SHA-ZA: Advanced Reinforcement Learning for Othello Mastery Using Proximal Policy Optimization

Mohammed Yousif

Independent Researcher
Email: Mohammed.yah.yousif@gmail.com (M.Y.)
Manuscript received July 12, 2024; revised August 3, 2024; accepted September 5, 2024; published February 25, 2025

***Abstract*—This paper introduces SHA-ZA (Strategic Heuristic Agent with Zero-human Advancement), an advanced reinforcement learning agent trained to master the game of Othello, drawing inspiration from DeepMind's AlphaZero, which achieved exceptional proficiency in chess, shogi, and Go through self-play and reinforcement learning. SHA-ZA employs similar methodologies, utilizing self-play with multiprocessing and Proximal Policy Optimization (PPO) to achieve superior performance without prior human knowledge.**

Trained on the equivalent of over 650 years of continuous human experience, totaling 33,587,200 games, SHA-ZA underwent rigorous testing against diverse opponents, resulting in significant strategic gameplay advancements. The findings illustrate SHA-ZA's ability to surpass advanced-level minimax engines, highlighting the effectiveness of combining PPO and self-play for mastering complex board games like Othello.

***Keywords*—reinforcement learning, Othello, AlphaZero, self-play, Proximal Policy Optimization (PPO), board games, artificial intelligence**

I. Introduction

The strategic board game Othello, commonly referred to as Reversi, was created in 1883 and is credited to Lewis Waterman and John W. Mollett. The game's modern form was popularized by Goro Hasegawa in Japan in 1971, and it quickly gained international acclaim, becoming a staple in competitive tournaments by 1977 [1]. The enormous state space and strategic complexity of Othello provide a significant challenge to artificial intelligence.

Although the exact complexity remains elusive, estimates suggest it is on the order of 10^{28} [2], a magnitude surpassing 1.33×10^{9} times the number of grains of sand on Earth. To appreciate this scale, envision the hypothetical scenario of generating one billion Othello positions per second, a task demanding over 3.1797×10^{11} years—23.09 times the age of the universe.

AlphaZero's [3] groundbreaking approach involved learning to play various complex games entirely through self-play. The generation of games utilized substantial hardware resources, with time per move comparable to classical search algorithms. This computational complexity is due to the use of Monte Carlo Tree Search (MCTS) [4], which performs a series of simulated games during each search.

In recent years, the landscape of reinforcement learning has seen significant advancements with the introduction of Double Deep Q-learning [5] and policy gradient algorithms such as Trust Region Policy Optimization (TRPO) [6] and its enhanced version, Proximal Policy Optimization (PPO) [7], which is widely used in Reinforcement Learning with Human Feedback (RLHF) [8]. Furthermore, asynchronous methods like Asynchronous Advantage Actor-Critic (A3C) [9] and its synchronized variant, Advantage Actor-Critic (A2C) [10], have further enhanced the effectiveness and stability of reinforcement learning algorithms.

In addition, recent advancements in deep learning techniques have significantly accelerated convergence rates and improved overall performance. SHA-ZA was trained using a collection of advanced reinforcement learning and deep learning techniques. Rather than running parallel simulations, the next move is inferred directly from the policy network. SHA-ZA achieves markedly faster execution times compared to its rivals, requiring less hardware for training while demonstrating superior gameplay performance.

II. Literature Review

The game of Othello has served as a fertile ground for experimentation with various reinforcement learning techniques. From early off-policy temporal difference algorithms like Deep Q-learning [11] and Double Deep Q-Learning [5], to different evolutionary algorithms, and eventually Monte-Carlo Tree Search (MCTS) [4].

The state representation in [2], which documents an early use of Q-learning, was based solely on the arrangement of white and black pieces. Rewards were deferred until the game's conclusion, contingent solely upon the outcome. The study investigated two feedforward multilayer network architectures:

1) **Single-NN Q-learner**: Implemented a single multi-layer feed-forward neural network with distinct outputs corresponding to each action.
2) **Multi-NN Q-learner**: Employed a separate multi-layer feed-forward neural network for each action.

A deterministic strategic policy that prioritized corner locations and the enhancement of player's mobility had been utilized for evaluation. After training for 15 million episodes, both agents outperformed the static mobility policy. Notably, the multi-NN Q-learner demonstrated a win rate of 79%.

The study in [2] could benefit from a more sophisticated state representation. Additionally, comparing performance against stronger opponents would provide a more reliable assessment.

The combination of Temporal Difference Learning (TDL) and Coevolutionary Learning (CEL) yielded a novel algorithm, which is described in [12]. CEL operates on the principle of survival and adaptation among a population's fittest players, whereas TDL compares current predictions with actual future rewards. These two concepts are integrated into the CTDL algorithm, which manages the population and alternates between TDL and CEL phases.

CTDL surpassed both TDL and CEL in performance. It is

suggested that the population variance introduced by CTDL prevents TDL from becoming stuck in suboptimal strategies, while also leading to superior long-term strategies compared to CEL alone. However, the use of evolutionary algorithms does not usually allow for the training of larger architectures due to computational overhead.

An implementation of concepts derived from AlphaZero's [3] approach applied to Othello is discussed in [13]. Monte-Carlo Tree Search (MCTS) was employed to enhance the learning process and balancing exploration and exploitation. The agent displayed constant improvement against both random and greedy policies, eventually reaching superhuman performance.

OLIVAW's [14] training process has demonstrated a more efficient approach to MCTS and self-play, achieving this through three key changes:

1) **Richer Training Data:** The process utilizes training data that includes positions explored frequently during MCTS, rather than only the moves that were played.
2) **Dynamic MCTS Simulations**: Instead of employing a fixed number of simulations, this approach allocates fewer simulations (100, 200, or 400) during the earlier training generations. As the agent improves in later generations, the number of simulations is increased (up to 400), compared to the fixed 1600 simulations used by AlphaZero.
3) **Dynamic Training Window:** The number of generations included in the training process gradually increased from the last 2 to the last 5 as training progresses. In contrast, AlphaZero uses a fixed window of the last 5×10^5 games (approximately 20 generations) for its training data.

Evaluation was conducted through a series of games against human experts, where OLIVAW demonstrated superhuman performance. The training process lasted 30 days using Google TPU v2-8 on Colaboratory [15].

The approaches detailed in [13] and [14] continue to rely on multiple simulations (playouts) to make a single decision. Furthermore, it remains uncertain if these methods have incorporated recent advancements in deep neural networks.

III. Materials and Methods

A. PPO Loss Function Implementation

Proximal Policy Optimization PPO [7]: is an on-policy reinforcement learning algorithm created by OpenAI. It optimizes the policy using a clipped surrogate objective function, proximal policy loss consists of three losses:

1) **Policy Surrogate Loss:** use clipped probability ratio to prevent large, potentially harmful updates.
2) **Value Loss:** Mean squared error loss between predicted and actual returns better say regression error.
3) **Entropy Loss:** Encourages exploration by adding entropy to maintain randomness in action selection.

$$L_{PPO} = L_{clip} + c_1 \cdot L_{vf} - c_2 \cdot L_{entropy} \quad (1)$$

Here, c_1 and c_2 are, in order, the value and entropy coefficients. These two control the general behavior of the algorithm, including stability and exploration.

The surrogate loss function computes the clipped ratio between the new policy π_θ and the old policy $\pi_{\theta\,old}$ at state s_t The probability ratio for an action a_t is given by:

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta old}(a_t|s_t)} \quad (2)$$

$$cr(\theta, \epsilon) = clip(r_t(\theta), 1-\epsilon, 1+\epsilon) \quad (3)$$

This ratio $r_t(\theta)$ measures how much the probability of choosing an action a_t under the new policy π_θ has changed relative to the old policy. ϵ is a training hyperparameter used in calculating the clipped ratio $cr(\theta, \epsilon)$, which influences the stability of the training process. The advantage A_t is defined as the difference between the actual and expected returns:

$$A_t = R(s_t, a_t) - V(s_t) \quad (4)$$

$R(s_t, a_t)$ is the return generated from taking an action a_t at a state s_t through the interaction of the agent with the game simulation. The complete formula for surrogate policy loss can be written as:

$$L_{clip}(\theta) = -E_t[\min(r_t(\theta) \cdot A_t\,,\, cr(\theta, \epsilon) \cdot A_t)] \quad (5)$$

Mean Square Error MSE has been used for calculating value loss L_{vf}:

$$L_{vf} = MSE\big(V(s_t), R(s_t, a_t)\big) \quad (6)$$

The entropy loss $L_{entropy}$ encourage exploration by penalizing certainty in action selection. This helps the agent avoid premature convergence to a suboptimal policy by maintaining a higher level of randomness in its actions:

$$L_{entropy} = -\sum_a \pi_\theta(a \mid s_t) \cdot \log\big(\pi_\theta(a \mid s_t)\big) \quad (7)$$

The PPO algorithm was implemented in PyTorch, with the 'PPO_Loss' class encapsulating the loss computation, including the implementation of equations 1 through 7. The 'forward' method in this class takes inputs such as 'old_policy', 'new_policy', 'actions', 'value_head_output', and 'return_values' to compute the total loss. Detailed code implementations can be found in the Appendix.

B. Simulation Environment, State Representation, and Exploration Strategies

A custom simulation environment compatible with PyTorch was developed using NumPy and Numba JIT for high-performance board operations and return computations. Additionally, a custom Minimax Engine with Alpha-Beta pruning was implemented similarly to measure the agent's performance more accurately. The 'self_play' method within the 'OthelloBoard' class accepts a model, shuffle depth, and device type as arguments, and returns pairs of boards and return values for each. The discount factor γ is automatically calculated when the board is initiated using the following formula:

$$\gamma = e^{\frac{\ln(0.01)}{size^2-4}} \quad (8)$$

Where $size$ here is the board size, this version was train on boards size of 8 ($size = 8$). The 'parallelSelfPlay' function

in the '_othello' module leveraged Python's multiprocessing module to efficiently run parallel simulations. Detailed code implementations for these components are also provided in the Appendix.

The Board state s_t at time t is represented using three components:

1) Legal Moves: positional representation of all legal moves an agent can take at a given state s_t. This is also used to restrict the agent from choosing invalid moves.
2) Pieces Under Attach: positional representation of all opponent's pieces that can be captured within a single move.
3) Board: All player's pieces are represented by 1, and all opponent's pieces are represented by -1.

Exploration is integral to any reinforcement learning algorithm. In this study, each self-played game starts with a random board state that is shuffled with a random number of moves, between 0 and 32. Experiments were conducted to incorporate $\varepsilon - greedy$ [16] exploration, but it was found to marginally slow down progress. Consequently, ε was set to 0.0 during training the last version of SHA-ZA. The output of the Policy network comprises a 2D grid of probabilities. During self-play tournaments, each permissible action is assigned a nonzero probability of selection, akin to Boltzmann exploration [16].

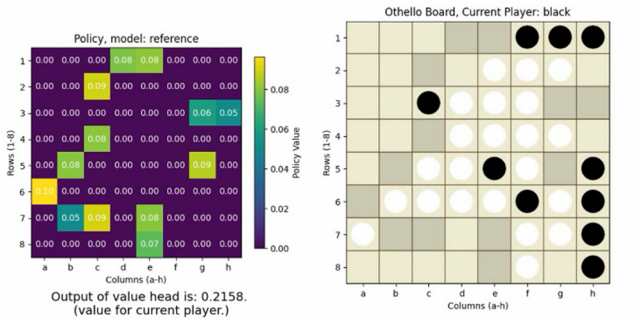

Fig. 1. On the right: the board state, showcasing a game position. On the left: the probability distribution output of the policy network, indicating the model's decision-making process for the next move.

Unlike Monte Carlo Tree Search (MCTS) [4] based methods, this approach selects actions directly from the probability distribution generated by the policy network, as depicted in Fig. 1, rather than through simulations.

While training, rewards are assigned only at the end of each game. A win results in a reward of 1.0, a draw is rewarded with 0.0, and a loss incurs a reward of -1.0. The number and specific locations of player's pieces do not contribute to the final reward.

C. Neural Network Architecture and Attention Mechanism

SHA-ZA's agent network architecture took advantage of recent advancements in the landscape of deep learning. The agent consisted of two networks: the policy network and the value network. The policy network maps the input state to a probability distribution over all actions, while the value network provides a holistic evaluation of the state. Both networks are based on ConvNeXt [17], a modern convolutional neural network architecture inspired by the design principles of vision transformers (ViTs) [18]. Leaky ReLU was used in place of the original GeLU activation function in the ConvNeXt block to increase inference speed.

The Convolutional Block Attention Module (CBAM) [19] is another essential element that quickens the learning process. This update improves the neural network's feature representation by highlighting significant features and stifling unimportant features. It is divided into two consecutive sub-modules:

1) Channel Attention Module: This emphasizes informative channels and suppresses less useful ones by computing channel-wise attention.
2) Spatial Attention Module: This enhances important spatial features and suppresses irrelevant ones by computing spatial attention maps.

Attention mechanisms are often applied at the end of a convolutional block, as demonstrated in [20], [21], and [22]. SHA-ZA distinguishes its architecture by positioning the CBAM layer after the expansion in the inverted bottleneck, as illustrated in Fig. 2's convolutional block architecture. This strategic placement enriches channel attention with a more diverse feature set, enabling subsequent layers within the block to leverage enhanced representations. Preliminary experiments with CBAM placement revealed that convergence was accelerated by this configuration for the problem at hand.

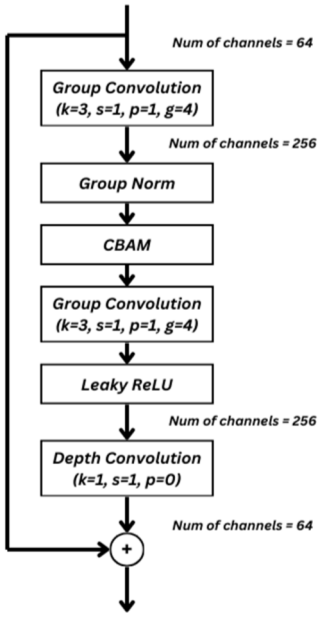

Fig. 2. Convolutional block architecture of SHA-ZA, illustrating the sequence of operations within the network.

The policy network consists of eight convolutional blocks, while the value network is composed of three similar blocks. The policy network takes as input the 3-component state representation described in the previous section. The value network, on the other hand, receives the state representation along with the output from the policy network as its input.

D. Training and Evaluation Strategies

SHA-ZA's training cycle, as illustrated in Fig. 3, diverges from those described in [3] and [23] in two significant ways. First, it omits the replay buffer typically used in off-policy methods, due to the adoption of PPO, an on-policy method. Although a combination of a state buffer with PPO was experimented with, it did not yield successful results for this application. Second, the training cycle incorporates extensive gradient accumulation, updating the agent only once after

processing an average of 196,608 positions, which significantly enhanced training stability.

The training cycle begins by using the previous policy and value networks to generate self-played games in parallel, utilizing N processes, for the final version, $N = 16$. Each process generates a specific number I of self-played games After generating these games, the agent is updated using the PPO loss and $AdamW$ [24] optimizer. After K steps, the agent's performance is evaluated.

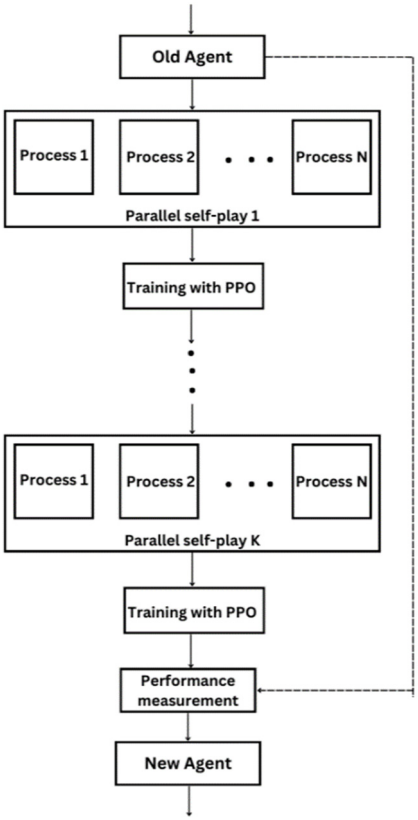

Fig. 3. Logical structure of SHA-ZA's training cycle, highlighting key processes such as parallel game generation, PPO, and performance evaluation.

In this case performance is evaluated every 25 steps ($K = 25$), therefore, total number of simulated games in each cycle is:

$$N_{games} = N \cdot I \cdot K \tag{9}$$

SHA-ZA's latest version was trained at an average speed of 37.35 games per second during parallel simulations, generating an average of 1,792.8 positions per second. The training utilized a single Nvidia RTX A5000 GPU and an AMD Ryzen 9 CPU, covering a total of 33,587,200 games. An exponential decay scheduler managed the learning rate, while other hyperparameters were manually fine-tuned, as detailed in Table 1.

For evaluation, the win probability of the new agent is measured against multiple opponents:

1) The best Agent so far.
2) A random policy.
3) A minimax with alpha-beta pruning engine (depth = 7).
4) Another pre-trained SHA-ZA agent (reference model).

Each evaluation scenario has the agent play a predetermined number of games beginning with a randomly shuffled board. The shuffle depth is a random number between 0 and 32 in all four circumstances. In situations a, b, and d, the agent plays 250 games. In scenario c, the number of games is limited to 125 due to the greater processing demands.

The Minimax engine used in scenario c has a board evaluation function which calculates the absolute difference in the number of pieces between the two opponents. As it progresses along the game tree, it seeks to capitalize on this disparity. The detailed implementation of the Minimax engine is provided in the Appendix.

The reference model in scenario d is an earlier version of SHA-ZA, with a shallower network architecture and trained for 2000 steps. This reference model serves as a more challenging opponent, providing a benchmark to measure the progress of the latest version of SHA-ZA.

Table 1. Hyperparameters change during training

Hyperparameter	Value for steps 0-3000	Value for steps 3000-5200	Value for steps 5200-6000
Batch size	256	256	256
Value Coefficient (c_1)	0.5	0.5	0.5
Entropy Coefficient (c_2)	0.09	0.045	0.09
Clip Parameter (ϵ)	0.2	0.2	0.2
Learning Rate	0.5e-4 (step 0)	0.25e-4 (step 3000)	0.125e-4 (step 6000)
Games / Step	4096	8192	4096

IV. Result and Discussion

SHA-ZA demonstrated significant progress during its training phase, reaching key performance milestones early on. Multiple checkpoints were recorded to track this advancement. By step 5,725, SHA-ZA achieved the following win rates:

1) Against random policy: **100.0**%.
2) Against Minimax (depth = 7): **96.748**%.
3) Against the reference agent: **68.85**%.

Following the training phase, SHA-ZA was exhaustively tested in standard matches against Minimax agents using search depths ranging from 6 to 12. Tests were conducted with SHA-ZA in both initiating and non-initiating roles. As shown in Table 2, SHA-ZA consistently outperformed the Minimax algorithm across all tested depths.

Table 2. Performance comparison of SHA-ZA against minimax algorithm at varying depths

Match	Minimax Depth	Initiating player	Winner	Video Link
Match 1	6	SHA-ZA	**SHA-ZA**	video 1
Match 2	6	Minimax	**SHA-ZA**	video 2
Match 3	9	SHA-ZA	**SHA-ZA**	video 3
Match 4	9	Minimax	**SHA-ZA**	video 4
Match 5	12	SHA-ZA	**SHA-ZA**	video 5
Match 6	12	Minimax	**SHA-ZA**	video 6

An intriguing observation from the fifth match against Minimax occurs between 0:22 and 0:32 in the video. During this period, SHA-ZA was significantly outnumbered in material, yet the value network's output remained between

0.2 and 0.3, indicating an optimistic evaluation. This suggests that the model focuses primarily on the game's final outcome and is capable of making strategic sacrifices when necessary. A similar pattern is observed in the sixth and most challenging match: despite being outperformed throughout the game, SHA-ZA managed to turn the tables and secure a decisive victory with a margin of 50 pieces.

A. Review of the Progress of the Training Process

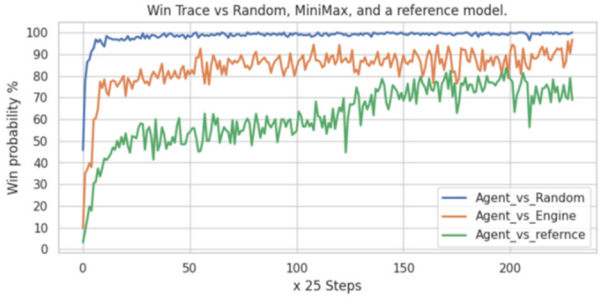

Fig. 4. Evolution of the win rate of SHA-ZA.

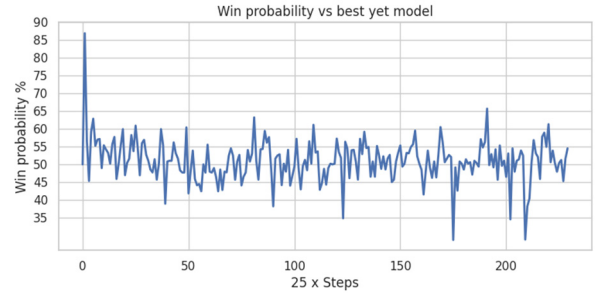

Fig. 5. Evolution of win rate of new agent against the best agent.

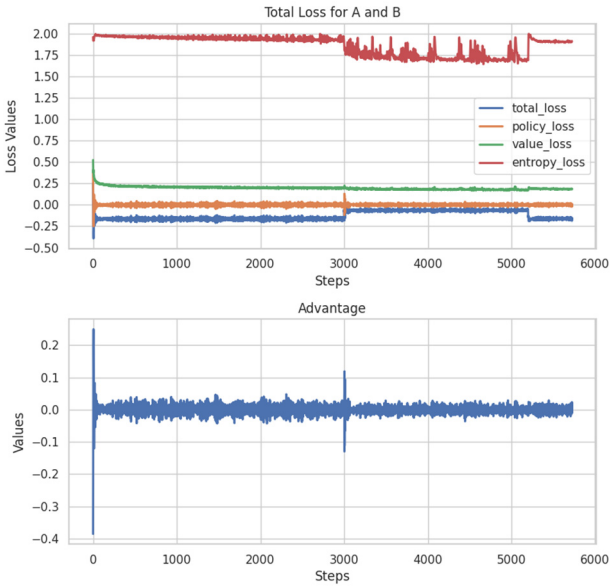

Fig. 6. The upper graph illustrates the progression of the three loss function components along with the total loss, while the lower graph depicts the advantage.

An Overall consistent progress in terms of win rate was observed.

As shown in Fig. 4 above, the reduction of the entropy coefficient c_2 from 0.09 to 0.045, which theoretically reduces exploration during steps 3000 to 5200, led to a rapid performance improvement. Reverting the entropy coefficient back to 0.09 after step 5200 appeared to enhance the overall stability of the training process.

In the above Fig. 5, as expected, during earlier steps the win rate against the best-yet model was high, indicating a quick progress of learning process, the average win rate against the best model came closer to 50% during late steps.

In Fig. 6, entropy, policy, and value losses decreased slowly and studly during steps from 0 to 3000, the advantage was heavily alternating in the beginning, which is typical, all curves were affected by the change of hyperparameters between step 3000 and 5200, and by the change at 5200 as well.

B. Review of Inference Performance and Training Duration

Unlike Monti-Carlo Tree Search (MCTS) methods [4], there is no need to run multiple simulations for selecting an action. As shown in Table 3, this approach resulted in faster training and higher inference performance compared to the Minimax search algorithm.

Table 3. average execution[1] time per single move for different agents

Agent	Avg Time per Move	Ratio to SHA-ZA's CPU inference time
SHA-ZA	0.016 sec (CPU) 0.00998 sec (GPU)	1.0 0.62375
Minimax (depth=3)	0.002465 sec	0.154
Minimax (depth=6)	0.0765 sec	4.78
Minimax (depth=9)	5.13416667 sec	320.875

[1] Measurement were performed using google Colaboratory [15], GPU is Nvidia Tesla T4, CPU is Intel(R) Xeon(R) CPU @ 2.00GHz, Minimax is an agent that uses the minimax algorithm with alpha-beta pruning, SHA-ZA's CPU move inference average time was used to calculate the ratios column.

Training was conducted in two phases, with the second phase taking longer on average per step due to a change in hyperparameters after step 3000:

- **Phase 1** (steps 0 to 3000): 156.75 hours.
- **Phase 2** (steps 3000 to 6000): 264.9 hours.

Considering the phases of the training cycle outlined in the methodology, the percentage of time allocated to each major phase is as follows:

- **Parallel self-play:** ~59.9% of total time.
- **Model parameter updates:** ~32.76% of total time.
- **Evaluation:** ~7.29% of total time.

In comparison, SHA-ZA achieved its results in just **~58.53%** of the total training time required by a similar program utilizing Monte Carlo Tree Search (MCTS), as reported in [14].

V. Conclusion

The combination of Proximal Policy Optimization (PPO) and self-play has resulted in a high level of mastery of Othello, a complex, large state space game. SHA-ZA improved consistently throughout the training phase, eventually outperforming the Minimax engine at various depths, demonstrating advanced strategic planning. This was accomplished despite the Minimax algorithm requiring significantly more time for move inference.

Compared to Monte Carlo Tree Search (MCTS), PPO eliminated the need for playouts (simulation of the game from the current position to a terminal state) and replay buffers, resulting in faster inference, lower hardware requirements, shorter training times, and a more streamlined

system design.

Furthermore, experiments with various network architectures revealed that using ConvNeXt and CBAM architectures significantly improved learning performance. Massive gradient accumulation significantly improved training stability, while hyperparameter adjustments after step 3000 resulted in additional notable performance improvements, further refining SHA-ZA's capabilities.

Appendix

I have made my code publicly available to ensure transparency and reproducibility of my results. The code can be accessed via the following link: Click Here [GitHub:proximalpolicy-optimization-for-othello-mastery]. This includes the complete implementation of PPO and the training script.

Conflict of Interest

The author declares no conflict of interest.

Author Contributions

The research, software, and writing were all conducted by Mohammed Yousif, who also approved the final version.

References

[1] British Othello Federation, "Othello history," British Othello, [Online]. Available: https://www.britishothello.org/othello-history (accessed on Jul. 7, 2024).

[2] M. van Wezel and N. J. van Eck, "Reinforcement learning and its application to Othello," EI 2005-47, Dec. 2005.

[3] D. Silver *et al*., "A general reinforcement learning algorithm that masters chess, shogi, and go through self-play," *Science*, vol. 362, no. 6419, pp. 1140-1144, Dec. 2018. doi: 10.1126/science.aar6404.

[4] C. B. Browne *et al*., "A survey of Monte Carlo tree search methods," *IEEE Trans. Comput. Intellig. AI Games*, vol. 4, no. 1, pp. 1-43, Mar. 2012, doi: 10.1109/TCIAIG.2012.2186810.

[5] H. Van Hasselt, A. Guez, and D. Silver, 'Deep reinforcement learning with double q-learning', in *Proc. the AAAI conference on Artificial Intelligence*, 2016, vol. 30.

[6] J. Schulman *et al*., "Trust Region Policy Optimization," arXiv.org, 2015.

[7] J. Schulman *et al*., "Proximal Policy Optimization Algorithms," arXiv, vol. arXiv:1707.06347, 2017.

[8] T. Kaufmann *et al.,* "A survey of reinforcement learning from human feedback," arXiv preprint arXiv:2312.14925, 2023.

[9] V. Mnih *et al*., "Asynchronous methods for deep reinforcement learning," in *Proc. The 33rd International Conference on Machine Learning*, 20-22 Jun 2016, vol. 48, pp. 1928-1937.

[10] Y. Wu *et al.,* "Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation," *Neural Information Processing Systems*, vol. 30, pp. 5279-5288, Jan. 2017.

[11] V. Mnih, "Playing Atari with Deep Reinforcement Learning," arXiv preprint arXiv:1312.5602, 2013.

[12] M. Szubert, W. Jaskowski, and K. Krawiec, "Coevolutionary Temporal Difference Learning for Othello," in *Proc. 2009 IEEE Symposium on Computational Intelligence and Games*, 2009, pp. 104-111.

[13] S. Thakoor, S. Nair, and M. Jhunjhunwala, "Learning to Play Othello Without Human Knowledge," *Stanford University, Final Project Report*, 2016.

[14] A. Norelli and A. Panconesi, "Olivaw: Mastering Othello without Human Knowledge, nor a Fortune," *IEEE Transactions on Games*, vol. 15, no. 2, pp. 285-291, 2023.

[15] T. Carneiro *et al.,* "Performance Analysis of Google Colaboratory as a Tool for Accelerating Deep Learning Applications," *IEEE Access*, vol. 6, pp. 61677-61685, 2018.

[16] S. Amin *et al*., "A Survey of Exploration Methods in Reinforcement Learning," arXiv preprint arXiv:2109.00157, 2021. [Online]. Available: https://arxiv.org/abs/2109.00157.

[17] Z. Liu *et al.,* "A ConvNet for the 2020s," in *Proc. 2022 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022, pp. 11966-11976.

[18] A. Dosovitskiy, "An Image Is Worth 16x16 Words: Transformers for Image Recognition at Scale," arXiv preprint arXiv:2010.11929, 2020. [Online]. Available: https://arxiv.org/abs/2010.11929.

[19] S. Woo *et al.,* "Cbam: Convolutional block attention module," in *Proc. the European Conference on Computer Vision (ECCV)*, 2018, pp. 3-19.

[20] Z. Baozhou *et al.,* "An Attention Module for Convolutional Neural Networks," arXiv preprint arXiv:2108.08205, 2021.

[21] R. D. Kwon *et al*., "ResNet with Integrated Convolutional Block Attention Module for Ship Classification Using Transfer Learning on Optical Satellite Imagery," arXiv preprint arXiv:2404.02135, 2024.

[22] M.-H. Guo *et al*., "Segnext: Rethinking convolutional attention design for semantic segmentation," *Advances in Neural Information Processing Systems*, vol. 35, pp. 1140-1156, 2022.

[23] J. Schrittwieser *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604-609, 2020.

[24] I. Loshchilov and F. Hutter, "Decoupled Weight Decay Regularization," arXiv preprint arXiv:1711.05101, 2017.