STA314: Statistical Methods for Machine Learning I

Lecture 12 - AlphaGo and game-playing

Chris Maddison

University of Toronto, Fall 2021
Recap of different learning settings

So far the settings that you’ve seen imagine one learner or agent.

**Supervised**
Learner predicts labels.

**Unsupervised**
Learner organizes data.

**Reinforcement**
Agent maximizes reward.
Today

We will talk about learning in the context of a two-player game.

Game-playing

This lecture only touches a small part of the large and beautiful literature on game theory, multi-agent reinforcement learning, etc.
Game-playing in AI: Beginnings

• (1950) Claude Shannon proposes explains how games could be solved algorithmically via tree search
• (1953) Alan Turing writes a chess program
• (1956) Arthur Samuel writes a program that plays checkers better than he does
• (1968) An algorithm defeats human novices at Go

slide credit: Profs. Roger Grosse and Jimmy Ba
Game-playing in AI: Successes

- (1992) TD-Gammon plays backgammon competitively with the best human players
- (1996) Chinook wins the US National Checkers Championship
- (1997) DeepBlue defeats world chess champion Garry Kasparov

slide credit: Profs. Roger Grosse and Jimmy Ba
Today

• Game-playing has always been at the core of CS.
  • Simple well-defined rules, but mastery requires a high degree of intelligence.

• We will study how to learn to play Go.
  • The ideas in this lecture apply to all zero-sum games with finitely many states, two players, and no uncertainty.
  • Go was the last classical board game for which humans outperformed computers.
  • We will follow the story of AlphaGo, DeepMind’s Go playing system that defeated the human Go champion Lee Sedol.

• Combines many ideas that you’ve already seen.
  • supervised learning, value function learning...
The game of Go: Start

- Initial position is an empty $19 \times 19$ grid.
The game of Go: Play

• 2 players alternate placing stones on empty intersections. Black stone plays first.

• (Ko) Players cannot recreate a former board position.
The game of Go: Play

- **Capture** Capture and remove a connected group of stones by surrounding them.
The game of Go: End

- **Territory** The winning player has the maximum number of occupied or surrounded intersections.
Outline of the lecture

To build a strong computer Go player, we will answer:

• What does it mean to play optimally?
• Can we compute (approximately) optimal play?
• Can we learn to play (somewhat) optimally?
Why is this a challenge?

- Optimal play requires searching over $\sim 10^{170}$ legal positions.
- It is hard to decide who is winning before the end-game.
  - Good heuristics exist for chess (count pieces), but not for Go.
- Humans use sophisticated pattern recognition.
Optimal play
Game trees

• Organize all possible games into a tree.
  • Each node $s$ contains a legal position.
  • Child nodes enumerate all possible actions taken by the current player.
  • Leaves are terminal states.
  • Technically board positions can appear in more than one node, but let’s ignore that detail for now.

• The Go tree is finite (Ko rule).
Game trees

black stone’s turn

white stone’s turn

black stone’s turn

white stone’s turn

black stone’s turn
Evaluating positions

- We want to quantify the utility of a node for the current player.
- Label each node $s$ with a value $v(s)$, taking the perspective of the black stone player.
  - $+1$ for black wins, $-1$ for black loses.
  - Flip the sign for white’s value (technically, this is because Go is zero-sum).
- Evaluations let us determine who is winning or losing.
Evaluating leaf positions

Leaf nodes are easy to label, because a winner is known.

-1 +1 +1 +1 -1 -1 +1 +1 +1 -1 -1 -1 -1 -1 +1 -1
black stones win white stones win
Evaluating internal positions

- The value of internal nodes depends on the strategies of the two players.
- The so-called maximin value $v^*(s)$ is the highest value that black can achieve regardless of white’s strategy.
- If we could compute $v^*$, then the best (worst-case) move $a^*$ is

$$a^* = \arg \max_a \{ v^*(\text{child}(s, a)) \}$$
Evaluating positions under optimal play
Evaluating positions under optimal play
Evaluating positions under optimal play

\[ v^*(s) = +1 \]

\[ \max \]

\[ \min \]

\[ \max \]

\[ \min \]
Value function $v^*$

• $v^*$ satisfies the **fixed-point equation**

\[
    v^*(s) = \begin{cases} 
        \max_a \{ v^*(\text{child}(s, a)) \} & \text{black plays} \\
        \min_a \{ v^*(\text{child}(s, a)) \} & \text{white plays} \\
        +1 & \text{black wins} \\
        -1 & \text{white wins} 
    \end{cases}
\]

• Analog of the optimal value function of RL.

• Applies to other two-player games
  • Deterministic, zero-sum, perfect information games.
What is the maximin value $v^*(s)$ of the root?

1. -1?
2. +1?

Recall: black plays first and is trying to maximize, whereas white is trying to minimize.
Quiz!

What is the maximin value $v^*(s)$ of the root?

1. -1?
2. +1?

Recall: black plays first and is trying to maximize, whereas white is trying to minimize.
In a perfect world

• So, for games like Go, all you need is $v^*$ to play optimally in the worst case:

$$a^* = \arg \max_a \{v^*(\text{child}(s, a))\}$$

• Claude Shannon (1950) pointed out that you can find $a^*$ by recursing over the whole game tree.

• Seems easy, but $v^*$ is wildly expensive to compute...
  • Go has $\sim 10^{170}$ legal positions in the tree.
Approximating optimal play
Depth-limited Minimax

- In practice, recurse to a small depth and back off to a static evaluation $\hat{v}^\ast$.
  - $\hat{v}^\ast$ is a heuristic, designed by experts.
  - Other heuristics as well, e.g. pruning.
  - For Go (Müller, 2002).
Progress in Computer Go

Minimax search for Go

adapted from Sylvain Gelly & David Silver, Test of Time Award ICML 2017
Expected value functions

- Designing static evaluation of $v^*$ is very challenging, especially so for Go.
  - Somewhat obvious, otherwise search would not be needed!
- Depth-limited minimax is very sensitive to misevaluation.
- Monte Carlo tree search resolves many of the issues with Minimax search for Go.
  - Revolutionized computer Go.
  - To understand this, we will introduce **expected value functions**.
Expected value functions

If players play by rolling fair dice, outcomes will be random.

This is a decent approximation to very weak play.
Expected value functions

Averaging many random outcomes $\rightarrow$ expected value function.

$$v(s) = -1/9$$

Contribution of each outcome depends on the length of the path.
Consider two players that pick their moves by flipping a fair coin, what is the expected value $v(s)$ of the root?

1. $1/3$?
2. $1/2$?
Consider two players that pick their moves by flipping a fair coin, what is the expected value $v(s)$ of the root?

1. $1/3$?
2. $1/2$?
**Expected value functions**

- Noisy evaluations $v_n$ are cheap approximations of **expected outcomes**:

  \[
  v_n(s) = \frac{1}{n} \sum_{i=1}^{n} o(s'_i) \\
  \approx \mathbb{E}[o(s') := v(s)]
  \]

  $o(s) = \pm 1$ if black wins / loses.

- Longer games will be underweighted by this evaluation $v$, but let's ignore that.
Monte Carlo tree search

- Ok expected value functions are easy to approximate, but how can we use $v_n$ to play Go?
  - $v_n$ is not at all similar to $v^*$.
  - So, maximizing $v_n$ by itself is probably not a great strategy.
  - Minimax won’t work, because it is a pure exploitation strategy that assumes perfect leaf evaluations.

- Monte Carlo tree search (MCTS; Kocsis and Szepesvári, 2006; Coulom, 2006; Browne et al., 2012) is one way.
  - MCTS maintains a depth-limited search tree.
  - Builds an approximation $\hat{v}^*$ of $v^*$ at all nodes.
Monte Carlo tree search

- Select an existing leaf or expand a new leaf.
- Evaluate leaf with Monte Carlo simulation $v_n$.
- Noisy values $v_n$ are backed-up the tree to improve approximation $\hat{v}^*$.
Monte Carlo tree search

• Selection strategy greedily descends tree.
• MCTS is robust to noisy misevaluation at the leaves, because
  the selection rule balances exploration and exploitation:

\[
a^* = \arg \max_a \left\{ \hat{v}^*(\text{child}(s, a)) + \sqrt{\frac{2 \log N(s)}{N(\text{child}(s, a))}} \right\}
\]

• \(\hat{v}^*(s)\) = estimate of \(v^*(s)\), \(N(s)\) number of visits to node \(s\).
• MCTS is forced to visit rarely visited children.
• Key result: MCTS approximation \(\hat{v}^* \to v^*\) (Kocsis and Szepesvári, 2006).
Progress in Computer Go

Monte Carlo tree search for Go

adapted from Sylvain Gelly & David Silver, Test of Time Award ICML 2017
Scaling with compute and time

• The strength of MCTS bots scales with the amount of compute and time that we have at play-time.
• But play-time is limited, while time outside of play is much more plentiful.
• How can we improve computer Go players using compute when we are not playing? Learning!
  • You can try to think harder during a test vs. studying more beforehand.
Learning to play Go
This is where I come in

• 2014 Google DeepMind internship on neural nets for Go.
  • Working with Aja Huang, David Silver, Ilya Sutskever, I was responsible for designing and training the neural networks.
  • Others came before (e.g., Sutskever and Nair, 2008).
• Ilya Sutskever’s (Chief Scientist, OpenAI) argument in 2014: expert players can identify a good set of moves in 500 ms.
  • This is only enough time for the visual cortex to process the board—not enough for complex reasoning.
  • At the time we had neural networks that were nearly as good as humans in image recognition, thus we thought we would be able to train a net to play Go well.

• Key goal: can we train a net to understand Go?
Neural nets for Go

Neural networks are powerful parametric function approximators.

board \( s \)

\[
\text{net}(s, x)
\]

parameters \( x \)

Idea: map board position \( s \) (input) to a next move or an evaluation (output) using simple convolutional networks.
Neural nets for Go

- We want to train a neural policy or neural evaluator, but how?
- Existing data: databases of Go games played by humans and other compute Go bots.
- The first idea that worked was learning to predict expert’s next move.
  - Input: board position $s$
  - Output: next move $a$
Policy Net (Maddison et al., 2015)

- **Dataset**: KGS server games split into board / next-move pairs \((s_i, a_i)\)
  - 160,000 games → 29 million \((s_i, a_i)\) pairs.
- **Loss**: negative log-likelihood,
  \[
  - \sum_{i=1}^{N} \log \pi_{\text{net}}(a_i|s_i, x).
  \]

- Use trained net as a Go player:
  \[
  a^* = \arg \max_a \{ \log \pi_{\text{net}}(a|s, x) \}.
  \]

(Silver et al., 2016)
Like learning a better traversal

As supervised accuracy improved, searchless play improved.
Progress in Computer Go

Progress in my internship

adapted from Sylvain Gelly & David Silver, Test of Time Award ICML 2017
Can we improve MCTS with neural networks?

- These results prompted the formation of big team inside DeepMind to combine MCTS and neural networks.
- To really improve search, we needed strong evaluators.
  - **Recall**: an evaluation function tells us who is winning.
  - $\pi_{\text{net}}$ rollouts would be a good evaluator, but this is too expensive.
- Can we learn one?
Value Net (Silver et al., 2016)

Failed attempt.

- **Dataset**: KGS server games split into board / outcome pairs $(s_i, o(s_i))$
- **Loss**: squared error,
  
  $$\sum_{i=1}^{N} (o(s_i) - v_{net}(s_i, x))^2.$$ 

- **Problem**: Effective sample size of 160,000 games was not enough.

(Silver et al., 2016)
**Value Net** (Silver et al., 2016)

Successful attempt.

- Use Policy Net playing against itself to **generate millions of unique games**.
- **Dataset**: Board / outcome pairs \((s_i, o(s_i))\), each from a unique self-play game.
- **Loss**: squared error,

\[
\sum_{i=1}^{N} (o(s_i) - v_{\text{net}}(s_i, x))^2.
\]

(Silver et al., 2016)
**AlphaGo** (Silver et al., 2016)

- The Value Net was a very strong evaluator.

![AlphaGo Results](image)

- The final version of AlphaGo used rollouts, Policy Net, and Value Net together.
  - Rollouts and Value Net as evaluators.
  - Policy Net to bias the exploration strategy.
Progress in Computer Go

AlphaGo Team (Silver et al., 2016)

adapted from Sylvain Gelly & David Silver, Test of Time Award ICML 2017
Impact
Go is not just a game

• Go originated in China more than 2,500 years ago. Reached Korea in the 5th century, Japan in the 7th.

• In the Tang Dynasty, it was one of the four arts of the Chinese scholar together with calligraphy, painting, and music.

• The aesthetics of Go (harmony, balance, style) are as essential to top-level play as basic tactics.
2016 Match—AlphaGo vs. Lee Sedol

• Best of 5 matches over the course of a week.
• Most people expected AlphaGo to lose 0-5.
• AlphaGo won 4-1.
Human moments

Lee Sedol is a titan in the Go world, and achieving his level of play requires a life of extreme dedication.

It was humbling and strange to be a part of the AlphaGo team that played against him.
Game 2, Move 37
Thanks!

I played a key role at the start of AlphaGo, but the success is owed to a large and extremely talented team of scientists and engineers.

Course Evals

Use this time to finish course evals.


