Review of AlphaGo Research Papers

Oleg Medvedev

December, 2017

Abstract

Review of two research papers [1] and [2] authored by Google DeepMind, describing the evolution of their AlphaGo engine.

Problem Statement

Until recently the game of Go remained as one of the only major classical games where expert human players remained undefeated against AI. Similar to chess the exhaustive search of actions space in Go is not feasible due to both the large breadth and depth of the game. Furthermore while the chess was principally “solved” by DeepBlue in 1996 Go due to it’s larger action space (both due to larger depth and breadth) was believed to be at least a decade away.

Prior to DeepMind’s work the two primary techniques used were the position evaluation to reduce breadth and Monte Carlo tree search (MCTS) for optimum policy to reduce depth. Both were sufficient to create Go agents able to play adequately at strong human amateur level and easily defeated by professional Go players.

Novel Techniques

In it’s first paper [1] authors utilized deep convolutional neural networks to create three networks for position evaluation and for prediction of the probability of the next move and one shallow network for fast policy rollout calculations. Please see the descriptions below.

Supervised Learning (SL) policy network

It is a 13-layer neural network trained on 30 million of positions from KGS Go Server. The goal is to be able to predict the human expert move in the given position. After training this model managed to achieve the 57% accuracy using all input features (total of 48, shown below) or 55.7% using only raw board position, which is a dramatic improvement over 44.4% of prior existing models.

Rollout policy

It is a shallow version of SL policy network, consisting of only linear softmax layer. It achieved a much lower accuracy of 24.2%, however was three orders faster than SL policy network (2μs vs 3ms). It used more features than SL policy network, see below.

Reinforcement Learning (RL) policy network

This network was used in the 2nd stage of training pipeline. Its structure is identical to SL network. It was trained by playing games between randomly chosen different iterations of itself to further improve the SL network. The games were played to the terminal state and then result was back propagated to modify weight

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Extended Data Table 2: **Input features for neural networks.** Feature planes used by the policy network (all but last feature) and value network (all features).

Figure 1: Network features

Feature	# of patterns	Description
Response	1	Whether move matches one or more response features
Save atari	1	Move saves stone(s) from capture
Neighbour	8	Move is 8-connected to previous move
Nakade	8192	Move matches a *nakade* pattern at captured stone
Response pattern	32207	Move matches 12-point diamond pattern near previous move
Non-response pattern	69338	Move matches 3×3 pattern around move
Self-atari	1	Move allows stones to be captured
Last move distance	34	Manhattan distance to previous two moves
Non-response pattern	32207	Move matches 12-point diamond pattern centred around move

Extended Data Table 4: **Input features for rollout and tree policy.** Features used by the roll-out policy (first set) and tree policy (first and second set). Patterns are based on stone colour (black/white/empy) and liberties $(1, 2, \geq 3)$ at each intersection of the pattern.

Figure 2: Rollout policy

correspondingly. After training the RL policy network achieved 80% win rate against SL network and 85% win-rate against some of the available open-source Go programs.

Value network

Value network was used to perform a position evaluation, i.e. provide a +1 or -1 value for the current state. Once again it used the similar structure as policy networks. In order to train it without the overfitting (as it appeared to memorize all KGS games) authors generated a new self-play data set of 30 million positions. The MSE was around 0.22-0.23 for both training and test datasets. This model allows evaluating position with accuracy similar to Monte Carlo rollouts, but with 15,000 times less computation.

Combining everything

AlphaGo uses four described networks in standard MCTS algorithm (see image below). During the *selection* phase the we select the edge of tree with the highest action value based on policy network. Once we reach the leaf node it may be expanded and is evaluated in two ways - by running the rollout to the end with the fast rollout policy and by using the value network. Both estimates are then mixed with some weight. The obtained result is then back propagated to all edges on the path. It is interesting that at least in the first games of AlphaGo with human experts the weaker SL policy was used instead of stronger one RL, as RL network was more focused on providing a single move, instead of giving a variety of promising moves and thus limiting exploration during MCTS.

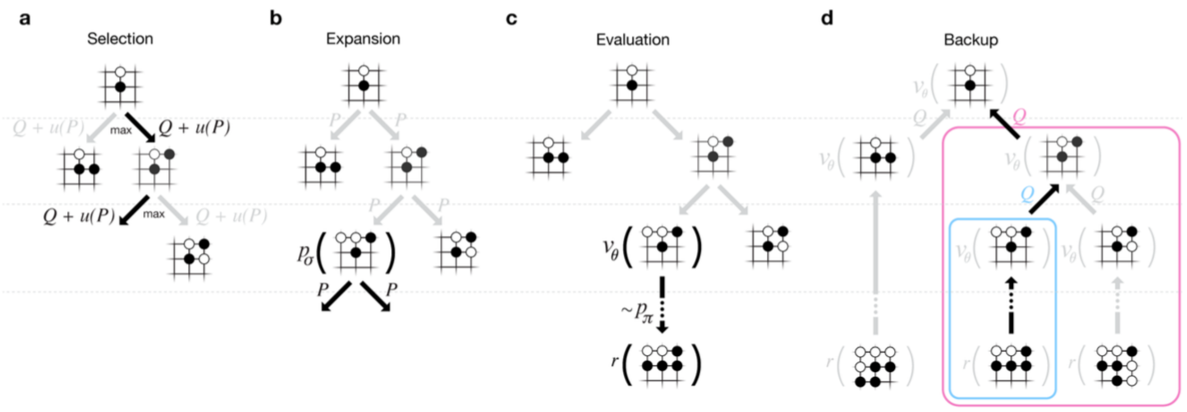

Figure 3: **Monte-Carlo tree search in *AlphaGo*.** **a** Each simulation traverses the tree by selecting the edge with maximum action-value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b** The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c** At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r. **d** Action-values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action.

Figure 3: Monte Carlo tree search

Results

Single-machine AlphaGo has 99.8% success rate against available Go programs, and is at least few dans stronger. At the time of the publication the described model managed to win a 5 game match (5:0) with 2 professional dan (2p) player on a full board without handicap, which was never achieved before. It is known that shortly after the similar model with minor modifications managed to win 4 games and the overall match against one of the strongest 9p human player.

Overall, it may be concluded that authors combined several previously known techniques (CNN, RL, MCTS) into an effective framework, which managed to provide a step change in performance.

AlphaGo Zero

In October, 2017 authors released a new paper [2], describing the continuation of their work. They developed a new agent, which they called AlphaGo Zero, which not only defeated the previous version described earlier, but did so without any prior human knowledge, i.e. *tabula rasa*. Let's review the main differences vs the paper described above.

The first difference is that authors combined the SL-policy and value network into one (in fact they already had similar architecture), which predicts both the probability of the next move and the current state of the board from 19x19 board and 17 features for a given board grid. These features are essentially just tracking whether there is a stone (and its color) in the given grid and the history for the given grid for the past 7 moves, which is important to treat the Ko rule properly.

The second and much more important difference is the reinforcement learning part. Authors still use MCTS to output probabilities π for each move at given state. They note that these probabilities are much stronger than probabilities predicted by policy network p, so MCTS may be considered as a powerful RL tool by itself. Thus they use MCTS as a self-play algorithm which predicts the best move π based on neural network parameters and given state (the move is selected based the exponentiated visit count for each move). Neural network is initialized to random parameters and then a self-game is played at every iteration. During a self-game MCTS uses the neural network weights from previous iteration to predict the best moves π. Once the final result of the game is reaches, whether it's actual win or loss, or surpassing a length threshold or resignation threshold the neural network weights are updated to minimize the error between the prediction of the value network and actual game result and also to minimize the deviation between MCTS predicted moves π and policy network predictions p.

The results are shown on the image below extracted from author's paper. One may see that approximately only after 3 days of training AlphaGo Zero surpasses the AlphaGo Lee version which defeated one of the strongest human players Lee Sedol. Note that that version was trained on 30 million of positions and AlphaGo Zero started from zero prior knowledge. And after about 30 days of training it defeated AlphaGo Master which was still based on some prior knowledge and defeated a number of best human players while playing online. Truly remarkable result.

As a conclusion let me copy the last sentence from author's paper: *Humankind has accumulated Go knowledge from millions of games played over thousands of years, collectively distilled into patterns, prov-erbs and books. In the space of a few days, starting tabula rasa, AlphaGo Zero was able to rediscover much of this Go knowledge, as well as novel strategies that provide new insights into the oldest of games.*

References

[1] "Mastering the game of Go with deep neural networks and tree search", doi:10.1038/nature16961

[2] "Mastering the game of Go without human knowledge", doi:10.1038/nature24270

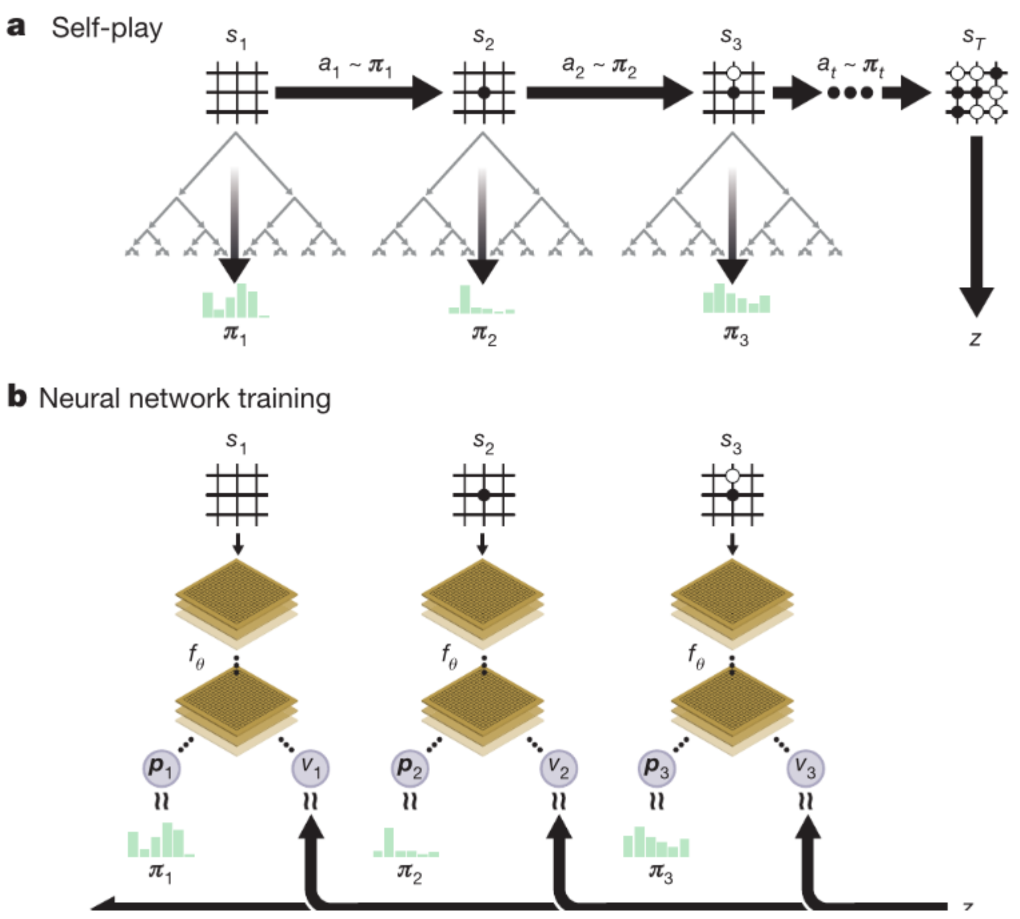

Figure 1 | Self-play reinforcement learning in AlphaGo Zero. a, The program plays a game $s_1, ..., s_T$ against itself. In each position s_t, an MCTS α_θ is executed (see Fig. 2) using the latest neural network f_θ. Moves are selected according to the search probabilities computed by the MCTS, $a_t \sim \boldsymbol{\pi}_t$. The terminal position s_T is scored according to the rules of the game to compute the game winner z. **b**, Neural network training in AlphaGo Zero. The neural network takes the raw board position s_t as its input, passes it through many convolutional layers with parameters θ, and outputs both a vector $\boldsymbol{p}_t$, representing a probability distribution over moves, and a scalar value v_t, representing the probability of the current player winning in position s_t. The neural network parameters θ are updated to maximize the similarity of the policy vector $\boldsymbol{p}_t$ to the search probabilities $\boldsymbol{\pi}_t$, and to minimize the error between the predicted winner v_t and the game winner z (see equation (1)). The new parameters are used in the next iteration of self-play as in **a**.

Figure 4: AlphaGo zero reinforcement learning algorithm

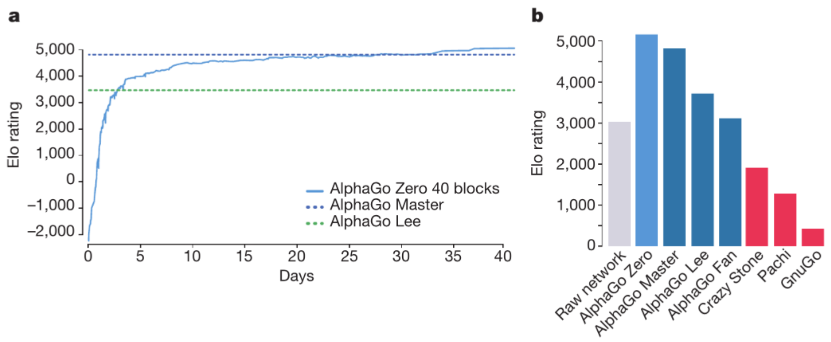

Figure 6 | Performance of AlphaGo Zero. a, Learning curve for AlphaGo Zero using a larger 40-block residual network over 40 days. The plot shows the performance of each player α_{θ_i} from each iteration *i* of our reinforcement learning algorithm. Elo ratings were computed from evaluation games between different players, using 0.4 s per search (see Methods). **b**, Final performance of AlphaGo Zero. AlphaGo Zero was trained for 40 days using a 40-block residual neural network. The plot shows the results of a tournament between: AlphaGo Zero, AlphaGo Master (defeated top human professionals 60–0 in online games), AlphaGo Lee (defeated Lee Sedol), AlphaGo Fan (defeated Fan Hui), as well as previous Go programs Crazy Stone, Pachi and GnuGo. Each program was given 5 s of thinking time per move. AlphaGo Zero and AlphaGo Master played on a single machine on the Google Cloud; AlphaGo Fan and AlphaGo Lee were distributed over many machines. The raw neural network from AlphaGo Zero is also included, which directly selects the move *a* with maximum probability p_a, without using MCTS. Programs were evaluated on an Elo scale[25]: a 200-point gap corresponds to a 75% probability of winning.

Figure 5: AlphaGo Zero results