Alphello Zero: Learning Othello Tabula Rasa

David Kartchner, Sterling Suggs
Brigham Young University
Provo, UT 84602
`david.kartchner@math.byu.edu, sterling.suggs@byu.edu`

Abstract

A prevailing goal in artificial intelligence has been to create systems that can learn on their own to solve a given problem. Recently, researchers at Google DeepMind broke ground with their program AlphaGo Zero, which learned superhuman performance of the game of Go entirely by self-play reinforcement learning, with no human supervision Silver *et al.* (2017). The heart of the algorithm is a version of Monte Carlo Tree Search guided by a neural network. In this paper we implement the AlphaGo Zero algorithm and adapt it for the game Othello. We demonstrate the efficacy of this algorithm and show that it achieves top performance in Othello.

1 Introduction

Google DeepMind's program AlphaGo recently received wide attention as the first computer program to defeat, consistently, the world's master human Go players. AlphaGo used two neural networks, along with a version of Monte Carlo Tree Search, to suggest likely moves and estimate their value. These networks were trained with a blend of reinforcement learning, based on self play, and supervised learning, based on a dataset of human games.

AlphaGo Zero differs from the initial versions of the program in two important ways. First, it is trained exclusively by reinforcement learning, with no human supervision. Second, it uses only one neural network to suggest moves and estimate their value. DeepMind's results show that AlphaGo Zero outperforms all older versions of the AlphaGo algorithm.

In this work, we implement the AlphaGo Zero algorithm and adapt it for the game Othello. Played on an 8x8 board, with a branching factor of 10 on average, Othello is computationally a far simpler game than Go, but it remains a favorite problem for the study of AI in games because it is challenging to come up with a good heuristic to estimate move values. In the game Othello, the black and white players take turns placing double-sided tiles on the game board. A line of tiles of one color is captured by the other color when the other color sandwiches that line between tiles of its own. Sandwiches can occur horizontally, vertically, or diagonally. The captured tiles are then flipped over to the other color. The object of the game is to have most of the tiles displaying your color when the last playable square is filled.

2 Methods

We describe our version of Monte Carlo Tree Search and the neural network architecture.

2.1 Monte Carlo Tree Search

The core of the Alphello Zero algorithm is a variant of Monte Carlo Tree Search (MCTS), which uses the predictions of a neural network to guide its search.

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

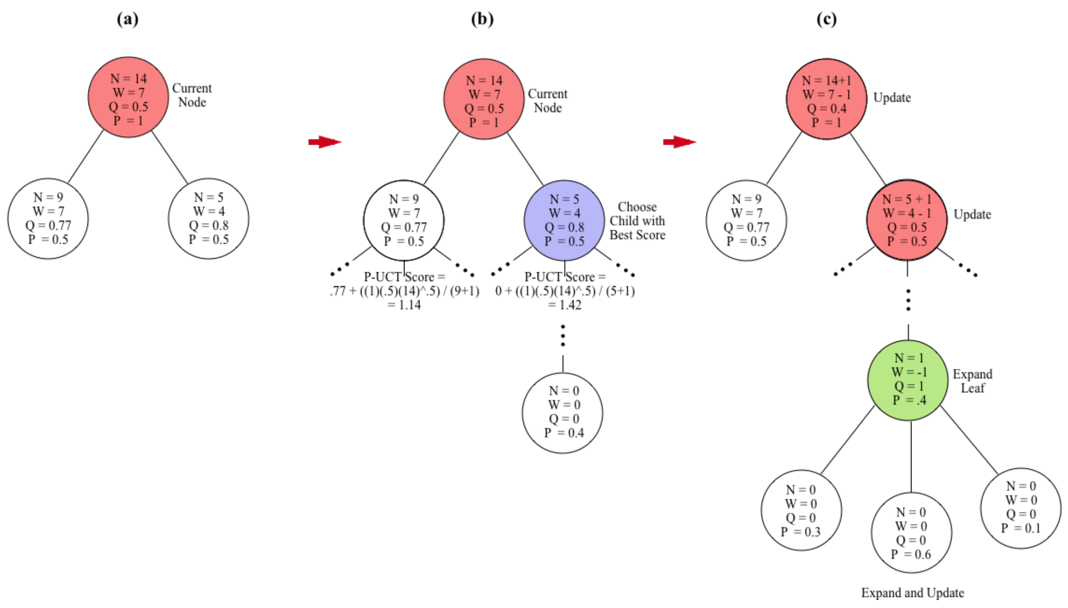

Figure 1: Visualization of how a Monte Carlo Tree Search proceeds. Part (a) shows a potential current game state, which only sees it's immediate children. In (b), the current node selects the child with the highest P-UCT score. The selected child becomes the current node and this process repeats until a leaf node is reached. In (c), the tree search finally reaches a leaf node, expands the tree, and updates the states of all parent nodes.

At its simplest, MCTS simulates a large number of random games from a given game state to estimate the value, or likelihood of winning, from that state. MCTS has proved useful in games with a large branching factor, such as Go, because it requires no domain knowledge, and it does not need to search as much of the tree before making a decision, compared with other common approaches such as minimax (Browne *et al.* , 2012).

The version used by AlphaGo Zero builds on this basis as follows. While the search tree is expanded, each node (representing a particular choice of action and resulting game state) stores four values:

- N: The number of times that choice has been taken in the tree search
- W: The total value of the current state, usually calculated as the number of wins in simulations where that move was taken. Note that this number of corresponds to estimates of winning probability from the value head of our neural network, not to a count of actual wins.
- Q: The mean value of that position (could also be considered expected game outcome, where 1 is winning and -1 is losing)
- P: The probability weight of selecting that move given by an expert policy

Actions to search are chosen based on a function of the state value and number of times the state has been visited, favoring unexplored states to ensure good coverage of the state space. To balance exploration and exploitation, the "goodness" of a new state is determined using the a Policy-based variant of the Upper Confidence Tree (P-UCT) bound:

$$g_i = Q_i + c * P_i * \frac{\sqrt{N_a}}{N_i + 1} \tag{1}$$

where c is a parameter that determines the rate of exploration, N_a is visit count of the current node, and N_i, Q_i, and P_i are the visit counts, mean value, and policy-guided probability of each of the children of the current node. When an unexplored leaf node is reached, the neural network provides the initial estimation of move probabilities and state value, and values along the search path are recursively updated. At a particular state, a single iteration of tree search proceeds as follows:

```
Result: Get Move Winning Probabilities
$a$ = current_game_state;
for j in range(maxiters) do
    while not leaf node do
        if $a$ is a leaf node then
            $W_a = ValueNetwork(a.board)$ ;
            Calculate child probs: $[P_1, P_2, ..., P_k] = PolicyNetwork(a.board)$;
            Set a.leaf = False ;
            for node in a.parents do
                node.W += $W_a$ ;
                node.N += 1;
                node.Q = $\frac{node.W}{node.N}$ ;
            end
        else
            Calculate the goodness of each possible subsequent move $i$: $g_i = Q_i + c * P_i * \frac{\sqrt{N_a}}{N_i+1}$ ;
            Update current node: a $= \text{argmax}_i\{g_i\}$ ;
        end
    end
end
```

Algorithm 1: Monte Carlo Tree Search

A figure showing the progression of MCTS is shown in Figure 1. After expanding this search tree for a predetermined amount of time or number of simulations, an action is chosen based on the visit counts of each of the children nodes. For exploratory learning, moves are chosen from a categorical distribution with $prob(i)\alpha N_i^\tau$, where τ is a temperature parameter. In competitive play, the action with the highest visit count is selected.

2.2 Neural Net Architecture

The input to the network is an 8x8 stack of 3 board planes. The first of these shows the positions of the black pieces, the second the positions of the white, and the third layer indicates which player is to play. Ours differs from the version in the paper in that we do not keep a history of game states as part of the input; we reasoned that all relevant game information is contained in the current board state, and our results suggest this was sufficient, though we made no comparisons.

This input stack is passed through a convolutional layer with 256 3x3 filters, and then through 40 residual convolutional layers (He *et al.* , 2015) of the same size. Batch normalization and ReLU activation takes place between each layer. There are two output heads of the network. The value head passes the output of the network through a fully connected layer and returns a scalar value estimate of the given board state, which represents the likelihood of the current player to win. The policy head also applies a fully connected layer to predict a vector of 65 move logit probabilities (every board square and pass). The full details can be found in Silver *et al.* (2017).

The network is trained against the move probabilities $\boldsymbol{\pi}$ suggested by the Monte Carlo Tree Search, and the eventual game winner z, from every game state experienced during self-play. The loss function contains three terms. The first applies mean squared error to the predicted winner of each state, and the second minimizes the cross entropy between move probabilities suggested by the network and those suggested by MCTS. The third term is an L2 weight regularization. In particular, the network parameters θ are adjusted by gradient descent to minimize the following function:

$$l = (z - v)^2 - \boldsymbol{\pi}^\top \log(\mathbf{p}) + c||\theta||^2$$

where $\mathbf{p}$ and v represent the policy and value estimates of the network, and c scales the weight of the regularizer (Silver *et al.* , 2017).

During self play, game states are stored along with MCTS move probabilities and the eventual winner. After every 1000 games, the collected data is used to train the network. The new trained network then competes with the old version for 400 games; if the new network beats the old 55% of the time, it is declared the new best player and plays itself to continue generating new data.

3 Results

We trained our network for several hours on a solitary GPU with a variety of learning parameters. Because a single GPU provides significantly less computing power than the 5000 TPUs that Google used, we were unable to train sufficiently long to see so dramatic results.

The algorithm has a built-in performance metric. If the newly trained network wins more games than the old one, it is objectively better. In this sense, each iteration of the network is a baseline for the next. Thus the original randomly initialized network provides the naive baseline to which all other performance is compared.

Our first experiments showed that after improving during the first few iterations, performance actually deteriorated. During training, the network's loss consistently decreased until it was reset to a previous version that consistently outperformed it in head-to-head games. The pattern then repeated until we stopped the training to reset parameters. A figure illustrating the resultant sawtooth loss pattern is included in the appendix. We varied a wide expanse of parameters including the number of residual layers and convolutional filters, the number of training loops between evaluations, and the depth of the MCTS. While our network gradually improved over a few training epochs, it took 12 hours of training on a GPU to perform comparably with a randomly initialized network using the same MCTS depth. We believe that these results could potentially be improved by substantially increasing the depth of MCTS, but we have been unable to test this assumption due to lack of sufficient computational resources.

4 Related Work

Due to the success of MCTS in games such as Go and chess, a number of papers have compared MCTS to regular $\alpha\beta$ pruning as a method (Knuth & Moore, 1975) for playing Othello. Nijssen found that MCTS fares poorly against traditional $\alpha\beta$ approaches, likely since it is hard to foretell the winner of a game until the last few moves (Nijssen, 2007). Hingston and Masek found that using weighted MCTS rollouts improved performance slightly, but still fall short of $\alpha\beta$ performance (Hingston & Masek, 2007). Others used temporal difference learning to improve MCTS on Othello, but also found similar results (Robles *et al.* , 2011; Osaki *et al.* , 2008).

5 Conclusion

Reinforcement learning is a powerful technique for learning complicated tasks. Google's paper (Silver *et al.* , 2017) shows that given enough computing power and a cleverly designed algorithm, computers can learn even the most complex of human games, and perform with superhuman abilities at specific tasks.

One important thing to note is that this algorithm, while very powerful, was designed and tuned for only one task. The application of the AlphaGo Zero algorithm to another task, namely Othello, requires non-trivial adaptation in choices of parameters, number of layers, game state representation, and so forth. Also, vast difference in available computing resources necessitated adaptation in training choices, such as how often to train, how many game histories to store, how many training loops, and how large a batch size. The sheer number of variables, and the lengthy duration of each training experiment, present a combinatoric challenge requiring extensive time to achieve optimum play. In addition, generating game data takes time; at our rate of roughly half a minute per game, DeepMind's 4.9 million games would take us over four years to generate, as opposed to their three days(Silver *et al.* , 2017).

We believe that the AlphaZero algorithm could be well suited to learn the game of Othello, but the computational cost and training time required make AlphaZero ill-suited to a game that is already well-solved by multi-move look-ahead combined with $\alpha\beta$ pruning. While AlphaZero is an important step in computer intelligence, there remains much work to be done before such gains can be realized by general AI practitioners.

References

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., Cowling, P. I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., & Colton, S. 2012. A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games*, **4**(1), 1–43.

He, Kaiming, Zhang, Xiangyu, Ren, Shaoqing, & Sun, Jian. 2015. Deep Residual Learning for Image Recognition. *CoRR*, **abs/1512.03385**.

Hingston, Philip, & Masek, Martin. 2007. Experiments with Monte Carlo Othello. *Pages 4059–4064 of: Evolutionary Computation, 2007. CEC 2007. IEEE Congress on*. IEEE.

Knuth, Donald E., & Moore, Ronald W. 1975. An analysis of alpha-beta pruning. *Artificial Intelligence*, **6**(4), 293 – 326.

Nijssen, JPAM. 2007. Playing Othello Using Monte Carlo. *Strategies*, 1–9.

Osaki, Y., Shibahara, K., Tajima, Y., & Kotani, Y. 2008 (Dec). An Othello evaluation function based on Temporal Difference Learning using probability of winning. *Pages 205–211 of: 2008 IEEE Symposium On Computational Intelligence and Games*.

Plaat, Aske, Schaeffer, Jonathan, Pijls, Wim, & de Bruin, Arie. 2014. Nearly Optimal Minimax Tree Search? *CoRR*, **abs/1404.1518**.

Robles, D., Rohlfshagen, P., & Lucas, S. M. 2011 (Aug). Learning non-random moves for playing Othello: Improving Monte Carlo Tree Search. *Pages 305–312 of: 2011 IEEE Conference on Computational Intelligence and Games (CIG'11)*.

Silver, David, Schrittwieser, Julian, Simonyan, Karen, Antonoglou, Ioannis, Huang, Aja, Guez, Arthur, Hubert, Thomas, Baker, Lucas R, Lai, Matthew, Bolton, Adrian, Chen, Yutian, Lillicrap, Timothy P., Hui, Fan Xin, Sifre, Laurent, van den Driessche, George, Graepel, Thore, & Hassabis, Demis. 2017. Mastering the game of Go without human knowledge. *Nature*, **550 7676**, 354–359.

Appendix A: Loss Plots

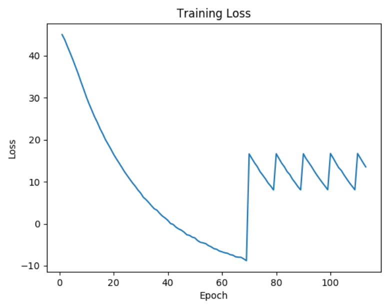

Figure 2: Loss before we stopped reverting the training net due to worse performance. Note that doing so creates a sawtooth pattern that keeps the net stuck in a problematic loop.

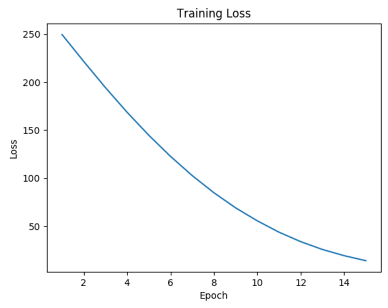

Figure 3: Loss after tweaking parameters and increasing tree search depth. While the notably decreases, this decrease doesn't seem to be very correlated with improvements in game play. We believe that this could be reflective of an overly-shallow MCTS depth that prevents the game from learning good estimations of move probabilities until near the end of the game.

Appendix B: Code

All code used for this project can be found at https://github.com/swsuggs/AlphelloZero