Introduction

This document will include the problem definition of Go board game in AI as well as the techniques used to solve the problem including (MCTS: Monte Carlo Tree Search – CNN: Convolutional Neural Network – RL: Reinforcement Learning), Then we will talk about the competitors and their techniques to solve the problem, Finally we will give our approach to solve the problem which we recommend to the design and implementation.

1-Problem Definition

Games are a favorite subject for AI research, and it’s not just because they’re fun. They also simplify some of the complexities of real life, so you can focus on the algorithms you’re studying.

A Go board is a square grid, Stones go on the intersections, not inside the squares. The standard board is 19 × 19, but sometimes players use a smaller board for a quick game. The most popular smaller options are 9 × 9 and 13 × 13 boards.

(The size refers to the number of intersections on the board, not the number of squares.)

The core idea behind boardgame AI is tree search. Think about how humans play a strategy game. First, we consider a possible move for our next turn. Then we need to think of how our opponent is likely to respond, then plan how we’d respond to that, and so on. We read out the variation as far as we can, and then judge whether the result is good. Then we backtrack a bit and look at a different variation to see if it’s better.

This closely describes how the tree search algorithms used in game AI work. Of course, humans can keep only a few variations in their heads at once, while computers can juggle millions with no trouble. Humans make up for their lack of raw computing power with intuition. Experienced chess and Go players are scary good at spotting the handful of moves worth considering.

Ultimately, raw computing power won out in chess. But a Go AI that could compete with top human players took an interesting twist: bringing human intuition to computers.

2- Techniques to solve the problem

Reducing the number of moves to consider using MCTS

In the context of game tree search, the number of possible moves on a given turn is the branching factor. In chess, the average branching factor is about 30. At the start of the game, each player has 20 legal options for the first move; the number increases a bit as the board opens up. At that scale, it’s realistic to read out every single possible move to four or five moves ahead, and a chess engine will read out the more promising lines much deeper

than that. By comparison, the branching factor in Go is enormous. On the first move of the game, there are 361 legal moves, and the number decreases slowly. The average branching factor is around 250 valid moves per turn. Looking ahead just four moves requires evaluating nearly 4 billion positions. It’s crucial to narrow down the number of possibilities.

Monte Carlo tree search (MCTS) provides a way to evaluate a game state without any strategic knowledge about the game. Instead of a game specific heuristic, the MCTS algorithm simulates random games to estimate how good a position is. One of these random games called a rollout or a playout. In this book, we use the term rollout. Monte Carlo tree search is part of the larger family of Monte Carlo algorithms, which use randomness to analyze extremely complex situations. The element of randomness inspired the name, an allusion to the famous casino district in Monaco.

Each round of the MCTS algorithm takes three steps:

1. Add a new board position to the MCTS tree.



1. Simulate a random game from that position.



3. Update the tree statistics with the results of that random game.

You repeat this process as many times as you can in the time available. Then the statistics at the top of the tree tell you which move to pick.

How to select which branch to explore?

Your game AI has a limited amount of time to spend on each turn, which means you can perform only a fixed number of rollouts. Each rollout improves your evaluation of a single possible move. Think of your rollouts as a limited resource: if you spend an extra rollout on move A, you have to spend one fewer rollout on move B. You need a strategy to decide how to allocate your limited budget. The standard strategy is called the upper confidence bound for trees, or UCT formula. The UCT formula strikes a balance between two conflicting goals.

The first goal is to spend your time looking at the best moves. This goal is called exploitation (you want to exploit any advantage that you’ve discovered so far). You’d spend more rollouts on the moves with the highest estimated winning percentage. Now, some of those moves have a high winning percentage just by chance. But as you complete more rollouts through those branches, your estimates get more accurate. The false positives will drop lower down the list. On the other hand, if you’ve visited a node only a few times, your estimate may be way off. By sheer chance, you may have a low estimate for a move that’s really good. Spending a few more rollouts there may reveal its true quality. So, your second goal is to get more accurate evaluations for the branches you’ve visited the least. This goal is called exploration.

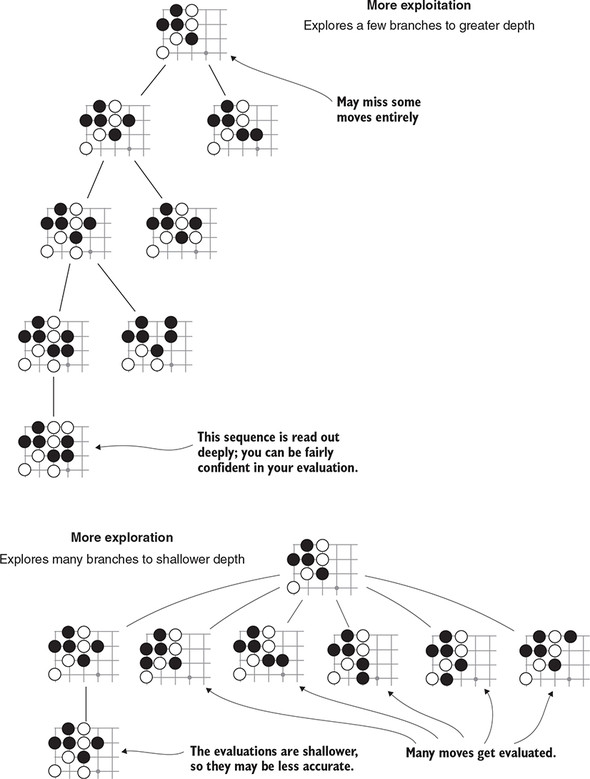
Finally, the UCT equation is calculated as:

UCT = W+C \* sqrt(log N/n)

where N is the total number of rollouts, n is the number of rollouts that started with the node under consideration to represent the exploration goal, W is the winning percentage to represent the exploitation goal

and c is a parameter that represents your preferred balance between exploitation and exploration and

The UCT formula gives you a score for each node, and the node with the highest UCT score is the start of the next rollout.



With a larger value of c, you’ll spend more time visiting the least explored nodes. With a smaller value of c, you’ll spend more time gathering a better evaluation of the most promising node. The choice of c that makes the most effective game player is usually found via trial and error.

**GO and Machine Learning: CNN**

Before discussing neural networks, let’s start with a concrete use case. If you want to make a program to predict digits from handwritten image data reasonably well, with about 95% accuracy, it would be a very difficult task.

But you can do it by exposing only the pixel values of the images to a neural network; the algorithm will learn to extract relevant information about the structure of digits on its own.

And that’s the power of machine learning in general.

But in our case it would be very helpful if a program can determine not only the legal moves to play a go game but the logical ones that can lead to winning, and we can accomplish that by treating the game board as the digits images from earlier and feed it to the neural network and the output would be the best logical move to play from that board state.

Here you hope to use machine learning to get a fast approximation to a slow tree search. This concept is a key part of AlphaGo Zero, the strongest version of AlphaGo.

We can use the NN approach to lessen the width of the search tree by only considering the best moves and not all of the legal moves.

In Go, you often see particular local patterns of stones over and over again. Human players learn to recognize dozens of these shapes, and often give them evocative names (like *tiger’s mouth*, *bamboo joint*, or my personal favorite, the *rabbitty six*). To make decisions like a human, our Go AI will also have to recognize many local spatial arrangements. A particular type of neural network called a *convolutional network* is specially designed for detecting spatial relationships like this. Convolutional neural networks, or CNNs, have many applications beyond games: you’ll find them applied to images, audio, and even text.

By training the CNN on games played by experts on the game of go we would be accomplishing what we spoke about in the previous paragraphs which is making our go bot imitate the human players by recognizing patterns on the board.

You can get your hands-on games played by experts from any go server in The Smart Game Format (SGF), initially called Smart Go Format, has been developed since the late 80s. Its current, fourth major release was released in the late 90s.

SGF is a straightforward, text-based format that can be used to express games of Go, variations of Go games (for instance, extended game commentaries by professional players), and other board games.

after parsing these games, you now have a data set to train your CNN.

That’s the general idea of how to use CNNs to predict good go moves but this procedure will only get you so far.

By that I mean that your bot will not exceed the expert player’s level.

You can improve your CNN to be superhuman by a technique called reinforcement learning which we will tack about in the next section

**Learning by practice: reinforcement learning**

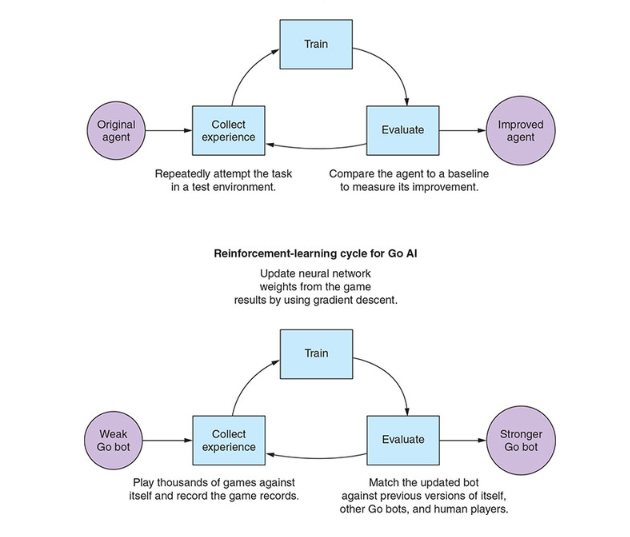
If you read a dozen books on Go, all written by strong pros from China, Korea, and Japan. yet you will be just an intermediate amateur player. Why can’t you reach the level of these legendary players? Have I forgotten their lessons?

Nobody knows the full recipe for becoming a top Go star, but we know at least one difference between the amateurs and Go professionals: practice.

A Go player probably clocks in five or ten thousand games before qualifying as a professional. Practice creates knowledge, and sometimes that’s knowledge that you can’t directly communicate.

You can *summarize* that knowledge—that’s what makes it into Go books. But the subtleties get lost in the translation. If I expect to master the lessons I’ve read, I need to put in a similar level of practice.

If practice is so valuable for humans, what about computers? Can a computer program learn by practicing? That’s the promise of *reinforcement learning*. In reinforcement learning (RL), you improve a program by having it repeatedly attempt a task. When it has good outcomes, you modify the program to repeat its decisions. When it has bad outcomes, you modify the program to avoid those decisions. This doesn’t mean you write new code after each trial: RL algorithms provide automated methods for making those modifications.



Reinforcement learning isn’t a free lunch. For one thing, it’s slow: your bot will need to play thousands of games in order to make a measurable improvement. In addition, the training process is fiddly and hard to debug. But if you put in the effort to make these techniques work for you, the payoff is huge. You can build software that applies sophisticated strategies to tackle a variety of tasks, even if you can’t describe those strategies yourself.

Many algorithms implement the mechanics of reinforcement learning, but they all work within a standard framework. This section describes the reinforcement-learning cycle, in which a computer program improves by repeatedly attempting a task

The goal of reinforcement learning is to make the agent as effective as possible. In this case, you want your agent to win at Go.

1. you have your Go bot play a batch of games against itself; during each game, it should record every turn and the final outcome. These game records are called its *experience*.
2. you *train* your bot by updating its behavior in response to what happened in its self-play games. This process is similar to training the neural networks which you are familiar with.

The core idea is that you want the bot to repeat the decisions it made in games it won, and stop making the decisions it made in games it lost.

The training algorithm comes as a package deal with the structure of your agent: you need to systematically modify the behavior of your agent in order to train. There are many algorithms for doing this such as with the *policy gradient* algorithm, Q*-learning* algorithm and the *actor-critic* algorithm.

1. After training, you expect your bot to be a bit stronger. But there are many ways for the training process to go wrong, so it’s a good idea to evaluate the bot’s progress to confirm its strength. To evaluate a game-playing agent, have it play more games. You can pit your agent against earlier versions of itself to measure its progress. As a sanity check, you can also periodically compare your bot to other AIs or play against it yourself.

Then you can repeat this entire cycle indefinitely:

* Collect experience
* Train
* Evaluate

**Deep Mind’s Alpha go**

The most successful go bots are of course Deep Mind’s alpha go and alpha go zero but that doesn’t mean that there are no others.

Prior to 2015, the best Go programs only managed to reach [amateur dan](https://en.wikipedia.org/wiki/Go_ranks_and_ratings#Kyu_and_dan_ranks) level. On the small 9×9 board, the computer fared better, and some programs managed to win a fraction of their 9×9 games against professional players. Prior to AlphaGo, some researchers had claimed that computers would never defeat top humans at Go.

in October 2015, [Google DeepMind](https://en.wikipedia.org/wiki/Google_DeepMind) program [AlphaGo](https://en.wikipedia.org/wiki/AlphaGo) beat [Fan Hui](https://en.wikipedia.org/wiki/Fan_Hui), the European Go champion, five times out of five in tournament conditions.

In March 2016, AlphaGo beat [Lee Sedol](https://en.wikipedia.org/wiki/Lee_Sedol) in the first three of five matches. This was the first time that a [9-dan](https://en.wikipedia.org/wiki/9-dan) master had played a professional game against a computer without handicap. Lee won the fourth match, describing his win as "invaluable". AlphaGo won the final match two days later.

In May 2017, AlphaGo beat [Ke Jie](https://en.wikipedia.org/wiki/Ke_Jie" \o "Ke Jie), who at the time was ranked top in the world, in a [three-game match](https://en.wikipedia.org/wiki/AlphaGo_versus_Ke_Jie) during the [Future of Go Summit](https://en.wikipedia.org/wiki/Future_of_Go_Summit).

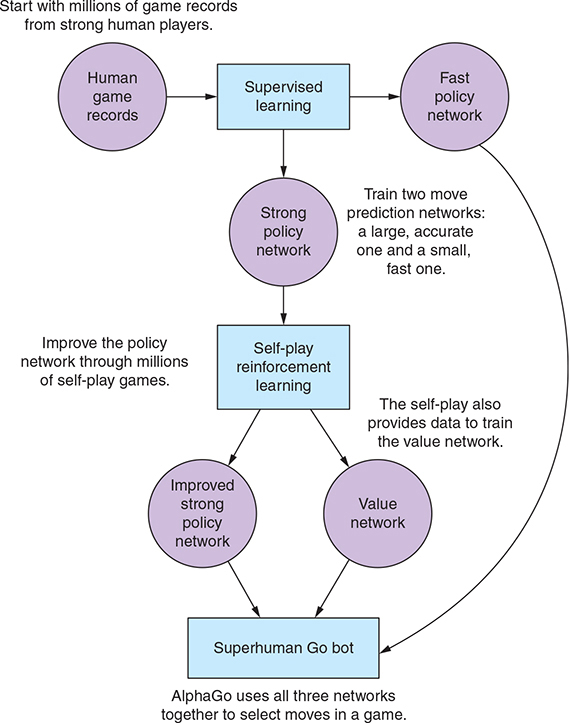
In October 2017, DeepMind revealed a new version of AlphaGo, trained only through self-play, that had surpassed all previous versions, beating the Ke Jie version in 89 out of 100 games.

Since the basic principles of AlphaGo had been published in the journal *Nature*, other teams were able to produce high-level programs. By 2017, both [Zen](https://en.wikipedia.org/wiki/Zen_(software)) and [Tencent](https://en.wikipedia.org/wiki/Tencent)'s project [Fine Art](https://en.wikipedia.org/wiki/Fine_Art_(software)) were capable of defeating very high-level professionals some of the time and the open source [Leela Zero](https://en.wikipedia.org/wiki/Leela_Zero) engine was released.

So, Let’s understand what Deep Mind did to accomplish superhuman level in go.

* You start off by training two deep convolutional neural networks (policy networks) for move prediction. One of these network architectures is a bit deeper and produces more-accurate results, whereas the other one is smaller and faster to evaluate. We’ll call them the strong and fast policy networks, respectively.
* The strong and fast policy use a deeper architecture to implement the Convolutional neural networks we discussed before.
* After the first training step of policy networks is complete, you take the strong policy network as a starting point for self-play (RL) ,If you do this with a lot of compute power, this will result in a massive improvement of your bots.
* As a next step, you take the strong self-play network to derive a value network, this completes the network training stage, and you don’t do any deep learning after this point.
* To play a game of Go, you use tree search as a basis for play, but instead of plain Monte Carlo rollouts, you use the fast policy network to guide the next steps. Also, you balance the output of this tree-search algorithm with what your value function tells you.

Performing this whole process from training policies, to self-play, to running games with search on a superhuman level requires massive compute resources and time.



Our approach

After understanding how alpha go works and how deep mind solved the problem

We will base our approach on the book by max pumberla called deep learning and the game of go

We choose an approach similar to theirs but we took into consideration our time and processing power limits.

1.we will start with supervised learning so that we have an agent that can predict expert moves from a given board state

2.improving that agent using RL techniques as we discussed before

3.improving the monte Carlo algorithm by using the two networks from above to guide the exploitation and exploration

4.till now we have an agent that can predict the best play

We will analyze the performance after completing the RL stage and if the agent is too slow to use in rollouts(exploitation) we will use the supervised network in rollouts and the improved (stronger) network in selecting the moves (exploration)

For the human vs bot mode

1. We will recommend the best move using our networks
2. When the human plays his move, we will evaluate his move by doing simulations and congratulating him if his move turned out to be better than ours
3. After finishing the game, we will save the game as experience and after a certain number of games we will train our model again to learn from the human player

Work Load

|  |  |
| --- | --- |
| Ahmed Salama | Mohamed Talaat |
| Researching the problem | Researching the problem |
| Monte carlo tree search | CNN |
| RL | RL |
| Alpha go zero | Alpha go |
| Deep learning and the game of go(book) | Deep learning and the game of go(book) |
| Understanding beta go | Understanding beta go |