Extended Abstract

Motivation Self-play has many uses. By virtue of not needing human-labeled data (or human interference at all), it is able to perpetually learn, invent new strategies, and achieve world class performance on many tasks. A shining example of this is DeepMind's AlphaGo algorithm. AlphaGo was fine-tuned using self-play, and was able to beat the Go champion Lee Sedol 4-1 in a show match in 2016. While often a good option at the industrial scale, self-play can be less useful in situations where compute is constrained, since training requires many game simulations and weight updates. While research on increasing the efficiency of self-play or using warm start techniques has been done, less attention has been given to the affect of opponent pools. This paper aims to help alleviate this by presenting how changes to opponent pools affect the self-play learning process across different games in a compute restricted environment. The hope is that these findings will be able to help inform the setup of other self-play pipelines in compute scarce settings.

Method To test the effects of opponent pool size, 8 models are being trained across 2 games and 4 pool sizes. The games are a miniaturized version of Go (Go is normally played on a 19x19 board, this version is played on a 9x9 board) and a 5x5 version of Hex. Both games with be trained with opponent pool sizes of 1, 5, 10, and 20. When the pool size is 1, the model will always play against a copy of itself. When the pool size is 5, 10, or 20, the model has a 50% of playing against itself, and a 50% chance of playing a prior version of itself. The amount of prior versions that can be selected scales with pool size. When a prior version is to be selected, it is uniformly picked from the pool. Three main metrics will be collected: first player bias, policy entropy and KL divergence. In both games, the player who moves first has an advantage. If the model can only win when it is the first player (measured by first person bias), this is a sign that the policy is collapsing and unable to learn. Policy entropy gauges how varied the policy of a model is. A model with a very low policy entropy, which does the same thing in every situation, may be a victim of policy collapse. Lastly, KL convergence measures how much the model has changed between checkpoints. This could imply a model is unable to settle, or is too stationary for learning to be occurring.

Implementation All experiments use the same pipeline, which features an AlphaZero style ResNet and several workers. The workers repeatedly played games and sent the resulting trajectories back to a manager process. The manager process is responsible for using the game trajectories to update the current self-play model.

Results Results from the Go experiment showed that the Go model which used an opponent pool of size 1 suffered policy collapse, but that Go models with larger opponent pools did not. Results from the Hex experiment were inconclusive, and no obvious model failure was detected or protected against.

Discussion It is hard to make sweeping claims with data from only two tasks. That being said, the Go model was able to learn much more effectively with larger opponent pools. Areas for future research include conducting similar studies on wider ranges of tasks, as well as trying more pool sizes.

Conclusion Overall, the only claim that can be made is that opponent pools with size greater than 1 ward of policy collapse in miniaturized versions of Go. Opponent pool size did not seem to matter very much when testing Hex. A plausible theory is that it can be used as an optional protective measure.

The Affect of Opponent Pool Size on the Policy Stability of Compute Constrained Self-play Tasks

Jonathan Lutch
Department of Computer Science
Stanford University
jlutch03@stanford.edu

1 Introduction

With the introduction of more capable computing platforms, artificial intelligence models have dramatically increased in both capability and feasibility. Even now, these models are being deployed throughout society to assist in everything from programming to banking to personal health. One of the consistent, significant impediments of this technology has been data. Regardless of architecture, these models cannot learn unless they have coherent, high quality data, and they often need lots of it. However, we are reaching the point where our primary data source, the internet, has been used up, and performance plateaus in many of the fields where these models are being deployed and trained are beginning to become a concern. While several potential solutions to this problem exist, self-play has emerged as an important tool for teachings agents to perform certain tasks.

As the name suggests, self-play is a training technique in which a model learns to play a game or complete a task by cooperating or playing against a copy of itself. Imagine chess as an example. One epoch of the algorithm would entail having the model complete a game against itself. All of the board states and moves at those states would be recorded. Depending on whether the outcome was a win, loss, or draw, the likelihood of the model making certain moves would be increased or decreased. This process then repeats many, many times, until the model is able to play well.

The most prominent benefits of self-play are the automatic data generation, and the lack of need for human intervention. In other learning schemes, researchers must supply rewards or correct behavior for the model to learn from. Self-play generates labeled data while playing itself in the form of game trajectories, removing the often costly and difficult step of data collection. Additionally, the strength of the models' opponent scales with the capability of the model. This is critical, since an opponent who was too weak would always lose, leading to useless learning signal, while an opponent who was too strong would always win, leading to no learning signal. Since the self-play model plays against itself, it is always matched against a fair opponent.

One of the most famous instances of self play is the AlphaGo model that DeepMind created in 2016. The model was designed to play the game of Go, and was initially trained on the games of expert humans. During the fine-tuning process, self-play was used to achieve superhuman performance and discover new strategies. In March of 2016, AlphaGo played a show match against Lee Sedol, who is considered to be one of the best Go players of all time. AlphaGo won the match 4-1, proving the efficacy of self-play. Beyond go, self-play has also been used to play chess (Leela Chess Zero), StarCraft II (AlphaStar), and Dota 2 (OpenAI Five).

One common thread between the examples mentioned above is that they were all trained using significant compute. Since self-play often leads to monotonic improvement, especially at advanced stages of training, more compute generally corresponds to a better final model. Additionally, it allows for a decrease in the quality and quantity of human-supplied training data. For example, a chess bot that was initially trained on human games played by amateur or even novice players could achieve a very high level if it underwent enough self-play training. AlphaZero took this to the extreme, and did not start with any base model when learning chess!

While large companies are able to afford the infrastructure to use self-play in this way, many smaller labs and universities are not. As a result, many of the self-play phenomena that occur at a smaller compute scale are not as well studied. Specifically, the research into how opponent pools affect self-play when training with limited compute is not well understood. As the name suggests, opponent pools are the opponents a self-play model might face during training. If the opponent pool consists of just a copy of the model, then the pool has a size of 1. This is sometimes referred to as pure self-play. Opponent pools can be filled with older versions of the model, static models designed by experts, and models utilizing specific strategies. By testing how various opponent pool sizes affect the performance of self-play trained models in miniaturized versions of Go and Hex, this paper puts forth suggestions on how to set opponent pool parameters, and discusses how different opponent pools affect learning outcomes.

2 Related Work

Two of the most well-known examples of self-play are the AlphaGo and AlphaZero models, which were both created by Google's DeepMind. While seminal, neither of them focus on the affects of opponent pools or were conducted in compute constrained environments. Moreover, AlphaGo started with a base model, and used self-play as a sort of fine-tuning step.

In the realm of compute constrained approaches, most research has been around how to best create "warm starts" for self-play algorithms. In setups where a model must learn everything from self-play, a lot of time is spent understanding basic rules and play patterns. Warm starts are meant to help the model through these early phases of training by tweaking certain parts of the algorithm. An example of this is the RAVE paper by Hui et al. In it, RAVE and RHEA are proposed tweaks to the early parts of training to help the model learn basic rules faster.

Chen et al. have researched some of the affects of opponent pools. Using the board game Colonel Blotto as an environment, they found that training a model against a constantly updated, first-in-first-out opponent pool consisting of past versions of itself led to better outcomes (in this case, higher expected game scores) than a model trained with pure self-play. While an indication that opponent pools can be useful, it does not include comparisons between games.

3 Method

In order to test how different opponent pools affect the learning process in self-play models, several experiments were done over two games. The first game was a miniaturized version of Go. While Go is normally played on a 19x19 grid, all experiments related to the game were done on a 9x9 board to accommodate the limited compute and time available for these tests. Go is a classic 2 player board game in which the goal is to control more territory than your opponent by the end of the game. If you are able to encircle a subset of your opponent's stones with your own, the encircled pieces are removed from the board. In Go, the first player has an advantage, but this is nullified by komi, a set amount of points given to the second player to neutralize their starting disadvantage. The second game was Hex. Hex is played on a rhombus shaped board split up into hexagonal tiles. Each player gets two opposing sides. The goal of the game is to connect your two opposing sides via a continuous path. Players take turns claiming hexagons for themselves. Once a hexagon belongs to a player, they own that space for the rest of the game. In the experiments below, the game is played on a 5x5 board. Games of Hex cannot end in a draw. Notably, the player who moves first in hex has an advantage, and can always win the game.

An Nvidia 4080 GPU and a Ryzen 3950x were used to run experiments. The training pipeline consists of five main pieces. The first is a manager which initializes all the other processes and helps coordinate them. Additionally, it updates the weights of the current model using completed game data. The second is the inference server, which returns a move distribution and value for each game position it is given. The inference server runs on the GPU, and uses an AlphaZero style ResNet. Third, there is a second inference server, which is identical in function except for the fact that it runs previous versions of the model. The fourth is a set of game playing processes. These processes repeatedly generate new game data by simulating games. They use Monte Carlo Tree Search (MCTS), and get policy and value estimates from the inference server. Eight of these game simulation workers were run in parallel at once on separate CPU cores. These games get sent to the manager to do

gradient steps with. Lastly, to track the non-transitivity of the model, an evaluator process plays games between different model versions during training.

The main goal of these experiments is to test the affect of opponent pool size on model training. Opponent pools are the set of the players a model can train against when learning a game. For example, if the opponent pool consists of just a copy of the model, then the opponent pool would have a size of one. If the opponent pool contained a copy of the current model, static, human-trained model, and a random player, then the pool size would be three. For both Go and Hex, four models were trained. All four models were identical setup wise, but had differing pool sizes. The sizes tested were 1, 5, 10, and 20. Each pool always contained a copy of the current model. The rest of the opponents were older versions of the current model. Specifically, every 5,000 gradient steps, a new model snapshot was added to the pool. For all experiments, the pools always have the n-1 most recent model snapshots (excluding the current model), where n is the pool size. During training, each worker played the same opponent. Opponents were switched every 5 minutes or 200 games. In cases where the pool size was greater than 1, at each change over, there was a 50% chance that the new opponent would be the current version of the model, and a 50% chance that one of the old model versions would be used. Each old model had the same chance to be chosen.

To compare how the stability of the learning process changes across pool size, three metrics are tracked. The first is first player bias. Both Go and Hex are games in which a large advantage is gained by going first. If the one player (regardless of whether they go first or second) always wins, especially in compute constrained environments, this is a sign that the model is collapsing and failing to learn.

The second metric is policy entropy. Policy entropy was calculated by averaging the Shannon entropy of a policy over 512 fixed game positions. If a model's policy entropy is too high, this can imply that the model is mainly guessing, and that learning has not really occurred. On the other hand, a policy entropy that is too low could symbolize a collapse in the model, as it only plays the same way each game.

Lastly, the KL divergence between successive policy checkpoints is tracked. KL divergence measures the change in policy. A high KL divergence indicates that the model is still making large changes to its play, while a smaller value indicates it is making smaller adjustments.

When playing models against older versions of themselves, stochastic first moves were made to ensure the outcome was not identical every time. Specifically, a move from the eight most likely was uniformly chosen.

4 Experimental Setup

For both Go and Hex, the training pipeline was the same. Additionally, they shared the same set of hyperparameters. The main differences were game size and rules, as well as the size of the ResNet used for each game. The ResNet used for Hex had two residual blocks, 32 channels, and 6 input planes. Go is a larger, more complex game, and the ResNet it used had four residual blocks with sixty four channels and eleven input planes. The learner for each used the Adam optimizer. Learning hyperparameters can be found in Table 1. Policy and value losses were equally weighted. For both games, 50 MCTS simulations were done per move. The first ten moves were selected by taking the move with the highest model value to times visited ratio. The rest were selected greedily.

Go experiments took 36 hours and took roughly a 1,000,000 gradient steps each. The experiment times for Hex were much shorter. They ran for only 30 minutes and took roughly 170,000 gradient steps. This was mainly a time based consideration, as too many repeated, long experiments would have made the project intractable in the span of a quarter.

5 Results

For two games, Go and Hex, self-play was used to train four different models. Each model used a different sized opponent pool, either 1, 5, 10, or 20. Results were split. Some patterns that showed up in Go also were present in Hex. Others seem independent. Notably, the Hex and Go metrics rarely ever trend in opposite directions.

Table 1: Optimization hyperparameters.

Hyperparameter	Value
Optimizer	Adam
Learning rate (α)	1×10^{-3}
β_1, β_2	0.9, 0.999
ϵ	1×10^{-8}
Weight decay (L_2)	1×10^{-4}
Gradient clipping (max norm)	1.0
Batch size	256

Table 2: Steady-state metrics by game and pool size (mean over the final 25% of training).

Game	Pool	First-player bias	Policy entropy	KL (prev. ckpt)
Go	1	-0.682	1.878	0.212
Go	5	0.247	1.248	0.253
Go	10	0.259	1.231	0.271
Go	20	-0.300	1.263	0.289
Hex	1	0.240	1.443	0.102
Hex	5	0.141	1.385	0.135
Hex	10	0.218	1.328	0.126
Hex	20	0.142	1.304	0.131

The category with the most similarity is first player bias. First player bias decreases in both games when pool sizes are switched from 1 to 5. The affect is more pronounced in Go, but still significant in Hex. Moreover, they remain relatively constant for pool sizes 5, 10, and 20. Hex sees an increase in first player bias when jumping from pool size 5 to 10, but it is less pronounced than the fall from pool size 1 to 5.

Policy entropy is initially high for Go when the opponent pool size is 1, but falls by roughly a third when the pool size is increased to 5, 10, or 20. Policy entropy for Hex monotonically decreases, though slowly. The KL divergence of both games is roughly stationary. Overall, Go's metrics increase and decline more sharply than Hex's.

5.1 Quantitative Evaluation

The first thing to note is the decrease in first player bias given an increase in pool size. Across Go and Hex, there is a significant drop when increasing the pool size from 1, 63% and 41% respectively. Notably, while further increasing the pool size does alter the numbers a little, the affects are less pronounced.

For Go, policy entropy decreased by 24% when going from a pool size of 1 to a pool size of 5. However, it hangs at around the same spot for pool sizes of 5, 10, and 20. The policy entropy for Hex never changed dramatically, though it did show a slight monotonic decrease with pool size.

The KL divergence increased by roughly 20% for Go and 32% for Hex when going from a pool size of 1 to 5. Beyond that, all other changes are within 10% of each other.

5.2 Qualitative Analysis

When looking at the Go data when the opponent poll size is 1, it is clear that the model has undergone policy collapse. First, the magnitude of the first player bias is too large. Unlike in Hex, whether you go first or second in Go does not significantly affect your chances of winning due to komi. Thus, a well trained model should have a first player bias which has an absolute value near 0. Secondly, the policy entropy is significantly higher when the opponent pool is of size 1, as opposed to 5, 10, or 20. This is likely because the model is not able to find successful patterns of play, and thus cannot focus in on a set of moves and tries many. Moreover, the model likely has a low KL divergence since it does not have much signal to follow and thus does not change much from step to step.

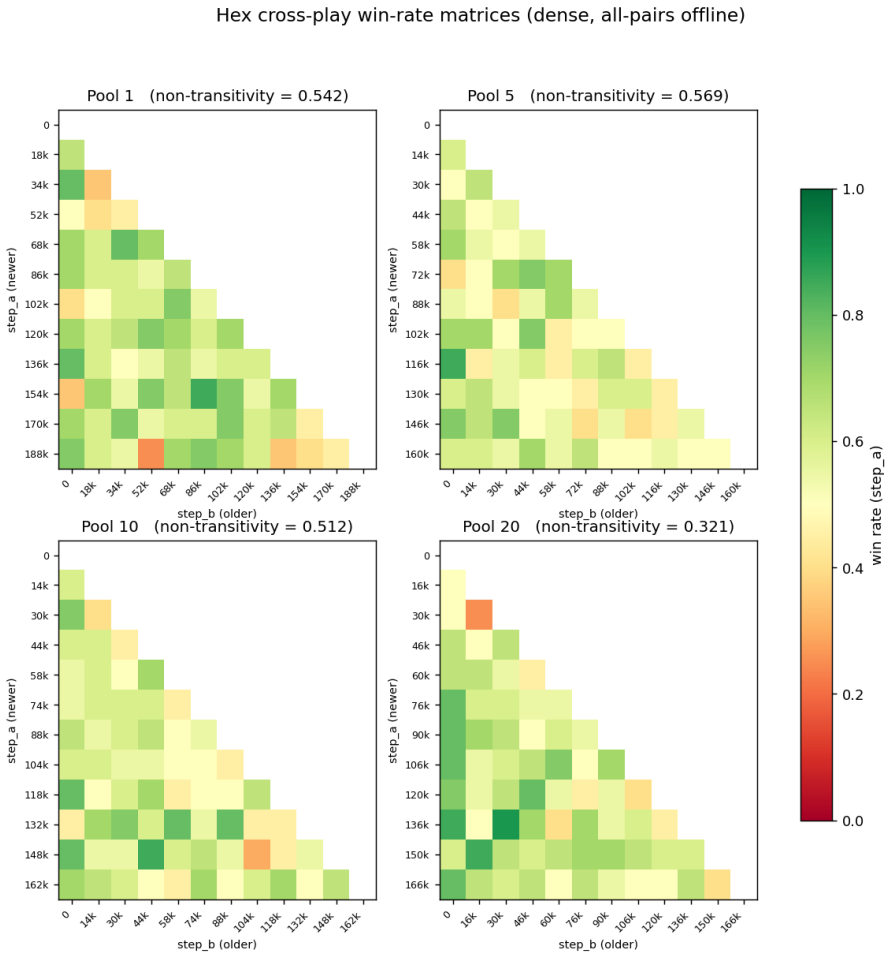

Figure 1: Hex cross-play win-rate matrices. Pool sizes from top left to bottom right: 1, 5, 10, 20.

While opponent pool sizes of 5, 10, and 20 all yield a Go model that appears to be learning healthily, it does not seem like there are any major differences between them.

The picture is slightly more complicated for Hex. Hex does have a first mover advantage, and the player who moves first can always win. The decrease in first player win bias could be mitigating policy collapse, but it could also be due an increased pool size making it harder for the model to learn. Overall, it is difficult to say anything about the affect of opponent pool sizes.

With limited data, it is hard to suggest specific opponent pool size ranges. However, to avoid cases of policy collapse like the one seen in the version of Go tested here, having an opponent pool size of at least 5 is recommended.

6 Limitations and Future work

This project had several notable limitations. Primarily, experiments were only done over two games. In future experiments, greater game diversity would allow for wider claims to be made about pool size. Additionally, each Hex model was only trained for about 170,000 gradient steps as opposed to the roughly 1,000,000 used for Go. An equivalent amount of steps, or at least time spent training, would make for a better comparison. The same is true of ResNet size.

There are several interesting potential offshoots of this work. Playing the models presented in this paper against external baselines would give more of a grounding to some of the metrics. For example, comparing the strength of two models with different non-transitivity scores with a baseline could

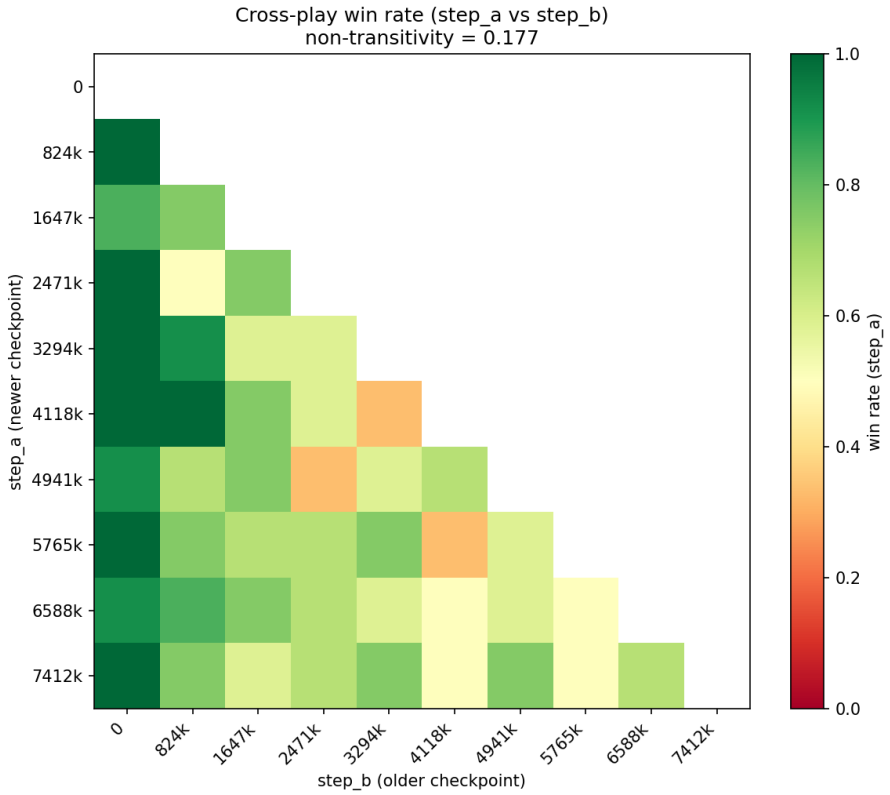

Figure 2: Go win-rate matrix for a pool of size 5.

let you quantify how non-transitivity harms performance. Additionally, running experiments with dynamically changing pool sizes may reveal novel self-play learning behavior.

7 Conclusion

Overall, this paper showed that self-play models can be vulnerable to policy collapse, and that a potential protection against this is to increase opponent pool size. While not enough scenarios were tested, choosing an opponent pool size of at least 5 is recommended in more compute constrained scenarios.

8 Team Contributions

- **Jonathan Lutch:** As I was the only member of the group, I did all of the work.

9 AI

Claude Code was used to help set up my coding pipeline, and was used to turn my data into graphs. It was also used for debugging.

Changes from Proposal The proposal was focused on how opponent pools effected how self-play learned the game chess. I chose to find a less complex game to learn as I was worried that I would not have enough time to run any meaningful experiments. Additionally, I chose to add in some shorter experiments with the game Hex. This change was due to some feedback I got during the poster session, which encouraged adding a second game so I could get more information about how pool size effected learning.

References

Luying Chen, Jie Hou, and Xianlin Zeng. 2025. Multi-resource Attack-Defense Strategy Optimization in the Blotto Game Based on Pool-PPO. In *Chinese Conference on Swarm Intelligence and Cooperative Control*. Springer, 399–411.

David Silver and Demis Hassabis. 2017. *AlphaGo Zero: Starting from Scratch*. Google DeepMind. `https://deepmind.google/blog/alphago-zero-starting-from-scratch`

David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. 2018. A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. *Science* 362, 6419 (2018), 1140–1144.

Hui Wang, Mike Preuss, and Aske Plaat. 2020. Warm-start AlphaZero self-play search enhancements. In *International Conference on Parallel Problem Solving from Nature*. Springer, 528–542.