A. Detailed system and software design of ELF OpenGo

We now describe the system and software design of ELF OpenGo, which builds upon the DarkForest Go engine (Tian & Zhu, 2015) and the ELF reinforcement learning platform (Tian et al., 2017).

A.1. Distributed platform

ELF OpenGo is backed by a modified version of ELF (Tian et al., 2017), an Extensive, Lightweight and Flexible platform for reinforcement learning research; we refer to the new system as $\texttt{ELF}^{++}$. Notable improvements include:

- **Distributed support.** $\texttt{ELF}^{++}$ adds support for distributed asynchronous workflows, supporting up to 2,000 clients.
- **Increased Python flexibility.** We provide a distributed adapter that can connect any environment with a Python API, such as OpenAI Gym (Brockman et al., 2016).
- **Batching support.** We provide a simple autobatching registry to facilitate shared neural network inference across multiple game simulators.

A.2. Go engine

We have integrated the DarkForest Go engine (Tian & Zhu, 2015) into ELF OpenGo, which provides efficient handling of game dynamics (approximately $1.2\ \mu s$ per move). We use the engine to execute game logic and to score the terminal game state using Tromp-Taylor rules (Tromp, 2014).

A.3. Training system

ELF OpenGo largely replicates the training system architecture of AlphaZero. However, there are a number of differences motivated by our compute capabilities.

Hardware details We use NVIDIA V100 GPUs for our selfplay workers. Every group of eight GPUs shares two Intel Xeon E5-2686v4 processors.

We use a single training worker machine powered by eight V100 GPUs. Increasing the training throughput via distributed training does not yield considerable benefit for ELF OpenGo, as our selfplay throughput is much less than that of AlphaZero.

To elaborate on this point, our ratio of selfplay games to training minibatches with a single training worker is roughly 13:1. For comparison, AlphaZero's ratio is 30:1 and AlphaGo Zero's ratio is 7:1. We found that decreasing this ratio significantly below 10:1 hinders training (likely due to severe overfitting). Thus, since adding more training workers proportionally decreases this ratio, we refrain from multi-worker training.

GPU colocation of selfplay workers We use GPUs instead of TPUs to evaluate the residual network model. The primary difference between the two is that GPUs are much slower for residual networks. [9] Neural networks on GPUs also benefit from batching the inputs; in our case, we observed near-linear throughput improvements from increasing the batch size up to 16, and sublinear but still significant improvements between 16 and 128.

Thus, to close the gap between GPU and TPU, we co-locate 32 selfplay workers on each GPU, allowing the GPU to process inputs from multiple workers in a single batch. Since each worker has 8 game threads, this implies a theoretical maximum of 256 evaluations per batch. In practice, we limit the batch size to 128.

This design, along with the use of half-precision floating point computation, increases the throughput of each ELF OpenGo GPU selfplay worker to roughly half the throughput of a AlphaGo Zero TPU selfplay worker. While AlphaGo Zero reports throughput of 2.5 moves per second for a 256-filter, 20-block model with 1,600 MCTS rollouts, ELF OpenGo's throughput is roughly 1 move per second.

Asynchronous, heterogenous-model selfplays This colocation, along with the inherent slowness of GPUs relative to TPUs, results in much higher latency for game generation (on the order of an hour). Since our training worker typically produces a new model (i.e. processes 1,000 minibatches) every 10 to 15 minutes, new models are published faster than a single selfplay can be produced.

There are two approaches to handling this:

Synchronous (AlphaGo Zero) mode In this mode, there are two different kinds of clients: `Selfplay` clients and `Eval` clients. Once the server has a new model, all the `Selfplay` clients discard the current game being played, reload the new model and restart selfplays, until a given number of selfplays have been generated. Then the server starts to update the current model according to Equation 1. Every 1,000 minibatches, the server updates the model and notifies `Eval` clients to compare the new model with the old one. If the new model is better than the current one by 55%, then the server notifies all the clients to discard current games, and restart the loop. On the server side, the selfplay games from the previous model can either be removed from

[9] According to December 2018 DawnBench (Coleman et al., 2017) results available at `https://dawn.cs.stanford.edu/benchmark/ImageNet/train.html`, one TPU has near-equivalent 50-block throughput to that of eight V100 GPUs

Parameter/detail	AGZ	AZ	ELF OpenGo
c_{puct} (PUCT constant)	?	?	1.5
MCTS virtual loss constant	?	?	1.0
MCTS rollouts (selfplay)	1,600	800	1,600
Training algorithm	SGD with momentum = 0.9		
Training objective	value squared error + policy cross entropy + $10^{-4} \cdot$ L2		
Learning rate	$10^{\{-2,-3,-4\}}$	$2 \cdot 10^{\{-2,-3,-4\}}$	$10^{\{-2,-3,-4\}}$
Replay buffer size	500,000	?	500,000
Training minibatch size	2048	4096	2048
Selfplay hardware	?	5,000 TPUs	2,000 GPUs
Training hardware	64 GPUs	64 TPUs	8 GPUs
Evaluation criteria	55% win rate	none	none

Table S1. Hyperparameters and training details of AGZ, AZ, and ELF OpenGo. "?" denotes a detail that was ambiguous or unspecified in Silver et al. (2017) or Silver et al. (2018)

the replay buffer or be retained. Otherwise, the `Selfplay` clients send more selfplays of the current model until an additional number of selfplays are collected by the server, and then the server starts another 1,000 batches of training. This procedure repeats until a new model passes the win rate threshold.

Asynchronous (AlphaZero) mode Note that AlphaGo Zero mode involves a lot of synchronization and is not efficient in terms of boosting the strength of the trained model. In AlphaZero mode, we release all the synchronization locks and remove `Eval` clients. Moves are always generated using the latest models and `Selfplay` clients do not terminate their current games upon receiving new models. It is possible that for a given selfplay game record, the first part of the game is played by model A while the second part of the game is played by model B.

We initially started with the synchronous approach before switching to the asynchronous approach. Switching offered two benefits: (1) Both selfplay generation and training realized a drastic speedup. The asynchronous approach achieves over 5x the selfplay throughput of the synchronous approach on our hardware setup. (2) The ratio of selfplay games to training minibatches increased by roughly 1.5x, thus helping to prevent overfitting.

The downside of the asynchronous approach is losing homogeneity of selfplays – each selfplay is now the product of many consecutive models, reducing the internal coherency of the moves. However, we note that the replay buffer that provides training data is already extremely heterogeneous, typically containing games from over 25 different models. Consequently, we suspect and have empirically verified that the effect of within-selfplay heterogeneity is mild.

A.4. Miscellany

Replay buffer On the server side, we use a large replay buffer (500,000 games) to collect game records by clients. Consistent with Silver et al. (2017), who also use a replay buffer of 500,000 games, we found that a large replay buffer yields good performance. To increase concurrency (reading from multiple threads of feature extraction), the replay buffer is split into 50 queues, each with a maximal size of 10,000 games and a minimal size of 200 games. Note that the minimal size is important, otherwise the model often starts training on a very small set of games and quickly overfits before more games arrive.

Fairness of model evaluation In synchronous mode, the server deals with various issues (e.g., clients die or taking too long to evaluate) and makes sure evaluations are done in an unbiased manner. Note that a typically biased estimation is to send 1,000 requests to `Eval` clients and conclusively calculate the win rate using the first 400 finished games. This biases the metric toward shorter games, to training's detriment.

Game resignation Resigning from a hopeless game is very important in the training process. This not only saves much computation but also shifts the selfplay distribution so that the model focuses more on the midgame and the opening after learning the basics of Go. As such, the model uses the bulk of its capacity for the most critical parts of the game, thus becoming stronger. As in Silver et al. (2017), we dynamically calibrate our resignation threshold to have a 5% false positive rate; we employ a simple sliding window quantile tracker.

B. Auxiliary dataset details

B.1. Human games dataset

To construct the human game dataset, we randomly sample 1,000 professional games from the Gogod database from 2011 to 2015. [10]

B.2. Ladder dataset

We collect 100 games containing ladder scenarios from the online CGOS (Computer Go Server) service, where we deployed our prototype model. [11] For each game, we extract the decisive game state related to the ladder. We then augment the dataset 8-fold via rotations and reflections.

C. Practical lessons

ELF OpenGo was developed through much iteration and bug-fixing on both the systems and algorithm/modeling side. Here, we relate some interesting findings and lessons learned from developing and training the AI.

Batch normalization moment staleness Our residual network model, like that of AGZ and AZ, uses batch normalization (Ioffe & Szegedy, 2015). Most practical implementations of batch normalization use an exponentially weighted buffer, parameterized by a "momentum constant", to track the per-channel moments. We found that even with relatively low values of the momentum constant, the buffers would often be stale (biased), resulting in subpar performance.

Thus, we adopt a variant of the postprocessing moment calculation scheme originally suggested by Ioffe & Szegedy (2015). Specifically, after every 1,000 training minibatches, we evaluate the model on 50 minibatches and store the simple average of the activation moments in the batch normalization layers. This eliminates the bias in the moment estimators, resulting in noticeably improved and consistent performance during training. We have added support for this technique to the PyTorch framework (Paszke et al., 2017). [12]

Dominating value gradients We performed an unintentional ablation study in which we set the cross entropy coefficient to $\frac{1}{362}$ during backpropogation. This change results in optimizing the value network much faster than the policy network. We observe that with this modification, ELF OpenGo can still achieve a strength of around amateur dan level. Further progress is extremely slow, likely due to the minimal gradient provided by the policy network. This suggests that any MCTS augmented with only a value heuristic has a relatively low skill ceiling in Go.

[10]`https://gogodonline.co.uk/`

[11]`http://www.yss-aya.com/cgos/19x19/standings.html`

[12]As of December 2018, this is configurable by setting `momentum=None` in the `BatchNorm` layer constructor.