arXiv:1902.10565v3 [cs.LG] 17 Sep 2019

Accelerating Self-Play Learning in Go

David J. Wu*
Jane Street Group

September 18, 2019

Abstract

By introducing several improvements to the AlphaZero process and architecture, we greatly accelerate self-play learning in Go, achieving a 50x reduction in computation over comparable methods. Like AlphaZero and replications such as ELF OpenGo and Leela Zero, our bot KataGo only learns from neural-net-guided Monte Carlo tree search self-play. But whereas AlphaZero required thousands of TPUs over several days and ELF required thousands of GPUs over two weeks, KataGo surpasses ELF's final model after only 19 days on fewer than 30 GPUs. Much of the speedup involves non-domain-specific improvements that might directly transfer to other problems. Further gains from domain-specific techniques reveal the remaining efficiency gap between the best methods and purely general methods such as AlphaZero. Our work is a step towards making learning in state spaces as large as Go possible without large-scale computational resources.

1 Introduction

In 2017, DeepMind's AlphaGoZero demonstrated that it was possible to achieve superhuman performance in Go without reliance on human strategic knowledge or preexisting data [18]. Subsequently, DeepMind's AlphaZero achieved comparable results in Chess and Shogi. However, the amount of computation required was large, with DeepMind's main reported run for Go using 5000 TPUs for several days, totaling about 41 TPU-years [17]. Similarly ELF OpenGo, a replication by Facebook, used 2000 V100 GPUs for about 13-14 days[1], or about 74 GPU-years, to reach top levels of performance [19].

In this paper, we introduce several new techniques to improve the efficiency of self-play learning, while also reviving some pre-AlphaZero ideas in computer Go and newly applying them to the AlphaZero process. Although our bot KataGo uses some domain-specific features and optimizations, it still starts from random play and makes no use of outside strategic knowledge or preexisting data. It surpasses the strength of ELF OpenGo after training on about 27 V100 GPUs for 19 days, a total of about 1.4 GPU-years, or about a factor of 50 reduction. And by a conservative comparison,

*email:dwu@janestreet.com. Many thanks to Craig Falls, David Parkes, James Somers, and numerous others for their kind feedback and advice on this work.

[1] Although ELF's training run lasted about 16 days, its final model was chosen from a point slightly prior to the end of the run.

KataGo is also at least an order of magnitude more efficient than the multi-year-long online distributed training project Leela Zero [14]. Our code is open-source, and superhuman trained models and data from our main run are available online[2].

We make two main contributions:

First, we present a variety of domain-independent improvements that might directly transfer to other AlphaZero-like learning or to reinforcement learning more generally. These include: (1) a new technique of *playout cap randomization* to improve the balance of data for different targets in the AlphaZero process, (2) a new technique of *policy target pruning* that improves policy training by decoupling it from exploration in MCTS, (3) the addition of a *global-pooling* mechanism to the neural net, agreeing with research elsewhere on global context in image tasks such as by Hu et al. (2018) [8], and (4) a revived idea from supervised learning in Go to add *auxiliary policy targets* from future actions tried by Tian and Zhu (2016) [20], which we find transfers easily to self-play and could apply widely to other problems in reinforcement learning.

Second, our work serves as a case study that there is still a significant efficiency gap between AlphaZero's methods and what is possible from self-play. We find nontrivial further gains from some domain-specific methods. These include *auxiliary ownership and score targets* (similar to those in Wu et al. 2018 [22]) and which actually also suggest a much more general meta-learning heuristic: that predicting subcomponents of desired targets can greatly improve training. We also find that a set of standard game-specific input features still significantly accelerates learning, showing that AlphaZero does not yet obsolete even simple additional tuning.

In Section 2 we summarize our architecture. In Sections 3 and 4 we outline the general techniques of *playout cap randomization*, *policy target pruning*, *global-pooling*, and *auxiliary policy targets*, followed by domain-specific improvements including *ownership and score targets* and input features. In Section 5 we present our data, including comparison runs showing how these techniques each improve learning and all similarly contribute to the final result.

2 Basic Architecture and Parameters

Although varying in many minor details, KataGo's overall architecture resembles the AlphaGoZero and AlphaZero architectures [18, 17].

KataGo plays games against itself using Monte-Carlo tree search (MCTS) guided by a neural net to generate training data. Search consists of growing a game tree by repeated playouts. Playouts start from the root and descend the tree, at each node n choosing the child c that maximizes:

$$\mathrm{PUCT}(c) = V(c) + c_{\mathrm{PUCT}} P(c) \frac{\sqrt{\sum_{c'} N(c')}}{1 + N(c)}$$

where $V(c)$ is the average predicted utility of all nodes in c's subtree, $P(c)$ is the policy prior of c from the neural net, $N(c)$ is the number of playouts previously sent through child c, and $c_{\mathrm{PUCT}} = 1.1$. Upon reaching the end of the tree and finding that the next chosen child is not

[2] https://github.com/lightvector/KataGo. For interested enthusiasts, using our code starting from nothing, it is possible to reach strong or top human amateur strength on ordinary consumer hardware in just days.

allocated, the playout terminates by appending that single child to the tree.[3]

Like AlphaZero, to aid discovery of unexpected moves, KataGo adds noise to the policy prior at the root:

$$P(c) = 0.75 P_{\text{raw}}(c) + 0.25\,\eta$$

where η is a draw from a Dirichlet distribution on legal moves with parameter $\alpha = 0.03 * 19^2/\text{N}(c)$ where N is the total number of legal moves. This matches AlphaZero's $\alpha = 0.03$ on the empty 19×19 Go board while scaling to other sizes. KataGo also applies a softmax temperature at the root of 1.03, an idea to improve policy convergence stability from SAI, another AlphaGoZero replication [13] .

The neural net guiding search is a convolutional residual neural net using a *preactivation* architecture [7], with a trunk of b residual blocks with c channels. Similar to Leela Zero [14], KataGo began with small nets to make early training cheaper, and progressively increased their size, concurrently training the next larger size on the same data until it began to overtake the smaller. In KataGo's main 19-day run, (b, c) began at $(6, 96)$ and switched to $(10, 128)$, $(15, 192)$, and $(20, 256)$, at roughly 0.75 days, 1.75 days, and 7.5 days, respectively. The final size approximately matches that of AlphaZero and ELF.

The neural net has several output heads attached to the trunk. Sampling positions from the self-play games, a *policy* head predicts probable good moves while a *game outcome value head* predicts if the game was won or lost from that position. The loss function is:

$$L = c_{\text{g}} \sum_r z(r) \log(\hat{z}(r)) - \sum_m \pi(m) \log(\hat{\pi}(m)) + c_{L2}||\theta||^2$$

where $r \in \{\text{win}, \text{loss}\}$ is the outcome for the current player, z is a one-hot encoding of it, $\hat{z}$ is the neural net's prediction of z, m ranges over the set of possible moves, π is a target policy distribution derived from the playout distribution of the MCTS search, $\hat{\pi}$ is the prediction of π, $c_{L2} = 3\text{e-}5$ is an L2 penalty coefficient on the model parameters θ, and $c_{\text{g}} = 1.5$ is a scaling constant. As described in later sections, we also add additional terms corresponding to other heads that predict auxiliary targets.

Training uses stochastic gradient descent with a momentum decay of 0.9 and a batch size of 256 samples. It uses a fixed per-sample learning rate of 6e-5, except that the first approximately 5 million samples use a learning rate of 2e-5 to reduce instability from large gradients early on. In KataGo's main run, the per-sample learning rate was also dropped to 6e-6 starting at about 17.5 days to maximize final strength. Samples are drawn uniformly from a growing moving window of the most recent data, with window size beginning at 250,000 samples and increasing to about 22 million by the end of the main run. See Appendix C for details.

Training uses a version of *stochastic weight averaging* [9]. Every roughly 250,000 training samples, a snapshot of the weights is saved, and every four snapshots, a new candidate neural net is produced by taking an exponential moving average of snapshots with decay = 0.75 (i.e., averaging four snapshots of lookback). Candidate nets must then pass a *gating* test by winning at least 100 out of 200 test games against the current net to become the new net for self-play. See Appendix E for details.

[3] When $N(c) = 0$ and $V(c)$ is undefined, unlike AlphaZero but like Leela Zero, we define: $V(c) = V(n) - c_{\text{FPU}}\sqrt{P_{\text{explored}}(n)}$ where P_{explored} is the total policy mass of explored moves and $c_{\text{FPU}} = 0.2$ is a "first-play-urgency" reduction coefficient, except $c_{\text{FPU}} = 0$ at the root if Dirichlet noise is enabled.

In total, KataGo's main run lasted for 19 days using a maximum of 28 V100 GPUs at any time (averaging 26-27) and generated about 241 million training samples across 4.2 million games. Self-play games used Tromp-Taylor rules [21] modified to not require capturing stones within pass-alive-territory[4]. "Ko", "suicide", and "komi" rules also varied from Tromp-Taylor randomly, and some proportion of games were randomly played on smaller boards.[5]See Appendix D for other details on game initialization.

3 Major General Improvements

3.1 Playout Cap Randomization

One of the major improvements in KataGo's training process over AlphaZero is to randomly vary the number of playouts on different turns to relieve a major tension between policy and value training.

In the AlphaZero process, the game outcome value target is highly data-limited, with only one independent sample per entire game, a noisy binary win or loss. Holding compute fixed, it would likely be beneficial for value training to use only a small number of playouts per turn to generate more games, even if those games are of slightly lower quality. For example, in the first version of AlphaGo, self-play using only a single playout per turn (i.e., directly using the policy) was still of sufficient quality to train a decent value net [16].

However, informal prior research by Forsten (2019) [6] has suggested that at least in Go, ideal numbers of playouts for policy learning are much larger, in fact not far from AlphaZero's choice of 800 playouts per move [17]. Although the policy gets many samples per game, with too few playouts the search usually does not deviate sufficiently from the policy prior, so the policy does not readily improve over time unless the number of playouts is much larger than ideal for value training.

We introduce *playout cap randomization* to mitigate this tension. On a small proportion p of turns, we perform a full search, stopping when the tree reaches a cap of N nodes, and for all other turns we perform a fast search with a much smaller cap of $n < N$. Only turns with a full search are recorded for training. For fast searches, we also disable Dirichlet noise and other explorative settings, maximizing strength. For KataGo's main 19-day run, we chose $p = 0.25$ and $(N, n) = (600, 100)$ initially, annealing up to $(1000, 200)$ after the first two days of training.

Because most moves use a fast search, more games are played, improving value training. But since n is small, fast searches cost only a limited fraction of the computation time, so the drop in the number of good policy samples per computation time is not large. The ablation studies presented in section 5.2 indicate that playout cap randomization indeed outperforms a variety of fixed numbers of playouts.

[4]In Go, a version of Benson's algorithm [1] can prove areas safe even given unboundedly many consecutive opponent moves ("pass-alive"), enabling this minor optimization.

[5]Almost all major AlphaZero reproductions in Go have been hardcoded to fixed board sizes and rules. Although not the focus of this paper, KataGo's randomization allows training a *single* model that generalizes across all these variations.

3.2 Forced Playouts and Policy Target Pruning

Like AlphaZero and other implementations such as ELF and Leela Zero, KataGo uses the final root playout distribution from MCTS to produce the policy target for neural net training. However, KataGo does *not* use the raw distribution. Instead, we introduce *policy target pruning*, a new method which enables more effective exploration via *forced playouts*.

We observed in informal tests that even if a Dirichlet noise move was good, the neural net's evaluation of it might initially be negative, preventing further search and leaving the move undiscovered. Therefore, for each child c of the root that has received any playouts, we ensure it further receives a minimum number of *forced playouts* based on the noised policy and the total sum of playouts so far:

$$n_{\text{forced}}(c) = \left(2P(c) \sum_{c'} N(c') \right)^{1/2}$$

We do this by setting the MCTS selection urgency $\text{PUCT}(c)$ to infinity whenever a child of the root has fewer than this many playouts. In practice this does not cost more than a few percent of playouts.

However, the vast majority of the time, noise moves are bad moves, and in AlphaZero since the policy target is the playout distribution, we would train the policy to predict these extra bad playouts. Therefore, we perform a *policy target pruning* step that subtracts playouts unless a move actually was found to be good. In particular, we identify the child c^* with the most playouts, and then from each other child c, we subtract up to n_{forced} playouts so long as it does not cause $\text{PUCT}(c) >= \text{PUCT}(c^*)$, holding constant the *final* utility estimate for both. Additionally, we outright prune children that are reduced to only a single playout. See Figure 1 for a visualization of how this affects the learned policy.

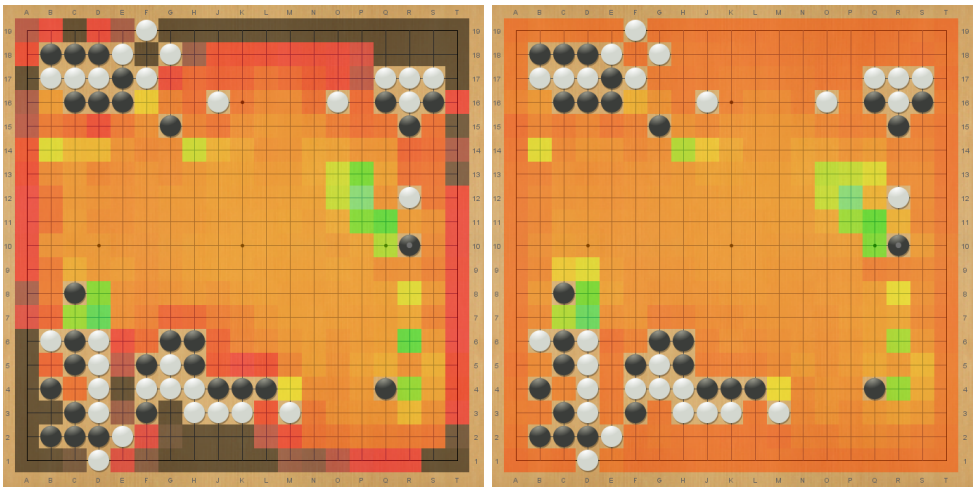

Figure 1: Log policy of 10-block nets, white to play. Left: trained with forced playouts and policy target pruning. Right: trained without. Dark/red through bright green ranges from about $p =$ 2e-4 to $p = 1$. Pruning reduces the policy mass on many bad moves near the edges.

The critical feature of such pruning is that it allows *decoupling the policy target in AlphaZero from the dynamics of MCTS or the use of explorative noise*. There is no reason to expect the optimal

level of playout dispersion in MCTS to also be optimal for the policy target and the long-term convergence of the neural net. Our use of policy target pruning with forced playouts, though an improvement, is only a simple application of this method. We are eager to explore others in the future, including alterations to the PUCT formula itself[6].

3.3 Global Pooling

Another improvement in KataGo over earlier work is from adding *global pooling* layers at various points in the neural network. This enables the convolutional layers to condition on global context, which would hard or impossible with the limited perceptual radius of convolution alone.

In KataGo, given a set of c channels, a *global pooling layer* computes (1) the mean of each channel, (2) the mean of each channel scaled linearly with the width of the board, and (3) the maximum of each channel. This produces a total of $3c$ output values. These layers are used in a *global pooling bias structure* consisting of:

- Input tensors X (shape $b \times b \times c_X$) and G (shape $b \times b \times c_G$).
- A batch normalization layer and ReLu activation applied to G (output shape $b \times b \times c_G$).
- A global pooling layer (output shape $3c_G$).
- A fully connected layer to c_X outputs (output shape c_X).
- Channelwise addition with X (output shape $b \times b \times c_X$).

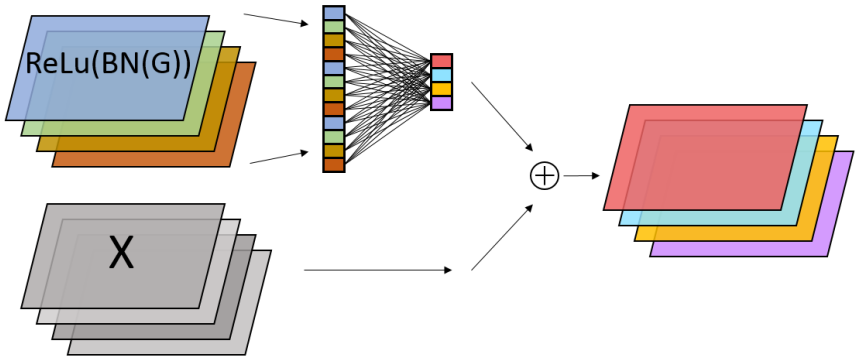

Figure 2: Global pooling bias structure, globally aggregating values of one set of channels to bias another set of channels.

See Figure 2 for a diagram. This structure follows the first convolution layer of two to three of the residual blocks in KataGo's neural nets, and the first convolution layer in the policy head. It is also used in the value head with a slight further modification. See Appendix A for details.

In Section 5.2 our experiments show that this greatly improves the later stages of training. As Go contains explicit nonlocal tactics ("ko fights"), this is not surprising. But global context should

[6]The PUCT formula $V(c) + c_{\text{PUCT}} P(c) \frac{\sqrt{\sum_{c'} N(c')}}{1+N(c)}$ has the property that if V is constant, then playouts will be roughly proportional to P. Informal tests suggest this is important to the convergence of P, and without something like target pruning, alternate formulas can disrupt training even when improving match strength.

help even in domains without explicit nonlocal interactions. For example, in a wide variety of strategy games, strong players, when winning, alter their local move preferences to favor "simple" options, whereas when losing they seek "complication". Global pooling allows convolutional nets to internally condition on such global context.

The general idea of using global context is by no means novel to our work. For example, Hu et al. (2018) have introduced a "Squeeze-and-Excitation" architecture to achieve new results in image classification [8]. Although their implementation is different, the fundamental concept is the same. And though not formally published, Squeeze-Excite-like architectures are now in use in some online AlphaZero-related projects [10, 11], and we look forward to exploring it ourselves in future research.

3.4 Auxiliary Policy Targets

As another generalizable improvement over AlphaZero, we add an auxiliary policy target that predicts the opponent's reply on the following turn to improve regularization. This idea is not entirely new, having been found by Tian and Zhu in Facebook's bot Darkforest to improve supervised move prediction [20], but as far as we know, KataGo is the first to apply it to the AlphaZero process.

We simply have the policy head output a new channel predicting this target, adding a term to the loss function:

$$-w_{\text{opp}} \sum_{m \in \text{moves}} \pi_{\text{opp}}(m) \log(\hat{\pi}_{\text{opp}}(m))$$

where π_{opp} is the policy target that will be recorded for the turn *after* the current turn, $\hat{\pi}_{\text{opp}}$ is the neural net's prediction of π_{opp}, and $w_{\text{opp}} = 0.15$ weights this target only a fraction as much as the actual policy, since it is for regularization only and is never actually used for play.

We find in Section 5.2 that this produces a modest but clear benefit. Moreover, this idea could apply to a wide range of reinforcement-learning tasks. Even in single-agent situations, one could predict one's own future actions, or predict the environment (treating the environment as an "agent"). Along with Section 4.1, it shows how enriching the training data with additional targets is valuable when data is limited or expensive. We believe it deserves attention as a simple and nearly costless method to regularize the AlphaZero process or other broader learning algorithms.

4 Major Domain-Specific Improvements

4.1 Auxiliary Ownership and Score Targets

One of the major improvements in KataGo's training process over AlphaZero comes from adding auxiliary ownership and score prediction targets. Similar targets were earlier explored in work by Wu et al. (2018) [22] in supervised learning, where the authors found improved mean squared error on human game result prediction and mildly improved the strength of their overall bot, CGI.

To our knowledge, KataGo is the first to publicly apply such ideas to the reinforcement-learning-like context of self-play training in Go[7]. While the targets themselves are game-specific, they also highlight a more general heuristic underemphasized in transfer- and multi-task-learning literature.

[7] A bot "Golaxy" developed by a Chinese research group appears also capable of making score predictions, but we are not currently aware of anywhere they have published their methods.

As observed earlier, in AlphaZero, learning is highly constrained by data and noise on the game outcome prediction. But although the game outcome is noisy and binary, it is a direct function of finer variables: the final score difference and the ownership of each board location[8]. Decomposing the game result into these finer variables and predicting them as well should improve regularization.

Therefore, we add these outputs and three additional terms to the loss function:

- Ownership loss:

$$-w_o \sum_{l \in \text{board}} \sum_{p \in \text{players}} o(l,p) \log\left(\hat{o}(l,p)\right)$$

  where $o(l,p) \in \{0, 0.5, 1\}$ indicates if l is finally owned by p, or is shared, $\hat{o}$ is the prediction of o, and $w_o = 1.5/b^2$ where $b \in [9, 19]$ is the board width.

- Score belief loss ("pdf"):

$$-w_{\text{spdf}} \sum_{x \in \text{possible scores}} p_s(x) \log(\hat{p}_s(x))$$

  where p_s is a one-hot encoding of the final score difference, $\hat{p}_s$ is the prediction of p_s, and $w_{\text{spdf}} = 0.02$.

- Score belief loss ("cdf"):

$$w_{\text{scdf}} \sum_{x \in \text{possible scores}} \left(\sum_{y<x} p_s(y) - \hat{p}_s(y) \right)^2$$

  where $w_{\text{scdf}} = 0.02$. While the "pdf" loss rewards guessing the score exactly, this "cdf" loss pushes the overall mass to be near the final score.

We show in our ablation runs in Section 5.2 that these auxiliary targets noticeably improve the efficiency of learning. This holds even up through the ends of those runs (though shorter, the runs still reach a strength similar to human-professional), well beyond where the neural net must have already developed a sophisticated judgment of the board.

It might be surprising that these targets would continue to help beyond the earliest stages. We offer an intuition: consider the task of updating from a game primarily lost due to misjudging a particular region of the board. With only a final binary result, the neural net can only "guess" at what aspect of the board position caused the loss. By contrast, with an ownership target, the neural net receives direct feedback on which area of the board was mispredicted, with large errors and gradients localized to the mispredicted area. The neural net should therefore require fewer samples to perform the correct credit assignment and update correctly.

As with auxiliary policy targets, these results are consistent with work in transfer and multi-task learning showing that adding targets or tasks can improve performance. But the literature is scarcer in theory on *when* additional targets may help – see Zhang and Yang (2017) [23] for discussion as well as Bingel and Sgaard (2017) [2] for a study in NLP domains. Our results suggest

[8] In Go, every point occupied or surrounded at the end of the game scores 1 point. The second player also receives a *komi* of typically 7.5 points. The player with more points wins.

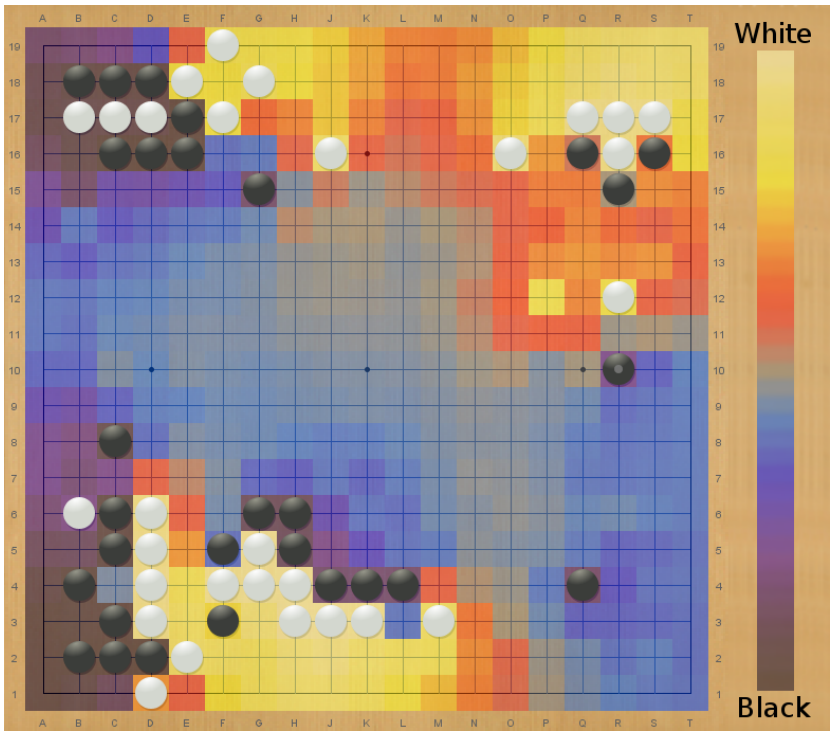

Figure 3: Visualization of ownership predictions by the trained neural net.

a heuristic: *whenever a desired target can be expressed as a sum, conjunction, or disjunction of separate subevents, or would be highly correlated with such subevents, predicting those subevents is likely to help.* This is because such a relation should allow for a specific mechanism: that gradients from a mispredicted sub-event will provide sharper, more localized feedback than from the overall event, improving credit assignment.

We are likely not the first to discover such a heuristic, and further research would be valuable to test and confirm it. And of course, it may not always be applicable. But we feel it is worth highlighting both for practical use, and because when applicable, it is a potential avenue by which meta-learning approaches might reduce the gap between the best current general methods and better domain-specific methods.

4.2 Game-specific Features

In addition to raw features indicating the stones on the board, the history, and the rules and komi in effect, KataGo includes a few game-specific higher-level features in the input to its neural net, similar to those in earlier work [4, 3, 12]. These features are liberties, komi parity, pass-alive regions, and features indicating ladders (a particular kind of capture tactic). See Appendix A for details.

Additionally, KataGo uses two minor Go-specific optimizations, where after a certain number of consecutive passes, moves in pass-alive territory are prohibited, and where a tiny bias is added to favor passing when passing and continuing play would lead to identical scores. Both optimizations slightly speed up the end of the game.

To measure the effect of these game-specific features and optimizations, we include in Section 5.2 an ablation run that disables both ending optimizations and all input features other than the locations of stones, previous move history, and game rules. We find they contribute noticeably to the learning speed, but account for only a small fraction of the total improvement in KataGo.

5 Results

5.1 Testing Versus ELF and Leela Zero

We tested KataGo against ELF and Leela Zero 0.17 using their publicly-available source code and trained networks.

We sampled roughly every fifth Leela Zero neural net over its training history from "LZ30" through "LZ225", the last several networks well exceeding even ELF's strength. Between every pair of Leela Zero nets fewer than 35 versions apart, we played about 45 games to establish approximate relative strengths of the Leela Zero nets as a benchmark.

We also sampled KataGo over its training history, for each version playing batches of games versus random Leela Zero nets with frequency proportional to the predicted variance $p(1-p)$ of the game result. The winning chance p was continuously estimated from the global Bayesian maximum-likelihood Elo based on all game results so far[9]. This ensured that games would be varied yet informative. We also ran ELF's final "V2" neural network using Leela Zero's search engine[10], with ELF playing against both Leela Zero and KataGo using the same variance-based sampling.

Games used a 19x19 board with a fixed 7.5 komi under Tromp-Taylor rules, with a fixed 1600 visits, resignation below 2% winrate, and multithreading disabled. To encourage opening variety, Leela Zero was set to randomize with a temperature of 0.2 in the first 20 turns, and KataGo with a temperature of 0.3 decaying to 0 with a 30-turn halflife. Both also used a "lower-confidence-bound" move selection method to improve match strength [15]. Final Elo ratings were based on the final set of about 43000 games.

To compare the efficiency of training, we computed a crude indicative metric of total self-play computation by modeling a neural net with b blocks and c channels as having cost $\sim bc^2$ per query[11]. For KataGo we just counted self-play queries for each size and multiplied. For ELF, we approximated queries by sampling the average game length from its public training data and multiplied by ELF's 1600 playouts per move, discounting by 20% to roughly account for transposition caching. For Leela Zero we estimated it similarly, also interpolating costs where data was missing[12]. Leela Zero also generated data using ELF's prototype networks, but we did *not* attempt to estimate this cost[13].

KataGo compares highly favorably with both ELF and Leela Zero. Shown in Figure 4 is a plot of Elo ratings versus estimated compute for all three. KataGo outperforms ELF in learning efficiency

[9] Using a custom implementation of BayesElo [5].

[10] ELF and Leela Zero neural nets are inter-compatible.

[11] This metric was chosen in part as a very rough way to normalize out hardware and engineering differences. For KataGo, we also conservatively computed costs under this metric as if all queries were on the full 19x19 board.

[12] Due to online hosting issues, some Leela Zero training data is no longer publicly available.

[13] At various points, Leela Zero also used data from stronger ELF OpenGo nets, likely causing it to learn faster than it would unaided. We did *not* attempt to count the cost of this additional data.

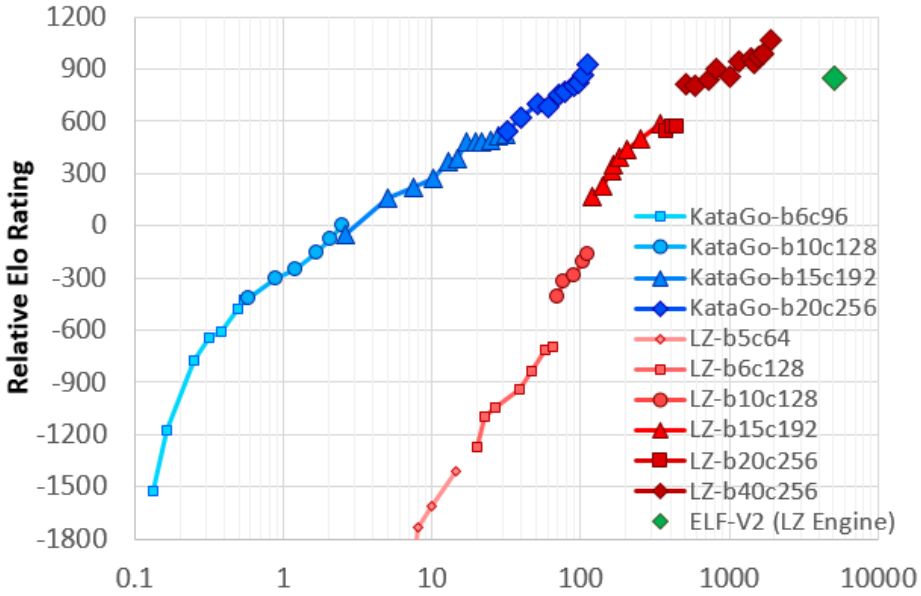

Figure 4: 1600-visit Elo progression of KataGo (blue, leftmost) vs. Leela Zero (red, center) and ELF (green diamond). X-axis: self-play cost in billions of equivalent 20 block x 256 channel queries. Note the log-scale. Leela Zero's costs are highly approximate.

Match Settings	Wins v ELF	Elo Diff
1600 playouts/mv no batching	239 / 400	69 ± 36
9.0 secs/mv, ELF batchsize 16	246 / 400	81 ± 36
7.5 secs/mv, ELF batchsize 32	254 / 400	96 ± 37

Table 1: KataGo match results versus ELF, with the implied Elo difference (plus or minus two std. deviations of confidence).

under this metric by about a factor of 50. Leela Zero appears to outperform ELF as well, but the Elo ratings would be expected to unfairly favor Leela since its final network size is 40 blocks, double that of ELF, and the ratings are based on equal search nodes rather than GPU cost. Additionally, Leela Zero's training occurred over multiple years rather than ELF's two weeks, reducing latency and parallelization overhead. Yet KataGo still outperforms Leela Zero by a factor of 10 despite the same network size as ELF and a similarly short training time. Early on, the improvement factor appears larger, but partly this is because the first 10%-15% of Leela Zero's run contained some bugs that slowed learning.

We also ran three 400-game matches on a single V100 GPU against ELF using ELF's native engine. In the first, both sides used 1600 playouts/move with no batching. In the second, KataGo used 9s/move (16 threads, max batch size 16) and ELF used 16,000 playouts/move (batch size 16), which ELF performs in 9 seconds. In the third, we doubled ELF's batch size, improving its nominal speed to 7.5s/move, and lowered KataGo to 7.5s/move. As summarized in Table 1, in all three matches KataGo defeated ELF, confirming its strength level at both low and high playouts and at both fixed search and fixed wall clock time settings.

5.2 Ablation Runs

To study the impact of the techniques presented in this paper, we ran shorter training runs with various components removed. These ablation runs went for about 2 days each, with identical parameters except for the following differences:

- FixedN - Replaces playout cap randomization with a fixed cap $N \in \{100, 150, 200, 250, 600\}$. For $N = 600$ the window size was also doubled, as an informal test without doubling showed major overfitting due to lack of data.
- NoForcedTP - Removes forced playouts and policy target pruning.
- NoGPool - Removes global pooling from residual blocks and the policy head except for computing the "pass" output.
- NoPAux - Removes the auxiliary policy target.
- NoVAux - Removes the auxiliary ownership and score targets.
- NoGoFeat - Removes all game-specific higher-level input features and the minor optimizations involving passing listed in Section 4.2.

We sampled neural nets from these runs together with KataGo's main run, and evaluated them the same way as when testing against Leela Zero and ELF: playing 19x19 games between random versions based on the predicted variance $p(1-p)$ of the result. Final Elos are based on the final set of about 147,000 games (note that these Elos are not directly comparable those in Section 5.1).

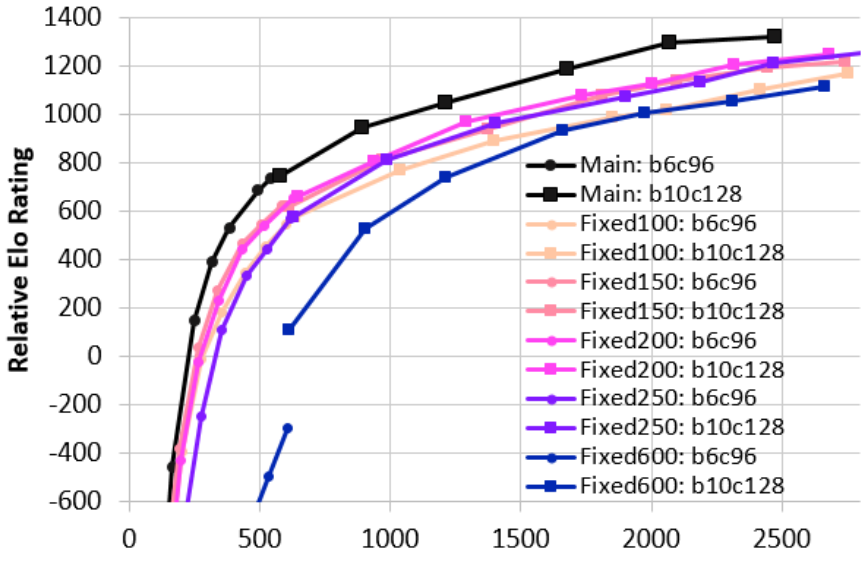

Figure 5: KataGo's main run versus Fixed runs. X-axis is the cumulative self-play cost in millions of equivalent 20 block x 256 channel queries.

As shown in Figure 5, playout cap randomization clearly outperforms a wide variety of possible fixed values of playouts. This is precisely what one would expect if the technique relieves the tension between the value and policy targets present for any single fixed number of playouts. Interestingly,

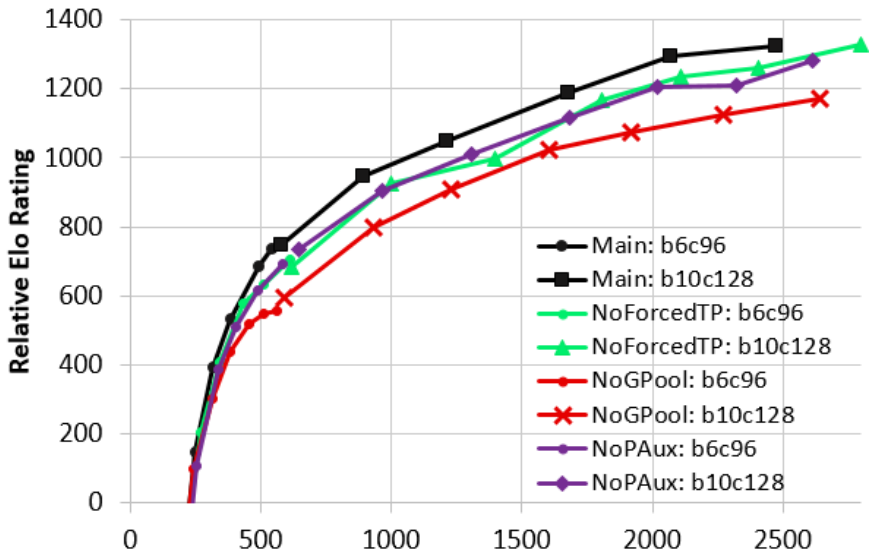

Figure 6: KataGo's main run versus NoGPool, NoForcedTP, NoPAux. X-axis is the cumulative self-play cost in millions of equivalent 20 block x 256 channel queries.

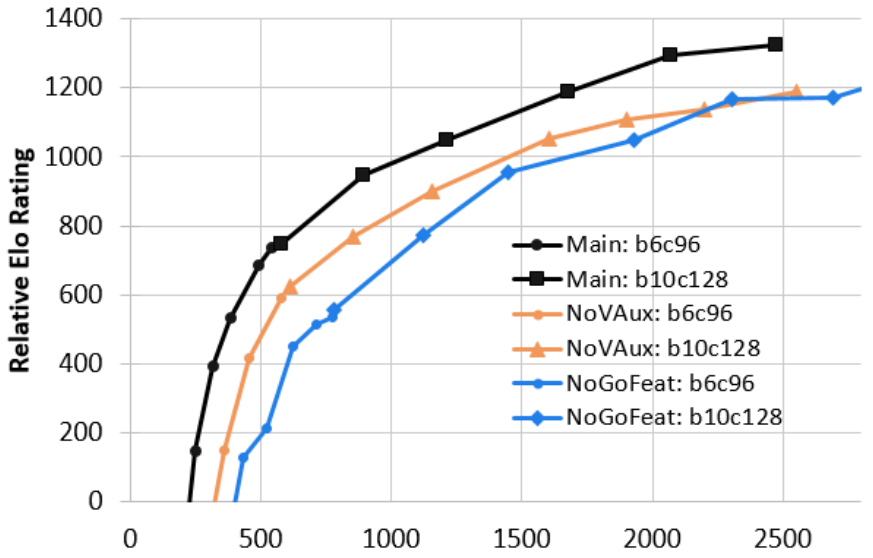

Figure 7: KataGo's main run versus NoVAux, NoGoFeat. X-axis is the cumulative self-play cost in millions of equivalent 20 block x 256 channel queries.

the 600-playout run, unlike any other run, showed a large jump in strength when increasing neural net size. We suspect this is due to poor convergence of the smaller net from early overfitting that was not entirely mitigated by doubling the training window.

As shown in Figure 6, global pooling noticeably improved learning efficiency, and forced playouts with policy target pruning and auxiliary policy targets also provided smaller but clear gains. Interestingly, all three show very little effect on the earliest parts of training compared to their later effects, greatly increasing over time. It is quite possible that their relative value continues to increase beyond the two-day mark at which we stopped the ablation runs. These plots suggest that the total value of these general enhancements to self-play learning, along with playout cap randomization, is large.

As shown in Figure 7, removing auxiliary ownership and score targets resulted in a noticeable drop in learning efficiency. These results confirm the value of these auxiliary targets and the value, at least in Go, of regularization by predicting subcomponents of one's actual desired targets. Also, we observe a significant drop in efficiency from removing Go-specific input features and optimizations, demonstrating that there is still significant value in such domain-specific methods, but also accounting for only a part of the total speedup achieved by KataGo. See Table 2 for a summary of these runs.

Removed Component	Elo	Slowdown
(Main Run)	1329	-
Playout Cap Randomization	1242	1.37x
F.P. and Policy Target Pruning	1276	1.25x
Global Pooling	1153	1.60x
Auxiliary Policy Targets	1255	1.30x
Aux Owner and Score Targets	1139	1.65x
Game-specific Features and Opts	1168	1.55x

Table 2: For each technique, the Elo of the best ablation run without it as of reaching 2.5G equivalent 20b x 256c self-play queries ($\approx$ 2 days), and the slowdown factor for achieving that Elo relative to the main run. Factors are *approximate* and likely to significantly differ in a full-length run.

6 Conclusions And Future Work

Still beginning only from random play with no external data, our bot KataGo achieves a level competitive with some of the top AlphaZero replications, but with an enormously greater efficiency than all such earlier work. In this paper, we presented a variety of the techniques we used to improve self-play learning, many of which could be readily applied to other games or to problems in reinforcement learning more generally. Furthermore, our domain-specific improvements demonstrate a gap between basic AlphaZero-like training and what could be possible with better methods, while also suggesting principles and possible avenues for improvement in general methods. We hope our work lays a foundation for further improvements in the data efficiency of reinforcement learning.

References

[1] David Benson. Life in the game of go. *Information Sciences*, 10:17–29, 1976.

[2] Joachim Bingel and Anders Sgaard. Identifying beneficial task relations for multi-task learning in deep neural networks. In *European Chapter of the Association for Computational Linguistics*, 2017.

[3] Tristan Cazenave. Residual networks for computer go. *IEEE Transactions on Games*, 10(1):107–110, 2017.

[4] Christopher Clark and Amos Storkey. Training deep convolutional neural networks to play go. In *32nd International Conference on Machine Learning*, page 17661774, 2015.

[5] Remi Coulom. Bayesian elo rating, 2010. https://www.remi-coulom.fr/Bayesian-Elo/.

[6] Henrik Forsten. Optimal amount of visits per move, 2019. Leela Zero project issue, https://github.com/leela-zero/leela-zero/issues/1416.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In *European Conference on Computer Vision*, page 630645. Springer, 2016.

[8] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, and Enhua Wu. Squeeze-and-excitation networks. In *IEEE Conference on Computer Vision and Pattern Recognition*, pages 7132–7141, 2018.

[9] Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, and Andrew Gordon Wilson. Averaging weights leads to wider optima and better generalization. In *Conference on Uncertainty in Artificial Intelligence*, 2018.

[10] Gary Linscott et al., 2019. Leela Chess Zero project main webpage, https://lczero.org/.

[11] Tom Madams, Andrew Jackson, et al., 2019. MiniGo project main GitHub page, https://github.com/tensorflow/minigo/.

[12] Chris Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move evaluation in go using deep convolutional neural networks. In *International Conference on Learning Representations*, 2015.

[13] Francesco Morandin, Gianluca Amato, Marco Fantozzi, Rosa Gini, Carlo Metta, and Maurizio Parton. Sai: a sensible artificial intelligence that plays with handicap and targets high scores in 9x9 go (extended version), 2019. arXiv preprint, arXiv:1905.10863.

[14] Gian-Carlo Pascutto et al., 2019. Leela Zero project main webpage, https://zero.sjeng.org/.

[15] Jonathan Roy. Fresh max_lcb_root experiments, 2019. Leela Zero project issue, https://github.com/leela-zero/leela-zero/issues/2282.

[16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484489, 2016.

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

[18] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550:354359, 2017.

[19] Yuandong Tian, Jerry Ma, Qucheng Gong, Shubho Sengupta, Zhuoyuan Chen, James Pinkerton, and C. Lawrence Zitnick. Elf opengo: An analysis and open reimplementation of alphazero. In *Thirty-Sixth International Conference on Machine Learning*, 2019.

[20] Yuandong Tian and Yan Zhu. Better computer go player with neural network and long-term prediction. In *International Conference on Learning Representations*, 2016.

[21] John Tromp. The game of go, 2014. http://tromp.github.io/go.html.

[22] Ti-Rong Wu, I-Chen Wu, Guan-Wun Chen, Ting han Wei, Tung-Yi Lai, Hung-Chun Wu, and Li-Cheng Lan. Multi-labelled value networks for computer go. *IEEE Transactions on Games*, 10(4):378–389, 2018.

[23] Yu Zhang and Qiang Yang. A survey on multi-task learning, 2017. arXiv preprint, arXiv:1707.08114.

Appendix A Neural Net Inputs and Architecture

The following is a detailed breakdown of KataGo's neural net inputs and architecture. The neural net has two input tensors, which feed into a *trunk* of residual blocks. Attached to the end of the trunk are a *policy head* and a *value head*, each with several outputs and subcomponents.

A.1 Inputs

The input to the neural net consists of two tensors, a $b \times b \times 18$ tensor of 18 binary features for each board location where where $b \in [b_{\min}, b_{\max}] = [9, 19]$ is the width of the board, and a vector with 10 real values indicating overall properties of the game state. These features are summarized in Tables 3 and 4

# Channels	Feature
1	Location is on board
2	Location has {own,opponent} stone
3	Location has stone with {1,2,3} liberties
1	Moving here illegal due to ko/superko
5	The last 5 move locations, one-hot
3	Ladderable stones {0,1,2} turns ago
1	Moving here catches opponent in ladder
2	Pass-alive area for {self,opponent}

Table 3: Binary spatial-varying input features to the neural net. A "ladder" occurs when stones are forcibly capturable via consecutive inescapable atari (i.e. repeated capture threat).

# Channels	Feature
5	Which of the previous 5 moves were pass?
1	Komi / 15.0 (current player's perspective)
2	Ko rules (simple,positional,situational)
1	Suicide allowed?
1	Komi + board size parity

Table 4: Overall game state input features to the neural net.

A.2 Global Pooling

Certain layers in the neural net are *global pooling layers*. Given a set of c channels, a *global pooling layer* computes:

1. The mean of each channel
2. The mean of each channel multiplied by $\frac{1}{10}(b - b_{\text{avg}})$
3. The maximum of each channel.

where $b_{\text{avg}} = 0.5(b_{\min} + b_{\max}) = 0.5(9 + 19)$. This produces a total of $3c$ output values. Scaling by board size in (2) allows training weights that work across multiple board sizes, and the subtraction and scaling by 1/10 is to improve orthogonality and ensure values remain near unit scale. In the *value head*, (3) is replaced with the mean of each channel multiplied by $\frac{1}{100}((b - b_{\text{avg}})^2 - \sigma^2)$ where $\sigma^2 = \frac{1}{11}\sum_{b'=9}^{19}(b' - b_{\text{avg}})^2$. This is since the value head computes values like score difference, that scale quadratically with board width. As before, scaling and subtracting σ^2 improves orthogonality.

Using such layers, a *global pooling bias structure* takes input tensors X (shape $b \times b \times c_X$) and G (shape $b \times b \times c_G$) and consists of:

- A batch normalization layer and ReLU activation applied to G (output shape $b \times b \times c_G$).
- A global pooling layer (output shape $3c_G$).
- A fully connected layer to c_X outputs (output shape c_X).
- Channelwise addition with X (output shape $b \times b \times c_X$).

A.3 Trunk

The *trunk* consists of:

- A 5x5 convolution of the binary spatial input tensor outputting c channels. In parallel, a fully connected linear layer on the overall game state input tensor outputting c channels, producing biases that are added channelwise to the result of the 5x5 convolution.
- A stack of n residual blocks. All but two or three of the blocks are ordinary pre-activation ResNet blocks, consisting of the following in order:
  - A batch-normalization layer.
  - A ReLU activation function.
  - A 3x3 convolution outputting c channels.
  - A batch-normalization layer.
  - A ReLU activation function.
  - A 3x3 convolution outputting c channels.
  - A skip connection adding the convolution result elementwise to the input to the block.
- The remaining two or three blocks use global pooling, consisting of the following in order:
  - A batch-normalization layer.
  - A ReLU activation function.
  - A 3x3 convolution outputting c channels.
  - A *global pooling bias structure* pooling the first c_{pool} channels to bias the other $c - c_{\text{pool}}$ channels.
  - A batch-normalization layer.
  - A ReLU activation function.

  - A 3x3 convolution outputting c channels.
  - A skip connection adding the convolution result elementwise to the input to the block.
- At the end of the trunk, a batch-normalization layer and one more ReLU activation function.

A.4 Policy Head

The *policy head* consists of:

- A 1x1 convolution outputting c_{head} channels ("P") and in parallel a 1x1 convolution outputting c_{head} channels ("G").
- A *global pooling bias structure* pooling the output of G to bias the output of P.
- A batch-normalization layer.
- A ReLU activation function.
- A 1x1 convolution with 2 channels, outputting two policy distributions in logits over moves on each of the locations of the board. The first channel is the predicted policy $\hat{\pi}$ for the current player. The second channel is the predicted policy $\hat{\pi}_{\text{opp}}$ for *the opposing player on the subsequent turn.*
- In parallel, a fully connected linear layer from the globally pooled values of G outputting 2 values, which are the logits for the two policy distributions for making the pass move for $\hat{\pi}$ and $\hat{\pi}_{\text{opp}}$, as the pass move is not associated with any board location.

A.5 Value Head

The *value head* consists of:

- A 1x1 convolution outputting c_{head} channels ("V").
- A *global pooling layer* of V outputting $3c_{\text{head}}$ values ("V_{pooled}").
- A game-outcome subhead consisting of:
  - A fully-connected layer from V_{pooled} including bias terms outputting c_{val} values.
  - A ReLU activation function.
  - A fully-connected layer from V_{pooled} including bias terms outputting 9 values.
    * The first 3 values are a distribution in logits whose softmax $\hat{z}$ predicts among the three possible game outcomes *win*, *loss*, and *no result* (the latter being possible under non-superko rulesets in case of long-cycles).
    * The fourth value is multiplied by 20 to produce a prediction $\hat{\mu}_s$ of the final score difference of the game in points[14].
    * The fifth value has a softplus activation applied and is then multiplied by 20 to produce an estimate $\hat{\sigma}_s$ of the standard deviation of the predicted final score difference in points.

    - The sixth through ninth values have a softplus activation applied are predictions $\hat{\text{rv}}_i$ of the expected variance in the MCTS root value for different numbers of playouts[15].
    - All predictions are from the perspective of the current player.

- An ownership subhead consisting of:
  - A 1x1 convolution of V outputting 1 channel.
  - A tanh activation function.
  - The result is a prediction $\hat{o}$ of the expected ownership of each location on the board, where 1 indicates ownership by the current player and -1 indicates ownership by the opponent.

- A final-score-distribution subhead consisting of:
  - A scaling component:
    - A fully-connected layer from V_{pooled} including bias terms outputting c_{val} values.
    - A ReLU activation function.
    - A fully-connected layer including bias terms outputting 1 value ("γ").
  - For each possible final score value s:

    $$s \in \{-S + 0.5, -S + 1.5, \ldots, S - 1.5, S - 0.5\}$$

    where S is a an upper bound for the plausible final score difference of any game[16], in parallel:
    - The $3c_{\text{head}}$ values from V_{pooled} are concatenated with two additional values:

      $$(0.05 * s, \text{Parity}(s) - 0.5)$$

      0.05 is an arbitrary reasonable scaling factor so that these values vary closer to unit scale. Parity(s) is the binary indicator of whether a score value is normally possible or not due to parity of the board size and komi[17].
    - A fully-connected layer (sharing weights across all s) from the $3c_{\text{head}} + 2$ values including bias terms outputting c_{val} values.
    - A ReLU activation function.
    - A fully-connected layer (sharing weights across all s) from V_{pooled} including bias terms, outputting 1 value.
  - The resulting $2S$ values multiplied by softplus(γ) are a distribution in logits whose softmax $\hat{p}_s$ predicts the final score difference of the game in points. All predictions are from the perspective of the current player.

[14] 20 was chosen as an arbitrary reasonable scaling factor so that on typical data the neural net would only need to output values around unit scale, rather than tens or hundreds.

[15] In training the weight on this head is negligibly small. It is included only to enable future research on whether MCTS can be improved by biasing search towards more "uncertain" subtrees.

[16] We use $S = 19 * 19 + 60$, since 19 is the largest standard board size, and the extra 60 conservatively allows for the possibility that the winning player wins all of the board *and* has a large number of points from *komi*.

[17] In Go, usually every point on the board is owned by one player or the other in a finished game, so the final score difference varies only in increments of 2 and half of values only rarely occur. Such a parity component is very hard for a neural net to learn on its own. But this feature is mostly for cosmetic purposes, omitting it should have little effect on overall strength).

A.6 Neural Net Parameters

Four different neural net sizes were used in our experiments. Table 5 summarizes the constants for each size.

Size	b6×c96	b10×c128	b15×c192	b20×c256
n	6	10	15	20
c	96	128	192	256
c_{pool}	32	32	64	64
c_{head}	32	32	32	48
c_{val}	48	64	80	96

Table 5: Architectural constants for various neural net sizes.

Appendix B Loss Function

The loss function used for neural net training in KataGo is the sum of:

- Game outcome value loss:
$$c_{\text{value}} \sum_{r \in \{\text{win,loss}\}} z(r) \log(\hat{z}(r))$$
where z is a one-hot encoding of whether the game was won or lost by the current player, $\hat{z}$ is the neural net's prediction of z, and $c_{\text{value}} = 1.5$.

- Policy loss:
$$- \sum_{m \in \text{moves}} \pi(m) \log(\hat{\pi}(m))$$
where π is the target policy distribution and $\hat{\pi}$ is the prediction of π.

- Opponent policy loss:
$$-w_{\text{opp}} \sum_{m \in \text{moves}} \pi_{\text{opp}}(m) \log(\hat{\pi}_{\text{opp}}(m))$$
where π_{opp} is the target opponent policy distribution, $\hat{\pi}_{\text{opp}}$ is the prediction of π_{opp}, and $w_{\text{opp}} = 0.15$.

- Ownership loss:
$$-w_o \sum_{l \in \text{board}} \sum_{p \in \text{players}} o(l,p) \log\left(\hat{o}(l,p)\right)$$
where $o(l,p) \in \{0, 0.5, 1\}$ indicates if l is finally owned by p, or is shared, $\hat{o}$ is the prediction of o, and $w_o = 1.5/b^2$ where $b \in [9, 19]$ is the board width.

- Score belief loss ("pdf"):
$$-w_{\text{spdf}} \sum_{x \in \text{possible scores}} p_s(x) \log(\hat{p}_s(x))$$
where p_s is a one-hot encoding of the final score difference, $\hat{p}_s$ is the prediction of p_s, and $w_{\text{spdf}} = 0.02$.

- Score belief loss ("cdf"):
$$w_{\text{scdf}} \sum_{x \in \text{possible scores}} \left(\sum_{y < x} p_s(y) - \hat{p}_s(y) \right)^2$$
where $w_{\text{scdf}} = 0.02$.

- Root variance loss:
$$w_{\text{rv}} \sum_{i=0}^{3} (\text{rv}_i(y) - \hat{\text{rv}}_i(y))^2$$
where rv_i are the recorded values of variance in the MCTS root value between 1 and $\{4, 16, 64, 256\}$ visits, $\hat{\text{rv}}_i$ are the neural net's predictions of these values, intended for future research, and $w_{\text{rv}} = 0.2$ (in practice, the variances are small and therefore this loss term is negligible).

- Score belief mean self-prediction:

$$-w_{\text{sbreg}}\text{Huber}(\hat{\mu}_s - \mu_s,\, \delta = 10.0)$$

where $w_{\text{sbreg}} = 0.004$ and

$$\mu_s = \sum_x x\hat{p}_s(x)$$

and $\text{Huber}(x, \delta)$ is the *Huber loss function* equal to the squared error loss $f(x) = 1/2x^2$ except that for $|x| > \delta$, instead $\text{Huber}(x, \delta) = f(\delta) + (|x| - \delta)\frac{df}{dx}(\delta)$. This avoids some cases of divergence in training due to large errors just after initialization.

Note that neural net is predicting itself - i.e. this is a regularization term for an otherwise unanchored output $\hat{\mu}_s$ to roughly equal to the mean score implied by the neural net's full score belief distribution. The neural net easily learns to make this output highly consistent with its own score belief[18].

- Score belief standard deviation self-prediction:

$$-w_{\text{sbreg}}\text{Huber}(\hat{\sigma}_s - \sigma_s,\, \delta = 10.0)$$

where

$$\sigma_s = \left(\sum_x (x - \mu)^2 \hat{p}_s(x) \right)^{1/2}$$

Similarly, the neural net is predicting itself - i.e. this is a regularization term for an otherwise unanchored output $\hat{\sigma}_s$ to roughly equal to the standard deviation of the neural net's full score belief distribution. The neural net easily learns to make this output highly consistent with its own score belief[18].

- Score belief scaling penalty:

$$w_{\text{scale}}\gamma^2$$

where γ is the activation strength of the internal scaling of the score belief and $w_{\text{scale}} = 0.0005$. This prevents some cases of training instability involving the multiplicative behavior of γ on the belief confidence where γ grows too large.

- L2 penalty:

$$c||\theta||^2$$

where θ are the model parameters and $c = 0.00003$, so as to bound the weight scale and ensure that the effective learning rate does not decay due to batch normalization.

[18]These are partly for implementation convenience. KataGo's play engine uses a separate GPU implementation so as to run independently of TensorFlow, and this allows us to avoid implementing the score belief head. Also for technical reasons relating to dynamic score utility and tree re-use, using only the first and second moments instead of the full distribution is convenient.

Appendix C Training Details

In total, KataGo's main run lasted for 19 days using 16 V100 GPUs for self-play for the first two days and increasing to 24 V100 GPUs afterwards, and 2 V100 GPUs for gating, one V100 GPU for neural net training, and additionally one V100 GPU for neural net training when running the next larger size concurrently on the same data. It generated about 241 million training samples across 4.2 million games, across four neural net sizes, as summarized in Tables 6 and 7.

Size	Days	Train Steps	Samples	Games
b6×c96	0.75	98M	23M	0.4M
b10×c128	1.75	209M	55M	1.0M
b15×c192	7.5	506M	140M	2.5M
b20×c256	19	954M	241M	4.2M

Table 6: Training time of the strongest neural net of each size in KataGo's main run. "Days" is the time of finishing a size and switching to the next larger size , "Train Steps" indicates cumulative gradient steps taken measured in samples, "Samples" and "Games" indicate cumulative self-play data samples and games generated.

Size	Elo vs LZ/ELF	Rough strength
b6×c96	-1276	Strong/Top Amateur
b10×c128	-850	Strong Professional
b15×c192	-329	Superhuman
b20×c256	+76	Superhuman

Table 7: Approximate strength of the strongest neural net of each size in KataGo's main run at a search tree node cap of 1600. Elo values are versus a mix of various Leela Zero versions and ELF, anchored so that ELF is about Elo 0.

Training used a batch size of 256 and a per-sample learning rate of $6 * 10^{-5}$, or a per-batch learning rate of $256 * 6 * 10^{-5}$. However, the learning rate was lowered by a factor of 3 for the first five million samples of training steps for each neural net to reduce early training instability, as well as lowered by a factor of 10 for the final b20×c256 net after 17.5 days of training for final tuning.

Training samples were drawn uniformly from a moving window of the most recent N_{window} samples, where

$$N_{\text{window}} = c\left(1 + \beta\frac{(N_{\text{total}}/c)^{\alpha} - 1}{\alpha}\right)$$

where N_{total} is the total number of training samples generated in the run so far, $c = 250{,}000$ and $\alpha = 0.75$ and $\beta = 0.4$. Though appearing complex, this is simply the sublinear curve $f(n) = n^{\alpha}$ but rescaled so that $f(c) = c$ and $f'(c) = \beta$.

Appendix D Game Randomization and Termination

KataGo randomizes in a variety of ways to ensure diverse training data so as to generalize across a wide range of rulesets, board sizes, and extreme match conditions, including handicap games and positions arising from mistakes or alternative moves in human games that would not occur in self-play.

- Games are randomized uniformly between positional versus situational superko rules, and between suicide moves allowed versus disallowed.
- Games are randomized in board size, with 37.5% of games on 19x19 and increasing in KataGo's main run to 50% of games after two days of training. The remaining games are triangularly distributed from 9x9 to 18x18, with frequency proportional to $1, 2, \ldots, 10$.
- Rather than using only a standard komi of 7.5, komi is randomized by drawing from a normal distribution with mean 7 and standard deviation 1, truncated to 3 standard deviations and rounding to the nearest integer or half-integer. However, 5% of the time, a standard deviation of 10 is used instead, to give experience with highly unusual values of komi.
- To enable experience with handicap game positions, 5% of games are played as handicap games, where Black gets a random number of additional free moves at the start of the game, chosen randomly using the raw policy probabilities. Of those games, 90% adjust komi to compensate White for Black's advantage, by iteratively querying the neural net for the expected final score difference, and then adding that amount to komi. The maximum number of free Black moves is 0 (no handicap) for board sizes 9 and 10, 1 for board sizes 11 to 14, 2 for board sizes 15 to 18, and 3 for board size 19.
- To initialize each game and ensure opening variety, the first r moves of a game are played randomly directly proportionally to the raw policy distribution of the net, where r is drawn from an exponential distribution with mean $0.04 * b^2$. where b is the width of the board, and during the game, moves are selected proportionally to the target-pruned MCTS playout distribution raised to the power of $1/T$ where T is a temperature constant. T begins at 0.8 and decays smoothly to 0.2, with a halflife in turns equal to the width of the board b.
- In 2.5% of positions, the game is branched to try an alternative move drawn randomly from the raw policy of the net 70% of the time with temperature 1, 25% of the time with temperature 2, and otherwise with temperature infinity. A full search is performed to produce a policy training sample (the $MCTS$ search winrate is used for the game outcome target and the score and ownership targets are left unconstrained). This ensures that there is a small percentage of training data on how to respond to or refute moves that a full search might not play. Recursively, a random quarter of these branches are continued for an additional move, otherwise they are terminated.
- In 5% of games, the game is branched after the first r turns where r is drawn from an exponential distribution with mean $0.025 * b^2$. Between 3 and 10 moves are chosen uniformly at random, each given a single neural net evaluation, and the best one is played. Komi is adjusted to be fair as in a handicap game. The game is then played to completion as normal. This ensures that there is always a small percentage of games with unusual openings or joseki, for example openings involving the 5-4 points or center-based openings.

Additionally, unlike in AlphaZero, games are played to completion without resignation. However, during self-play if for 5 consecutive turns, the MCTS winrate estimate p for the losing side has been less than 5%, then to finish the game faster the number of visits is capped to $\lambda n + (1 - \lambda)N$ where n and N are the small and large limits used in playout cap randomization and $\lambda = p/0.05$ is the proportion of the way that p is from 5% to 0%. Additionally, training samples are recorded with only $0.1 + 0.9\,\lambda$ probability, stochastically downweighting training on positions where AlphaZero would have resigned.

Appendix E Gating

Similar to AlphaGoZero, candidate neural nets must pass a *gating* test to become the new net for self-play. Gating in KataGo is fairly lightweight - candidates need only win at least 100 out of 200 games against the current self-play neural net. Gating games use a fixed cap of 300 search tree nodes (increasing in KataGo's main run to 400 after 2 days), with the following parameter changes to minimize noise and maximize performance:

- The rules and board size are still randomized but komi is not randomized and is fixed at 7.5.
- Handicap games and branching are disabled.
- From the first turn, moves are played using full search rather than using the raw policy to play some of the first moves.
- The temperature T for selecting a move based on the MCTS playout distribution starts at 0.5 instead of 0.8.
- Dirichlet noise and forced playouts and visit cap oscillation are disabled, tree reuse is enabled.
- The root uses $c_{\text{FPU}} = 0.2$ just the same as the rest of the search tree instead of $c_{\text{FPU}} = 0.0$.
- Resignation is enabled, occurring if both sides agree that for the last 5 turns, the worst MCTS winrate estimate p for the losing side has on each turn been less than 5%.

Appendix F Score Maximization

Unlike most other Go bots learning from self-play, KataGo puts nonzero utility on maximizing (a dynamic monotone function of) the score difference, to improve use for human game analysis and handicap game play.

Letting x be the final score difference of a game, in addition to the utility for winning/losing:

$$u_{\text{win}}(x) = \text{sign}(x) \in \{-1, 1\}$$

We also define the score utility:

$$u_{\text{score}}(x) = c_{\text{score}} f\left(\frac{x - x_0}{b}\right)$$

where c_{score} is a parameter controlling the relative importance of score, x_0 is a parameter for centering the utility curve, $b \in [9, 19]$ is the width of the board and $f : \mathbb{R} \to (-1, 1)$ is the function:

$$f(x) = \frac{2}{\pi} \arctan(x)$$

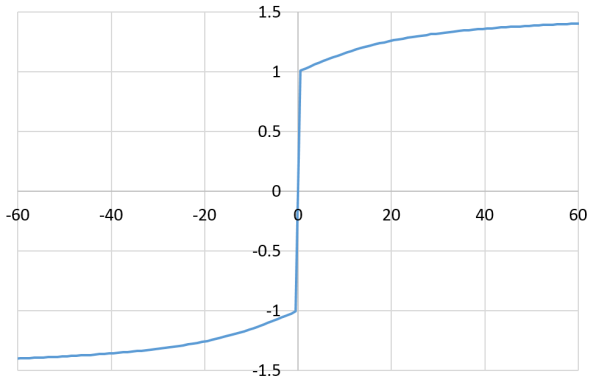

Figure 8: Total utility as a function of score difference, when $x_0 = 0$ and $b = 19$ and $c_{\text{score}} = 0.5$.

At the start of each search, the utility is re-centered by setting x_0 to the mean $\hat{\mu}_s$ of the neural net's predicted score distribution at the root node. The search proceeds with the aim to maximize the sum of u_{win} and u_{score} instead of only u_{win}. Estimates of u_{win} are obtained using the game outcome value prediction of the net as usual, and estimates of u_{score} are obtained by querying the neural net for the mean and variance $\hat{\mu}_s$ and $\hat{\sigma}_s^2$ of its predicted score distribution, and computing:

$$E(u_{\text{score}}) \approx \int_{-\infty}^{\infty} u_{\text{score}}(x) N(x, \hat{\mu}_s, \hat{\sigma}_s^2) dx$$

where the integral on the right is estimated quickly by interpolation in a precomputed lookup table.

Since similar to a sigmoid f saturates far from 0, this provides an incentive for improving the score in simple and likely ways near x_0 without awarding overly large amounts of expected utility for pursuing unlikely but large gains in score or shying away from unlikely but large losses in score. For KataGo's main run, c_{score} was initialized to 0.5, then adjusted 0.4 after the first two days of training.