International Research Journal of Engineering and Technology (IRJET) e-ISSN: 2395-0056
Volume: 07 Issue: 06 | June 2020 www.irjet.net p-ISSN: 2395-0072

Accelerating AlphaZero's Performance for chess

Yash Aggarwal[1], Shashank S[2]

[1]*Student, Dept. of Computer Science and Engineering, Mysore, India*
[2]*Asst. Professor, Dept. of Computer Science and Engineering, Mysore, India*

---***---

Abstract - *This paper presents several improvements to Google's AlphaZero chess engine. A wide range of chess engines have been created, either by using handcrafted features or machine learning algorithms, with the purpose of pushing the boundaries of what a computer can do and to understand the game itself. Under this project, several studies have been performed to further improve AlphaZero's performance, both by analyzing existing engines and several reinforcement learning technologies. Ultimately, a chess engine, MantisShrimp was designed that is able to learn the game features much faster than AlphaZero. This was majorly achieved through modifying the deep learning network used by AlphaZero and changing the learning methodology. The loss function and action representation were also modified to force the network into learning domain specific features. Instead of just self-play, the network underwent supervised training for a short period to speed up the learning process, fine tune the hyper-parameters and prevent initial learning bias.*

Key Words: chess, reinforced learning, deep learning, zero-sum game, monte carlo tree search, minimax

1. INTRODUCTION

A co-relation has always been made between chess and intelligence. It was, at a time, considered impossible that a computer would ever be able to play chess. Chess is an extremely complex game which requires unparalleled reasoning and creativity and as such, it is an extremely difficult task to program a computer to play chess at a reasonably good level.

In 1997, IBM's deep blue defeated the then reigning champion and FIDE Grandmaster Gary Kasparov setting a milestone for computer intelligence. In the coming years, both the hardware and software advanced computers so far that no human can you hope to match against any standard chess program. A human searching 5-6 positions in future will obviously overpowered by the brute force of a machine that can analyze hundreds of positions per second. While the early chess programs like Deep Blue leveraged their hardware and relied on their brute computational power, modern chess programs have focused on the software to improve their performance. The open source chess engine Stockfish can now easily defeat the top player in the world running on a low-end smartphone.

Any chess program has 2 major components: move search and move evaluation. The programs from last decade like Stockfish and Komodo uses a tree-based searching from a position to create a game tree till a specified depth, and then evaluates these positions using an algorithm to decide the next best move. The evaluation algorithm consists of handcrafted features and positions determined after long research and statistical analysis. These programs play the game in a piece-centric manner, as opposed to position focused approach taken by human players.

Researchers have tried to replace this evaluation function with machine learning technologies, but it wasn't until Google's AlphaZero in 2017 that machine learning made a serious dent in the field. AlphaZero's move searching algorithm is similar or comparable to the search algorithms of traditional engines, but the evaluation function is actually a Convolutional Neural Network, similar to the ones used for state of art image recognition technologies. This project aims at improving AlphaZero's performance at playing chess through changing the deep learning architecture and modifying the learning methodology. Instead of depending on self-play, initial learning is done through having it play against Stockfish, and it is only allowed to play against itself, after it has learned a certain level. Some new advancements in deep learning were utilized to further speed up the learning process.

2. BACKGROUND AND RELATED WORK

2.1 Stockfish

Minimax is a recursive algorithm for traversing the game move tree used by Stockfish. It is similar to DPS algorithm that traverses the tree assuming that opponent always picks the most optimal move. A game of chess, however, has an average branching factor of 35 and an average game has around 40 moves. So during a game, the Minimax is usually done to some fixed depth which is decided based on the game type. When the maximum depth is reached, the evaluation function is used to assign a score to those leaf positions instead of going to higher depths. Quiescent search is also used to avoid horizon effect. There can be some positions that looks bad at current max depth but would have been bad if maximum depth were increased. Horizon effect can occur even if we keep increasing maximum depth. Such troublesome positions need to be eliminated by going deeper in every search process until the leaf nodes are all relatively quiet positions. Quiet positions in chess are chess are ones that are not a capture, check or immediate threat to the opposition. These positions do not possess any horizon effect. Stockfish uses 2 separate functions, one for endgame evaluation and one for middlegame evaluation. During the start, middle-game score has higher weightage. As the game progresses, scale factor for endgame score increases and phase decreases making endgame score more prominent.

2.2 AlphaZero Chess

Monte Carlo tree search or MCTS is the search algorithm used by Alpha Zero instead of Minimax. Minimax has a major disadvantage in that it tends to scan all or atleast more than half branches even with $\alpha-\beta$ pruning. This is highly impractical for a game like chess. MCTS overcomes this by not going to the full depth of every branch and works in a probabilistic fashion. AlphaZero uses PUCT instead of UCB1 for traversing the search tree [1]. The evaluation function is replaced by a CNN here, with ResNet modules to prevent diminishing gradient problem.

3. MantisShrimp

3.1 Architecture

The input state is similar to that of AlphaZero. MantisShrimp uses 5 step history instead of AlphaZero's 8 step history. Some redundant planes have been combined in the input such that the input highlights long plies of slow progress. A major difference is in action representation. While AlphaZero automatically masks illegal moves predicted by the neural neural network in each stage of tree search, MantisShrimp is forced to only make legal moves. This is done by modifying loss function such that the network is heavily penalized for illegal moves. The loss function uses relative entropy (Kullback–Leibler divergence [10]) rather than cross entropy to further emphasize the illegal move error.

Table -1: MantisShrimp Input representation

Feature	Planes
P1 Piece	6
P1 Piece	6
Repetitions	2
	x5
Color	1
P1 castling	2
P2 castling	2
Half-move Count	3
Total	78

$$loss = (z-v)^2 - \sum_i^{H\times W\times C} \pi(i)\mathbf{q}(i)\log\frac{\mathbf{p}(i)}{\pi(i)} + \lambda||\theta||$$

z = Output from value head.

v = 1 if move resulted in a win, -1 if loss and 0 if draw.

π = Expected probability distribution.

p = Probability distribution by Neural Network after scaling of illegal move probabilities.

q = Vector for illegal move penalization. If the corresponding move is legal, $\mathbf{q}(i)$ is 1 and if move is illegal, $\mathbf{q}(i)$ is 10.

λ = Regularization Coefficient.

θ = Network Weights.

By penalizing the network for illegal moves, we expect it to learn the game's features better, and it would also prevent internal layers from focusing on details that will not lead to a legal move.

The ResNet blocks in the original network have been modified with squeeze-and-excitation's global pooling paths. This is expected to significantly reduce channel co-dependencies [4]. Fig. 1 shows a Squeeze-and-Excitation block. While normal CNN blocks give equal weightage to all channels, the Squeeze-and-Excitation path learns and acts as a weight for different channels.

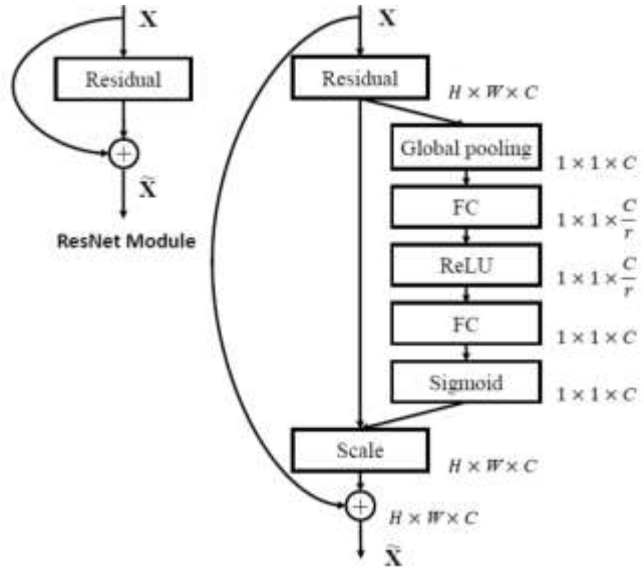

Fig. 1: Converting a ResNet module to squeeze-and-excitation ResNet module

Squeeze-and-Excitation blocks show prominent effects in deeper layers where the Convolutional channels have more pronounced features. AlphaZero's use of modified PUCT brings about another problem. Functions like PUCT and UCB1 are designed to balance exploitation and exploration. The original function has been modified to also account for probability given by neural network *P(S)*.

$$PUCT(S_i) = \overline{V_i} + C_{puct}.P(S_i).\sqrt{\frac{\ln N}{1+n_i}}$$

S_i = ith state

V_i = Average score of ith state

C_{puct} = Exploitation parameter. A constant used to balance exploitation and exploration

N = No of visits of parent ith state

n_i = Visit count for state

$P(S_i)$ = The probability of node from the CNN network.

If any move has been visited fewer times compared to other moves, the algorithm would want to explore that move more. Exploration refers to visiting less visited nodes and Exploitation refers to visiting nodes with higher chance of win. The constant Cpuct is used to balance the two. But if the prior probability of that move is extremely low according to the neural network, it might not be visited even with very low visit count. This will cause an unintended horizon effect. To avoid this, the child moves from root with very low visit count have their PUCT score set to infinite so they are visited even if network assigns them really low probabilities. The game of chess can have average branching factor of 35-38. Since every Monte Carlo Tree Search (MCTS) does have maximum iterations set to 800, it was decided to modify PUCT such that all moves that have been visited less than 0.5%, i.e. less than 4 times will be visited first.

So that MCTS does not always select same move, a randomness is introduced through temperature τ. Temperature allows MCTS to select moves randomly from a range of moves that are close to best, rather than selecting best moves every time. A temperature of 10 means the moves selected will have probability ±1%. While AlphaZero kept temperature constant, MantisShrimp varies it for some games to have more variation in training set.

3.2 Training Configuration

The network weights are initialized as random. Instead of using self-play from start, the network is first trained on 500 games played by top players to speed up learning initially. The network then does 10 steps of training by playing against Stockfish in batches of 500 games. This kind of supervised learning prevents the network from acquiring bias in the beginning. Other option to avoid such bias is to have game start from some popular opening positions as done by AlphaZero. MantisShrimp uses both.

At this point, the program is made to play against itself, 800 games a batch. The MCTS maximum iterations is limited to 800 as well. A replay buffer/queue of latest 80000 games is maintained for the network to learn from. The network samples 2048 games from the replay buffer for a step of training. The new network is then made to play against the old network in a 1000 game match. The losing network is discarded and the process is started again with new network.

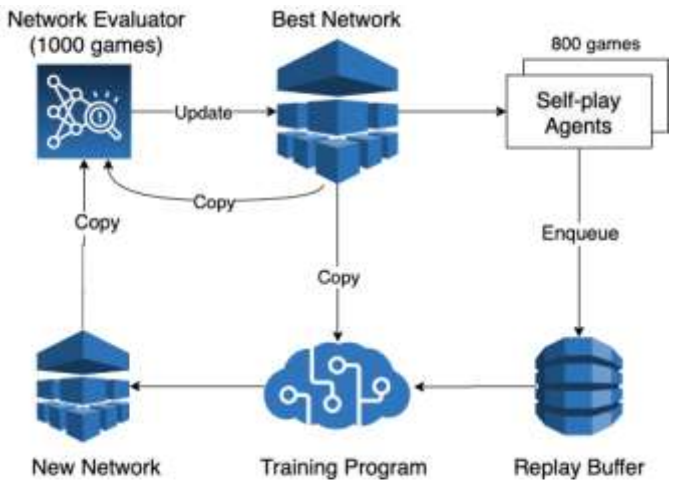

Fig. 2: Learning Process

It was observed through experiments that Stockfish performs better than LeelaZero (the best performing open source AlphaZero clone) during endgames, when they are made to continue a game position by human players. A major reason for that can be lower variation in endgames for the games the network is trained on. To solve this, temperature τ is kept low during opening and is gradually increased as game reaches the endgame phase for 40% of the self-play games.

3.3 Performance Evaluation and Results

ELO is a rating system used in chess that assigns a relative score to players. MantisShrimp is written in python so it cannot be accurately compared against any mainstream chess programs. Therefore, an AlphaZero clone was created to

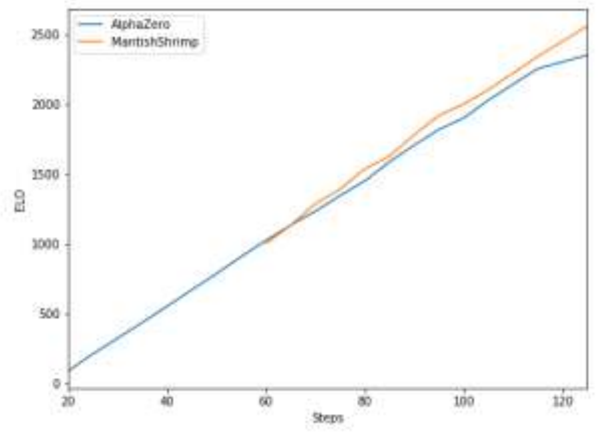

Fig. 3: Relative ELO for AlphaZero and MantisShrimp. MantisShrimp's ELO before self-learning is 1003, so its plot has been shifted for better comparison

compare learning speed for the two programs. If ELO is known for one program, ELO for another program can be determined by making the two programs play against each other. Stockfish 11 was used for this comparison. It's ELO was set at 4000 and the programs were made to play against it per 5 steps of training. Fig. 3 compares relative ELO of the two engines. AlphaZero was trained as in the original paper. The increase is ELO is almost linear till an ELO of 2300 for both programs. MantisShrimp is able to demonstrate a significant increase in learning rate which is only going to be more pronounced as the training continues. Since the program is written in python, it cannot be fairly compared against popular engines. Three extra hyper-parameters are introduced in MantisShrimp: The rate of change for τ, penalty for illegal moves, and minimum playouts needed per node N_{min}. Further tuning of these would definitely improve the learning rate even more. These results also demonstrated that when the network is heavily penalized for wrong moves instead of masking them, the network has a better understanding of the game. This would have slowed down gradient descent as such a loss function would make learning much slower due to irregular gradient. The increase in speed from squeeze-and-excitation network helped make up for that.

4. FUTURE WORK

Current plans for future work are limited to rewriting the program in C++ and optimizing the code to allow for a robust comparison with other popular engines. There were, however, several ideas that were not explored due to time limitations. One possibility is to reward the network for choosing moves that end the game in less moves. As of now, the value head assigns a score of 1 for a win which can be changed to provide lower loss for games won in shorter time. Another avenue to explore would be more diverse training data. Dirichlet noise or temperature may be increased during opening phase for a percentage of game. It might also be possible to modify MCTS for better performance. MCTS's random rollout has been replaced with Neural Network evaluation, but random rollouts can be considered when evaluation function is too expensive. Random rollouts can be done some *x*% times. If a good balance is found, the time taken per move can be reduced significantly without considerable hit to performance. Possibility of some form of Quiescent search can also be investigated for Monte Carlo Search Tree algorithm.

ACKNOWLEDGEMENT

My deep gratitude to Shashank S, Asst. Professor in the Department of Computer Science & Engineering at National institute of Engineering, Mysore, for his guidance and support during the course of this project.

REFERENCES

[1] Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T.P., Simonyan, K., & Hassabis, D. (2017). Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm. ArXiv abs/1712.01815.

[2] Stockfish evaluation guide. hxim.github.io/Stockfish-Evaluation-Guide

[3] P Auer, N Cesa-Bianchi, P Fischer. Finite-time analysis of the multirmed badit problem. Machine learning 47 (2-3), 235-256.

[4] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, Enhua Wu. Squeeze-and-Excitation Networks. ArXiv abs/1709.01507.

UCI: Communication Protocol for chess engines. wbec-ridderkerk.nl/html/UCIProtocol.html

[5] Fuller, Samuel H.; Gaschnig, John G.; and Gillogly. Analysis of the alpha-beta pruning algorithm (1973). Computer ScienceDepartment.Paper 1701

[6] The Secret of Chess. Lyudmil Tsvetkov. ISBN-10: 1522041400. ISBN-13: 978-1522041405

[7] Sylvain Gelly, Yizao Wang. Exploration exploitation in Go: UCT for Monte-Carlo Go. NIPS: Neural Information Processing Systems Conference On-line trading of Exploration and Exploitation Workshop, Dec 2006, Canada. ffhal-00115330

[8] Kullback, S.; Leibler, R.A. (1951). On information and sufficiency. Annals of Mathematical Statistics. 22(1): 79-86. doi:10.1214/aoms/117772969