Monte Carlo Tree Search and Neural Networks in Small Board Go†

Meijke Balay[1], Kyle Chan[1], Duke Hong[1]

***Abstract*— Deep neural networks and their variants constitute the most versatile class of machine learning methods known today and have been successfully applied in numerous domains, including computer vision, speech recognition, natural language processing, and game theory. Deep neural networks have recently achieved superhuman performance in games with large game-trees like Chess and Go when combined with Monte Carlo Tree Search and reinforcement learning. In this work, we briefly illustrate the mathematical foundations of these ideas and apply them to create a program that learns how to intelligently play 9×9 Go.**

I. Introduction

It can be proved that every finite 2-player sequential game with perfect information has a subgame perfect equilibrium [2][33]. In other words, popular 2-player board games such as tic-tac-toe, Connect-4, checkers, chess, and Go are all theoretically "solved" in the sense that in any given position, there exists at least one optimal move. While the optimal solution for simple games such as tic-tac-toe may be found relatively quickly via exhaustive search, this approach is computationally intractable for more complex games such as Go where the number of legal positions is roughly 2.08^{170} [28]. Instead, given board position s we must find an approximation of the optimal solution by estimating the optimal value function $v^*(s)$, which is defined as the outcome of a game starting from position s assuming perfect play by all players. Since by von Neumann's minimax theorem $v^*(s)$ is unique for 2-player zero-sum games [29], if we can recover $v^*(s)$ exactly, we can recover the optimal solution. One technique for approximating $v^*(s)$ is Monte Carlo Tree Search (MCTS) which computes an estimate of $v^*(s)$ by randomly simulating $k < \infty$ games starting from s and at each iteration reweighting each legal turn according to the outcome of the simulated game [7]. Moves resulting in winning outcomes receive more weight whereas losing turns receive less weight; the move with the most weight at the end of the k simulated games is ultimately played (ties are broken arbitrarily). Exactly how weights are distributed at each iteration is governed by a search heuristic.

Although MCTS converges to the optimal value function $v^*(s)$ as $k \to \infty$ [15], this convergence is impractically slow unless the algorithm is equipped with an effective search heuristic that efficiently guides the expansion of the search tree towards optimal moves. Several search heuristics exist for Go, typically in the form of hand-crafted expert advice [19][5][10], but the most powerful search policies are those obtained from neural networks, particularly convolutional neural networks (CNNs) [16], and reinforcement learning [23][25]. The neural network policy is trained by predicting moves in games played by human experts while the reinforcement learning policy is trained by maximizing winning percentage through self-play.

In this work, we mimic the work of DeepMind [23] on 9×9 Go (instead of a full 19×19 board) by synthesizing the neural network and reinforcement learning approaches into a single policy network. That is, we will first train our policy network on human expert games, and then refine the policy network further by switching to training via self-play in order to orient the policy network towards the goal of maximizing the number of wins instead of the number of correctly predicted moves. Furthermore, an additional neural network (henceforth known as the value network) with the same architecture as the policy network will be trained to evaluate board positions. The two networks will then be incorporated into the search heuristic for MCTS on 9×9 Go. Ideally, this search heuristic will outperform the raw human expert heuristics in GNU Go [9].

A. Organization

In Section II, we provide brief overviews of MCTS, CNNs, and Markov decision processes (MDPs) in reinforcement learning. In Section III, we consolidate the techniques discussed in Section I and Section II into a 9×9 Go-playing program and analyze its performance against GNU Go. Finally, in Section IV, we offer some conclusions.

II. Mathematical Foundations

A. Markov Decision Processes

A Markov Decision Process (MDP) is a simple model for an agent interacting with an environment. It consists of a set of states $\mathcal{S}$, a set of actions $\mathcal{A}(s)$ for each $s \in \mathcal{S}$, and a set of real-valued rewards $\mathcal{R}$. At each time step $t = 0, 1, \ldots$ the agent is in a state s_t and selects an action a_t for which it receives a (possibly random) reward $r_t \in \mathcal{R}$ and ends in state s_{t+1}. For our purposes we consider MDPs with a finite number of states, actions, and rewards, and finite lookahead horizon $t \leq T$. A sequence of these actions $(s_0, a_0, s_1, \ldots, s_T)$ is called a *trajectory*, and has a total reward $R_T = \sum_{t=0}^{T-1} r_t$.

The state-transition probability $P(s'|s, a)$ is the probability that taking an action a in state s puts the environment in state s'. We require these probabilities to satisfy the Markov property, i.e. they only depend on the previous state and action (s, a). Thus we can view a finite MDP as a generalized Markov chain.

[1] Department of Mathematics, University of California, Los Angeles, 520 Portola Plaza, Los Angeles, CA 90095, USA

* Equal contribution

A *policy* π gives for each state s a probability distribution over the possible actions $\mathcal{A}(s)$. The agent follows a policy π by sampling actions from this distribution at each time step, which we denote $a \sim \pi$. Corresponding to each policy is a value function $v^\pi : \mathcal{S} \to \mathbb{R}$ defined by

$$v^\pi(s) = \mathbb{E}_{a\sim\pi}[R_T | s_t = s],$$

that is, the expected reward over all trajectories starting from s and following π. A related function is the *action value*

$$Q^\pi(s,a) = \mathbb{E}_{a\sim\pi}[R_T | s_t = s, a_t = a].$$

We thus seek a policy π^* with corresponding value function v^* that is optimal in the sense that $v^*(s) \geq v^\pi(s)$ for all policies π and all $s \in \mathcal{S}$.

A zero-sum, 2-player, perfect-information game [30] like Go can be modeled as a simple MDP[1] where the states are board states, the actions are legal moves, and the state-transitions are deterministic. The terminal time T is the end of the game when the score determines the winner. A trajectory $(s_0, a_0, \ldots, s_T)$ has rewards $r_t = 0$ for $0 \leq t < T$ and a terminal reward r_T which is $+1$ if the current player in state s_0 won the game, and -1 if they lost.

An optimal deterministic policy can be obtained recursively with the minimax algorithm; $v^*(s_T) = r_T$ and $v^*(s_t) = -\max_{a\in\mathcal{A}(s_t)} v^*(s_{t+1})$ for $t < T$. One can then simply select moves with maximum value. However, this approach quickly becomes infeasible for large game trees.

B. Neural Networks

In order to approximate an optimal policy, we can use supervised learning to train a neural network on examples of optimal actions. We can view this as a multi-label classification problem, where the inputs are states s and the labels are optimal actions $a \in \mathcal{A}(s)$. Thus if π_θ is the supervised learning policy with parameters θ, we minimize the cross entropy loss

$$E(s,a) = -\log(\pi_\theta(a|s))$$

using stochastic gradient descent (SGD). The drawback of this approach is having to gather a large dataset of optimal actions or near-optimal actions, which in many cases are unavailable.

1) Policy Gradient Methods: A second approach, known as the policy gradient method, is to update the policy parameters to maximize the expected total reward over all trajectories. More precisely, the reward for a policy π_θ with value function v^θ is defined as

$$J(\theta) = \sum_{s\in\mathcal{S}} v^\theta(s) = \sum_{s\in\mathcal{S}} d^\pi(s) \sum_{a\in\mathcal{A}(s)} Q^\pi(s,a)\pi_\theta(a|s)$$

where d^π is the stationary distribution of the Markov chain with transition probabilities given by π_θ. In order to do gradient ascent, we need an efficient way to compute the gradient of J in terms of the gradient of π_θ. The policy gradient theorem [26, Section 13.6] shows that

$$\begin{aligned}\nabla_\theta J(\theta) &\propto \mathbb{E}_{a\sim\pi}[Q^\pi(s,a)\nabla_\theta \log \pi_\theta(a|s)] \\ &= \mathbb{E}_{a\sim\pi}[R_T \nabla_\theta \log \pi_\theta(a|s)].\end{aligned}$$

We can therefore approximate the gradient by sampling trajectories and computing the bracketed quantity. This is the basis of the basic reinforcement learning algorithm:

Algorithm 1 REINFORCE

Require: learning rate α
  for trajectory $(s_0, a_0, \ldots, a_{T-1}, s_T) \sim \pi_\theta$ **do**
    for $t = 0$ to $T-1$ **do**
      $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(a_t|s_t) \cdot R_T$
    end for
  end for

In practice, we can update the parameters with SGD rather than the full gradient. Once the neural network learns a policy, we can train a corresponding value neural network with supervised learning by generating state-reward data from trajectories following the policy.

2) Convolutional Neural Networks: For simpler MDPs such as CartPole [4] or Backgammon [27], fully connected neural networks with few hidden layers are sufficient to learn an optimal policy; however the board states of Go contain complex 2D patterns which suggests deeper networks with convolutional layers are necessary.

We recall that a convolutional layer with a size $k \times k$ kernel has a set of weights $w_{i,j}$, $0 \leq i, j < k$ that operates on an $n \times n$ input of activations $a_{i,j}$ by computing the 2D discrete convolution $w * a$, whose i,jth term is

$$\sum_{\ell=0}^{k} \sum_{m=0}^{k} w_{\ell,m} a_{i+\ell,j+m}.$$

A shared bias b is added to each term and a sigmoidal activation function σ is then applied, outputting a $(n-k+1) \times (n-k+1)$ plane $\sigma(w * a + b)$.

Convolutional layers are commonly used in computer vision because the $k \times k$ plane, also known as the "local receptive field," detects features of the input image in a translation-invariant manner. The complete convolutional layer will typically have several different *filters*, each detecting a different feature; for this reason the output planes are called feature maps. Feature detection is applicable to Go because local "shapes" are important in evaluating moves and board positions (see Fig. 2).

We note that convolutional networks are *a fortiori* universal function approximators like linear layers [8], however they are also naturally universal approximators for more general equivariant maps [32].

C. Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) selects actions in MDPs by estimating action values with random simulations. The search tree is built from the root state s_0 with several *rollouts*, which proceed in four stages:

[1]there are rare positions in Go called "super-kos" where the Markov property would be violated, but for our purposes we ignore these.

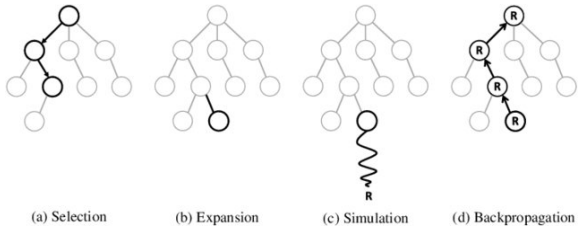


Fig. 1. Stages of MCTS rollout; diagram from [13]

1) *Selection*: From the root node s_0, select actions according to a "tree policy" until a leaf node is reached.
2) *Expansion*: Expand the tree to include one or more of the leaf node's children.
3) *Simulation*: Evaluate the leaf node by simulating a trajectory and finding its total reward.
4) *Backpropagation*: Propagate the reward from the simulation back to the root.

Each edge (s, a) in the search tree stores statistics $N(s, a)$ and $W(s, a)$ – the number of times the edge was visited during rollouts and the accumulated rewards from the rollouts, respectively. After several rollouts, one selects $\arg\max_a N(s_0, a)$, i.e. the action from the root with the most visits. We will call an agent that selects actions according to this process will be called an ***MCTS agent***.

Given computational constraints on the number of rollouts, the performance of the MCTS agent depends closely on the tree policy used in the selection stage, as well as the method of simulation. In the simplest case, the tree policy selects actions with greatest simulated action value

$$Q(s, a) = \frac{W(s, a)}{N(s, a)},$$

and simulates uniformly random actions. The tree policy can also use domain knowledge to bias towards heuristically promising actions.

A well known tree policy is UCT (Upper Confidence bound for Trees), which uses the UCB1 algorithm [3], treating each selection as a mutli-armed bandit problem. The UCT algorithm selects actions with the largest "upper confidence bound"

$$Q(s, a) + c \cdot \frac{\sqrt{\log T(s)}}{N(s, a)} \tag{1}$$

where $T(s) = \sum_a N(s, a)$ is the total number of visits to s and $c > 0$ is a constant. The second term decays with $N(s, a)$, which encourages exploration of other actions. The authors of [15] prove that MCTS with UCT selection is asymptotically optimal. More precisely, if n is the number of rollouts, the probability of selecting a sub-optimal action at the root converges to zero in polynomial time as $n \to \infty$. Moreover, the difference between the optimal value and the expected reward of the MCTS agent (a quantity called the *regret*) is $O\left(\frac{b \log n}{n}\right)$, where b is the number of children of the root node.

For large b such as in Go, this bound can still be insufficient. The PUCB (Predictor + UCB) algorithm [22] shows that one can improve the regret bound by introducing a "predictor" penalty to (1). In particular, if $P(s, a)$ estimates the probability a is optimal, then the worst-case regret improves by a factor of

$$\frac{1}{b}\left(\sum_a \sqrt{P(s, a)}\right)^2.$$

Using the ideas from UCT and PUCB, the authors of [23] consider the following PUCT variant of the upper confidence bound:

$$Q(s, a) + c \cdot P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)}. \tag{2}$$

In AlphaGo, the probability $P(s, a)$ was provided by a policy neural network $\pi(a|s)$, and the action value was estimated using both a value neural network and simulation rewards. Specifically, during evaluation of the leaf node in step 3) of the rollout, the state is evaluated according to the value network, and backpropagated with the simulation reward in step 4). The edges thus store an extra statistic $V(s, a)$, the accumulated value network evaluations from the descendants of s. The action value is estimated as the weighted sum

$$Q(s, a) = \frac{\lambda W(s, a) + (1 - \lambda) V(s, a)}{N(s, a)}. \tag{3}$$

Although no detailed theoretical analysis of (2) has been done, the performance of AlphaGo and its successors, as well as empirical analysis in other games [18] show that the MCTS agent with PUCT consistently outperforms UCT.

III. Methods

We trained a policy network and value network with PyTorch [21] and used them to implement MCTS with PUCT (2) for 9×9 Go.

A. Policy Network

1) Supervised Learning: We trained both a supervised learning policy π_{SL} and a reinforcement learning policy π_{RL}, with identical neural net architectures. The input to the network is a representation of the game state consisting of 27 9×9 input features. We then apply one layer consisting of

1) Zero padding on each 9×9 plane to produce a 13×13 plane
2) A convolutional layer with kernel size 5×5, stride 1, and 128 9×9 feature maps.
3) A rectified linear unit

followed by 5 layers consisting of

1) Zero padding to produce 7×7 planes
2) A convolution layer with kernel size 3×3, stride 1, and 128 9×9 feature maps.
3) A rectified linear unit.

The last layer is a convolution with a 1×1 kernel and stride 1 with untied biases, i.e. different biases for each kernel position. The output is a single 9×9 plane. Finally, a softmax

Fig. 2. Convolutional layers learned to detect common shapes such as hane, ponnuki (diamond), keima (knight's move) and elephant's jump.

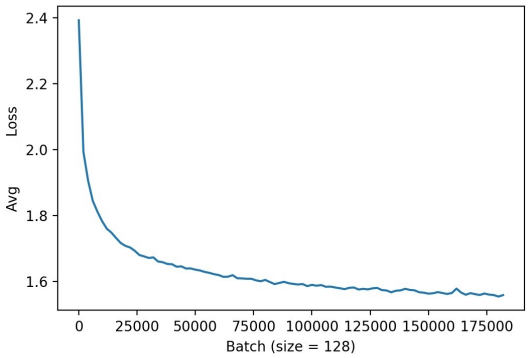


Fig. 3. π_{SL} Cross entropy loss

is applied to get a probability distribution over the board coordinates.

This architecture is a scaled down version of the networks used in state of the art programs that play 19×19 Go such as AlphaGo, Leela [20], and KataGo [31], which use between 20 and 40 convolutional layers and at least 256 feature maps per layer. We used smaller networks primarily due to computational constraints, but the reduced complexity of 9×9 Go suggests that a smaller networks may be sufficient.

To train π_{SL} we obtained 63,000 amateur games from the KGS Go Server [1]. The games were processed into pairs (s, a), where a is the move the human player chose in a board state s. We augmented the data by adding all pairs $(r(s), r(a))$, where $r \in D_4$ is one of the eight dihedral symmetries of the 9×9 board. The final dataset had $\approx 23,000,000$ state-action pairs.

During training we minimized the cross-entropy for mini-batches of 128 datapoints. We performed gradient descent using the Adam (Adaptive Moment Estimation) algorithm [14], a common variant of SGD that dynamically updates the learning rate based on estimates of the gradient's mean and variance. The learning rate was initialized to $\alpha = 10^{-3}$.

To improve training speed and stability we added decoupled weight decay regularization [17] and applied batch normalization [12] between each convolutional layer. One epoch took 3 hours with 8 CPUs and 1 Tesla P100 GPU on the Google Cloud Platform[2].

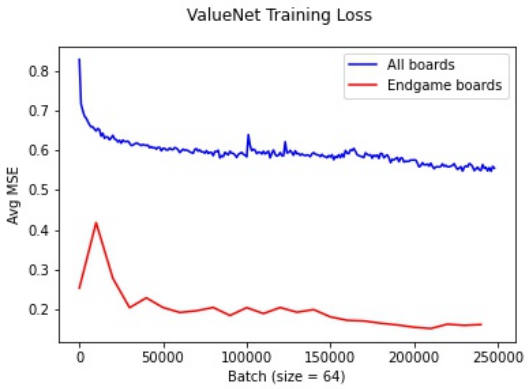
2) Reinforcement Learning: We implemented a variant of REINFORCE (see Algorithm 1) with rewards provided by the result of self-play. The parameters of the training policy π were first initialized to the parameters of π_{SL}. For each epoch, an opponent policy π_{opp} was initialized from the one of the previous policy parameters. 8 games are played in parallel between policies π and π_{opp}; π is the black player in four of the games, and white in the other four.

The parameters are updated with Adam after a batch of 16 games according to the cross entropy loss

One epoch consisted of 512 such updates. We trained the policy for 50 epochs for approximately 2 days. Although the final policy π_{RL} achieved an 80% winrate against π_{SL}, this part of the training was unstable and required close monitoring; the winrate of the training policy occasionally dropped to $< 10\%$ due to nonsensical moves being played. This could be due to instability of Adam, or more likely an artifact of our imperfect implementation.

B. Value Network

The value network v consists of the same input and convolutional layers as the policy network, with the addition of 2 fully connected layers with 81 and 64 neurons. Finally $\tanh$ is applied to obtain a value $v(s) \in (-1, 1)$.



To train the value network, we sample a trajectory from π_{RL} up to a randomly chosen time step t, then choose a uniform random legal move. From this board state s_{t+1}, π_{RL} finishes the game. We thus generated $\approx 1,000,000$ examples (s_{t+1}, r_T) where the reward is from the perspective of s_{t+1}. Using this dataset we did gradient descent with Adam using the MSE $|v(s_{t+1}) - r_T|^2$ for mini-batches of 64 boards.

[2]https://cloud.google.com/compute

C. Boke Go vs. GNU Go

We wrote a GTP (Go Text Protocol[3]) Engine using our value network and policy network π_{RL} in MCTS with PUCT, with constants $c = 4$ and $\lambda = \frac{1}{2}$ in (2) and (3). The simulations are done by sampling trajectories from π_{RL}. We named the engine "Boke Go.[4]" We evaluated the strength of Boke Go against an open source GTP Engine named GNU Go. GNU Go v3.8 uses MCTS with UCT, combined with hand-coded heuristics that evaluate influence and attack/defense status, as well as a library of joseki (common patterns) and fuseki (openings) [9]. It has achieved a 3-4 kyu rank on the 9×9 board on the Online Go Server and won the 2006 Computer Olympiad in 19×19 Go.

We played a traditional jubango (ten game match) between Boke Go and GNU Go. The rule set was Chinese, with 5.5 komi and time controls of 60 seconds per move. Boke Go won all ten games decisively (Examples shown in Figs. 4-7).

To upper bound the current strength of Boke, we note that it could not beat the first author, who is ranked 2 dan (4 ranks above 3 kyu). Qualitatively, Boke demonstrated good positional judgement and shape, but was weak in reading variations, especially in semeai (capturing races), where one must accurately read 10-20 moves ahead to avoid a fatal error. This is likely due to the comparatively small number of rollouts our implementation is currently capable of; most MCTS programs parallelize rollouts on mutliple processes or threads [6]. Moreover, AlphaGo uses a "fast rollout policy" based on local heuristics for simulations, whereas simulations with our π_{RL} is computationally expensive.

IV. Conclusions

The success of Boke Go vs GNU Go illustrates the power of deep learning combined with MCTS compared to previous techniques which focused on extensive input of human domain knowledge. In fact, the current strongest Go programs are based on AlphaGo Zero [24], a modification of AlphaGo that removes the input of human games entirely. In this variant, the leaf nodes are evaluated solely based on the value network ($\lambda = 0$ in (3)) and the MCTS agent itself is trained via self-play with combined parameters from the policy and value network, rather than training the policy and value network separately. The networks are also much deeper, containing 20 to 40 residual blocks [11]. The same architecture was also able to learn Chess and Shogi, beating the strongest computer programs.

We therefore expect that MCTS agents combined with deep neural networks will be instrumental in developing computationally feasible solutions to other MDPs with large state spaces, as well as more general reinforcement learning problems.

V. Acknowledgements

The authors thank their professor Jamie Haddock for her machine learning class, and the Google Education Grant for cloud computing credits.

[3] https://www.gnu.org/software/gnugo/gnugo_19.html
[4] https://github.com/meiji163/bokego

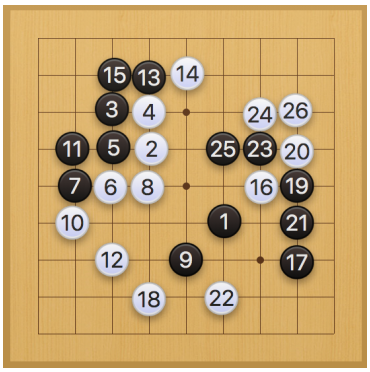


Fig. 4. GNU Go (B) vs Boke (W)

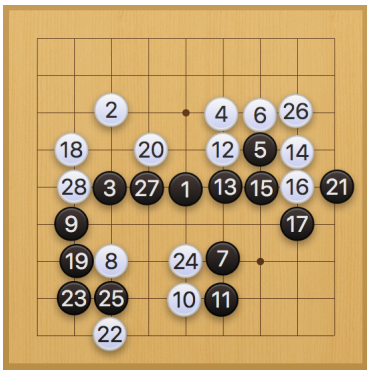


Fig. 5. Boke (B) vs GNU Go (W)

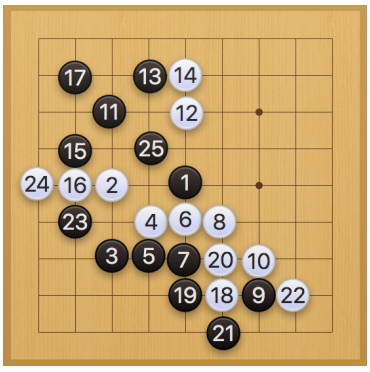


Fig. 6. GNU Go (B) vs Boke (W)

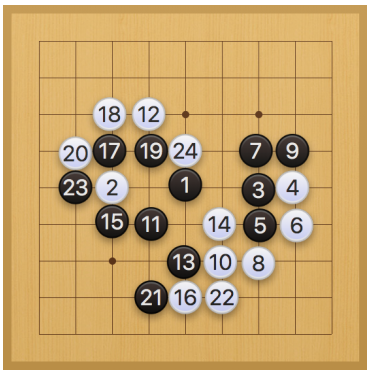


Fig. 7. GNU Go (B) vs Boke (W)

REFERENCES

[1] Kgs go server. https://www.gokgs.com/.

[2] C. D. Aliprantis. On the backward induction method. *Economics Letters*, 64(2):125 – 131, 1999.

[3] Peter Auer, Nicolò Cesa-Bianchi, and Paul Fischer. Finite-time analysis of the multiarmed bandit problem. *Mach. Learn.*, 47(2–3):235–256, May 2002.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[5] Guillaume Chaslot, Christophe Fiter, Jean-Baptiste Hoock, Arpad Rimmel, and Olivier Teytaud. Adding expert knowledge and exploration in monte-carlo tree search. In H. Jaap van den Herik and Pieter Spronck, editors, *Advances in Computer Games*, pages 1–13, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[6] Guillaume M. J. B. Chaslot, Mark H. M. Winands, and H. Jaap van den Herik. Parallel monte-carlo tree search. In H. Jaap van den Herik, Xinhe Xu, Zongmin Ma, and Mark H. M. Winands, editors, *Computers and Games*, pages 60–71, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[7] R. Coulom. Efficient selectivity and backup operators in monte-carlo tree search. In *In: Proceedings Computers and Games 2006*. Springer-Verlag, 2006.

[8] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2, 1989.

[9] G. Farneback D. Bump and A. Bayer. Gnu go.

[10] M. Enzenberger, M. Müller, B. Arneson, and R. Segal. Fuego—an open-source framework for board games and go engine based on monte carlo tree search. *IEEE Transactions on Computational Intelligence and AI in Games*, 2(4):259–270, 2010.

[11] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[12] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[13] Steven James, George Konidaris, and Benjamin Rosman. An analysis of monte carlo tree search. 02 2017.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[15] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In Johannes Fürnkranz, Tobias Scheffer, and Myra Spiliopoulou, editors, *Machine Learning: ECML 2006*, pages 282–293, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[16] Yann LeCun and Yoshua Bengio. *Convolutional Networks for Images, Speech, and Time Series*, page 255–258. MIT Press, Cambridge, MA, USA, 1998.

[17] Ilya Loshchilov and Frank Hutter. Fixing weight decay regularization in adam. *CoRR*, abs/1711.05101, 2017.

[18] K. Matsuzaki. Empirical analysis of puct algorithm with evaluation functions of different quality. In *2018 Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, pages 142–147, 2018.

[19] M. Müller. Computer go. *Artif. Intell.*, 134:145–179, 2002.

[20] Pascutto, Gian-Carlo. Leela zero.

[21] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library, 2019.

[22] C. Rosin. Multi-armed bandits with episode context. *Annals of Mathematics and Artificial Intelligence*, 61:203–230, 2011.

[23] David Silver, Aja Huang, Christopher 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*, 529:484–503, 2016.

[24] 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):354–359, 2017.

[25] 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*, 550:354–, October 2017.

[26] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[27] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural Comput.*, 6(2):215–219, March 1994.

[28] John Tromp and Gunnar Farnebäck. Combinatorics of go. In H. Jaap van den Herik, Paolo Ciancarini, and H. H. L. M. (Jeroen) Donkers, editors, *Computers and Games*, pages 84–99, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[29] J. v. Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, December 1928.

[30] J. Von Neumann and O. Morgenstern. *Theory of games and economic behavior.* Theory of games and economic behavior. Princeton University Press, Princeton, NJ, US, 1944.

[31] David J. Wu. Accelerating self-play learning in go. *CoRR*, abs/1902.10565, 2019.

[32] D. Yarotsky. Universal approximations of invariant maps by neural networks. *ArXiv*, abs/1804.10306, 2018.

[33] E. Zermelo. Über eine anwendung der mengenlehre auf die theorie des schachspiels. In *Proceedings of the fifth international congress of mathematicians*, volume 2, pages 501–504. II, Cambridge UP, Cambridge, 1913.