ECAI 2023
K. Gal et al. (Eds.)

doi:10.3233/FAIA230279

Expediting Self-Play Learning in AlphaZero-Style Game-Playing Agents

Yngvi Björnsson, Róbert Leó Þormar Jónsson and Sigurjón Ingi Jónsson

Department of Computer Science, Reykjavik University

Abstract. One of the main appeals of AlphaZero-style game-playing agents, which combine deep learning and Monte Carlo Tree Search, is that they can be trained autonomously without external expert-level domain knowledge. However, training such agents is generally computationally expensive, with the most computationally time-consuming step being generating training data via self-play. Here we propose an improved strategy for generating self-play training data, resulting in higher-quality samples, especially in earlier training phases. The new strategy initially emphasizes the latter game phases and gradually extends those phases to entire games as the training progresses. In our test domains, the games Connect4 and Breakthrough, we show that game-playing agents using the improved training approach learn significantly faster than counterpart agents using a standard approach. Furthermore, we empirically show that the proposed strategy is (in our test domains) superior to several recently proposed strategies for expediting self-play learning in game playing.

1 Introduction

Over the past years, in part prompted by the success of AlphaZero [12, 13, 11], deep reinforcement learning has become mainstream for training game-playing agents to play abstract board games at an expert level [12, 9, 7, 15]. One of the main appeals of that approach is that it requires no pre-coded expert domain knowledge nor human involvement during the learning process. However, this approach comes at a cost, particularly access to massive computing resources. For example, AlphaZero played 40 million chess games with the help of 5,000 special-purpose computing units (TPUs) to reach state-of-the-art performance in the game of chess [12].

The most computationally time-consuming component of the learning process is generating the training data, i.e., the games, via self-play. Thus, a natural question is whether one can achieve similar expertise by using significantly less computing resources. Several avenues of research have addressed this, most notably model improvements, i.e., altering the neural-network architecture for faster convergence and better performance [4, 17, 2] and more efficient generation and use of the training data [14, 8, 17]. Here, we are only concerned with the latter and investigate it further using the games Connect4 and Breakthrough as our test domains.

The paper's main contribution is an improved strategy for generating training data via self-play, resulting in higher-quality training samples, especially in earlier training phases. The new strategy, which we call *Late-To-Early Simulation Focus (LATE)*, initially emphasizes training experiences gathered from the late stages (i.e., endgame) of individual training games; however, as training progresses, it expands its focus to consider entire games. In our test domains, game-playing agents using the new approach for training reach superior playing strength than counterpart agents using a standard training approach, requiring significantly less training time. Furthermore, we empirically show and quantify the quality improvement in the training data when using our new approach and contrast its effectiveness to that of several other recently published approaches [8, 17].

The paper's organization is as follows. The next section provides the necessary background material. The two following sections describe and empirically evaluate the enhancements for expediting the game agent's learning, respectively. Finally, we conclude and discuss future work.

2 Background

This section provides the necessary preliminaries. We start by giving an overview of the workings of AlphaZero-style game-playing agents and their training, followed by a summary of recent related approaches at having such agents generating self-play training data more effectively.

2.1 AlphaZero-Style Agents

AlphaZero-style game-playing agents use a (deep) neural-network-guided Monte Carlo Tree Search (MCTS) [6, 3] to decide which moves to play. A neural network, denoted by $f_{\boldsymbol{\theta}}$, takes a game state s as input and generates a policy prior $\boldsymbol{p}$ and a value estimate v, i.e.:

$$(\boldsymbol{p_\theta}, v_\theta) = f_{\boldsymbol{\theta}}(s).$$

The policy prior $\boldsymbol{p_\theta}$ is a distribution of real numbers over all available moves in state s, evaluating their relative promise, whereas the real number v_θ estimates the merit of state s from the player's to move perspective, for example, as the expected game outcome. The neural network outputs are used for guiding an MCTS, modifying it in two critical ways.

First, the MCTS *selection phase*, which applies while choosing actions while traversing the tree (built and kept in memory), is extended to use the policy priors from the neural network for early guidance, that is, action selection in the tree is performed by choosing in each state s the action a that maximizes:

$$\mathrm{PUCT}(s,a,s') = V(s') + c_{\mathrm{PUCT}} \cdot p_{\boldsymbol{\theta}}(s,a)\frac{\sqrt{N(s)}}{N(s')+1},$$

* Authors are listed in alphabetical order. All authors contributed equally.

where s' is the successor state resulting from taking action a in state s, and, as in regular MCTS, $V : S \rightarrow \mathbb{R}$ and $N : S \rightarrow \mathbb{N}$ hold the average backup-value and the number of visits to state s, respectively (where S represents the state space).

Second, the *playout* and *expansion* phases of a regular MCTS are combined into a single *evaluation-and-expansion* phase. This phase is entered after the selection phase, and a new node s is added to the tree using the neural network to label the node's $V(s)$ and $p_{\boldsymbol{\theta}}(s, a)$ values for non-terminal nodes (and the game's outcome for terminal states). Unlike in traditional MCTS, for non-terminal states, no playout simulations are performed.

The *back-propagation* phase functions the same as in traditional MCTS, updating the value estimate of all ancestors of s in the tree. Also, move decisions are made in a traditional manner, with the *selection*, *evaluation-and-expansion*, and *back-propagation* phases repeatably run successively until allotted deliberation resources are up (typically a limit on time or number of simulations), when the most frequently visited action at the root is played.

2.2 AlphaZero-Style Training

The neural network $f_{\boldsymbol{\theta}}$ is trained using data gathered during agents' self-play. Initially, an agent plays a fixed number of games against itself using an untrained network. For each game, all occurring states are recorded and labeled with the game outcome (from the player's perspective to move), z, and the relative frequency at which the MCTS explored the actions available in the state, π (normalized to be a probability distribution). This results in labeled samples for training the next-generation network for the agents. This process continues in a bootstrapping fashion until playing performance converges.

The network training uses the following loss function:

$$L(\boldsymbol{\theta}, s, z, \boldsymbol{\pi}) = (v_{\boldsymbol{\theta}}(s) - z)^2 - \boldsymbol{\pi}^\top \log(\boldsymbol{p}_{\boldsymbol{\theta}}(s)) + c\|\boldsymbol{\theta}\|^2$$

with z and $\boldsymbol{\pi}$ the target outcome and policy, respectively. The training process keeps relevant training samples in a so-called replay buffer during training, from which it samples mini-batches for updating the network. In the traditional approach, this is a fixed-sized buffer storing training samples from one or more recent agent generations.

Extra precaution is taken during training to encourage exploration. Firstly, Dirichlet noise is added to the tree's root node r during self-play, modifying the policy prior as follows:

$$\boldsymbol{p}'(r) = \lambda \boldsymbol{p}(r) + (1 - \lambda)\eta$$

where, η is sampled from a Dirichlet distribution with parameter α. Secondly, instead of always greedily playing the most promising move, a move is chosen probabilistically using a softmax policy for the first few moves of each game.

In the default training approach, the neural network of the self-play agents is updated periodically with a better-tuned version; however, the agents' search behavior stays the same otherwise. In particular, a fixed number of MCTS simulations is performed per action decision. It can take many generations for the training process to converge. In early generations, valuable signals from the training games may be sparse as the agents act more or less randomly. However, gradually, the agents start to improve, and the games become more representative of expert play. The premise behind our method for expediting self-play, as well as the others we review in the following subsection, is avoiding spending much computing resources on generating and training with samples having low-quality target signals.

2.3 Related Work

OLIVAW [8], an AlphaZero-style agent for playing the game of Othello, introduces several techniques for reducing self-play training time. Two of those are related to our work. First, it uses a *dynamic training window (DTW)* that gradually increases the number of past-generation training samples stored in the replay buffer as training progresses. Second, it incrementally adjusts the number of MCTS simulations per action decision as training progresses, increasing them stepwise (100 for the first four generations, 200 for the next seven, and from thereon, 400). We use a slight variation of this enhancement that more gradually increases the simulation count, which we call *gradual simulation increase (GIS)*. The rationality behind both those enhancements is to move quickly away from the earlier generations.

KataGo [17] introduces several architectural and self-play-related enhancements to AlphaZero-style learning. The improvements are both domain-specific (to the game of Go) and domain-independent. The most closely related domain-independent enhancement is called *randomized playout cap (RPC)*, which limits the number of simulations per move to n, where n is much smaller than the regular simulation number N and applies to most move decisions throughout a game. During training, one ignores the training samples resulting from the fast searches, thus getting fewer training samples per game. This strategy leads to more varied games being played, albeit resulting in a slightly smaller but more diverse training set — a tradeoff the paper claims is still beneficial for training. As the training advances, the number of simulations in both full and partial searches incrementally increases.

3 Method

Traditionally, after a self-play game, all training samples resulting from the game are labeled with the game outcome (the target for the value function) and the relative visit counts to the children of the MCTS root node (the target for the policy function).

In early training generations, the gameplay is poor, with both sides making many mistakes. This results in the signal the target labels provide being noisy and only loosely related to the actual merits of moves and states. Nonetheless, the training samples provide more than enough helpful feedback to improve the agents' playing strength. The gameplay thus generally improves quickly between early training generations. The previous strategies for expediting self-play training, which we reviewed earlier, capitalize on this. However, another factor is also at play, hitherto unexploited – the labeling quality also differs within a game.

The think-ahead process of MCTS can reliably look some moves ahead. So, when close to terminal states, it quickly zooms in on the correct moves, resulting in the value and policy target labels both being reasonably accurate. On average, at least to start with, the labels tend to be significantly more accurate in later game phases than earlier ones (we confirm this assertion in our test domain in the result section). This observation is a central premise behind the LATE enhancement we propose.

3.1 Late-To-Early Simulation Focus

In the *Late-To-Early Simulation Focus (LATE)* enhancement, we vary the number of MCTS simulations performed during self-play based on *the current training generation and the number of moves played within a game*, according to a function $w(g, m) \in [0, 1]$, where g

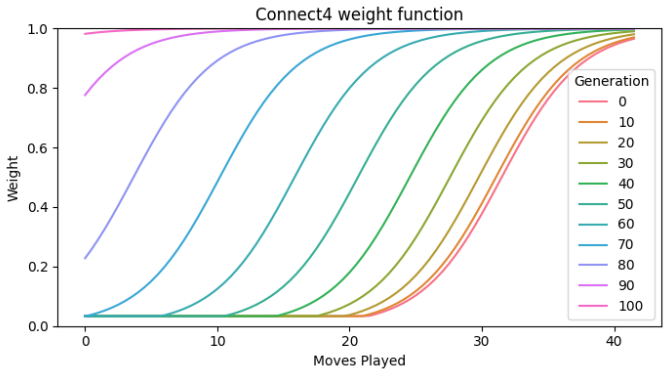

Figure 1: Weight function for Connect4

is the training generation number and m is the game's current move number. Assuming minimum and maximum numbers of simulations allowed being n and N, respectively, the number of simulations the agent performs in a given state when deciding on an action is:

$$w(g,m) \cdot N$$

where

$$w(g,m) = \max\left(\frac{n}{N}, s(g,m)\right)$$

and where the $s(g,m)$ function controls the simulation effort. A different function could be used, but here we opt for a shifted and scaled sigmoid function:

$$s(g,m) = \frac{1}{1+\exp(\rho(g) - \frac{m}{h})}$$

where h is a scaling hyper-parameter. The shift $\rho(g)$ transitions from the initial value ρ_0 to ρ_1 according to

$$\rho(g) = (\rho_1 - \rho_0)\left(\frac{g}{u}\right)^{\omega} + \rho_0$$

where u is the generation at which we want the weight to be a constant 1, and ω controls the acceleration.

Initially, the weight will be close to zero but then drifts towards one as the game progresses. The weighing adjustment occurs late in early training generations but earlier in later generations. Figure 1 depicts an example thereof, where the hyper-parameters have been adjusted for learning in Connect4.

The main impetus behind the $s(g,m)$ function is to get an easily parameterized smooth transition (from low to high) as both games and training progress. Other functions with similar properties might suffice too.

3.2 Training and Loss Weighting

During training, we use all self-play states but weigh the loss of samples with w. This approach mitigates the data quality problem by focusing the training on the late game, where higher-quality samples are found. The model can then gradually increase its strength towards the early game along with the weight function, utilizing its improved performance in later moves to enhance its moves in earlier stages.

3.3 Neural Networks

As in AlphaZero the neural networks feature a convolutional body, accompanied by a value head and a policy head. The convolutional body consists of ResNet [5] blocks that function as a feature extractor. The value head processes the body's output to generate an evaluation estimate, while the policy head produces a tensor representing a probability distribution over possible moves.

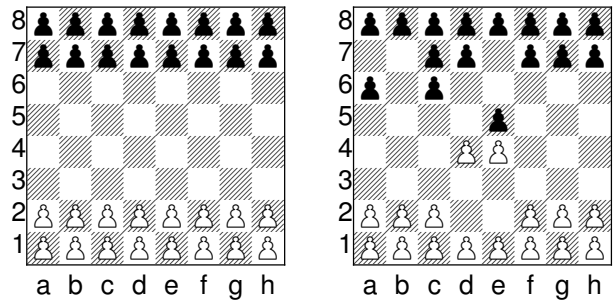

Figure 2: Breakthrough. The initial position to the left, with White to move. In the example position to the right, the White pawn on d4 can move to either c5, d5, or e5 (with capture), whereas the White pawn on e4 can move to either d5 or f5.

4 Results

We empirically evaluate the LATE enhancement by monitoring the quality of the learning process, the computing effort required, and the playing strength of the resulting agents. However, before presenting the empirical results, we describe the test domains, the experimental methodology, and the experimental setup.

4.1 Games

We use the two-player (perfect information, deterministic, turn-taking) abstract strategy board games *Connect4* and *Breakthrough* as our test domains.

Connect4 is a well-known game played on a vertically suspended 7x6 grid. The players take turns dropping colored disks, one per turn, into any remaining non-full grid column (once dropped, a disk falls to the lowest unoccupied grid cell in that column). The first player to form a horizontal, vertical, or diagonal line of length four with their color disks wins. If the grid fills without either player forming a line of four, the game outcome is a draw. The main reason for us including this game in our testbed is that it is strongly solved — we know the actual outcome of all possible game states. Having this information as ground truth allows us to systematically compute and compare the quality of the training samples generated by different self-play strategies.

The game Breakthrough is played on a chess-like board. The left diagram in Figure 2 shows the board's initial setup, with the white pieces along the bottom two rows and the Black pieces along the top two rows. White goes first and then the players take alternating turns, with each player moving one of their pieces per turn. A piece moves one square straight or diagonally forward (relative to the player), with the following restrictions: the target square must either be empty or, in the case of diagonals moves, can be occupied by an opponent's piece, which is then captured and removed from the board. The right diagram in Figure 2 shows examples of legal piece moves. The objective of the game is to be the first player to maneuver one of their pieces to the opponent's back-rank, and the first player to do so wins. Another way to win is to capture all the opponent's pieces. It follows from the game rules that one of the players always wins (no

draws). The strategic complexity of the game is rich enough to require non-trivial strategies to play well while simultaneously being small enough to allow us to train an expert-level agent in a reasonable time frame using moderate computing resources.

4.2 Experimental Methodology

Connect4 is a strongly solved game, which allows us to evaluate the quality of the generated training data by comparing the labels of the training data to ground truth. We used an existing solver [10] to provide the ground truth. We use $v^*(s) \in \{-1, 0, 1\}$ to represent the ground-truth game outcome of a state s assuming optimal play from both sides, where -1, 0, and 1 represent a loss, draw, and a win, respectively. Then, given states and comparison evaluations $(s_1, v_1), \ldots, (s_n, v_n)$, we use the *root mean squared error (RMSE)* between the ground-truth evaluations $v^*(s_i)$ and comparison evaluations v_i. The ground-truth policy, $\boldsymbol{\pi}^*$, is a probability distribution over all possible actions in a given state, where actions that lead to the best available game outcome from that state all get a value $1/b$ (assuming b such actions), while all other actions get the value 0. [1] We use the *cross-entropy* as an error measure between $\boldsymbol{\pi}^*$ and the comparison policy $\boldsymbol{\pi}$. Using the above error measures, we evaluate the quality of both the neural network's and MCTS outputs.

Furthermore, we created three datasets to ease the evaluation and reproducibility of our empirical evaluation. First, a dataset D_{c4} consisting of 10,000 Connect4 positions representative of expert-level gameplay. We generated the dataset by playing multiple games using an agent that chooses its actions with a softmax policy over all available moves as evaluated by the solver — the agent is thus more likely to pick a good move, albeit not always the best one. We then randomly pick one position from each game to store in the dataset, ensuring no duplications. We use this dataset to evaluate the quality of our model/agent throughout training (see subsection 4.4). Second, we created two datasets of opening positions four plies into the game, O_{c4} and O_{bt}, the former with 25 distinct positions and the latter with 50. We use them as starting points when matching agents against each other (see subsection 4.6). Both datasets were created using standard MCTS.

We also created an optimal agent for Connect4 using a solver. It plays perfectly, and when choosing between actions with the same game-theoretical value, it chooses the one that either wins the fastest or, when not winning, the one that puts up the most prolonged resistance. We use it as a baseline when evaluating the playing strengths of other Connect4 agents.

Throughout our experiments, we employ the cumulative number of executed simulations as a proxy to approximate the computational resources expended in training an agent. This approach is justified as the vast majority of computational resources dedicated to training an agent are consumed by the simulations conducted during self-play. Given that the number of simulations executed per generation varies across the different agents we train, the cumulative number of simulations serves as the most suitable and consistent measure.

4.3 Experimental Setup

The agents are written in C++, and the training is done in Python using PyTorch version 1.13.1. The source code is publicly available [1]. For the most part, we ran the experiments on a 16-core (32-thread) CPU (AMD Rysen 9 3950X) with an NVIDIA Geforce RTX 2080 Ti GPU card, and 32GB of memory. On that hardware, each training episode using the complete computing resources lasted, depending on the method, from a few hours to a full day for Connect4, and from a few days to over two weeks for Breakthrough. [2]

4.3.1 Self-Play Hyperparameters

All our agents apart from RPC (see below) play 5000 games per generation. For the default and DTW agents, we use 600 and 800 simulations per state in Connect4 and Breakthrough, respectively. In the GIS implementation, the number of simulations starts at 60 in Connect4 and increases linearly to 600 over the course of 100 generations. In Breakthrough, the number of simulations begins at 80 and follows a similar linear increase to 800 within the same timeframe.

When applying LATE to Connect4, we use $n = 20$, $h = 3$, $\rho_1 = -4$, $\rho_0 = 10.5$, $u = 100$, and $\omega = 2$. The weight function with these hyperparameters is depicted in Figure 1. For Breakthrough, we use $n = 50$, $h = 8.4$, $\rho_1 = 4$, $\rho_0 = 8.2$, $u = 100$, and $\omega = 1$. With these hyperparameters, the weight function for Breakthrough is analogous to the one used for Connect4, but is adjusted for the standard number of moves in Breakthrough. Additionally, the weight function transitions linearly towards the early game with $\omega = 1$, as a quadratic transition in Breakthrough is initially too slow due to the increased number of moves.

For RPC, the full search probability (p) is set to 0.25 in both games, as in KataGo. In Connect4, the parameters n and N (number of simulations in a partial and full search, respectively) start at 75 and 450, respectively, and increase linearly to 150 and 750 over 100 generations. In Breakthrough, the values of n and N increase from 100 and 600 to 200 and 1,000 during the same period. These values are similar to the ones used in KataGo. Now, because only states that get a full search are used for training, the average number of training states we obtain from a game is a factor of $1/p$ smaller. Thus, to generate an equivalent amount of training data as in the other runs, we multiply the number of games per generation by a factor of $1/p$, resulting in 20,000 games per generation for RPC experiments.

When incorporating Dirichlet noise at the root, we use $\alpha = 0.7$ for Connect4 and $\alpha = 0.3$ for Breakthrough, consistent with the AlphaZero approach, which inversely scales this parameter with the average branching factor. A softmax policy, as described in subsection 2.2, is used for the first four plies in Connect4, and for the first six plies in Breakthrough. All our agents use $c_{\mathrm{PUCT}} = 4$, determined through informal experiments.

4.3.2 Training Parameters

When training, we sample games uniformly from the last 20 generations in Connect4 and 25 generations in Breakthrough. The learning rates are set to a constant 10^{-3} for Connect4 and 10^{-4} for Breakthrough, and the weight decay parameter is $c = 10^{-4}$ for both games. These parameters were all determined through informal empirical experiments, except for the weight decay parameter, which is the same as in AlphaZero [12].

For DTW in Connect4, the training window increases from 1 to 20 over 60 generations. In Breakthrough, it increases from 1 to 40 over 80 generations.

[1] This is not the only way to form an optimal ground-truth policy, but we chose to do it this way because it is the policy that MCTS would converge to with an infinite number of simulations.

[2] Thanks to Lambda Labs (https://lambdalabs.com) for donating computing time for some of our early experiments.

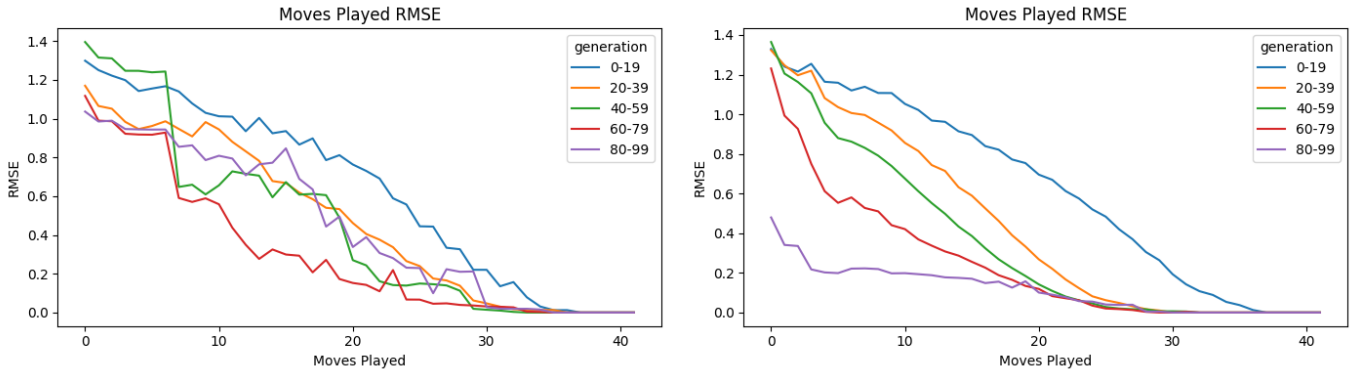

Figure 3: Training data quality for default agent (left) and LATE agent (right). Each curve is averaged over 20 generations. The high error in the first few moves is exacerbated by the measures we use during training to encourage exploration. Overall, the error reduces as both the training and, even more profoundly, the game progresses.

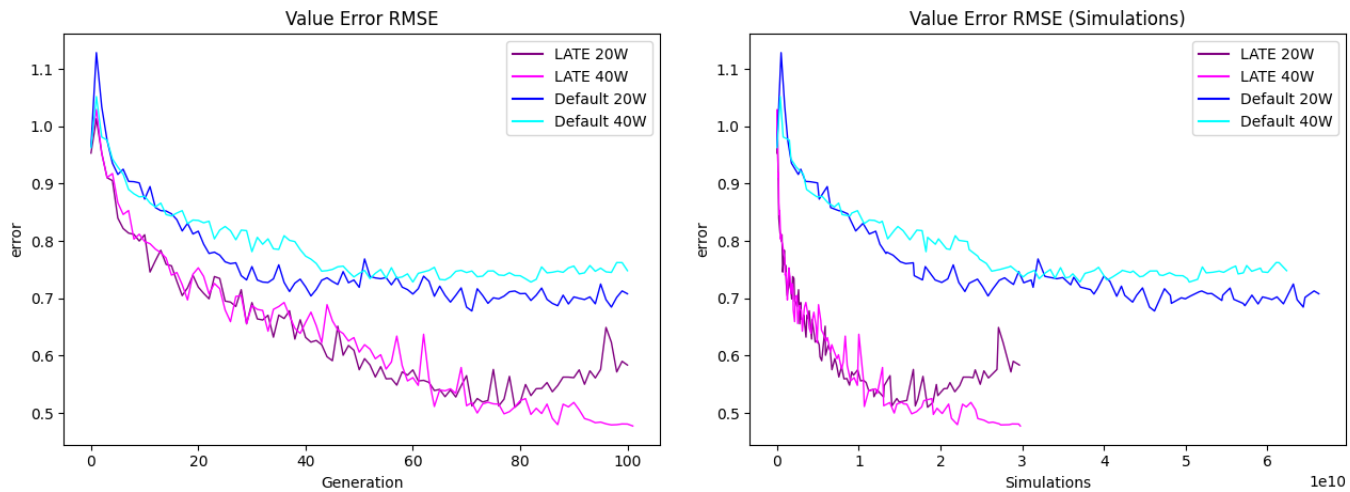

Figure 4: Comparing model quality for different training window sizes for LATE and default agent. The error is shown both as a function of generations (left) and simulations (right).

4.3.3 *Neural networks*

In Connect4, the convolutional body is composed of six 64-channel ResNet blocks, and the policy is represented by a single 7-dimensional vector that corresponds to the board columns.

In Breakthrough, the body is made up of seven 128-channel ResNet blocks, and the policy is represented by a $3 \times 8 \times 8$ tensor. The three channels indicates the possible ways to move a piece: channel 0 for left, channel 1 for forward, and channel 2 for right.

4.4 *Quality Assessment Using Ground-Truth*

Our ground-truth for Connect4 allows us to monitor the quality of the training data and our agents during training, thus gaining added insights. Here, we look at the value estimate quality (for brevity, we omit discussing the policy quality, but our playing-strength experiments indirectly demonstrate improved policy quality).

4.4.1 *Training Data*

Figure 3 shows the errors of the value labels of the training samples during Connect4 training for both the default and our enhanced agent. The different curves show the error averaged over different training generations. We see that, overall, the error reduces with further training (one anomaly where the default agent shows a sign of overfitting towards the end of training). Even more profound is the effect of the error reducing towards later game stages. By contrasting the left and right graphs, we see that the quality of the training samples generated by our enhanced agent seems at least as good as for the default agent despite running much fewer simulations in early game stages. Moreover, because of the sample weighing, the training of the enhanced agents places less emphasis on the (more erroneous) samples in the early game stages. This results in more effective training with respect to playing strength, as we see later.

4.4.2 *Model*

How do the errors in the training data affect the quality of the neural network and the agents' decisions? Figure 4 depicts the error of the network's value output (v) of positions from various game states, i.e., using the D_{c4} test dataset, plotted as a function of training generation (left) and the number of simulations (right). Each agent has two versions, one using the default training-window size of sampling from the past 20 generations and the other using twice as large a

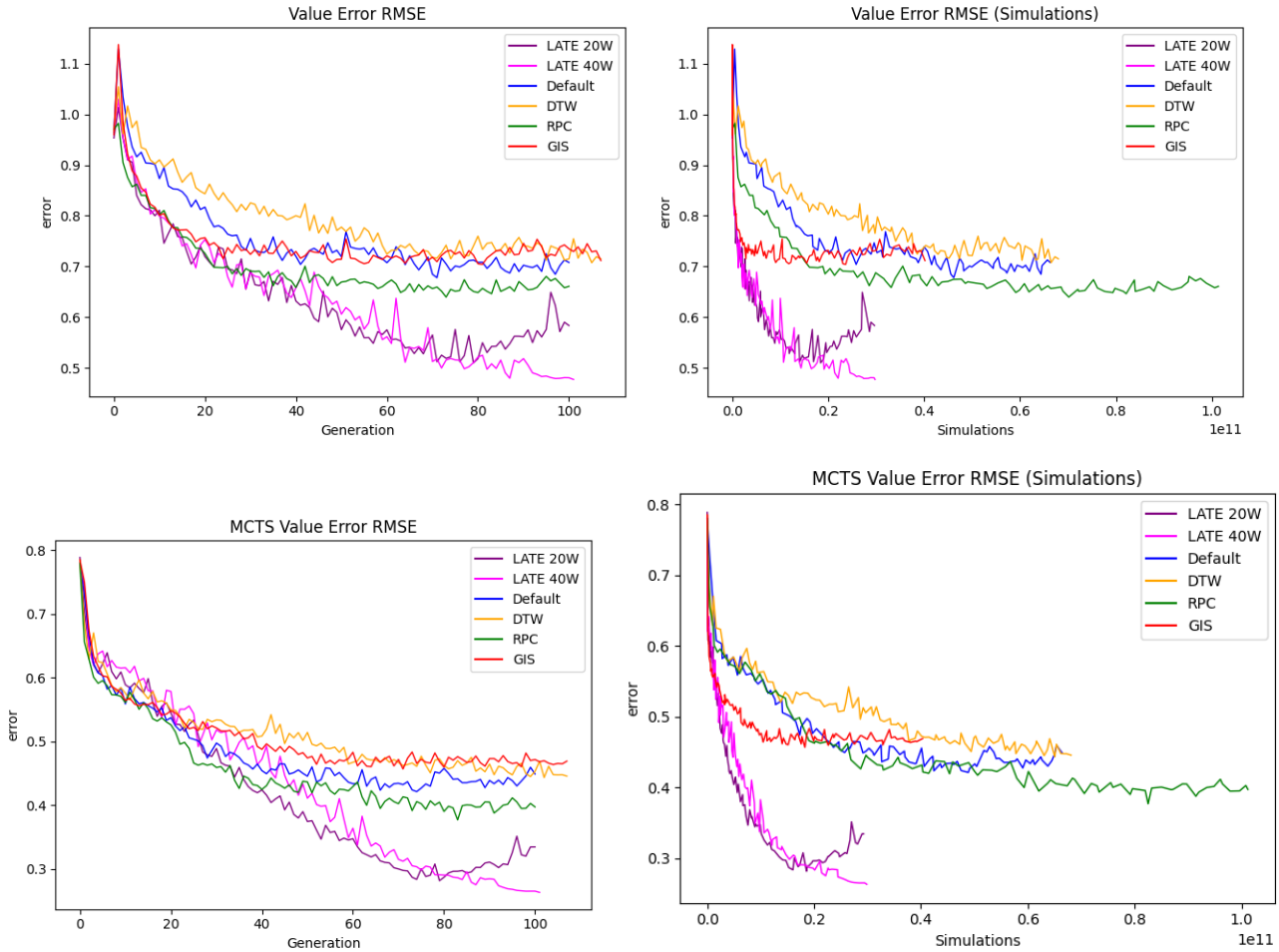

Figure 5: The upper plots show the model error of v_θ as a function of the number of training generations and total number of self-play simulations, respectively. The lower plots show similar plots for the agents MCTS root value estimate after 600 simulations.

window. Firstly, the network's value output quality is much better in the LATE agent than the default one, both given the same number of training samples (left) and the same self-play training-data generation effort (right). Secondly, we see a sign that the LATE agent using the smaller training window is starting to overfit its evaluations towards playing against itself, as its performance against the fully-trained baseline agents starts to degrade. This behavior is well-known in self-play learning settings and becomes more visible the better an agent gets. It can be rectified by including training data from earlier agent generations. Third, we see that the LATE agent benefits from a more extensive training window as its performance continues improving. Typically it is detrimental to use training data from much earlier generations, as it has lower overall quality; however, because of the importance of weighing the LATE agent uses in training, it seems it can still benefit from better quality endgame samples from early generations while not being distracted by the more erroneous early-game samples (unlike the default agent).

4.5 Comparison to Other Methods

Finally, we assess the quality of our approach in contrast to other similar approaches proposed in the literature (which we summarised in the related work section). Figure 5 depicts the error of positions from various game states, i.e., using the D_{c4} test dataset, plotted as a function of training generation (left) and the number of simulations (right). The upper and lower graphs show, respectively, the errors in the networks' value output ($v_{\boldsymbol{\theta}}(s)$), and the value estimate of the MCTS root after 600 simulations. The LATE method is in a class of its own regarding the value accuracy of both the neural-network model and the MCTS — as we will see later, this translates directly into improved playing strength. However, for fairness, the result should not be interpreted literally as all methods mostly use the default hyper-parameters and could potentially improve by careful tuning. Nonetheless, the result shows the LATE method's promise.

4.6 Playing Strength

Ultimately, we would like to compare the computing effort required for the different enhancements to generate agents of equal playing strength. We do that here for both Connect4 and Breakthrough.

4.6.1 Connect4

Figure 6 shows the results of all the trained Connect4 agents playing an optimal agent (described in section 4.2). The agents' winning rate is plotted as a function of the total number of simulations over 100 training generations (the agents use different strategies to decide on the number of simulations in each state; thus, the difference is the

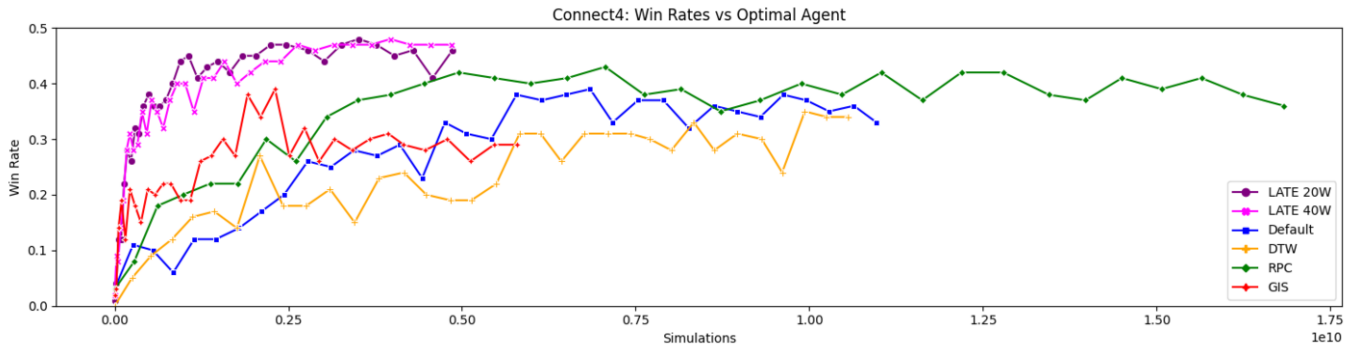

Figure 6: Connect4 agents win rate against a baseline player (an optimal agent).

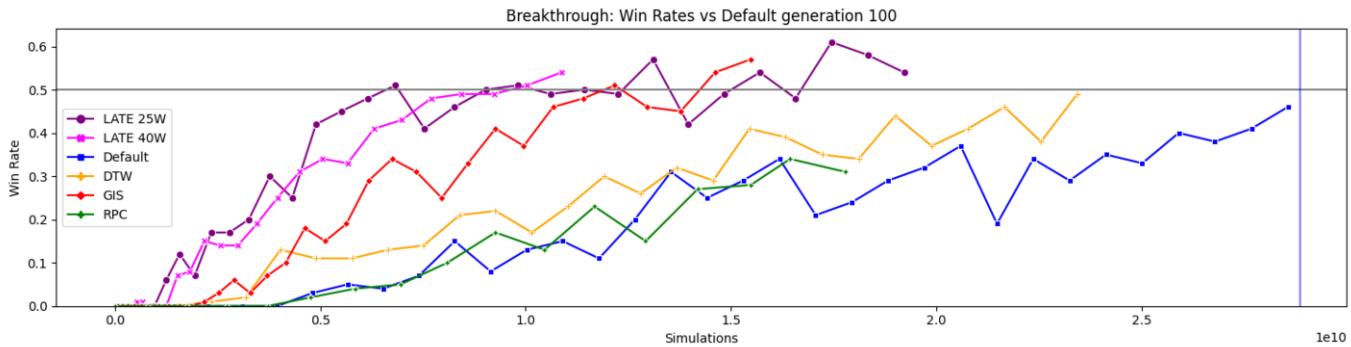

Figure 7: Breakthrough agents win rate against a baseline agent (the default trained 100 generations). All agents were trained for 100 generations except LATE-40W and RPC whose training is ongoing, here capped at 81 and 49 generations, respectively; however, the trend is clear.

total number of simulations between the agents). Each data point is based on 50 games beginning from the opening positions in O_{c4}, with the agents playing each position from both sides.

The LATE agents learn much faster than the others and, interestingly, are the only ones that, in the end, converge to playing on an (almost) equal level to the optimal agent. The LATE agent that uses the larger generation window holds a slight edge over its counterpart. The playing strength results are in harmony with the model quality results we presented earlier; after only a few simulations, the LATE agents reach greater playing strength than all the other agents and maintain that advantage throughout training. Impressively, even very early in their training, the LATE agents have reached a level of playing strength that the other agents have difficulty matching even at the end of their training (100 training generations).

4.6.2 Breakthrough

Figure 7 shows the results of all the Breakthrough agents competing against a baseline Breakthrough agent. The baseline agent is the default agent after training for 100 training generations (the maximum). Unfortunately, we do not have an objective measure of the baseline player's strength; however, as anecdotal evidence, it is a formidable opponent, consistently beating even expert-level chess players with both colorous. Each data point is based on 100 games played from the opening positions in O_{bt}, with the agents playing both sides.

The LATE agents again learn significantly faster than the others. Specifically, the agents require only ca. 20% of the default agent's training resources to match its performance. Moreover, it learns twice as fast as its closest rival of other self-play enhancements, GIS.

5 Conclusions and Future Work

We presented an enhanced scheme, LATE, to expedite self-play learning in AlphaZero-style game-playing agents and extensively evaluated it in two test domains. The former, Connect4, is a strongly solved game, which allowed us to continuously evaluate the quality of the training process in various ways, including how accurately the training samples were labeled, how accurate the trained models were, and how accurate the agents' decisions were. The latter test domain, Breakthrough, has a much larger state space and is strategically more complex than Connect4. In both domains, the LATE enhancement learned far more efficiently than a standard approach, requiring much less computation resource to reach similar or better quality of play. Also, unlike the standard approach, a positive side-effect of the enhancement is that it can effectively reuse data from much earlier training generations. Furthermore, we compared the LATE enhancements with several recently proposed enhancements and found that our approach outperformed the others in our test domains.

As for future work, we plan to carry on along three paths. First, we plan to experiment with the proposed enhancement in a broader range of games. Second, to make the first plan more easily attainable, we plan to automate the process of selecting various hyper-parameters, particularly those involved in computing $w(g, m)$; currently, they are manually selected based on expected game length and the number of training generations. Third, we plan a more thorough empirical comparison with related methods for improving self-play training — both those discussed in this paper and more recent ones [16] — where we tune each method more carefully for the domain at hand. This will allow us to determine their relative strengths and weaknesses more conclusively.

References

[1] Yngvi Björnsson, Róbert Leó Þormar Jónsson, and Sigurjón Ingi Jónsson. AlphaZeroLATE source code. https://github.com/Robertleoj/alpha_bsc_src/, 2023.

[2] Tristan Cazenave, Yen-Chi Chen, Guan-Wei Chen, Shi-Yu Chen, Xian-Dong Chiu, Julien Dehos, Maria Elsa, Qucheng Gong, Hengyuan Hu, Vasil Khalidov, Cheng-Ling Li, Hsin-I Lin, Yu-Jin Lin, Xavier Martinet, Vegard Mella, Jérémy Rapin, Baptiste Rozière, Gabriel Synnaeve, Fabien Teytaud, Olivier Teytaud, Shi-Cheng Ye, Yi-Jun Ye, Shi-Jim Yen, and Sergey Zagoruyko, 'Polygames: Improved zero learning', *CoRR*, **abs/2001.09832**, (2020).

[3] Rémi Coulom, 'Efficient selectivity and backup operators in monte-carlo tree search', in *Computers and Games, 5th International Conference, CG 2006, Turin, Italy, May 29-31, 2006. Revised Papers*, eds., H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. Donkers, volume 4630 of *Lecture Notes in Computer Science*, pp. 72–83. Springer, (2006).

[4] Chao Gao, Martin Müller 0003, and Ryan Hayward, 'Three-head neural network architecture for monte carlo tree search', in *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI 2018, July 13-19, 2018, Stockholm, Sweden*, ed., Jérôme Lang, pp. 3762–3768. ijcai.org, (2018).

[5] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, 'Deep residual learning for image recognition', *CoRR*, **abs/1512.03385**, (2015).

[6] Levente Kocsis and Csaba Szepesvári, 'Bandit based monte-carlo planning', in *Machine Learning: ECML 2006, 17th European Conference on Machine Learning, Berlin, Germany, September 18-22, 2006, Proceedings*, eds., Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, volume 4212 of *Lecture Notes in Computer Science*, pp. 282–293. Springer, (2006).

[7] Charles Lovering, Jessica Forde, George Konidaris, Ellie Pavlick, and Michael Littman, 'Evaluation beyond task performance: Analyzing concepts in alphazero in hex', *Advances in Neural Information Processing Systems*, **35**, 25992–26006, (2022).

[8] Antonio Norelli and Alessandro Panconesi, 'Olivaw: Mastering othello without human knowledge, nor a penny', *IEEE Transactions on Games*, 1–1, (2022).

[9] LCZero open-source authors. Leela chess zero.

[10] Pascal Pons. connect4. https://github.com/PascalPons/connect4, 2020.

[11] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis, 'Mastering the game of go with deep neural networks and tree search', *Nature*, **529**(7587), 484–489, (2016).

[12] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis, 'A general reinforcement learning algorithm that masters chess, shogi, and go through self-play', *Science*, **362**(6419), 1140–1144, (2018).

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

[14] Wu Ti-Rong, Wei Ting-Han, and Wu I-Chen, 'Accelerating and improving alphazero using population based training', in *Proceedings of the AAAI Conference on Artificial Intelligence*, pp. 1046–1053. AAAI Press, (2020).

[15] Yuandong Tian, Jerry Ma, Qucheng Gong, Shubho Sengupta, Zhuoyuan Chen, James Pinkerton, and Larry Zitnick, 'ELF OpenGo: an analysis and open reimplementation of AlphaZero', in *Proceedings of the 36th International Conference on Machine Learning*, eds., Kamalika Chaudhuri and Ruslan Salakhutdinov, volume 97 of *Proceedings of Machine Learning Research*, pp. 6244–6253. PMLR, (09–15 Jun 2019).

[16] Alexandre Trudeau and Michael Bowling, 'Targeted search control in alphazero for effective policy improvement', in *Proceedings of the 2023 International Conference on Autonomous Agents and Multiagent Systems, AAMAS 2023, London, United Kingdom, 29 May 2023 - 2 June 2023*, eds., Noa Agmon, Bo An, Alessandro Ricci, and William Yeoh, pp. 842–850. ACM, (2023).

[17] David J Wu, 'Accelerating self-play learning in go', *arXiv preprint arXiv:1902.10565*, (2019).