Spatial Average Pooling for Computer Go

Tristan Cazenave

Université Paris-Dauphine
PSL Research University
CNRS, LAMSADE
PARIS, FRANCE

Abstract. Computer Go has improved up to a superhuman level thanks to Monte Carlo Tree Search (MCTS) combined with Deep Learning. The best computer Go programs use reinforcement learning to train a policy and a value network. These networks are used in a MCTS algorithm to provide strong computer Go players. In this paper we propose to improve the architecture of a value network using Spatial Average Pooling.

1 Introduction

Monte Carlo Tree Search (MCTS) has been successfully applied to many games and problems [1]. The most popular MCTS algorithm is Upper Confidence bounds for Trees (UCT) [9]. MCTS is particularly successful in games [8]. A variant of UCT when priors are available is PUCT [11]. AlphaGo [12] uses a variant of PUCT as its MCTS algorithm. AlphaGo Zero [14] and AlphaZero [13] also use PUCT as their MCTS algorithm. Golois, our computer Go player, uses as its MCTS algorithm the same variant of PUCT as AlphaGo.

AlphaGo uses a policy network to bias the choice of moves to be tried in the tree descent, and a value network to evaluate the leaves of the tree. In AlphaGo Zero, the evaluation of a leaf is uniquely due to the value network and playouts are not used anymore. Moreover the policy and value network are contained in the same neural network that has two heads, one for the policy and one for the value.

AlphaGo and AlphaGo Zero were applied to the game of Go. The approach has been extended to chess and Shogi with AlphaZero [13]. After a few hours of self play and training with 5 000 Tensor Processing Units from Google, AlphaZero was able to defeat top Chess and Shogi programs (Stockfish and Elmo) using a totally different approach than these programs. AlphaZero uses 1,000 times fewer evaluations than Stockfish and Elmo for the same thinking time. It uses PUCT instead of AlphaBeta and a combined value and policy network.

The AlphaGo Zero approach has been replicated by many researchers. The Leela Zero program is a community effort to replicate the AlphaGo Zero experiments. People donate their GPU time to make Leela Zero play self-play games [10]. The networks trained on self play games are then tested against the current best network and replace it if the result of the match is meaningful enough. The best network is then used for randomized self-play. Most of the computing time used by programs replicating the AlphaGo Zero approach is spent in self-play.

The ELF framework from Facebook AI Research [15] is an open source initiative to implement reinforcement algorithms for games. It has been applied to the game of Go following the AlphaGo Zero approach [16]. The resulting ELF Go program running on a single V100 GPU has beaten top Korean professionals Go players 14 to 0 and Leela Zero 200 to 0. It was trained for two weeks using 2 000 GPUs. It is a strong superhuman level computer Go player, however it has the same kind of weaknesses as Leela Zero and other Zero bots: it sometimes plays a ladder that is not working and loses the game because of this ladder problem.

Another partially open source Go program is Phoenix Go by Tencent [20]. It won the last computer Go tournament at Fuzhou, China in April 2018 defeating FineArt and LeelaZero.

In this paper we are interested in improving a value network for Golois our computer Go program. We have previously shown that residual networks can improve a policy network [3, 2]. We also use residual networks for our value network which is trained on self-play games of the policy network. We propose to improve on the standard residual value network adding Spatial Average Pooling layers to the usual residual value network. Our experiments are performed using Golois with and without Spatial Average Pooling. The AQ open source Go program [19] also uses Spatial Average Pooling in its value network.

We now give the outline of the paper. The next section outlines the training of a value network. The third section details the PUCT search algorithm. The fourth section explains Spatial Average Pooling. The fifth section gives experimental results. The last section concludes.

2 Training a value network

Training of the value network uses games self-played by the policy network. Golois policy network has a KGS 4 dan level using residual networks and three output planes [17, 3, 2].

The playing policy is randomized, Golois chooses a move randomly between the moves advised by the policy network that have a probability of being the best move greater than the probability of the best move minus 0.2. This policy enable sufficient randomization while retaining a good level of play. This is the randomization strategy that was used to make Golois policy network play on KGS.

The architecture of the policy network uses nine residual blocks an three output planes, one for each of the three next moves of the game. The network was trained on games between professional players played between 1900 and 2015.

The architecture of our first value network is also based on residual networks and has nine residual blocks. The first layer of the network is a convolutional layer with 1x1 filters that takes the 47 input planes and transform them into 256 19x19 planes. The last layers of the network are a 1x1 convolution layer that converse the 256 planes to a single plane, the single plane is then reshaped into a one dimensional tensor and followed by two completely connected layers.

In order to be able to play handicap games, Golois uses nine outputs for its value network. One output for each possible final score of a self-play game between 180.0 and

189.0. The final score is the score of the Black player. All output neurons representing a score greater than the score of the self-played game are set to one during training and all neurons strictly smaller are set to zero. For example if the score of a game is 183.0, the first three outputs are set to zero and the next six outputs are set to one.

When using the value network for a game, the corresponding neuron is used for the evaluation of states. If the game is even and the komi is 7.5 the neuron for a score greater than 184.0 is used, if the game is handicap one and the komi is 0.5 the neuron for a score greater than 180.0 is used. Using multiple output planes for the value network has also been used independently for the CGI Go program [18].

3 PUCT

In order to be complete, the PUCT algorithm is given in algorithm 1. Lines 2-5 deals with getting the possible moves and stopping if the board is terminal. Line 6 gets the entry of the board in the transposition table. Each board is associated to a Zobrist hash code that enables to calculate the index of the board in the transposition table. An entry in the transposition table contains the total number of playouts that have gone through the state, the mean of all the evaluations of the children of the node and of the node itself, the number of playouts for each possible moves, and the prior for each possible move given by the policy network. The policy network uses a softmax activation for the output of the network, so the priors given by the policy network can be considered as probabilities of each move being the best. Lines 7-23 are executed when the state has already been seen and is present in the transposition table. The goal is to find the move that maximizes the PUCT formula. The PUCT formula is:

$$argmax_m(mean_m + c \times prior_m \times \frac{\sqrt{t}}{p_m})$$

with c being the PUCT constant, $prior_m$ being the probability for move m given by the policy network, t being the sum of the number of playouts that have gone through the node and p_m being the number of playouts that start with move m.

On line 20 the move that maximizes the PUCT formula is played, then the recursive call to PUCT is done online 22 for the selected child of the current node. When PUCT returns from the call and gets the evaluation of the new leaf, it updates the values of the current node with the result of the search on line 23. This means it increases by one the total number of playouts of the node, it increases by one the playouts of move m and updates the mean of move m with res.

Lines 25-27 are executed when the state is not part of the PUCT tree. In this case it adds an entry in the transposition table for this state and gets the evaluation of the board from the value network. We use MCTS without playouts. The leaves are evaluated by the value network alone. The value network is run on the eight symmetrical boards of the state to evaluate, and the average of the eight evaluations is the evaluation of the leaf.

We use tree parallelism for the PUCT search of Golois. Twelve threads are running in parallel and share the same tree. Each thread is assigned to one of the four GPUs and calls the forward pass of the policy and value networks on this GPU.

Algorithm 1 The PUCT algorithm.

```
1: PUCT (board, player)
2:   moves ← possible moves on board
3:   if board is terminal then
4:     return evaluation (board)
5:   end if
6:   t ← entry of board in the transposition table
7:   if t exists then
8:     bestValue ← $-\infty$
9:     for m in moves do
10:      $t \leftarrow t.totalPlayouts$
11:      $mean \leftarrow t.mean[m]$
12:      $p \leftarrow t.playouts[m]$
13:      $prior \leftarrow t.prior[m]$
14:      $value \leftarrow mean + c \times prior \times \frac{\sqrt{t}}{p}$
15:      if $value > bestValue$ then
16:        $bestValue \leftarrow value$
17:        $bestMove \leftarrow m$
18:      end if
19:    end for
20:    play (board, bestMove)
21:    player ← opponent (player)
22:    res ← PUCT (board, player)
23:    update t with res
24:  else
25:    t ← new entry of board in the transposition table
26:    res ← evaluation (board, player)
27:    update t
28:  end if
29:  return res
```

We have also found that it improves the number of nodes per second to use a minibatch greater than 8. The standard algorithm uses minibatch of size 8 since there are eight symmetrical states for a leaf of the tree. However current GPUs can be used more efficiently with larger minibatches. The best results we had were with minibatches of size 16. We only get the value of leaf every two leaves. It means that after the first call to PUCT, a second tree descent is performed to get a second leaf to evaluate corresponding to 8 more states. The second tree descent does not usually find the same leaf as the first tree descent since during each tree descent a virtual loss is added to the number of playouts of the selected move. This ensures that the further tree descents do not always select the same moves. So after the second descent, both the first leaf and the second leaf are evaluated, with 8 symmetrical states each, resulting in a minibatch of 16 states.

4 Spatial Average Pooling

Spatial Average Pooling takes the average of a rectangle of cells of the input matrix as the output of the layer. Table 1 illustrates the application of a 2x2 Spatial Average Pooling on a 4x4 matrix. The elements of the 4x4 matrix are split into four 2x2 matrices and each 2x2 matrix is averaged to give each element of an output 2x2 matrix.

Table 1. Spatial Average Pooling.

8	3	4	7
4	1	6	3
2	6	3	9
1	7	2	6

4	5
4	5

We used Spatial Average Pooling in the last layers of Golois value network with a size 2x2 and a stride of 2 as in the table 1 example.

When applying Spatial Average Pooling with a size 2x2 and a stride of 2 to 19x19 planes, we add a padding of one around the 19x19 plane. Therefore the resulting planes are 10x10 planes. When applying Spatial Average Pooling again to the 10x10 planes with a padding of one we obtain 6x6 planes. The last convolutional layer of the value network is a single 6x6 plane. It is flattened to give a vector of 36 neurons. It is then followed by a 50 neurons layer and the final 9 neurons output followed by a Sigmoid (the value network outputs the probability of winning between 0 and 1).

The Spatial Average Pooling is meaningful for a value network since such a networks outputs a winning probability that is related to the estimated score of the board. If neurons in the various planes represent the probability of an intersection to be Black territory in the end, averaging such probabilities gives the winning probability. So using Spatial Average Pooling layers can push the value network to represent probabilities of ownership for the different parts of the board and help the training process.

In AlphaGo Zero [14], policy and value networks share the same weights of a network with two different heads. One head for the policy network and one head for the value network. Our improvement of the value network can still be used in such an architecture, using Spatial Average Pooling for the value head.

5 Experimental results

Experiments make PUCT with a given network play against PUCT with another network. 200 games are played between algorithms in order to evaluate them. Each move of each game is allocated 0.5 seconds on a four GPU machine with 6 threads. This enables to play between 40 and 80 tree descents per move. In all experiments we use a PUCT constant of 0.3 which is the best we found.

The experiments were done using the Torch framework [7], combining C++ code for the PUCT search with lua/Torch code for the forward passes of the networks as well as for the training of the value networks. The minibatches are created on the fly with C++ code that randomly chooses states of the self played games and combine them in a size 50 minibatch. Each state is associated to the result of the self played game. Once the minibatch is ready it is passed to the lua code that deals with the computation of the loss and the Stochastic Gradient Descent. We use the same 1,600,000 self played games for training the value networks.

The value network including Spatial Average Pooling has 128 planes for each layer. It starts with six residual blocks then applies Spatial Average Pooling, followed by three residual blocks then another Spatial Average Pooling, followed by three other residual blocks. The two fully connected layers of 50 and 9 neurons complete the network. This value network is named SAP (6,3,3) in table 2.

The competing value network is the standard residual value network used in Golois. It has nine residual blocks with 256 planes per layer. It is named α (9,256) in table 2. Deeper residual value networks were trained for Golois without giving much better results, that is why we kept the nine blocks value network. The original AlphaGo used 13 layers convolutional networks while AlphaGo Zero uses either 20 or 40 residual blocks with 256 planes. Our self-play data is not as high level as the self-play data of AlphaGo Zero. That may explain why deeper networks make little difference.

Table 2 gives the evolution of the training losses of the two networks with the number of epochs. One epoch is defined as 5 000 000 training examples. The minibatch size is 50, so an epoch is composed of 100,000 training steps. We can see in table 2 that SAP (6,3,3) starts training with a smaller loss than α (9,256), but that eventually the losses are close after 63 epochs.

Table 2. Evolution of the training loss of the value networks.

Epochs	1	3	7	15	31	63
α (9,256)	677	560	532	522	516	510
SAP (6,3,3)	654	554	530	521	515	511

We made the SAP (6,3,3) value network play fast games against the α (9,256) value network. Both networks use the same policy network to evaluate priors and the parameters of the PUCT search such as the PUCT constant were tuned for the α (9,256) value network. We found that a small PUCT constant of 0.3 is best, this may be due to the quality of the policy network that implies less exploration and more confidence in the value network since it directs the exploration toward the good moves.

SAP (6,3,3) wins 70.0% of the time against the usual residual value network. The size of the network file for SAP (6,3,3) is 28,530,177 while the size of the network file for α (9,256) is 85,954,310. The training time for 100 minibatches is 6.0 seconds for SAP (6,3,3) while the training time for 100 minibatches is 12.5 seconds for α (9,256).

Using this network, Golois reached an 8d level on the KGS Go server running on a 4 GPU machine with approximately 2 500 tree descents and 9 seconds thinking time per move.

6 Conclusion

We have proposed the use of Spatial Average Pooling to improve a value network for the game of Go. The value network using Spatial Average Pooling is much smaller than the usual residual value network and has better results.

We have also detailed our parallel PUCT algorithm that makes use of the GPU power by making forward passes on minibatches of states instead of a single state.

The value network we have trained has multiple output neurons instead of one as in usual networks It enables it to be used with different komi values and therefore to play handicap games correctly. It is important for game play on servers such as KGS where due to its 8d strength it plays handicap games most of the time.

In future work we plan to use Spatial Average Pooling for the value head of a combined value/policy network. We also plan to improve the search algorithm and its parallelization [4, 6, 5].

Bibliography

[1] Cameron Browne, Edward Powley, Daniel Whitehouse, Simon Lucas, Peter Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of Monte Carlo tree search methods. *IEEE TCIAIG*, 4(1):1–43, March 2012.

[2] Tristan Cazenave. Improved policy networks for computer go. In *Advances in Computer Games - 15th International Conferences, ACG 2017, Leiden, The Netherlands, July 3-5, 2017, Revised Selected Papers*, pages 90–100, 2017.

[3] Tristan Cazenave. Residual networks for computer go. *IEEE Transactions on Games*, 10(1):107–110, 2018.

[4] Tristan Cazenave and Nicolas Jouandeau. On the parallelization of UCT. In *proceedings of the Computer Games Workshop*, pages 93–101. Citeseer, 2007.

[5] Tristan Cazenave and Nicolas Jouandeau. A parallel monte-carlo tree search algorithm. In *Computers and Games, 6th International Conference, CG 2008, Beijing, China, September 29 - October 1, 2008. Proceedings*, pages 72–80, 2008.

[6] Guillaume MJ-B Chaslot, Mark HM Winands, and H Jaap van Den Herik. Parallel monte-carlo tree search. In *International Conference on Computers and Games*, pages 60–71. Springer, 2008.

[7] Ronan Collobert, Koray Kavukcuoglu, and Clément Farabet. Torch7: A matlab-like environment for machine learning. In *BigLearn, NIPS Workshop*, number EPFL-CONF-192376, 2011.

[8] Rémi Coulom. Efficient selectivity and backup operators in Monte-Carlo tree search. In *Computers and Games*, pages 72–83, 2006.

[9] Levente Kocsis and Csaba Szepesvári. Bandit based Monte-Carlo planning. In *17th European Conference on Machine Learning (ECML'06)*, volume 4212 of *LNCS*, pages 282–293. Springer, 2006.

[10] Gian-Carlo Pascutto. Leela zero. `http://zero.sjeng.org/`, 2018.

[11] Christopher D Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61(3):203–230, 2011.

[12] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[13] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *arXiv preprint arXiv:1712.01815*, 2017.

[14] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton,

Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[15] Yuandong Tian, Qucheng Gong, Wenling Shang, Yuxin Wu, and C. Lawrence Zitnick. Elf: An extensive, lightweight and flexible research platform for real-time strategy games. In *Advances in Neural Information Processing Systems*, pages 2656–2666, 2017.

[16] Yuandong Tian, Jerry Ma*, Qucheng Gong*, Shubho Sengupta, Zhuoyuan Chen, and C. Lawrence Zitnick. Elf opengo. `https://github.com/pytorch/ELF`, 2018.

[17] Yuandong Tian and Yan Zhu. Better computer go player with neural network and long-term prediction. *ICLR*, 2016.

[18] T.-R. Wu, I. Wu, G.-W. Chen, T.-h. Wei, T.-Y. Lai, H.-C. Wu, and L.-C. Lan. Multi-Labelled Value Networks for Computer Go. *ArXiv e-prints*, May 2017.

[19] Yu Yamaguchi. AQ. `https://github.com/ymgaq/AQ`, 2018.

[20] Qinsong Zeng, Jianchang Zhang, Zhanpeng Zeng, Yongsheng Li, Ming Chen, and Sifan Liu. Phoenixgo. `https://github.com/Tencent/PhoenixGo`, 2018.