CONVOLUTIONAL MONTE CARLO ROLLOUTS FOR THE GAME OF GO

Peter H. Jin & Kurt Keutzer
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA
phj@eecs.berkeley.edu & keutzer@berkeley.edu

ABSTRACT

In this work, we present a Monte Carlo tree search-based program for playing Go which uses convolutional rollouts. Our method performs MCTS in batches, explores the Monte Carlo tree using Thompson sampling and a convolutional policy network, and evaluates convnet-based rollouts on the GPU. We achieve strong win rates against an open source Go program and attain competitive results against state of the art convolutional net-based Go-playing programs.

1 INTRODUCTION

Recent work in convolutional networks for playing Go (Silver et al., 2016; Tian & Zhu, 2015; Maddison et al., 2014; Clark & Storkey, 2015; Sutskever & Nair, 2008) have produced deep convolutional policy networks for Go with up to 57% classification accuracy (non-ensemble) on datasets of historical Go game records. However, even the most successful deep convolutional net and MCTS-based program to date, Silver et al. (2016), still uses a local pattern-based policy for the rollout phase of MCTS inspired by the pioneering Go-playing program MoGo (Gelly et al., 2006).

Our contribution is threefold: (1) we implement a MCTS-based Go-playing program that uses convolutional policy networks executed on the GPU for both the tree policy and rollout policy; (2) we perform MCTS in batches to maximize the throughput of convolutions during rollouts; and (3) we demonstrate that Thompson sampling (Thompson, 1933) during exploration of the search tree in MCTS is a viable technique for computer Go. Combining those three techniques, we address the earlier concerns, and our program consistently wins against the open source Go programs and is also competitive against other deep convolutional net-based Go programs.

2 BATCH THOMPSON SAMPLING MCTS

We would like to use a convolutional policy network to perform Monte Carlo rollouts initialized from the leaf states of the search tree. To effectively evaluate convolutional rollouts on hardware platforms such as modern GPUs, it is necessary to perform batch convolutions. Naively exploring the search tree in batches using UCB1 (Auer et al., 2002) would lead to many duplicate initial states during rollouts; whereas one would prefer the a more varied distribution of initial states is preferred to induce more exploration of the tree. Asynchronous parallel versions of MCTS (Enzenberger & Müller, 2009) avoid this problem by adding "virtual losses" to the tree statistics to introduce variance during exploration (Chaslot et al., 2008).

Instead, we substitute for UCB1 the probabilistic bandit algorithm Thompson sampling (Thompson, 1933) as the search policy in MCTS, a choice justified by recent empirical evidence (Chapelle & Li, 2011), as well as proofs of its comparable regret bounds to those of UCB1 (Agrawal & Goyal, 2012; Kaufmann et al., 2012). During exploration, at each node in the search tree, the index j^* of the action to take is determined by the binomial Thompson sampling decision rule:

$$q_j \sim \text{Beta}(w_j + 1, n_j - w_j + 1) \tag{1}$$

$$j^* = \arg\max_j q_j \tag{2}$$

Table 1: Input features for policy networks.

Feature	Channels
Ones	1
Stones (empty/black/white)	3
Turns since	8
Chain liberties	8
Chain size (black/white)	16
Ko	1
Opponent rank	6
Distance to center	1
Total	**44**

Table 2: Policy network architectures and their classification accuracy. The prior policy accuracy is shown for the KGS dataset, while the other two are shown for the GoGoD dataset. A very deep network trained on the GoGoD dataset has lower accuracy (not shown) but exhibits stronger play.

Policy	Layers	Hidden channels	Accuracy
Prior policy	12	384	54.2%
Rollout policy A	2	16	33.7%
Rollout policy B	3	32	37.8%

where w_j and n_j are the wins and total plays statistics of the j-th actions for the node.

3 EXPERIMENTS

3.1 FEATURES AND NETWORK ARCHITECTURE

We used a hybrid of Tian and Zhu features (Tian & Zhu, 2015) and AlphaGo features (Silver et al., 2016); see Table 1. For our "opponent rank" feature, we quantized the traditional amateur dan and professional dan ranks of Go into 6 levels. For our "distance to center" feature, we used the formula $x_{ij} = \exp(-\frac{1}{2}\sqrt{u_{ij}^2 + v_{ij}^2})$ where (u_{ij}, v_{ij}) is the offset of the point $(i, j) \in \mathbb{N}_{19}^2$ from the center of the board.

We trained 3 policy networks: (1) a very deep tree policy network with 12 layers; (2) a shallow rollout policy network with 2 layers; and (3) a deep rollout policy network with 3 layers. The very deep network is based on the architecture of Tian & Zhu (2015), while the smaller networks are more similar to those of Sutskever & Nair (2008); see Table 2 for details.

3.2 TRAINING

We trained the policy networks on two datasets: over 175,000 games from the KGS dataset (Görtz, 2015); and over 74,000 games from GoGoD Winter 2015 edition (Hall & Fairbairn). For each training set, we held out over 1000 games (equivalent to over 200,000 positions) for validation.

We used stochastic gradient descent to train the policy networks. The very deep tree policy network was trained using 3-step lookahead prediction (Tian & Zhu, 2015) with a step size of 2^{-6} and no momentum. The rollout networks were trained using only next-step prediction with step size of 2^{-4} and a momentum of 0.1.

3.3 EVALUATION

We primarily evaluate against the open source MCTS-based Go program Pachi version 11.00 "Retsugen" (Baudiš & Gailly, 2012), with fixed 10,000 playouts per move and pondering during its op-

Table 3: Win rates of our convolutional policy networks against Pachi, with standard error bars (sample size is 256 for each of our own programs). The greedy prior policy selects the highest probability move of the prior policy network that is legal; it uses no search. The batch-MCTS programs use the prior policy in conjunction with rollout policy A or B. We also show the win rates of other comparable convolutional net-based MCTS programs.

Policy	Wins v. Pachi(10k)
(Maddison et al., 2014)	47.4 ± 3.7%
darkforest (Tian & Zhu, 2015)	71.5%
darkfores2 + MCTS (Tian & Zhu, 2015)	99.2%
Greedy prior policy	79.8 ± 2.5%
Batch-MCTS(1k) + rollout policy A	87.9 ± 2.0%
Batch-MCTS(1k) + rollout policy B	82.0 ± 2.4%

ponent's turn disabled. We also ran tests on GNU Go version 3.8 at level 10; our program was able to defeat GNU Go over 99% of the time, losing occasionally due to misjudging ladder situations.[1]

Our own MCTS-based program was set to evaluate 1024 rollouts per turn with a batch size of 32 rollouts. We used progressive widening (Coulom, 2007) to prune the search tree. RAVE (Gelly & Silver, 2007) did not seem to significantly affect the strength of our program. We note that using reduced-size input features to the rollout policy network, it is possible to attain between 1000 and 2000 rollouts per second on a server with 8× high end NVIDIA Maxwell GPUs (Titan X or Tesla M40).

3.4 Results

We show the winning rate of different variants of our Go program against Pachi in 3, where the variants differed in whether they used batch-MCTS or not, and which rollout policy network was used. First, we see that our batch-MCTS program with a convolutional rollout policy is able to achieve competitive win rates compared to the most comparable work to ours (Tian & Zhu, 2015). Second, the deeper rollout policy with 3 convolutional layers performs much worse in conjuction with batch-MCTS compared to the shallower 2-layer rollout policy. While counterintuitive, this is a known paradox that previous authors of MCTS Go programs have encountered (Gelly & Silver, 2007). Possible solutions include using Monte Carlo simulation balancing (Silver & Tesauro, 2009) or self-play policy gradient reinforcement learning (Silver et al., 2016) to learn a more appropriate rollout policy network.

4 Discussion

In this work, we demonstrated that combining convolutional networks in the exploration phase of MCTS with convolutional nets in the rollout phase is practical and effective through Thompson sampling-based batched MCTS. Our Go program consistently wins against the open source program Pachi and is competitive against other convolutional net and MCTS-based programs. Future work includes improving the quality of the rollout policy for stronger play.

Acknowledgments

This research is partially funded by DARPA Award Number HR0011-12-2-0016 and ASPIRE Lab industrial sponsors and affiliates Intel, Google, Hewlett-Packard, Huawei, LGE, NVIDIA, Oracle, and Samsung. We would also like to thank Forrest Iandola for insightful discussions, and Kostadin Ilov for assistance with our computing systems. Any opinions, findings, conclusions, or recommendations in this paper are solely those of the authors and does not necessarily reflect the position or the policy of the sponsors.

[1] `http://www.gnu.org/software/gnugo/`

REFERENCES

Shipra Agrawal and Navin Goyal. Analysis of Thompson Sampling for the Multi-armed Bandit Problem. COLT '12, 2012.

Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. Finite-time Analysis of the Multiarmed Bandit Problem. *Machine Learning*, 47:235–256, 2002.

Petr Baudiš and Jean-loup Gailly. Pachi: State of the Art Open Source Go Program. In *Advances in Computer Games*, pp. 24–38. 2012.

Olivier Chapelle and Lihong Li. An Empirical Evaluation of Thompson Sampling. In *Advances in Neural Information Processing Systems 24*, pp. 2249–2257, 2011.

Guillaume M.J.-B. Chaslot, Mark H.H. Winands, and H. Jaap van den Herik. Parallel Monte-Carlo Tree Search. In *Computers and Games*, pp. 60–71. 2008.

Christopher Clark and Amos Storkey. Training Deep Convolutional Neural Networks to Play Go. In *Proceedings of the 32nd International Conference on Machine Learning*, pp. 1766–1774, 2015.

Rémi Coulom. Computing Elo Ratings of Move Patterns in the Game of Go. In *Computer Games Workshop*, 2007.

Markus Enzenberger and Martin Müller. A Lock-Free Multithreaded Monte-Carlo Tree Search Algorithm. In *Advances in Computer Games*, pp. 14–20. 2009.

Sylvain Gelly and David Silver. Combining Online and Offline Knowledge in UCT. In *Proceedings of the 24th International Conference on Machine Learning*, pp. 273–280, 2007.

Sylvain Gelly, Yizao Wang, Rémi Munos, and Olivier Teytaud. Modification of UCT with Patterns in Monte-Carlo Go. Technical report, INRIA, 2006.

Ulrich Görtz. Game records in SGF format. `http://www.u-go.net/gamerecords/`, 2015. Accessed October 2015.

T. Mark Hall and John Fairbairn. Games of Go on Download. `http://gogodonline.co.uk/`. Accessed January 2016.

Emilie Kaufmann, Nathaniel Korda, and Rémi Munos. Thompson Sampling: An Asymptotically Optimal Finite-Time Analysis. In *Proceedings of the 23rd International Conference on Algorithmic Learning Theory*, pp. 199–213, 2012.

Chris J. Maddison, Aja Huang, Ilya Sutskever, and David Silver. Move Evaluation in Go Using Deep Convolutional Neural Networks. *arXiv preprint arXiv:1412.6564*, 2014.

David Silver and Gerald Tesauro. Monte-Carlo Simulation Balancing. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pp. 945–952, 2009.

David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

Ilya Sutskever and Vinod Nair. Mimicking Go Experts with Convolutional Neural Networks. In *Proceedings of the 18th International Conference on Artificial Neural Networks, Part II*, pp. 101–110, 2008.

William R. Thompson. On the likelihood that one unknown probability exceeds another in view of the evidence of two samples. *Biometrika*, 25(3/4):285–294, 1933.

Yuandong Tian and Yan Zhu. Better Computer Go Player with Neural Network and Long-term Prediction. *arXiv preprint arXiv:1511.06410*, 2015.