AlphaGo
Superhuman artificial intelligence in exponentially growing spaces

Petar Veličković
(with Andrej Ivašković)

University of Cambridge
CAMBRIDGE CODING ACADEMY
Evening Tech Talk Series

25 August 2016
About me

- Research Assistant in Computational Biology/PhD student in the Artificial Intelligence Group of the University of Cambridge’s Computer Laboratory;

- Industrial/research experience with Microsoft, Jane Street and Nokia Bell Labs.

- Interested in integrating machine learning techniques with complex networks, particularly in low-data environments.
Introduction

- AlphaGo is a Go-playing agent developed by Google DeepMind—after convincingly defeating Fan Hui (5:0), the European Go champion, it has successfully defeated one of the best players of all time, Lee Se-dol (4:1).

- Without any overemphasis, this is one of the greatest achievements of artificial intelligence to date—with many experts predicting that such a result is beyond our reach for at least the following ten years.

- Streams of all five games (with detailed expert commentary) may be found on DeepMind’s YouTube channel.

- This talk represents a “digest” version of the original AlphaGo paper: “Mastering the game of Go with deep neural networks and tree search” (Silver et al., Nature, 2016).
AlphaGo challenge match
Hang on...

- An actual quote from one of my relatives: “Why are all of you AI people always so proud of solving games? Why don’t you solve something useful for a change…”

- This is indeed a good question! Let’s try to properly answer.

- Games provide an excellent environment for the development of AI, as they have precisely defined rules! The real world is often far worse (and commonly subjective…)

- One of the major implications of AlphaGo’s success is that in exponentially complex spaces (such as Go), it is possible to demonstrate superhuman performance without investigating any significant proportion of this space!
Imperfect rules...

Examples of real life being unfair:

Handwritten digit recognition (MNIST)

Image classification (ImageNet)

Figure 1. 'Popsicle' and 'Plunger' Example Images
Game hierarchy (xkcd.com/1002/)
(Outdated) game hierarchy (xkcd.com/1002/)

Computers can beat top humans

Computers still lose to top humans

(But focused R&D could change this)
Tic-Tac-Toe

- Hopefully, everyone is familiar with how this game works.

- There’s a (relatively) small number of states of the board (no more than $9! = 362880$) which we can represent as nodes in a directed acyclic graph; the edges would represent moves.
First few potential moves
Exhaustive search

- Given the relatively small state space, we may exhaustively evaluate *all of them*, backtracking from terminal states to figure out which states are *winning for X*, which are *losing for X*, and which will end in a *draw* (assuming optimal play).

- This way, we can also determine which move should be optimally played at each state of the board.
Optimal strategies (xkcd.com/832)
Chess is a $\approx 1500$ years-old game.
Research initiated in the 1940s; first chess-playing agents developed by the end of the 1950s; high-ranked players get defeated already in the 1970s.
In 1996, *Deep Blue* loses $4-2$ to Garry Kasparov. The following year, Deep Blue defeats Kasparov $3\frac{1}{2}-2\frac{1}{2}$.
Introduction

Key algorithms

Chess and Arimaa

Deep Blue

- Developed for seven years by IBM.
- 256 CPUs, 120 MHz each ($\sim 1.138 \times 10^{10}$ FLOPS).
How does it work?

▶ The key ingredient is an **evaluation function**, assigning a **value** to each state of the chessboard—it depends on parameters such as:
  ▶ material advantage;
  ▶ positional advantage;
  ▶ safety of the king;
  ▶ tempo...

▶ Designing hand-crafted evaluation functions is a very difficult problem, and knowledge of a good such function for one task typically does not help with a different task.
We apply a similar algorithm as for tic-tac-toe, but we only expand *up to a certain move depth*. We compute the evaluation function values on all nodes where the search stops, and determine an optimal move based on them.

A modern computer still needs \( \approx 2 \text{ min} \) for a depth of 3–4 moves. How to reach grandmaster level?

The issue lies in the fact that many of the states we explore are suboptimal—if we have previously encountered a solution that the currently considered move cannot possibly outperform, we may safely *ignore* all moves expanding from it.

This is the \( \alpha-\beta \) *pruning* algorithm, which suffices for reaching grandmaster level of play.
α–β pruning example

- As an example, we consider a game where three moves can be played at any stage, and we expand for a depth of two moves.
- The values of the bottom-level nodes represent the evaluation function values (i.e. the value of those positions for blue).
As an example, we consider a game where three moves can be played at any stage, and we expand for a depth of two moves. The values of the bottom-level nodes represent the evaluation function values (i.e., the value of those positions for blue).
\(\alpha-\beta\) pruning example

- As an example, we consider a game where three moves can be played at any stage, and we expand for a depth of two moves.
- The values of the bottom-level nodes represent the evaluation function values (i.e. the value of those positions for blue).
As an example, we consider a game where three moves can be played at any stage, and we expand for a depth of two moves. The values of the bottom-level nodes represent the evaluation function values (i.e. the value of those positions for blue).
As an example, we consider a game where three moves can be played at any stage, and we expand for a depth of two moves. The values of the bottom-level nodes represent the evaluation function values (i.e. the value of those positions for blue).
As an example, we consider a game where three moves can be played at any stage, and we expand for a depth of two moves. The values of the bottom-level nodes represent the evaluation function values (i.e. the value of those positions for blue).
As an example, we consider a game where three moves can be played at any stage, and we expand for a depth of two moves. The values of the bottom-level nodes represent the evaluation function values (i.e. the value of those positions for blue).
As an example, we consider a game where three moves can be played at any stage, and we expand for a depth of two moves. The values of the bottom-level nodes represent the evaluation function values (i.e. the value of those positions for blue).
I will briefly mention Arimaa, a game for which grandmaster-level AI was developed only last year.

The winning program, named SHARP, was designed by David Wu (of Jane Street, where I have interned last year).

The game is designed with the following goals in mind:

- being playable with chess figures;
- being far slower in tempo;
- being far more “visual” in strategy terms...
- …therefore, compared to chess, it is simpler for humans to form long-term strategies, and significantly harder for computers to explore the board state space.

SHARP’s algorithm uses, at its core, exactly the same methods as for chess ($\alpha$-$\beta$ pruning, with clever state evaluation functions).
Arimaa
Rules of Go

- Go is played on an (initially empty) $19 \times 19$ board.
- Black and White take turns in placing stones on the board; Black plays first.
- Players may skip their turn at any moment, but have to sacrifice one of their stones if they do so.
- The game ends after two consecutive turns skipped, with White always having the final turn.
The goal of the game is to *surround more territory* than the opponent using the player’s stones.

If Player A manages to completely surround a group of Player B’s stones, that group is removed from the board.

**Superko rule**: no state of the board may be repeated.
Difficulties

- The board is significantly larger than a chessboard—**exponentially** more possible states.

- State representation may be problematic, with needing to satisfy the superko rule.

- The game is inherently *visual* rather than rule-based, and thus predisposed for humans!

- Despite all of the above, *AlphaGo* managed to achieve the seemingly impossible at this time. **How?**
Monte Carlo Tree Search

- $\alpha-\beta$ pruning is insufficiently “aggressive” for a game like Go. In order to perform a move within the time restrictions, one needs to explore *significantly fewer* moves per depth level!

- Enter *Monte Carlo Tree Search* (*MCTS*); based on the knowledge gathered thus far, we will only expand a *very small subset* of potential moves, and *ignore* all others!
MCTS

AlphaGo: Superhuman artificial intelligence in exponentially growing spaces

Petar Veličković
Policy function

- Of course, we cannot successfully *cold-start* this algorithm without any prior beliefs—the branching factor is too large!

- We need a *function* that will determine an *initial judgement* of the relative effectiveness of *all* possible moves for a particular board state—we will call it the *policy function*.

- As the MCTS procedure is iterated, we may update the policy function values upon discovering more rewarding paths.

- Clearly, exactly representing this function would require as much effort as expanding the entire subtree. We need to *approximate* it somehow...
Neural networks

- Enter **neural networks**—structures of interconnected processing units (*neurons*) capable of precisely approximating a target function from a sufficiently large amount of labelled (input, output) examples.

- Each neuron computes a linear combination of its *inputs*, afterwards potentially applying an *activation function*, to produce its *output*.

- The particular neural network used in *AlphaGo* is a *deep feedforward convolutional neural network* (CNN). I will not focus on its specifics here—it is simply a special case of the general architecture that I will present.
A single neuron

Within this context sometimes also called a *perceptron* (...)

$$h(\vec{x}; \vec{w}) = \sigma \left( b + \sum_{i=1}^{n} w_i x_i \right)$$

Popular choices for the activation function $\sigma$:

- $\sigma(x) = x$ (*identity*);
- $\sigma(x) = \max(0, x)$ (*rectified linear unit* (ReLU));
- $\sigma(x) = \frac{1}{1+\exp(-x)}$; $\sigma(x) = \tanh x$ (*sigmoid functions*).
The most potent architecture is completely unrestricted and feedforward—sometimes also called a *multilayer perceptron* (MLP).
Introduction

Neural networks

A few details

- Neural networks are trained from known (input, output) samples. The training algorithm adapts the neurons’ weights to maximise *predictive power* on the training examples. Key words: *backpropagation, stochastic gradient descent*.

- It was (mathematically) proven that a network with just one hidden layer of sigmoid neurons is capable of arbitrarily precisely approximating *any continuous real function!* (Cybenko’s universal approximation theorem)

- Unfortunately, the proof is not constructive (does not provide a training algorithm for achieving this capability), and therefore a typical approach is to increase the number, rather than the size, of the hidden layers (*deep learning*).
Training the policy network

▶ For learning the policy function, the neural network was trained to be able to predict how likely it is that a human expert would play a particular move in a given position.

▶ This was the initial project for DeepMind, prior to any actual game-playing agents.

▶ Trained from a set of 30 million (position, move) pairs. Achieved 67% accuracy (> 44%).

▶ The ultimate goal was to be better than a human expert, however—so these values were further optimised purely from self-play! (using a reinforcement learning approach)
Policy network
Discarding a vast majority of moves with help from the policy network is significantly helpful for increasing performance, especially if our initial policy values are reasonably accurate.

A game of Go can last very long—in most cases we cannot reliably simulate a game until the end, and therefore we stop searching at some point. Similarly to chess, we need an evaluation function for this position.

However, it is often useful to get at least a rough estimate of what would happen if we played in this fashion until the end. For this purpose, a very shallow variant of the policy network, called the rollout network, was trained solely on expert moves.
Value network

We learn to approximate the evaluation function through another neural network, trained entirely from self-play.

N.B. this circumvents the previous difficulties of designing hand-crafted evaluation functions!
Putting it all together... (training)
Putting it all together… (MCTS)
Thank you!

Questions?

pv273@cam.ac.uk