Reproducing AlphaZero with ELF: What we learned

Yuandong Tian
Facebook AI Research
AlphaGo Series

AlphaGo Lee (Mar. 2016)

AlphaGo Master (May. 2017)

AlphaGo Zero (Oct. 2017)
AlphaGo Series

AlphaGo Lee (Mar. 2016)

AlphaGo Master (May. 2017)

AlphaGo Zero (Oct. 2017)

Impressive Results, No code, No model
Demystifying AlphaGoZero/AlphaZero

• Hard to reproduce
  • Details are missing in the paper
  • Huge computational cost (15.5 years to generate 4.9M selfplays with 1 GPU)
  • Sophisticated (distributed) systems.

• Lack of ablation analysis
  • What factor is critical for the performance?
  • Is the algorithm robust to random initialization and changes of hyper parameters?
  • How the ladder issue is solved?

• Lots of mysteries
  • Is the proposed algorithm really universal?
  • Is the bot almighty? Is there any weakness in the trained bot?
Reimplementation of AlphaGoZero / AlphaZero

Generate Training data

Zero-human knowledge

Update Models

θᵢ

Self-Replays

[Silver et al, Mastering the game of Go without human knowledge, Nature 2017]
AlphaGo Zero

$$(s_1, \pi_1, z)$$

$$(s_2, \pi_2, z)$$

$$(s_3, \pi_3, z)$$

Training samples for $\theta_{i+1}$

MCTS $\theta_i$

$\pi_1$

$\pi_2$

$\pi_3$

$z$
AlphaGo Zero

\[
J(\theta) = (z - V_\theta)^2 - \pi^T \log p_\theta + c\|\theta\|^2
\]

Input features (19x19x17): \((X, Y, X_{-1}, Y_{-1}, \ldots, X_{-7}, Y_{-7}, C')\)
AlphaGo Zero Strength

• 3 days version
  • 4.9M Games, 1600 rollouts/move
  • 20 block ResNet
  • Defeat AlphaGo Lee.

• 40 days version
  • 29M Games, 1600 rollouts/move
  • 40 blocks ResNet.
  • Defeat AlphaGo Master by 89:11
ELF OpenGo

- System can be trained with 2000 GPUs in 2 weeks (20 block version)
- Superhuman performance against professional players and strong bots.
- Abundant ablation analysis
- Decoupled design, code reusable for other games.

We open source the code and the pre-trained model for the Go and ML community.
ELF OpenGo Timeline

2017
- Oct: AlphaGoZero paper Release
- Nov: AlphaZero Arxiv Release

2018
- Jan: OpenGo Starts
- Feb: Amateur level
- Mar: Model takes off
- Apr: Prototype Models Match with professional players
- May 2, 2018: Release Code/model

2019
- Oct: Final Model Reproduce our own progress!
ELF OpenGo Performance

Vs top professional players

<table>
<thead>
<tr>
<th>Name</th>
<th>ELO (world rank)</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Kim Ji-seok</td>
<td>3590 (#3)</td>
<td>5-0</td>
</tr>
<tr>
<td>Shin Jin-seo</td>
<td>3570 (#5)</td>
<td>5-0</td>
</tr>
<tr>
<td>Park Yeonghun</td>
<td>3481 (#23)</td>
<td>5-0</td>
</tr>
<tr>
<td>Choi Cheolhan</td>
<td>3466 (#30)</td>
<td>5-0</td>
</tr>
</tbody>
</table>

Single GPU, 80k rollouts, 50 seconds
Offer unlimited thinking time for the players

Vs professional players

Single GPU, 2k rollouts, 27-0 against Taiwanese pros.

Vs strong bot (LeelaZero)

[158603eb, 192x15, Apr. 25, 2018]: 980 wins, 18 losses (98.2%)
ELF OpenGo Sample Game
**ELF**: Extensive, Lightweight and Flexible Framework for Game Research

**ELF**: A simple for-loop

```
while True:
    batched_states = GameContext.Wait()
    replies = model(batched_states)
    GameContext.Steps(replies)
```
How ELF works

Game Threads (C++)

Python

```
while True:
    batched_states = GameContext.Wait()
    replies = model(batched_states)
    GameContext.Steps(replies)
```
Distributed ELF (version 1)

Selfplay 1 Selfplay 2  •  •  •  Selfplay n

Selfplay data

Training procedure (8 GPUs)

Current best model

Model Zoo

Current trained model

Evaluation 1

Evaluation 2

Evaluation m

Win rate > 55%
Distributed System (version 1)

Selfplay 1 → Open a port
Selfplay 2 → Receive selfplay data via ZeroMQ
Selfplay n

Training procedure

8 GPUs

Current trained model

Model Zoo

Evaluation 1
Evaluation 2
Evaluation m

Evaluation Server
Distributed System (version 1)

Training procedure (8 GPUs) → Model Zoo

Selfplay 1 → Selfplay 2 → ... → Selfplay n → Evaluation 1 → Evaluation 2 → ... → Evaluation m

Pick the best model and keep selfplaying

Current best model

300-2k GPUs

Each selfplay client batches 32 parallel games in a batch size of 128
Distributed System (version 1)

Training procedure (8 GPUs) → Model Zoo

Selfplay 1 → Model Zoo
Selfplay 2 → Model Zoo
... → Model Zoo
Selfplay n → Model Zoo

Model Zoo → Evaluation Server
Update best model and next candidate Win rate > 55%
No GPU needed

Evaluation 1
Evaluation 2
... Evaluation m
Distributed System (version 1)

Training procedure (8 GPUs) 

Selfplay 1  
Selfplay 2  
Selfplay n  

Model Zoo  

Evaluation Server  
Send the current model pairs to evaluate  

Evaluation 1  
Evaluation 2  
Evaluation m  

100 GPUs  

Each evaluation client batches 2 parallel games
Distributed ELF (v2)

Putting AlphaGoZero and AlphaZero into the same framework
- AlphaGoZero (more synchronization)
- AlphaZero (less synchronization)

Server controls synchronization
Server also does training.
Adaptation
We put our bot on Fox server
What we learned?
Training Stage of Final Model

Prototype-\(\alpha\) = strong amateur level

Prototype-\(\beta\) = professional level

Prototype = superhuman level (model against professional players)

A lot of zig-zag in the training process
Overfitting issues

Overestimate white winrate
Black resigns prematurally
Black loses many games
Imbalanced replay buffer

Large replay buffer is the key
Adaptive resign threshold has delays
However, it is quite stable.

- Without policy head, it can still achieve ~2d level.
- With strong correlation in batch, it still train 1/3 of the time.
- With batchnorm with shifted mean/std, it still works to some extend.
Ladder Issues

Run a ladder and lost
Run shorter ladder and lost
Doesn’t run ladder

There is only one long path that is correct
Value propagation is really slow.
Did we solve ladder? No

Why is the model still strong? → It plays alternative moves to avoid these situations.
AlphaZero versus AlphaGoZero

• AlphaZero is much faster than AlphaGoZero
  • No synchronization locks
  • After a day's training, model trained with AZ won 100:0 against model trained with AGZ

• Essentially a value/policy iteration with function approximation.
  • No evaluation needed.

• Zig-zag slight overfitting which leads to improvement
Why MCTS is so important?

Look-ahead is how new knowledge is created.

On Final Model

White rollouts 2x $\rightarrow$ $\sim85\%$ winrate

Black rollouts 2x $\rightarrow$ $\sim65\%$ winrate

Training is almost always constrained by model capacity (why 40b > 20b)
How sensible moves are learned?

Hypothetically

Game End

Where the reward signal is

Training Progresses

Game Start

Practically

Move 61-120 grows at the same rate as move 121-180

Match rate of each move against the prototype model.
Further train with learning rate $10^{-5}$ ...

- Surprisingly, it is not stable any more.
- Once at capacity, new models becomes similar to each other.
- Replay buffer becomes uniform and models start to overfit.
Conclusion

• The algorithm has pros and cons
  • Inductive bias
  • Planning is the key

• A lot of mysteries remain.
  • Why the method still works even with zig-zag and high-variance?
  • How to build a theoretical framework?
  • Maybe population-based approach is more stable?
  • More research to do
Challenge in Reproducibility

• How to reproduce a distributed ML/RL system like AlphaZero?
  • On-policy RL system does not have fixed dataset.
  • Distributed system poses more challenges.

• Practice
  • Fix the random seeds.
  • Record the script, the command argument and git commit number
    • Put the commit number into C++ library compilation.
  • Save the raw logs (stdout / stderr) and the script from raw logs to figures
Thanks!