Learning to Play Checkers using Evolution

watki185

December 2021

Abstract: With games like checkers, there is no clear way for humans to improve than playing and learning. 'Learning' is ill-defined for computer-models with such problems, so we employ an evolutionary algorithm to improve an agent's play style. The evolutionary algorithm that we employ is similar to that found in nature, with traits from organisms that are better 'fit' for its environment being passed down each generation. We use this idea to employ neural-networks as an evaluation function for a Monte-Carlo tree search on a version of the two-player, zero-sum game checkers.

1 Introduction

Checkers is a two-player, zero-sum board game, in which each player attempts to use their pieces to 'remove' all of their opponent's. In this paper, a variant of checkers is used where it is played on an 6x6 board with six pieces per player to begin, placed on every other square (they all are on the same shade). Pieces can move one space diagonally forward or jump over an opponent's piece if it lies where would otherwise be a legal move, and the square after is empty, which will remove the jumped over piece. When a piece reaches the other side of the board, it is upgraded to a Queen and able to move one space along any diagonal, forward or backward. The game terminates when either player has no pieces left in play, or a draw is reached. A draw can occur either when a certain number of moves have been made, in this case 60 (or 30 per player) or the player whose turn it is has no possible moves (stalemate). Each player alternates turns until the game comes to a conclusion.

We are looking to create an Artificial Intelligence capable of playing checkers at a high level. While the game has been 'solved' in a certain sense, the memory required to use this would be massive, so we are looking for a more efficient method still able to perform well. It is difficult to construct good heuristics for the game because there are many factors and nuances involved.

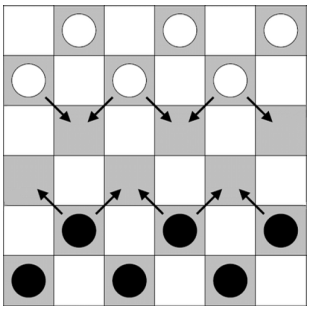

Figure 1: Six-by-Six Checkers Board with Possible Moves Outlined

2 Related Work

2.1 Background

The problem has far too many possible combinations of moves and states for any algorithm to calculate the game to completion. An accurate way to represent the given problem would be as a Mini-max problem, as each player is trying to minimize the best their opponent could do. Some form of state evaluation function is typically used to determine whether a given position on the board is likely to lead to a win. There are many different search methods that can be used to look through possible actions, such as the Alpha-Beta search or the Monte-Carlo Tree search. Possible heuristic functions can be constructed from previous domain-knowledge, such as checking which player has more pieces left, complex neural networks used in evaluating a state, or many other options.

In [1], the use of Evolutionary Algorithms in training is investigated. This is when many agents are initialized randomly, and those that perform the best according to some fitness score, are chosen to stay and create children, where the worst are gotten rid of. The children are slight variations of the previous agents that may allow for better performance.

The Monte-Carlo Tree Search (MCTS) is examined from the perspective of games in [2]. MCTS will look at possible actions and decide which one to pursue for further analysis. It attempts to balance pursuing potentially good lines with balancing exploration of other states that may not immediately seem optimal.

In [3], optimizing Alpha-Beta search for a board game is explored. The search functions by exploring all possible sequences of actions to a certain depth from a given state and choosing the best move with respect to a given evaluation function. It also examines the use of Quiescence Search, which looks to minimize the horizon effect. In checkers, an example would be capturing an opponent's piece as the final move may appear as an action leading to a good state, but it doesn't account for if the opponent can then capture back immediately. This search's idea is to extend the depth until no more capture occur and the position is in a 'steady state'.

Each of the presented solutions have merit when it comes to our problem

of checkers. The Alpha-Beta search and Monte-Carlo Tree search are similar in that they will search possibles action from the given state in an attempt to find the best move, but once they have been initialized no more improvements are made upon their heuristics. Their key difference is that the Alpha-Beta would run a pseudo-breadth search, considering all lines, while Monte-Carlo would focus primarily on the most promising. The evolutionary algorithms work differently by continuously 'playing' against itself and finding an optimal heuristic to use in playing, and there is far less emphasis on the search itself. The two searches would require more memory during a game to keep track of every position, than the evolutionary algorithms would as it uses a complex neural network heuristic. The evolutionary algorithm would require significant time beforehand to train itself to become a better solution, while the others would be able to immediately begin play after initialization. With a very good heuristic, Monte-Carlo would likely find better moves than the Alpha-Beta search, but this in of itself is a challenge. Without any prior domain knowledge, the evolutionary network would be able to find better moves as it learns. There are benefits to each algorithm, as well as their drawbacks.

2.2 Algorithm Variations and Improvements

When searching the space of possible actions from a given state, there are many different methods applicable. Historically, Alpha-Beta tree search has been used most but more recently Monte Carlo tree search has gained traction. The main improvement with Monte-Carlo tree search is that it has two properties, where it exploits good actions in it's search and explore nodes that haven't been looked into a sufficient amount. There are many promising variations of Monte-Carlo tree search, but the ones most useful to us is likely Multi-Player Monte-Carlo tree search and Temporal Difference Learning [4]. Multi-player Monte-Carlo tree search will create a vector of evaluation values at each state node, which contains a scoring from each players' perspective, and will choose the action that corresponds to the best state from a given point of view. The goal of Temporal Difference Learning is to use past actions taken and the resulting terminal state to train an agent, which is in essence a version of reinforcement learning. This updates an agents scoring of a given state from the final state reached much later. While this does provide data for a Neural Network to perform regression with, it does use a delay between an action being made and the consequences of it. Another of the improvements made was in the use of Reinforcement Learning [5]. It uses this technique to progressively train an agent to improve at games, by applying the final value of a game to each action played.

Most notably, Monte-Carlo tree search has been used in zero-sum games with two players. It has proven to be successful using a Deep Neural Network evaluation function in Alpha Zero [6]. After a relatively short time of self-play, it has been able to achieve super-human play in games such as Chess. Many complex problems have been investigated with generalized versions of Alpha Zero's advancements [7].

A feed-forward Neural Network takes a vector input and will perform a series

of calculation to return an output vector. Because there is not a perfect method to deduce the best Neural Network for any given problem, it could be helpful to make use of Evolutionary Algorithms [8]. We would have a population of agents, all competing with each other to be the 'best', which is based off of some fitness score. The best agents would be kept and mutated in hopes of obtaining an improved version, much like evolution. In some cases, these can often be better than other optimization algorithms[9].

One drawback of classical evolutionary algorithms is that the mutation rate must be selected manually, and there is often need for fine tweaking. There have been improvements upon basic evolutionary algorithms, with Genetic Algorithms specifically, which have become Adaptive Genetic Algorithms [10]. These Adaptive Genetic Algorithms change mutation rates as populations progress in order to maintain steady progress and avoid stalling in a local maxima or search too far from the goal.

Similar to Adaptive Genetic Algorithms, there are differential evolution algorithms with certain strategies for mutation [11]. Differential Evolution Algorithms work similarly to typical evolutionary algorithms, but instead of random mutations, they work towards mutations that minimize some error or loss function. Likewise, there are certain strategies in creating mutation rates within an algorithm.

3 Approach

3.1 Agent Design

Each agent performs a limited time Monte-Carlo Tree Search as outlined in [2], to search moves from a given position. The board's position is uses a Multilayer Perceptron Neural Network as a mid-state evaluation function, to determine the optimal move. The input to the neural network is a given position, which is formatted as a concatenation of two vectors: the first being the position of all the 'normal' pieces written as either 1 or -1 for each team, and the second being the position of all 'upgraded' or Queen pieces written as 1 or -1 based on the team. The neural networks will use a ReLu activation function for all hidden layers, and a sigmoid activation for the output layer, so as to give a 'rating' of a position between zero and one.

As discussed in [1], we initialize with some number of randomly generated agents. Every epoch, the set of agents will compete against a 'boss' agent from which their scores will be calculated: a zero for every loss, one half for every draw, and one point for every win. An agents fitness score is given by the sum of all points per game played in an epoch. From which, the top twenty-five percent of agents will be kept and altered to produce the next generation of agents. If an agent receives a score of three-quarters of the amount of games played, it then takes over the role of the 'boss' agent. This is to ensure that every agent is on a level playing field, playing the same opponent, and that the 'boss' agent will increase in skill relative to the rest of the agents.

3.2 Agent Alteration

We alter our agents with three different methods: mutation, crossover, and a form of reinforced learning. Each epoch will produce one-quarter of the initial population, so the next generation will be constructed for these. One-quarter will remain the same as the previous generation's winners. One-quarter will be produced from 'mutating' each agent. One-quarter is from performing a 'crossover' of each agent. The final quarter is from a form of reinforced learning.

Maintaining the best agents from the previous round can be used in Genetic Algorithms, which is called Elitism. It is used to ensure that the agents will either improve or remain the same level of skill, not decrease. Without this, there is a possibility that our agents would fluctuate between getting better and worse, and would rely heavily and luck.

The mutation alteration will choose one of the layers of the neural network and change the weights according to a normal distribution. This allows for agents to maintain similar evaluations of positions as its 'parent', yet still have ways to improve its scoring ability. The standard deviation of the normal distribution is chosen ahead of time, so that changes in weights are drastic enough to make a difference, and still similar enough to not entirely change an evaluation.

The reinforced learning alteration is similar to that used in [7]. Each agent will keep track of all positions played and their corresponding end-state evaluation, or how the game ended (loss, draw, or win). It then uses supervised learning to train the neural network to predict the end-state value from the given position. It improves its error based on a mean difference value, and according to a pre-chosen learning rate, again so as to allow for improvements, but not completely changing the neural network's weights.

3.3 Software

For playing the game of checkers we use a custom created code for the six by six variant of checkers, with no double, nor forced jumps. To create the neural network for each agent and to train with reinforced learning we use TensorFlow, a python open source machine learning library, to assist with the program. A custom-coded search function finds all possible moves from a given position and is used to construct the tree for a Monte-Carlo Tree search.

4 Experiment Design and Results

4.1 Design

The purpose of our experiment to see if it is possible for a relatively basic algorithm using neural networks and evolutionary algorithms can learn to play a two-play game well. We expand on this idea by seeing if improvements can be made to the learning rate by using different values for various things, such as the mutation/learning rate, number of layers in a neural network, and amount of games played by each agent per epoch. We begin with a base case of 5 layers

in a neural network, with a mutation rate of 2^{-3} and learning rate of 0.002, and 8 games played. We also test with layers being three and seven, mutation rate being 2^{0} and 2^{-6} with learning rates of 0.2 and 0.002, respectively, and games played being four and sixteen. We compare the results of each method to our base case to see if there are major differences in their convergence rate. Each experiment had a population size of 256, for 512 epochs, 2^{-4} seconds per move, and all other methods remained consistent. We also keep track of the 'boss' agent in each experiment at each epoch, so we can see how they improve over time.

4.2 Results

Our results show case the improvement of our base case–5 layers, Mutation Rate of 2^{-3}, Learning Rate of 0.02, and 8 games per agent–agent over time, as well as a comparison of the base case agent's final 'boss' to the alterations of our algorithm. The results from table 1, show how the 'boss' agent at each given epoch of the base case compares against the final 'boss' agent at epoch 512, after a series of eight games. Tables 2, 3, and 4 show how each variation of our algorithms compares to the original base case algorithm when changing the mutation and learning rate, amount of layers, and number of games played, respectively.

Epoch Number	Final Score	Time
100	0	1.0 hr
200	0.5	1.9 hr
300	2.5	2.9 hr
400	3.5	3.8 hr

Table 1: Base Algorithm Skill Over Time

	Time Per Epoch	Final Score	Beat Base Case Bot at Epoch
Increased Mutation and Learning Rate	32 sec	1.5	Did Not Beat
Decreased Mutation and Learning Rate	34 sec	3.0	Did Not Beat

Table 2: Comparison of Mutation and Learning Rate

5 Analysis

5.1 Improvement Over Time

From table 1 we can observe that the amount of time per epoch remains relatively constant throughout the training. We also have that the initial portion

	Time Per Epoch	Final Score	Beat Base Case Bot at Epoch
3 layers	34 sec	5.0	442
7 Layers	35 sec	2.0	Did Not Beat

Table 3: Comparison of Amount of Layers

	Time Per Epoch	Final Score	Beat Base Case Bot at Epoch
4 Games	17 sec	4.0	497
16	69 sec	3.5	Did Not Beat

Table 4: Comparison of Number of Games Played

of epochs don't show much progress on learning to play our variation of checkers well. However, later in training the skill level of the 'boss' agents seem to progress more rapidly, alluding to there being even more progress beyond the 512th epoch. This may be due to the first few hundred epochs simply doing a pseudo-random search of the entire 'space' of neural-networks until it stumbles upon a relatively useful evaluation function, which then would rapidly reproduce and become the majority of the population. So the initial epochs are spent finding 'good' weights to evaluate the board, while after the first few hundred the evolution is then spent developing a 'strategy' for the evaluation.

5.2 Comparing Variations

According to table 2, the optimal mutation rate that was found is with a mutation rate of 2^{-3} and learning rate of 0.002, which is relatively unsurprising because these values were chosen specifically after some rudimentary testing of alteration values. The increased mutation rate seems to not be able to finely tune a strategy because the weights vary too much and don't closely correspond to their 'parent' agents. While, the decreased mutation rate seems to be able to still learn to play effectively, just at a slower rate. The decreased rate would probably show improvement upon the base case rate if the algorithm were allowed to run for more epochs, as it would allow for more fine-tuning in strategy.

As can be seen in table 3, fewer layers appears to be better. With three layers, it is able to get a score of 5/8 against the 'boss' bot from the base case. This faster convergence may be due to the fewer layers needing less mutation to improve as there are fewer weights, a higher ratio of layers crossing-over between neural network, or something else entirely. On the other hand, seven layers appears to somewhat learn to play, but was unable to improve on the original five. This increase in complexity may be useful later in training, but at the beginning appears to be a hindrance.

The number of games played per agent to determine their fitness score seems relatively unimportant, as shown by table 4. When only 4 games are played, it

produces a relatively equally skilled opponent as the base case after 512 epochs, as well as after sixteen games. The only difference is that the amount of time per epoch is scaled proportionally to the number of games play. So it would be more efficient to use fewer games as a fitness score, to a certain point, as clearly zero games would yield no information, and one game may still not yield enough.

6 Conclusion

Our work has shown that genetic algorithms as a way to improve play on two-player games is a viable approach. Factors such as epochs trained, mutation rate, and complexity of evaluation function are all important factors in finding the best-suited algorithm for training. Luckily, the algorithm itself and its parameters are fairly easy to understand and produce, allowing for a wide range of testing and alterations of conditions. In future work, it would be interesting to see how the algorithms presented here would improve over a greater period of time and see if they uncover a 'perfect' play style. To decrease the computational time required, this paper focused on a six-by-six variant of checkers, so using it on a more complex game could yield interesting results and it would be intriguing to find a limit, if one exists, of the complexity such an algorithm could sustain and learn. With applying this algorithm to more mainstream games, it would be important to compare its effectiveness to other known-algorithms, especially those made using domain-specific heuristics, which six-by-six checkers lacks. While not using human-knowledge is a focus-point of this algorithm, it would be interesting to see how learning would progress if a heuristic were applied in scoring each agent, at least for the initial bit of training to help improve convergence speed.

References

[1] Mustafa E. Abdual-Salam, Hatem M. Abdul-Kader, and Waiel F. Abdel-Wahed. "Comparative study between Differential Evolution and Particle Swarm Optimization algorithms in training of feed-forward neural network for stock price prediction". In: *The 7th International Conference on Informatics and Systems (INFOS)* (2010), pp. 1–8.

[2] Istvan Szita Guillaume Chaslot Sander Bakkes and Pieter Spronck. "Monte-Carlo Tree Search: A New Framework for Game AI". In: *Bijdragen* (2010).

[3] A. Chrysopoulos A. Papadopoulos K. Toumpas and P. A. Mitkas. "Exploring optimization strategies in board game Abalone for Alpha-Beta search". In: *IEEE Conference on Computational Intelligence and Games (CIG)* (2008), pp. 63–70.

[4] Cameron B. Browne et al. "A Survey of Monte Carlo Tree Search Methods". In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), pp. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[5] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. arXiv: 1312.5602 [cs.LG].

[6] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: 1712.01815 [cs.AI].

[7] Mogens Dalgaard et al. "Global optimization of Quantum Dynamics with Alphazero Deep Exploration". In: *npj Quantum Information* 6.1 (2020). DOI: 10.1038/s41534-019-0241-0.

[8] P.G. Benardos and G.-C. Vosniakos. "Optimizing feedforward artificial neural network architecture". In: *Engineering Applications of Artificial Intelligence* 20.3 (2007), pp. 365–382. ISSN: 0952-1976. DOI: https://doi.org/10.1016/j.engappai.2006.06.005. URL: https://www.sciencedirect.com/science/article/pii/S0952197606001072.

[9] Ehsan Zahedinejad, Sophie Schirmer, and Barry C. Sanders. "Evolutionary algorithms for hard quantum control". In: *Phys. Rev. A* 90 (3 Sept. 2014), p. 032310. DOI: 10.1103/PhysRevA.90.032310. URL: https://link.aps.org/doi/10.1103/PhysRevA.90.032310.

[10] M. Srinivas and L.M. Patnaik. "Adaptive probabilities of crossover and mutation in genetic algorithms". In: *IEEE Transactions on Systems, Man, and Cybernetics* 24.4 (1994), pp. 656–667. DOI: 10.1109/21.286385.

[11] Rammohan Mallipeddi et al. "Differential evolution algorithm with ensemble of parameters and mutation strategies." In: *Appl. Soft Comput.* 11 (Jan. 2011), pp. 1679–1696.