CS229 Project: Building an Intelligent Agent to play 9x9 Go

Shawn Hu

Abstract— **We build an AI to autonomously play the board game of Go at a low amateur level. Our AI uses the UCT variation of Monte Carlo tree search algorithm to select its actions, with playouts weighted by prior knowledge of tactical features learned from records of master-level play. We achieve a relatively weak strength of 18 kyu due to computational constraints, but demonstrate significant improvement over raw MCTS.**

I. INTRODUCTION

Go is classically a very hard game for AI to learn. Because the game's complexity depends on a vast array of properties that emerge from a small set of simple rules, human gameplay depends on reducing the state space by applying a large set of heuristics that depend on local shape, learned proverbs, and a subtle mix of tactical and strategic considerations. For computers, this means that traditional approaches to games like alpha-beta minimax achieve extremely poor results for Go- the branching factor is too large, and it is extremely difficult to design an evaluation function that prunes the search tree well enough. The Monte Carlo search algorithm, invented in 2006, was the first search algorithm that allowed Go AI to achieve a high-amateur level on even the 9x9 board [1]. By contrast, pre-MCTS Go bots operated using a large collection of hard-coded positional heuristics [8], which largely depended on the Go knowledge of their authors. This project lies in the middle of the two approaches, and attempts to use machine learning to automatically learn some of these simple positional heuristics for use in a basic Monte Carlo Tree Search agent.

Acknowledgement: This project is closely related to a CS221 project, which is also about Go. The CS221 project concerns solving Go problems, and as such shares the architecture for the Go board and contains similar architecture for reading SGFs. It also contains a very basic variant of our Monte Carlo Tree Search agent.

II. DATASET

Our input consists of 13,175 SGF files which contain records of games played on the CGOS servers. The games were played at 2500-2800 ELO (5-9 dan), a high amateur to low professional rating. Each of these .sgf files is a textual representation of the sequence of moves played in the game. To integrate the data with our Python implementation, we processed the data to play out these .sgf's on a Python Go board, and we analyzed the resulting states.

III. ALGORITHM

Broadly, the structure of the overall method is as follows:

1) Learn weights for a set of features from the dataset.
2) Use these weights to define an evaluation function on actions and states.
3) Use the result of this evaluation function as a prior to provide a smart ordering for Monte Carlo tree search exploration.

A. The Monte Carlo Tree Search Algorithm

Monte Carlo tree search (MCTS) is an algorithm that works by iteratively building a search tree according to some randomized policy. After a new node on this tree is created, the game is played out according to an extremely weak (usually random) policy to determine the winner, and the result is propagated up the tree and records are stored in the tree's nodes.

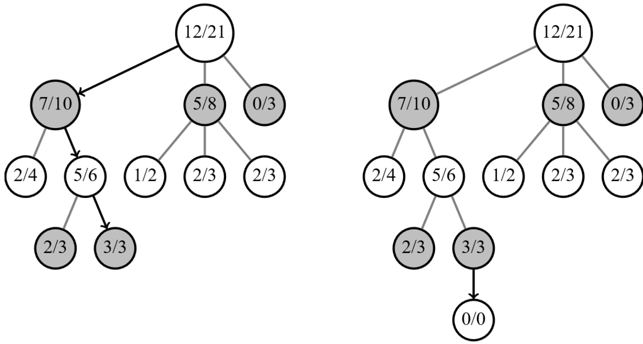
The policy is such that after multiple iterations have been executed, the agent follows those moves that are have won more often in previous playouts, thus leading the agent to spend most of its computation time on the most promising moves.

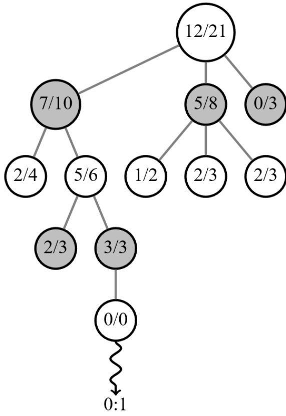
Each iteration of the MCTS algorithm consists of four stages:

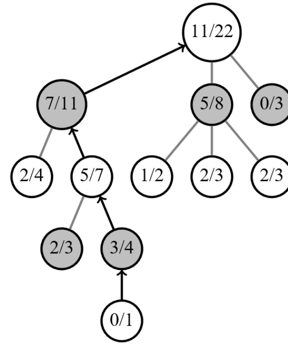
1) Selection: Starting from the root node, we select a node with probability proportional to its win percentage. We proceed until we reach a leaf node.

2) Expansion: Starting from this leaf node, me create a child node, which corresponds to taking a move from the leaf node's state. This move is chosen according to some prior distribution, which in our algorithm is calculated based on the features of the resulting states.

Fig. 1. The MCTS algorithm on an example tree.







3) Playout: From this child node, random moves are made until the game ends. The winner of the game is then calculated.

4) Backpropagation: The result is then recorded, and the record for each parent of the new node is updated to reflect the winrates of each state. In future playouts, this information may be used to affect the policy in nodes that are played sufficiently often.

B. The UCB1 Selection Algorithm for MCTS

UCB1 is a selection algorithm which is part of the UCT variation of MCTS. UCT, which stands for Upper Confidence Bound to Trees, is a variation which improves on standard MCTS as a method for making decisions based on prior knowledge in the selection phase. In particular, UCB1 works by defining a confidence interval for the value of every move, proportional to the amount of MCTS lines which have followed that move. Then, during each selection phase, UCB1 picks the move with the highest upper bound on its potential value. This behavior is desirable because with the right definition for the confidence interval, UCT can balance the time spent searching between following good lines of play and exploring many lines of play- if a line is good, its expectation increases, so it will be played more, until the confidence intervals for the other moves widen due to increasing uncertainty. Then, those moves are explored, and if they are not viable candidates the agent will return to exploring the strongest lines of play. In fact, it has been shown that using a confidence interval of width

$$\sqrt{\frac{2lnx}{x_i}}$$

, where x is the number of total plays and x_i is the number of plays on a fixed move, asymptotically minimizes the expected difference between the optimal strategy and the taken strategy.

The AI is given, for example, thirty seconds to perform its playouts. After calculation time ends, we choose the move with the most playouts, which by the nature of the algorithm often corresponds to a high winrate.

One huge advantage of UCB1-MCTS, for this project and for the field of Computer Go in general, is that it does not make use of an explicit evaluation function, and does not necessarily require any prior knowledge about how the game works- both things which are notoriously difficult for humans to translate well into code for Go. The algorithm benefits from the random policy in its playouts because they are extremely quick to compute, allowing the agent to quickly direct its search to the most promising nodes.

C. Features and Learning

Top-level Go bots, including AlphaGo [5] guide the MCTS search using a policy network and a value network. The policy network is used to immediately reduce the branching factor by favoring moves which are likely to be good based on tactical considerations. This is weakly analogous to the human method of choosing moves based on learned proverbs, or consideration of "good style". The value network defines an evaluation function on states, which either confirms or corrects the predictions of the policy network based on the quality of subsequent states. This is weakly analogous to the human practice of reading out lines of play, and then making decisions based on the predicted resulting states.

To mimic this approach, we extracted features from the moves of the winning player only (a common approach in amateur Computer Go) and the subsequent board states.

To define an evaluation function on states, we followed the approach of past master-level bots [1] and extracted features corresponding to the presence of certain 3x3, 2x2, and 1x1 patterns at every separate coordinate on the board. There are over a million such combinations of coordinates and patterns, and though some (such as a block of nine stones of one color) are unlikely to appear, we hypothesized that this should give reasonably strong behavior with respect to local tactics. To reflect the symmetry of the board, we had patterns share weights when identical up to horizontal, vertical, or diagonal reflection across the center of the board.

It should be noted that although this approach produces over a million possible features, for any given board fewer than 200 of these indicators will be nonzero, so computation of these features for a given board state is not excessively expensive.

We also extracted a weak set of features from the actions themselves (analogous to a policy network). These features were mostly designed to get the agent to lean towards obvious moves, and included:

- Whether or not the move leads to a direct capture;
- How many ataris (threatened captures) the move produces;
- Whether or not the move connects two groups;
- The Manhattan distance from the previous move, broken into separate features depending on the number of stones on the board (to reflect the behavior that non-local plays are more expected in the early and late game).

Notably, unlike with TD-learning approaches, we learned two sets of weights: one for the "policy network", corresponding to the value of an action given a previous state, and one for the "value network", corresponding to the value of the subsequent state. The weights were learned using gradient descent to minimize the squared loss between our predictions and the value of winning moves. We arbitrarily assigned a score of 1 to winning moves, so that on every winning action from a state s resulting in successor s', we applied the update rules

$$w_{s,a} := \eta[w \cdot \phi(s,a) - 1]\phi(s,a)$$

$$w_{s'} := \eta[w \cdot \phi(s') - 1]\phi(s')$$

IV. PREDICTION

The base implementation of our exact version of UCB1 operates by initializing from every state s a node corresponding to each action a. Each of these nodes is initialized with one win record and one loss record, so that the UCB1 formula treats them all equally. Our model incorporates the prior knowledge learned from the evaluation function by simply adding to each node $\phi(s,a) \cdot w_{s,a} + \phi(s') \cdot w_{s'}$ wins (with a hard minimum of 0.1 total wins in the case of negative dot product). Due to the nature of UCT-MCTS, we hypothesized that this alone would be enough to significantly affect the performance of the agent: because this method upweights obvious actions and tactically strong positions on the MCTS tree, the algorithm is allowed to spend far more time following the "obvious" lines of play. Conversely, with sufficient computational power, UCT-MCTS also acts as a safeguard against fully following any incorrect heuristics: given enough time, the playout history begins to outweigh the initialized wins given by the priors, and the agent returns to making the moves most likely to win based on the playouts.

V. RESULTS

Our agent, operating at a speed of about 10 playouts per second, achieved an estimated skill level of 18 kyu (low amateur). For even 9x9 Go, this is considered decent as a first (month-or-two) attempt; for reference, the upper bound in skill of a raw UCT agent (running at 2000 playouts per second) is estimated by the community to be around 5 kyu. Importantly, we observed through directly playing the agent that the incorporation of prior knowledge through the extracted features, which was the main interest of this project, made a notable difference.

TABLE I
ESTIMATED SKILL OF AGENT WITH VARIOUS SETS OF FEATURES

Both Sets of Features	18 kyu / 400 Elo
Action Features Only	23 kyu / 50 Elo
State Features Only	20 kyu / 100 Elo
Raw UCT-MCTS	$\geq$ 25 kyu / 0 Elo

VI. PERFORMANCE DETAILS, DISCUSSION AND FURTHER APPROACHES

This section discusses noticeable flaws with the agent's performance and their likely causes, and proposes various potential improvements to the model. We start by discussing computational power and move on to discussing different approaches to feature extraction.

We also discuss some interesting potential alternate modifications to our MCTS algorithm.

A. *Testing Architecture*

Currently, all estimations of the bot's skill come from the Go-playing judgment of its author. Standardized formats exist that allow the bot to play against other bots of varying strengths online, which would allow us to gauge its performance more concretely and adjust accordingly.

B. *Raw Computational Optimization*

It should be noted that a speed of 10 playouts per second is actually extremely slow by modern standards, and this limitation in computational power is by far the main factor which keeps the overall performance of the bot weak. For comparison, year-long project Go bots often have speeds of about 2,000 per second, high-end Go bots often reach speeds of 10,000 playouts per second, and top-end bots over 100,000 per second on the 19x19 board. Our inefficiency is due to some wasteful use of certain data structures, but more fundamentally due to our use of Python running on a single thread. Most serious (multiple-year-long) Go projects are implemented in C and executed in multiple parallel threads. With or without the features exhibited in this project, such computational gains would immediately massively advance the performance of the bot. Although the focus of the project is on the performance gains from the features and not from the absolute strength of the bot, it would be interesting to see whether the initial weighting helps much on a bot which performs a larger number of playouts.

C. *The Problem with Light Playouts*

Light playouts are playouts for which the policy is close to random, or at least relatively weak (in contrast to heavy playouts, in which significant computation is involved). In theory, the UCT-MCTS algorithm's policy will converge to optimal after sufficiently many playouts. However, convergence takes a significantly longer time for "fragile" positions. Consider a situation, relatively common in high level play, in which there exists one move which is tactically far superior to all others, but recognizing this involves reading out a ten-move sequence of subsequent moves. In this case, in order to recognize such a situation, the MCTS agent must make the decision to follow the search tree through these exact ten moves before recognizing that the first is very good at all. In such a situation, this can lead to the loss of a group and subsequently the whole game. More commonly, in high-level mid-to-endgame Go, complex tactical situations arise in which from a given position there is one surviving (and hence winning) move for White and 10 killing moves for Black. While the agent might not perform so badly once confronted with this exact board position, it is unlikely to ever put itself into such a winning position because it cannot predict the result during a light playout.

This behavior is a glaringly non-human weakness in the current agent; knowingly exploiting this by constantly playing into complex positions that its features don't immediately recognize can plummet the apparent performance of the agent by perhaps 5 kyu. These hurdles can be overcome with sufficient computational power, but as with classic minimax, the main way of coping with this problem is to decrease the effective branching factor at each node so that the playouts can "carve out" this line of play quickly. This can be accomplished to a great extent with a strong policy net; public visualizations of AlphaGo's thought process show that at times over 60 percent of all playouts from a node explore a path beginning with the same move.

D. *Caching*

This idea follows a simple concept: it is efficient to use the knowledge you have previously calculated. In particular, if a previous MCTS search has reached the current state and calculated the values of various actions from this state, then we can begin where the previous calculations left off, since the calculations from a subtree of a previous MCTS search are identical to the calculations from the current MCTS search (i.e, previous playouts which include the current state necessarily contain playouts from the current state). This effect compounds well with predicting heuristics such as the one explored in this project, since these heuristics fundamentally derive their strength by exploring the correct branches of the search tree to a more thorough extent. This sometimes has the problem of being quite space-inefficient, since MCTS trees can grow quite large, and it's not always clear which trees should be stored for even further use as the game progresses (for example, in some situations where multiple permutations of moves can result in the same end state). However, many mid-to-high end Go bots have such good policy networks that they almost always get to reuse significant portions of their MCTS tree, drastically speeding up calculations. In our case, is likely that even naively caching the results obtained from the previous move would increase the

performance by a small but palpable amount.

E. Dynamic Komi

One obvious and exploitable facet of the bot's performance is that it begins to play lazily whenever it has a lead. This is because when it has a lead, most of its MCTS playouts result in victory, causing many moves to appear good, when in reality there might be complex lines of play that force a loss. With dynamic komi, the agent automatically adjusts its required threshold for winning so that its MCTS playout win probabilities don't exceed a certain amount. In other words, if the bot expects to lead by ten points, to some extent it will try to maintain that lead through its play.

F. Bootstrapping

We tried to improve the bot's performance by following AlphaGos method of generating more data for the agent to learn from by having the agent play itself in simulated games, then following the previous approach of learning from the winners positions. This proved to be almost completely ineffective. One simple hypothesis is that the bot is just not strong enough to produce game records worth learning from, especially not in comparison to the original (master-level) training set. However, it is likely that a more important contributing factor lies in the structure of the model itself (see next section).

G. Narrowing the Search with Better Features through Neural Nets

Beyond raw computational power, the model is most fundamentally limited by the nature of its feature set. While the features did improve the bot's performance, and while indeed we were able to marginally increase the performance of the bot by introducing specific extra features, there will always be facets of strategy in the game that are not captured by a reasonably sized, elementary, static feature set. Due to the fundamental nature of this problem, its consequences are very pervasive- the observed benefit of UCB1, tree caching, and more generally the use of MCTS are all compounded by a very predictive set of features. It should be noted that top-end Go bots such as AlphaGo are able to thoroughly narrow down the search space with much stronger value and action networks, developed by extracting and learning their features using a combination of convolutional neural networks and deep neural networks. In the long run, this style of approach will most likely outperform any set of features that a human could reasonably design.

H. RAVE

RAVE, which stands for Rapid Action Value Estimation, is the name for a very commonly used heuristic in mid-level Go bots [11]. In essence, RAVE approximates the value of a move by taking the sample mean of its observed value over all playouts. The RAVE model is known to learn extremely fast, but are often inaccurate. Hence, like the evaluation function developed in this project, the values obtained from RAVE are commonly used as priors for MCTS search in other approaches. It may be interesting to combine this approach with our own, e.g, by obtaining the RAVE estimates from our evaluation function-weighted playouts.

VII. Conclusion

Our final AI was still quite weak in performance, but this is largely attributable to its lack of computational power and not generally concerning considering the amount of time invested in its development (compared to the years-long development of some other Go bots). Despite being weak overall, our approach to guiding MCTS search via tactical feature extraction was able to demonstrate a palpable improvement over raw MCT-UCTS without the author specifically hard-coding any weights or concrete heuristics into the agent's logic. There is a large abundance of potential approaches we can take to improve the overall performance of this bot, but the one with the highest overall potential and relevance to machine learning is to try extracting features and learning the weights with neural nets instead.

ACKNOWLEDGMENTS

- Christopher Hart, author of AncientGo, who gave me ideas on endgame behavior;
- Jeff Bradberry, who produced the base implementation of MCTS that our code is based off of;
- Hiroshi Yamashita, for the dataset;
- Andreas Garcia and Brian Liu, members of the related CS221 project and hence contributors to some of the basic architecture of the project.

References

[1] S. Gelly and D. Silver, "Achieving Master Level Play in 9 9 Computer Go," in Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence (2008)

[2] P.Baudis, MCTS With Information Sharing, Masters Thesis, 2011

[3] Source code for Michi, one of the best minimal MCTS Go implementations in Python. https://github.com/pasky/michi

[4] Source code for Pachi, a popular and moderately strong MCTS Go implementations with heavier playouts in C. https://github.com/pasky/pachi
[5] David Silver, Aja Huang, et al., Mastering the game of Go with deep neural networks and tree search. Nature, 06 January 2016.
[6] E.C.D van der Werf, Learning to Predict Life and Death from Go Game Records, 2005
[7] Remi Coulom, Computing Elo Ratings of Move Patterns in the Game of Go, ICGA Computer Games Workshop, Amsterdam, The Netherlands, June 2007
[8] Source code for GnuGo, one of the strongest non-MCTS Go agents. https://www.gnu.org/software/gnugo/
[9] Byung-Doo Lee, Life-and-Death Problem Solver in Go, Dept. of Computer Science, Univ. of Auckland, New Zealand
[10] Akihiro Kishimoto, Martin Muler, Search versus Knowledge for Solving Life and Death Problems in Go. The Twentieth National Conference on Artificial Intelligence and the Seventeenth Innovative Applications of Artificial Intelligence Conference, July 9-13, 2005.
[11] Sylvain Gelly, David Silver, Monte-Carlo tree search and rapid action value estimation in computer Go Artificial Intelligence, Volume 175, Issue 11, July 2011.
[12] Documentation on the SGF file format: http://www.red-bean.com/sgf/ff5/m_vs_ax.htm
[13] Documentation on the way goproblems builds on the base SGF file format: http://www.goproblems.com/instructions.php3
[14] Image credit for the MCTS diagram. MCTS diagram: Mciura [username] - CC BY-SA 3.0