Group 25
Travis Mason
Corey Briscoe
Andrew Gostomelsky

Push Fight, the Monte Carlo Algorithm, and Neural Networks

Abstract

Monte Carlo algorithms use many random inputs or decisions to generate plausible outcomes for problems that may be intractable or unsolvable otherwise. We are interested in comparing this classical artificial intelligence and game theory algorithm against newer approaches offered through machine learning on the domain of two-player zero-sum perfect-information games. For the project, we will be considering and comparing the performance of differing algorithms on the board game Push Fight. Push Fight as a game has considerable state complexity, stemming from a vast state space and multitude of possible move combinations. Navigating through such a complex state space is especially challenging for the kinds of algorithms typically introduced for two-player deterministic games such as minimax; therefore, we predict Monte Carlo principles -- specifically in the form of the Monte Carlo Tree Search algorithm -- will be better suited to such a problem. We use the Monte Carlo Tree Search algorithm to create an algorithmic AI capable of competently playing Push Fight, and use the win-chance predictions generated by this algorithm in gameplay to generate a dataset that we feed to a machine learning algorithm known as a gradient boosting machine (GBM) to make similar predictions and in turn create another AI capable of playing the game even better than the former. As such, we conclude that Monte Carlo algorithms are effective for these types of games, as well as for

generating datasets suitable for the training of machine learning models.

Introduction

The main algorithm we are investigating is the Monte Carlo algorithm. Monte Carlo was originally introduced by mathematician Stanislaw Ulam (Pease). Monte Carlo was used extensively on, and in some sense developed for, the Manhattan Project during World War II (Pease). The Manhattan Project was the codename for the clandestine research and development of nuclear weapons led by the United States government in the early-to-mid 20th century (History). The project was started after fears of German scientists utilizing nuclear technology for the creation of powerful and unprecedented weaponry. The Monte Carlo algorithm was used to plug random numbers into the equations for the nuclear bomb (Fylstra). This allowed them to make very good predictions of results and is credited with helping guide the creation of the nuclear bomb. Specifically, we are interested in Monte Carlo as it is used in game state search. This algorithm, known as Monte Carlo Tree Search, was introduced by Rémi Coulom (Coulom).

We are also implementing the machine learning algorithm known as a gradient boosting machine. Gradient boosting algorithms were developed by Jerome Friedman (Natekin). One of the more popular machine learning algorithms to date, gradient boosting machines can be used for a variety of different data driven applications. Gradient boosting machines are used in many analytical kaggle competitions (Feng).

One of the papers that guided towards our research question is "The Expected-Outcome Model of Two-Player Games" by Bruce Abramson. Another paper

that guided us towards our research on the game of Push Fight is "Computational Complexity of Generalized Push Fight" by Jeffrey Bosboom et al.

Research Questions

The main research question we are exploring is if we can use the Monte Carlo algorithm to generate data to effectively train a machine learning algorithm. We want to know if the datasets we create from the Monte Carlo algorithm can be used to train an optimal gameplay machine learning algorithm. Another research question we are going to address is how the Monte Carlo algorithm compares to a data-driven machine learning algorithm. Once we achieve competent gameplay for Push Fight with both of these algorithms we would like to compare the algorithms to determine which is the most effective during gameplay.

Methods

One algorithm we are using for our project is the Monte Carlo algorithm to generate data and it will be our first algorithm implemented. The next algorithm is gradient boosting machines (GBM), is a machine learning algorithm that will train from the Monte Carlo generated data.

Monte Carlo Tree Search, the most-common method of implementing the Monte Carlo algorithm to game trees, works under the idea of simulating random gameplay outcomes from a certain present game decision in order to evaluate how likely that decision is to lead to victory. By running enough of these simulations, a heuristic for evaluating potential decisions resulting from the current state is reached, and we can

create an algorithmic AI that uses these evaluations to make what we believe to be optimal gameplay decisions and play a game from start to finish.

Using MCTS, we will create a dataset of gameplay states and outcomes which we can use to train a machine learning model powered by gradient boosting. The data needs to be stored or converted to a form that makes it as easily understandable to the ML model as possible. We assume that this will not prove especially difficult and is likely achievable with the CSV file format, storing the information generally just as it appears in the code.

Gradient Boosting works by using a loss function to be minimized and many weak-learners, which are decision trees deliberately kept greedy and quick. As the algorithm continues, more of these weak-learners are added to shift the final result in the direction that minimizes loss. This process also involves modifying the weights of each weak-learner to achieve this.

We consider the time and space complexities of these algorithms to be as follows:

Monte Carlo Tree Search at evaluation - O(b^m) time complexity, O(bm) space complexity, where b = number of moves, m = maximum depth

Gradient Boosting Algorithm - n = the number of training samples, p = the number of features, n-trees = the numbers of trees. training: O(npn-trees) , Prediction: O(pn trees)

It goes without saying that Monte Carlo methods are not perfect as they rely on randomness to achieve results. However, they are accurate to a convincingly good extent, and they have demonstrable applications to real-world problems. On the other hand, the effectiveness of machine learning algorithms depends on the presence of

sufficient training examples to properly situate the model to the task. With this, the model can predict, within a reasonable expectation of accuracy, a logical solution, but no guarantees can be made for the perfection of such an algorithm in yet-unseen cases.

A possible alternative to Monte Carlo could be the MiniMax algorithm. An alternative to the Gradient Boosting algorithm could be adaboost which is a very similar algorithm to Gradient Boosting. There are also many other ways we could have chosen to train an AI to play Push Fight using different machine learning techniques.

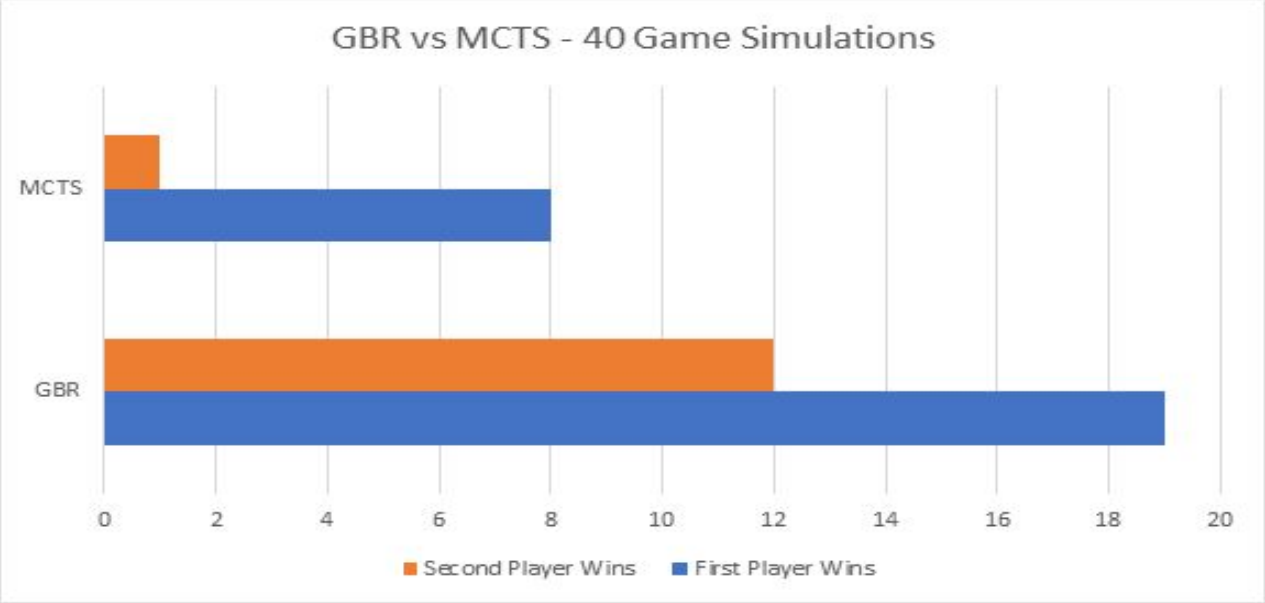
Monte Carlo will be used to generate data to be used by the Gradient Boosting algorithm. The games will be split up into player 1 winners and player 2 winners. The Gradient Boosting algorithm will use linear regression to predict what the next best move will be to win a game. Once we train the algorithm we will use it to play against humans and the Monte Carlo algorithm. Our first question will be answered if we successfully train the Gradient Boosting algorithm with the data we generated. The second question will be answered if the trained Gradient Boosting algorithm competes and beats humans as well as beats the Monte Carlo algorithm in Push Fight.

Starting off with the Monte Carlo algorithm to generate a bunch of game data could be comparable to a human playing a bunch of games to gain more information about the game. Once a human has played enough games they will make better predictions on what moves will win them the game. The Gradient Boosting algorithm will simulate the human thought process by trying to predict the best move to make to win the game. While humans don't have the brain capacity to calculate a bunch of weak solutions and build a strong one from them, humans do make multiple decisions and

usually pick the best one or compromise on multiple decisions and choose the best for all of them.

Results

We found that we could successfully generate data from the MCTS algorithm to serve as training data for the ML algorithm, GBR, to competently play the game push fight. After generating over 500,000 different board states with their win percent the GBR model was trained and saw an accuracy score of a little over 50%. While this score might not seem high the model outperforms the MCTS algorithm in almost every simulation.



As shown in the graph, GBR outperformed MCTS as both the first player and second player in the 40 simulations. While the first player seems to have an advantage it still did not stop GBR from winning 12 out of 20 times as the second player.

Discussion

Generating the data that was used to train the ML model worked generally as we expected it to. It took more training samples than we had expected for the ML model to be able to make accurate predictions about the value of a move. A possible reason that this was the case could be the additive nature of GBR. Since it does not eliminate individual weak-learners as the model trains, there will always exist the naive weak-learners pulling the predictions in meaningless directions and drowning out more sophisticated ones until the number of training cases is sufficiently great.

The fact that the ML model could outperform the MCTS algorithm may seem surprising, but after some consideration of why that is the case, the result makes sense. Our GBR model is trained from individual cases of how MCTS predicted possible outcomes. When MCTS is making these predictions, it does so based on a finite number of randomized playouts. As such, the probabilities MCTS finds are only accurate to a certain extent and can vary. The GBR model, through seeing hundreds of thousands of these training cases, crucially manages to offset the possible inaccuracies of individual cases as it recognizes patterns in piece positions/turns. Therefore, in a match, the MCTS AI is making predictions that are significantly less accurate than the predictions of the ML model, making the latter superior on average.

The state space of Push Fight, as mentioned previously, is very vast and we conjecture that an algorithm like minimax that aims to look at many of these states is likely to be infeasible to run in an acceptable timeframe when it comes to this kind of game. On the other hand, applying a Monte Carlo algorithm to this problem space turns

out to be both effective and flexible, as the algorithm can play the game with competency and we can adapt how many of these random simulations we want to play out to fit execution time within an allotted timeframe. This gives Monte Carlo simulation a distinct advantage over minimax in the problem space.

Furthermore, the fact that a ML model could be constructed and trained in such a manner as to exceed the algorithm from which its training cases were generated speaks not only to the effectiveness of Monte Carlo in designing agents to play the game at a respectable level, but also to its capacity to create data sufficient for training such a model.

Outside Code

There was only one instance of outside code that was needed. This one snippet of code was used for a specific situation and the library that was used is called ast. The code snippet used was ast.literal_eval(). This was needed because when saving the board data as a list it was converted into a string once saved to a csv. To get the data ready to train the GBR model the string list needed to be converted into a list of numbers. With over 500,000 lists needed to be converted we needed a fast easy way to do this. The code loops through each separate row, selects the string list, converts the string into a list as well as evals the data in the list which converts the numbers into integers. You can find the source code at https://stackoverflow.com/questions/23119472/in-pandas-python-reading-array-stored-as-string by shaktimaan. We also used the python libraries numpy, pandas, joblib, and sklearn. We did not use online code snippets for these, but we did use their

documentation for referencing. For sklearn we used their documentation for the GBR model, but we also had knowledge on how to implement this model and joblib from the IU course B455. The four documentations that we used are referenced here.

Sklearn GBR: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html

Pandas: https://pandas.pydata.org/docs/user_guide/index.html

Numpy: https://numpy.org/doc/1.18/user/quickstart.html

Joblib: https://joblib.readthedocs.io/en/latest/

Future Steps

Something we noticed in the algorithm's behavior was the tendency to prefer "self-preservation" over strategies that may lead to quicker wins. The blame for this may lie in the criteria we used to evaluate how good a move was -- which was simply a predicted probability that that board state led to a win. There seemingly are ways we could either improve the current algorithm or re-envision the algorithm to adjust the AI's behavior to seek out wins more quickly, perhaps through some heuristic or changing our MCTS implementation to prioritize moves according to which lead to a win in the least number of turns. It is uncertain as to whether making the latter modification would preserve the competency of the AI, while adding this kind of consideration to our existing algorithm would perhaps do so, though certainly at the expense of longer computation times overall.

We also believe that Monte Carlo could be introduced into this problem space specifically for designing an algorithm to come up with optimal starting positions. One of the major factors of high-level Push Fight is the placement phase at the start, which we removed for simplicity and better standardization of our results. While we initially configured the algorithm to place pieces randomly, we decided it was best for each player to have the same starting configuration: a line of three squares and one circle as far forward as possible, followed by the last circle close behind, in order to keep all pieces initially far from danger as possible and reduce a player's ability to move onto the opponent's side of the board. It seems likely that we could apply Monte Carlo principles to develop an AI that decides on an optimal configuration for pieces either to start whenever it places its pieces first, or to counter the opponent's placement when it places its pieces second -- and can therefore place pieces in such a way to exploit that.

We would have liked to include a comparison between other algorithms such as minimax to evaluate the competency of Monte Carlo in head-to-head matchups as we did with MCTS-versus-GBR, but due to time constraints we prioritized the aspects of the project we felt were most important and relevant to our research goals, the vast majority of which we were able to accomplish by the end of the project.

We conclude that using Monte Carlo, an algorithm inherently based on randomness and forethought, is an effective tool for training computer agents to play two-player zero-sum perfect-information games.

References

History of Monte Carlo, Pease
https://towardsdatascience.com/an-overview-of-monte-carlo-methods-675384eb1694

Manhattan Project, History
https://www.history.com/topics/world-war-ii/the-manhattan-project

Monte Carlo Manhattan Project, Daniel Fylstra
https://www.solver.com/press/monte-carlo-methods-led-atomic-bomb-may-be-your-best-bet-business-decision-making
Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search, Rémi Coulom.
https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.81.6817

Gradient Boosting Machines, Alexy Natekin; Alois Knoll
https://www.frontiersin.org/articles/10.3389/fnbot.2013.00021/full#B19

Gradient Boosting Machine, Zeyu Feng; Chang Xu; Dacheng Tao
https://easychair.org/publications/open/pCtK