatively between the two players. A turn consists of a player dropping their colored piece into a column on the board. Each turn a player must drop their colored piece into a column, they cannot 'miss their turn.' If a column is full (six pieces in the column), then the piece cannot be played into that column. Additionally once the piece has been played, it cannot be undone or removed from the board.

The objective of Connect Four is to be the first player to connect four of their colored pieces in a row (either vertically, horizontally or diagonally) with no gaps on the board between the four piece (See Figure 3). At this point the game is over and the player who connected the four pieces wins. Alternatively, if all of the pieces have been played, resulting in the board being full, then the game is called a tie (See Figure 4).

Figure 2: Connect Four pieces [3]

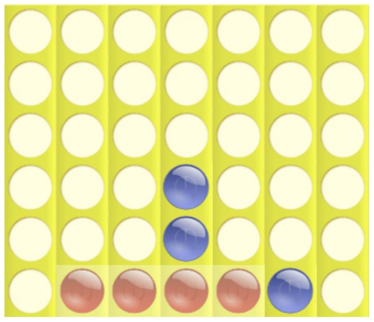
Figure 3: Winning Connect Four Board [4]

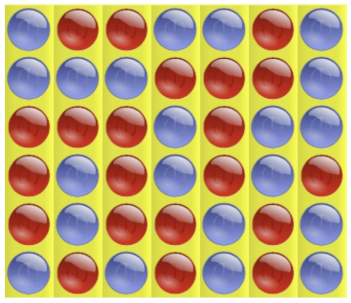
Figure 4: Tie Connect Four Board [4]

III. PROBLEM DESCRIPTION

Our project attempts to find an optimal Connect Four playing strategy given a Connect Four board. We define an optimal Connect Four playing strategy as the sequence of turn actions that maximizes the probability of a player winning a single Connect Four game.

A. Action Space

The action space for Connect Four is relatively small. Each player has at most seven possible actions that they can take at any given state. Therefore an action is defined as dropping a piece into one of the seven columns on the board. Because the number of actions a player may take at a given state depends on how many columns on the board are full, we calculate the number of actions possible at a given state as:

$$\textit{Actions Possible} = 7 - C \qquad (1)$$

$$C = \textit{Number of full columns at current state} \qquad (2)$$

B. State Space

The state space for Connect Four is considerably larger than the action space. A state in Connect Four is defined as the board with played pieces that a player sees. If the player starts the game, then the board will always have an even number of pieces. Alternatively if they are second player, the board will always have an odd number of pieces.

A very rough upper bound for the number of possible states of a connect four game can be calculated by taking into account that each position on the board can either be free, a player one piece or a player two piece. Then because the board size is 6×7, we get an upper bound of 3^{42}. However, this calculation does not take into consideration that possible boards are illegal due to gravity and the rules of the game. The best lower bound on the number of possible positions has been calculated by a computer program to be around 1.6×10^{13} [5].

IV. LITERATURE REVIEW

Scientists have already proven that if the first player places their piece in the center column, with perfect play they cannot lose. Similarly, if the first player does not place their first piece in the center column, with perfect play the second player can always draw [3]. However, given a certain state, what is the optimal move? This requires some complex calculations.

To compute the best decision each time, one can construct a game tree. Say it is player A's move. Player A has *7 - C* (see equation one) actions which can be taken. In constructing a game tree, each node from the parent state is the state arising from taking one of these actions. Then from these states, Player B can take *7 - C* actions, which lead to *7 - C* unique states, forming a subtree. This continues until an action is taken which results in the end of the game, and this state is called a leaf since there exists no further subtrees. By assigning leafs values, and calculating the entire game tree, it is possible to take the best possible action by propagating the values from the leafs and nodes all the way up the tree to the current state.

Calculating the game tree is pretty straightforward. However, there can be pretty severe run-time issues that make calculating the best decision using a game tree not feasible. Consider the problem where the game has many more moves before a winner is decided. With 7 columns, calculating n moves means storing approximately 7^n game states [6]. We see that run-time increases exponentially with the number of moves ahead to be calculated. Because of this, the focus has shifted more to creating algorithms that can calculate the best action to take very quickly and while using a small amount of memory.

One can optimize run-time with how objects are stored(e.g. using bitwise operators and bit strings instead of more expensive data structures [7]), but we gloss over these implementation details, focusing on algorithm changes. One simple algorithm is to check whether taking an action will give the opposing player a chance to connect 3 in a row some x moves down the line, where x can be manipulated by the programmer [6]. While very fast, this is shortsighted in that being in a winning or losing position in Connect Four is more nuanced than who has three in a row.

Other solutions include using a minimax algorithm, which minimizes the maximum losses for a player. This is done by assuming that the opponent is going to make the best possible decision, and therefore ignoring all other possibilities. This significantly reduces the number of states that need to be calculated, while still ensuring optimal outcome for a player [7]. However, it still takes a while, especially in the beginning of the game. That is where alpha-beta pruning comes in, a technique which prunes away the need to consider many subtrees. It involves considering whether the values in a subtree could possibly change the current action indicated by the tree, and if there is no value that could change the action then the subtree is not calculated and ignored.

V. IMPLEMENTATION

C. Docker

Docker is an open-source tool that makes it easier to create, deploy, and run applications through the use of containers. Containers isolate software from its environment which enables multiple containers to run separately, but on the same machine with isolated processes in user space [8]. We used docker to run different versions of our reinforcement learning algorithms at the same time in order to be more time efficient.

D. TensorFlow

TensorFlow is an end-to-end open source platform for machine learning. We used the core open source library to help develop and train deep learning models for Connect Four. It made it easy to work with large datasets as well as to build our deep learning reinforcement models which trained our agents to learn how to play Connect 4 against themselves and each other.

E. Simulator

Because the possible state space for Connect Four is so large, instead of gathering training data we used a simulator. In the simulator we could set up the agent we were training to play itself or another agent, we could then analyze the agents performance. To measure performance we played the different agents against each other and recorded the win rates and the number of moves per game.

F. Q-learning

Our first approach leverages Q-learning, a model-free reinforcement learning algorithm. This means that rather than using transition and reward models, we use the observed next state *s'* and reward *r*. Therefore for each observation (s, a, r, s') at time t, we perform the following incremental update rule

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r+\gamma maxaQ(s',a)-Q(s,a)) \quad (3)$$

with a learning α and discount factor γ.

G. Sarsa

For our second approach we sought to use Sarsa, an alternative to Q-learning. Unlike

Q-learning, which maximizes over all of the possible actions, Sarsa uses the actual action taken to update Q. Consequently for each observation (s, a, r, s') at time t, we perform the following incremental update rule

$$Q(s,a) \leftarrow Q(s,a) + \alpha(r+\gamma \max_a Q(s',a')-Q(s,a)) \quad (4)$$

with a learning α and discount factor γ.

VI. RESULTS

In implementing our project we decided to have two agents, one for Q-learning and another for Sarsa. Within each agent, we choose to vary both the rewards for the agent and their exploration factor. This provided us with the most reasonable approach to obtain valid results, considering that each test would run for ~5-6 minutes and limiting how many tests we could run. For the purposes of our project. a test means an agent played 1000 games of Connect 4. Whether an agent played against itself or the other agent, the specific rewards allocated, and what the exploration factor was, where all set for each test which we ran within multiple docker containers. We broke down the testing into three cases below, Q-learning vs. Q-learning, Sarsa vs. Sarsa, and finally Q-learning vs. Sarsa. From this we varied the exploration factor (05. - 0.9) and rewards (3 options).In the graphs, player 1 win percentage is calculated as follows:

*(player 1 wins + (0.5*ties))/(total games played)* (5)

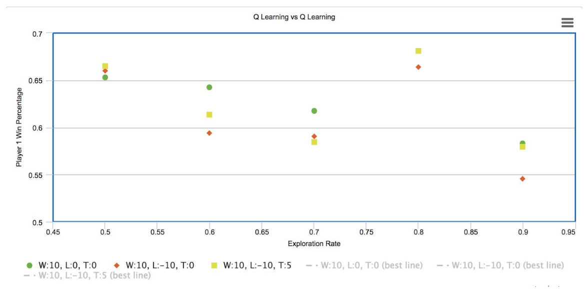

Figure 5: Q-learning.

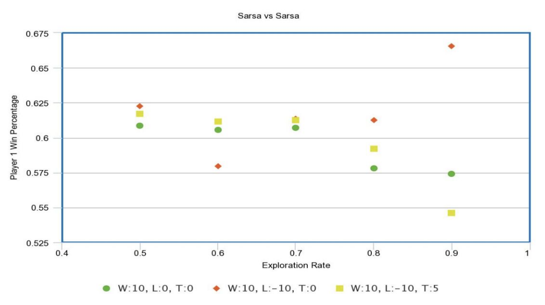

Figure 6: Sarsa.

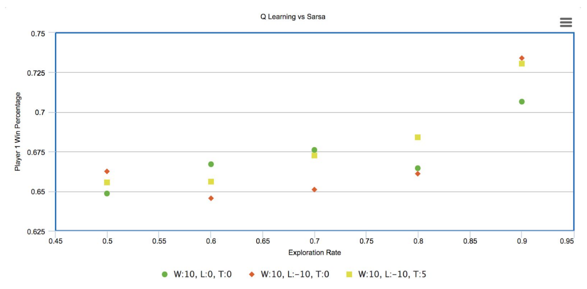

Figure 7: Q-learning (Player One) vs Sarsa (Player Two).

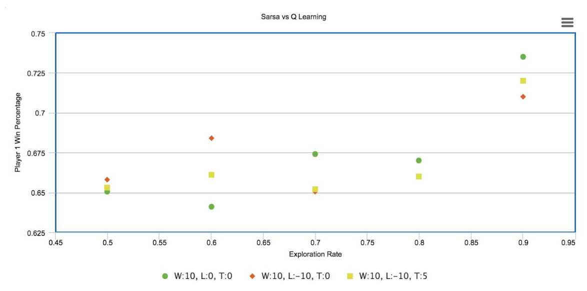

Figure 8: Sarsa (Player One) vs Q-learning (Player Two).

VII. ANALYSIS

In both our research of Connect 4 and analysis of our results we found a significant advantage to whichever player went first. Specifically, in our results we noticed that every single test run results in the majority of wins between the two agents always went to player 1, regardless of which agents were playing. This is consistent with the prior work that has been conducted on connect 4 showing that with the perfect decisions being taken, player 1 cannot lose. Player 1 does not win 100% of the time because it is not taking the optimal strategy every time, for it is learning how to play the game as it plays. However, while player 1 won the majority of games in every trial, we noticed interesting differences in these percentages across algorithm, exploration rate, and reward values.

The exploration rate appears to have the largest effect on the data. Starting with Q Learning vs Q Learning, in Figure 5 we see that there is a negative correlation between win percentage and exploration rate. There is an interesting outlier with an exploration rate of 0.8, which is actually the most successful exploration rate we tested, however the general trend is downwards. We suspect that lower exploration rates result in higher success rate because player 1 sticks to what works, and does not compromise its advantage by exploring sub optimal policies, that while they may pan out, more often give up the superior position that results from starting the game. We hypothesize that the blip at an exploration rate of 0.8 is due to a perfect combination of exploration and exploitation, where it abandons its strategy frequently enough to find more optimal solutions but not so often that it abandons successful strategies.

Sarsa vs Sarsa has the same negative correlation between exploration rate and player one win percentage that we observed with Q learning, with the difference being a lack of an outlier and a smaller spread between exploration rates (see Figure 6). We expect that the negative correlation exists for the same reasons described above. The smaller spread may be due to a poor exploration strategy for our Sarsa algorithm; if it begins as poor, a higher exploration rate has a lesser effect because the current strategy has a higher chance of being suboptimal, so switching away from it is less detrimental.

The most intriguing discovery about exploration rate was the effect that it had on Q Learning vs Sarsa (Figure 7), and similarly Sarsa vs Q Learning (Figure 8). Here, increasing exploration rate elevated the success of Q Learning over Sarsa. With an exploration rate of 0.5, Q Learning and Sarsa won about 66% of games when starting,whereas with an exploration rate of 0.9 both algorithms won around 72.5% of games when starting. This is curious because when playing against oneself, both machine learning algorithms fared worse with a higher exploration rate. However, when the algorithms played against each other, both benefited from a higher exploration rate when being player 1. We are unsure exactly what is accounting for this. One theory is that because player 1 has a shared algorithm when playing itself, player 2's moves are more predictable, and player 1 finds the optimal strategy faster. The additional exploration is unnecessary and moves player 1 to a suboptimal position. However, when player 1 is playing against a different algorithm, the difference in algorithm "confuses" player 1, and more exploration is necessary to find the optimal policy on both sides. If more exploration leads player 1 and player 2 to a more optimal strategy, we would expect to see player 1 win more often, as they are unbeatable given the perfect strategy.

The second parameter that we varied was the reward. Looking at the data, the effect of this variable is hard to deduce. Across none of our three tests did any of the rewards stand out as the dominant choice; each reward system had mixed success depending on the exploration rate, and no system was the most successful across every exploration strategy. To draw any definitive conclusion, I believe that we need to test more reward functions.

VIII. FUTURE WORK

Our implementation offers an optimal strategy for the generic 2-dimensional game of Connect Four using both Q-learning and Sarsa. Possible future work could extend to determining the optimal strategy for alternative versions of Connect Four.

The 3-dimensional version of Connect Four in particular would be interesting to implement an optimal strategy for. The 3-dimensional aspect of the game adds another layer of complexity to the model and the rules are adjusted also. Comparing the differences in the strategies could be an interesting point.

It would be conceptually intriguing to implement the optimal strategy with a large variety of board sizes as well. Including larger, smaller and different dimensional boards. Analysing and comparing the different resultant optimal strategies in order to identify or determine a pattern. Perhaps the findings could then be applied to other games or similar problems.

Finally adjusting the model to take into account the speed in which a game is won is another avenue that could be explored. Having the model weight actions and decisions that may lead to a possible win in less turns verse a guaranteed win in more turns.

Overall the work completed acts as a stepping stone into a greater exploration of the relationship between Connect Four and reinforcement learning.

IX. CONCLUSION

Memorizing and determining the optimal move for every possible board state in a Connect Four game is humanly impossible. However, in this paper, we depicted a reinforcement learning approach to the popular board game Connect Four in order to increase the probability of winning a game. Our Q-learning agent was able to achieve a peak win percentage of 73.4% when starting as player one versus the Sarsa agent. Alternatively when Sarsa started as player one versus Q-learning, our Sarsa agent was able to achieve a peak win percentage of 73.5%. Both algorithms never dipped below a win percentage of 50%, regardless of their opponent. Given that in 1988 Victor Allis solved the Connect Four game by showing that the first player can always win if they play the middle column first, we know that both our Q-learning and Sarsa method are ultimately suboptimal. Yet, this leaves room for much improvement.

Both Q-learning and Sarsa methods have the ability to be modified “to assign credit to achieving the goal to past states and actions using eligibility traces” [9]. This means that the reward that is associated with reaching the goal is “propagated backward to the states and actions leading up to the goal” [9]. The credit is also decayed exponentially so that the states that are closer to the goal are assigned larger state-action values. With eligibility traces and a suitable exploration rate might enable our agents to be more efficient and effective.

X. CONTRIBUTIONS

The entire group worked together to come up with agent strategies, implementation and to write the paper. We have contributed equally to the creation of this report and previous written submissions.

APPENDIX

Q-learning:

For this case, every test was a Q-learning agent vs a Q-learning agent where we varied the exploration factor and reward (shown below)

Reward: win-10, loss-0, tie-0

```
{1: 576, 2: 410, 0: 14, 'move_count': 24.564, exp: 0.9}
{1: 680, 2: 318, 0: 2, 'move_count': 19.614, exp: 0.8}
{1: 613, 2: 378, 0: 9, 'move_count': 21.891, exp: 0.7}
{1: 638, 2: 353, 0: 9, 'move_count': 21.082, exp: 0.6}
{1: 653, 2: 347, 0: 0, 'move_count': 18.741, exp: 0.5}
```

Reward: win-10, loss-(-10), tie-0

```
{1: 549, 2: 440, 0: 11, 'move_count': 24.641, exp: 0.9}
{1: 662, 2: 334, 0: 4, 'move_count': 19.458, exp: 0.8}
{1: 586, 2: 405, 0: 9, 'move_count': 22.092, exp: 0.7}
{1: 591, 2: 403, 0: 6, 'move_count': 20.787, exp: 0.6}
{1: 660, 2: 340, 0: 0, 'move_count': 18.81, exp: 0.5}
```

Reward: win-10, loss-(-10), tie-5

```
{1: 572, 2: 413, 0: 15, 'move_count': 25.012, exp: 0.9}
{1: 680, 2: 318, 0: 2, 'move_count': 19.218, exp: 0.8}
{1: 578, 2: 409, 0: 13, 'move_count': 21.916, exp: 0.7}
{1: 608, 2: 381, 0: 11, 'move_count': 21.314, exp: 0.6}
{1: 664, 2: 334, 0: 2, 'move_count': 19.296, exp: 0.5}
```

Sarsa:

In this case, every test was a Sarsa agent vs a Sarsa agent with the same varied exploration factor and rewards:

Reward: win-10, loss-0, tie-0

```
{1: 568, 2: 420, 0: 12, 'move_count': 24.398, exp: 0.9}
{1: 572, 2: 416, 0: 12, 'move_count': 23.252, exp: 0.8}
{1: 599, 2: 385, 0: 16, 'move_count': 22.175, exp: 0.7}
{1: 601, 2: 390, 0: 9, 'move_count': 20.801, exp: 0.6}
{1: 605, 2: 388, 0: 7, 'move_count': 20.535, exp: 0.5}
```

Reward: win-10, loss-(-10), tie-0

```
{1: 559, 2: 426, 0: 15, 'move_count': 24.653, exp: 0.9}
{1: 607, 2: 382, 0: 11, 'move_count': 22.633, exp: 0.8}
{1: 606, 2: 379, 0: 15, 'move_count': 21.64, exp: 0.7}
{1: 575, 2: 416, 0: 9, 'move_count': 21.167, exp: 0.6}
{1: 620, 2: 375, 0: 5, 'move_count': 20.688, exp: 0.5}
```

Reward: win-10, loss-(-10), tie-5

```
{1: 539, 2: 447, 0: 14, 'move_count': 24.521, exp: 0.9}
{1: 588, 2: 404, 0: 8, 'move_count': 22.764, exp: 0.8}
{1: 602, 2: 377, 0: 21, 'move_count': 22.102, exp: 0.7}
{1: 609, 2: 386, 0: 5, 'move_count': 21.177, exp: 0.6}
{1: 613, 2: 379, 0: 8, 'move_count': 20.453, exp: 0.5}
```

Sarsa vs Q-learning:

In this case, every test was a Q-learning agent as player one vs a Sarsa agent with the same varied exploration factor and rewards:

Reward: win-10, loss-0, tie-0
```
{1: 706, 2: 293, 0: 1, 'move_count': 20.186, exp: 0.9}
{1: 663, 2: 334, 0: 3, 'move_count': 19.499, exp: 0.8}
{1: 676, 2: 324, 0: 0, 'move_count': 18.852, exp: 0.7}
{1: 665, 2: 331, 0: 4, 'move_count': 19.019, exp: 0.6}
{1: 648, 2: 351, 0: 1, 'move_count': 18.716, exp: 0.5}
```

Reward: win-10, loss-(-10), tie-0
```
{1: 734, 2: 266, 0: 0, 'move_count': 20.32, exp: 0.9}
{1: 661, 2: 339, 0: 0, 'move_count': 19.197, exp: 0.8}
{1: 650, 2: 348, 0: 2, 'move_count': 19.102, exp: 0.7}
{1: 645, 2: 354, 0: 1, 'move_count': 19.081, exp: 0.6}
{1: 662, 2: 337, 0: 1, 'move_count': 19.098, exp: 0.5}
```

Reward: win-10, loss-(-10), tie-5
```
{1: 730, 2: 269, 0: 1, 'move_count': 19.99, exp: 0.9}
{1: 684, 2: 316, 0: 0, 'move_count': 19.44, exp: 0.8}
{1: 672, 2: 327, 0: 1, 'move_count': 18.644, exp: 0.7}
{1: 655, 2: 343, 0: 2, 'move_count': 19.449, exp: 0.6}
{1: 654, 2: 343, 0: 3, 'move_count': 18.988, exp: 0.5}
```

Sarsa vs Q-learning:

For this case, every test was a Sarsa agent as player one vs a Q-learning agent with the same varied exploration factor and rewards:

Reward: win-10, loss-0, tie-0
```
{1: 735, 2: 265, 0: 0, 'move_count': 20.311, exp: 0.9}
{1: 670, 2: 330, 0: 0, 'move_count': 19.396, exp: 0.8}
{1: 673, 2: 325, 0: 2, 'move_count': 19.181, exp: 0.7}
{1: 641, 2: 359, 0: 0, 'move_count': 18.785, exp: 0.6}
{1: 650, 2: 349, 0: 1, 'move_count': 18.968, exp: 0.5}
```

Reward: win-10, loss-(-10), tie-0
```
{1: 710, 2: 290, 0: 0, 'move_count': 20.346, exp: 0.9}
{1: 659, 2: 339, 0: 2, 'move_count': 19.773, exp: 0.8}
{1: 649, 2: 348, 0: 3, 'move_count': 18.683, exp: 0.7}
{1: 683, 2: 315, 0: 2, 'move_count': 18.479, exp: 0.6}
{1: 656, 2: 340, 0: 4, 'move_count': 19.146, exp: 0.5}
```

Reward: win-10, loss-(-10), tie-5
```
{1: 720, 2: 280, 0: 0, 'move_count': 20.28, exp: 0.9}
{1: 658, 2: 338, 0: 4, 'move_count': 19.72, exp: 0.8}
{1: 652, 2: 348, 0: 0, 'move_count': 18.96, exp: 0.7}
{1: 659, 2: 337, 0: 4, 'move_count': 18.723, exp: 0.6}
{1: 652, 2: 346, 0: 2, 'move_count': 18.922, exp: 0.5}
```

Explanation of results:
1: *# of Player 1 wins,*
2: *# of Player 2 wins,*
0: *# of tie games,*
'move_count'*: average # of moves per game*
'exp'*: exploration rate*

ACKNOWLEDGEMENT
We would like to thank Mykel Kochenderfer and the rest of the CS 238 staff.

REFERENCES

[1] Thon, Michael. “Connect 4.” *GamesCrafters*, UC Berkeley, 2018, gamescrafters.berkeley.edu/games.php?game=connect4.

[2] “Connect Four.gif.” *Wikimedia Commons*, 2011, commons.wikimedia.org/wiki/File:Connect_Four.gif.

[3] “Connect-Four Piece Red.” *Keyword Basket*, 2019, www.keywordbasket.com/Y29ubmVjdCBmb3VyLXBpZWNlIHJlZA/.

[4] “4 In A Line.” *MathIsFun*, 2018, www.mathsisfun.com/games/connect4.html.

[5] MIT. “Connect 4.” Connect 4, MIT, web.mit.edu/sp.268/www/2010/connectFourSlides.pdf.

[6] Pearce, Adam. “Connect 4 AI: How It Works.” Connect 4 AI: How It Works, Roadtolarissa, roadtolarissa.com/connect-4-ai-how-it-works/.

[7] Vandewiele, Gilles. “Creating the (Nearly) Perfect Connect-Four Bot with Limited Move Time and File Size.” Medium, Towards Data Science, 7 Jan. 2019, towardsdatascience.com/creating-the-perfect-connect-four-ai-bot-c165115557b0.

[8] Docker Inc. “What Is a Container?” *Docker*, 2019, www.docker.com/resources/what-container.

[9] Kochenderfer, M. (2015). Decision making under uncertainty : theory and application. Chapter 5. Cambridge, Massachusetts: The MIT Press.