Your Own AlphaGo

Advanced Play using only a Laptop

By Niall Cardin

niallc@gmail.com

A Machine that Plays Games

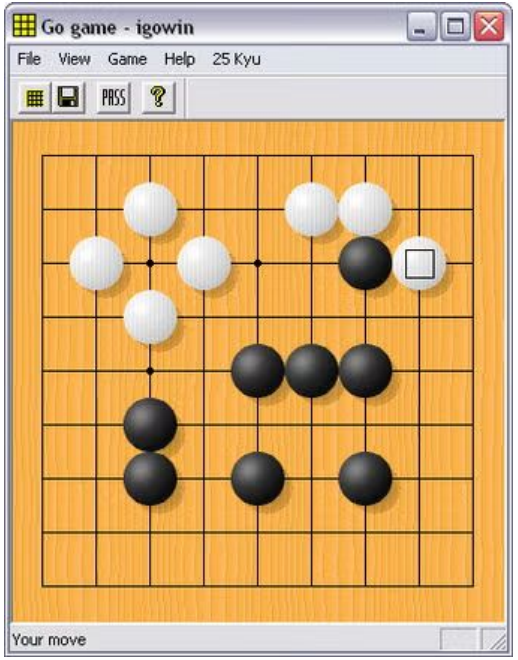
When I was a teenager I found a program called Igowin.

It was **so cool** that it could play go!

How can it do that?

Back in 1998 though, computers weren't good at Go.

(People had tried neural nets!)



Author: David Fotland
Smart Games LLC

AlphaGo

2016: DeepMind's AlphaGo surpassed humans.

I bumped into David Fotland ( ) at a party!

He told me it wasn't *that* hard, read the paper.

Article | Published: 27 January 2016

Mastering the game of Go with deep neural networks and tree search

David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel & Demis Hassabis

Nature **529**, 484–489 (2016) | Cite this article

Can a hobbyist make “an AlphaGo”?

It works fine if you just try but it might disappoint after a while

Want to touch on one important detail (only have 5 minutes so I can’t talk about everything I’d like to, there’s really a lot more to say!)

Using one laptop you **can** produce advanced, if not superhuman, play.

Note: others[1] did this way before me, and better.

[1] Leela, KataGo, ...

What can a Neural Network do?

For our purposes a neural network is a “powerful function fitter”.

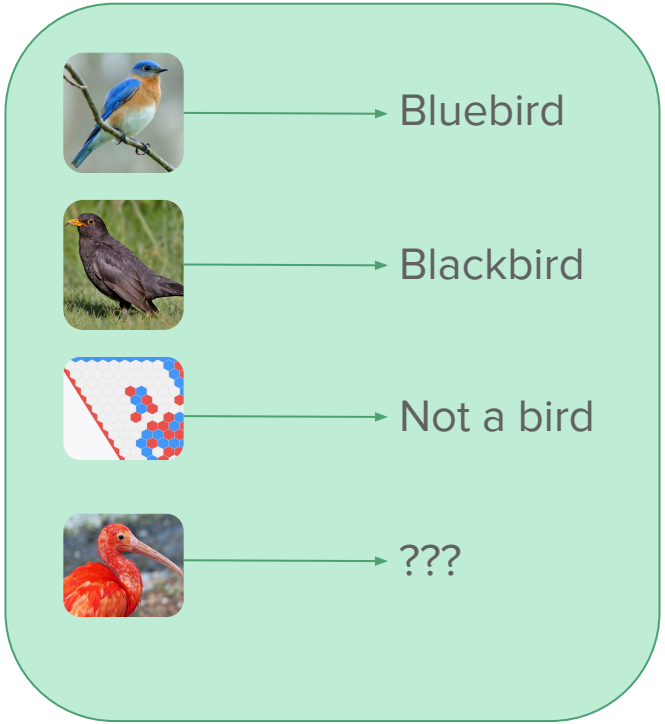
Step 1: Train on many ‘labeled’ examples of $A_i \rightarrow B_i$.

Step 2: You have an unlabeled A^* ask the network for B^*.

How: Neural nets are like stacked regression models that insert ‘hidden’ intermediate nodes. We then update to find good intermediate values via “back propagation” using “automatic differentiation”.

This approach turns out to be hugely flexible, so it can learn to map many different types of A → B

How does this help us play games?



Playing Games with Neural Networks – Policy

What is playing a game? Why is it mapping A ➔ B?

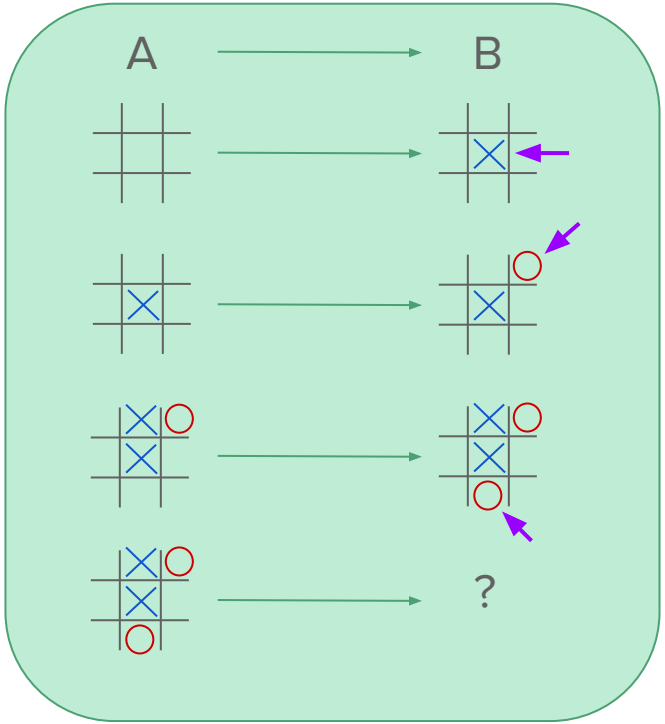
You repeatedly make a move given your situation.

So we're mapping (game-state ➔ move)

Given some games we can feed them to a neural network and it will learn to copy the moves.

This let’s us effectively imitate moves from a dataset

The full design lets play better than the training.



If there’s time (unlikely), we can talk about the fuller design

Is Training Going Well?

Training minimizes a loss function. Over time, loss usually improves.

That’s not always better.

1. The games we train on may not be the best
2. The model may fit too narrowly, to those specific games

Better evaluation: Do we win more games?

Hold tournaments between training checkpoints.

Computers but mostly do the same thing every time, by default

Randomness

Our network roughly produces a probability for each possible next move:

- 47% ➜ Move 1
- 18% ➜ Move 2
  9% ➜ Move 3
  ...
- 0.01% ➜ Move 361

We could play be choosing each move with that probability.

There are other (better) schemes for randomness, but this will do here.

Which model is better?

Suppose typical move values we as below:

Model 1

	Model Score	Real Value
Move 1	**90%**	7/10
Move 2	5%	5/10
Move 3 …	2%	3/10
Move 13	0.01%	10/10

Model 2

	Model Score	Real Value
Move 1	**35%**	10/10
Move 2	15%	7/10
Move 3 …	10%	5/10
Move 13	0.1%	1/10

With randomness, model 2 will play many bad moves.

Without randomness, model 2 plays great moves. Model 1 plays decent but not great moves.

Intuitively, I'd actually prefer Model 2 to Model 1.

A Better Evaluation

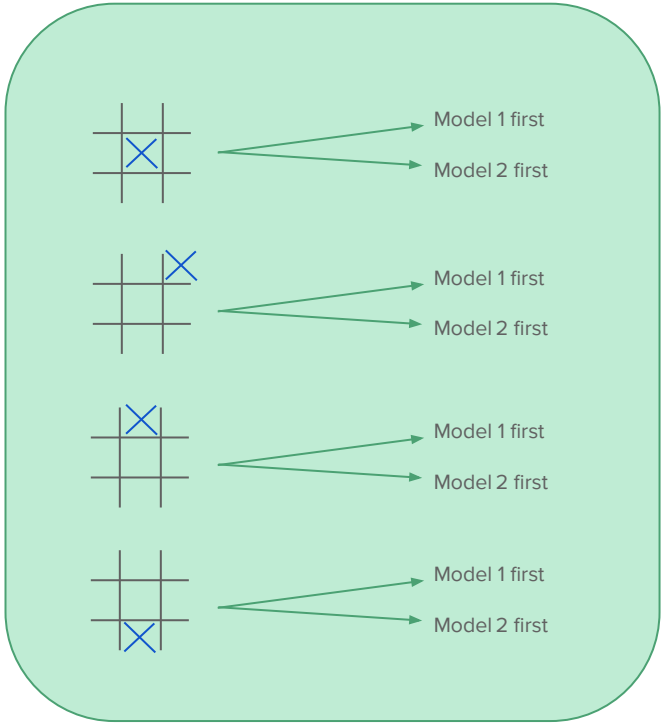
Have a library of existing games already

Sample a unique collection of openings

Play deterministically from those openings

Have each contestant play first per opening.

Winner: Model that’s best at picking the best move



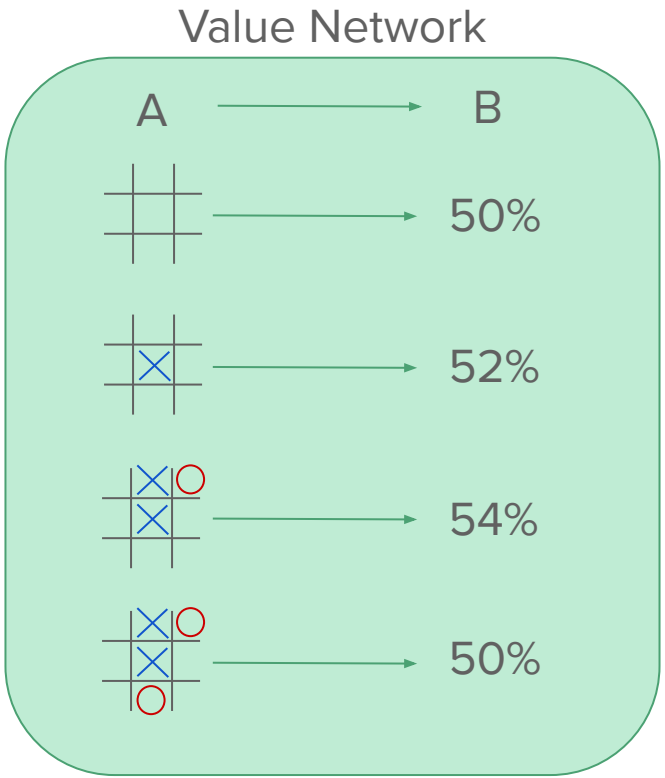
Extra Time: Value Network

Learning to play **better than the training data** – the really cool part

A 2nd “value” network maps position ➔ probability of a win.

This allows reading ahead to evaluate moves with tree search. Efficient tree search can gives games better than the training data.

Iterate training ➔ self-play ➔ training.



If there’s time (unlikely), we can talk about the fuller design

Extra Time 2 – Returning to Evaluation

To use self-play ➔ new training data, we need to use randomness again.

Does that mean evaluating deterministic play isn't quite right?

Ultimately the ideal solution is:

1. Decide how much randomness you need
2. For each model, tune the tree search to be as strong as possible, given 1.
3. Choose the best (model, tree search) combination to create new games

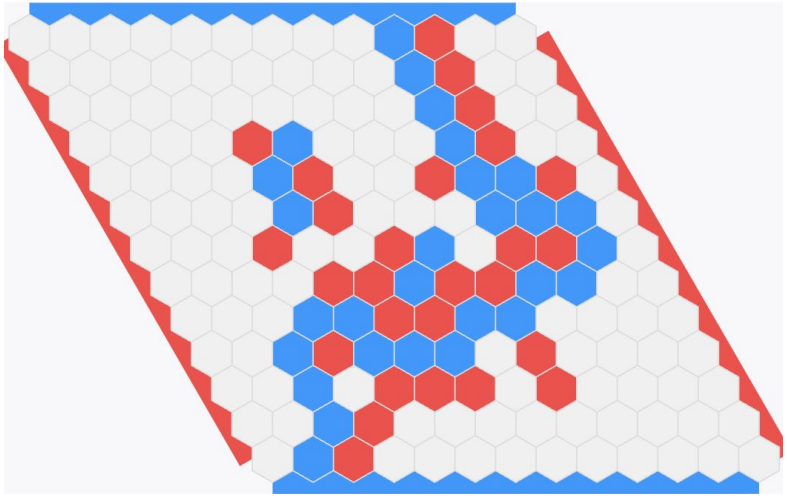
But this is computationally expensive. So without infinite compute we need to take shortcuts and use intuition. I've had good results using randomized openings.

Thanks for Listening!

Try it out!

A program to play hex (great game!).

sf25.niallcardin.com



github.com/niallc/Snowflake2025