Accelerating Self-Play Learning in Go

David J. Wu*
Jane Street Group

March 4, 2019

Abstract

By introducing several new Go-specific and non-Go-specific techniques along with other tuning, we accelerate self-play learning in Go. Like AlphaZero and Leela Zero, a popular open-source distributed project based on AlphaZero, our bot KataGo only learns from neural net Monte-Carlo tree-search self-play. With our techniques, in only a week with several dozen GPUs it achieves a likely strong pro or perhaps just-super-human level of strength. Compared to Leela Zero, we estimate a roughly 5x reduction in self-play computation required to achieve that level of strength, as well as a 30x to 100x reduction for reaching moderate to strong amateur levels. Although we so far have not tested in longer runs, we believe that our techniques hold promise for future research.

1 Introduction

In 2017, DeepMind's AlphaGoZero demonstrated in a landmark result that it was possible to achieve superhuman performance in the game of Go starting from random play and learning only via reinforcement learning of a neural network using self-play bootstrapping from Monte-Carlo tree search[9]. Moreover, AlphaGoZero used only fairly minimal game-specific tuning. Subsequently, DeepMind's AlphaZero demonstrated that the same methods could also be used to train extremely strong agents in Chess and Shogi. However, the amount of computation required was large, with DeepMind's main reported run taking about 41 TPU-years in total parallelized over 5000 TPUs [8]. The significant cost of reproducing this work has slowed research, putting it out of reach for all but major companies such as Facebook[11], as well as a few online massively distributed computation projects, notably Leela Zero for Go[14], and Leela Chess Zero for Chess[17].

In this paper, we introduce several new techniques, while also reviving some ideas from pre-AlphaZero research in computer Go and newly applying them to the AlphaZero process. Combined with minor domain-specific heuristic optimizations and overall tuning, these ideas greatly improve the efficiency of self-play learning. Still starting only from random play, training on merely about 30 GPUs for a week our bot KataGo reaches just below the strength of Leela Zero as of Leela Zero's 15-block neural net "LZ130", a likely professional or possibly just-superhuman level when run on strong consumer hardware. Based on estimates of the number of neural net queries involved,

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

we achieve approximately a 5x reduction in the self-play computation required to reach this level compared to Leela Zero, and earlier in training a 30x to 100x reduction for reaching moderate to strong amateur levels of strength.

In doing so, we make several contributions:

Firstly, from a broader machine-learning perspective: our results give evidence that there is a significant efficiency gap between the somewhat more general methods of AlphaZero and what is theoretically possible from self-play learning in Go. This suggests that there may still be significant room for improvement in those general methods. KataGo, while leveraging details of the domain, still learns entirely from MCTS-bootstrapped self-play without using any pre-existing human-generated data or expert strategic knowledge.

Additionally, many of the ideas we present are non-Go-specific and might be applied to AlphaZero-like learning in other games, or perhaps to other tasks entirely. These include our technique of *playout or visit cap oscillation* to improve the quantity of data generated by MCTS, an idea from older computer Go literature to add *auxiliary policy targets* from future actions for additional regularization, which might be applicable to other sequential action environments, or our observation that a *global-pooling* mechanism can add new representational power to a convolutional net, in agreement with other research on global context in image-related tasks. We also present a very simple trick for *sharing learned weights across multiple board sizes* which might be useful for convolutional networks with inputs of variable size in other contexts.

We also hope this work serves as a case study on how when learning is highly data-constrained, one might enrich the data. Something as simple as *adding new auxiliary outputs and training targets* to a neural net can improve the quality of predictions, even if those outputs are completely unused outside of training. And we find there are often many tradeoffs between data quantity and quality in different dimensions in reinforcement learning that can be tuned.

Lastly, for the computer-Go community: we hope that KataGo is of interest in that it shows how one might train a neural net to directly predict the final score difference of a Go game rather than merely the winner, to play well under a wide variety of komi rather than only a single fixed value[1], and to handle multiple rulesets and board sizes simultaneously. To our knowledge, most of these features are not yet present in most strong modern open-source Go programs. KataGo is open-source on GitHub and along with this paper demonstrates how they can be implemented[2].

A note of caution is warranted, however. KataGo has *not* yet had the resources to test with training runs nearly as long and large as those of larger research efforts. While KataGo learns much faster up to just-super-human levels, further beyond that it is possible that some of the techniques presented here could need to be modified or annealed away in the late stages of a longer run when fine-tuning for final strength. Nonetheless, we believe these techniques hold promise, and it is our hope that these ideas can serve as the seed for future research and more rigorous testing and experimentation.

[1] In Go, *komi* is the number of points given to the second player as compensation. It can also be varied to equalize winning chances between players of different skill levels.

[2] The code and links to download the neural nets and training data from KataGo's main run can be found at: `https://github.com/lightvector/KataGo` . For any interested enthusiasts, with our code just three or four strong GPUs is sufficient for anyone to train a bot from random all the way to amateur-dan-strength on the full 19x19 board in only a few days!

2 Related Work

Aside from AlphaZero, in recent years a few other notable projects have emerged in its footsteps. These initially included the open-source Leela Zero and Leela Chess Zero distributed projects, producing very strong programs in Go and Chess[14, 17]. Others include the MiniGo project, establishing basic technical details of the AlphaZero process not obvious from high-level descriptions in the original AlphaZero papers[21], as well as the SAI project, which has explored self-play learning on smaller boards as well as new architectures for handling komi and multi-valued predictions[6]. And most recently, a paper and new release by Facebook AI Research of ELF OpenGo has also contributed to basic understanding of AlphaZero in Go[10].

Our work in KataGo borrows a number of minor techniques and ideas from these other projects, particularly Leela Zero, and where appropriate we mention this and/or contrast the differences.

KataGo also draws from work prior to AlphaZero on the value of game-specific features[2] and of auxiliary prediction targets in the context of supervised learning[12, 13], and extends these ideas to self-play reinforcement learning. While AlphaZero and reproductions like ELF OpenGo and others have shown that such ideas are not strictly necessary to achieve superhuman strength given enough sheer computing resources, we provide evidence that many of these earlier ideas still provide benefits and should not be lightly discarded. In combination with future improvements and discoveries, they could potentially bring the otherwise prohibitively-expensive AlphaZero process in games as massive as Go down to a cost accessible to smaller research groups and institutions.

3 Overview

At a high level, KataGo's overall architecture resembles the AlphaZero architecture. It is based around iteratively improving a neural net that outputs both a policy distribution over legal moves and a prediction of the game outcome. The neural net is used in a Monte-Carlo tree search which generates data via self-play that is used to further train the neural net. Periodic snapshots of the neural net in training are taken, and subject to passing a gating mechanism to ensure the quality of the new net, it replaces the net to be used for subsequent self-play. In KataGo, all these steps run continuously and asynchronously.

We give an overview of the components of the architecture in three major sections:

- Neural Net Training: its architecture, inputs, outputs, loss function, and hyperparameters.
- Search and Target Generation: the tree search, exploration formula, utility function, and generation of the policy target for training from the search result.
- Self-play: the initialization of games, variation of search parameters, branching of game positions, and optimizations to playing and terminating those games.

In each, we will highlight the major and minor innovations and/or differences from AlphaZero, as well as the differences from other prior work, most notably that of Leela Zero, the most popular open-source Go project modeled on AlphaZero. After that, we will present data from KataGo's main run and other experimental results from a variety of ablation studies.

4 Neural Net Training

KataGo's neural net is a convolutional residual neural net that uses a *preactivation* architecture[3]. Like AlphaZero's, it is composed of a trunk of residual blocks, along with several output heads that in parallel transform the final layer of the trunk to produce various outputs. These include a *policy* and a *game outcome value* prediction that are trained towards targets generated from self-play.

We will focus here only on the major differences that we believe contribute to KataGo's improved learning efficiency. For a full description of our neural net architecture see Appendix A.

4.1 Auxiliary Ownership and Score Targets

One of the improvements in KataGo's neural net training over AlphaZero and Leela Zero in Go is from the use of auxiliary ownership and score prediction targets. The use of such targets was earlier explored in work by Ti-Rong Wu et al. in the context of supervised learning, and there they found that including these extra targets reduced the mean squared error on game result prediction by a neural net and mildly improved the strength of their overall bot, CGI[13].

We find in KataGo that such targets also greatly improve the neural net's ability to learn from limited data in the reinforcement learning context of self-play training as well. The resulting improvement demonstrates the heuristic that *when data is noisy or scarce, it can be beneficial to add more data-rich auxiliary targets.*

In the AlphaZero process, noise and data-scarcity are particularly severe for the game outcome prediction, as compared to the other important prediction by the neural net, the policy. Whereas the policy target receives one sample per move of a game, each of which is a rich distribution over legal moves[3], the game outcome target receives only one independent sample per entire game and that sample is merely a single noisy binary win or loss.

This makes the game outcome prediction particularly prone to fitting poorly, greatly benefiting from regularization from additional targets. In fact, Silver et al. found in AlphaGoZero that forcing the same neural net to predict both the policy and the game outcome value greatly improved the quality of the value prediction over training on value alone[9]. So, even if one cared only about the value prediction and not at all about the policy, one would still want to predict the policy purely to regularize the value head!

Since the final winner in Go is the player who owns more of the board (plus komi) and the game terminates not just with a win/loss outcome but a numerical score difference[4], predicting these as well should provide far better regularization than the policy alone. So following the same motivation as CGI, we introduce into KataGo the auxiliary targets of:

- Ownership - For each point on the board, predict the expectation of the final owner of that board point, equal to 1 if the final owner is the current player and -1 if it is the opponent (and 0 in the rare case of neither).
- Score Belief - For each possible final score difference, predict the probability that the game ends with exactly that score difference.

[3] In the AlphaZero process, the policy target is the full MCTS playout distribution, rather than merely a one-hot encoding of what move was actually played.

We then augment the loss function as follows. We begin first with the basic loss function:

$$L = c_{\text{value}} \sum_{r \in \{\text{win,loss}\}} z(r) \log(\hat{z}(r)) - \sum_{m \in \text{moves}} \pi(m) \log(\hat{\pi}(m)) + c_{L2} ||\theta||^2$$

where z is a one-hot encoding of whether the game was won or lost by the current player[5], $\hat{z}$ is the neural net's prediction of z, π is the target policy distribution, $\hat{\pi}$ is the predicted policy distribution, c_{L2} is a standard L2 penalty coefficient on the model parameters θ, and $c_{\text{value}} = 1.5$ is a scaling constant for the game outcome value target relative to the policy target[6].

We then add three additional terms:

- Ownership loss:

$$-w_o \sum_{l \in \text{board}} \frac{1+o(l)}{2} \log\left(\frac{1+\hat{o}(l)}{2}\right) + \frac{1-o(l)}{2} \log\left(\frac{1-\hat{o}(l)}{2}\right)$$

  where $o(l) \in \{-1, 0, 1\}$ is the actual final owner of board location l, $\hat{o}(l) \in [-1, 1]$ is the neural net's prediction, and w_o is a coefficient weighting this objective.

- Score belief loss ("pdf"):

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

  where $p_s(x)$ is a one-hot encoding of whether the final score difference is exactly x, and $\hat{p}_s(x)$ is the neural net's predicted probability that the final score difference is exactly x, and w_{spdf} is a coefficient weighting this objective.

- 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_{scdf} is a coefficient weighting this objective. Whereas the "pdf" loss rewards guessing the score exactly, this "cdf" loss is smoother, encouraging the bulk mass of the predicted distribution to be near the final score.

Additionally as a technical detail, the architecture of KataGo's score belief distribution head internally contains a scaling component that unchecked can sometimes result in training instability, so we add an additional regularization penalty $w_{\text{scale}}\gamma^2$ where γ is the internal activation value of the scaling component.

[4] In Go, every point occupied or surrounded by a player at the end of the game scores 1 point. The second player also receives a *komi* of typically 7.5 points for going second, and then the player who has more points is the winner.

[5] As a minor difference from AlphaZero, KataGo uses a cross-entropy loss for the game outcome instead of squared error, treating it as a two-category classification. This allows extension to Japanese Go rules, under which a game can end in a third category *no-result* since the Japanese rules do not prohibit long cycles.

[6] $c_{\text{value}} = 1.5$ was chosen so that that typical gradients from the cross-entropy value loss would be roughly comparable to those of a squared error.

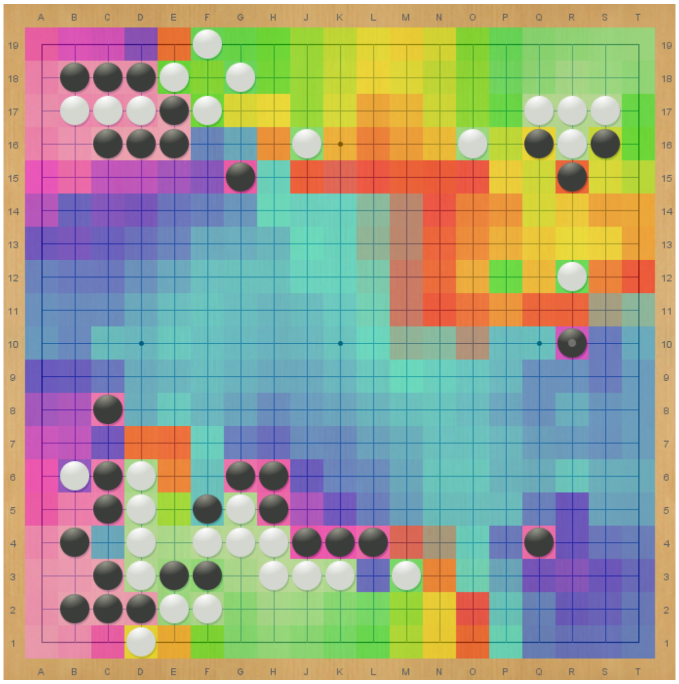

Figure 1: Visualization of predicted ownership of areas of an example board position by KataGo's neural net. Red through green increasingly indicate white ownership. Cyan through magenta increasingly indicate black ownership.

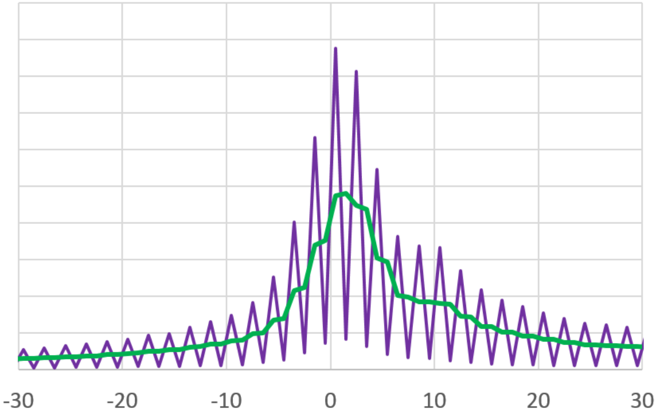

Figure 2: Distribution of predicted final score difference in points by the neural net in the same board position as in Figure 1 (purple), along with a smoothed version (green). White is predicted to win about 65% of the time, the mean score difference is about +6.

We show in our experimental results in section 7.2 that these auxiliary targets greatly improve the efficiency of learning, even up through strong amateur strength. This might be surprising in some ways. Presumably past beginner-level strength the neural net must have already "discovered" that the game outcome is highly correlated with the sum of control of regions of the board, so why would predicting ownership continue to provide a benefit beyond the very early stages of training?

Although with no formal evidence, we offer one intuition: consider the task of learning from a game lost due to evaluating a certain pattern of stones as safe when in fact the opponent managed to capture them. Even at very strong levels, games can hinge on the uncertain life or death of groups of stones. With only the final binary win/loss result, the neural net must "guess" at what aspect of the board position caused the loss, and may require more examples to infer the correct credit assignment. By contrast, with an ownership target the neural net receives direct feedback on exactly which stones it would have predicted as safe instead were captured, and therefore should require fewer samples to generalize.

See Appendix A for the architecture of the output heads for these targets, as well as Appendix B for the values of the various weighting constants.

4.2 Auxiliary Policy Targets

As another difference from AlphaZero and other bots, in KataGo we also add an auxiliary policy-related target predicting the opponent's reply on the following turn. This idea is not entirely new either, having been found by Tian and Zhu in Facebook AI Research's bot Darkforest to provide benefits in the context of supervised move prediction [12], but as far as we know, KataGo is the first to apply it to the AlphaZero process.

In KataGo, this is done simply with another output and new term in 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 training distribution that will be recorded for the turn *after* the current turn, $\hat{\pi}_{\text{opp}}$ is the neural net's prediction of that target, and $w_{\text{opp}} = 0.15$ weights this target only a fraction as much as the actual policy.

The neural net prediction $\hat{\pi}_{\text{opp}}$ is completely unused thereafter, but in informal early testing, we found that although the benefit was very small, adding a reasonable small weight on this target slightly improved the speed of learning. Unlike Tian and Zhu in Darkforest, we did not investigate predicting further additional moves. This might be a direction for future improvements, although there will likely be diminishing returns.

Notably, this additional policy target relies on none of the properties of Go and *potentially could apply to any sequential action environment.* In any specific environment, it is of course not guaranteed to provide a benefit, but wherever generation of data is the bottleneck, as in the AlphaZero process, this target could potentially be an easy and cheap way to achieve slightly better regularization.

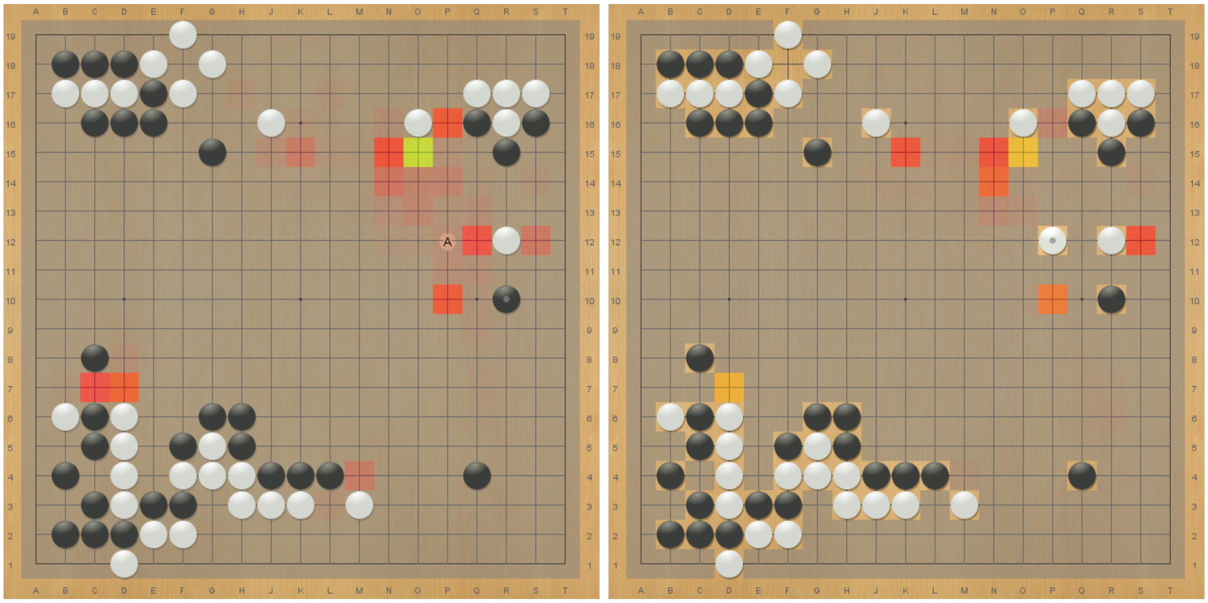

Figure 3: Visualization of auxiliary policy target prediction by the neural net. Red to yellow indicates increasingly probable moves. Left: prediction of opponent's replies to our likely moves. Right: actual policy prediction on the opponent's turn if our most likely move occurs.

4.3 Sharing Weights Across Board Sizes

KataGo also uses a simple new method of sharing the same neural network weights across multiple board sizes, training jointly on all sizes. We see no reason why this method cannot also be applied to other games with variable board sizes, or perhaps even more broadly to some image processing tasks for training on multiple image sizes without cropping!

KataGo trains on a mixture of games with random board sizes ranging from 9x9 to 19x19.[7]Positions from all sizes are shuffled together into the same training batches, training on all sizes jointly.

Mechanically, the obvious way to do this would be to embed the representation for smaller boards into larger tensors, padding the remainder with zeros. For example, embedding a 9x9 board into the upper-left 9x9 square of a 19x19 tensor. However, during convolution, non-zero values will be computed in the region within the 19x19 tensor but outside the 9x9 board. Then, subsequent convolutions near the lower-right of the 9x9 board will observe those non-zero values rather than the zero-padding they would observe for a properly-sized tensor.

We solve this by including with each training sample a *binary mask channel* indicating all on-board locations. Then:

- Between every pair of convolutions and/or other affected operations, we apply this mask, ensuring that the next convolution receives only zeros for off-board locations. This is a cheap pointwise multiplication, which can easily be fused with adjacent GPU kernels.

[7] Since smaller boards require less training data, KataGo uses an asymmetric weighting, randomly choosing between sizes $\{9, 10, 11, \ldots, 18, 19\}$ with probabilities proportional to $\{1, 2, 3, \ldots, 10, 11\}$. But since 19 is by far the most important size for human play, we then triple the weight on size 19 up to 33, annealing further to 55 and 110 when we anneal upward the number of playouts in self-play, as described in section 6.1.

- Similar masking is applied to spatial outputs - for example, the ownership prediction.
- In any spatial mean pooling operations, rather than summing tensor entries and dividing by the dimensions of the tensor, we sum masked entries and divide by the number of on-board locations, i.e. the number of ones present in the mask.

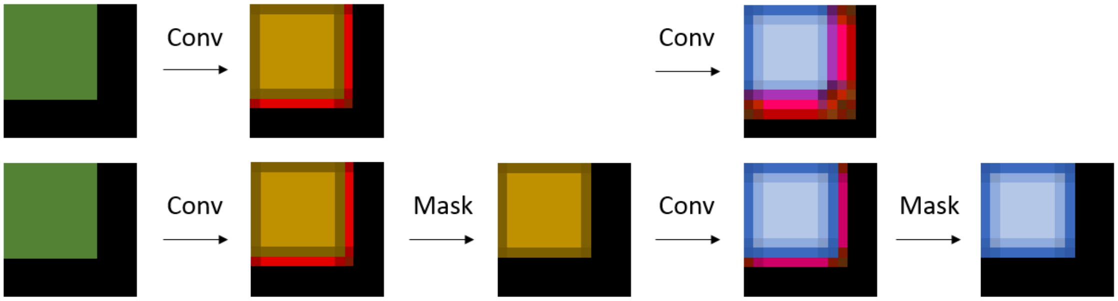

Figure 4: Convolution produces values within oversized tensor area. Top: The next convolution corrupts the final blue square. Bottom: Masking ensures a final result independent of tensor size.

The result is a single set of neural net weights that at inference time can be used with varying board sizes. Moreover, at inference time, if the neural net is playing on only a single board size at a time rather than multiple sizes in parallel, batches will consist of only a single board size. Then, one can just size the tensors to match the board size and leave no off-board space, and then all masking steps can be omitted, making this trick *zero-cost* at inference time!

Given the simplicity of the method, it would not surprise us if it were already used elsewhere. But to our knowledge, prior work neural net research in Go has only used fixed hardcoded board sizes, training separate neural nets per board size[8]. Looking more broadly at the image deep-learning literature, we have also not found mention of masking-based techniques like this as an option for handling variable image sizes, although given the vastness of the literature it would have been very easy to miss.

Our experiments in section 7.2 suggest that later in training, training on multiple board sizes does have a cost in the final 19x19 strength of the bot despite slightly accelerating learning early on. The early faster learning is presumably due to generalization of patterns and tactics from the smaller boards. As discussed by Morandin et al. in the SAI project, learning is much faster on small boards[6], and it is not surprising that fast small-board learning might generalize enough to drive faster early large-board learning.

But later, it is not surprising that playing well on small boards might detract from large-board play, likely due to limited neural net capacity. Although KataGo shows that this is no impediment to reaching pro-level and beyond with just a single set of weights, if strictly optimizing for maximal strength one might imagine separate training runs for specific important sizes (e.g. 19x19, 13x13), along with a more general run using our trick with shared weights and masking to simultaneously handle all other sizes (15x15, 17x17, even non-square boards, etc).

[8] A bot "Golaxy" developed by a Chinese research group has been seen to run on multiple board sizes, but we are not aware of anywhere they have published their methods.

4.4 Global Pooling

Another innovation in KataGo's neural net over AlphaZero and Leela Zero is the use of *global pooling* between convolutional layers, where at certain points a special layer aggregates per-channel information to be rebroadcast across the entire spatial extent of the board. Global pooling gives a way for the convolutional layers of the neural net to condition on global context, something that would otherwise be hard or impossible with the limited perceptual window of convolution alone.

In KataGo, given a set of C channels, a *global pooling layer* computes:

- The mean of each channel.
- The mean of each channel multiplied by $\frac{1}{10}(B - B_{\text{mid}})$
- The maximum of each channel.

where $B \in [B_{\text{min}}, B_{\text{max}}] = [9, 19]$ is the length of the board and $B_{\text{mid}} = \frac{1}{2}(B_{\text{min}} + B_{\text{max}})$.

This produces a total of $3C$ pooled values. The reason for having both mean values and mean values scaled with board length is to allow the neural net flexibility to choose how the pooling for each channel should vary as a function of the size of the board. Subtracting B_{mid} improves orthogonality, and $\frac{1}{10}$ is an arbitrary reasonable scaling constant so that resulting values remain near unit scale.

Global pooling layers are then used in a *global pooling bias structure* consisting of:

- Input tensors X (shape $B \times B \times C_X$) and G (shape $B \times B \times C_G$), followed by:
- A batch normalization layer and ReLu activation applied to G (output shape $B \times B \times C_G$).
- A global pooling layer on the result (output shape $3C_G$).
- Multiplication by a matrix of size $3C_G \times C_X$ (output shape C_X).
- Channelwise addition with X, treating the C_X different values as per-channel bias terms (output shape $B \times B \times C_X$).

This structure is used for the first convolutional layer of two of the residual blocks in KataGo's neural net, and for the first convolution layer in the policy head. It is also used in the value head with a slight modification to allow mean pooled values to also scale quadratically with board length, for the score-related value outputs.

In section 7.2 our experiments show that global pooling provides an improvement in the later stages of training. This accords with our prior research in supervised move prediction showing neural nets with global pooling can handle global tactics like "ko" even when their convolutional receptive field alone is too small[19]. They also automatically discover other global notions such as game phase (e.g. opening vs midgame vs endgame).

More broadly, global context can be valuable for games even without explicit global interactions. For example, in a wide variety of strategy games, expert human players alter their local move preferences when winning to favor options that "simplify" the position, whereas when losing they

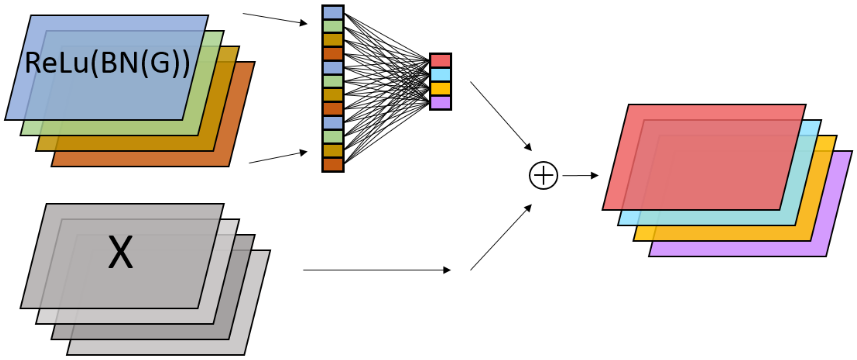

Figure 5: Global pooling bias structure, globally aggregating values of one set of channels to bias another set of channels.

seek "complication". Similar emergent behavior can be observed in computer players as well. Global pooling adds representational power to convolutional nets to express these ideas.

The idea of leveraging global context is by no means novel to KataGo, or even to game-playing. In image processing for example, within the last year and a half, Hu et al. have introduced a "Squeeze-and-Excitation" architecture to achieve new results in image classification[4]. Although the details are different, the fundamental concept is the same - giving the power to convolutional layers to condition on global context. This is a capacity missing from the AlphaZero architecture.

The Squeeze-Excite architecture is in fact now being tested by other AlphaZero-related projects[18, 22], and we look forward to trying it ourselves in future research. It is possible that somewhat more-flexible Squeeze-Excite concept may make our method of global pooling redundant, or may lead to even larger and more robust improvements than the one we observed.

4.5 Other Neural Net Training Differences

In addition to the major areas above, KataGo's neural net architecture and training also differ from AlphaZero's in a number of other minor ways.

4.5.1 Input Features

Rather than only using a raw representation of the Go board as AlphaZero does, KataGo includes a few higher-level features. The input to the neural net consists of two tensors, one of them a $19 \times 19 \times 22$ tensor of 22 binary spatial features, and the other a one-dimensional tensor with 14 global floating point values indicating properties of the game state not tied to any specific board location. See Tables 1 and 2 for the input features used.

Since the input features include parameters for the game rules, the neural net can simultaneously learn to handle multiple different rulesets. During self-play, as explained in Appendix C, these rules and komi are randomized. The result is that unlike AlphaZero or Leela Zero, KataGo can play well

[9] In Go, usually every board point is owned by one player or the other in a finished game, so the final score difference varies in increments of 2. Therefore usually only every second point of komi "matters" depending on whether the board size is even or odd. Such a parity component is *extremely* hard for a neural net to learn on its own.

# Channels	Binary Spatial Feature
1	Location is on board
2	Location has {own,opponent} stone
3	Location has chain with {1,2,3} liberties
1	Illegal move due to ko/superko
5	Previous move {1,2,3,4,5} location
3	Stone in or one move from inescapable atari {0,1,2} turns ago ("ladder")
1	Move puts opponent in inescapable atari ("ladder")
2	Pass-alive {own,opponent}, or area surrounded by pass-alive {own,opponent} stones
4	Features used for Japanese rules only (unused in this paper)

Table 1: KataGo binary spatial input features. A *pass-alive* stone is one that cannot be captured by the opponent even if the opponent were to get unboundedly many consecutive turns.

# Channels	Global Feature
5	Previous move {1,2,3,4,5} was a pass
1	Komi / 15.0 (from current player's perspective)
1	Simple ko rules (1.0) vs superko rules (0.0)
1	Simple ko (0.0) vs positional superko (0.5) vs situational superko (-0.5)
1	Suicide allowed (1.0) vs not allowed (0.0)
1	Komi + board size parity[9]
4	Features used for Japanese rules only (unused in this paper)

Table 2: KataGo global input features.

under multiple possible rulesets for Go, including nonstandard values of komi. Additionally, for each $i \in \{0, 1, 2, 3, 4\}$ on each training sample independently with a probability of 2% we truncate the history at i moves. This ensures that the neural net will behave reasonably in mid-game positions where history is not known or not available, such as when the position was human-constructed for study rather than the result of earlier play.

4.6 Progressive Neural Net Sizing

KataGo also employs the technique of *progressive neural net sizing* used by Leela Zero to speed up the early stages of learning, when large and expensive neural nets are not necessary for improvement. Training begins with a 6-block 96-channel net, then progresses to a 10-block 128-channel net, then progresses to a 15-block 192-channel net. The last matches the size of one of Leela Zero's still most-popular neural net sizes, a size capable of reaching super-human playing strength[10].

We make no attempt to transfer learned parameters between neural nets during size increases. Since training is very cheap compared to self-play, we simply choose a point to begin training the next larger neural net in parallel on the same data and then switch to the larger net once it begins to overtake the smaller net in predictive accuracy. See Table 3 for the schedule for switching.

[10] The next reasonable step would be 20 blocks and 256 channels, matching a basic neural net size for AlphaGoZero and/or AlphaZero, but we have not yet been able to test a run that progresses that far.

4.7 Other Training Details

Like AlphaZero and Leela Zero, KataGo trains its neural net via batched stochastic gradient descent with momentum. Momentum decay is set to 0.9, and for the runs in this paper, each size of neural net was trained with a batch size of 256 samples using a *per-sample* learning rate that varied according to the schedule:

$$\alpha_0(\lambda T + 1)^{-4/3}$$

where α_0 is the initial learning rate, λ controls the time-scale on which the learning rate decays, and T is the number of training steps performed with that particular neural net, measured in training samples (i.e. batches × 256). For all experiments in this paper, $\alpha_0 = $ 6e-5, and λ was set to $10^{-6} \times \{0.1, 0.075, 0.05\}$ respectively for the three neural net sizes used.

Neural Net	Began when $T_{\text{prev}} \approx$	Overtook when $T_{\text{prev}} \approx$	Overtook when new $T \approx$
6b×96c	-	-	0
10b×128c	70M	100M	20M
15b×192c	25M	100M	40M

Table 3: Approximate schedule for each neural net size during KataGo's main run. T_{prev} is the training steps for the previous net when this net began training or overtook it.

KataGo also uses a form of *stochastic weight averaging*[5]. Every approximately 250,000 training samples, a snapshot of the trained weights is taken, and every approximately 1 million training samples, a new candidate neural net is produced whose weights are an exponential moving average of snapshots with decay = 0.75 (i.e. averaging 4 snapshots of lookback). Based on a gating process similar to that of AlphaZero's, described in section 6.2.4, the candidate may become the new net used for self-play.

The data for training consists of batches drawn from a uniformly random permutation from a moving window of the last N_{window} samples of data. Unlike in AlphaZero or Leela Zero, N_{window} is not constant, but rather grows as more training samples are generated. The window initially consists of $c = 250{,}000$ samples of data resulting from self-play where a random number generator is used in place of a neural net, and then grows as:

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

where N_{total} is the total number of training samples[11]. Although appearing complicated, this is simply the polynomial curve:

$$f(n) = n^{\alpha}$$

except scaled by c and stretched so that the initial growth rate of the window is β increase in window size per sample generated. For KataGo, following some informal experimentation, we chose $\alpha = 0.75$ for slightly sublinear long-term growth and $\beta = 0.4$ for early quick turnover of data.

[11] For the purpose of computing N_{total}, we also cap the number of initial random game samples at 250,000 even if more are generated. This makes runs slightly more consistent since it reduces dependence on timing of when the training and self-play machines are first started.

5 Search and Target Generation

Like AlphaZero and Leela Zero, KataGo uses a form of Monte-Carlo tree search (MCTS)[12]heavily biased by its neural net to generate self-play data for use in training, although a variety of the technical details differ. The result of a search in a self-play game is used to produce a training sample for the neural net.

KataGo's search is very similar to that of AlphaZero and Leela Zero. However, there are a number of differences and minor innovations worth highlighting in how KataGo performs search and produces a training sample from the search. We will give an overview of these details and differences.

5.1 PUCT, First-play urgency

KataGo currently shares the same basic exploration formula as in the original publication of AlphaGoZero[9]. Search consists of updating a gradually-growing game tree in memory by repeated playouts. Playouts start from the root and descend down the tree at each node n choosing the child c that maximizes:

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

where $V(c)$ is an estimate of the utility of c based on the average neural net evaluation of all nodes in c's subtree, $P(c)$ is the policy prior of c from the neural net, and $N(c)$ is the number of playouts of previously sent through child c. Upon reaching falling off the end of the tree in memory, a single new child is allocated and added to the tree, and the playout is terminated.

In the case where $N(c) = 0$ so that there is nothing with which to directly estimate $V(c)$, unlike AlphaZero but modeling closely after Leela Zero, KataGo uses the value estimate of the parent node n with a reduction:

$$V(c) \approx V(n) - c_{\text{FPU}} \sqrt{\sum_{c'} I\left(N(c') > 0\right) P(c')}$$

where $I(N(c') > 0)$ is the indicator function of whether a child c' has at least one playout and c_{FPU} is a "first-play-urgency" reduction coefficient where larger values discourage exploration.

For self-play, KataGo uses $c_{\text{PUCT}} = 1.1$ and $c_{\text{FPU}} = 0.2$ within the tree but $c_{\text{FPU}} = 0.0$ at the root. Additionally in KataGo, $V(c)$, rather than being a direct average of neural net evaluations under child c, is instead a weighted average that slightly downweights subchildren highly unlikely to be part of the principal variation. However, the behavior difference is very small[20].

One interesting detail in AlphaZero, is that unlike KataGo, the value estimate when $N(c) = 0$ is set to that of a complete loss[13]. Experiments by the MiniGo project confirm that the net effect of this relative to other first-play-urgency methods was to result in deeper searches[23] with seemingly positive results on learning efficiency, agreeing with similar informal experiments by Leela Chess

[12] As a reminder, the form of MCTS used by these programs and KataGo is actually *deterministic*, except for multithreading or randomness explicitly injected via board symmetries. The search algorithm was originally developed for use with Monte-Carlo rollouts, but unlike the original AlphaGo or earlier programs, AlphaZero uses only a neural net for evaluation without random rollouts, leaving "Monte-Carlo" a misnomer that has unfortunately stuck.

Zero. Although we are eager to try it, KataGo has not had the chance to test it since the time it was clarified that this was the original method of AlphaZero.

5.2 Root Noise and Scaling

Like AlphaZero, to promote discovery of unexpected new moves, KataGo adds noise according to a Dirichlet distribution[8]. Additionally, KataGo makes use of an idea from the SAI project and at the root applies a small temperature to the raw policy distribution to discourage the policy from too-rapidly converging to a single possible move when the estimated difference with alternative moves is small[24]. Both the noise and temperature are both only applied at the root.

The formula used is:

$$P(c) = w_{\text{noise}} \text{ Dirichlet}\left(\alpha = \frac{A}{\#\text{ Legal Moves}(c)}\right) + (1 - w_{\text{noise}}) \frac{P_{\text{raw}}(c)^{1/T_{\text{policy}}}}{\sum_{c'} P_{\text{raw}}(c')^{1/T_{\text{policy}}}}$$

where $w_{\text{noise}} = 0.25$, $A = 10.83 = 0.03 * 19^2$ is a constant chosen to match the AlphaZero Dirichlet noise of parameter $\alpha = 0.03$ on the empty 19×19 Go board but scale smoothly with the number of moves on the board so as to generalize to smaller board sizes, and $T_{\text{policy}} = 1.03$ is the temperature for the policy.

5.3 Forced Playouts and Target Pruning

Like AlphaZero and Leela Zero, as KataGo uses the final root playout distribution of each search to produce the training target for the neural net's policy prediction. However, KataGo does *not* use the raw root playout distribution. Instead, we introduce a new method of *policy target pruning* to transform the playout distribution first. Additionally, we introduce a new mechanism of *forced playouts* in the search that combines with the Dirichlet noise to improve the quality of exploration.

In KataGo, we observed in informal tests that even if a noise move could be good, the neural net's evaluation of it might initially be negative, necessitating several additional ply to actually reveal it as good, and the Dirichlet noise alone might only ensure a very shallow search[14]. Therefore, for each child c of the root that has received any playouts at all, we ensure it receives a minimum number of *forced playouts* based on the noised policy and the number of playouts in all children so far:

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

We do this by simply setting the MCTS selection urgency $\text{PUCT}(c)$ to infinity whenever a child of the root has fewer than this many playouts. This ensures that random moves selected by Dirichlet noise receive a minimum amount of exploration, yet in practice does not use more than a few percent of additional playouts on average.

[13] This fact is ambiguous in DeepMind's published papers due to ambiguity about whether a [-1,1] or [0,1] scale was used, and was only clarified much later by an individual researcher in a forum post here: `http://talkchess.com/forum3/viewtopic.php?f=2&t=69175&start=70#p781765`

[14] In AlphaZero, this issue might be partly mitigated due to the deeper searches resulting from the "first-play-urgency = loss" detail discussed earlier.

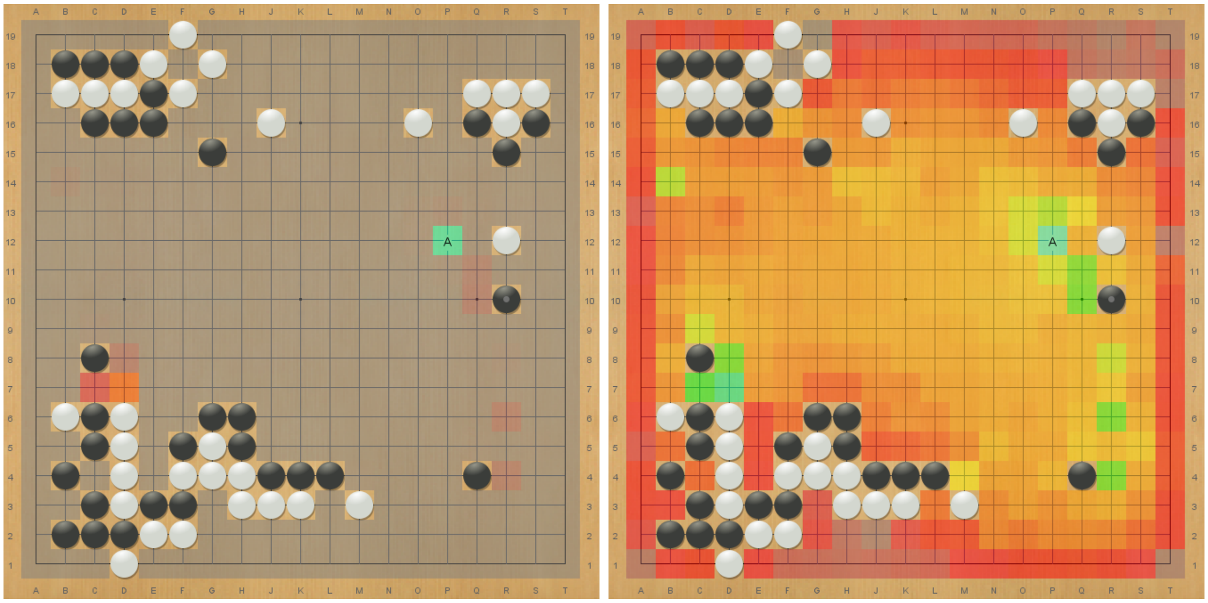

Figure 6: Left: policy prediction. Right: log of policy prediction. Ranking among even very-low-probability moves shows a clear gradient of likely move value, possibly a result of target pruning, although we have not compared to training without target pruning. Probability difference between center (orange, 2e-4) and upper right (faint red, 1e-5) is roughly a factor of 20.

However, the vast majority of the time, noise moves will be very bad moves[15]. To avoid contaminating the policy training target with the extra playouts on these bad moves, we perform a *target pruning* step that subtracts playouts that the search would not have chosen on its own given the final utility estimate for all children. In particular, we identify the child c^* with the most playouts, and then for each other child, we subtract up to n_{forced} many playouts from it so long as subtracting a playout 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.

Target pruning can also prune some playouts that are chosen by the search naturally! This can happen when playouts discover a child is worse than expected and a different child is proven to be much better, such that if the search had known the *final* estimated values of all children to begin with, the PUCT formula would never have originally invested playouts in the worse child. By forcing playouts and target pruning, we increase exploration in KataGo while keeping the policy training clean.

5.4 Score Maximization

Another feature of KataGo is that unlike AlphaZero or Leela Zero, KataGo puts nonzero utility on maximizing (a dynamically-determined monotone function of) the score difference.

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

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

[15] Even if a low percent of the policy target mass, this can disrupt the *relative* ordering of low-prior moves, which can be relevant when a search "goes wide" due to promising moves turning out poorly or due to long search times.

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 maximizing score, x_0 is a parameter for centering the utility curve, $B \in [9, 19]$ is the length of the board and $f : \mathbb{R} \to (-1, 1)$ is the function:

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

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. The sum of estimated utilities, averaged across nodes by MCTS, forms the signed value $V(c)$ for the PUCT formula in section 5.1.

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 the experiments in this paper, we set c_{score} initially to 0.5, then anneal it to 0.4 and then 0.3 at the same times as we also anneal the number of playouts upward (see section 6.1 below).

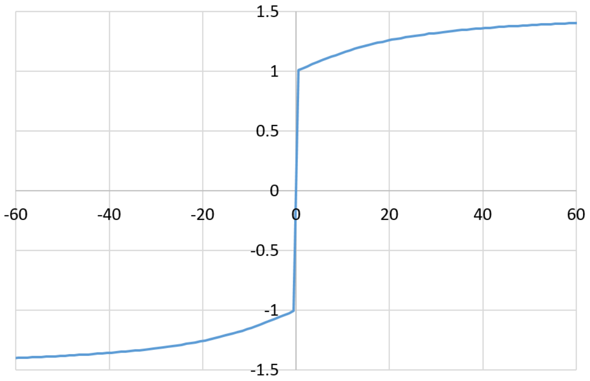

Figure 7: Total utility as a function of score difference, when $x_0 = 0$ and $B = 19$ and $c_{\text{score}} = 0.5$.

There are several motivations for maximizing score:

- Putting some utility on score should reduce variance in the ownership target for training. Without it for example, sometimes the bot will be indifferent to losing a territory that would not affect the game result, even if preventing the loss would be trivial and those points "should" be safe.

- Maximizing score might ensure a sharper and more consistent policy training target. If a player is strongly winning or losing, u_{win} may no longer meaningfully distinguish between different moves. However, moves that maximize score are highly likely to still be good moves if the game were closer. Score maximization ensures that the policy target remains sharp and continues to give useful data.
- Maximizing score is more useful for human study and play. Humans often find it valuable to study and learn the sharpest lines of play even when from a bot's perspective the game's outcome is no longer in doubt.

In our experiments in section 7.2, we find evidence that score maximization during self-play does provide a sharpening of the policy target, although its effect on ownership was less clear. Overall, it clearly improves the efficiency of learning.

5.5 Tree Reuse

Unlike AlphaZero or Leela Zero, KataGo deliberately does *not* preserve the search tree between turns during self-play, instead rebuilding it from scratch every turn.

We know of three common approaches for self-play data generation:

- Reuse the search tree between turns, always adding a fixed number of additional playouts to the tree, referred to as a *playout cap* (used by AlphaZero[16]).
- Reuse the search tree between turns, always searching until the tree reaches a certain total size, referred to as a *visit cap* (used by Leela Zero, at least for most of its computational lifetime[17]).
- Discard the search tree before each turn, searching until the tree reaches a certain total size (KataGo's and possibly some other projects' approach).

Holding computational cost per game constant, the first approach tends to spend more time adding playouts during game sequences that are highly forced or predictable. This results in larger searches on average, since more playouts are spent precisely sequences in which tree reuse is higher. However, compared to the second approach, it results in significantly smaller searches when moves have been less predictable or well-understood by the neural net, where the marginal playout might be more beneficial for improving the policy.

Unfortunately, the second approach has the possible disadvantage that on turns where almost all the tree is reused, there will be too few playouts remaining to properly explore moves upweighted by the Dirichlet noise, since the noise is only applied once a node becomes the root node.

The third approach, taken by KataGo, avoids this disadvantage, ensuring that the full number of playouts is always available to spend exploring noised moves. All neural net evaluations are cached in a large hash table, so rebuilding parts of the game tree that were explored on previous turns only costs some CPU time, which is quite minor given that the search is generally GPU-bound.

[16] To our knowledge, this detail of AlphaZero is not explicitly clarified in any of DeepMind's official publications, but was explained by an individual engineer here: `http://talkchess.com/forum3/viewtopic.php?p=782297#p782297`

[17] See discussion at: `https://github.com/leela-zero/leela-zero/issues/1416`

6 Self-play

In this section, we describe the high-level process of self-play, including the initialization, randomization, and termination of individual games. KataGo introduces the major innovation of *playout or visit cap oscillation*, as well as differing in several minor ways as well.

6.1 Playout Cap Oscillation

One of the major innovations in KataGo is to randomly reduce the number of playouts or visits spent on different turns to expand the amount of training data for the value head.

The game outcome value target is relatively data-starved even with the regularization of the ownership and score targets, since all of these targets receive only one new unique sample per entire game. For improving the value prediction of the net, it would be likely beneficial to reduce the number of playouts used per turn to generate more independent data samples per amount of computation, even if the quality of those samples would be slightly worse[18].

However, if the number of playouts is too low, the quality of the policy target diminishes rapidly. Some prior work[15][16] has heuristically suggested that at least in Go, ideal numbers of playouts for efficient policy learning per unit of computation are in the high 100s or low 1000s. With lower numbers, the search is given little opportunity to *deviate* significantly from the policy prior except to prevent major blunders, so the policy learns poorly. In an experiment in section 7.2 we do indeed observe that low playouts harms the overall learning process.

As a result, there is significant tension between the ideal number of playouts for policy training and for value training. In KataGo, we mitigate this tension through the technique of *visit cap oscillation* or *playout cap oscillation*, where the full number of playouts N is only used for a small proportion p of turns, and for all other turns a much smaller number of playouts $n < N$ is used. Only turns that use N playouts are recorded for training. On turns with only n, we also do *not* clear the search tree and we treat n as a visit cap, moving immediately if the tree already has size $>= n$. Since these turns are not used for training, we also disable Dirichlet noise and forced playouts and use $c_{\text{FPU}} = 0.2$, maximizing playing strength.

For the runs in this paper, we choose $p = 0.25$ and $(N, n) = (600, 100)$ early in training, annealing up to $(900, 150)$ and then $(1200, 200)$ later in training at about 90 million and 130 million self-play data samples generated, respectively, corresponding to about 1.4 million and 2.1 million games.

Holding computation fixed, this helps because most moves are played spending only n or fewer new playouts, so many more games are played, obtaining many more independent value training samples. However, since n is small, the majority of the computation is still on turns using N playouts, so the drop in the number of good policy training targets is not large.

The ablation studies presented in section 7.2 indicate that the net benefit of playout oscillation is large. Holding computational cost constant, in our runs it noticeably improves the efficiency of the overall self-play process.

[18] For a point of intuition about how high playouts might be less important for the value head, the first AlphaGo paper[7] showed that even a *single* playout per turn (i.e. directly using a policy net), was still sufficient to create data of enough quality to train a value net strong enough to defeat professional human players with the aid of search!

6.2 Other Self-play Differences

In addition to the major difference above, KataGo's self-play parameters also differ from AlphaZero in a number of other minor ways.

6.2.1 Reducing Playouts Instead of Resignation

Unlike AlphaZero or Leela Zero, KataGo does not terminate games early via resignation during self-play.

Instead, in the case of extreme winning chances, KataGo reduces the number of visits used in the search. During self-play, if both sides agree that for the last 5 turns, the worst MCTS winrate estimate p for the losing side has been less than 5%, then the number of visits is capped to $\lambda n + (1 - \lambda)N$ where n and N are the small and large limits used in visit cap oscillation and $\lambda = p/0.05$ is the proportion of the way that p is from 5% to 0%. Additionally, the training positions are recorded with only $0.1 + 0.9 * \lambda$ weight, downweighting training samples where the original AlphaZero process would have already resigned.

Playing out the full game allows the final determination of the ownership of all areas and the final score difference for training the auxiliary targets. Additionally, avoiding resignation also reduces the number of positions that are incorrectly recorded, improving the quality of the data.

6.2.2 Game Variety and Exploration

KataGo also differs slightly from AlphaZero and Leela Zero in the ways that it introduces entropy into self-play to encourage variety. Firstly, a wide variety of minor randomizations are applied to ensure a diversity of data with different rule sets and values of komi, as well as handicap games. See Appendix C for details.

KataGo also borrows the technique of *branching* the game to try an alternative line of play from the SAI project. In the SAI training process, this was primarily used to obtain data about the score values of positions by varying komi between branches[6], but here we adapt it instead as an aid for exploration. KataGo uses two branching mechanisms to ensure the neural net (1) has some experience in refuting unusual or bad moves, and (2) learns how to play openings resulting from unusual early opening moves:

1. In 2.5% of positions, the game is temporarily 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.

2. In 5% of games, the game is permanently 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 favorite by the neural net is played. Komi is adjusted to compensate the disadvantage of that move by iteratively querying the neural net for the score difference then adding it to komi. 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.

Since alternate moves are often bad moves, branching the game enables occasionally exploring them without contaminating the value training targets with a game result affected by the bad move, since all positions prior to the branch point are still trained towards the original game result.

6.2.3 Minor Endgame Optimizations

KataGo also performs a few Go-specific optimizations to speed up play.

In Go, define a group of stones to be *pass-alive* if none of those stones can be captured by the opponent even if the opponent gets unboundedly many consecutive moves. This can be determined efficiently by Benson's algorithm [1][19]. Furthermore, for each player $p \in \{\text{Black,White}\}$, define a maximal connected non-p region (possibly including stones of p's opponent) to be *pass-alive-territory* for p if the region is bordered by p and only p, and all bordering groups are pass-alive, and all but zero or one points of the region are adjacent to a bordering group.

Both AlphaZero and Leela Zero both only use the Tromp-Taylor rules for self-play learning in Go[20]. In KataGo in addition to randomizing other aspects of the rules, we also deviate from the Tromp-Taylor rules for scoring in which, informally, all stones are considered "alive" if left at the end of the game. We use an alternative scoring rule under which any *pass-alive-territory* belongs to that player *even if it contains dead opponent stones*, allowing the player to omit the moves to capture those dead stones.

Under these scoring rules it is easy to prove that if every region of the board is pass-alive or pass-alive-territory for at least one player, optimal play consists of both players passing and ending the game. So under these rules as a provably-safe optimization in such a case, we immediately terminate the game.

Additionally, we add two minor heuristic optimizations: firstly, if the opponent has passed at least four times in a row, we prohibit moves in either player's pass-alive territory. Secondly, we add a tiny bias at the root of search against moves in areas that the ownership prediction of the net indicates that the opponent almost certainly owns, or that the current player almost certainly owns unless filling opponent liberties or connecting non-pass-alive groups. The bias is smaller than any in-game score increment so as to only introduce a preference when the bot otherwise considers moves equal. These heuristic optimizations mildly speed up the end of the game.

[19] See also: `https://en.wikipedia.org/wiki/Benson%27s_algorithm_(Go)`

[20] No humans in practice use Tromp-Taylor rules, but they are computer-friendly for self-play and close enough to other rules that with minor hacks bots trained with them can be adapted for practical use. See: `https://senseis.xmp.net/?TrompTaylorRules`

6.2.4 Gating

Like some versions of AlphaZero and like Leela Zero, KataGo performs gating. New candidate neural nets from training are first tested against the current net being used for self-play to ensure that the new candidate is likely not much worse than the current net before replacing it.

KataGo's gating is fairly light so as to minimize latency and overhead. Candidate nets must win at least 100 out of 200 total games against the current net to be accepted as the new net for self-play. A fixed 300 visits are used (searching until the tree is size 300), annealing up to 400 and 500 at the same time as self-play visits are annealed upwards. Additionally, a variety of the self-play settings adding exploration are disabled in order to maximize playing strength. See Appendix D for details.

7 Experiments And Results

7.1 Testing Versus Leela Zero

KataGo's primary run took approximately one week, using 16×V100 GPUs[21]for self-play[22], 2×V100 for gating games, and 1×V100 for training, and an additional 1×V100 when a larger net was being concurrently trained to eventually overtake the smaller. We also increased the number of GPUs for self-play to 24×V100 and to 32×V100 at the time of annealing up to 900 and 1200 visits, respectively. The final 15-block neural net size was trained for about 270 million data samples, by which point about 160 million data samples had been generated from 2.5 million self-play games.

We then tested this run against Leela Zero:

- We sampled every fifth Leela Zero neural net from LZ30 up through LZ150, as well as LZ157 which is Leela Zero's strongest 15-block neural net prior to transitioning to larger sizes[23]. This roughly matches KataGo's largest neural net size. We also sampled KataGo's neural net over the course of its main run.
- Between every pair of Leela Zero nets less than 35 versions apart, we played approximately 120 games to establish approximate relative strengths of the Leela Zero nets as a benchmark.
- For each KataGo net, we played batches of games versus random Leela Zero nets choosing them with probability roughly proportional to the predicted variance $p(1-p)$ of the game result. The winning chance p was estimated from the Bayesian maximum likelihood Elo[24]based on all game results between all versions so far. This ensured that most games would be with Leela Zero nets close in strength, but with plenty of variety.

Games were played on a 19x19 board with a fixed 7.5 komi under Tromp-Taylor rules, with a fixed 800 visits and time management disabled and resignation enabled at a threshold of 2% winrate. KataGo's score maximization utility was set to 0.25. To ensure additional game variety, both

[21] Nvidia Tesla V100 GPU

[22] Playing roughly 3200 games in parallel (200 per GPU) to take advantage of batching of neural net queries.

[23] On strong consumer hardware, LZ30 might be beginner level, LZ50 weak/mid club player level, LZ80 experienced amateur level, LZ110 pro strength, and LZ130 either strong pro strength or a little beyond.

[24] We used a custom-implemented slight variant of https://www.remi-coulom.fr/Bayesian-Elo/

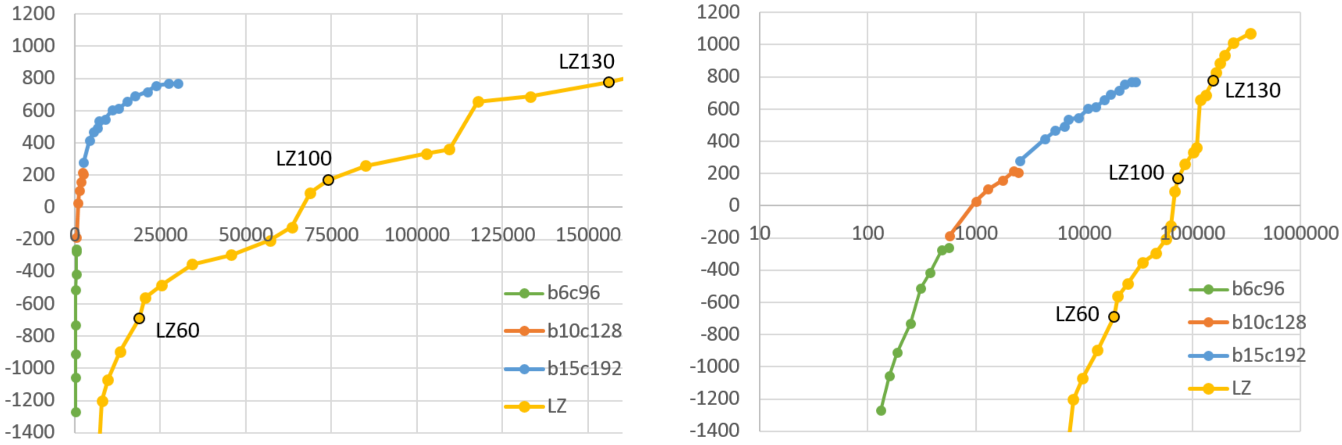

Figure 8: Relative 800-visit Elo strength of KataGo's main run's three net sizes (b6c96, b10c128, b15c192) vs. Leela Zero versions up through LZ157. X-axis is the cumulative self-play cost in millions of equivalent 20 block x 256 channel neural net queries, Y-axis is relative Elo. For Leela Zero, we ignored ELF games in computing cost. Left: linear scale. Right: log scale.

KataGo and Leela Zero were set to additionally randomize early in the game, Leela Zero with a temperature of 0.3 in the first 30 turns[25], and KataGo with a temperature of 0.3 decaying to 0.0 with a 30-turn halflife. Maximum-likelihood relative Elo ratings were computed using the final set of approximately 50000 games.

For both KataGo and Leela Zero, we estimated the cumulative self-play computation by crudely assuming that the evaluation of a neural net with b blocks and c channels has cost proportional to bc^2 and multiplying by number of neural net evaluations (assuming games are played in parallel so that batching reduces GPU overhead of small nets)[26]. For KataGo we simply counted the neural net evaluations. For Leela Zero we estimated it by scanning the public training data and multiplying the number of rows by the known number of playouts or visits used at that time, discounting by 20% as a rough estimate of neural net caching for transpositions. For tree reuse by Leela Zero, we conservatively assumed 100% reuse of all visits in the top move summed across training rows and we did *not* attempt to estimate the cost of ELF games generated by Leela Zero[27].

Shown in Figure 8 is the result of plotting Elo ratings versus estimated computation for both. As is shown, the combined effect of KataGo's techniques results in much faster learning. The reduction in computation required is close to a factor of 100(!) for the early parts of the process, and still a factor of about 5 at the furthest point we were able to progress KataGo's main run, just below the strength of LZ130, Leela Zero's 131st neural net. We also ran an additional 500 games using 6400 visits per turn between KataGo's final net and LZ130, achieving 226/500 wins (45%) and confirming a strength slightly below but near LZ130 at larger numbers of visits as well.

[25] For reference, the command line used for Leela Zero: `./leelaz --gtp --weights WEIGHTS.gz --threads 1 --visits 800 --resignpct 2 --noponder --timemanage off --randomcnt 30 --randomtemp 0.3 --log lz.log`

[26] Since for a given number of blocks and channels our neural nets are very close in computational cost to Leela Zero nets, this metric was chosen as a very rough way to normalize out hardware and implementation differences.

[27] Based on informal chat with some Leela Zero developers, starting around LZ132 the Leela Zero project also began using training data generated by the then-much-larger-and-stronger ELFv0 neural net from Facebook AI. In addition to possibly causing LZ132-LZ157 to be stronger than a 15-block net might achieve alone, we did *not* attempt to count the cost of this additional data.

Bot	Self-play 20b×256c Evals Used	Games	Elo	On good hardware, likely
Leela Zero LZ30	5800M	1.2M	-2307	Beginner
Leela Zero LZ80	46000M	4.0M	-295	Strong Club Player
Leela Zero LZ105	85000M	5.3M	259	Top Amateur
Leela Zero LZ130	157000M	7.1M	774	>= Strong Pro
Leela Zero LZ157	346000M	8.7M	1073	Superhuman
KataGo	562M	0.4M	-260	Strong Club Player
KataGo	2466M	1.0M	206	Top Amateur
KataGo	7111M	1.4M	534	Professional
KataGo	30000M	2.5M	767	>= Strong Pro

Table 4: Selected neural net versions from Leela Zero and from KataGo's main run, comparing the amount of self-play computation measured in equivalent 20 block x 256 channel neural net queries, again ignoring ELF games, against measured Elo ratings versus Leela Zero with 800 visits.

We did experience some diminishing returns near the end of KataGo's run, as is seen on the log-scale plot of Figure 8. It is certainly possible in longer runs that certain techniques would need to be adjusted to ensure maximal final strength. For example, perhaps playout or visit cap oscillation could need to be removed when fine-tuning near the end of a longer run for maximal-strength self-play. Alternatively, it is possible that auxiliary targets may consume a small part of the neural net's capacity, and reducing their weight later in training would improve strength by freeing some capacity for other targets. We did not test this. Our ablation runs in the next section also suggest that if strength only on 19x19 boards is desired, specializing to that size would also improve strength yet further.

7.2 Ablation Runs

We ran a variety of shorter ablation runs removing various major components and techniques presented in this paper to study the effect of their removal:

- NoAux - Removes the ownership, score, and opponent policy auxiliary training targets. Also removes score maximization behavior, since the neural net can no longer predict score.
- NoPAux - Removes the opponent policy auxiliary training target only.
- NoScore - Removes score maximization behavior only.
- NoOsc600a - Removes playout/visit oscillation, using a fixed 600 visits.
- NoOsc600b - Removes playout/visit oscillation, using a fixed 600 visits, and doubles the training window size.
- NoOsc200 - Removes playout/visit oscillation and reduces visits to a fixed 200 visits.
- NoGPool - Removes global pooling from residual blocks. Removes global pooling from the policy head *except* for computing the "pass" output, as KataGo's policy head is otherwise fully convolutional, leaving no other way to compute the pass output. Pooling is *not* removed from the value head.

- NoGoFeat - Removes all higher-level Go input features, including liberties, inescapable atari, pass-alive area, parity. Leaves only on-board, stones, illegal-ko location, history, and features indicating the rules. Also removes the Go-specific minor heuristics from section 6.2.3 involving passing and the tiny utility bias.
- NoSmall - Removes training on small boards, self-play and gating only occur on 19x19 boards.

To evaluate these runs, we sampled neural nets from all of these runs together along with KataGo's main run. Then, as with testing against Leela Zero, we repeatedly iterated through all sampled versions playing games on 19x19 with a fixed 7.5 komi using 800 visits against random opponents proportionally to the variance $p(1-p)$ of the game result. Again p was determined based on a maximum-likelihood Elo model all game results so far. Games were played in batches for GPU-efficiency. Maximum-likelihood relative Elo ratings were computed using the final set of approximately 76000 games (note that resulting Elos are not directly comparable with the Elos versus Leela Zero in section 7.1).

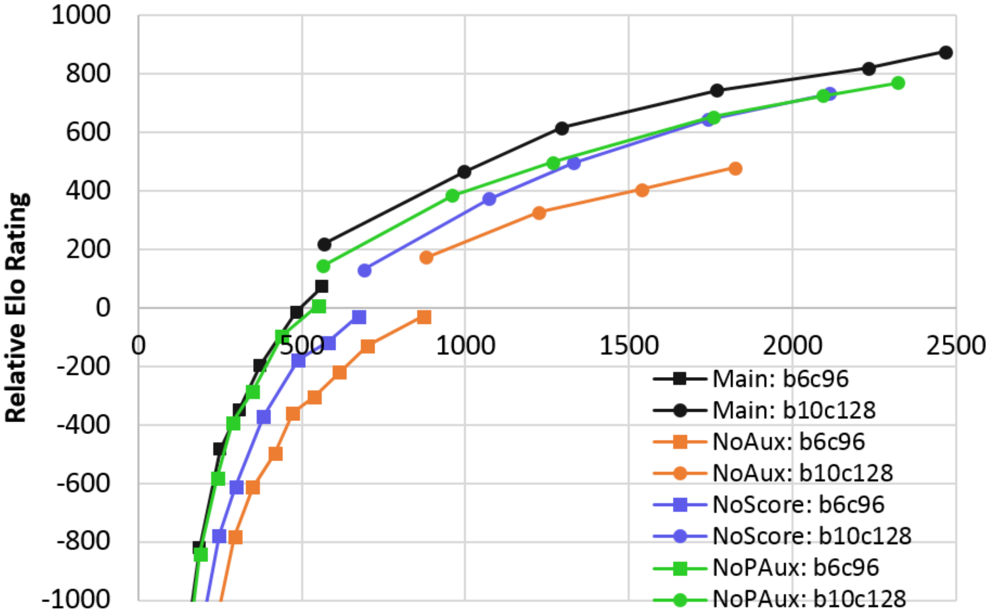

Figure 9: KataGo's main run versus NoAux and NoPAux and NoScore. X-axis is the cumulative self-play cost in millions of equivalent 20 block x 256 channel neural net queries.

As shown in Figure 9, removing auxiliary training targets and score maximization resulted in a noticeable drop in learning efficiency, confirming that at least up to the expert amateur level of the 10-block 128-channel neural net, the targets provide useful regularization. Removing the auxiliary policy target alone also harmed learning, indicating its usefulness separately from the other targets. And removing score maximization behavior alone also harmed learning. Figure 10 shows that its removal significantly increased the average entropy of the policy training target distribution, consistent with one of our original motivations for score maximization as a way to keep the policy target sharp and informative. Its effect on the ownership target was less clear.

As shown in Figure 11, removing playout oscillation resulted in a massive drop in learning efficiency. Since KataGo's current hyperparameters are adapted to the data generation rate expected from

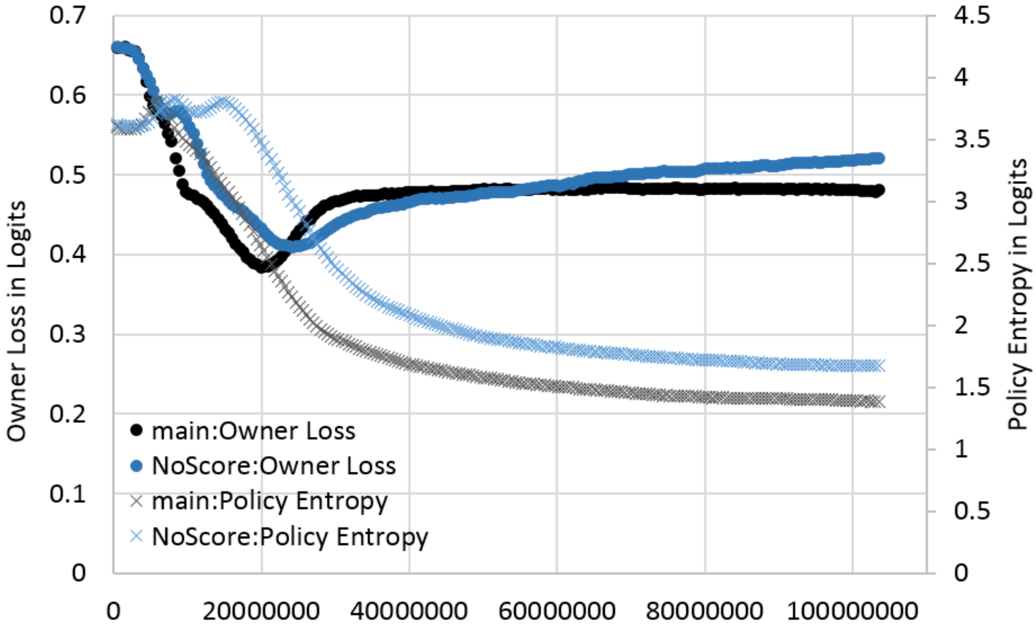

Figure 10: Training ownership loss and entropy of the policy target distribution, over the course of training the 6-block 96-channel neural net for main and NoScore runs. X-axis is the number of training samples.

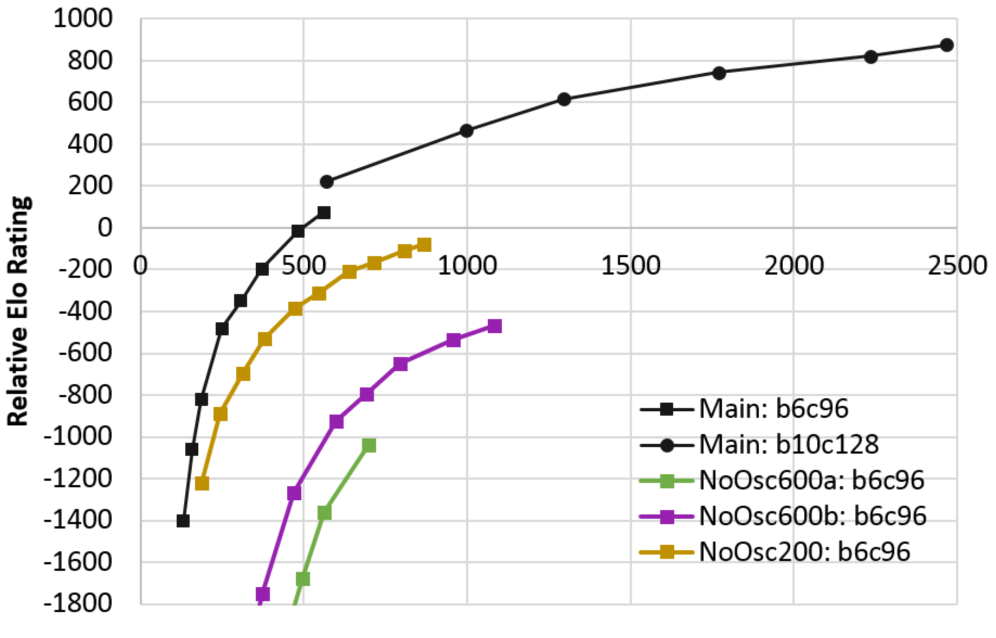

Figure 11: KataGo's main run versus NoOsc runs. X-axis is the cumulative self-play cost in millions of equivalent 20 block x 256 channel neural net queries.

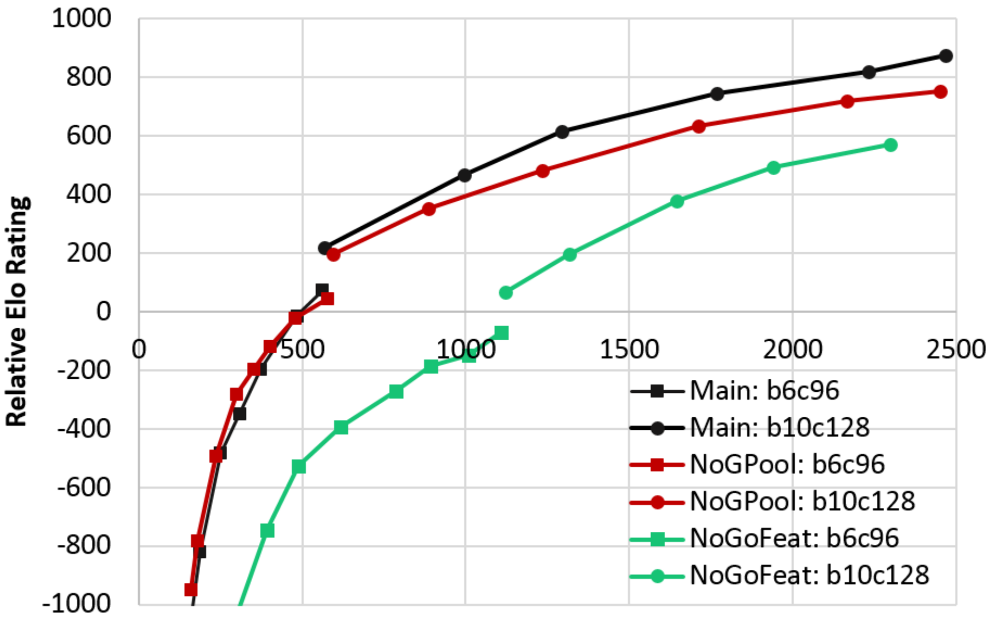

Figure 12: KataGo's main run versus NoGPool and NoGoFeat. X-axis is the cumulative self-play cost in millions of equivalent 20 block x 256 channel neural net queries.

oscillation, likely this drop could be reduced by hyperparameter re-tuning (indeed, doubling the training window mitigated a fraction of the drop by compensating for the reduced amount of value data). However, the magnitude of the drop and the fact that oscillation outperformed both a small (200) and larger (600) fixed number of visits is evidence that the technique is beneficial, at least this early in training.

The removal of global pooling shown in Figure 12 was interesting. Removal actually accelerated the earliest parts of learning but resulted in a longer-term lag. The long-term lag makes sense since at strong levels, distinguishing board sizes and other global context should be valuable. But perhaps global context is not relevant when play is still weak, or perhaps it hampers early learning by making it easy for the neural net to distinguish between board sizes, reducing transfer of learning from smaller boards. More tests would be needed to investigate this or other hypotheses.

Also in the same Figure 12, we show the result of removing higher-level features in KataGo that are arguably less in the spirit of "learning from zero", namely the Go-specific input features and some of the minor optimizations. Unsurprisingly, we observe a large drop in training efficiency, but far less than the total speedup obtained. Alongside the other ablation studies, this clearly indicates the value of the other improvements beyond merely specialized Go-specific input features and heuristics.

Lastly, we find in Figure 13 training only on 19x19 boards instead slightly slows early learning but achieves a slightly higher strength on 19x19, growing more over time (all testing games for measuring Elo were played only on 19x19). The fact that learning is slightly slower early and the final gain is gradual indicates that there is much shared learning between board sizes, since in KataGo's main run while 19x19 is upweighted to be a common size it is not even a majority of the data. It also suggests that at the cost of specializing to only 19x19, if desired we could also reach somewhat higher strengths than KataGo's main run did in the same amount of compute.

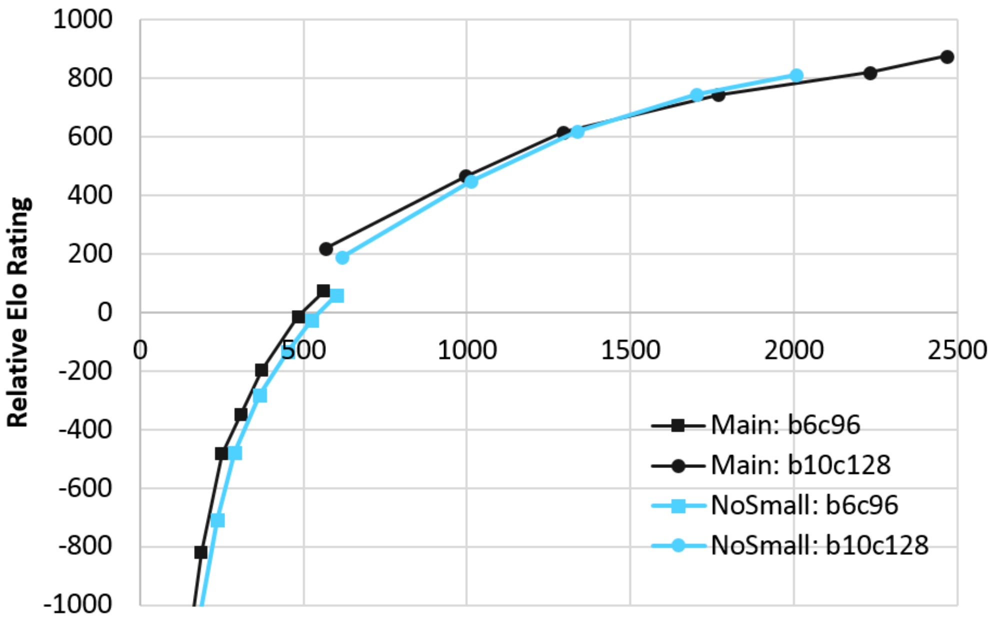

Figure 13: KataGo's main run versus NoSmall. X-axis is the cumulative cost self-play in millions of equivalent 20 block x 256 channel neural net queries.

8 Conclusions And Future Work

In this paper, we introduced a variety of techniques and methods for improving self-play learning in Go, demonstrating a gap between basic AlphaZero-like training and what could be possible with better methods. Our bot KataGo, up to the point we were able to test, achieves a substantial improvement in learning efficiency of more than 5x over the open-source Leela Zero in reaching a strong level of play, and can handle multiple board sizes and rulesets with a single set of learned weights. The speedup in the very earliest stages is even greater, such that with our released code, training a full 19x19 Go bot from zero up to at least moderate amateur strength may now be within the reach of individual consumer hardware!

However, we were unable to yet test these techniques in runs up to the full length of AlphaZero or of later replications such as that of Leela Zero's full run or Elf OpenGo. It is possible that some of the techniques presented will need to be adjusted when fine-tuning strength in a longer run, and future experimentation down those lines would be interesting and exciting. Moreover as we have discussed, many could have application to other games or to other broader problems in reinforcement learning. It is our hope that by presenting these ideas and their effective use so far in KataGo, we lay some groundwork for future research.

References

[1] David Benson. *Life in the Game of Go.* Information Sciences vol. 10, pp 17-29, 1976.

[2] Christopher Clark and Amos Storkey. *Training Deep Convolutional Neural Networks to Play Go.* In 32nd International Conference on Machine Learning, pp. 17661774, 2015.

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. *Identity Mappings in Deep Residual Networks.* In European Conference on Computer Vision, pages 630645. Springer, 2016. arXiv:1603.05027

[4] Jie Hu, Li Shen, Samuel Albanie, Gang Sun, Enhua Wu. *Squeeze-and-Excitation Networks.* IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pp. 7132-7141, 2018. arXiv:1709.01507

[5] Pavel Izmailov, Dmitrii Podoprikhin, Timur Garipov, Dmitry Vetrov, Andrew Gordon Wilson. *Averaging Weights Leads to Wider Optima and Better Generalization.* In Conference on Uncertainty in Artificial Intelligence, 2018. arXiv:1803.05407

[6] F. Morandin, G. Amato, R. Gini, C. Metta, M. Parton, G. C. Pascutto. *SAI, a Sensible Artificial Intelligence that plays Go.* arXiv preprint, arXiv:1809.03928v1 [cs.AI], 2018-09-11.

[7] 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 Vol. 529, pp. 484489, 2016-01-28.

[8] 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, Vol. 362, Issue 6419, pp. 1140-1144, 2018-12-07.

[9] 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, Vol. 550, pp. 354359, 2017-10-19.

[10] Yuandong Tian, Jerry Ma, Qucheng Gong, Shubho Sengupta, Zhuoyuan Chen, James Pinkerton, C. Lawrence Zitnick. *ELF OpenGo: An Analysis and Open Reimplementation of AlphaZero.* arXiv:1902.04522 [cs.AI], 2019-02-13.

[11] Yuandong Tian and Larry Zitnick. *Facebook Open Sources ELF OpenGo.* `https://research.fb.com/facebook-open-sources-elf-opengo/` Facebook Research online blog post, 2018-05-02, accessed 2019-02-16.

[12] Yuandong Tian and Yan Zhu. *Better Computer Go Player with Neural Network and Long-term Prediction.* In International Conference on Learning Representations, 2016. arXiv:1511.06410

[13] Ti-Rong Wu, I-Chen Wu, Guan-Wun Chen, Ting-han Wei, Tung-Yi Lai, Hung-Chun Wu, Li-Cheng Lan. *Multi-Labelled Value Networks for Computer Go.* IEEE Transactions on Games, Vol. 10, No. 4, pp. 378-389, December 2018. arXiv:1705.10701

[14] *Leela Zero.* `https://zero.sjeng.org/` Leela Zero project, main webpage, accessed 2019-02-16.

[15] Henrik Forsten ("Ttl"). *Bigger, stronger, not faster.* `https://github.com/leela-zero/leela-zero/issues/1030#issuecomment-374328038` Leela Zero project, GitHub issue comment, 2018-03-19, accessed 2019-02-16.

[16] Henrik Forsten ("Ttl"). *Optimal amount of visits per move.* https://github.com/leela-zero/leela-zero/issues/1416 Leela Zero project, GitHub issue, 2018-05-13, accessed 2019-02-16.

[17] *Leela Chess Zero.* https://lczero.org/ Leela Chess Zero project, main webpage, accessed 2019-02-16.

[18] *Technical Explanation of Leela Chess Zero.* https://github.com/LeelaChessZero/lc0/wiki/Technical-Explanation-of-Leela-Chess-Zero Leela Chess Zero GitHub wiki page, accessed 2019-02-09.

[19] David J. Wu ("lightvector"). *GoNN - Global Pooled Properties.* https://github.com/lightvector/GoNN#global-pooled-properties-dec-2017 GitHub page section, 2017-12, accessed 2019-02-16.

[20] David J. Wu ("lightvector"). *GoNN - Altering how MCTS performs averaging* https://github.com/lightvector/GoNN#altering-how-mcts-performs-averaging-nov-2018 GitHub page section, 2018-09, accessed 2019-02-16.

[21] *MiniGo.* https://github.com/tensorflow/minigo/ MiniGo project, GitHub repo, accessed 2019-02-16.

[22] *Experiment Squeeze and Excitation.* https://github.com/tensorflow/minigo/issues/683 MiniGo project, GitHub issue, 2019-02-04, accessed 2019-02-16.

[23] *(MiniGo results summary page).* https://github.com/tensorflow/minigo/blob/8e2b4b851a77a212bc6fbdea886a3cc20178d066/RESULTS.md MiniGo project, GitHub file, accessed 2019-02-09.

[24] *Pipeline description: self-play temperatures.* https://github.com/sai-dev/sai/issues/8 SAI project, GitHub issue, 2018-09, accessed 2019-02-16.

Appendix A Neural Net Architecture

The following is a detailed breakdown of KataGo's neural net architecture.[28]

The input to the neural net consists of two tensors. The first is size $B \times B \times C_{\text{spatial}}$ where B is the size of the board and $C_{\text{spatial}} = 22$ is the number of spatial input channels. The second is a vector of length C_{global} where $C_{\text{global}} = 14$ is the number of global input channels, whose entries are properties of the whole game state rather than of specific locations on the board.

The neural net consists of a main trunk, upon which multiple heads are attached, each head producing a different output for a different purpose. All value-related predictions by all heads are from the perspective of the current player.

The trunk consists of:

- A 5x5 convolution of the spatial input tensor outputting C channels, and in parallel, a matrix multiplication of the global input tensor outputting C channels.
- A channelwise addition where the C channels from the transformed global input are added as biases to the C channels of the 5x5 convolution.
- A stack of N residual blocks. $N - 2$ 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 from the start of the block that is added elementwise with the result.
- The remaining two blocks (positioned about halfway and three-quarters-way deep in the stack) consist of the following in order:
  - A batch-normalization layer.
  - A ReLu activation function.
  - A 3x3 convolution outputting C channels.
  - A *global pooling bias structure* (described below) that globally pools C_{pool} of the 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 from the start of the block that is added elementwise with the result.

[28] In the source code, there is an additional output head not described here that was used to support a regularization term for Japanese rules, but both it and Japanese rules were unused for the experiments in this paper.

- At the end of the stack of blocks, a batch-normalization layer.
- A ReLu activation function.

The *policy head* consists of:

- A 1x1 convolution outputting C_{head1} channels ("P") and in parallel a 1x1 convolution outputting C_{head1} channels ("G").
- A *global pooling bias structure* (described below) that globally pools 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 matrix multiplication of 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.

The *value head* consists of:

- A 1x1 convolution outputting C_{head1} channels ("V").
- A global pooling layer (described below) of V outputting $3C_{\text{head1}}$ values ("V_{pooled}").
- A game-outcome subhead consisting of:
  - A fully-connected layer from V_{pooled} including bias terms outputting C_{head2} 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[29].
    - 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[30].
    - 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_{head2} values.
    - A ReLu activation function.
    - A fully-connected layer including bias terms outputting 1 value ("γ").
  - For each possible final score value s:

    $$s \in [-S + 0.5, -S + 1.5, \ldots, -1.5, -0.5, 0.5, 1.5, \ldots, S - 1.5, S - 0.5]$$

    where S is a an upper bound for the plausible final score difference of any game[31], in parallel:
    - The $3C_{\text{head1}}$ 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[32].
    - A fully-connected layer (sharing weights across all s) from the $3C_{\text{head1}} + 2$ values including bias terms outputting C_{head2} 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.

A global pooling layer in KataGo takes a tensor with C channels (shape $B \times B \times C$) and outputs a vector of length $3C$ containing:

[29] 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.

[30] 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.

[31] In KataGo, we set $S = 19 * 19 + 60$, 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*.

[32] 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).

- The mean of each channel.
- The mean of each channel multiplied by $\frac{1}{10}(B - B_{\text{mid}})$
- The maximum of each channel.

where $B \in [B_{\min}, B_{\max}] = [9, 19]$ is the length of the board and $B_{\text{mid}} = \frac{1}{2}(B_{\min} + B_{\max})$. B_{mid} is subtracted to improve orthogonality, and $\frac{1}{10}$ is a arbitrary reasonable scaling constant so that the resulting values remain near unit scale.

In the value head, the third item 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 need to scale the mean quadratically with board length. $\frac{1}{100}$ is an arbitrary reasonable scaling constant to ensure unit-scale magnitudes, and subtracting σ^2 is to improve orthogonality with the other channels.

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 on the result (output shape $3C_G$).
- Multiplication by a matrix of size $3C_G \times C_X$ (output shape C_X).
- Channelwise addition with X, treating the C_X different values as per-channel bias terms (output shape $B \times B \times C_X$).

Three different neural net sizes are used for the experiments in this paper. The values of all the above constants for these three sizes can be found in Table 5.

Size	6b×96c	10b×128c	15b×192c
N	6	10	15
C	96	128	192
C_{pool}	32	32	64
C_{head1}	32	32	32
C_{head2}	48	64	80

Table 5: Architectural constants for various neural net sizes.

Appendix B Loss Function

The loss function 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[33], $\hat{z}$ 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 predicted policy distribution.

- 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 predicted opponent policy distribution, and $w_{\text{opp}} = 0.15$.

- Ownership loss:
$$-w_o \sum_{l \in \text{board}} \frac{1+o(l)}{2} \log\left(\frac{1+\hat{o}(l)}{2}\right) + \frac{1-o(l)}{2} \log\left(\frac{1-\hat{o}(l)}{2}\right)$$
where $o(l) \in \{-1, 1\}$ is the actual final owner of board location l, $\hat{o}(l) \in [-1, 1]$ is the neural net's prediction, and $w_o = 1.5/B^2$ where B is the length of the board.

- Score belief loss ("pdf"):
$$-w_{\text{spdf}} \sum_{x \in \text{possible scores}} p_s(x) \log(\hat{p}_s(x))$$
where $p_s(x)$ is a one-hot encoding of whether the final score difference is exactly x, and $\hat{p}_s(x)$ is the neural net's predicted probability that the final score difference is exactly x, 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 to be used for future research, and $w_{\text{rv}} = 0.2$ (in practice, the variances are small and therefore this loss term is mostly 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[34].

- 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[34].

- 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.

[34] 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 Game Initialization

Aside from the more interesting game branching mechanism described in section 6.2.2, KataGo randomizes in a variety of minor ways to ensure diverse training data. We enumerate these minor ways here:

- Since KataGo is designed to support multiple rulesets, games are randomized uniformly between positional versus situational superko rules, and between suicide moves allowed versus disallowed. Although KataGo supports it, for this paper simple ko rules are not used.
- As mentioned earlier in section 4.3, games are randomized in board size from 9 to 19 with frequency weights $1, 2, \ldots, 11$ but with the weight on size 19 further multiplied by 3, 5, or 10 as training progresses.
- To enable experience with different values of komi, rather than using a fixed 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. This ensures that almost all games are played under close-to-fair conditions for maximally informative learning, but that there is still some data with much more 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 proportionally to the raw policy distribution of the neural net. Of those games, 90% use the neural net to adjust komi to compensate White for Black's advantage. This is done after handicap placement by iteratively several times performing a neural net query for the expected final score difference given the placement, 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 length of the board.
- 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 down to 0.2 based on the turn number, with a halflife in turns equal to the length of the board B.

Appendix D Gating Game Initialization

Compared to the game initialization and randomization described in Appendix C, the following changes are made for gating games:

- The rules and board size are still randomized but komi is not randomized and is fixed at 7.5.
- Handicap games 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 $c_{\text{FPU}} = 0.0$.
- Since there is no need to complete the game to obtain ownership and score targets, 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%.