Learning to Play Othello Without Human Knowledge

Shantanu Thakoor
Stanford University
thakoor@stanford.edu

Surag Nair
Stanford University
surag@stanford.edu

Megha Jhunjhunwala
Stanford University
meghaj@stanford.edu

Abstract

Game playing is a popular area within the field of artificial intelligence. Most agents in literature have hand-crafted features and are often trained on datasets obtained from expert human play. We implement a self-play based algorithm using neural networks for policy estimation and Monte Carlo Tree Search for policy improvement, with no input human knowledge that learns to play Othello. We evaluate our learning algorithm for 6x6 and 8x8 versions of the game of Othello. Our work is compared with random and greedy baselines, as well as a minimax agent that uses a hand-crafted scoring function, and achieves impressive results. Further, our agent for the 6x6 version of Othello easily outperforms humans when tested against it.

1 Introduction

Game playing is a popular area within the field of artificial intelligence. One of the earliest works in this field was a checkers engine developed in (Samuel 2000), that learned through self-play and machine learning, and not through rule-based methods. An early triumph was Deep Blue (Campbell, Hoane, and hsiung Hsu 2002), a computer program capable of superhuman performance on Chess respectively, beating the top human players. These are relatively simple games, where the branching factor for each state is small, and it is easy to evaluate how good a non-terminal position is. It was estimated that games like Go, which have a large branching factor, and where it is very difficult to determine the likely winner from a non-terminal board position, would not be solved for several decades. However, AlphaGo (Silver et al. 2016), which uses recent deep reinforcement learning and Monte Carlo Tree Search methods, managed to defeat the top human player, through extensive use of domain knowledge and training on the games played by top human players.

Many of the existing approaches for designing systems to play games relied on the availability of expert domain knowledge to train the model on and evaluate non-terminal states. Recently, however, AlphaGo Zero (Silver et al. 2017b) described an approach that used absolutely no expert knowledge and was trained entirely through self-play. This new system, AlphaGo Zero, even outperforms the earlier AlphaGo model. This represents a very exciting result, that computers may be capable of superhuman performances entirely through self-learning, and without any guidance from humans.

In our work, we extract ideas from the AlphaGo Zero paper and apply them to the game of Othello. We use board sizes of 6x6 and 8x8, for which learning through self-play is more tractable on the computing resources available to us. For evaluation, we compare our trained agents to random and greedy baselines, as well as a minimax agent with hand-crafted features. We also compared against humans, and found that our 6x6 version achieves superhuman performance very quickly.

2 Related Work

Self-play for learning optimal playing strategies in games has been a widely studied area. For example, 9x9 Go has been studied in (Gelly and Silver 2008). Chess, though widely played using alpha-beta search strategies, has also seen some work through self-play methods in (Heinz 2001). (Wiering 2010) study the problem of learning to play Backgammon through a combination of self-play and expert knowledge methods.

In particular, (Van Der Ree and Wiering 2013) learn to play Othello through self-play methods, and (Nijssen 2007) apply Monte Carlo methods to Othello. For the 6x6 version, a perfect strategy for player 2 is known to exist [1].

(Silver et al. 2016) and (Silver et al. 2017b) have trained a novel neural network agent to achieve state of the art results in the game of Go. Very recently (just 4 days before submission of this report!), this approach has also been extended to a general game-playing strategy in (Silver et al. 2017a), achieving state of the art in the games of Chess and Shogi.

3 Methods

We provide a high-level overview of the algorithm we employ, which is based on the AlphaGo Zero (Silver et al. 2017b) paper. The algorithm is based on pure self-play and does not use any human knowledge except

Stanford University CS238 Final Project Report

[1] Solved by Joel F Feinstein

this is absolutely a dream come true

the rules of the game. At the core, we use a neural network that evaluates the value of a given board state and estimates the optimal policy. The self-play is guided by a Monte-Carlo Tree Search (MCTS) that acts as a policy improvement operator. The outcomes of each game of self-play are then used as rewards, which are used to train the neural network along with the improved policy. Hence, the training is performed in an iterative fashion- the current neural network is used to execute self-play games, the outcomes of which are then used to retrain the neural network. The following sections describe the different components of our system in more detail.

th tóm li là self play da vào MCTS hay NN?

3.1 Neural Policy and Value Network

We use a neural network f_θ parametrised by θ that takes as input the board state s and outputs the continuous value of the board state $v_\theta \in [-1, 1]$ from the perspective of the current player, and a probability vector $\vec{p}$ over all possible actions. $\vec{p}_\theta$ represents a stochastic policy that is used to guide the self-play.

The neural network is initialized randomly. At the end of each iteration of self-play, the neural network is provided training examples of the form $(s_t, \vec{\pi}_t, z_t)$. $\vec{\pi}_t$ gives an improved estimate of the policy after performing MCTS starting from s_t (described in Section 3.2), and $z_t \in \{-1, 1\}$ is the final outcome of the game from the perspective of the current player. The neural network is then trained to minimize the following loss function:

how do we calculate z_t? through the NN?

$$l = \sum_t (v_\theta(s_t) - z_t)^2 + \vec{\pi}_t \log(\vec{p}_\theta(s_t))$$

initial nn vs improved nn; initial policy vs improved policy

We use a neural network that takes the raw board state as the input. This is followed by 4 convolutional networks and 2 fully connected feedforward networks. This is followed by 2 connected layers- one that outputs v_θ and another that outputs the vector $\vec{p}_\theta$. Training is performed using the Adam (Kingma and Ba 2014) optimizer with a batch size of 64, with a dropout (Srivastava et al. 2014) of 0.3, and batch normalisation (Ioffe and Szegedy 2015). The code is implemented in PyTorch[2].

so nn first then MCTS?

3.2 Monte Carlo Tree Search for Policy Improvement

We use a Monte Carlo Tree Search (Browne et al. 2012) to improve upon the policy learned by the neural network. MCTS is a policy search algorithm that balances exploration with exploitation to output an improved policy after a number of simulations of the game. MCTS explores the tree where nodes represent different board configurations and a directed edge exists between two nodes $(i \to j)$ if a valid action can cause state to transition from state i to state j. For each edge, we maintain a Q value denoted by $Q(s, a)$ which is the expected reward for taking that action and $N(s, a)$ which represents the number of times we took action a from state s across different simulations. We also keep track of $P(s, \cdot) = \vec{p}_\theta(s)$, which is the prior probability of taking a particular action from state s according to the policy returned by our neural network. From these, we calculate $U(s, a)$, which is an upper confidence bound on the Q value of our edge. These values are calculated as

$$U(s, a) = Q(s, a) + c_{puct} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}$$

Here, c_{puct} is a hyperparameter controlling the degree of exploration (set as 1.0 in our experiments).

propagate v to where; how to update Q(s,a)?

When using MCTS to find a policy from a given state s, we start creating the MCTS tree with s as the root. At each step of our iteration, we calculate the action to take as the a which maximizes the upper confidence bound $U(s, a)$. If our next state already exists in our MCTS tree, we continue our simulation. If it does not exist, we create a new node in our tree and initialize its $P(s, \cdot) = \vec{p}_\theta(s)$ and the expected reward $v = v_\theta(s)$ from our neural network, and initialize $Q(s, a)$ and $N(s, a)$ to 0 for all a. We then propagate the reward v back up the MCTS tree, updating all the $Q(s, a)$ values seen during the simulation, and start again from the root. On the other hand, if we encounter a terminal state, we propagate the actual reward found from the board and restart our MCTS.

so the proba of taking a action = the stochastic policy?

Now, after a few simulations of the MCTS, our $N(s, a)$ values provide a good approximation for the optimal stochastic process from each state. Hence, the action we take is randomly sampled from a distribution π_s, with probability proportional to $N(s, a)^{\frac{1}{\tau}}$, where τ is a temperature parameter. Setting τ to a high value gives us almost uniform distribution, while setting it to 0 makes us always select the best action. τ is hence another hyperparameter controlling the degree of exploration during our learning. Hence, the training example generated from the MCTS starting at s is (s, π_s, r), where $r \in \{+1, -1\}$ which is determined at the end of the game by considering whether the current player won or lost. Pseudocode of the MCTS search is provided in Algorithm 1.

2 kinds of reward: state-reward and action-reward

is this true? 1 playout 8 examples => 800 examples per iteration?

3.3 Policy iteration through Self-play

We now describe the complete training algorithm. We initialize our neural network with random weights, thus starting with a random policy. In each iteration of our algorithm, we play a number of episodes (100 in our experiments) of self-play using MCTS. This results in a set of training examples of the form $(s_t, \vec{\pi}_t, z_t)$. We exploit the symmetry of the state space to further augment our dataset. In our experiments, since Othello is invariant to rotations and flips of the board, we thus obtain 7 extra training examples per examples in our dataset.

Then, we update our neural network using our new training examples, to get a new neural network. We then play our old and new networks against each other for a number of games (40 in our experiments). If the new network wins more than a set threshold number of times

[2] www.pytorch.org

this is reversed for implementation

how do we calculate Q value?

Algorithm 1 Monte Carlo Tree Search

```
procedure MCTS($s, \theta$)
    if $s$ is terminal then
        return $game\_result$                this is the end of an iteration
    if $s \notin Tree$ then                  if s does not exist
        $Tree \leftarrow Tree \cup s$
        $Q(s, \cdot) \leftarrow 0$
        $N(s, \cdot) \leftarrow 0$
        $P(s, \cdot) \leftarrow \vec{p_\theta}(s)$          using nn here
        return $v_\theta(s)$
    else                                     where we update
                                             information on each node
        $a \leftarrow \text{argmax}_{a' \in A} U(s, a')$
        $s' \leftarrow$getNextState($s, a$)
        $v \leftarrow$MCTS($s'$)                 is this the playout step?
        $Q(s, a) \leftarrow \frac{N(s,a)*Q(s,a)+v}{N(s,a)+1}$
        $N(s, a) \leftarrow N(s, a) + 1$
        return $v$
```

(60% in our experiments), we update the network and continue with the next iteration, resetting the MCTS tree. Else, we continue with the old network and the old MCTS tree, and conduct another iteration to augment our training examples further. Experimentally, we find that when the new network was not better than the old network, the new network obtained after a further iteration of training was far better. Hence, in one or two iterations we almost always improve our network. In our experiments, the temperature parameter τ is set to 1 for the first 25 turns in an episode, to encourage early exploration, and then set to 0. It is always set to 0 during evaluation. Pseudocode of the policy iteration algorithm is provided in Algorithms 2 and 3.

4 Experiments

The above sections describe a general approach to game-playing. In our experiments, we specifically tackled the problem of learning to play the game of Othello. Othello is traditionally played on an 8x8 sized board. The size of the state space is exponential in the size of the board. Experimentally, we found that converging to an optimal policy on the 8x8 board with limited computing resources would take a very long time. In order to show the effectiveness of our approach, we also ran experiments on a 6x6 version of Othello[3]. The 8x8 version was trained with 50 simulations of the MCTS per step, while the 6x6 version was trained with 25. Both were trained on training examples of 100 episodes per training iteration. The 6x6 version completed 78 iterations of training, while the 8x8 version completed 30 iterations of training. Both were trained for over 72 hours on a Google Compute Engine instance with a GPU.

Algorithm 2 Policy Iteration through Self-Play

```
procedure POLICYITERATIONSP                  => There will be
    $\theta \leftarrow$initNN()                        numEpisodes * 8 examples
    $trainExamples \leftarrow []$
    for $i$ in $[1, \ldots, numIters]$ do
        for $e$ in $[1, \ldots, numEpisodes]$ do
            $ex \leftarrow$ executeEpisode(nn)
            $trainExamples$.append($ex$)
        $\theta_{new} \leftarrow$ trainNN($trainExamples$)
        if $\theta_{new}$ beats $\theta \geq thresh$ then
            $\theta \leftarrow \theta_{new}$
    return $\theta$
```

Algorithm 3 Execute Episode

```
procedure EXECUTEEPISODE($\theta$)
    $examples \leftarrow []$
    $s \leftarrow$ gameStartState()
    while True do
        for $i$ in $[1, \ldots, numSims]$ do
            MCTS($s, \theta$)
        $examples$.add(($s, \pi_s$, _))
        $a^* \sim \pi_s$
        $s \leftarrow$ gameNextState($s, a^*$)
        if gameEnded($s$) then
            //fill _ in examples with reward
            $examples \leftarrow$assignRewards($examples$)
            return $examples$
```

4.1 Baselines

We implemented two baselines for comparison with our trained AI player. The first is a greedy player that always chooses a move that causes the maximum number of flips in the next step of the game. The second is a random player baseline. A random player chooses from one of the valid moves randomly at each step in the game.

We also used a minimax agent[4] as a third baseline which tries to maximize the worst-case gain assuming that the opponent plays perfectly at each move by exploring the game tree up to a certain depth. The results of the different baselines are listed in Table 1.

4.2 Human Evaluation

We also implemented an interface where a human player can play against any of our baselines or our learned strategies. For the 6x6 version, we evaluated our bot against a local player who has been playing Othello from childhood. These results are also available in Table 1. Since the 8x8 version took a lot more time to train, we did not get a chance to evaluate against humans.

4.3 Analysis of Experiments

We analyze our performance as a function of training time. In Figure 1 and Figure 2, we plot our performance against the greedy and random baselines against the

[3] Environment adapted from `https://github.com/JaimieMurdock/othello`

[4] From `https://github.com/Zolomon/reversi-ai`

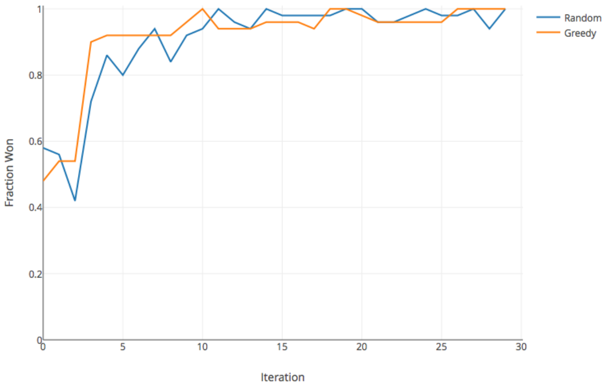

Figure 1: Performance Against Random and Greedy baselines over 30 iterations (6x6)

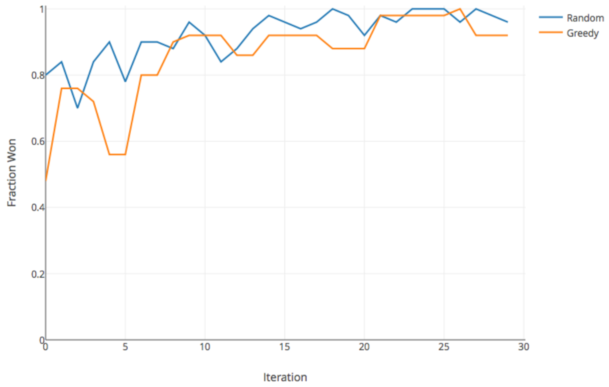

Figure 2: Performance Against Random and Greedy baselines over 30 iterations (8x8)

Baseline	6x6 board	8x8 board
Greedy	20/20	20/20
Random	20/20	18/20
Minimax	30/30	29/30
Human	6/6	-

Table 1: Number of games won against various baselines by our final models

number of iterations trained. As we see, these simple baselines are quickly beaten by the 6x6 version in a few iterations. However, learning a good agent for the 8x8 bot is much more difficult. If performance against a more sophisticated baseline is observed, it would help decide when our model has converged and we can stop training.

Aside from the comparisons against baselines, we follow the (Silver et al. 2016) approach of analyzing the games played by our agent, to try and understand its strategies. In Figure 3 we examine our agent's early game strategies against the minimax bot. Our agent is black, while the opponent is white. Boards are shown on each turn after our agent has made a move. We see that the strategy adopted is to quickly grow towards the walls and corners, and capture the pieces there. This is indeed a strong high-level strategy that human players use, since pieces at corners and walls are very difficult for the opponent to flip. It is quite remarkable that our agent is able to display such subtle strategies through self-play, even against a strong minimax opponent.

In Figure 4, we examine some late game moves of our agent against the minimax strategy. We observe that 4 moves before the end, in terms of number of pieces we do not appear to be performing significantly different from our opponent. However, by the endgame our agent has learned to position its pieces very strategically. Instead of placing a position in a place which would maximize the number of flips in one move (as the greedy baseline would do), it places them in such a way that the opponent has no moves left and is forced to pass. Hence, it can quickly cover a larger portion of the board without the opponent moving and thus completely dominate the board at the end of the game.

5 Conclusions

We implement an agent that learns to play Othello through pure self-play, without using any human knowledge. Our agent convincingly beats all baselines including greedy, random and the standard alpha-beta minimax AI baseline. Further, the time taken to make a move is much less for our agent, since it is just a feed forward operation in a neural network, compared to the minimax algorithm, which involves exploring an exponential state space to a large depth to get good results. As seen in Section 3, our framework is very generic in its implementation, and can be easily extended to many other games such as Chess or Go.

The original implementation by DeepMind (Silver et al. 2017b) uses orders of magnitude more raw computational power on industry hardware (4TPUs, 64GPUs, and 19CPUs, for several days). In our work, we show that it is possible to train similar networks on commodity hardware for smaller problems. We plan to release our implementation for the open source community.

References

[Browne et al. 2012] Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1):1–43.

[Campbell, Hoane, and hsiung Hsu 2002] Campbell, M.; Hoane, A.; and hsiung Hsu, F. 2002. Deep blue. *Artificial Intelligence* 134(1):57 – 83.

[Gelly and Silver 2008] Gelly, S., and Silver, D. 2008. Achieving master level play in 9 x 9 computer go. In *AAAI*, volume 8, 1537–1540.

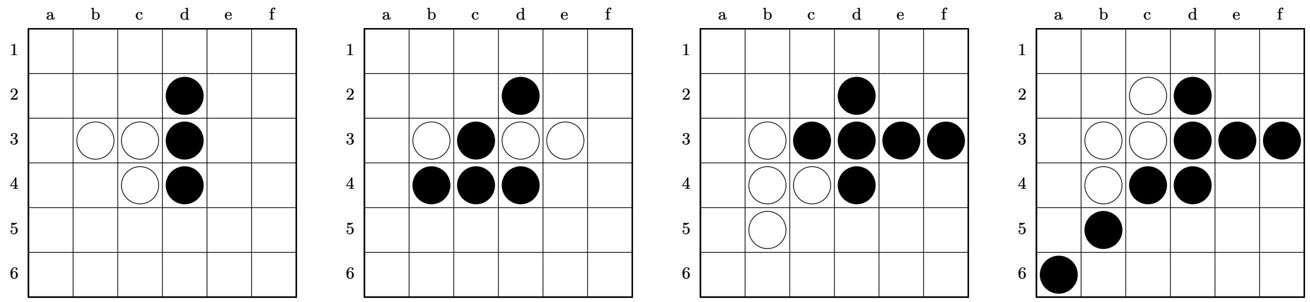

Figure 3: Early game play of our agent(B) vs minimax(W), capturing walls and corners

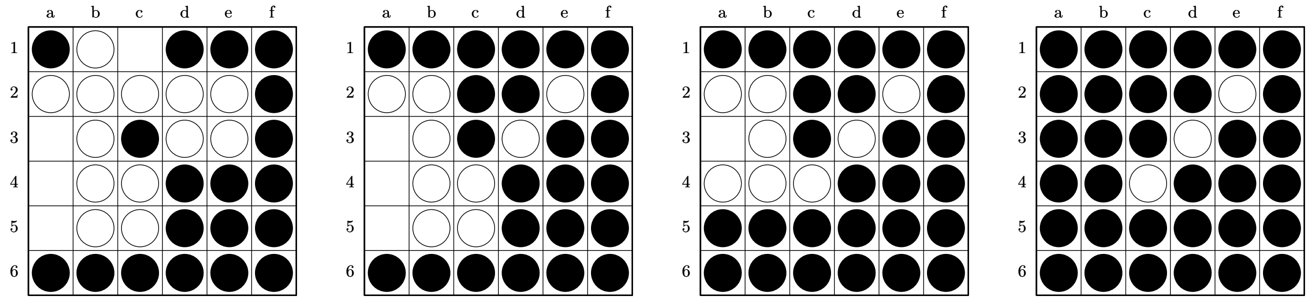

Figure 4: Late game play of our agent(B) vs minimax (W), forcing passes

[Heinz 2001] Heinz, E. A. 2001. *New Self-Play Results in Computer Chess*. Berlin, Heidelberg: Springer Berlin Heidelberg. 262–276.

[Ioffe and Szegedy 2015] Ioffe, S., and Szegedy, C. 2015. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International Conference on Machine Learning*, 448–456.

[Kingma and Ba 2014] Kingma, D., and Ba, J. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[Nijssen 2007] Nijssen, J. 2007. Playing othello using monte carlo. *Strategies* 1–9.

[Samuel 2000] Samuel, A. L. 2000. Some studies in machine learning using the game of checkers. *IBM Journal of Research and Development* 44(1.2):206–226.

[Silver et al. 2016] Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; van den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; Dieleman, S.; Grewe, D.; Nham, J.; Kalchbrenner, N.; Sutskever, I.; Lillicrap, T.; Leach, M.; Kavukcuoglu, K.; Graepel, T.; and Hassabis, D. 2016. Mastering the game of go with deep neural networks and tree search. *Nature* 529(7587):484–489. Article.

[Silver et al. 2017a] Silver, D.; Hubert, T.; Schrittwieser, J.; Antonoglou, I.; Lai, M.; Guez, A.; Lanctot, M.; Sifre, L.; Kumaran, D.; Graepel, T.; Lillicrap, T.; Simonyan, K.; and Hassabis, D. 2017a. Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. *ArXiv e-prints*.

[Silver et al. 2017b] Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017b. Mastering the game of go without human knowledge. *Nature* 550(7676):354–359.

[Srivastava et al. 2014] Srivastava, N.; Hinton, G. E.; Krizhevsky, A.; Sutskever, I.; and Salakhutdinov, R. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research* 15(1):1929–1958.

[Van Der Ree and Wiering 2013] Van Der Ree, M., and Wiering, M. 2013. Reinforcement learning in the game of othello: learning against a fixed opponent and learning from self-play. In *Adaptive Dynamic Programming And Reinforcement Learning (ADPRL), 2013 IEEE Symposium on*, 108–115. IEEE.

[Wiering 2010] Wiering, M. A. 2010. Self-play and using an expert to learn to play backgammon with temporal difference learning. *Journal of Intelligent Learning Systems and Applications* 2(02):57.