Artificial Intelligence for Games
Seminar Report

Mastering the game of Go with deep neural networks and tree search (Silver et al., 2016)

by

Florian Brunner
3540651
sc223@uni-heidelberg.de

Supervised by Prof. Dr. Ullrich Köthe

Heidelberg University

Submitted:
August 8, 2019

Contents

1. Introduction

Go is a board game which is very popular in large parts of Asia. It was invented around 2,500 years ago in ancient China and some players have even dedicated their whole life to mastering it. It requires strong intuition and abstract thinking to play. Hence, it was believed that only humans would be good at playing Go. Only until a couple of years ago, AI experts believed Go would be unsolvable for some more decades due to its complexity which originates from its enormous search space and the difficulty of evaluating board positions and moves. Because of that, it was also considered as one of the "grand challenges" of artificial intelligence [6].

In October 2015, DeepMind's AlphaGo defeated the European champion, Fan Hui, a *2 dan* professional, in formal five-game match, which was the first time a computer Go program has defeated a human professional player (without handicap) in a full game of Go [6]. In March 2016, AlphaGo even defeated the world champion, Lee Sedol, a *9 dan* professional, the highest rank a player can achieve, at the Google DeepMind challenge, which was an event held in Seoul, South Korea [4]. This was a huge milestone in artificial intelligence research.

2. The Game of Go

2.1 Basic Rules

Go is a perfect information, zero-sum game. A full-sized game of Go is played on a 19x19 board. Stones are placed on the intersections in alternating order. Black starts by putting a stone on a free intersection. Each adjacent, empty intersection is called *liberty*. Adjacent stones of the same color form *group*. If a stone, or a group of stones only has one liberty left, it is said to be in *atari*. As soon as one stone or group has no liberties, that is, they are completely surrounded by the opponent's stones, they are captured and become prisoners. In the end, both players count the vacant points inside their territory, add one point for each prisoner, and the player who has more points wins the game. Because Black has a natural advantage of playing first, White is compensated with additional points called *komi*. In tournaments, this komi is usually set to 7.5 to avoid draws. [1]

2.2 Complexity

Like for all perfect information games (such as chess and checkers), the outcome solely depends on the strategy of both players, and there exists an optimal value function $v^*(s)$, which determines the outcome of the game from every board position, given that both players play perfectly. To solve such games, this optimal value function could be computed recursively in a search tree containing b^d possible

sequences of moves. The problem with solving Go, however, is its board size: there are 10^{170} possible game states. In chess for comparison, there are 10^{43}. The average branching factor b in Go is approximately 250, which means that in the game tree, in each state, there are on average 250 different actions to take. The depth d, i.e. the game length, is approximately 150 in Go. (The game tree of chess for comparison has $b \approx 35$, $d \approx 80$.) Hence, exhaustive search is highly infeasible, which was the main reason why people believed Go would be unsolvable for some more decades. However, the effective search space can be reduced by two general principles: 1) reducing the game tree's depth, and 2) reducing its breadth. The first can be accomplished by truncating the search tree at a state s and replace the subtree by an approximate value function. That means, from a given state, we try to estimate the possible outcome. This approach already worked for chess and checkers, however Go was considered to be "too complex" for this approach to work. The breadth of the tree can be reduced by sampling actions from a policy $p(a \mid s)$. A policy is a probability distribution over possible moves a in a given state s. Then, one could select the move with the highest probability because it yields the highest chance of winning. [6]

3. Methodology

This chapter covers the building blocks of AlphaGo as presented in the original paper from 2016 by David Silver and his team at Google DeepMind. [6]

3.1 Rollout Policy

The rollout policy $p_\pi(a \mid s)$ (also called *fast policy*) is used to rapidly sample actions during the Monte Carlo rollouts. It is a linear softmax policy and its weights π have been trained from 8 million board positions from games between human expert players obtained from the KGS Go server to maximize log likelihood by stochastic gradient descent. This policy is based on local pattern-based features consisting of both "response" patterns around the previous move a that led to state s, and "non-response" patterns around the candidate move a in state s. Additionally, common-sense Go rules were encoded into this policy by a small number of handcrafted local features. This policy achieved an accuracy of 24.2% (i.e. selecting the same move as the human expert did), using just $2\mu s$ to select an action.

3.2 Supervised Learning Policy Network

The supervised learning policy network $p_\sigma(a \mid s)$ is a 13-layer convolutional neural network which was trained from 30 million positions from games between expert human players. This network takes the current board state as a 19x19x48 batch as

input and outputs a probability distribution over all legal moves a. The network has been trained on randomly sampled state-action pairs (s, a), using stochastic gradient ascent to maximize the log likelihood of selecting move a in state s

$$\Delta\sigma \propto \frac{\partial \log p_\sigma(a \mid s)}{\partial \sigma} \tag{3.1}$$

This network predicted expert moves with an accuracy of 57.0% (55.7% using only the raw board position and the move history). This was a great improvement compared to the state-of-the-art from other research groups where an accuracy of 44.4% was achieved. The authors showed that small improvements in accuracy drastically improves the playing strength, i.e. the win rate, as shown in Figure 3.1. Policy networks with 128, 192, 256, and 384 convolutional filters per layer were evaluated during training. The plot shows the win rate of AlphaGo using the respective policy network against the match version of AlphaGo. The SL policy network requires 3*ms* though to select an action. Therefore, it is also referred to as the *slow policy*.

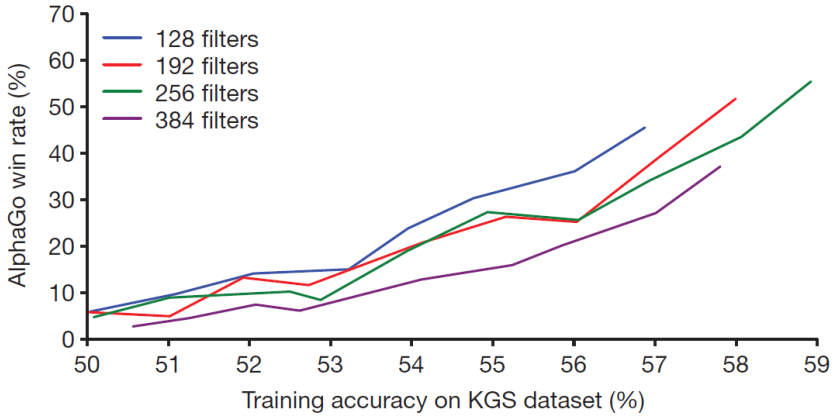

Figure 3.1: Training accuracy on KGS dataset

3.3 Reinforcement Learning Policy Network

In the next stage of the training pipeline, the authors tried to improve the policy network by policy gradient reinforcement learning. Their intuition was that if they train on human expert positions, they can only try to imitate the human, but they will never be able to surpass him. So, the bias has now been shifted towards actually winning games, i.e. predicting winning moves rather than predicting human expert moves. The reinforcement learning policy network p_ρ has an identical structure to the supervised learning policy network, and its weights ρ are initialized to the same values, $\rho = \sigma$. Games have been played between the current policy network and a randomly selected previous iteration of the policy network to prevent overfitting to the current policy. Games are played until the end. The outcome z_t is the terminal reward $r(s_T)$ at the end of the game, and it equals 1 for a win, and -1 for a loss (from the current player's perspective). It is 0 at each non-terminal time step. At

each time step, weights are updated by stochastic gradient ascent in the direction that maximizes the expected outcome

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t \mid s_t)}{\partial \rho} z_t \tag{3.2}$$

This RL policy network won more than 80% of games against the SL policy network, and 85% of games against Pachi, back then, one of the strongest open-source Go programs solely relying on MCTS.

3.4 Value Network

The authors trained a value network v_θ which is used to evaluate board positions, i.e. estimating a value function $v^p(s)$ that predicts the outcome from position s of games played by using policy p for both players

$$v^p(s) = \mathbb{E}[z_t \mid s_t = s, a_{t\ldots T} \sim p] \tag{3.3}$$

This value network is similar to an evaluation function as used in DeepBlue but learned rather than designed [2]. The network has a similar architecture to the policy network, however its output is a single prediction instead of a probability distribution. The value network has been trained by regression on state-outcome pairs, using stochastic gradient descent to minimize the MSE between predicted

value $v_\theta(s)$ and outcome z

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial \theta}(z - v_\theta(s)) \tag{3.4}$$

A self-play data set has been generated consisting of 30 million distinct positions, each sampled from a different game, by letting the RL policy play against itself. This value network was consistently more accurate compared to the rollout policy. Furthermore, it also approached the accuracy of Monte Carlo rollouts using the RL policy network, but using four orders of magnitude less computations.

3.5 Searching with Policy and Value Networks

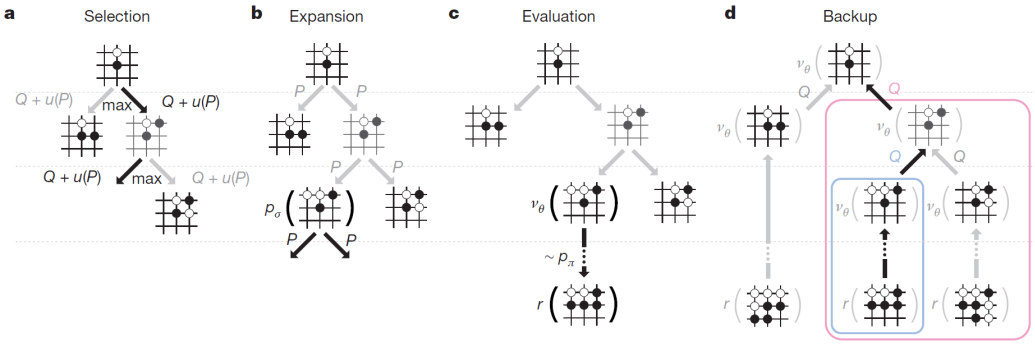

Figure 3.2: Monte Carlo tree search in AlphaGo

The authors implemented a modified version of the basic MCTS called *asynchronous policy and value Monte Carlo tree search (APV-MCTS)* (see Figure 3.2). The selection step works similar to basic MCTS, however they changed how leaf nodes are expanded and how action edges are evaluated. Instead of using stored

action values to select an unexplored edge, AlphaGo's MCTS uses the probabilities supplied by the SL policy network [8]. They chose the SL policy network over the RL policy network here because it offers a variety of good (human) moves, while the RL policy network is trained to predict the single best move. Each edge (s,a) stores an action value $Q(s,a)$, visit count $N(s,a)$, and prior probability $P(s,a)$. At each time step, actions a_t are selected from state s_t

$$a_t = \underset{a}{\operatorname{argmax}}(Q(s_t,a) + u(s_t,a)) \tag{3.5}$$

so as to maximize action value plus a bonus (to encourage exploration)

$$u(s,a) \propto \frac{P(s,a)}{1+N(s,a)} \tag{3.6}$$

For evaluation, APV-MCTS runs the fast policy to simulate the game until the end (actions for both players are selected by this policy) and also applies the value network on the current board state. The result of the simulation and the value network are combined using a mixing parameter λ into a leaf evaluation $V(s_L)$ and backpropagated to the root node

$$V(s_L) = (1-\lambda)v_\theta(s_L) + \lambda z_L \tag{3.7}$$

After the simulation is finished, the action value and visit count of each traversed edge is updated ($1(s,a,i)$ is 1 if edge (s,a) was traversed in the ith simulation)

$$\begin{aligned} N(s,a) &= \sum_{i=1}^{n} 1(s,a,i) \\ Q(s,a) &= \frac{1}{N(s,a)} \sum_{i=1}^{n} 1(s,a,i) V(s_L^i) \end{aligned} \tag{3.8}$$

4. Evaluation

For evaluating AlphaGo's playing strength, DeepMind held an internal tournament, where AlphaGo played against several other Go programs. Out of 495 games, AlphaGo only lost a single game. The playing strength of AlphaGo, indicated by the Elo rating, by far exceeded those of all previous Go programs. Also, the authors show that choosing $\lambda = 0.5$ for leaf evaluation in MCTS performs best, indicating that the combination of the value network with rollouts was particularly important for AlphaGo's success. Namely, these evaluation methods complemented each other: the value network evaluated the RL policy which is too slow for live play, while the rollouts add precision to the value network's evaluations by using the weaker but much faster rollout policy. The authors also implemented a distributed version of AlphaGo, which won 77% of the games against single-machine AlphaGo and 100% against the other Go programs.

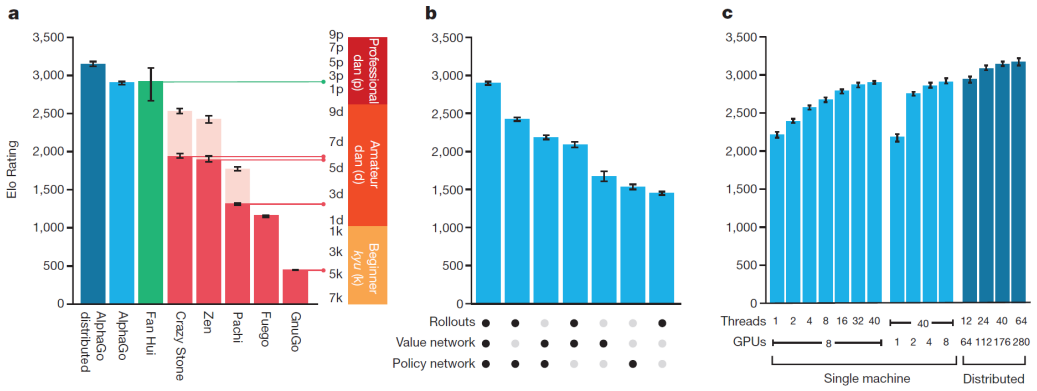

Figure 4.1: Tournament evaluation of AlphaGo
a) Elo rating of AlphaGo compared to different Go programs, **b)** Performance of AlphaGo on a single machine for different combinations of components, **c)** comparison of single-machine and distributed AlphaGo

5. Conclusion

In this seminar report, Google DeepMind's AI AlphaGo is presented. AlphaGo has been the first computer program that has defeated a human professional player in the full-sized game of Go. After covering the basic rules of Go, we first had a look at how computers solve it by traversing the game tree. Then, we presented AlphaGo's architecture and its individual components, followed by how AlphaGo combines these in an MCTS algorithm that selects actions by lookahead search. Finally, playing strength evaluations performed by the DeepMind team were shown and discussed.

Interestingly, other researchers say that AlphaGo is not a breakthrough technology but rather a consequence of the recent research in computer Go because all the methods that AlphaGo uses have been known and developed for a long while [3]. Moreover, the DeepMind team achieved even superhuman performance with AlphaGo's sucessors AlphaGo Zero [7] and AlphaZero [5]. Besides, one of AlphaGo's greatest advantages is that it uses domain-independent algorithms. It shows us, that complex problems like Go, could be solved by state-of-the-art techniques. Furthermore, deep learning has been successfully applied to several other fields like computer vision, speech recognition, and bioinformatics. So perhaps, the pipeline introduced in AlphaGo bears the potential to be also applied to other domains with minor modifications [9].

Bibliography

[1] British Go Association. *An Introduction to Go*. 2017. URL: https://www.britgo.org/intro.

[2] Henry. *A brief introduction of AlphaGo and Deep Learning: How it works*. Oct. 2017. URL: https://medium.com/@kinwo/a-brief-introduction-of-alphago-and-deep-learning-how-it-works-76e23f82fe99.

[3] Steffen Hölldobler, Sibylle Möhle, and Anna Tigunova. "Lessons Learned from AlphaGo". In: June 2017.

[4] BBC News. *Artificial intelligence: Google's AlphaGo beats Go master Lee Se-dol*. Mar. 2016. URL: https://www.bbc.com/news/technology-35785875.

[5] David Silver et al. "A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play". In: *Science* 362.6419 (2018), pp. 1140–1144.

[6] David Silver et al. "Mastering the game of Go with deep neural networks and tree search". In: *nature* 529.7587 (2016), p. 484.

[7] David Silver et al. "Mastering the game of go without human knowledge". In: *Nature* 550.7676 (2017), p. 354.

[8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018, pp. 444–447. URL: http://incompleteideas.net/book/the-book-2nd.html.

[9] Fei-Yue Wang et al. "Where does AlphaGo go: From church-turing thesis to AlphaGo thesis and beyond". In: *IEEE/CAA Journal of Automatica Sinica* 3.2 (2016), pp. 113–120.