Playing Go with Deep Learning

Siddarth Sampangi **Haresh Chudgar**

Aditya Nagarajan **Addison Mayberry**

Abstract

Our class project was to implement a DQN to play the game of Go, based on Nathan Sprague's Atari-playing agent. In this report we discuss the rules of Go and prior art in the field of Go-playing agents, and we describe our attempted implementation in terms of the techniques we tried and the problems we encountered. In the end, we were unable to get consistent results from our network due to a variety of extreme challenges inherent to this domain. We present an analysis of our results and other possible approaches that may better address the issues we encountered.

1 The Game of Go

Go is an ancient Chinese board game composed of a grid of intersecting perpendicular lines. The rules are simple - there are two players, and each player may place one piece (or "stone") on the board. Stones can only be placed on the intersection between two lines on the grid, and a player may only place a stone in a space that is not occupied by another stone. One player uses black stones, the other, white.

Go is widely considered to be the last largely unsolved classical game in the field of AI. This is due to two major contributing factors:

1. The extremely large possible number of games. A standard board has $19 \times 19 = 381$ spaces in which to play, leading to 10^{171} possible board states - compare this to 10^{47} for chess. To consider all possible options for the next four moves would typically require examining 3.2×10^{11} possible board states.
2. The difficulty of evaluating a board state or evaluating the effectiveness of a given play. Go is known to be an extremely subtle game, with seemingly irrelevant plays in one area of the board having cascading effects only witnessed tens of moves later or more. This makes it extremely difficult for a machine to explore the space of potential moves or evaluate the current game state, even when guided by heuristics.

Due to these limitations, no one has as of yet been able to succesfully generate a Go agent that can compete at anything higher than a novice level. In this project, we attempted to expand upon (a) past work in the field of developing Go-playing agents and (b) recent work in developing deep-reinforcement-learning networks for playing games, in order to implement a deep-RL-based agent capable of playing Go at some level of competitiveness.

2 Prior Art

Monte Carlo Tree Search

Prior to the recent successes in exploring the Go problem with machine learning techniques, the best results in the field of Go agents were obtained using Monte Carlo techniques. The best and most recent example of this technique was Baudis & Gailly's Pachi [X], an open-source Go player built on the Monte Carlo Tree Search Algorithm (MCTS).

MCTS is based on an incrementally built probabilistic minimax tree. When the agent must make move, it takes the current game state as a node in the tree and expands out all possible moves this turn as child nodes. Any nodes already generated in previous iterations are included automatically. Whenever a new node is generated, it is assigned a score based on its expected value over the course of the game. This score is estimated by doing a series of Monte Carlo "playouts," in which moves are chosen completely randomly or randomly with heuristics. There are a large variety of tunable parameters in this model - the number of MC playouts to compute, how and when to expand out all the children of a node explicitly, how to compute a node's score, and what heuristics (if any) to guide the MC player towards more likely or useful moves all must be chosen by the implementer.

Baudis & Gailly iterated on a large body of MCTS-based Go work by adjusting the specific heuristics used for the MCTS player to include more advanced strategies. They do not present any quantitative measure of the strength of the Pachi player, but it was one of the first to play competitively against humans with some success. A side-by-side comparison of the following projects with Pachi included in [X] shows that it is weaker than the supervised learning approaches discussed next.

Deep Convolutional Neural Net

There have been two major advances in developing Go agents with machine learning techniques in the past year. The first was Clark and Storkey's paper [X] which used a deep convolutional neural network (DCNN). The network was trained on two large publicly available datasets of Go games.

To compile the training data, the games were broken into individual moves. The input is the board state and the output is the move chosen by the player. The authors used a set of about 16.5 million board - move pairs between the two datasets.

The authors experimented with a number of board encoding techniques to attempt to explicitly inform the network about common abstractions that players of almost all levels use to make decisions about which move to play. The most basic input encoding they used had three channels - the first two to indicate the positions of stones of each player, and the third to enforce a simple rule that prevents the game from devolving into an infinite sequence of repeated plays - this rule is known as "ko". More advanced forms of board encodings they used explicitly captured simple features such as the the reflective properties of the game board or tactical information such as the number of "liberties" (neighboring empty board spaces) available to each piece on the board. They found that all of these encodings increased performance.

The best network architecture they reported involved eight layers. The first seven layers were convolutional layers with filters of decreasing size, which were zero-padded out to the width of the board to prevent the size of the outputs from progressively shrinking. The last layer was a fully-connected layer with a softmax output over the entire board, which is interpreted as the network's prediction of the most likely move a human player would make.

One game feature they were forced to encode directly into the DCNN was illegal moves, which ended up being a very difficult problem for us in our DQN implementation as discussed below. The basic rule of placement is simple - do not place a piece on top of another piece. However, as we discovered, teaching an agent not to do this directly from data can be extremely difficult. The authors attempted this by only backpropagating values on legal board positions during training and were able to teach the network to avoid most illegal moves, but still not all. In the end, they were forced to zero-out any output probability given to illegal moves during testing and re-normalize the remaining probability over the space of legal positions.

The results reported by these authors were, at time of publication, the best so far for any approach incorporating supervised learning. They reported a test accuracy of 44% for predicting the move picked by a human player, and had a 91% win rate against GnuGo, a popular open-source Go player.

DCNN / Monte Carlo Hybrid

Two very recent works, by Maddison et al. at DeepMind and by Tian & Zhu at Facebook Research combined the described two techniques to yield the best results in the field so far [X]. They assert that one of the major weaknesses of the DCNN approach is its inability to search the space of moves, which makes it tactically weak as it cannot look ahead at the impact of future moves easily or naturally. They claim that search allows the agent to build a nonparametric local model based on the current state of the board, which has a great deal of utility alongside a global "best-move" model like the one provided by the DCNN implementation. Both the DeepMind and Facebook projects use MCTS to generate the search tree nodes, which are then passed to the DCNN component for evaluation. The two approaches differ in the method of synchronization - the DeepMind implementation allows the MCTS algorithm to run in parallel with the DCNN evaluation, whereas Facebook's system requires MCTS to halt until the DCNN evaluation of a node is completed. The former emphasizes high playout volume, the latter, better heuristic guidance of the rolled out nodes within MCTS.

3 Initial Setup

Building an Opposing Player

Our first goal was to choose an enemy agent for the DQN to play against while learning. Our initial thought was to use Clark & Storkey's trained DCNN model as the opponent. We were able to get the weight parameters for their model, but when we ran the DCNN using their published parameters, its behavior appeared to be completely random. We were not able to configure the DCNN such that it was able to play with any kind of intelligence, and we assume that we were missing one or more parameters that were not explicitly mentioned in the paper.

We next turned to Pachi, Baudis & Gailly's MCTS-based player. We were able to get it working, but it was very slow to compute a next move. AM: Can we put a rough estimate of the execution time? This would have been far too slow to be useful for the purpose of running hundreds of training epochs for the DQN.

We finally settled on GnuGo, a standard open-source heuristic-based Go player. This program is the baseline for comparison in most of the recent papers on Go, so we considered it to be a good choice for training our agent. GnuGo has a scale of difficulty levels it will play at, from 1 to 10, with 10 being the most advanced. There is a tradeoff, however, as more advanced levels also take significantly more time to decide on a move. We left it at the default setting of 10 in order to push our agent towards better play from the beginning.

Porting the Atari Agent

We based our work on Nathan Sprague's Atari-playing code from the midterm project. Fortunately, the interface between the learner and ALE was designed to abstract away most of the details specific to any particular Atari game, since they needed to be able to run it over all of the games in the list. We were able to swap out ALE for GnuGo by doing a few minor modifications to the training code and creating a simple interface from GnuGo to the DQN trainer. This allowed the training code to request score and game state from GnuGo the same as it did with ALE.

The biggest change we had to make to the trainer was to adjust the input and output representations. The output was set to be a 361-unit softmax, one unit for each position on the board, as in the original DCNN work. We tried working with two different styles of input representation, also based on the prior art. The first version we tried had three input channels, two of which encoded each player's positions, with the third representing the set of possible legal moves.

We also worked with a seven-channel representation. AM: I don't know what was in the 7-channel representation, need to talk to Haresh or get him to fill this in. Also don't know which one we settled on, 3-channel or 7-channel.

4 Evaluation of Network Parameters

Parameter Tuning and Results

Once the DQN trainer was connected to GnuGo and working, we began adjusting the parameters to more closely mirror those of Clark & Storkey's DCNN. In this section we discuss the network parameters we tried and the results we were able to achieve with them. To test any given set of parameters, we trained the network for 30 games and observed the opponent's score. We focused on the opponent's score because we expected that, as the network improved its tactics, the changes would manifest in the GnuGo opponent having more difficulty scoring higher points.

We first reduced Sprague's default training parameters by 4x to account for the fact that, in the Atari case, the network would receive 4 frames at once from ALE, whereas with GnuGo it was receiving one board state at a time. Table **??** gives the various parameters we attempted to tune and the specific settings we tried.

We also attempted to integrate weight tying, which was used in the prior DCNN to force the network to take advantage of board symmetry by tying weights together within the convolutional layers. However, doing this with the tools provided by Theano / Lasagne proved very challenging. In the end, we were not able to find a straightforward way to implement this feature in the time allotted. This was a disappointment because intuitively this kind of understanding would help the agent greatly (and this intuition was borne out in Clark & Storkey's work, which found that incorporating symmetry did increase performance.)

Another issue that proved to be surprisingly challenging, as previously mentioned, was to find a way to prevent the agent from making illegal moves. AM: didn't have time to add the board errors discussion and graph yet.

Unfortunately, in all cases we were unable to get consistent performance, regardless of specific parameter settings. Figure 1 shows one particular training session that we allowed to run for several hundred epochs.

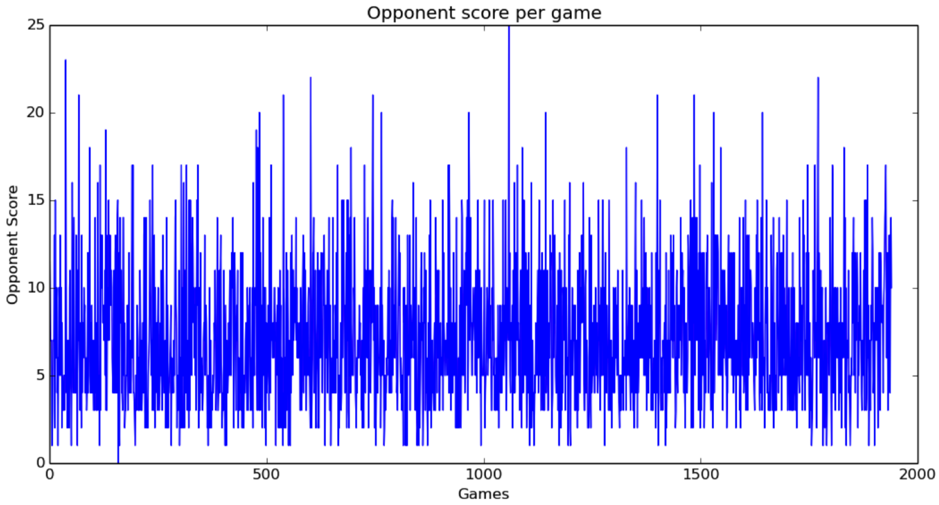

Figure 1: Opponent (GnuGo) score per training epoch.

Analysis

AM: need some analysis on why it didn't work... could possibly migrate this into the Future Work section instead.

5 Future Work

References

AM: references need to be wrapped up but I'll take care of it, just leaving this as a note to myself.

[2] Bouzy, B. & Chazlot, G. (2006) Monte-Carlo Go Reinforcement Learning Experiments *The Book of GENESIS: Exploring Realistic Neural Models with the GEneral NEural SImulation System.*

[1] Clark, C. & Storkey, A. (2015) Training Deep Convolutional Neural Networks to Play Go. *Proceedings of the 32nd International Conference on Machine Learning - ICML '15.*