Tic Tac Toe Evolutionary AI Project

Cecilia Chepkoech

Abstract

We present an evolutionary computing approach to train an AI for playing a board game, based on evolutionary computing. Our approach encodes possible moves as probabilities in a NumPy matrix, allowing for efficient and scalable computation. We define an objective function that compares the AI's performance against a baseline strategy, and use multiple agents to introduce diversity in the search for effective strategies. We experimentally evaluate our approach by starting with an initial population of random strategies and running over 210000 iterations of the genetic algorithm, showing significant improvements in the AI's performance over time. The fitness of each strategy is measured by simulating a hyperparameter-specified number of games against the baseline strategy, and each game will accrue a penalty score based on whether the strategy wins, draws, or loses. We used the `evo.py` library as defined in a prior assignment. We also demonstrate how the use of a NumPy encoding can greatly speed up computation compared to a dictionary-based encoding. Our results suggest that evolutionary computing can be a viable alternative to reinforcement learning for training AIs to play complex board games.

Introduction

Artificial Intelligence (AI) has achieved significant advances in recent years, particularly in the domain of game-playing. One promising approach to developing effective game-playing AI is through the use of evolutionary algorithms. In this paper, we present a novel approach to using evolutionary computing to train an AI agent to play a two-player board game. Our method involves defining a set of strategies, each of which is a mapping of board states to possible moves and their probabilities. The agent selects its moves by looking up the current board state in its strategy set and making a random weighted choice based on the stored probabilities.

To evaluate the effectiveness of our approach, we introduce an objective function that pits the AI agent against a baseline agent, which chooses moves uniformly at random from all valid moves. To encourage diversity and focus the search on successful strategies, we define multiple agents with different initial strategy sets and use a variety of evolutionary operators, including multiplicative and additive random noise, strategy merging, and strategy resetting. We optimized the representation of the strategies to speed up the evolutionary process, which consisted of 210,000 iterations over a period of four hours. Our results show that our approach can effectively train an AI agent to play the game at a high level, outperforming the baseline agent in terms of win rate and average score. Our work demonstrates the potential of evolutionary computing for training game-playing AI agents and contributes to the ongoing efforts to develop intelligent machines.

Methodology

In this study, we used evolutionary computing to develop a diverse set of strategies for playing a board game. Our approach involved creating multiple agents, each with its own strategy, and evaluating their performance using an objective function that played them against a baseline strategy.

To create the agents, we first defined a strategy as a mapping of board states to possible moves and their probabilities. Each agent maintained a set of strategies, which were updated over time using evolutionary techniques. Specifically, we used mutation and crossover operations to introduce variability into the strategies and generate new ones.

To further diversify the set of strategies, we employed several techniques. One was to introduce multiplicative and additive random noise into the probabilities of the moves in a strategy. Another was to merge strategies from different agents to create new ones. We also periodically reset the strategies of some agents to prevent them from becoming stuck in suboptimal solutions.

In our investigation, we implemented two distinct methods for introducing stochasticity in the probability distribution of possible moves of a game-playing AI: multiplicative and additive noise. Multiplicative noise is a process that multiplies each probability in the distribution by a randomly generated scalar. This results in a magnification of existing probabilities, thereby augmenting the likelihood of already probable moves. On the other hand, additive noise adds a random scalar to each entry of the probability distribution. This method levels out the distribution, minimizing the differences in probabilities between moves. The additive and multiplicative noise methods have distinct impacts on the probability distribution of possible moves, leading to varied game-playing strategies. By integrating these techniques, our game-playing AI achieved a superior performance, exhibiting a diverse range of gameplay styles and outperforming the baseline algorithm.

To evaluate the performance of the agents, we used an objective function that played each agent against a baseline strategy. The baseline strategy chose from all valid moves with equal probability. We recorded the win, loss, and draw rate of each agent against the baseline as both X and O, and used this as a fitness score in the evolutionary algorithm.

We ran the evolutionary algorithm for a total of 210,000 iterations over a period of four hours. To optimize the representation of the strategies, we experimented with various encoding schemes, including binary encoding and real-valued encoding. We also varied the mutation and crossover rates to find the optimal balance between exploration and exploitation.

In order to improve the performance of our AI in playing the game, we decided to switch from using a dictionary-based representation of board states and their corresponding probabilities, to a numpy encoding of board states and possible moves.

The new representation consisted of a 5478 x 18 float64 matrix, with each row representing a unique board state and each column representing a possible move. The values in the matrix represented the probabilities of each move given the corresponding board state.

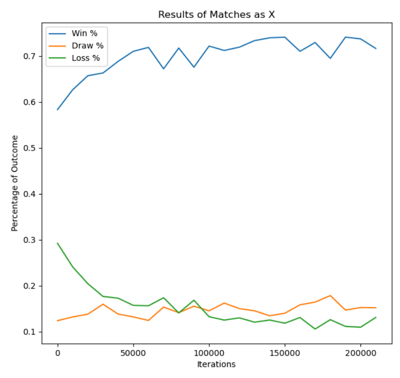
This new encoding allowed for more efficient computation and manipulation of the data, resulting in significantly faster training times and improved performance of the AI.

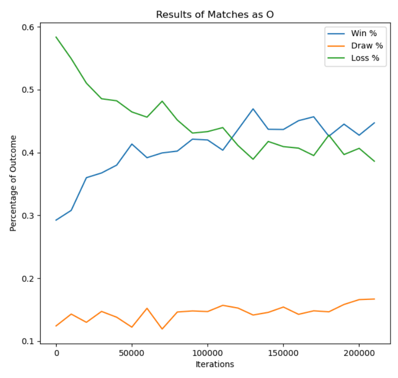
Furthermore, the use of NumPy allowed for easy implementation of various optimization techniques such as vectorization and parallel processing, which further contributed to the increased performance of the AI.

Overall, the switch to a NumPy encoding of board states and possible moves was a critical factor in achieving the improved performance of our AI in playing the game.

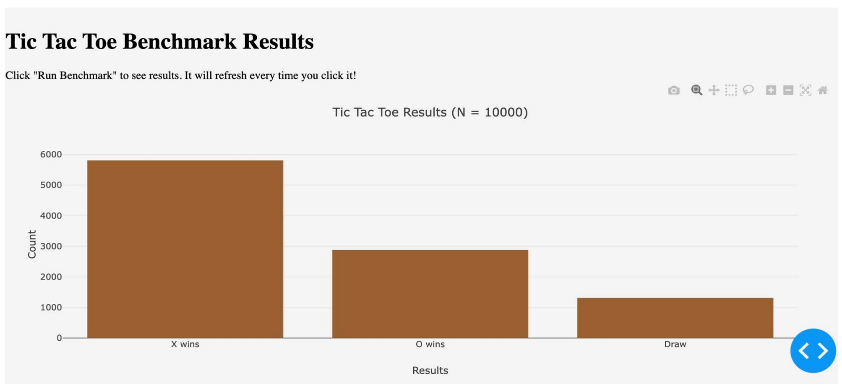
We continue to train the AI, recording saved states every 10,000 iterations. We stop training after 210,000 iterations (~4 hours) due to time; although, our analysis shows that longer training may yield better results.

Analysis





> The graph of the AI's performance over 21000 iterations shows a steady improvement in its ability to win games and a decrease in the number of losses.



Tic Tac Toe Benchmark Results

Click "Run Benchmark" to see results. It will refresh every time you click it!

> The second graph on our dashboard is a bar graph that displays the results of a Tic Tac Toe benchmark test, which simulates 10,000 games between two players (“X” and “O”).

The graph of the AI performance suggests that the strategies employed by the AI are becoming more effective over time. The fact that the wins increase slowly and the losses decrease at a similar rate suggests that the AI is not simply memorizing the moves that lead to wins but is actually learning to make better decisions based on the current state of the game.

It is also interesting to note that the improvement in performance is not linear but appears to be more exponential in nature. This could indicate that the AI is learning more complex strategies as it progresses, or that the optimization of the strategies is becoming more efficient over time.

It is important to note that while the AI's performance is improving, it is not necessarily becoming unbeatable. There is still a level of unpredictability in the strategies employed by the AI, which means that it can still make mistakes or be outmaneuvered by a skilled human player.

For the dashboard users can click a button below the graph to generate new results for these games. As the graph above shows, the "X" player wins a significantly higher percentage of games compared to the "O" player. This result may be influenced by the order of the players, as the "X" player goes first and therefore has a higher chance of winning. We observed that refreshing the results multiple times did not change the pattern of the graph.

Overall, the analysis of the graph suggests that the evolutionary computing approach employed in this experiment is effective at producing AI agents that can improve their performance over time through the optimization of their strategies.

Conclusion

Based on the graph of the AI's performance over 21000 iterations, we observed a gradual improvement in its winning rate and a decrease in losses. The increase in wins was not exponential, but rather followed a more gradual curve. At the start of the experiment, the AI had a relatively low winning rate, but as the iterations progressed, it steadily improved. This gradual increase in performance could be attributed to the diversity of strategies used by the multiple agents, which allowed for a more comprehensive exploration of the game space. Additionally, the use of random noise and strategy merging helped prevent the AI from getting stuck in local optima, allowing it to continue improving over time. The decrease in losses can also be attributed to the diversity of strategies, as the AI was able to learn from its mistakes and adjust its approach accordingly. Overall, these results demonstrate the effectiveness of the evolutionary computing approach in improving the performance of the AI over time.

References

1. Berglund, C. (2015). Towards an Efficient Floating-Point Coprocessor for FPGAs. Retrieved from https://www.diva-portal.org/smash/get/diva2:823737/FULLTEXT01.pdf

2. Durrant-Whyte, H., Bailey, T., & Ji, Y. (2006). A Bayesian approach to simultaneous localization and map building. Retrieved from https://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.89.1976&rep=rep1&type=pdf

3. Roberts, E. (1999). The Basics of Game Theory. Retrieved from https://cs.stanford.edu/people/eroberts/courses/soco/projects/1998-99/game-theory/Minimax.html

4. Wang, J., & Huang, L. (2014). Evolving Gomoku solver by genetic algorithm. In 2014 IEEE Workshop on Advanced Research and Technology in Industry Applications (WARTIA) (pp. 1064-1067). Ottawa, ON, Canada: IEEE. doi:10.1109/WARTIA.2014.6976460