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 $19 \times 19 = 361$ 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 successfully 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

2.1 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 [1], 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 a move, it takes the current game state as a node in the tree and expands all possible moves of 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 including 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.

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 [4] shows that it is weaker than the supervised learning approaches discussed next.

2.2 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 [2] 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.

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.

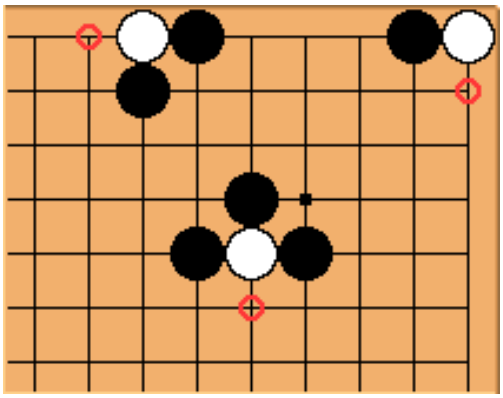
2.3 DCNN / Monte Carlo Hybrid

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 [3,4]. 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 non-parametric 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

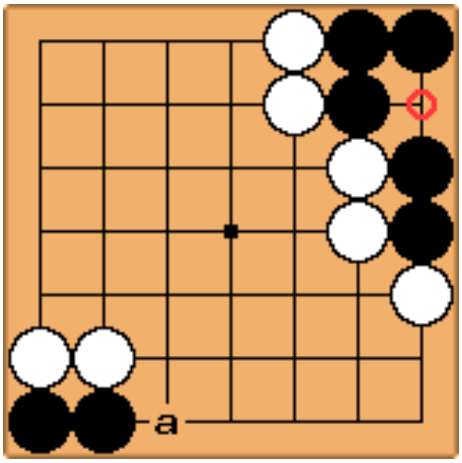
system requires MCTS to halt until the DCNN evaluation of a node is completed. The former emphasizes high playout volume, the latter, better heuristic guidance of the rolled out nodes within MCTS.

3 Deep Q Learning for solving Go

In this section, we detail our initial motivation behind using deep Q learning (specifically, DeepMind's approach) to solve Go. As detailed in the previous section, Go has traditionally been solved using Monte Carlo approaches which search through the complete future state space of the game. However, this is time consuming considering the size of the game's state space, and so various approximate techniques have to be used to cull the search space. This requirement is the main drawback of Monte Carlo approaches.
Human experts use an approach that can be better likened to pattern recognition. Professional players analyze positions using a large vocabulary of shapes, such as joseki (corner patterns) and tesuji (tactical patterns). The most simple example is that of an atari which is shown in Figure to illustrate the idea. If we regard the game state as an image (each position is a pixel), then a Convolutional Neural Network is better suited to identifying these patterns and abstract shapes. This is likely the reason behind the success of applying convolutional neural networks to Go. However a CNN lacks the concept of time, and can only decide on the next move. Thus, it fares badly against a Monte Carlo approach, which can plan multiple moves ahead - a key necessity in strategy based games.



(a) Example 1: The white stones are all in atari and can be captured if black plays the next move



(b) Example 2:The five black stones on the top right are in atari. The only liberty they have is the circled intersection. The group of two black stones on the lower left is also in atari. It may be captured by white (when white plays at a).

Figure 1: Concept of Atari

3.1 Reinforcement Learning

We present a brief overview of reinforcement learning. Reinforcement learning is an area of computer science that is designed to solve problems that have a notion of risk vs. reward. It is especially suitable for designing agents that can learn to perform actions in a given environment to maximize reward, where the reward can even be intermittent. Thus, it is well matched for solving problems such as games, which have very clear definitions of possible actions and rewards. Briefly, popular reinforcement learning techniques involve learning a function $Q(s, a)$ that returns a particular reward r for an action a taken on a given state s. The agent then simply learns to take actions that

maximize the cumulative reward. Reinforcement learning was given the spotlight in 1992 when Gerald Tesauro designed an agent that learned to play backgammon at a world-class skill level through self-play. However, further achievements were not possible at the time because the space of possible state-action combinations was simply too large in most other tasks for an agent to efficiently learn a good policy.

3.2 DeepMind's Atari Agent

The breakthrough came in 2014 when DeepMind designed an agent that could (very quickly) learn to play a variety of Atari arcade games at superhuman performance. The concept of approximating the Q-function when its true state space was too large was not new. However, DeepMind was the first to successfully train a convolutional neural network (deemed a Q-network) to approximate the Q-function. With this advancement, their agent was able to operate on the raw pixel data of the atari emulator and learn the best action.

3.3 Testing DeepMind's approach on Go

For us, the natural next question was whether we could make minimal modifications to DeepMind's approach, and train an agent that would learn to play Go at a decent skill level. While the state space of Go is far larger than that of the Atari games, we hypothesized that training an agent to perform to an acceptable or even poor (but better than random) level would be possible, given the advantage of explicit board representations and much smaller board sizes.

4 Building an Opposing Player

Our first goal was to choose an enemy agent for the DQN to play against while learning. We evaluated three different enemies.

4.1 Clark & Storkey's trained DCNN model

Clark & Storkey used ConvnetJS to build their network. To use their network, we had to parse their network file and import it to lasagne. We found that the converted network was not playing as well as the online version posted by the authors. This is probably because ConvnetJS and Lasagne represent weights differently. We couldnt afford to spend more time on the issue, and so we moved on to evaluating other approaches.

4.2 Pachi/Michi

We next turned to Pachi, Baudis & Gailly's MCTS-based player. While playing against it, we realized that it was slow in generating moves (each move took 1-2 seconds). This would be far too slow for the purpose of generating training examples for DQN.

4.3 GnuGo

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.

Another important change we had to make was the input representation. We wanted to try out different representations of the go game and evaluate each one of them. This part was a bit tricky as the input representation is tied with almost every class in the code.

5 Experiments

There are three major parameters in our system. We now describe them.

5.1 Input representation of the game

We evaluated three different matrix representations of the Go board.

5.1.1 One channel representation

Translating the board to a matrix directly by encoding a white stone as 255, a black stone as 0 and an empty position as 127. This was nearest to the Atari representation, but we felt that it trued to unnecessarily "compress" the data into one channel.

5.1.2 Three channel representation

The first and second channels encode black and white stones respectively by placing a one in the matrix if there is a stone, 0 otherwise. The third channel places a one if there is a stone, zero if it is empty; this is basically the sum of first two channels and we believed it could help by more explicitly representing the illegal moves that our agent was most prone to attempt (placing pieces in occupied positions).

5.1.3 Seven channel representation

This is inspired from Clark & Storkey's input to the CNN which encodes information about the game itself by using the concept of liberties as shown in Figure 2. A stone is said to have 1 liberty for every neighboring unoccupied position (in Figure 1, a black stone with 2 liberties is shown, as it has 2 neighboring white stones). If two stones of the same color are adjacent, the liberties of each stone are shared (this is depicted in the top left corner of Figure 1, where the group of three stones have a liberty count of 7 - the diagonally placed stone has a liberty of four and is not part of the group.). In this representation, we encode black positions in the first three channels by placing a one in the first channel if that stone has more than three liberties, one in the second channel if that stone has two, and one in the last channel if that stone has one liberty. The next three channels similarly encode positions of white stones. Each channel has a padding of three on all sides, so each channel of a 7 x 7 board will be of size 10 x 10. For the first six channels the padding is set to 0, whereas the seventh channel is 0 everywhere except at the padding where it is set to 1. This padding encodes the board's boundary. As it is not the focus of this paper, we leave it to the reader to find more details on the motivation for this final channel in Clark & Storkey's paper.

5.2 Architecture of the convolution neural network

Briefly, the different network configurations we experimented with are as follows:
i. Two layer CNN with one dense layer and one softmax
ii. Three layer CNN with one dense layer and one softmax
iii. Five layer CNN with one dense layer
iV. Seven layer CNN with one dense layer
In addition, we varied the filter sizes of the convolution layers.

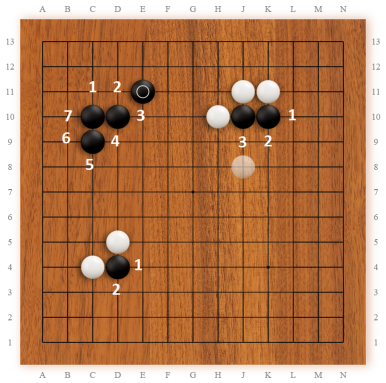


Figure 2: Concept of Stone Liberties

5.3 Parameters of Q Learning

We also varied the experience replay size and discount factor to find the right range of values.
Each experiment was performed with the hope that our main issue was simply that of finding the right network that can approximate our Q-function. However, as we will discuss later, we believe there are more fundamental issues with our approach that prevent deep reinforcement learning from successfully tackling Go without other meaningful improvements.

5.4 Weight tying

Apart from the configurations mentioned, we also attempted to integrate weight tying, which was used in the by [2] to force the network to take advantage of board symmetry by tying weights together within the convolutional layers. However, we were limited by Lasagne as it allows weight tying only between filters and not within each filter; we could not find an alternative way to implement this feature in the time allotted. This was a disappointment because intuitively this kind of understanding would help in speeding up the convergence as the network would have to learn only one eighth of the filter weights.

5.5 Illegal Moves

Another issue that proved to be challenging, as previously mentioned, was to find a way to prevent the agent from making illegal moves. The fundamental difference between the game of Go and Atari games is the action space is dynamic. Initially the action space is square of the board size as all the places are empty, however the space keeps changing as positions get filled or freed as stones are added or captured by the players. This resulted in a huge negative reward as the network consistently gave illegal moves and surprisingly did not learn from the negative reward. One of the reasons for this could be the constantly varying game state with no repetition. We dealt with this problem by adding a multiplicative layer at the output of the last dense layer which zeroed out the Q values of illegal moves.

6 Results

We tested out different configurations of the experiments mentioned in the last section and evaluated them using the difference in scores of the network and GnuGo. We expected that, as the network improved its tactics, the changes would manifest in the GnuGo opponent having more difficulty scoring higher points thus increasing the difference from negative to positive.

Unfortunately, in all cases we were unable to get consistent performance, regardless of specific parameter settings. It did win occasionally, but overall GnuGo beat the network consistently. In the games which it won, we could not make out a clear strategy so we attribute the wins to chance. Figure 3 shows one particular training session that we allowed to run for 2000 games.

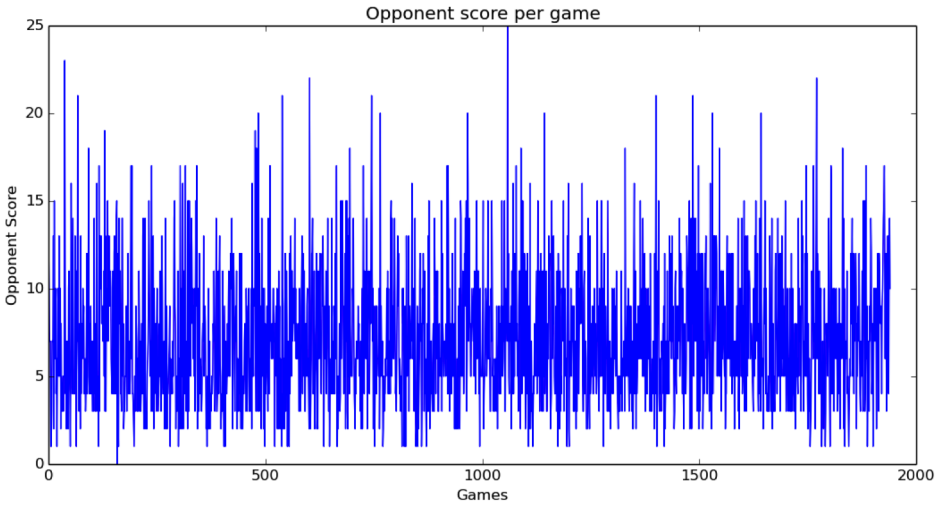


Figure 3: Opponent (GnuGo) score per game on a 7×7 board. The network was provided with a 7 layer input and consisted of 5 convolution layers

7 Analysis and Future Work

In this section, we offer hypotheses that attempt to explain why our agent was not able to learn an appropriate Q-network for Go. First, we simply could not generate a sufficient number of quality training examples. A large number of our training examples were illegal moves with a reward of -1. Even if we disregard this, the speed with which we were able to generate training examples was simply far from sufficient. Considering that [2] trained on 16 million examples, and that, assuming approximately 50 moves per game on a board of size 7, it took us 3 days to generate 100,000 examples, it is simply infeasible to expect to generate a data set of comparable size. We must then rely on using training examples in a more efficient way, or we must avoid training a network from scratch.

As mentioned before, weight tying is one way to use training examples more efficiently. Still, weight tying only offers an improvement of an approximate factor of 8. We can think of this as allowing us to generate 800,000 moves in 3 days, which is still far from satisfactory. While it might not be sufficient, implementing weight-tying can undeniably help in training the Q-network, and is a good direction for future work.

Initializing the Q-network to the weights of the DCNN from [2] is also likely to help generate examples of significantly higher quality. Since the DCNN is already trained to mostly make legal moves, it is possible that initializing with the DCNN will ”kickstart” our system into generating and training with a good Q-function. While we were not able to implement this (we could import the network weights, but they would consitently return random moves), it is a definite direction for future work.

Additionally, it might very well be completely necessary to use a combination approach involving Monte-Carlo and convolutional networks, as recent results indicate.

While we have not been able to train a Q-network, we believe that we have still made a contribution to the community with our code. As of now, anyone can download our code from github and quickly experiment with their own modifications to the Q-network or other modules. Just as the open source ALE environment allowed DeepMind to test their system, our code will allow others in the community to test their strategies on Go.

References

[1] Baudi, Petr, and Jean-loup Gailly. "Pachi: State of the art open source Go program." *Advances in Computer Games. Springer Berlin Heidelberg*, 2012. 24-38.

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

[3] Maddison, C. J., Huang, A., Sutskever, I., & Silver, D. Move Evaluation in Go Using Deep Convolutional Neural Networks. arXiv preprint arXiv:1412.6564. 2015.

[4] Tian, Y., & Zhu, Y. (2015). Better Computer Go Player with Neural Network and Long-term Prediction. arXiv preprint arXiv:1511.06410. 2015.