Student's Handbook Guide to CMPUT 496

Search, knowledge, and simulation

Everything you need to know about CMPUT 496

When making decisions, computers rely on three main ideas: search, knowledge and simulations. Knowledge is created by machine learning techniques such as deep neural networks. Search and simulations help to understand the short and long-term consequences of possible actions. This course leads from basic concepts to recent successes such as DeepMind's AlphaGo program.

By slaying his assignment minions, defeating his midterm champion(s) and ending the final themselves. The heroic legacy of your achievement will be remembered on your academic transcript.

Contents

Part I

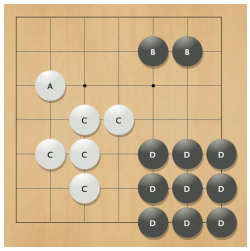
Problem Solving for Human and Computers

Go

- Go is a turn-based strategy game where the goal is to gain as much territory as possible. It has a simple rule-set but complex strategy.
- It consists of 19 by 19 grid that starts empty, and two players black and white. Where black goes first.
- A move consists of placing a stone at an intersection of the grid.
- Connections are stones of the same colors that are connected in a Manhattan pattern, no diagonal connections.
- Liberties are empty points adjacent to a block.

Game of Go Rules - Blocks

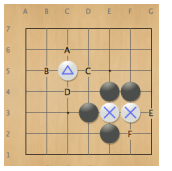
CMPUT 496

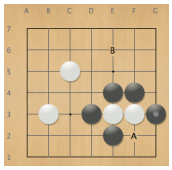


- Connected stones of the same color are called *blocks*
- A is a single stone block
- Two stones B are connected by a line. They are one block
- C is a single block of 5 white stones
- D is a block of 9 black stones
- A and C are *not* in the same block
  - No connection diagonally

Game of Go Rules - Liberties

CMPUT 496



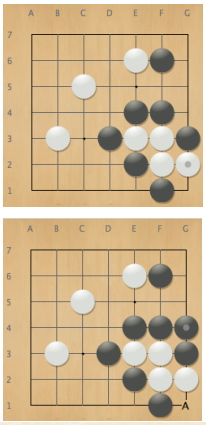


- Empty points adjacent to a block are called *liberties*
- The single marked white stone has four Liberties A, B, C, D
- The block of two marked black stones has two liberties, E and F
- After White plays on 1, the black stones have only one liberty left
- A block that loses its last liberty is *captured* (see next slide)

- Capture happens if a block loses all of its liberties.
- Suicide is an illegal move that is forbidden in most version of go. Capture takes precedence over suicide.
- Basic Ko is an illegal move that leads to repetition of the game state. If the next move leads to the same board state from the previous move then the move is illegal (For most cases).

Illegal Move - Suicide

CMPUT 496



- Same example, but Black to play
- Black at A would be *suicide*
- Black would take its own last liberty
- Suicide is forbidden in most versions of Go rules
  - In this course: we *never* allow suicide
- Capturing always takes precedence over suicide - see next slide

Repetition Rules - Basic Ko

CMPUT 496

- From top to middle picture: White can capture one black stone by playing A
- From middle to bottom picture: Now if Black captures back one white stone...
- The position would repeat, infinite loop
- This is called a (basic) ko.
- Go rules forbid such repetition

- A player can pass their move and allow their opponent to play. Usually, there are some moves better than a pass. Competent players only pass near the end of the game.
-The game ends after two successive passes.
- The person with the most territory and captured opponent's stone wins. The white player add the komi as an adjustment for going second. The winner is the player with the higher score. Draws are possible of the komi is an integer.

End of Game and Scoring

CMPUT 496

- Game ends after two successive passes
  - Some rule versions require three passes
- Next, count the score for each player - stones plus territory
- Add the *komi* (adjustment for going second)
- The winner is the player with higher score
- Draws are possible if the komi is integer

Here is an example of a scoring.

Scoring Example

CMPUT 496

- Assume komi = 7.5
- Black score = 37
  - 13 Black stones +
  - 24 empty points surrounded by Black
- White score = 51.5
  - 17 White stones +
  - 27 empty points surrounded by White +
  - 7.5 komi
- White wins by 51.5 - 37 = 14.5 points

Go program 0

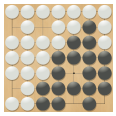
Go0: Random Player on 7 × 7 Board

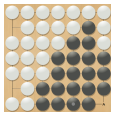
CMPUT 496

- `Go0` is our first example
- Algorithm:
  - Create list L of all legal moves on board
  - If L is empty, then play pass
  - Else select one move m from L uniformly at random
  - Play m
- Python 3 program: `Go0.py`
- Our demo uses a 7 × 7 board

Problem With Go0 Player

CMPUT 496





- `Go0` fills the board, but then ...
- It never seems to stop with two passes
- It cannot keep any stones safe
- It fills its own liberties and territories
- Eventually, even strong-looking stones get captured
- Game never ends...

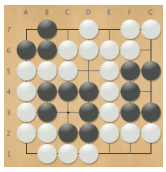
In order to fix this problem we have to eliminate certain moves from consideration. Moves that are obviously stupid.
We also have to make sure that the game ends in a reasonable amount of time.
We also have to make sure that safe stones don't get captured.
Surrounding territory is also a big factor in Go.
Filling one's own territory afterward is usually bad.

Fixing the case using one point eyes

A eye is a pint that is surrounded by one color. An eye makes stones safer as the opponent cannot place their stone within the eye without making a suicide, if capture is not possible.
While this isn't a foolproof method of safety as the eye can be captured if the eye is the only liberty, it is certainly better than nothing.

How to Recognize a Simple Eye?

CMPUT 496



Simple eyes for Black:
1. top left corner
2. right edge of board
3. center

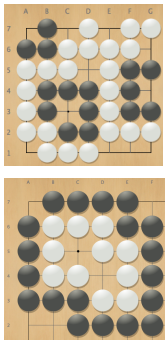
Definition of simple eye:

1. Single empty point p
2. All neighbor points $nb(p)$ occupied by stones of the *same color*
3. All these stones are *connected* in a single block

- Question:
  by the definition above, which points are simple eyes for White?
- There are other, more complex kinds of eyes (later)

Detecting Simple Eyes Locally

CMPUT 496

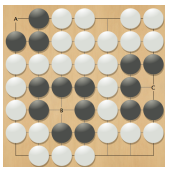


- Can detect most simple eyes locally
  - Only look at neighbors and diagonals
  - Corner, edge:
    need all diagonal points to connect (1 in corner, 2 on edge)
  - Center: need at least 3 of 4 diagonal points to connect
- Can connect along some longer path
  - Pretty rare, ignored in `Go1`
  - Example: A7 is an eye
    Stones A6 and B7 connected over a long path

This is a faster implementation by detecting the eye locally. However, this does not cover all cases. For example, the eye can be connected by a longer path.

One Eye is Not Enough

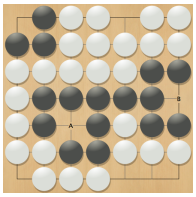
CMPUT 496



- One eye is not enough
- Move inside eye becomes legal if it is a capture
  - Examples: move takes the last liberty of the surrounding stones
  - So: one eye helps, but not enough for safety

Stones with Two Eyes are Safe

CMPUT 496



- Here, Black has one block surrounding two eyes A and B
- White cannot attack
  - Both A and B are suicide
- Black is safe as long as Black leaves the eyes alone
- Black should NEVER play A or B
  - Can always pass, if no good moves left

The simple eye method is a quick and simple implementation that stops players from filling its eyes and causing it to be captured. It still not a great algorithm but better than randomly placing pieces on the board.

The Go1 program implements this simple eye method.

From Go0 to Go1

CMPUT 496

- `Go1` algorithm avoids filling simple eyes
- Implementation in `board_util.py` function `generate_random_move`

```
moves = board.get_empty_points()
np.random.shuffle(moves)
for move in moves:
    legal = not board.is_eye(move, color) \
            and board.is_legal(move, color)
    if legal:
        return move
return PASS
```

Problem Solving and Decision Making

Heuristic: in CS are solutions to simplified problems.
Auxiliary Problem: Find an easier problem that will help solve the original problem.
Decomposing and Recombining: Breakdown a big problem into smaller parts, solve each parts, and combine them together.
Mathematical Optimization: Find the best possible solution to a given problem.
Bounded Rationality: is making the best decision based on the limited resources and information currently available.

Paradigms of decision making

From Herb Simon:

1. Find an optimal solution in a simplified world
2. Find a "good enough" solution for a more realistic world

Both are valid and neither are better than the other. Both are used in specific ways.

Main strands of research in Decision Theory

Normative decision theory

Analyze decision problems
Tell the user what is the best move/decision.

Descriptive Decision Theory

Analyze how agents make decision.

Utility

How good or bad a certain outcome is to the agent. Came from economics to measure satisfaction of a consumer with a good.
Utility is the amount that the buyer is willing to spend on the good.
For money however, it does not necessary scale linearly. It is heavily influenced from a person's behavior, is the person risk prone where utility scale faster than the rise in expected value. Or risk averse, utility does not scale well as expected value increases.

> As utility increases the rate of return falls, thus marginal utility decreases as utility increases.

Kahneman and Tversky - Anchoring

People tend to "anchor" on first impression. Later decisions made relative to this, not in absolute terms. People focus more on changes in their utility than on absolute utilities.

Decision Making as a Search Problem

Due to the potential factors involved in decision making which creates a huge search space, and time limitation. We first simplify a lot by using assumptions.

- The state of the world is completely known at each time
- There are terminal states where we can evaluate the result precisely
- The possible actions in each state are known
- An action changes the state to a new state in a known deterministic way
- No element of chance.

Games like Rubik's cube, solitaire, Sudoku, which are single-players fit this simple setting.
Two-player games like checkers, or chess also fit this setting.

Games like DnD, all dice games, and most card games do not fit into this simplified setting.
Online and computer games like physics simulations, and any real time elements also does not fit into our simplified setting.

> Generally, "Classical" games like go, and chess are used to study decision-makings in CS as it's
>
> - simple, controlled environment
> - still difficult to solve or play well
> - interesting for many People
> - games and results are easy to understand
> - Playing games well requires good decision-making skills
> - We can study the core problem of decision-making without being distracted by too many complications, and ad-hoc cases

Formal model for our game

Terminology

State, Game state: Complete description of the current situation. In games, board position or cards, etc (whose turn it is), Often includes (parts of) history, a sequence of moves from start of game to current position.
In our Go game we don't have history.
Move, actions: Leads from one state to another. Often created by the rule set and the state of the game.
History, game record, move sequence: contains the history of all previous moves. Sometimes requires, sometimes not. For example, TicTacToe does not need history as the placements are fixed, For Go, the Ko rule requires history to prevent the illegal move. You can get to the current state by only using the history.
State Space: A graph with all the possible state of a problem, Edges in graph show how states are connected by actions.
Terminal State: has no possible moves. No outgoing edges in the graph.
In Go games can end in two ways:

- A player resigns
- Both players pass in turn

Rewards: Later, in reinforcement learning, we will talk a lot about rewards (or costs: negative rewards). In many games, the only reward is at the end, in the terminal state.

Types of State Spaces

Assume root at the top is the current state:

1. Tree
2. DAG (Directed acyclic graph)
3. DCG (Directed cyclic graph)
4. Tree is easiest for search, DCG is the hardest

For example: Tic-Tac-Toe: DAG: different move order can lead to the same result.
Go without repetition or even simple repetition rules have DCG as you can have a game that keeps repeating itself to the same position.
Go with the full repetition rule are DAG.

Types of Game Representation

Move history - Move list
Different APIs like GTP
Many games are grid-based. A simple naive way is a 2-d array, a faster method is a row-major, or column-major row ordering 1-d array. Bitmaps can be very fast.
There are optimizations that can be done to improve the run-time performance of the agent. For example, only keeping the difference of the rewards instead of storing the rewards from all state, including no reward state.

Estimating Search Effort

Assume that we need to visit every state in order to solve a game, in general this isn't the case as there are many collisions, but for now lets consider the possibility.

How long will it take to traverse through all of them?

The main factors are:

- Speed of the program
- Size of the state space

For a 7 by 7 empty go board, assume that we can process 1000 states per second with the simplest tree model of 49 branches per level. What depth d can we reach in which time?

At 7 levels deep it takes about 21.5 years to compute.

To solve the game we need to see to the end, and we want to make a decision in less than 5 minutes.

- Increase the speed of the program
- Decrease the branching factor, b
- Decrease depth, d

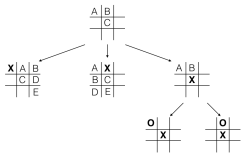
Decreasing the Branching Factor

The Branching Factor is the growth of the number of states per level or in this case per turn.

Example - Use Symmetry in TicTacToe

CMPUT 496

Sequential Decision-Making



- At root: Only 3 of 9 total moves are different
  - Corner (A), Edge (B), Center (C)
- All 6 other moves lead to a symmetric position, same result as A or B
- Symmetries at level 1:
  - After corner or edge: 5 distinct cases
  - After center: 2 distinct cases
- Limitation: most symmetries broken after few moves

Symmetry

CMPUT 496

Sequential Decision-Making

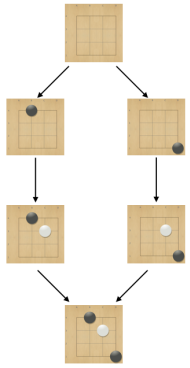
- Typical example to reduce state space by symmetry
- Good reduction at depth 1 or 2
- Then *symmetry breaks*
- Almost no reduction deeper in the tree
- Reduction of whole state space is limited to some constant factor
  - Less than 8 in Go
- Using a DAG can be much more effective

DAG (Directed Acyclic Graph)

DAG (Directed Acyclic Graph)

CMPUT 496

Sequential Decision-Making



- Idea: single node for all *equivalent* states
- Different paths to same node
- Can lead to huge reduction in state space
  - Why? Because the whole *subtree* below is no longer duplicated

Because there are many different paths that can lead to the same state a lot of the states in a simple tree are just duplicates that can be removed.

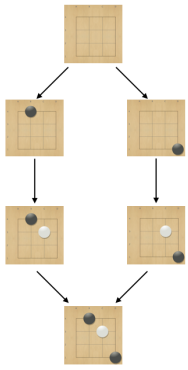
In a tree model each action leads to a new node that may or may not been duplicated.

In a DAG model however, there is only one node for equivalent states.
This can lead to a massive reduction in the size of the state space better than symmetry.
The advantages of the DAG model is that it avoids redundant computations of sub-trees or sub-DAGs, it also reused in many decisions - you compute once, and re-use them often.
The limitations of the DAG model is the memory requirement and the recognition of equivalent states. Moreover, there are some algorithms only designed for trees, not for DAGs. Propagating information up towards the root is also a problem as there are many parents to choose from. States with different history are another limitation as they cannot merge into one node as they are not equivalent. Example, simple ko - is capture allowed? The board looks the same but the moves are different.

Counting States in a DAG

CMPUT 496

Sequential Decision-Making



- Simplified Go example, ignore symmetry and captures
- Depth 0: 0 black, 0 white stones
- Depth 1: 1 black, 0 white stones
- Depth 2: 1 black, 1 white stones
- Depth 3: 2 black, 1 white stones
- Depth d: $\lceil d/2 \rceil$ black stones and $\lfloor d/2 \rfloor$ white stones
- How many ways to put that many stones on a board with 49 points?

Counting States in a DAG (continued)

CMPUT 496

Sequential Decision-Making

- Example:
- Board with 49 squares
- How many different ways to place 5 black stones?
- Answer: $\binom{49}{5} = 1906884$
- How many different ways to place 5 black stones and 3 white stones?
- Answer: $\binom{49}{5} \times \binom{44}{3} = 1906884 \times 13244 \approx 25.2$ billion
- Why?
  - $\binom{49}{5}$ ways to place 5 black stones
  - $49 - 5 = 44$ empty points remaining
  - $\binom{44}{3}$ ways to place the 3 white stones there
  - Each different choice for black or white leads to different position, so multiply

b^d Model and DAG

Instead of having a constant branching factor at every level there could a variable amount of branching depending on the state and the level. There is an effective branching factor b_d depending for each depth d.

$$b_d = \frac{\#\ nodes\ at\ depth\ d+1}{\#\ nodes\ at\ depth\ d} \tag{1}$$

Based on this equation a general form for the amount of nodes at depth d can be calculated:

$$Nodes = 1 + \Sigma_{i=0}^{d-1} \Pi_{j=0}^{i}\ b_j \tag{2}$$

b^d Model vs Reality: Some Case Studies (1)

CMPUT 496

Sequential Decision-Making

- How realistic is the b^d model for size of state space?
- We'll look at some popular games
  - Go, TicTacToe:
  - Roughly, $b_n \approx b_0 - n$
  - Why? One less empty square with each move
  - One less possibility for next move
- Not exact:
  - Ignores illegal move rules
  - Ignores games that end earlier
- Setting $b_n = b_0 - n$ gives $b_0!$ leaf nodes
  - Earlier TicTacToe example: $9 \times 8 \times ...1$

b^d Model vs New estimate: 7×7 Go

CMPUT 496

Sequential Decision-Making

- 7×7 Go estimate from Lecture 4:
- 25 moves on average during a game, game length about 30 moves
- Rough b^d estimate $25^{30} \approx 10^{42}$
- New model:
- $b_0 = 49$, and $b_n \approx b_0 - n$
- Stop game at $n = 30$
- $49 \times 48 \times ... \times (49 - 30) = 49!/18! \approx 10^{47}$
- Still ignores captures, ko, different game lengths,...

b^d Model vs Reality: Checkers

CMPUT 496

Sequential Decision-Making

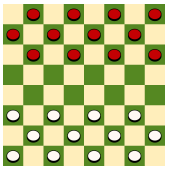


Image source:

https://en.wikipedia.org

- Checkers: complicated b
- Beginning: many pieces blocked
- Pieces unblocked:, b increases
- Forced captures: $b = 1$
- When pieces get captured, b decreases again
- When checkers become kings, b strongly increases
- Length of game d varies wildly
- Estimated average over typical game: $b \approx 2.8$

b^d Model vs Reality: Chess

CMPUT 496

Sequential Decision-Making

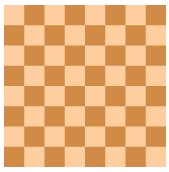


Image source:

https://en.wikipedia.org

- Chess: also complicated
- Pieces such as queens can have many moves, but may be blocked
- King in check: often only few legal moves
- When pieces get captured, b decreases
- Estimated average over typical game: $b \approx 35$
- Length of game d varies wildly

b^d Model vs Reality: Shogi

CMPUT 496

Sequential Decision-Making

Image source:

https://en.wikipedia.org

- Shogi, Japanese chess
- Similar to chess, plus:
- Captured opponent pieces can be *reused* for yourself in a future move
- With captures, b *increases*
- Estimated average over typical game: $b \approx 92$
- b can be several hundred in endgame with many captured pieces available for "dropping" back on board

Sequential Decision Making

The process of looking ahead into future states in order to make a good decision now. We need to consider sequences of actions, until we reach a terminal state.
There are many factors that are involved in good decisions, short and long term consequences, evaluating different options and choosing the best-looking one.

Making Sequential Decisions

CMPUT 496

Sequential Decision-Making

Very general model:

Loop:

- Get current state of world
- Analyze it
- Select an action
- Observe the world's response
- If not done: go back to start of loop

Practically important question:

- Can we do this in a *simulation model* as opposed to the real world?

A good example is Path Planning: First Decision: fly, drive, take the bus or walk? If drive or walk: each street corner is a decision point and there is a long sequence of decisions required to arrive to the destination. There are trade offs to consider: speed vs costs vs scenery vs etc.

Formal Framework

Formal Framework

CMPUT 496

Sequential Decision-Making

- Sequence of states and actions
- Start state s_0
- Action a_i leads to next state, s_{i+1}
- Keep going until reach a terminal state s_n
- Sequence $(s_0, a_0, s_1, a_1, ...s_n)$
- Sometimes we only write the actions $(a_0, a_1, ...a_n)$
  - Example: games where states are determined from game rules and actions

Formal Framework with Rewards

CMPUT 496

Sequential Decision-Making

- Formal framework can also include rewards (or costs)
- Simple case (most games we consider): single reward r at end
- General case: reward r_i after each action a_i
  - Write rewards as part of sequence:
  - $(s_0, a_0, r_1, s_1, a_1, r_2, ..., r_n, s_n)$

Partial Sequences

CMPUT 496

Sequential Decision-Making

- Full sequence goes all the way to terminal state
- *Partial sequence* can stop after any number of actions
- Two full sequences always share a *common prefix*
- In worst case, it might only contain the start state
  - Example - Go game
  - $(s_0$, Black B3, s_1, White A2, s_{2a}, Black D4$)$
  - $(s_0$, Black B3, s_1, White A4, s_{2b}, Black D4$)$
  - Common prefix $(s_0$, Black B3, $s_1)$

Re-Interpreting the Tree and DAG Models

CMPUT 496

Sequential Decision-Making

- Our model so far:
- State space as a graph
- Nodes are states, edges are actions
- Tree and DAG are special cases of graphs
- New view:
  we can view the same trees and DAGs
  as a way to organize all action sequences

Organizing Sequences in Trees

CMPUT 496

Sequential Decision-Making

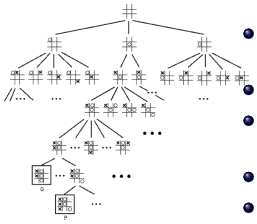


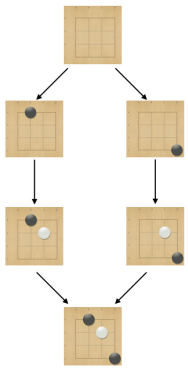
Image source: `http://web.emn.fr`

- Consider the (huge) set of all possible state-action sequences
- Organize them such that:
- Any two sequences share their *longest common prefix*
- Branch as soon as they differ
- Result: we get exactly the tree representation of the state space

Organizing Sequences in a DAG

CMPUT 496

Sequential Decision-Making



- Similarly, we can relate sequences to the DAG model
- Start with sequences-as-tree model
- Then, merge two different sequences when they both reach equivalent states
- Result: we get exactly the DAG representation

Profiling and Code Optimization

Profiling and Code Optimization

- Our `Go0` and `Go1` Python sample codes are very slow
- They were written for simplicity, not speed
- This is usually a good first approach - see quotes next slide
- Optimization is very important in search, but it can wait a bit
- We can optimize *if* and when we need it
- First, look where the time is spent
- Profiling is an easy way to check this

Amdahl's Law

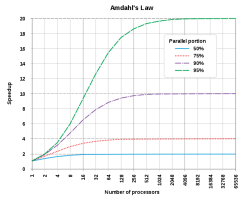


Image source:

`https://en.wikipedia.org/wiki/Amdahl's_law`

- Amdahl's Law (1967)
- How does speeding up one part of program speed up the whole?
- Often used for parallel programming
- Main idea: the parts of the program that are not optimized limit the overall speedup

Some Famous Quotes

Fred Brooks, The Mythical Man-Month (1975)

> *The management question, therefore, is not whether to build a pilot system and throw it away. You will do that. [...]*
> *Hence plan to throw one away; you will, anyhow.*

Don Knuth, Structured Programming with go to Statements (1974)

> *We should forget about small efficiencies, say about 97% of the time: premature optimization is the root of all evil.*

Amdahl's Law - Formula

- p = percentage of program that is speeded up
- s = speedup for that part
- Runtime before optimization: 1
- Runtime after optimization: $(1-p)+p/s$
- Speedup limit for the whole program:
  - limit = $\frac{1}{(1-p)+p/s}$
- Simplified version: assume s very large, ignore p/s
  - limit $\approx \frac{1}{1-p}$

Limits of Optimization

- There is often an (approximate) 80-20 rule: 80% of the improvement can come from 20% of the code
- With search, it can be even higher
- However, consider Amdahl's law
- Assume a program spends 80% of its time in one function
  - We manage to speed this function up 100x
- How much is the overall speedup?
  - Less than 5x

Amdahl's Law - Same Example Revisited

- 80% of program speeded up, so $p = 0.8$
- $s = 100$ speedup for the optimized function
- Speedup limit for the whole program:
  - limit = $\frac{1}{(1-p)+p/s} = \frac{1}{(1-0.8)+0.8/100} \approx 4.81$
- Simplified version:
  - limit $\approx \frac{1}{1-p} = 5$

Profiling

- Define a test that runs your program with a typical workload
- Run it with a special program called profiler
- Profiler tells you details of the program execution
- Profilers can be on the function level or instruction level
- How often was piece of code executed?
- How long did it take?
- Possibly, lower level details such as cache misses

Ways of Profiling in Python

- `cProfile` is a built-in module, no need to install anything
- Downside: overhead of profiling is also measured
- More advanced profilers are available for download:
  - `Profilehooks`
  - `pycallgraph`

See our Python language page:
`https://webdocs.cs.ualberta.ca/~mmueller/courses/496-current/python3-language.html#profiling`

Simple Profiling in Python with `cProfile` - Code

See code `profile_Go1.py`

```
import cProfile
from Go1 import Go1
...
def play_moves():
    """
    play 100 random games of 100 moves each
    for profiling.
    """
    ...
cProfile.run("play_moves()")
```

Speeding Up `Go1`

- `Go1` is slow
- For search and simulation, speed is very important
- How to improve the code?
- Both low-level optimizations and better algorithms help
- Case study: a series of improvements to `Go1`
  - Result: `Go2` - same algorithm as `Go1` but faster

Simple Profiling in Python

- See code `profile_Go1.py`
- Try it out with
- `./profile_Go1.py > profile.txt`
- `sort -k 2 -r profile.txt`
- This sorts by total time per function
- Try other options for `-k` to sort by other criteria
- Example: `sort -k 1 -r profile.txt`

Ideal Optimization Procedure

- First, pick a test to measure the speed
- Here: play 100 games on 7×7 board
- Repeat:
  - Run test games with profiler
  - Identify the most expensive functions
  - Try to improve them by optimization or better algorithms

Profiling `Go1`

- Profile with cprofile
- Total time: 6.2 seconds
- Worst 5 individual functions listed below (all in `simple_board.py`)

Calls	Time	Name
561025	1.960	neighbors_of_color
2287541	0.680	get_color
610480	0.679	_neighbors
43441	0.662	_block_of
18268	0.405	play_move

Read the Code

- Start by reading the expensive code carefully
- Can we avoid unneeded computation?
- Here: read `_has_liberty, neighbors_of_color`

```
def _has_liberty(self, block):
    for stone in where1d(block):
        empty_nbs = self.neighbors_of_color(
                    stone, EMPTY)
        if empty_nbs:
            return True
    return False
```

Profiling `Go1`

- Also look at cumulative time
- Function itself plus other functions it calls
- Sort by column 4:
  `sort -k 4 -r profile.txt`
- Some interesting functions listed below (all in `simple_board.py`)

Calls	Cumulative Time	Name
10974	4.429	is_legal
25584	3.566	_detect_and_process_capture
43441	3.368	_block_of
561025	3.351	neighbors_of_color
43441	1.359	_has_liberty

Read the Code


```
def neighbors_of_color(self, point, color):
    nbc = []
    for nb in self._neighbors(point):
        if self.get_color(nb) == color:
            nbc.append(nb)
    return nbc
```

- We do not need to compute the whole list
- Stop if we find one liberty
- `neighbors_of_color` is still used in other places
- Add a function that is optimized for our task

Strategies for Optimization

- Best: avoid calling a function
- Second best: speed up a function, avoid unneeded computation
- Here: detecting captures is most expensive

New Version


```
def find_neighbor_of_color(self, point, color):
    for nb in self._neighbors(point):
        if self.get_color(nb) == color:
            return nb
    return None

def _has_liberty(self, block):
    for stone in where1d(block):
        if self.find_neighbor_of_color(stone, E
            return True
    return False
```

Profiling Again

- Total time reduced from 6.2 to 6 seconds
- Reduction in `_has_liberty` by calling cheaper `find_neighbor_of_color` instead of `neighbors_of_color`
- Nice improvement for a little work, but not a huge win
- Can we avoid the many floodfills altogether?
- We do the floodfill for each neighbor of a stone
- We only need to know "does block have at least one liberty"?
- Can we check that more effectively?

CMPUT 496
Go Rules Revisited
Profiling and Code Optimization

Profiling Again

Calls	Time	Name
66323	0.669	find_neighbor_of_color
18645	0.396	play_move
32369	0.367	_is_surrounded
264389	0.321	_neighbors
147455	0.294	neighbors_of_color
828018	0.257	get_color

CMPUT 496
Go Rules Revisited
Profiling and Code Optimization

Optimizing Floodfill

- We can store such a liberty for each stone `s`
- In the code: `liberty_of[s]`
- Check capture: just check if board at location `liberty_of[s]` is still empty
- If yes, no floodfill is needed (why?)
- If no, we just played there
  - Do floodfill to try to find a *different* liberty for `s`
  - If success: update `liberty_of[s]`
  - If fail: yes it is a capture

CMPUT 496
Go Rules Revisited
Profiling and Code Optimization

Optimizing Neighbors, First Try

```
def _neighbors(self, point):
    return [point-1, point+1,
            point-self.NS, point+self.NS]
```

- Called often: compute list of neighbors of a point
- Each call creates a new list
- Some neighbors are off the board (state `BORDER`), causing more tests in code
- Precompute a `neighbors` array for each point
- Include only on-board neighbors
- Result: EPIC FAIL, runtime over 11 seconds
- Why? board is copied and neighbors array recomputed over 11000 times

CMPUT 496
Go Rules Revisited
Profiling and Code Optimization

Result, and More Floodfill Optimization

- Total time reduced from 6 to 4.4 seconds
- Success!
- Next: try to reduce calls to expensive floodfill functions
- Idea: instead of always computing a block:
- First check the 4 neighbors of the stone if there is a liberty there
- Result: Total time reduced from 4.4 to 3.7 seconds
- Cost: more complex code, adds special case

CMPUT 496
Go Rules Revisited
Profiling and Code Optimization

Optimizing is_legal

```
def is_legal(self, point, color):
    board_copy = self.copy()
    legal = board_copy.play_move(point, color)
    return legal
```

- This function is the reason for FAIL with previous optimization
- Slow: copy the board, then try to play the candidate move to see if it is legal
- Solution: Implement is_legal without play_move
- Success! Total time reduced from 4.4 to 2.5 seconds
- Cost: increased code complexity, some redundancy in is_legal and play_move

Details

Calls	Time	Name
51038	0.528	find_neighbor_of_color
75984	0.288	neighbors_of_color
21163	0.227	_is_surrounded
166427	0.207	_neighbors
495786	0.181	get_color
7418	0.145	play_move

- play_move calls: less than half as many
- Many other function calls also significantly reduced

Optimizing Neighbors, Second Try

- Now we are no longer copying the board at each legal move check
- Now the neighbors optimization works beautifully
- Result: Total time reduced from 2.5 to 2 seconds
- Success!
- There are more opportunities to optimize but I stopped here

Summary

- Discussed profiling and optimization
- Some concrete case studies
- Overall about 3x faster now, from 6 to 2 seconds on test
- Save computation, precompute, compute data incrementally when there are only small changes, catch and handle frequent simple cases early
- Very few optimizations are win-win. The speed often comes at the cost of code complexity
- Remember Knuth:
premature optimization is the root of all evil

Part II

Search And Knowledge

You can use search to lookup information, but search can also be used to discover new information. Information may change during search like the internet. Data may be stored in a structured (like a tree) or unstructured (like a hashed) way.
There is a whole bunch of search algorithms some better than others.

Often state spaces are generated during run-time based on the rules of the game.

Blind Search

No extra information that helps us search. We assume only that we recognize the goal and terminal states when we visit them. E.X. DFS, BFS, etc.

The data structure of the graph didn't matter in this search, we also assume that each node is visited once. Basically we assume that we can visit and check an unique node in constant time.

Random Search: randomly go down each path of the state space to find the treasure. This is an incomplete search method as not every node will be checked. Moreover, some nodes will be checked more than once.

In general it takes about

$$\frac{1}{n}(1+2+\ldots+n) \approx \frac{n}{2}$$

As there is an equal probability that the goal is in one of the states, it takes take anywhere between 1 to n steps to get their so long as you visit each state once.

This idea works regardless of the shape of the state space, the details of the search algorithm does not matter either. In practice some states are more costly to visit then others, i.e., states stored on hard disk vs SSD vs memory vs cache. This model doesn't account in the variances in visiting those states. We can visit the cheapest one first but then we are using some knowledge which doesn't count as a blind search.

Moreover, if we have information about where the goal is we can use that information in our heuristic, therefore it is not longer a blind search.

Random sampling

Is the idea that we go down a random path at each step we check if the treasure is found, if we did find the treasure then we stop or stop at a leaf node. In the latter case we try another random walk.

This search is probabilistic, it is not guaranteed to find the goal. May travel the same path again. There is a bias towards nodes that are closer to the root as they are more likely to be visited more often compared to the extremities.

DFS and BFS

Are better as they are guaranteed to find the treasure in the state space if it exist.

Optional Activity: Expected Number of Steps for Random Sampling

CMPUT 496

- We saw that for dfs, bfs, the expectation is about $n/2$ steps
- How about random sampling?
- More steps expected because of re-visiting the same nodes
- How much more? depends on b and d
- Hint: analyze level by level as when we estimated the DAG state space
- You can start with the case $d = 1$. One root, b children

Discussion and Optional Activity: Random Sampling Without Repetition?

CMPUT 496

- Could we eliminate the duplication in random sampling?
- Yes, with extra book-keeping
- Keep track of which subtrees still have unexplored nodes
- Only sample among children that are roots of such subtrees
- When a subtree is completely explored
  - Check if parent is also fully explored
  - If yes, recursively check grandparent,...
- Stop algorithm when treasure found, or root is marked as fully explored
- Optional activity: implement this version of sampling
- Discuss: why is this not done in practice?

Summary of Blind Search

CMPUT 496

- Blind search uses no heuristics
- Single goal in random location:
  on average, explores half the state space
- Examples: standard graph search algorithms dfs, bfs
- State space too large: cannot complete search
- Random sampling also works
  - Not quite competitive on these examples
  - Re-visits nodes more than once

Example - Treasure Hunt with Heuristic

CMPUT 496

- Heuristic can help direct the search
- Which action is most likely to lead to treasure?
- If heuristic is good:
  enormous reduction in search effort possible
- Extreme case - perfect heuristic:
  - Always picks a correct action
  - Goes directly to goal
- Python demo `heuristic_search_on_tree.py`
- With probability p: follow path to treasure
- Demo: vary p and see what happens!

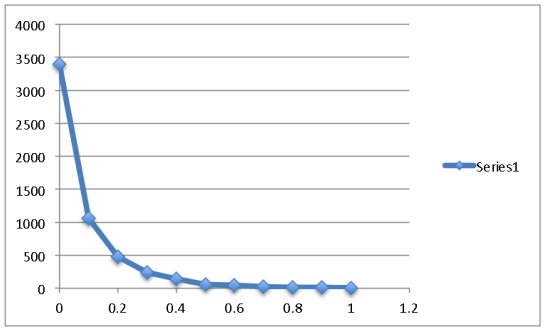
Blind Search and Two Player Games

CMPUT 496

- Most of what I said is relevant for the two player case
- More complex because of opponent actions
- There is not a single “goal state”
- Much more on the two player case in future lectures
- Next: how does having a heuristic change single agent search?

Heuristic vs Simulations

CMPUT 496



- X-axis: p, Y-axis: average nodes searched

From Blind Search to Heuristic Search

CMPUT 496

- We can solve small problems by brute force, systematic search
- We can sometimes solve large problems by luck (e.g. sampling)
- We can sometimes solve large problems by knowledge, without any search
- What can we do in all other cases?
- Answer: use heuristic to *guide a search process*
- Combine power of search with power of knowledge

Examples: Applications of Heuristic Search

CMPUT 496

- Solve the game of checkers
- Play chess, backgammon, Go better than any human
- Find a good path for a character in a video game
- Solve a puzzle such as Rubik’s cube
- Plan a tour for the Traveling Salesman Problem
- Control an unmanned vehicle in traffic
- Plan the motion of a robot arm
- Land a spacecraft on a comet
- Control mobile robots in an automated warehouse
- Optimize the location of warehouses for delivery

What are Heuristics Good For?

CMPUT 496

- They help guide the search through a (large) state space
- Following heuristic is often (much) better than blind search
- Besides state evaluation, we can use other forms of heuristics
  - Example: heuristic for move ordering in games
  - Evaluate moves, not states

When do we Need Heuristic Search?

CMPUT 496

Typical scenario for modern heuristic search:

- Huge state space
- Heuristics are good, but far from perfect
- Goals:
  - Use heuristic when it works
  - Have a robust method, works even when heuristic fails
- Important tool for search: exploration

Why Exploration?

CMPUT 496

- Classical search methods:
  - search algorithm
  - heuristic evaluation
  - search is **greedy**, always trusts the heuristic
  - can get into deep trouble
- Exploration can help
  - Look at other parts of the state space, not only where the heuristic leads us
  - Examples: simulation, random walks, many other methods
  - Much more later in this course

Many modern heuristic search methods combine all three elements

- Search - systematically look ahead into the future, we can exploit the heuristic but explore other states as well
- Heuristic Evaluation - how good is a state or an action?
- Simulation - look ahead into the future by sampling sequences of future states

Heuristic Function

CMPUT 496

- Heuristic function $h(s)$ defined for each state s
- $h(s)$ estimates the "goodness" of s
- Two *very different* meanings for:
  - Single agent search
  - Two player search
- Heuristic is usually not perfect, but "useful"
- Many ways to define a heuristic function (many examples later)

Heuristic Evaluation Function - Single Agent Search

CMPUT 496

Single agent search meaning of heuristic $h(s)$:

- $h(s)$ estimates the **distance** from s to a goal
- Examples of goals:
  - Reach the treasure
  - Reach the destination in path planning
  - Solve Rubik's Cube
- Goals correspond to terminal states in the state space
- Requirements:
  - $h(s) = 0$ if s is a goal
  - $h(s) > 0$ if s is not a goal
- Intuition: state with smaller h is likely better

Heuristic Evaluation Function - Two Player Games

CMPUT 496

Two player zero sum games:

- What does a heuristic position evaluation mean here?
- Three different popular measures:

1. How likely is a player to win from here?
2. What is player's expected score, if win = 1 and loss = -1?
3. In point-scoring games:
   - What score will player get at the end?
   - Example: $h(s)$ = +14.5 points

1. Heuristic as Winning Probability

CMPUT 496

- $h(s)$ = probability of winning from state s
- For one specific player
  - Example: for Black
  - Example: for the *current* player (toPlay)
- Requirements:
  - $h(s) = 0$ if s is a sure loss for the player
  - $h(s) = 1$ if s is a sure win for the player
  - $0 < h(s) < 1$ for all other s
- Winning probability for *opponent*: $1 - h(s)$
  - This assumes one player has to win (no draws)

2. Heuristic as Payoff in Win/Loss Games

CMPUT 496

- Requirements:
  - $h(s) = -1$ if s is a sure loss for the player
  - $h(s) = 1$ if s is a sure win for the player
  - $-1 < h(s) < 1$ for all other s
- Often, a draw has value $h(s) = 0$
- Popular translation between winning probability and payoff:
  - Winning probability p
  - Payoff v
  - Translate by $v = 2p - 1$
  - Linear mapping
  - Matches requirements for sure wins and losses
- Payoff for *opponent*: $-h(s)$

3. Heuristic in Point-Scoring Games

CMPUT 496

- Requirement:
  - $h(s) =$ true score if s is a terminal position
  - Payoff for *opponent*: $-h(s)$ (zero sum)
- In games, no real requirements for non-terminal states
  - Just try to predict final score as well as possible
- Compare with single-agent case (see resources) - many special types of heuristic, e.g. admissible, consistent,...

Using a Heuristic vs Constructing a Heuristic

CMPUT 496

- *Search* part of course (now):
  - How to *use* a heuristic in search?
- *Knowledge* and *machine learning* parts (later):
  - How to *construct* or *learn* a good heuristic?

Examples - Sources of Heuristics

CMPUT 496

- Human knowledge from books
- Mathematical analysis of the problem
- Domain knowledge
  - Example: Euclidean or Manhattan distance for path-finding on a map (ignores obstacles)
- Rule-based systems
- Abstractions, such as *pattern databases*
- Solution of *relaxed*, simplified problem
- Trial and error

Learning a Heuristic

CMPUT 496

Learn a heuristic evaluation function:

- From examples in master games (supervised learning)
- From self-play (unsupervised)
- Construct function from *simple features* of a state
  - Learn weights of those features
- Design a deep neural network for evaluation
  - Learn weights of the network

Example of Misleading Heuristic

CMPUT 496

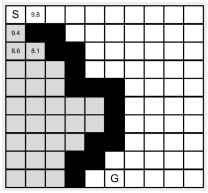


Image source: Fan Xie

A single agent problem:

- Find path from S to G
- Dark squares are obstacles
- Heuristic: Euclidean distance to G
- Misleading heuristic at start prefers to go down, not right
- Imagine making the grid finer and finer...
- ... more and more search effort wasted on the left side

Follow the Heuristic, or Explore?

CMPUT 496

- Blind search is hopeless in large state spaces
- Focusing only on the heuristic is too risky
- Need to find a balance
  - Follow heuristic when it works well
  - Do not trust heuristic blindly
  - Use exploration when stuck, or heuristic is misleading
  - Always use *some* exploration to guard against biases in heuristic
  - How much exploration is best?
- Big questions: when, where, and how to explore?
- Today, we have many case studies, but no full theory

Blind Search as Heuristic Search

CMPUT 496

- Remember blind search assumptions:
  - "Know the rules" - can generate all successors of a state
  - Recognize a goal state once we reach it
  - No other knowledge
- Equivalent in single agent heuristic search: "trivial heuristic"
  - $h(s) = 0$ if s is a goal state
  - $h(s) = 1$ if s is not a goal state
  - Search with this heuristic is like blind search, no extra information

Summary

CMPUT 496

- Discussed heuristic vs blind search
- Exploit heuristic knowledge
- Explore to avoid over-reliance on less than perfect heuristics
- Big question: how to find a balance?
- Next topics: solving two player games, minimax and alphabeta

Solving Games

CMPUT 496

- What does solving a game mean?
- Find the correct outcome of the game
  - With best play...
  - ...by **both** players.
- *How* to play if we have a win?
- Need a *winning strategy*:

How a Winning Strategy Looks Like

CMPUT 496

- In a game, if it is our turn, we have a choice
- Play move1 or move2 or...
  - It is enough to know **one** winning move
- If it is the opponent's move, we need to win against *all* their moves
- Win against move1 and move2 and ...
  - We need to include *all* their moves in our strategy

Winning Strategy as a Tree (or DAG)

CMPUT 496

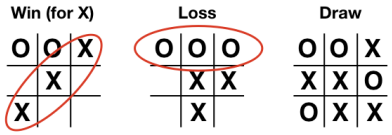
- Consequence: the winning strategy is a tree (or DAG)
- The winning strategy includes:
- One move when it's our turn
- All moves when it's the opponent's turn
- The tree (or DAG) of a winning strategy is much smaller than the whole state space
- It can still be very large
- It branches at every *second* level

Wins, Losses and Draws

CMPUT 496

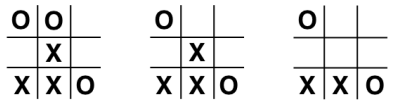
Terminal states

Win (for X) | Loss | Draw



Using search to find Win or Loss

X wins in one move | O loses in two moves | X wins in three moves



Proving a Win

CMPUT 496

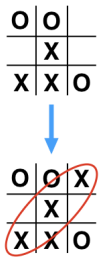
- To prove a win we need to find a winning strategy
- Usually, we do not store it
  - We just use search to prove a win (see later)
- Conceptually, we can build a strategy bottom-up
- First question:
  - What are winning terminal positions?
  - The rules of the game give the answer

Winning Strategy - Depth 1

CMPUT 496

Winning strategy

X wins in one move



- X can win in one move
- Winning strategy just contains that move

Evaluation of Terminal Positions

CMPUT 496

Game over, what's the result?
Different Types of evaluation:

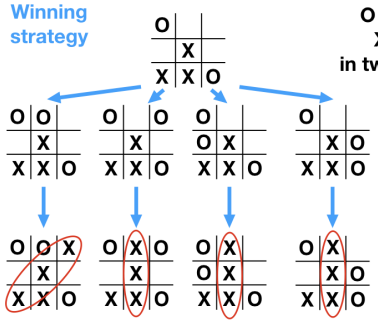
- Simplest case: binary (or boolean) evaluation, win-loss
  - Examples: Coin toss, Go with non-integer komi
- Popular case: win-draw-loss
  - Examples: TicTacToe, chess, checkers
  - Go with integer komi
- More general case: games with score
  - Examples: win by 5 points
  - Win $10.000.000

Winning Strategy - Depth 2

CMPUT 496

Winning strategy

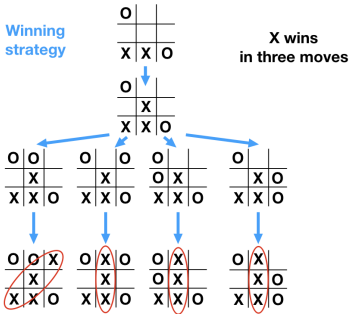
O to play X wins in two moves



- Winning strategy: d=1: **all opponent moves**, d=2: one reply for each $\rightarrow$ win

Winning Strategy - Depth 3

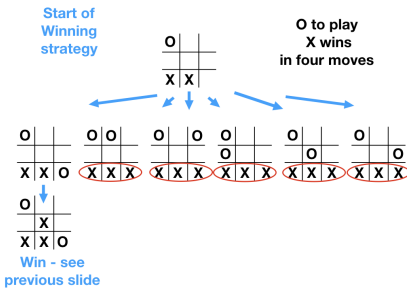
CMPUT 496



- d=1: One move, d=2: one branch for each opponent reply,
  d=3: one move in each branch → win

Winning Strategy - Depth 4

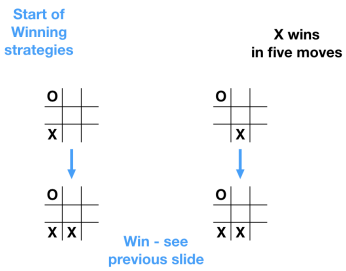
CMPUT 496



- d=1: all opponent moves, d=2: one move, leads to a known winning position

Winning Strategies - Depth 5

CMPUT 496



- d=1: One move in each case,
  both lead to a known winning position

Tic Tac Toe Sample Code

CMPUT 496

- `tic_tac_toe.py` has board representation, rules
- Similar to `Go1`, board stored in 1-d array of size 9
- Status codes
  `EMPTY = 0, BLACK = 1` for 'X', `WHITE = 2` for 'O'
- Useful functions `legalMoves, endOfGame, play, undoMove,...`

```
Board indexing:
0 1 2
3 4 5
6 7 8
```

AND/OR Tree

CMPUT 496

- In a game tree:
- Position where it is our turn:
  OR node
- Position where it is the opponent's turn:
  AND node
- Alternating play
  - Each move from an OR node leads to an AND node
  - Each move from an AND node leads to an OR node

Leaf Nodes

CMPUT 496

- *Leaf nodes* are *terminal states* of the game
- Game is over
- Can determine the result from the rules
- Examples:
  - Count the score in Go → winner
  - TicTacToe 3-in-a-row → win
  - TicTacToe board full, no 3-in-a-row → draw

Winning in an OR Node

CMPUT 496

- Our turn
- Finding one winning move is enough
- OR node n
- Children $c_1, ..., c_k$
- win(n) = win(c_1) **or** win(c_2) **or** ... **or** win(c_k)
- Shortcut evaluation:
  can stop at the *first* child that is a win
- We can play that move to win from n

Minimax Algorithm - Boolean Version - OR Node

CMPUT 496

- Each player tries to win.
  Zero-sum - opponent's win is my loss
- OR node: If I have *at least one* winning move,
  I can win (by playing that move)
- If all my moves lose, I lose.

```
// Basic Minimax with boolean outcomes
bool MinimaxBooleanOR(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    foreach successor s of state
        if (MinimaxBooleanAND(s))
            return true
    return false
```

Winning in an AND Node

CMPUT 496

- Opponent's turn
- We win only if we win after *all* opponent moves
- AND node n
- Children $c_1, ..., c_k$
- win(n) = win(c_1) **and** win(c_2) **and** ... **and** win(c_k)
- Shortcut evaluation:
  can stop at the *first* child that is a loss
- The opponent can play that move to make us lose from n

Minimax Algorithm - Boolean Version - AND Node

CMPUT 496

- AND node: All opponent moves need to win for me
- If any of their moves lose me the game, I lose.

```
// Basic Minimax with boolean outcomes
bool MinimaxBooleanAND(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    foreach successor s of state
        if (NOT MinimaxBooleanOR(s))
            return false
    return true
```

What if the State Space is a DAG?

CMPUT 496

- Exactly the same concepts work in DAG
- Difference in practice:
- We can store and share wins and losses computed earlier
- Different paths to reach the same node
- Only prove a win (or loss) for a node once,
  then remember
- Main technique to store states and results:
  hash table
  also called *transpositon table*
- Details later

Minimax Algorithm - Boolean Version (2)

CMPUT 496

- Less abstract pseudocode showing execute, undo move
- Python3 code `boolean_minimax.py`

```
// Minimax, boolean outcomes, execute/undo
bool MinimaxBooleanOR(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    foreach legal move m from state
        state.Execute(m)
        bool isWin = MinimaxBooleanAND(state)
        state.Undo()
        if (isWin)
            return true
    return false
```

Boolean Minimax Algorithm - AND Node

CMPUT 496

- Less abstract version showing execute, undo move

```
// Minimax, boolean outcomes, execute/undo
bool MinimaxBooleanAND(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    foreach legal move m from state
        state.Execute(m)
        bool isWin = MinimaxBooleanOR(state)
        state.Undo()
        if (NOT isWin)
            return false
    return true
```

Python Implementation and Solve TicTacToe

CMPUT 496

- Boolean negamax solver `boolean_negamax.py.py`
- Use to solve TicTacToe: `boolean_negamax_test_tictactoe.py`
- Main question: how to handle draws?
- Boolean solver only deals with two outcomes
- We can choose whether draws should count for Black or White
  - In TicTacToe code: function `setDrawWinner`
- More on this topic next class

Negamax Algorithm - Main Idea

CMPUT 496

- All evaluation in `StaticallyEvaluate()`, `MinimaxBooleanOR(s)` and `MinimaxBooleanAND(s)` is from a *fixed* player's point of view
- We can also evaluate from the point of view of the *current* player
- $\Rightarrow$ Negamax formulation of minimax search
- Current player changes with each move - negate result of recursive call
- My win is your loss, my loss is your win

Boolean Minimax - Discussion

CMPUT 496

- Basic recursive algorithm
- Runtime depends on:
  - depth of search
  - width (branching factor)
  - **move ordering** - stops when first winning move found
- Easy modification to compute *all* winning moves
  - Add a top-level loop which does not stop at the first win
- Questions: best-case, worst-case performance?

Negamax Algorithm - Boolean Version

CMPUT 496

```
// Negamax, boolean outcomes
bool NegamaxBoolean(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
        // CHANGE: evaluate from toPlay's
        // point of view
    foreach legal move m from state
        state.Execute(m)
        bool isWin = NOT NegamaxBoolean(state)
        state.Undo()
        if (isWin)
            return true
    return false
```

Boolean Minimax - Discussion (2)

CMPUT 496

- Boolean case is simpler special case of minimax search
- Efficient pruning - stops as soon as win is found
- Important tool used in more advanced algorithms later
- What is the runtime? Depends on *move ordering*
- Simple model: uniform tree, *depth d*, *branching factor b*
- What is best case, worst case?

Boolean Minimax - Efficiency

CMPUT 496

- Best case: about $b^{d/2}$, first move causes cutoff at each level
  - Exact calculation for best case - a little later
- Worst case: about b^d, no move causes cutoff
- Just count number of all nodes in the tree, as before:
- $1 + b + b^2 + ... + b^d = (b^{d+1} - 1)/(b - 1)$

Comments on Proof Tree

CMPUT 496

- Exactly the same definitions work on DAG, even on arbitrary graph
- Another name for proof tree: solution tree
- Efficiency: want to find a *minimal* or at least a small proof tree

Proof Tree

CMPUT 496

- A winning strategy for a player
- Dual concept: disproof tree - proves that we cannot win
- A subset of a game tree
- Gives us a winning move in each position we may encounter (as long as we follow the strategy...)
- Covers all possible opponent replies at each point when it's their turn

Size of Proof Tree

CMPUT 496

- Scenario: uniform (b, d) tree, OR node at root, we win
- How many nodes at each level?
- Level 0: 1 node (root)
- Level 1: $\geq$ 1 nodes (at least one child...), best case 1
- Level 2: $\geq b$ nodes (all children of level 1 nodes), best case b
- General pattern for best case: $1, 1, b, b, b^2, b^2, b^3, b^3, ...$
- Activities: Find formulas for size of proof trees in the best case

Definition of Proof Tree

CMPUT 496

- Subtree P of game tree G is a proof tree iff:
- P contains the root of G
- All leaf nodes of P are wins
- If interior AND node is in P, then *all its children* are in P
- If interior OR node is on P, then *at least one child* is in P

Best Case For Boolean Minimax Search

CMPUT 496

- Search is most efficient if it looks only at the proof tree
- This means, at OR nodes we only look at a winning move
  - We never look at a non-winning move first
- In practice, that's usually impossible - too hard.
- Good *move ordering* is crucial for efficient search
  - Compare with heuristic in treasure hunt example, Lecture 7
- We can use good *move ordering heuristics*, or techniques based on successively deeper searches
- More later

Summary of Solving Games

CMPUT 496

- Concepts: winning strategy, AND/OR trees
- Solving OR nodes, AND nodes
- Boolean Minimax and Negamax
- Efficient pruning of tree - stop at first winning move
- Good move ordering finds that first move faster
- Next: Assignment 2 preview

Example: Boolean OR and Maximum of 0, 1

CMPUT 496

- Example shows equivalence between
  - Logical OR
  - Taking the maximum of numbers in the set { 0, 1 }
- Booleans
  - True = we win
  - False = we lose
  - win(n) = win(c_1) **or** win(c_2) **or** ... **or** win(c_k)
  - win(n) if win(c_i) = True for at least one i
- Numbers in the set { 0, 1 }
  - 1 = we win
  - 0 = we lose
  - score(n) = max(score(c_1), score(c_2), ... score(c_k))
  - score(n) = 1 if score(c_i) = 1 for at least one i

Minimax Search: From Two to Many Different Outcomes

CMPUT 496

- Last time: boolean negamax solver for games with win-loss outcomes
- What about win-loss-draw?
- What about general numeric scores?
- Similar principles
  - A little bit more involved
  - Remember our setting: two player zero sum games, no chance element, perfect information
- Minimax search:
  - We maximize score
  - Opponent minimizes our score
- Zero-sum: each point we win, the opponent loses

MAX Node with Numeric Scores

CMPUT 496

- Example: MAX node n
- Five children with scores 2, 5, -3, 6, 10
- score(n) = max(2, 5, -3, 6, **10**) = 10
- Do we always have to evaluate all children now?
- With scores, usually yes
- We can stop early in two scenarios
  - We know the highest possible score, and one child achieves it (similar to boolean case)
  - We have a *bound*, and only want to know if we can reach at least that bound. Can stop as soon as one child achieves bound

OR Node = MAX Node

CMPUT 496

- Our turn, we maximize
- Example, win-draw-loss game:
  - Set win-score $>$ draw-score $>$ loss-score
  - For example, can use win = +1, draw = 0, loss = -1
- OR node n, children $c_1, ..., c_k$
- score(n) = max(score(c_1), score(c_2), ... score(c_k))

Examples - Stopping Early in MAX Nodes

CMPUT 496

- Scenario 1: maximum possible score is say 1000
- score(c_1) = 527
  - Keep searching...
- score(c_2) = 1000
  - Reached maximum
  - No need to look at c_3, c_4...
- Scenario 2: we want to exceed a bound, say 500
- score(c_1) = 527
  - First child is good enough, stop.

AND Node = MIN Node

CMPUT 496

- Opponent minimizes among all their moves
- AND node n, children $c_1, ..., c_k$:
- score(n) = min(score(c_1), score(c_2), ... score(c_k))
- Compare win/loss case: n is win iff all children are wins

Boolean AND vs Computing Minimum

CMPUT 496

- Boolean AND is equivalent to taking MIN over { 0, 1 } scores
- win(n) = win(c_1) **and** win(c_2) **and** ... **and** win(c_k)
- win(n) if win(c_i) = True for all i
- score(n) = min(score(c_1), score(c_2), ... score(c_k))
- score(n) = 1 if score(c_i) = 1 for all i

Naive Minimax Search, General Case

CMPUT 496

- Similar to boolean case
- Compute max over all children in OR node
- Compute min over all children in AND node
- Two different functions `MinimaxOR`, `MinimaxAND`
- They call each other recursively
- Stop in terminal state, evaluate statically

Naive Minimax Search - OR node

CMPUT 496

Changes to boolean minimax in **bold**

```
int MinimaxOR(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    int best = -INFINITY
    foreach legal move m from state
        state.Execute(m)
        int value = MinimaxAND(state)
        if (value > best)
            best = value
        state.Undo()
    return best
```

Naive Minimax Search - AND node

CMPUT 496

```
int MinimaxAND(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    int best = +INFINITY
    foreach legal move m from state
        state.Execute(m)
        int value = MinimaxOR(state)
        if (value < best)
            best = value
        state.Undo()
    return best
```

Negamax Search for Numbers

CMPUT 496

- Similar to boolean case
- Evaluation from current player's point of view
- Single `Negamax` function, calls itself recursively
- Negate result of children to change to current player's view
  - Result of children always from other player's view
- Compute the max of the negated results

Naive Negamax Search - No Pruning

CMPUT 496

```
int Negamax(GameState state)
    if (state.IsTerminal())
        return state.StaticallyEvaluateForToPlay()
    int best = -INFINITY
    foreach legal move m from state
        state.Execute(m)
        int value = -Negamax(state)
        if (value > best)
            best = value
        state.Undo()
    return best
```

Pruning Idea From Earlier Scenario 1

CMPUT 496

- If maximum possible value is reached:
- Return directly, prune remaining moves
- Easy to implement
- Powerful with only two values { 0, 1 }
- May not help much if we have many scores
- It is rare to win by the maximum score in real games

```
int Negamax(GameState state)
    ...
        int value = -Negamax(state)
        if (value > best)
            best = value
            if best == MAXVALUE:
                return best
    ...
```

Python Codes

CMPUT 496

- `minimax_sample_tree.py`, `minimax_sample_tree_data.py`
  artificial game tree to illustrate minimax and alphabeta
- `naive_minimax.py`, `naive_negamax.py`, `naive_minimax_negamax_test.py`
  Minimax and Negamax without any pruning, tests on sample tree

Pruning Idea From Earlier Scenario 2

CMPUT 496

- Idea was: prune when reaching “good enough” value
- Reduces search to the boolean case
- What does “good enough” mean?
- Answer: better than a bound
- We look at two cases
- First: bound is already given to us
- Second: compute, update bounds during the search
  - One bound for each player (alpha and beta)

Inefficiency of Plain Minimax/Negamax

CMPUT 496

- Inefficient. No pruning
  - In (b, d) tree, searches all b^d paths
  - Compare to efficient pruning in boolean case
- What’s wrong? How can we prune moves?
- Revisit our two pruning scenarios above
  - One idea will be of limited use in practice
  - Other idea is very powerful, leads to alphabeta algorithm

Reduce Minimax Search to the Boolean Case

CMPUT 496

- Assume we already have a candidate minimax value m (discuss: where might m come from?)
- We can do **two** boolean searches to verify if m is the minimax result
- Remember: each terminal state will be evaluated with its score (a number)
- We replace those scores with a boolean win/loss result
  - win: score above a threshold m
  - loss: score below a threshold m
  - What about score = m?
    - It depends, see next slides

Reduce Minimax Search to Two Boolean Searches

CMPUT 496

- Assume we already have a candidate minimax value m
  - First search if we can get *at least* m:
    scores $v \geq m$ are wins,
    scores $v < m$ are losses
  - Second search: can we get *more than* m:
    scores $v > m$ win,
    $v \leq m$ lose
- Assume:
  - Search 1 returns a win
  - Search 2 returns a loss
- Then m must be the minimax value

Example - Solve TicTacToe

CMPUT 496

- Example: solve TicTacToe
- Set win-score = 1, draw-score = 0, loss-score = -1
- Set m = draw-score = 0
- First boolean search: test ($v \geq m$), can X draw-or-win?
  - Search result: **yes**
- Second boolean search: test ($v > m$), can X win?
  - Search result: **no**
- Together, both searches prove:
  - TicTacToe is a draw...
- See Python code
  `boolean_negamax_test_tictactoe.py`

Understanding the Boolean Search Result(s)

CMPUT 496

- Candidate minimax value m
- Three possible results:
  greater than, smaller than, equal to m
- What if Search 1 returns a loss?
  - Stop, no need for Search 2,
    minimax value smaller than m
- What if Search 1 returns a win?
  - Do Search 2
  - What if Search 2 also returns a win?
    - Minimax value greater than m
  - Search 1 returns win, Search 2 returns loss:
    - Minimax value equal to m

Discussion

CMPUT 496

- We learn something useful with both search outcomes
  - Search with boolean test ($v \geq m$) or ($v > m$)
  - Result True: lower bound on true minimax value
  - Result False: upper bound on true minimax value
- Important variants of alpha-beta search
  are based on this idea
  - SCOUT, NegaScout/PVS, MTD(f),...
- We will discuss the standard alpha-beta algorithm now
- Return to these ideas later
  - How to use boolean searches to speed up alpha-beta

Boolean Searches and Proof Trees

CMPUT 496

- Scenario:
  - Win with test ($v \geq m$)
  - Loss with test ($v > m$)
- Proof tree of the first search:
  - Our winning strategy: achieve at least m
- Disproof tree of the second search:
  - Opponent's winning strategy:
    prevent us from getting more than m
- Together, these two strategies prove that:
  - No player can do better than m ...
  - ... against a perfect opponent

Alpha-beta Search

CMPUT 496

- Use if we have more than two outcomes,
  e.g. numeric score
- Idea: keep lower and upper bounds (α, β)
  on the true minimax value
- prune a position if its value v falls outside the (α, β)
  window
  - $v < \alpha$ we will avoid this position,
    we have a better alternative
  - $v > \beta$ opponent will avoid this position,
    has a better alternative
  - If $v = \beta$ opponent can also ignore this position,
    has equally good alternative

Alpha-beta Search - Negamax Style

CMPUT 496

```
int AlphaBeta(GameState state, int alpha, int beta)
    if (state.IsTerminal())
        return state.StaticallyEvaluate()
    foreach legal move m from state
        state.Execute(m)
        int value = -AlphaBeta(state, -beta, -alpha)
        if (value > alpha)
            alpha = value
        state.Undo()
        if (value >= beta)
            return beta
    return alpha
```

- Changes from naive negamax in bold.
- Initial call:
  `AlphaBeta(root, -INFINITY, +INFINITY)`

Negamax Alphabeta Details

CMPUT 496

- Negamax - everything is from *current player*'s point of view
- Avoids two separate cases for AND, OR nodes
- Negate scores when changing from player to opponent on each level
- Example: score +5 for player becomes -5 for opponent
- Window (α, β) becomes $(-\beta, -\alpha)$ for opponent
- Example:
  - window (+5, +10) for current player
  - window (-10,-5) for opponent
  - These are exactly the same window!
  - Imagine mirroring the window along $x = 0$ axis

How does Alphabeta work? (1)

CMPUT 496

- Let v be value of node,
  $v_1, v_2, ..., v_n$ values of children
- By definition:
  in OR node, $v = \max(v_1, v_2, ..., v_n)$
- Fully evaluated child establishes lower bound on parent
- Example: if $v_1 = 5$,
  - $v = \max(5, v_2, ..., v_n) \geq 5$
- Other moves of value ≤ 5 do not help us
  - They can be pruned
- In code:
  - Set alpha to the best value so far
  - From now on, ignore moves of lesser (or equal) value

How does Alphabeta work? (2)

CMPUT 496

- By definition: in AND node, $v = \min(v_1, v_2, ..., v_n)$
- Fully evaluated child establishes upper bound
- Example: if $v_1 = 2$,
  - $v = \min(2, v_2, ..., v_n) \leq 2$

How does Alphabeta work? (2)

CMPUT 496

Main idea of pruning in alphabeta: the beta cut

- **`if (value >= beta) return beta`**
- The move is too good for the current player - cut.
- How can a move be too good?
- beta corresponds to -alpha for opponent one level up
- `value >= beta`
  is same as `-value >= -alpha` one level up for opponent
- That's the same as `value <= alpha` for opponent
- The opponent can already get alpha else where,
  is not interested in `value` and will not play to here

Python Codes for Alphabeta

CMPUT 496

- `tic_tac_toe_integer_eval.py`
  Static evaluation win = +1, draw = 0, loss = -1 instead of boolean evaluation at leaves
- `alphabeta.py`
  Algorithm, negamax style
- `alphabeta_test.py`
  Try on artificial game tree

From Exact Search to Heuristic Search

- All our algorithms so far search each move sequence until the end of game
- This is needed for exact solver
- Heuristic play:
  - Stop search earlier (e.g. at depth of d moves)
  - Evaluate leaf node by a heuristic
- Depth-limited searches can be good for move ordering
- Idea (details later):
  iterative deepening, increase depth 1, 2, 3,...
- Next slide: alphabeta with depth limit

Today's Topics - Lecture 10

- More on Minimax and Alphabeta
- Python sample codes
- Solve TicTacToe again, now with alphabeta
- Compare alphabeta with naive negamax and boolean negamax
- Iterative depening
- Alphabeta and proof trees; principal variation
- Search enhancements: transposition table

Depth-limited Alpha-beta Search


```
int AlphaBeta(GameState state, int alpha, int beta, int depth)
    if (state.IsTerminal() OR depth == 0)
        return state.StaticallyEvaluate()
    foreach legal move m from state
        state.Execute(m)
        int value = -AlphaBeta(state, -beta, -alpha, depth - 1)
        if (value > alpha)
            alpha = value
        state.Undo()
        if (value >= beta)
         return beta // or value - see failsoft
    return alpha
```

Python code: `alphabeta_depth_limited.py`, `alphabeta_depth_limited_tictactoe_test.py`

Review - Minimax and Alpha-Beta

- Solve game tree for general case
- More than two (win-loss) outcomes
- Result in leaf nodes: numerical score
- Example: win-loss-draw, coded as e.g.
  win = +1, draw = 0, loss = -1
- Minimax: player maximizes their score,
  opponent minimizes
- Alphabeta: prune if move is outside alphabeta window
- Meaning of window: moves outside are too bad for one of the players, that player will make a different choice earlier on

Summary

- From boolean case to numeric scores
- Naive Minimax and naive Negamax search
- Boolean searches to prove bounds on numeric scores
- Alphabeta search
- Next time:
  - Search improvements for boolean negamax and alphabeta
  - Search on DAGs
  - Reduce search depth in Go endgame solver

Minimax and Alphabeta Sample Code

- New static evaluation function in `tic_tac_toe_integer_eval.py`
- The example is for negamax, from `toPlay`'s point of view
- Can also be used for depth-bounded search, if evaluation is also called for interior nodes: `alphabeta_depth_limited_tictactoe_test.py`
- Note: this uses *no* heuristic, so it is blind search
- Evaluation is exact at leaf nodes, 0 everywhere else

Solve TicTacToe with Negamax and Alphabeta

- Compare Three Search Algorithms
  - Solve TicTacToe in three different ways
- Naive negamax, alphabeta, boolean negamax
- boolean negamax test in `boolean_negamax_test_tictactoe.py`
- Naive negamax, alphabeta test in `alphabeta_tictactoe_test.py`
- All solve the game
- Performance of Alphabeta, boolean negamax is similar
- Naive negamax not competitive - no pruning at all
- None of these programs use a heuristic
- We can easily add a heuristic

Activity 10a-c Questions to Explore

- For “open two”, should they be the same value for your own color and the opponent? Or is one more valuable than the other?
- How about a completely open line ... ? Should it be neutral, or a small advantage for the current player?
- What works better: adding up all features, or finding the most important one? Why?
- What are good “feature weights”? Experiment with different choices.
- Activities 10b and 10c: Test if the solver needs fewer nodes, and becomes faster.

Activity 10a: add a heuristic for TicTacToe

- Add a heuristic in the function `staticallyEvaluateForToPlay()`
- Idea: highest value for sure wins (3 in a row complete), lowest for losses
- If not won or lost: scan all eight lines on the board (3 horizontal, 3 vertical, 2 diagonal)
- Compute a score for each line depending on how good or bad it is for you
- Add up all those scores to get an evaluation function

Depth-limited Alphabeta

From `alphabeta_depth_limited.py`

```
def alphabetaDL(state, alpha, beta, depth):
    if state.endOfGame() or depth == 0:
        return state.staticallyEvaluateForToPlay()
    ...
        value = -alphabetaDL(state, -beta,
                             -alpha, depth - 1)
    ...

# initial call with full window
def callAlphabetaDL(rootState, depth):
    return alphabetaDL(rootState, -myInfinity,
                       myInfinity, depth)
```

Activity 10a Continued

- How to evaluate a line?
- Check different “features” and give a bonus or penalty when they are present
- If the line is blocked for both, then value 0
  - Examples: `xox` `.ox` `xxo`
- If a line is an “open two”, then very valuable
  - Examples: `xx.` `o.o`
- If a line is an “open one”, then has some value for that player
  - Examples: `x..` `.o.`
- Completely open line
  - Example: `...`

Experiment: Explore Depth-limited Alphabeta

`alphabeta_depth_limited_tictactoe_test.py`

- TicTacToe with evaluation scores in `tic_tac_toe_integer_eval.py`
- Runs alphabeta with different depth limits: iterative deepening
- Example 1: empty board. Result always 0
- Example 2: win for X. Result changes to win score when win is proven at depth 5
- Interpretation: against best response:
  - Black can win in 5 moves
  - Black cannot win in 4 or fewer moves

Alphabeta and Proofs

- Assume the minimax value of a game is m
- (Unbounded) alphabeta search will return the score m
- We can view alphabeta as finding two proofs at the same time
- Max player can get at least m
- Min player can keep score to m or below

Alphabeta and Playing Optimally - AND Node

- c_i is an AND node
- score(c_i) = m = min(score(c_{i1}), score(c_{i2}), ...
- AND can find (at least) one child c_{ij} with score(c_{ij}) = m, and play that move
- This leads to an OR node
- Repeat the same arguments until the end of the game

Alphabeta and Proofs - Continued

- If we store information about all nodes in the search, then we have these two strategies stored explicitly
- Also remember the relation to boolean minimax:
- If we know a candidate value m:
  - We can do two boolean searches with tests $\geq m$ and $> m$
  - Together they can verify that m is the minimax result

Principal Variation (PV)

- If both players play best moves: they follow a *principal variation* or PV of the search
- This is a move sequence with the property that each node in the sequence has a score of m
- Even with a depth-limited search and heuristic evaluation, a PV exists
- It will only go as deep as the search
- All nodes will share the same value as the heuristic evaluation of the last node in the sequence

Alphabeta and Playing Optimally - OR Node

- Assume both players follow best play based on the stored values
- Assume the root n is an OR node with minimax value $score(n) = m$
- Children $c_1, ..., c_k$:
- score(n) = m = max(score(c_1), score(c_2), ... score(c_k))
- OR can find (at least) one child c_i with score(c_i) = m, and play that move

Principal Variation and Proof Trees

- Consider the two proofs that the minimax value is m
  - Proof that the max player can get at least m
  - Proof that the min player can get m or less
- If both players follow their proof, they will play out a PV
- The reverse is also true:
- Assume both players follow a move sequence S, such that for all nodes along that sequence the minimax score is m
- Then there exist two proof trees:
  - $P1$ for max
  - $P2$ for min
  - S is the intersection of $P1$ and $P2$ (set of nodes that are in both trees)

Summary of Basic Solving Algorithms

- Alphabeta and Negamax
- Alphabeta performance similar to boolean negamax here
- Naive negamax much worse, no pruning
- Discussed relation between alphabeta and proof trees
- Principal variation: a line with best play for both

Minimax Search Enhancements

Search Enhancements - Overview

- Hashing to cache search results
- Transposition table
- From searching a tree to searching a DAG
- Iterative deepening and move ordering
- History heuristic
- More alphabeta search improvements

Big Idea: Caching Information

- A cache is an information store
- Example: on-chip cache for CPU
  - Accesses data much faster than loading from main memory
- Example: cache for rendered web pages
- Data in cache is stored locally as opposed to loading from web, parsing html, loading images, building on-screen image, recomputing...

How to Store Information About Search Nodes?

- In our minimax codes so far we did not store any information on search states
- The searches just returned a boolean, or an integer
- The searches used *depth-first* order
  - Go to first child, then first child of first child,...
- What if we want more detailed search results, and store them?
- Examples:
  - Store the best move
  - Store a proof tree
  - Re-use search results for a later, deeper search

Hashing and Transposition Table

- Idea:
  Store game positions and its search information
- Examples: minimax score, win/loss flag, best move, search depth reached in iterative search, number of nodes searched, timestamp (when solved),...
- How to store?
- Typically, a fixed-size array is used for a search
- For our simple example,
  we just use a Python dictionary

Get Best Move and PV

- It is easy to modify search to return both the score and a PV
- However, the overhead is quite large
- Very many move sequences are created during search
- Almost all of them are discarded later
- Much more efficient approach: use a **transposition table**
- Can get best move and PV information almost for free
- Several other important benefits

Storing a TicTacToe Position

- How to store?
- Standard approach: compute a *code* or *hash code* for the position
- Store in the transposition table under this code
- For TicTacToe, less than $3^9 = 19,683$ states total
- Can easily store all states that we search

Code for a TicTacToe Position

- Remember our 1-d array representation
- Number code for each point
  - EMPTY = 0
  - BLACK = 1, used for 'X'
  - WHITE = 2, used for 'O'

```
# Board stored in array of size 9:
# 0 1 2
# 3 4 5
# 6 7 8
```

- View state as array of codes:
- `s = [1, 1, 2, 2, 2, 1, 1, 0, 0]`
- Example: `s[0] = 1` means top left corner is 'X'

Example: Boolean Negamax with Simple Transposition Table

- `boolean_negamax_tt.py`
- Always try lookup first
- If succeeds:
  - Done, no search needed
- Otherwise:
  - Do the regular search
  - Store result in table before return from function

```
def negamaxBoolean(state, tt):
    result = tt.lookup(state.code())
    if result != None:
        return result
    ...
```

Store Code and Data in Simple Transposition Table

- Array of codes: s = [1, 1, 2, 2, 2, 1, 1, 0, 0]
- Code of state: treat codes as a base 3 integer = 112221100 in base 3
- code(s) =
  $1 \times 3^8 + 1 \times 3^7 + 2 \times 3^6 + 2 \times 3^5 +$
  $+2 \times 3^4 + 1 \times 3^3 + 1 \times 3^2 + 0 \times 3^1 + 0 \times 3^0$
- Store pairs `(code(s), data(s))`
- To store in a Python dictionary
  - Use `code(s)` as the key
  - Store `data(s)` as the value under that key

`boolean_negamax_tt.py` continued


```
    if state.endOfGame():
        result = state.staticallyEvaluateForToPlay(
        return storeResult(tt, state, result)
    for m in state.legalMoves():
        state.play(m)
        success = not negamaxBoolean(state,tt)
        state.undoMove()
        if success:
            return storeResult(tt, state, True)
    return storeResult(tt, state, False)

def storeResult(tt, state, result):
    tt.store(state.code(), result)
    return result
```

Example: Simple Transposition Table

- `transposition_table_simple.py`
- Store boolean result score (True or False) as value
  - It is easy to store best move as well
- Use code as key
- Lookup failure: return `None`
- Lookup success - return score: `True`, or `False`

```
class TranspositionTable(object):
  ...
  def store(self, code, score):
      self.table[code] = score

  def lookup(self, code):
      return self.table.get(code)
```

Apply Transposition Table to Solving TicTacToe

- `tic_tac_toe_solve_with_tt.py`
- About 3x faster than without table
- For larger problems (Go, Gomoku, ...) using the table can be several orders of magnitude faster

Full Transposition Table

- Problem with our simple approach so far:
- Does not scale to large searches
- Using dictionary to store all states will fill memory within seconds
  - For a fast program written in something like C++ anyway...
- We need a solution that works with fixed memory limit
- Only store most important states
- Need information-losing hash codes (see next slide)

Bitwise xor in Python

- Option 1: use ^
- `123 ^ 456`
- Option 2:
- `from operator import xor`
- `xor(123,456)`

Example: Code for 19 $\times$ 19 Go

- Why do we not always use the full code?
- Example: Full code for 19 $\times$ 19 Go
- 3 states per point, $19 \times 19 = 361$ points
- Total $3^{361} > 2^{572}$ different codes
- Not even considering history here, which is needed for Ko rule - Go has even more distinct states
- Storing everything in a table is not feasible
- Using full 573+ bit codes is not necessary
- Standard today: use 64 bit codes

Transposition Table in Fixed Size Array

- Where in table to store state s?
- For a fixed size array, we need to compute an array index from the 64 bit code of s
- Typical solution:
  - Use array of some size 2^n
  - Take the first n bits of `code(s)` as the array index
- Avoid collisions: store full 64 bit code as part of data
- At each lookup, compare full 64 bit code
- Do not trust 64 bit codes for proofs!
  - Verify solution tree without using hashing

Zobrist Hash Codes

- How to compute a good 64 bit code for a state?
- Standard: Zobrist hashing, `https://en.wikipedia.org/wiki/Zobrist_hashing`
- Prepare one random number `code[point][color]` for each (point, color) combination
- Code of state is bitwise logical `xor` over all points on the board
- example:
  `board[0] = WHITE, board[1] = EMPTY, board[2] = BLACK,...`
- `hashcode = code[0][WHITE] xor code[1][EMPTY] xor code [2][BLACK] xor ...`

Transposition Table Entries

- What data to store?
- Depends on type of search
- For boolean negamax we only needed one bit
  - True/False minimax value of state
- For alphabeta, iterative deepening, need to store more
  - Best move from this state
  - Search score
  - Flags: exact value or upper or lower bound
  - Search depth
  - A flag whether it is exact result or heuristic score
- Details - `https://chessprogramming.wikispaces.com/Transposition+Table`

State Space of TicTacToe

Enumerating the State Space of TicTacToe

- TicTacToe has a small enough state space to create and count all states
- We will do it both for the tree and the DAG model
- How much do we save from using Transposition Table?

Enumerating the Tree of TicTacToe


```
def countAtDepth(t, depth, positionsAtDepth):
    positionsAtDepth[depth] += 1
    if t.endOfGame():
        return
    for i in range(9):
        if t.board[i] == EMPTY:
            t.play(i)
            countAtDepth(t, depth + 1, positior
            t.undoMove()
```

Estimating the Tree of TicTacToe

- `tic_tac_toe_estimate_tree.py`
- Estimate for the tree model
- Branching factor: 9 at root, then 8, 7, ...
- Model as in Lecture 4

Estimated Tic Tac Toe positions in the tree model:
[1, 9, 72, 504, 3024, 15120, 60480, 181440, 362880, 362880]

Enumerating the DAG of TicTacToe

- With transposition table, we can now count the size of the TicTacToe state space in the DAG model
- `tic_tac_toe_count_dag.py`

```
def countTicTacToeDAG():
    tt = TranspositionTable()
    t = TicTacToe()
    positionsAtDepth = [0] * 10
    countAtDepth(t, 0, positionsAtDepth, tt)
    print("Tic Tac Toe positions in DAG model:
          positionsAtDepth)
```

Enumerating the Tree of TicTacToe

- Now we do exact count for the tree model
- `tic_tac_toe_count_tree.py`

```
def countTicTacToeTree():
    t = TicTacToe()
    positionsAtDepth = [0] * 10
    countAtDepth(t, 0, positionsAtDepth)
    print("Tic Tac Toe positions in tree model:
```

Enumerating the DAG of TicTacToe


```
def countAtDepth(state, depth, positionsAtDepth
    result = tt.lookup(state.code())
     if result != None:
         return
    tt.store(state.code(), True)
    positionsAtDepth[depth] += 1
    if state.endOfGame(): return
    for i in range(9):
        if state.board[i] == EMPTY:
            state.play(i)
            countAtDepth(state, depth + 1,
                         positionsAtDepth, tt)
            state.undoMove()
```

TicTacToe - DAG vs Tree Comparison

Run `tic_tac_toe_estimate_tree.py`, `tic_tac_toe_count_tree.py`, `tic_tac_toe_count_dag.py`

```
Estimated positions in the tree model:
[1, 9, 72, 504, 3024, 15120, 60480, 181440,
362880, 362880]
positions in tree model:
[1, 9, 72, 504, 3024, 15120, 54720, 148176,
200448, 127872]
positions in DAG model:
[1, 9, 72, 252, 756, 1260, 1520, 1140,
390, 78]
```

TicTacToe - DAG vs Tree Discussion

- Tree: estimate is exact at depth 0 . . . 5 (why?)
- DAG: No savings at lower levels (why?)
- Massive savings deeper in DAG

Another Application of Transposition Table: Solve All TicTacToe States

- `tic_tac_toe_solve_all.py`: Traverse whole state space
- Solve each state
- Store all solved nodes in transposition table
- Optional activity: modify the code to print each DAG state only once

```
Solve all TicTacToe states black win/draw/white win
Depth 0: 0 black, 1 draws, 0 white, 1 total positions
Depth 1: 0 black, 9 draws, 0 white, 9 total positions
Depth 2: 48 black, 24 draws, 0 white, 72 total positions
Depth 3: 128 black, 276 draws, 100 white, 504 total positions
Depth 4: 2336 black, 544 draws, 144 white, 3024 total positions
Depth 5: 5472 black, 3168 draws, 6480 white, 15120 total positio
Depth 6: 38016 black, 7200 draws, 9504 white, 54720 total positi
Depth 7: 59472 black, 28800 draws, 59904 white, 148176 total pos
Depth 8: 81792 black, 46080 draws, 72576 white, 200448 total pos
Depth 9: 81792 black, 46080 draws, 0 white, 127872 total positio
```

More Alphabeta Improvements

Alphabeta Improvement: Iterative deepening and Move Ordering

- We have seen iterative deepening before
- Search with depth limit of 1, 2, 3, ...
- Scenario now: heuristic alphabeta search with a (good) evaluation function
- Even shallow searches will often find a good move
- Remember - alphabeta is most effective if strongest move is tried first
- Alphabeta window reduced most, can cut more moves
- Idea: first try the strongest move from previous search
- This is a very strong heuristic and used in most alphabeta implementations

Alpha Zero vs Alphabeta Enhancements

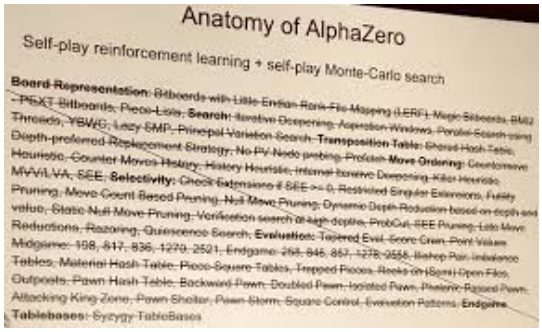


Image source: lifein19x19.com

- When the Alpha Zero approach works: no!
- (If we have enough computing power)

Alphabeta Improvement: History Heuristic

- Invented by Jonathan Schaeffer (past Dean of Science) in 1983
- Game-independent improvement
- Idea: keep track of which moves are effective in causing beta cuts in the search
- Give a bonus for those moves, try them earlier among all children
- Similar idea: countermove heuristic (Uiterwijk) - store good reply to a move

Summary

- This concludes our discussion of standard minimax algorithms
- Next topic: closer look at using knowledge in search
- After that: Monte Carlo Tree Search (MCTS) - a quite different way to approach minimax search problems
- However, it has the same goals as alphabeta
  - Heuristic search: play as well as possible when time limit is given
  - Solve: with unlimited time, eventually find the (perfect play) minimax solution
  - Most work on MCTS is on heuristic search, play well

Many More Alphabeta Enhancements

- Huge number of ideas have been tried in last 70 years
- Examples:
  - Minimal window search, Scout, PVS
  - Quiescence search
  - Parallel search
  - Late move reductions
- Very good website: `https://chessprogramming.wikispaces.com/`
- Do we still need to learn all these enhancements?

Knowledge in Heuristic Search

We can use knowledge in a heuristic search to evaluate state, actions and in other ways.
Probabilities, preferences, ordering, and rankings are values that are used in heuristic search.
We can represent knowledge by using rules, patterns, features, neural net
Acquiring knowledge either by manual or machine learning.

Knowledge for heuristic search

There are two big kinds of knowledge for heuristic search

1. state evaluation
2. move evaluation
3. time control
4. search depth control
5. algo optimization and tuning
6. many more

We will talk about the first two

State evaluation

State evaluation is a mapping from (full) state to one number. It basically describes now good is that state, there are other terms like evaluation function and positive evaluation.

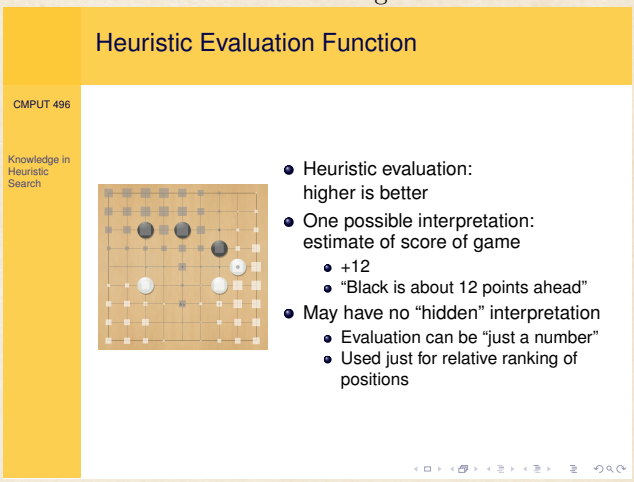
In the terminal states or the leafs nodes in the state space, we know the exact evaluation either a win, loss, draw, or a numerical value.
In games there are two kinds of state evaluation

1. heuristic evaluation - where the higher number is better and is used for ordering state evaluations
2. winning probability - the probability of winning in that state

The most important use of state evaluation is the evaluation function in search. The leaf nodes of search are evaluated by this function. **The heuristic evaluation is used in non-terminal states**.
Interior nodes in search are evaluated by the minimax rule, it returns the search evaluation from the leaf/terminal node.

Example: Chess

In chess the queen piece can have a value of 9, rook a value of 5, pawn a value of 1, and the state evaluation could be the material sum of all the pieces. The higher the value the "better" it is. There isn't a "deeper meaning" with regards to this number, it just a number used for move ordering considerations.



Relative vs absolute evaluation

For decision-making the evaluation number themselves don't matter, only the ordering given by the number matters. It decides preference on what moves to evaluate first from a particular state.

This means that states with the same evaluation value are "equally as good" to each other. It could also mean that states are "similar" to each other as well.

Winning probability

If a game is determined by chance then it has a winning probability. Moreover, the winning probability **is the minimax score**. For example, the probability of rolling two sixes is 1/36. The 1/36 is the evaluation for that state.

If the game doesn't have probability then the probabilities are binary. The perfect player wins 100% or 0%.

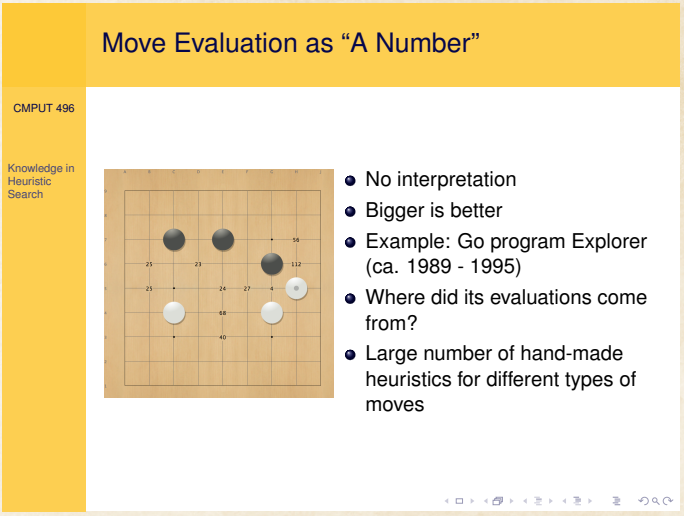
Probabilities in general can arise from

- imperfect understanding of the game
- generalization and abstraction in our evaluation
- randomness in our strategy
- randomness or imperfect understanding of the opponent's strategy

Move evaluation

Move evaluation is the mapping of a move (action) to a number. It measures how good a move is, for example the probability that this is the best move. Knowing the utility of the move can be used to filter out bad moves. Other terms include action value, move value, and Q-value.

The main use is for action selection in search, and in simulation. It can also be used for move ordering in search and pruning. Like the state evaluation we have a probabilistic and a non-probabilistic interpretation.



Move Evaluation as a probability

Assume move i has probability p_i There could be two interpretation:

1. p_i is the probability that move i is a win
2. p_i is the probability that move i is the best move

Which one you use depends on how you compute or estimate those numbers.

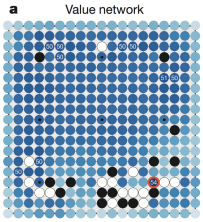
Mixing exact and heuristic evaluation

We can mix exact and heuristic evaluations to get true proofs of wins and losses, if we are careful. The heuristic evaluation can be used to speed up an exact proof by ordering the best moves to be searched first.

Example - Winning Probabilities in Value Net

CMPUT 496

Knowledge in Heuristic Search



- Can use machine learning to *learn estimates of win probabilities*
- Example: AlphaGo's value network
- Deep neural net
- Maps states to learned win probabilities

Image source: Silver et al, Mastering the game of Go with deep neural networks and tree search, Nature

Move vs State Evaluation

Case 1

We have a state evaluation and we need a move evaluation. This is easy to do as one can evaluate all possible neighboring states from the current state to obtain the move evaluation. The problem with this is speed, as you need to evaluate all neighboring states.

Case 2

We have a move evaluation but we need a state evaluation. This is harder as we don't know how good the current state is to begin with. (We could have great moves but a bad state which cancel itself out basically). You can do greedy by choosing the best moves in sequence, but greedy algorithms aren't really the best in most cases.

Acquiring Evaluation knowledge

We currently get them from machine learning, but older methods involve hand-coding rules and local goal-directed searches.

In order to acquire knowledge we first need to represent them, currently this is done by neural nets.

Hand-coded rule-sets

Handcoded Rules

CMPUT 496

Knowledge in Heuristic Search

```
def selfatari(board, move, color):
    maxoldliberty = maxliberty(board, move,
                               color, 2)
    if maxoldliberty > 2:
        return False
    cboard = board.copy()
    isLegal = cboard.move(move, color)
    if isLegal:
        newliberty = cboard.liberty(move, color)
        if newliberty == 1:
            return True
    return False
```

- Most direct way
- Example: heuristic move filter, "avoid selfatari"

Simple features

Simple Features

CMPUT 496

Knowledge in Heuristic Search

```
enum FeBasicFeature{
FE_PASS_NEW,
FE_PASS_CONSECUTIVE,
FE_CAPTURE_ADJ_ATARI,
...
FE_CAPTURE_MULTIPLE,
FE_EXTENSION_NOT_LADDER,
FE_EXTENSION_LADDER,
...
FE_TWO_LIB_SAVE_LADDER,
FE_TWO_LIB_STILL_LADDER,
...
FE_SELFATARI,
FE_ATARI_LADDER,
...
FE_DOUBLE_ATARI,
FE_DOUBLE_ATARI_DEFEND,
FE_LINE_1,
FE_LINE_2,
FE_LINE_3,
...
}
```

- Idea: each feature is a boolean statement about a state, or a move
- Each feature is simple and easy to compute
- With machine learning, we can construct an evaluation function from a combination of many simple features
- Move feature vector: (0,0,1,...,1,1,0,...,1,0,...0,0,...)
- Examples: next few slides

Remi Coulom's Simple Features (1)

CMPUT 496

Knowledge in Heuristic Search

Feature	Level	γ	Description
Pass	1	0.17	Previous move is not a pass
	2	24.37	Previous move is a pass
Capture	1	30.68	String contiguous to new string in atari
	2	0.53	Re-capture previous move
	3	2.88	Prevent connection to previous move
	4	3.43	String not in a ladder
	5	0.30	String in a ladder
Extension	1	11.37	New atari, not in a ladder
	2	0.70	New atari, in a ladder
Self-atari	1	0.06	
Atari	1	1.58	Ladder atari
	2	10.24	Atari when there is a ko
	3	1.70	Other atari
Distance to border	1	0.89	
	2	1.49	
	3	1.75	
	4	1.28	

Remi Coulom's Simple Features (2)

CMPUT 496

Knowledge in Heuristic Search

Distance to previous move	2	4.32	$d(\delta x, \delta y) = \lvert\delta x\rvert + \lvert\delta y\rvert + \max(\lvert\delta x\rvert, \lvert\delta y\rvert)$
	3	2.84	
	4	2.22	
	5	1.58	
	...	...	
	16	0.33	
	≥ 17	0.21	
Distance to the move before the previous move	2	3.08	
	3	2.38	
	4	2.27	
	5	1.68	
	...	...	
	16	0.66	
	≥ 17	0.70	

Source: Remi Coulom, Computing Elo Ratings of Move Patterns in the Game of Go

Neural Nets

Neural Nets

CMPUT 496

Knowledge in Heuristic Search

Convolutional Neural Network (CNN)

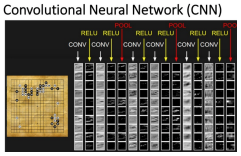


Image source:

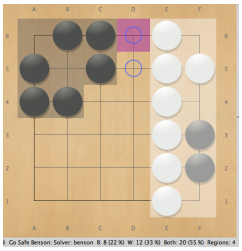
https://www.slideshare.net/ShaneSeungwhanMoon/how-alphago-works

- Represent knowledge in (large number of) weights of the neural net
- Lower levels of net encode local knowledge (e.g. 3x3, 5x5)
- Higher levels can express global evaluation
- Much more on nets later in the course

Example of Exact Knowledge: Benson's Algorithm in Go

CMPUT 496

Knowledge in Heuristic Search



- Benson's algorithm finds stones and territories that are *unconditionally alive*
- No matter what opponent plays: stones never captured
- A generalization of the "two eyes" concept
- Can be used as an exact filter in a program - do not generate moves in safe territory

Pattern DB

Pattern Databases

CMPUT 496

Knowledge in Heuristic Search

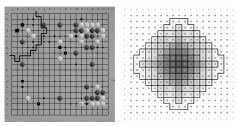


Image source: Stern et al, Bayesian Pattern Ranking for Move Prediction in the Game of Go

- Large patterns can be learned from master games, if they are frequently used
- In Go, typically we have many different sizes of pattern:
- From small 3x3 patterns to full board
- A main question is how to evaluate such patterns
- Measure how often the move in the center is played immediately, or later

Part III

Simulation and Monte Carlo Tree Search

Simulation

Simulation is the imitation of the operation of a real-world process or system over time

The main idea is to learn learn information from random sequences of decision steps.

For example we can estimate pi by generating random points within a 1x1 square= and see if that point lands within a circle. The fraction of points within the circle estimate for its area of 4π

Limitation

Convergence of the basic method is slow, it has high variance but no bias. Much faster methods exist if the function have a nice properties such as smoothness. The basic Monte Carlo (MC) is a fall back for cases without "nice" structures

Simulation Model

To simulate something we need a model.
Physics for neutron diffusion, laws of physics for path tracing, games have a legal move set, etc.

Simulation Model

CMPUT 496

- To simulate something we need a model
- Neutron diffusion: physics - laws of motion, speed of neutrons, absorption by different materials, radioactive decay of different uranium isotopes,...
- Path tracing: light sources, laws of optics, shadows, reflection/light scattering, indirect light, ...
- Games:
  legal moves, outcome at the end
- Games with chance:
  simulated dice throws,
  possible distributions of unknown cards,...

Garbage In - Garbage Out Principle (GIGO)

CMPUT 496

- A simulation can only be as good as the underlying model
- If you feed great data to an invalid model, you typically get garbage
- Examples:
  - Missing relevant physics
  - Wrong initial conditions
  - Numerical instability, cascading errors
  - Bugs in computer code and/or hardware
  - Not implementing the rules of Go properly

Simulation and Random Walks in Heuristic Search

CMPUT 496

- Early application: GSAT and WalkSAT for Boolean Satisfiability Problem (SAT)
- Given a boolean formula
- Example: $(x_0 \vee x_2) \wedge (\neg x_1 \vee \neg x_2)$
- Check if it is *satisfiable*: an assignment of true and false to the variables which makes the formula true
- Example: Set $x_0 = \mathit{True}, x_1 = \mathit{False}, x_2 = \mathit{False}$
- Whole formula becomes true
- Solving large SAT formulas is a difficult problem
- Best solvers use heuristic search

Solving SAT by Systematic Search

CMPUT 496

- Given SAT formula with n variables, $x_0, \ldots x_{n-1}$
- Can solve by systematic search, trial and error
- Set $x_0 = \mathit{True}$
  - Simplify formula
  - Solve SAT problem with $n-1$ variables
- If fail: Set $x_0 = \mathit{False}$
  - Simplify formula
  - Solve SAT problem with $n-1$ variables
- Worst case cost: exponential in n
  - Need to try many of the 2^n variable assignments

GSAT and WalkSAT: Solving SAT by Biased Random Walk

CMPUT 496

- Local search methods developed in 1990's
- Start with random assignment of True or False to each variable
- If formula is true: stop, success
- If not, use heuristics to *flip* value of one variable
- Balance *exploitation* and *exploration*
  - Exploitation: flip a variable that makes the "largest possible improvement" of the formula
  - Exploration: flip a random variable
- Restart if no progress for a while
- How does local search compare to systematic search?
  - No clear winner, different strengths/weaknesses

From Games with Chance Elements to Games With No Chance

CMPUT 496

- Games with chance element (dice, hidden cards)
  - Using random simulations is a natural idea
  - Tried even in the earliest programs
- Games without chance element (chess, checkers, Go,...)
  - Using random simulations is much less natural
  - It took a lot longer to develop those methods
  - Often, it also works very well
  - Our first example: TicTacToe
  - Second example: Go

Simulation in Game Tree Search - Backgammon

CMPUT 496

Image sources:

`www.backgammoned.net`

- Early success of simulation methods in games: Backgammon
- "Rollout" games with many different sequences of dice throws
- Play move that is most successful in these rollouts
- Backgammon was also an early success story for neural networks
  We will discuss that story later

Random Simulation in TicTacToe

CMPUT 496

- From given state, finish game with moves selected uniformly at random
- In TicTacToe, all empty squares are legal moves
- End simulation when game is over by rules:
  - Three in a row
  - Board full
- Implementation: method `simulate(self)`
  - In file `tic_tac_toe.py`

Simulation in Backgammon

CMPUT 496

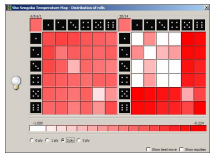


Image sources: `www.bkgm.com/gnu/AllAboutGNU.html`

- Picture: simulation result of all possible dice throws
- white = good winrate, red = bad
- Right side: risky move
  - Many throws lead to sure win
  - Many other throws lead to sure loss
- Left side: safe move
  - Outcomes more similar
  - Here, this move is better in expectation
- Knowing this distribution allows you to make better decisions

Method TicTacToe.simulate()

CMPUT 496

```
def simulate(self):
    allMoves = self.legalMoves()
    random.shuffle(allMoves)
    i = 0
    while not self.endOfGame():
        self.play(allMoves[i])
        i += 1
    return self.winner(), i
```

- For random play in TicTacToe, it's enough to shuffle the list of moves once, then play in that order
- Not true in Go (discuss why?)

Use Simulations as Evaluation Function

CMPUT 496

Evaluate: how good is a game state?

- Exact answer:
  - Run solver
  - Compute minimax value
- Heuristic, first (old) answer:
  - Run depth-limited alphabeta search
  - At depth limit: call heuristic evaluation function
  - Compute minimax value
  - Problem: how to create evaluation function?
- **Heuristic, second (new) answer:**
  - Run simulations, score final result
    - Win = 1, loss = 0 (draw = 0.5)
  - Compute *winrate* over all simulations

Simulation Player Implementation - simulate

CMPUT 496

- `SimulationPlayer.simulate(self, state, move)`
- Play `move` from given `state` - changes state
- Evaluate state after the move:
- Run `self.numSimulations` from it
- After simulations: `undoMove` to restore previous state
- Evaluation of move:
  average outcome of these simulations

Simulation-Based Player

CMPUT 496

- Uses 1-step (1-ply) lookahead to evaluate all moves
- For each legal move:
  - Run *n* simulations
  - Measure the winrate (winning percentage) for these simulations
- After all simulations:
  - Play move with highest winrate
- Implementation: `simulation_player.py`

SimulationPlayer.simulate

CMPUT 496

```
def simulate(self, state, move):
    stats = [0] * 3
    state.play(move)
    moveNr = state.moveNumber()
    for _ in range(self.numSimulations):
        winner, _ = state.simulate()
        stats[winner] += 1
        state.resetToMoveNumber(moveNr)
    state.undoMove()
    eval = (stats[BLACK] + 0.5 * stats[EMPTY])
           / self.numSimulations
    if state.toPlay == WHITE:
        eval = 1 - eval # Negamax view
    return eval
```

Flat Monte Carlo

CMPUT 496

- The method based on 1-ply lookahead + simulations is sometimes called *Flat Monte Carlo*
- Monte Carlo method: uses random simulations
- Flat: does not build a deep tree, only 1 ply (1 move) lookahead
- Contrast: Monte Carlo Tree Search builds a (often very deep) tree

Simulation Player Implementation - genmove

CMPUT 496

- `SimulationPlayer.genmove(self, state)`
- For each move: Evaluate it by simulation
- Collect and compare winrates for all moves
- Pick the move with best winrate

SimulationPlayer.genmove

CMPUT 496

```
def genmove(self, state):
    moves = state.legalMoves()
    numMoves = len(moves)
    score = [0] * numMoves
    for i in range(numMoves):
        move = moves[i]
        score[i] = self.simulate(state, move)
    bestIndex = score.index(max(score))
    best = moves[bestIndex]
    return best
```

Match 1: 10 simulations/move, 100 games each color

CMPUT 496

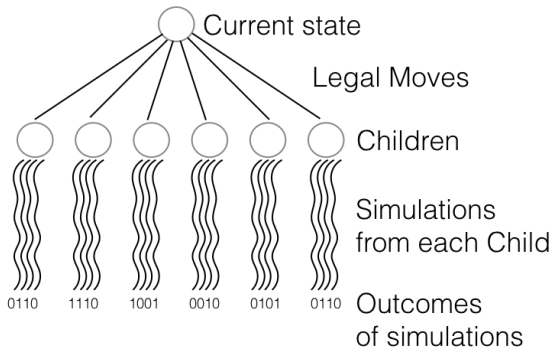
- Results in table:
  - Black player (X, name on left side)
  - Result vs White player (O, name on top)
- Perfect player never loses with either color
- Going first is a big advantage (unless both are perfect)
- Note: numbers will change if re-run, but results will be similar with high probability

Table: Wins/Draws/Losses (W/D/L), 10 simulations/move, 100 games each color.

Black	Sim(10)	Perfect	Random
Sim(10)	**62W**/21D/17L	0W/**76D**/24L	**97W**/3D/0L
Perfect	**77W**/23D/0L	0W/**100D**/0L	**100W**/0D/0L
Random	9W/5D/**86L**	0W/20D/**80L**	**64W**/7D/29L

Simulation-Based Player

CMPUT 496



Scaling of Simulation Player vs Perfect

CMPUT 496

- Vary number of simulations 1, 10, 100, 1000
- Separate stats as Black, as White
- Results for Random and Perfect added for comparison
- Increasing simulations clearly helps
- 1000 simulations/move seem to play almost perfectly?
- Activity 7b: re-try this experiment, run more games
- TicTacToe is simple. In Go, Sim(1000) still plays poorly

Player	As Black	%	As White	%
Random	0W/20D/**80L**	10%	0W/0D/**100L**	0%
Sim(1)	0W/19D/**81L**	9.5%	0W/7D/**93L**	3.5%
Sim(10)	0W/**80D**/20L	40%	0W/24D/**76L**	12%
Sim(100)	0W/**100D**/0L	50%	0W/**77D**/23L	38.5%
Sim(1000)	0W/**100D**/0L	50%	0W/**100D**/0L	50%
Perfect	0W/**100D**/0L	50%	0W/**100D**/0L	50%

Match Simulation-Based Player vs Perfect and Random Players

CMPUT 496

- `simulation_player.py`
- `perfect_player.py` solves game at each step
- `random_player.py` selects move uniformly at random
- `play_match.py` run test games, print statistics
- How do these players compare?
- How does the strength of the Simulation Player change if we increase the number of simulations?

Scaling Simulation Player vs Random

CMPUT 496

- Vary number of simulations 1, 10, 100, 1000
- Separate stats as Black, as White
- Results for Random and Perfect added for comparison
- Increasing simulations clearly helps
- Sim(1000) as white better than perfect???

Player	As Black	%	As White	%
Random	64W/7D/29L	67.5%	29W/7D/64L	32.5%
Sim(1)	82W/9D/9L	86.5%	63W/15D/22L	70.5%
Sim(10)	97W/1D/2L	97.5%	78W/8D/14L	82%
Sim(100)	99W/1D/0L	99.5%	88W/9D/3L	92.5%
Sim(1000)	97W/3D/0L	98.5%	**91W/5D/4L**	93.5%
Perfect	100W/0D/0L	100%	**80W/20D/0L**	90%

Comments on Experiments

CMPUT 496

- 100 games is not enough to get precise numbers
  - Still large statistical error
  - Enough to get a rough first idea
- Benefit of more simulations is clear
- Does it play perfectly?
  - In TicTacToe, maybe close to perfect
  - In harder games like Go, not at all

Simulations in Go3

CMPUT 496

- As in TicTacToe, simulations used as state evaluation
- Use simulations to finish game many times from current position
- Keep winrate statistics to evaluate state
- How to do simulation in Go?
- Two simulation methods implemented in Go3
  - Almost-random
  - Rule-based (discussed later)

Comments on Experiments (2)

CMPUT 496

- Sim(1000) can exploit Random better than the perfect player
- Confirmed with 1000 games - see below
- Probable reason:
- Tie-breaking towards moves that are more successful in random simulations
- Optional activity: write a perfect player with simulation-based tiebreaking

Player	As Black	%
Sim(1000)	988W/12D/0L	99.4%
Perfect	991W/9D/0L	99.55%

Player	As White	%
Sim(1000)	908W/59D/33L	**93.75%**
Perfect	799W/201D/0L	**89.95%**

Almost-Random Simulations in Go3

CMPUT 496

- Remember Go1 and Go2, random Go players
- Selected moves uniformly at random
  - Except: do not fill one-point eyes
- Almost-random simulation in Go3 works the same way
- It will choose almost-random moves in its simulations
- Filter eye-filling moves only
- Pick all other moves with equal probability
- Pass in simulation only if all board moves are eye-filling

Go3 - Simulation-Based Go Players

CMPUT 496

- Go3 implements several variations of simulation-based players
- All choose their best move based on success in simulations
- Go3 implements two different simulation policies
- Go3 implements two different move selection algorithms at the root
- Go4 will have even more simulation policies

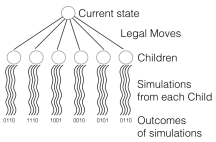
Move Selection in Go3

CMPUT 496

- Two algorithms: simple and UCB
- Simple is the same as in `simulation_player.py`
  - For each move, try n simulations starting with this move
- Second algorithm is UCB (later)
  - Smarter choice of which moves to simulate more often

Simple Move Selection Details

CMPUT 496



- For each legal move m_i
  - Play m_i
  - Run n random simulations
  - Undo move m_i
  - Count number of wins w_i
  - Compute winrate w_i/n
- Play the move with maximum winrate
- move $= \arg\max_i w_i/n$
- Difference to TicTacToe:
  - Legal moves include pass

Scaling of Go3 vs Go2

CMPUT 496

- Board size 5×5, komi 4.5
- Simulations/move: 10, 20, 50, 100
- Opponent: Go2 random player
- Go3 clearly better than random

Player	Wins %
Sim(10)	98% ($\pm$1.4)
Sim(20)	100% ($\pm$0)
Sim(50)	100% ($\pm$0)
Sim(100)	100% ($\pm$0)

Pass in Simulation vs Pass in Game

CMPUT 496

Simulations

- Regarding passing, behave like Go1 and Go2
- No pass except at very end to avoid filling eyes

Move selection for player

- Go3 player move selection is different from simulations, Go1 and Go2
- In Go, pass is always legal
- Go3 player can pass earlier if it has the best winrate
- Examples:
  - All moves on board are bad tactically
  - All moves on board are in own or opponent territory

Simulation Speed in Go Revisited

CMPUT 496

- Example: 7×7 Go
- Playing one move in simulation is more complex in Go than TicTacToe
  - Simulation can be longer than 50 moves
- Many more legal moves than TicTacToe (almost 50 at start of game)
- Example: 100 simulations/move: total $100 \times 50 \times 50 = 250000$ simulated moves *for making a single move decision in opening*
- To play whole 7×7 game: many millions of simulated moves

Simulation Speed in Go vs TicTacToe

CMPUT 496

- Speed in Go is quite slow
- Simulations take much longer than in TicTacToe
- Max. 9 moves in TicTacToe
- Roughly $n \times n$ on board size n in the opening
- Example: 7×7 Go
- Simulation can be longer than 50 moves
- Reason:
  - Capture large blocks
  - Play back onto those newly empty points

Uniformly Random Simulations in Go - Strengths and Weaknesses

CMPUT 496

- Last class: simulation-based player Go3
- How does Go3 play Go?

Strengths:

- Can find many simple tactics
- Can find sure wins near the end
- Can avoid many simple blunders that the random player may play
- Improves with number of simulations, up to a limit

Uniformly Random Simulations in Go - Strengths and Weaknesses

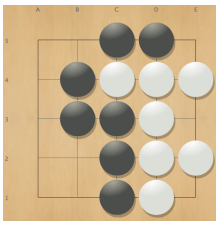
CMPUT 496

Weaknesses:

- Main weakness:
  assumes that the opponent plays randomly...
- It will play the moves that work best against random
- Those moves may fail against strong opponents
- Example: make a “silly threat”
  - A strong opponent will answer the threat, no gain
  - A random opponent will only answer with small probability
    - Effect: the threat looks good in simulation

Uniformly Random Simulations in Go - Example 3 - strength

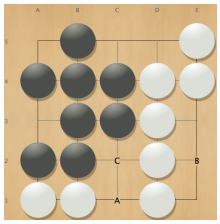
CMPUT 496



- Black to play, 3.5 komi
- e5 is blunder, “self atari” endangers three stones
- FlatMC can clearly see that e5 is worse than other moves
- Winrate goes down since these stones are often captured in simulations
- Almost any other move wins (b5 is best)

Uniformly Random Simulations in Go - Example for Strength

CMPUT 496

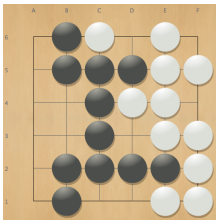


Win rates:
(e2, 1.0),
(c2, 0.75),
(c1, 0.72), ...

- Black to play, 5.5 komi
- e2 is only winning move - kills whole white group
- c1 captures two stones, but is not enough to win
- With 100 simulations, Go3 clearly finds e2 is best
- Weakness: program also thinks that bad moves c1 and c2 win
  - It does not matter in practice since e2 is ranked higher

Uniformly Random Simulations in Go - Example 4 - strength

CMPUT 496

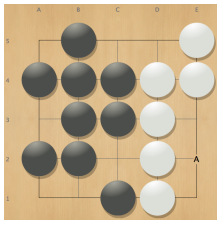


Win rates:
(d6, 0.86),
(d1, 0.79),
(d3, 0.72),...

- d6 is worth 2 points territory (plus the stone played)
- d1 is worth 1 extra point
- d3 is neutral, no extra points
- Strength: program gets correct ordering of move values $v(d6) > v(d1) > v(d3)$
- Weakness: difference in winrates is not large...
  - Lots of random noise from plays after the first move

Uniformly Random Simulations in Go - Example Continued

CMPUT 496

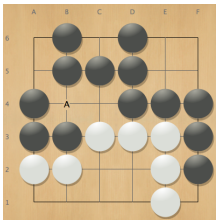


Win rates:
(e2, 0.98),
(a1, 0.36),
(c5, 0.36),...

- Position after black mistake playing c1
- FlatMC now finds a win for White very clearly
- Simulations find that making two eyes leads to win for White
- Emergent good behavior, achieved without any special knowledge of how to make eyes
- The simulations show the value of this move

Uniformly Random Simulations in Go - Example 5 - Weakness

CMPUT 496

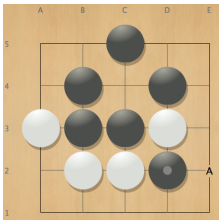


Win rates:
(b4, 0.35),
(a5, 0.24),...

- White to play, 1.5 komi
- b4 and a5 are silly threats, bad moves, lose points
- Simulation player likes them best
- Against the random opponent in the simulations, these moves often work
- The quiet move at c4 would win
- However, its simulation winrate is low
  - Why? My guess is that the white corner group often dies in simulation

Uniformly Random Simulations in Go - Example 6 - Weakness

CMPUT 496



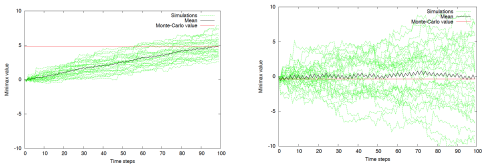
Win rates: **(e2, 0.34)**, (d1, 0.32), (e3, 0.3),...

- Simulations are over-optimistic here
- White totally destroyed
- White still has > 30 % winrate in random simulation
- White sometimes captures the black stone d2 in the simulations and wins
- e2, d1, e3 are all threats to capture d2 that:
  - Fail against a competent opponent
  - Can work against random

Bias vs Variance - What is More Important?

CMPUT 496

- Silver and Tesauro 2009: low bias is much more important for good Monte Carlo simulations
- High bias: drift, result gets worse with more time steps in simulation (left picture)
- High variance can be addressed by doing more simulations (right picture)



Ideal Simulation vs Reality

CMPUT 496

- An *ideal simulation* would preserve wins and losses exactly
  - If starting position is a win with best play by both: $\rightarrow$ simulation is also a win
  - If starting position is a loss $\rightarrow$ simulation is also a loss
- This would imply:
  - Each simulation is a perfect game
  - Simulation gives a perfect evaluation function
- That is VERY unrealistic!
  - If we know how to play a game perfectly, we do not need any search or simulation...
- What are properties of *good* simulations in practice?

Improving Simulations

CMPUT 496

- Random simulations make many mistakes
- Random simulations have some systematic weaknesses, causing bias and variance
- How to improve them?
- Answer from early Go program MoGo: add game-specific filters and rules
- One way is to hardcode “obvious local replies”
- Goal: make the simulation more stable
  - Safe stones and territories should remain safe for the rest of the simulation

Real Simulations - Mean, Bias and Variance

CMPUT 496

- Evaluate a state by doing n simulations
- Simulation i outcome is win ($s_i = 1$) or loss ($s_i = 0$)
- Evaluation = winrate (mean outcome) of simulations
- *Mean* outcome μ of the simulations
  - $\mu = \frac{\sum s_i}{n}$
- *Variance*
  - $var = \frac{\sum (s_i - \mu)^2}{n}$
- Ideal playout: mean = minimax score, variance = 0
- Real playout: mean $\neq$ minimax score, variance > 0
- *Bias* = Difference between mean and minimax score m
  - bias = $|m - \mu|$

The MoGo Program

CMPUT 496

- MoGo was one of the first successful Monte Carlo Go programs
  - Developed around 2007/2008
  - Won many competitions
  - Won (high handicap) games vs human professionals
- Two main reasons for success:
  - First Go program to use the UCT algorithm for Monte Carlo Tree Search (later in this course)
  - Improved playout policy by simple tactical rules and 3×3 patterns (this class)

Improving Go3 Almost-Random Policy

CMPUT 496

- Go3 contains several ways to change the simulations
- Improve on almost-random default simulations in Go3
- Filter:
  - Avoid some huge blunders in simulated games
  - Examples:
    - Do not fill eyes (already in Go3)
    - New: avoid "selfatari"
- Selective Move Policies:
  - Find *promising* moves based on rules
  - If promising moves exist: ignore all other moves

More Efficient: Lazy Filtering

CMPUT 496

- Simple algorithm is inefficient
- Calls `filter` for all legal moves on the whole board
- Much faster way: pick random move first, then check

```
def filterMovesAndGenerate(moves):
    while len(moves) > 0:
        candidate = random.choice(moves)
        if filter(candidate):
            remove candidate from moves
        else:
            return candidate
    return None

def generateMoveWithFilter():
    moves = generate_moves()
    return filterMovesAndGenerate(moves)
```

Details on Filtering

CMPUT 496

- A filter decides whether a move should be used or not
- We have seen a simple filter already
- Random players Go1, Go2 filter one point eyes
- The Go3 default simulation policy also filters these eye-filling moves
- We can filter other bad moves, too
- How does filtering work?

Filtering - How to Efficiently Remove a Move

CMPUT 496

- `moves` stored in Python list or array
- How to remove move at some index `i`?
- Slow way: move all elements `i+1, i+2, ...` down by one
- Much faster way: replace `moves[i]` by last element

```
moves[i] = moves[-1]
moves.pop()
```

- Note: much faster but changes order of list items

```
>>> moves = [1,2,3,4,5,6,7,8,9]
>>> i=2
>>> moves[i] = moves[-1]
>>> moves.pop()
>>> moves
[1, 2, 9, 4, 5, 6, 7, 8]
```

Simplest Filtering Algorithm

CMPUT 496

- Here, `filter` can be any move-checking function
- A checking function returns a boolean result - should a move be filtered?
- Examples: eye-filling, selfatari, ...

```
def naiveGenerateMoveWithFilter():
    moves = generate_moves()
    for move in moves:
        if filter(move):
            remove move from moves
    if len(moves) > 0:
        return random.choice(moves)
    else:
        return None
```

Filters in Go3

CMPUT 496

- Always: filter eye-filling moves
- Optional: also filter selfatari moves
- Next slides:
  - What is atari and selfatari?
  - Implement the selfatari filter

Atari

CMPUT 496

- "Be in atari" means stones have only 1 liberty
- "Give atari" means to reduce the opponent's stones to 1 liberty
- Most direct form of threat in Go
  - First picture: White e5
    "gives atari on Black d5 and c5"
  - d5 and c5 now have only one liberty at b5
  - Second picture:
    To defend, Black connects at b5
  - Third picture: Black played elsewhere, allowing White to capture on b5

Selfatari Filter for Existing Block

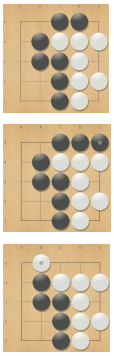
CMPUT 496

```
def selfatari(block, move):
# move is on a liberty of block
oldNumLiberties = libertyCount(block)
if oldNumLiberties == 2:
    play(move)
    newNumLiberties = libertyCount(block)
    undoMove()
    if newNumLiberties == 1:
        return True
return False
```

- Existing block, play on own liberty
- Would it have only 1 liberty afterwards?

Prevent Selfatari

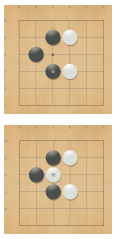
CMPUT 496



- Selfatari means to reduce your own stones to one liberty
- Most of the time, selfatari is a basic type of blunder
- Stones c5, d5 have two liberties
- Bad move e5 Black takes away one liberty
- Three stones now have only one liberty
- Now White can capture three stones

Single Stone Selfatari Filter

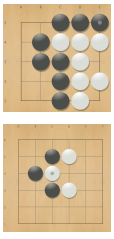
CMPUT 496



- Move creates new block, has no neighbors of own color
- Move is not a capture
- Stone has only 1 liberty
- Usually, such moves are very bad, only strengthen the opponent
- Sometimes, such a move is a good sacrifice

Implementing Selfatari Filter

CMPUT 496



Two cases

- Case 1, as in example before
  - Existing block with two liberties
  - Own move fills one liberty
- Case 2, single stone selfatari
  - New block, single stone
  - Placed so that it has only one liberty

From Filters to Rules and Patterns

CMPUT 496

- Filters are one way to improve simulations
  - Avoid moves that are typically bad
- We can also go the opposite way:
  - Generate only moves that are typically good, urgent
- Rules and patterns follow this approach

Rule-Based Randomized Playout

CMPUT 496

- Heuristic rule selects subset of all legal moves
- Choose randomly among those moves

```
def generateMoveWithRulesAndFilter():
    moves = ruleBasedGenerateMoves()
    move = filterMovesAndGenerate(moves)
    if move != None:
        return move
    moves = generateOtherLegalMoves()
    return filterMovesAndGenerate(moves)
```

From Rules to Patterns

CMPUT 496

- So far we looked mostly at tactical filters and rules
- Those were based on liberties of blocks
- Another choice: look at a small local area around move
- Decide which moves are good or bad, based on *pattern* of stones nearby
- Example from original MoGo program: 3×3 and 2×3 patterns

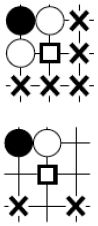
Combining Rules and Filters

CMPUT 496

- We can use both rules and filters
- Example: rule selects move m, but it is eye-filling
- Still filter such moves
- Call `filterMovesAndGenerate` for list of rule-based moves only
- If one move survives the filter, play it
- If all rule-based moves are filtered:
  - Choose (with filtering) among remaining moves
  - Implementation: see previous slide

MoGo Patterns - Idea

CMPUT 496



- Urgent response patterns
- Opponent just played move m
- Check empty points p near that move
- Up to 8 neighbors and diagonal neighbors
- Check 3×3 area around p
- Pattern decides: should the program play p with high priority?
- Point labeled with square in middle of pattern: urgent MoGo pattern move

Examples of Rules

CMPUT 496

- Atari Capture: Capture *last* opponent stone
- Atari Defense: Save nearby own stones from capture
- Capture *any* opponent stone
- Extend number of liberties
- Play “good shape” pattern
- Many more...

MoGo Patterns, Part 1

CMPUT 496

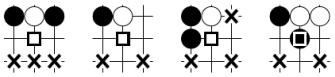


Fig. 5. Patterns for Hane. True is returned if any pattern is matched. In the right one, a square on a black stone means true is returned if and only if the eight positions around are matched and it is black to play.

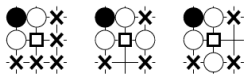


Fig. 6. Patterns for Cut1. The Cut1 Move Pattern consists of three patterns. True is returned when the first pattern is matched and the next two are not matched.

MoGo Patterns, Part 2

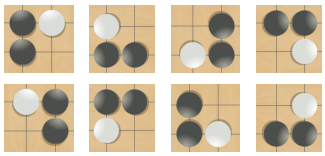
CMPUT 496

Fig. 7. Pattern for Cut2. True is returned when the 6 upper positions are matched and the 3 bottom positions are not white.

Fig. 8. Patterns for moves on the Go board side. True is returned if any pattern is matched. In the three right ones, a square on a black (resp. white) stone means true is returned if and only if the positions around are matched and it is black (resp. white) to play.

Dealing with Symmetries

CMPUT 496



- Total $2 \times 2 \times 2 \times 2 = 16$ possible combinations of operations
- Each choice gives a different symmetry of the same pattern
- Self-symmetric patterns have fewer cases
- Example: pattern can be equal to its flipped version

How were MoGo Patterns Found?

CMPUT 496

- Why these moves and not others?
- I don't know for sure
- Manual process, probably by trial and error
- Most of these moves are urgent or "stabilizing" local responses
- Soon we will study machine learning to replace the manual process

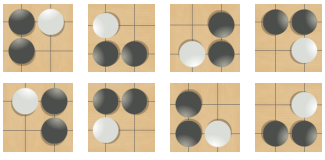
Programming MoGo-Style Patterns

CMPUT 496

- Implementation from open source program Michi
- Code in `Go3, util/pattern.py`
- Also see `https://github.com/pasky/michi`
- Table of patterns with different "wildcards"
- Wildcards match more than one state on board - among empty, black, white, off board
- Expand wildcards to all matching 3×3 patterns

Dealing with Symmetries

CMPUT 496



- Four operations can generate symmetrical patterns
- 2 rotations: 0 degrees, 90 degrees
- 2 vertical flips: don't flip, flip
- 2 horizontal flips: don't flip, flip
- 2 colors: don't swap colors, swap colors (not shown)

Sample Pattern Definitions

CMPUT 496

```
["XOX",  # hane pattern - enclosing hane
 "...",
 "???"],

["XO.",  # hane pattern - non-cutting hane
 "...",
 "?.?"],
```

Codes:

- `X` = black stone
- `O` = white stone
- `.` = empty
- `?` = any color

Expanding Wildcards Example

CMPUT 496

- Pattern with wildcard `?` matching any color

```
["?OX",  # side pattern - cut
 "X.O",
 "   "]
```

- Expanded - wildcard replaced by each possible color:

```
"XOX"          "OOX"          ".OX"
"X.O"          "X.O"          "X.O"
"   "          "   "          "   "
```

Expanding Wildcards - Details

CMPUT 496

- State of point:
  `X` = black stone, `O` = white stone,
  `.` = empty, " " = off board (for edge patterns)
- Wildcards can stand for more than one color
- `?` = "any color": `X`, `O`, `.`
- `x` = "not `X`": `O`, `.`
- `o` = "not `O`": `X`, `.`
- `pat3_expand`:
  - Expand all wildcards in a pattern
  - Create list of full patterns without wildcards
- Example on previous slide

MoGo Style Simulation Policy

CMPUT 496

- Use `generateMoveWithRulesAndFilter()` from a few slides ago, with pattern-based filter
- `ruleBasedGenerateMoves()` checks 3x3 patterns nearby
- `filter()` checks selfatari *for pattern moves only*

```
def ruleBasedGenerateMoves():
  patternMoves = []
  for p in empty 8-neighbors of last move:
      if 3x3-area around p matches a pattern:
          patternMoves.append(p)
  ...
```

`Go3` Player Revisited

CMPUT 496

- Default `Go3`: Run with `python3 Go3.py`
- Simple Monte Carlo Player
- Almost-random simulations
  - Filter eye-filling moves only
  - No selective rules
  - All legal moves (after filter) equally likely
- Optional arguments: next slide

Options in `Go3`

CMPUT 496

- Number of simulations/move (default 10)
  - Example: run with 100 simulations/move
  - `python3 Go3.py -sim 100`
- Move selection method (default simple)
  - Example: use UCB for move selection
  - `python3 Go3.py -moveselect ucb`
- Type of simulations (default random)
  - Example: use rule-based simulations
  - `python3 Go3.py -simulations rulebased`

Summary

CMPUT 496

- Pure random playouts have strengths, but also systematic weaknesses
- Simulations as evaluation: have bias and variance
- Try to reduce errors by using filters and rule-based move generation
- Examples of moves removed by filters:
  - eye-filling moves
  - selfatari
- Example of rule-based move generation
  - 3×3 pattern responses

Stochastic vs Deterministic Simulation Policies

Stochastic Simulation-based player. The simulation need to be stochastic, randomized move, simulations need to explore different move sequences.

The opposite of Stochastic is **deterministic** all simulations from the same start state plays the same sequence, therefore they have the same result. This is useless, all moves would have either a 0% or a 100% win rate.

Having variety in simulations is very important, it gives us more information about the huge state space. This is the main idea is sampling. We hope that errors in simulation "average out" through randomness. This is true if the simulations have no bias. The *variance* can be reduced by getting more samples. A deterministic policy will produce the same result over and over.

Rules-based to probabilistic simulation policies

we have seen two types of policies so far. The uniform random where all legal moves are equally likely to happen, and rule-based: all moves from a (short) list equally likely.
Now we introduce a third type of policy: probabilistic.

The reason why is because rule-based policies work but a bit crude. What if we want a better distribution over all moves? i.e., play pattern moves with some higher probability.

`prob_select.py`: Probabilistic Selection from a List

CMPUT 496

Statistical Analysis of Repeated Simulations

- Imagine a large table with a selection of different drinks
- There are more of some drinks than others
- Random experiment: waiter randomly grabs one drink
- Implementation in `prob_select.py`
- Given probability of selecting each drink
  - `drinks = [("Coffee", 0.3), ("Tea", 0.2), ("OJ", 0.4),...]`
- Repeat random experiment many times
- Measure drink selection frequency empirically

`prob_select.py` Sample Run

CMPUT 496

Statistical Analysis of Repeated Simulations

```
python3 prob_select.py
Experiment 0: OJ
Experiment 1: OJ
Experiment 2: Tea
Experiment 3: Coffee
Experiment 4: OJ
...
Experiment 99: OJ
Coffee probability 0.3, empirical frequency 0.26
Tea probability 0.2, empirical frequency 0.22
OJ probability 0.4, empirical frequency 0.4
Milk probability 0.07, empirical frequency 0.08
RootBeer probability 0.03, empirical frequency 0.04
```

Probabilistic Simulation Policy

CMPUT 496

Statistical Analysis of Repeated Simulations

- Same idea as in `prob_select.py`
- Used for one move decision step in a simulated game
- Given a game position in a simulated game
- Position has n legal moves
- Move i chosen with probability p_i
- Probabilities sum to 1: $\sum_{i=0}^{n-1} p_i = 1$
- Idea: heuristic to give (likely) better moves a higher chance of being played
- Can also use as a “soft” filter: give (likely) bad moves a low probability

Probabilities in Simulation Policies - So Far

CMPUT 496

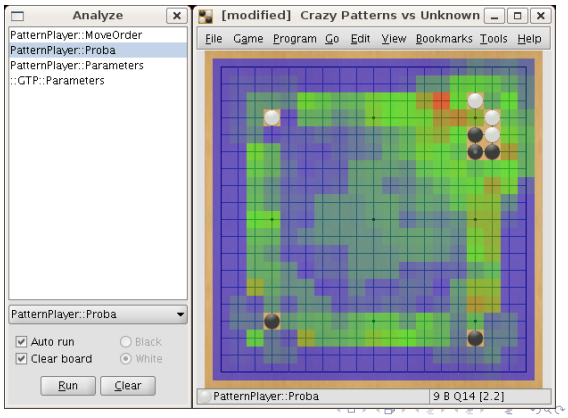
Statistical Analysis of Repeated Simulations

- So far we have seen two kinds of policies
- Both can be viewed as (simple) probabilistic policies
- Uniform random
  - n possible moves
  - Each chosen with probability $1/n$
- Rule-based policy
  - n possible moves
  - $m \leq n$ of them selected by a rule (e.g. patterns)
  - Each chosen with probability $1/m$
  - All $n - m$ other moves have probability 0

Coulom Move Patterns, https://www.remi-coulom.fr/Amsterdam2007/

CMPUT 496

Statistical Analysis of Repeated Simulations



General Probabilistic Simulation Policy

CMPUT 496

Statistical Analysis of Repeated Simulations

- n moves
- Move i has probability p_i
- $\sum_{i=0}^{n-1} p_i = 1$
- Give moves that "look strong" a higher p_i than others
- The paper by Remi Coulom that you are reading explains one way to come up with such probabilities
- It is based on learning knowledge from game records

Rule-based vs Probabilistic Policies

CMPUT 496

Statistical Analysis of Repeated Simulations

- Which is better, rule-based or probabilistic policy?
- No clear answer
- Rules are easier to code efficiently
- Probabilities are better suited for many machine learning methods

Visualizing Probabiliies

CMPUT 496

Statistical Analysis of Repeated Simulations

- Next slide shows "heat map"
- Moves on Go board encoded in different colors
- High probability moves in red/orange (probably good)
- Medium probability moves in green (probably mediocre)
- Low probability moves in blue (likely bad/meaningless)

Fuego: Mixing Rule-based and Probability-based Policies

CMPUT 496

Statistical Analysis of Repeated Simulations

- Our Go program `Fuego` uses both rules and probability
- Most of its simulation policy is rule-based as in `Go3`
  - About a dozen different rules and filters
  - AtariCapture, AtariDefend, LowLiberty, Patterns,...,Random
- Probabilistic selection:
  - For 3x3 patterns
  - Fuego uses a pre-computed table of probabilities for each pattern
  - More urgent pattern moves chosen more often

Summary of Simulation Policies

- Looked at three types of simulation policies
- Uniform random
- Simple rules and filters
- Probability-based
- Where do probabilities come from?
- Answer: from machine-learned knowledge

Statistical Analysis of repeated Simulations

Next topic: Better Top-Level Algorithm

CMPUT 496

Statistical Analysis of Repeated Simulations

- So far, we focused on better simulations
- Random, rulebased, probabilistic
- Next, we focus on top level algorithm in FlatMC
- So far: uniform move selection
- Use same number n of simulations to evaluate each move
- This is not smart!
- See example on next slide

Statistical Analysis of Repeated Simulations

CMPUT 496

Statistical Analysis of Repeated Simulations

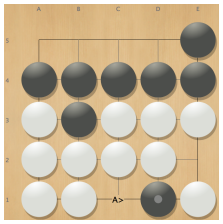
- We study some concepts from statistics
- Needed to understand the UCB algorithm for move selection
- Law of Large Numbers
- Bernoulli distribution
- Benefits and limits of doing more simulations
- More concepts: Binomial Distribution, confidence intervals, confidence level

Example - Winrates of FlatMC

CMPUT 496

Statistical Analysis of Repeated Simulations

Winrates with 10, 100 and 1000 simulations per move.



- 10 Simulations/move winrates: [**('c1', 1.0)**, ('b5', 0.6), ('a5', 0.5), ('c5', 0.5), ('d5', 0.5), ('Pass', 0.4), ('e2', 0.0)]
- 100 Simulations/move winrates: [**('c1', 1.0)**, ('b5', 0.63), ('Pass', 0.58), ('c5', 0.53), ('a5', 0.49), ('d5', 0.46), ('e2', 0.06)]
- 1000 Simulations/move winrates: [**('c1', 1.0)**, ('Pass', 0.572), ('b5', 0.565), ('d5', 0.524), ('a5', 0.523), ('c5', 0.484), ('e2', 0.087)]

Borel's Law of Large Numbers

CMPUT 496

Statistical Analysis of Repeated Simulations

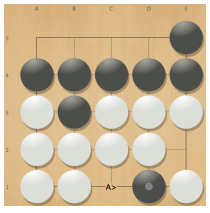
- There are several Laws of Large Numbers
  - A group of theorems in probability theory
  - General idea: repeating experiments many times will get results close to expectation
- Borel's law:
- An event E has probability p
- E occurs x times in n experiments
- As $n \to \infty$:
  $x/n \to p$
- Empirical frequency x/n approaches probability p
- Consequence: can *use* x/n to estimate an unknown p
- This estimate will be very rough when n is small
- Improves as n gets larger, and approaches true p

Example - Comparing Winrates

CMPUT 496

Statistical Analysis of Repeated Simulations

10 Sim:
c1: 1.0, e2: 0.0
100 Sim:
c1: 1.0, e2: 0.06
1000 Sim:
c1: 1.0, e2: 0.087



- Do we really need 1000 simulations to be convinced that c1 is better than e2? No.
- Smarter algorithm:
- *Explore* all moves in the beginning
- Focus much more on a few highest-percentage moves soon
- This leads to better decisions, less wasted time
- Example: the famous **UCB** algorithm

Bernoulli Distribution

CMPUT 496

Statistical Analysis of Repeated Simulations

- Bernoulli distribution (Jacob Bernoulli, 1655 - 1705)
- One of the simplest probability distributions
- Random variable X with two different values
  - 0 (loss) or 1 (win)
- Example: coin flip
- Example: win/loss outcome of a single simulation in Go
- Not a Bernoulli distribution:
  outcome of a simulation in TicTacToe
  Why not? three outcomes, win/loss/draw

Bernoulli Distribution (2)

CMPUT 496

Statistical Analysis of Repeated Simulations

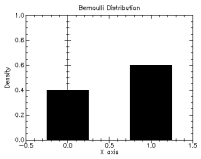


Image source: https://planet.racket-lang.org/package-source/williams/science.plt/3/1/planet-docs/science/random-distributions.html

- Given fixed probability p with $0 \leq p \leq 1$
- (Wikipedia says $0 < p < 1$, but in games p can be equal to 0 or 1)
- Probabilities for outcomes 1 and 0
  $Pr(X = 1) = p$
  $Pr(X = 0) = 1 - p$
- Sometimes, q is written for $1 - p$
- Example: $p = 0.6, q = 1 - p = 0.4$

Simulation is a Bernoulli Experiment

CMPUT 496

Statistical Analysis of Repeated Simulations

- Why is random sampling from a game tree a Bernoulli experiment?
- Proof sketch
- Finite tree, finitely many leaves, finitely many paths to leaves
- Each leaf has fixed value 0 or 1
- In each node, the simulation policy has a fixed distribution over its children
- We can compute the probability of choosing each path as the product of the probabilities of choosing each move on the path

Bernoulli Experiment

CMPUT 496

Statistical Analysis of Repeated Simulations

Image source: https://en.wikipedia.org/wiki/Coin_flipping

- Random experiment, typically repeated many times, same fixed p
- Each single experiment draws from Bernoulli distribution for p
- Example: coin flip with fair coin, $p = q = 0.5$
- Implementation: `bernoulli.py` - also see Activity

```
def bernoulli(p, limit):
    wins = 0
    for _ in range(limit):
        if random.random() < p:
            wins += 1
    return wins / limit
```

Simulation is a Bernoulli Experiment - Continued

CMPUT 496

Statistical Analysis of Repeated Simulations

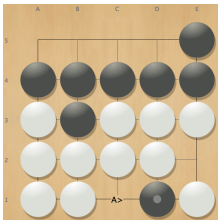
- The probability of choosing some specific path to a leaf is a (small) constant
- The winning probability at the root is just the probability of choosing a path leading to a win
- This is a sum of (a huge number of) constants, so is a constant p
- Each simulation is like a Bernoulli experiment with parameter p
- We win by choosing a winning path, which happens with probability p

Simulation-based Evaluation as Bernoulli Experiments - Example

CMPUT 496

Statistical Analysis of Repeated Simulations

Running a fixed simulation policy from a fixed Go position is a Bernoulli experiment



- Example: winrates after playing each move, with 10, 100, 1000 sim.
- For each move: converges to a fixed probability

Move	10	100	1000
c1	1.0	1.0	1.0
b5	0.6	0.63	0.565
a5	0.5	0.49	0.523
c5	0.5	0.53	0.484
d5	0.5	0.46	0.524
Pass	0.4	0.58	0.572
e2	0.0	0.06	0.087

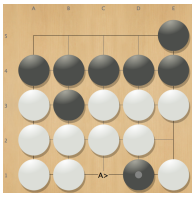
Analysis

CMPUT 496

Statistical Analysis of Repeated Simulations

- Analyse “flat” simulation player (1 ply tree)
- Runs n simulations on each child c of the root
- Focus on one child c now
- If we increase n, run more and more simulations, the winrate for c will stabilize
  - Reason: law of large numbers
- Limit, infinite number of simulations:
  - Winrate will converge to the “true winrate”
  - For one particular random simulation policy
  - For one particular start state
  - Winrate may be far from true minimax value
    - Reason: bias of simulation policy

Simple Move Selection is Inefficient



1000 Simulations/move winrates: [**('c1', 1.0)**, ('Pass', 0.572), ('b5', 0.565), ('d5', 0.524), ('a5', 0.523), ('c5', 0.484), ('e2', 0.087)]

- We really do not need 1000 simulations to figure out that e2 is bad
- Huge gap between winrates of bad move e2 and best move c1
- Very limited gain from running more and more simulations on worst moves
- Very inefficient use of time
- We need to *explore* all moves, but...
- We should *focus* most effort on the most likely good moves

Benefits of More Simulations

Benefits of running more simulations:

- Reduce variance
- Better selection when several moves are almost tied for first
- Rule out "unlucky" cases which occur with low number of simulations:
  - Bad move wins many simulations - estimated winrate too high
  - Good move loses many simulations - estimated winrate too low

`mystery-bernoulli.py`: Guessing the Winrate

- Activity: experiment with `mystery-bernoulli.py`
- Program generates a random `p`
- Runs a number of Bernoulli experiments
- Outputs the empirical winrate
- How well can you guess the true `p`?
- How does the number of simulations affect it?

Optimize What Simulations Tell Us

- Goal: a smarter way to decide:
- Which moves do we most need to evaluate better by running more simulations?
- Let's study the results of doing many simulations on one move
- The outcomes follow a *binomial distribution*

Optional Activity: Mystery game

- Program your own simulation-based game
- First, choose number of moves n
- Next, generate the true winrates p_i for each move i
- Next, ask the user for number of simulations/move
- Run that many simulations and collect empirical winrates (as in Go3)
- Print out the empirical winrates
- Let the user guess the best move
- Now, let the user know the true winrates and true best move
- Discuss: when is this game easy? When is it hard?

Binomial Distribution

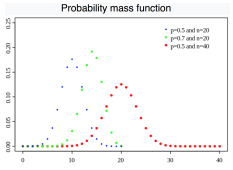


Image source:

`https://en.wikipedia.org/wiki/Binomial_distribution`

- Repeat the same Bernoulli experiment many times
- Number of wins is binomially distributed
- $B(n, p)$ = number of wins in n tries, where each has win probability p
- Expectated value of $B(n, p)$: np
- As n grows, distribution of $B(n, p)/n$ becomes more narrowly centered around p
- Probability of being far from p decreases as n grows

Bandit Algorithms and UCB - Motivation

CMPUT 496

Statistical Analysis of Repeated Simulations

- Consider top 2 moves from example
- After 10 simulations: **('c1', 1.0), ('b5', 0.6)**
- After 100 simulations: **('c1', 1.0), ('b5', 0.63)**
- How sure are we that c1 is better?
- We want to compare the two *true means* of c1 and b5
- We only have the *empirical means* for moves c1 and b5
- In theory, b5 (or another even lower-ranked move) could still be better
- It is extremely unlikely given the results so far - so we could ignore that move...
- How unlikely? We need some more statistics to answer that

Confidence Interval for repeated Bernoulli experiment

CMPUT 496

Statistical Analysis of Repeated Simulations

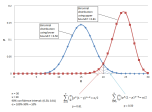


Image source:

http://www.biyee.net

- For a given confidence level, we can define an interval around the empirical mean that likely contains the true mean
  - Example: empirical mean 0.666..
- For lower confidence level, the intervals are smaller - more chance of error
- For higher confidence level, the intervals are larger - less chance of error
- Example: 90% confidence interval around 0.666 = (0.50, 0.81)
- For any value of p in $0.50 < p < 0.81$:
  - The empirical result 20/30 is within the "middle 90%" of outcomes

Confidence Interval

CMPUT 496

Statistical Analysis of Repeated Simulations

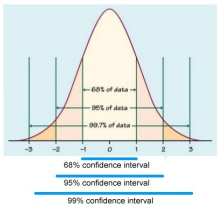


Image source:

https://www.quora.com

- Confidence Interval in statistics:
  - A range in which the true value is estimated to be
- Confidence level:
  - Probability that the range contains the true value

Back to Finding the Best Move

CMPUT 496

Statistical Analysis of Repeated Simulations

- Given different moves, each with empirical winrate
- We can compute confidence intervals for true mean of each move
- Goal: separate best move from all others
- Separation means:
- The whole confidence interval for the best move
- ... is above the intervals of all other moves
- In practice, that often takes far too long
- In the UCB algorithm, we use the upper confidence bound instead - the upper end of the confidence interval
- Details: next lecture

Confidence Interval for repeated Bernoulli experiment

CMPUT 496

Statistical Analysis of Repeated Simulations

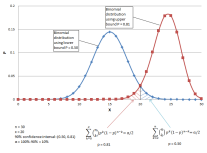


Image source:

http://www.biyee.net

- Repeated Bernoulli experiment with unknown win probability p
- Given empirical data Example: 20 wins in 30 tries
- From this, we need to estimate the unknown true mean p
- For any mean p, distribution of number of wins out of 30 experiments is the binomial distribution $B(30, p)$
- p is likely to be close to the empirical mean $20/30 = 0.66..$

Bandit problem

Bandit Problems

CMPUT 496

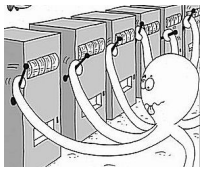


Image source: https://blogs.mathworks.com/loren

- Simulation-based players:
  - Run many simulations for each move as evaluation
  - Choose move with best winrate
- These decision problems are often called "bandit problems". Why?
- Multi-armed bandits (slot machines in Casino)
- Each bandit has an arm we can pull
- Which arm has the best payoff?
- To find out, need to play and estimate winrates

Wrong Choices and Regret

CMPUT 496

- Scenario: play each arm a number of time
- Pick arm based on results, e.g. best empirical winrates
- We will make mistakes since we make decisions based on random experiments
- How to measure mistakes?
- (At least) three popular ways
  - Probability of making wrong choice
  - Simple regret
  - Cumulative regret (used in UCB)

Types of Regret

CMPUT 496

- Regret: difference between expected value of best arm, and expected value of arm played
- Regret = 0 if you play a best arm
- Regret > 0 if you don't
- Cumulative regret: each arm pull costs money
- Simple regret: can try out arms for free. Measure only regret of final arm selection

Probability of making wrong choice, Simple Regret and Cumulative Regret

CMPUT 496

- Probability of making wrong choice
  - Arm i has best winrate p_i, but we choose arm j with $p_j < p_i$
  - What is the probability of that happening
- Simple regret
  - Evaluate how bad our move choice j is compared to best choice i
  - Simple regret is the difference $p_i - p_j$
  - Simple regret is 0 if we pick a best move, > 0 otherwise
  - Simple regret is higher if we pick a really bad move
- Cumulative regret
  - Regret $p_i - p_j$ for every pull of an arm j
  - Cumulative regret is the sum of all these regrets

Using Regret In Algorithms

CMPUT 496

- UCB is designed to minimize cumulative regret
- For simulations in games, simple regret would make more sense
  - Trying bad moves in simulation does not cost us anything
  - It is useful since it helps identifying a bad move
  - Only the final move decision is important
- Still, UCB-based algorithms work well
- Much current research on algorithms for simple regret

Example

CMPUT 496

- Three arms 1, 2, 3 with $p_1 = 0.8, p_2 = 0.5, p_3 = 0.1$
- Arm 1 is best (but we don't know that)
- We pull each arm once. Only arm 2 wins.
- We choose arm 2. Simple regret $p_1 - p_2 = 0.3$
- Cumulative regret 0 (pull arm 1) + 0.3 (pull arm 2) + 0.7 (pull arm 3)
- In terms of making the wrong choice, both arm 2 and arm 3 are equally bad
- For simple regret, it is important that we choose arm 1 **in the end**. But choosing arm 2 is still better than arm 3.
- For cumulative regret, it is important that we choose arm 1 most of the time **over the whole experiment**

Wrong Choice in Bandits

CMPUT 496

- Code in `binomial-select.py`
- How often do bandits based on Bernoulli experiments make the wrong choice?
- Code implements special case: only two arms, exact probability calculations
- Error probability depends on how many simulations we do
- More simulations give lower error prob.
- Result **strongly** depends on how close the two arms are in winrate
- See experiments in python code and `binomial-select-experiment.txt`

Error Rate - Theory vs Practice

CMPUT 496

- In practice, this exact error calculation is not used
- We don't know the true winrates
- It gets too complex with more than two arms or more simulations
- In most applications simple or cumulative regret is used instead

Notation for UCB Algorithm

CMPUT 496

- Goal: select best of k moves m_i, $0 \leq i \leq k-1$
- n_i: Number of times move i has been tried
- Total number of simulations so far: $N = \sum n_i$
- w_i: number of wins for move i among n_i tries
- Empirical winrate of move i: $\hat{\mu}_i = w_i/n_i$

UCB Algorithm

CMPUT 496

- Our simulation players so far used simple move selection strategy
- All first moves were simulated equally often
- We saw that this is wasteful
- UCB does better
- UCB allocates simulations to moves in a smart way
- It is designed to minimize *cumulative regret*
- UCB demo from `http://mdp.ai/ucb/`
- Written by UofA grad student Eugene Chen

UCB Formula

CMPUT 496

- UCB stands for upper confidence bound
- Define Upper Confidence Bound for move i by

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}}. \tag{1}$$

- C is the *exploration constant*
- Larger C: require higher confidence level

UCB Demo on `http://mdp.ai/ucb/`

CMPUT 496

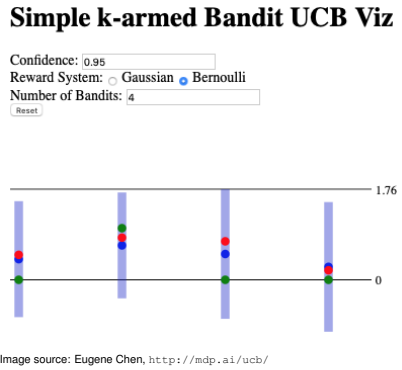


Image source: Eugene Chen, `http://mdp.ai/ucb/`

UCB Algorithm For Bandit Problems

CMPUT 496

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}} \tag{2}$$

$$move = \arg\max_{i \in \text{moves}} UCB(i) \tag{3}$$

- Loop:
  - Compute $UCB(i)$ for all moves i
  - Pick a move i for which $UCB(i)$ is largest
  - Run one Bernoulli experiment for move i
  - Increase w_i if the experiment was a win
  - Increase n_i and N
- At end: play the **most-pulled** arm

UCB Illustration

CMPUT 496

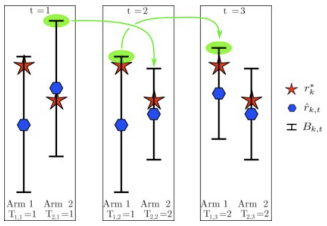
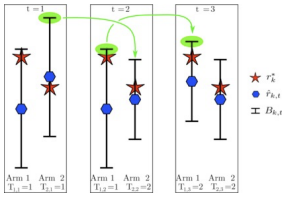


Image source: http://iopscience.iop.org/article/10.1088/1741-2560/10/1/016012

- Graphics show 3 steps in running UCB
- Red star: unknown true value
- Blue circle: empirical mean
- Black line: confidence interval
- Green: select arm with highest UCB

UCB Illustration Step 1

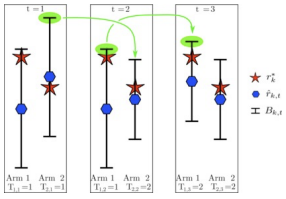
CMPUT 496



- Leftmost picture
- Arm 1 is best arm (highest true value = red star)
- Arm 1 was unlucky so far
- Its empirical mean is far below true mean
- Arm 2 has higher UCB (green)
- Step 1: select arm 2

UCB Illustration Step 2

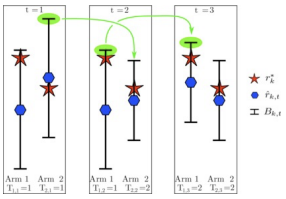
CMPUT 496



- Arm 2 was selected
  - Consequence: Confidence interval for arm 2 shrinks
- Arm 2 lost in the new simulation
  - Consequence: Mean of arm 2 drops
- Results shown in middle picture
- Both consequences lower the UCB of arm 2
- Arm 1 now has highest UCB
- Step 2: Arm 1 selected

UCB Illustration Step 3

CMPUT 496



- Rightmost picture
- Arm 1 was selected
  - Consequence: Confidence interval for arm 1 shrinks, its UCB drops
- Arm 1 won in the new simulation
  - Consequence: Mean of arm 1 increases, UCB increases more than the drop from shrinking interval
- Arm 1 remains best by UCB, gap larger than before

UCB Code Main Loop

CMPUT 496

- `stats[move][0]` = number of wins
- `stats[move][1]` = number of simulations

```
stats = [[0,0] for _ in range(arms)]
for n in range(maxSimulations):
    move = findBest(stats, C, n)
    if simulate(move):
        stats[move][0] += 1 # win
    stats[move][1] += 1
```

UCB Code `ucb` and `findBest`

CMPUT 496

```
def ucb(stats, C, i, n):
    if stats[i][1] == 0:
        return INFINITY
    return mean(stats, i)
         + C * sqrt(log(n) / stats[i][1])

def findBest(stats, C, n):
    best = -1
    bestScore = -INFINITY
    for i in range(len(stats)):
        score = ucb(stats, C, i, n)
        if score > bestScore:
            bestScore = score
            best = i
    return best
```

Three Details of UCB

CMPUT 496

- What if $n_i = 0$ at the beginning? Divide by zero problem
  - Answer 1: simulate each move once at the start, so $n_i = 1$
  - Answer 2: in my code I return a large constant INFINITY, so such moves will be chosen first
- How to choose exploration constant C?
  - In practice, we tune that constant for best results
  - Theory (later) shows us which choices are safe
- When does the loop end?
  - Can use fixed limit on total number of simulations
  - Can stop if one move is "clearly best", i.e. with high confidence

Optimism in the Face of Uncertainty

CMPUT 496

*Principle of **optimism in the face of uncertainty**: assume the best plausible outcome for each move*

- Using the upper confidence bound implements this principle in UCB
- Upper confidence bound represents the best plausible value of a move

UCB vs Simple Simulation Player

CMPUT 496

- UCB is much more efficient
- UCB will quickly focus almost all of its effort on small number of most promising moves
- UCB will never stop exploring other moves because of the log N term
- UCB will try the really bad-looking moves only very rarely

Exploration vs Exploitation Tradeoff

CMPUT 496

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}}. \quad (5)$$

- How to trade off between exploring and exploiting?
- Exploration constant C
- C small: focus on exploitation, $\hat{\mu}_i$ term is most important
- C large: focus on exploration, $1/n_i$ term is most important
- C very large: UCB becomes very similar to the simple uniform exploration strategy

Exploration vs Exploitation in UCB

CMPUT 496

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}}. \quad (4)$$

- Exploitation: $\hat{\mu}_i$. Prefer moves with high winrate
- Exploration: $1/n_i$ term. Prefer moves with large uncertainty, small n_i
- Exploration: $\log N$ term. Never stop exploring, try bad-looking moves again eventually

Code `ucb.py` and Examples

CMPUT 496

- `ucb.py` implements UCB algorithm and two examples
- Two cases
- Easy case: difference in arms quite large
- 10 arms, true winrates 0, 0.1,...,0.8, **0.9**
- Hard case: top two arms very close together
- payoff = [0.5, **0.61, 0.62**, 0.55]

UCB in Go3

CMPUT 496

- Switch on with command line option
- `moveselect=UCB`
- Select *average* number of simulations/move with `-sim`
- Example: 50 simulations/move
- Assume we have 20 legal moves in total
- `moveselect=simple` will run *exactly* 50 sim. on each move, total 1000 sim.
- `moveselect=UCB` will also run 1000 sim. in total
- It will choose the first move in each simulation by UCB
- Effect: much more focus on strongest moves
- You can change the exploration parameter C

A Small Scale Test of UCB in Go3

CMPUT 496

- Two versions of Go3 against each other
- `moveselect=simple` vs `moveselect=UCB`
- 5x5 board
- 50 simulations/move
- movefilter=false, simulations=random
- Win rate: 74% ($\pm$ 4.4) for UCB

Summary and Limitations of UCB

CMPUT 496

- UCB fixes an efficiency problem of the simulation player
- It does not waste much time on hopeless moves
- It does not fix any other problem of the simulation player
- It reaches the performance limits of simple simulation-based play more quickly
- Main limitation: still only 1 ply deep "tree search"
- Below that, still vulnerable to all biases in the simulation policy
- After move 1, still plays randomly for both opponent and player
- Only deeper tree search can fix that

Summary and Next Topics

CMPUT 496

Summary:

- Bandit problems
- From confidence bounds to UCB algorithm
- Strengths and limitations of UCB

Next Topics:

- High-level overview of search and simulation-based algorithms so far
- Selective search
- Monte Carlo Tree Search (MCTS) framework
- UCT Algorithm: Combines MCTS with UCB

Event tomorrow - A smart move: Artificial intelligence & strategy games

CMPUT 496

- Tomorrow 6:30pm, presentation of two new books on board games and programs
- Ryan Hayward: Hex, the Full Story
- Jonathan Schaeffer: Man vs Machine: Challenging Human Supremacy at Chess
- Panel discussion - Ryan, Jonathan, Michael Buro and me
- Free registration - see `https://www.ualberta.ca/science/events/smart-move-ai-strategy-games`

Exact Search, Selective Search, and Simulations

CMPUT 496

- Big-picture overview of algorithms so far
- For each method, focus on three questions:
  - Which parts of the game tree does it visit?
  - How does it back-up results to the root of the tree?
  - Exact or selective?

Review - (b,d) Tree Model and Solving a Game

CMPUT 496

- Search space in (b,d) tree model:
- Branching factor b, depth d
- Alternating min and max levels in tree
- b^d leaf nodes
- $(b^{d+1} - 1)/(b - 1) \approx b/(b - 1)b^d \approx b^d$ nodes in whole tree
- Size of proof tree in best case: very roughly $b^{d/2}$
- Minimum amount of search to solve a game

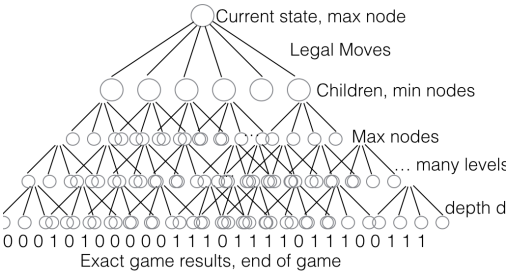
Naive Minimax (or Negamax) - Exact Solver

CMPUT 496

- Which parts of the game tree does it visit?
  - Explores the full game tree
  - All children searched in each node
- How does it back-up results to the root of the tree?
  - Minimax:
    Minimum over children at min nodes,
    maximum at max nodes
  - Negamax is a different but equivalent formulation, same result
- Exact or selective?
  - Exact
  - Terminal nodes are true end-of-game
  - Uses only exact scores at terminal nodes for evaluation
  - Result is proven correct

Naive Minimax - Exact Solver

CMPUT 496



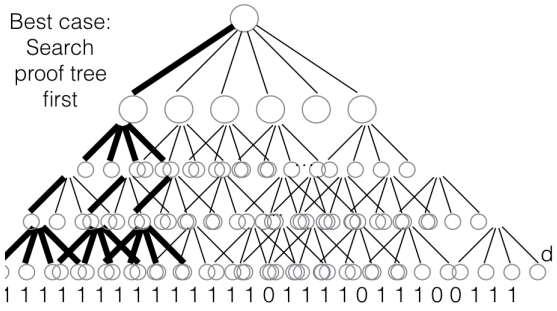
Efficient Minimax (or Negamax) - Boolean Minimax, Alphabeta

CMPUT 496

- Which parts of the game tree does it visit?
  - Some parts of tree may be cut by exact pruning rules
  - Best case: Visit only 1 child for winner
  - Needs to try all moves or loser
- How does it back-up results to the root of the tree?
  - Minimax
  - For alphabeta, some back-up values are "good-enough" upper or lower bounds, not exact values
- Exact or selective?
  - Exact.

Efficient Minimax (or Negamax) - Boolean Minimax, Alphabeta

CMPUT 496



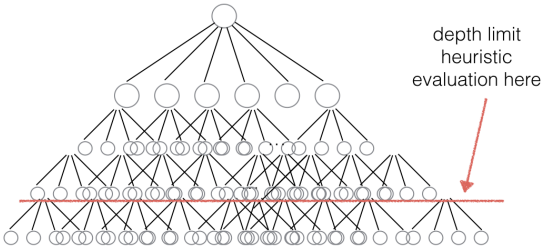
Depth-limited Alphabeta Search

CMPUT 496

- Which parts of the game tree does it visit?
  - As in alphabeta, but only up to depth limit
- How does it back-up results to the root of the tree?
  - Min and max
- Exact or selective?
  - Selective
  - Heuristic evaluation at terminal nodes
  - Search process is exact, but evaluation of leaves is not
  - Source of error: heuristic evaluation in leaf nodes

Depth-limited Alphabeta Search

CMPUT 496



Selective Alphabeta Search

CMPUT 496

- Which parts of the game tree does it visit?
  - Does not consider all legal moves in each node
  - Often depth-limited as well
- How does it back-up results to the root of the tree?
  - Min and max
- Exact or selective?
  - Selective
  - Heuristic evaluation at terminal nodes
  - Skips some legal moves
  - Source of error: heuristic evaluation in leaf nodes
  - Source of error: may prune the best move from a node

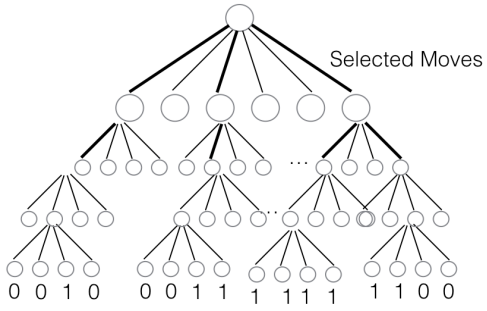
Selective Alphabeta Search with Fixed Time or Node Budget

CMPUT 496

- For many games even $O(b^{d/2})$ nodes for best-case proof is far too large
- In practice: fixed time or node limit for the search (e.g. 30 seconds, or 10^{12} nodes)
- What can we search within that budget?
- First answer was depth-limited search: reduce d until search fits within budget
- *New: Second answer - selective search*: reduce both b and d until search fits within budget

Selective Alphabeta Search

CMPUT 496



Selective Alphabeta Search - Methods

CMPUT 496

- How to do selective search?
- Search “interesting” moves much deeper than others
- Choice 1: Prune moves by using selective minimax algorithms such as ProbCut (Buro) or Nullmove pruning
- Choice 2: Prune moves using knowledge
  - Details: `https://chessprogramming.wikispaces.com/Selectivity`
- Choice 3: expand search tree selectively
  - Example: Monte Carlo Tree Search (MCTS)

Selective Alphabeta Search for Large Problems

CMPUT 496

- Large problems (chess, checkers, ...)
- Reducing b not enough
- Reduce both b and d - selective search with heuristic evaluation
- Before Monte Carlo, this was the standard approach for most of the complex games

Selective Alphabeta Search for Go?

CMPUT 496

How about Go?

- $>$ 200 moves on average for 19x19 Go
- Usually, only 1-10 of them are good
- Can we reduce *b* down to this range, without missing important good moves?
- Many attempts failed in the past - too many good moves missed
- MCTS was the first approach that worked well
- Later, strong move selection heuristics based on neural nets also helped a lot
  - Neural nets were not tried much with alphabeta, since MCTS worked so well in Go

Simulation-Based Player with Repeated Sampling

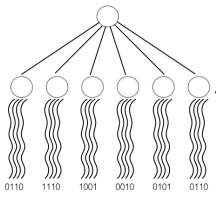
CMPUT 496

Repeated sampling in Simulation Player

- With small number of samples
  - Samples a few moves close to the start
  - does not improve the branching factor lower in the tree
- With large, unlimited number of samples
  - Eventually samples all nodes in the full (b,d) tree infinitely often
  - With selective policy (e.g. patterns, filters), samples some subtree infinitely often

Simulation-based Players

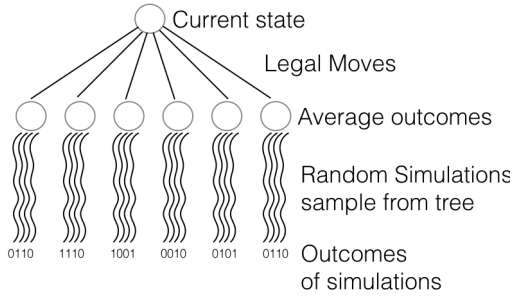
CMPUT 496



- Review: simple simulation-based players (e.g. Go3)
- 1 ply search at the root
- Move selection - simple or UCB
- Simulations - (almost) random, rule-based, or probabilistic
- How do these algorithms compare to selective search?

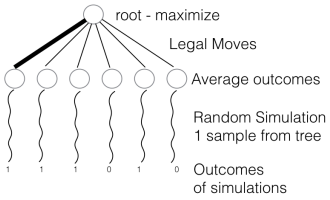
Simulation-based Player - Uniform Random Simulation Policy

CMPUT 496



Simulation-Based Player as Selective Minimax Search

CMPUT 496



- Extreme case of selective search:
- Simulation-based player with one simulation per move
- Branching factor *b* at root, complete search
- Branching factor 1 at all later levels...

Simulation-based Player - Uniform Random Simulation Policy

CMPUT 496

- Which parts of the game tree does it visit?
  - Eventually, visits all nodes
- How does it back-up results to the root of the tree?
  - Max at root only
  - Average over all simulations
- Exact or selective?
  - Selective
  - Not exact because of averaging instead of minimax
  - Source of error/risk: bias - average may be far from min, max
  - Source of error: variance - large uncertainty with small number of samples

Simulation-based Player - Non-Uniform Simulation Policy

CMPUT 496

- Which parts of the game tree does it visit?
  - All, except subtree below moves that are never selected by policy
- How does it back-up results to the root of the tree?
  - Same as Uniform Random: 1-ply max + average over simulations
- Exact or selective?
  - Selective
  - Similar to Uniform Random
  - Strength: average over better samples may be closer to min, max
  - Risk: can miss totally by hard-pruning all good moves

Monte Carlo Tree Search (MCTS)

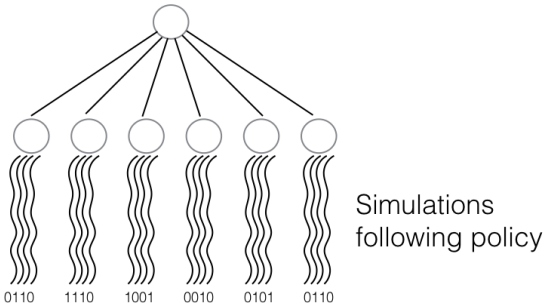
CMPUT 496

Next algorithm: Monte Carlo Tree Search (MCTS)

- Which parts of the game tree does it visit?
  - Tree search at the start, simulations to finish
- How does it back-up results to the root of the tree?
  - Weighted averages over children
  - Weight of child = number of simulations for that child
  - Approaches min, max if best child has much higher weight than rest
- Exact or selective?
  - Selective
  - Much deeper search for moves with better winrates
  - Converges to exact if given enough time to grow whole tree
  - Weighted average

Simulation-based Player - Non-Uniform Simulation Policy

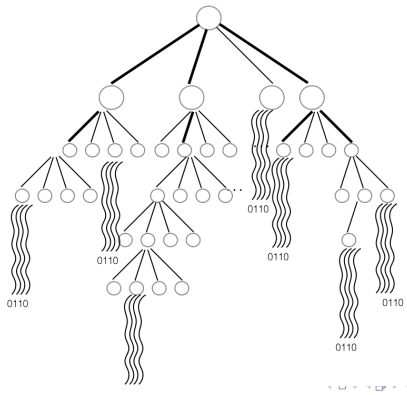
CMPUT 496



- Some nodes in tree may never be sampled:
- If some move on path to node never selected by policy

Monte Carlo Tree Search

CMPUT 496



Simulation-based Player - Simple vs UCB Move Selection

CMPUT 496

- Move selection:
  both simple and UCB behave the same in principle
- Both compute average over all simulations
- Difference: in UCB, average is taken:
  - Over more simulations for good move
  - Over fewer simulations for bad move
  - With a tree, in MCTS with UCT, this will be important
- Another difference:
  - UCB selects most-simulated move
  - Simple selects move with highest winrate
  - These moves are usually, but not always the same

Monte Carlo Tree Search

CMPUT 496

- Weakness of simulation-based players so far:
- No tree search after move 1
- Everything from move 2 is random(ized) simulations only

MCTS + UCT approach

- Add selective tree search
- Adapt UCB idea to work in trees - UCT algorithm
- UCT = Upper Confidence bounds on Trees
- Run simulation from leaf of tree for evaluation

Adding a Game Tree to Simulation-Based Player

CMPUT 496

- First idea: combine what we have:
  - Depth-limited alphabeta
  - Evaluation by simulation
- This fails miserably.
  - Too noisy - need many simulations to get reasonably stable evaluation
  - Too slow - even 1-ply simulation-based player is slow
  - Result: Simulation-based approaches were ignored for over 10 years in Go
- Smarter way to combine search and simulation
  - MCTS, UCT

Monte Carlo Tree Search(MCTS) Model

CMPUT 496

Algorithm 1 General MCTS approach.

```
function MCTSSEARCH(s_0)
    create root node v_0 with state s_0
    while within computational budget do
        v_l ← TREEPOLICY(v_0)
        Δ ← DEFAULTPOLICY(s(v_l))
        BACKUP(v_l, Δ)
    return a(BESTCHILD(v_0))
```

Image source: Browne et al, A Survey of Monte Carlo Tree Search Methods

- v_l = leaf node in tree
- Δ = result of simulation
- a(..) = action to move to best child

Monte Carlo Tree Search(MCTS) Model

CMPUT 496

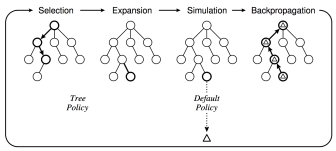


Fig. 2. One iteration of the general MCTS approach.

Image source: Browne et al, A Survey of Monte Carlo Tree Search Methods

Four steps, repeated many times

- Selection - traverse existing tree using formula such as UCT to select a child in each node
- Expansion: add node(s) to tree
- Simulation: follow randomized policy to end of game
- Backpropagation: update winrates along path to root

Monte Carlo Tree Search Example

CMPUT 496

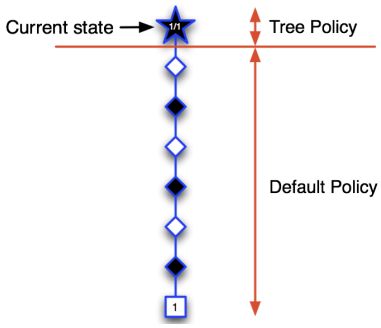


Image source: David Silver

Using MCTS to Play Games

CMPUT 496

- To play one move:
  - Run MCTS search from current state
  - After search: select best best move at root, play it
- To play a whole game:
  - Run MCTS every time it is the program's turn
  - May store and re-use parts of tree from previous search

Monte Carlo Tree Search Example

CMPUT 496

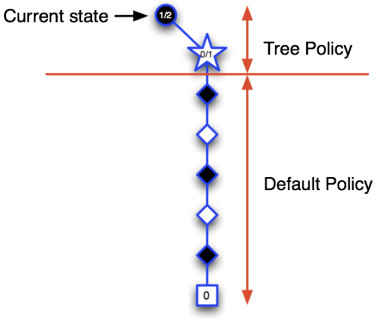


Image source: David Silver

Monte Carlo Tree Search Example

CMPUT 496

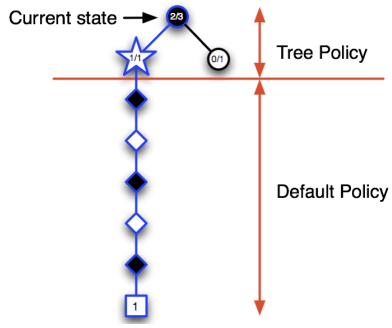


Image source: David Silver

MCTS Tree Traversal

CMPUT 496

- Start from root of tree
- Repeat:
  - Go to best child
  - Until reached leaf node in tree
- What is the best child?
- Use a formula to evaluate all children
  - UCT is popular (see next slides)
- Many other extended formulas are possible
  - Example: add kmowledge-based term

Monte Carlo Tree Search Example

CMPUT 496

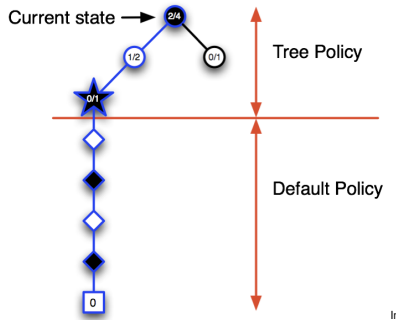


Image source: David Silver

From UCB to UCT

CMPUT 496

- UCT algorithm by Kocsis and Szepesvari (2006)
- It is still *the* classic algorithm for Monte Carlo Tree Search
- Not the first child selection algorithm use in MCTS
- The first based on sound theory
- Worked better in practice than earlier ad hoc algorithms
- Original paper has over 2200 citations - hugely influential

Monte Carlo Tree Search Example

CMPUT 496

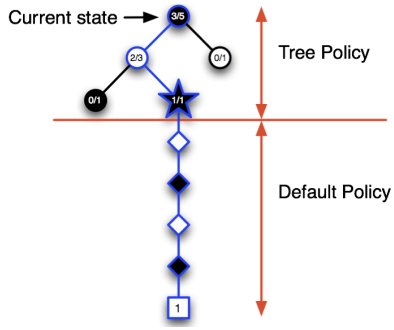


Image source: David Silver

UCT Algorithm Main Ideas

CMPUT 496

- Algorithm for child selection in Monte Carlo Tree Search
- Name UCT is often used for MCTS with this algorithm
- Combines tree search with simulations
- Uses results of simulations to guide growth of the game tree
- Uses UCB-like rule to select “best” child of a tree node
- Goal: select a good path in the tree to explore/exploit next
- Grows the tree over time
- Stores winrate statistics in each node, used for child selections

Exploration vs Exploitation

CMPUT 496

- Like UCB, UCT tries to balance Exploration and Exploitation
- Exploitation: focus on most promising moves
- Exploration: focus on moves where uncertainty about evaluation is high
- Difference: evaluate UCT formula in every node along a path in the search tree

From UCB to UCT

CMPUT 496

- Review - UCB formula

$$UCB(i) = \hat{\mu}_i + C\sqrt{\frac{\log N}{n_i}}. \quad (1)$$

- UCT is very similar: UCT value of move i from parent p:

$$UCT(i) = \hat{\mu}_i + C\sqrt{\frac{\log n_p}{n_i}}. \quad (2)$$

- Only difference in exploration term
  - UCB: uses global count of all simulations N
  - UCT: uses simulation count of parent n_p
- For root, UCT is identical to UCB
  - N = simulation count of root

MCTS Tree Expansion

CMPUT 496

- How to grow the tree?
- Simplest case: add one node per iteration
- Add one node from current simulation
- Tree grows very selectively
  - paths with strong moves become much deeper than others
- If memory fills too quickly:
  - Use an *expansion threshold* t_e
  - Only add a node if the leaf has at least t_e visits
  - Example: Fuego program, default $t_e = 3$

MCTS Simulations

CMPUT 496

- Run one simulation from the leaf node of tree
- Can use any simulation policy
  - Uniform random, rule-based, or probabilistic
- Result of simulation is win (1) or loss (0)
- Can run more than one simulation from each leaf node
  - Tradeoff between speed and accuracy
  - Tradeoff between time spent in updating tree vs running simulations
  - Example: for Fuego, on some hardware 2 simulations per leaf works better than 1

MCTS Backpropagation - Update Statistics

CMPUT 496

- Update wins and visit counts along path to root
- Negamax style implementation - flip wins/losses at each step
- `value = 1-value` changes from wins to losses and back

```
def backprop(node, value):
    while node:
        node._wins += value
        node._n_visits += 1
        value = 1 - value
        node = node._parent
```

MCTS Move Selection

CMPUT 496

- Run as many iterations of MCTS as you can
- Then select move to play at root
- How?
- Browne's paper mentions several approaches
- We discuss the main ones

MCTS Move Selection

CMPUT 496

- Max child: child with highest number of wins
- Robust child: Select the most visited root child. This is popular
- Highest winrate
  - Not a good/stable method with MCTS
  - Why not stable: see next slides
- Max-Robust child (see later slide)

Dangers of Selecting Move by Winrate in MCTS

CMPUT 496

- A 78 wins / 100 visits, winrate 78%
- B 8 wins / 10 visits, **winrate 80%**
- If we select B because of highest winrate:
- High risk of being wrong
- The value of A is much more certain
- The value of B still has much higher variance
- Remember discussion of binomial distribution of simulations
- Probability of error is high

Dangers of Selecting Move by Winrate in MCTS

CMPUT 496

- MCTS usually expands the move with best winrate (exploitation)
- But sometimes, it explores an inferior-looking move
- This can lead to trouble for selecting a move by best winrate
- A move with low simulation count and high uncertainty about its value might get selected
- See example next slide

Max-Robust Child: Extending Search

CMPUT 496

- What if most-simulated move and highest winrate move are different?
- Search may just have found a new best move
  - B is really better than A
- Or B may be a fluke
  - B just got some “lucky” wins, but is worse than A in the long run
- Very little evidence to decide which is true
- One solution: extend the search in such cases

Dangers of Selecting Move by Winrate in MCTS

CMPUT 496

- Example: two moves A and B
- A 78 wins / 100 visits, winrate 78%
- B 6 wins / 8 visits, current winrate 75%
- Assume B has higher UCT score, so we explore B
- B gets a win, now has 7 wins / 9 visits, current winrate 77.8%
- Explore B again
- B gets another win, now has 8 wins / 10 visits, current winrate 80%
- Assume we stop search now

Max-Robust Child: Extending Search

CMPUT 496

- Extending the search can distinuish two cases:
- If B is really good:
  - B will now receive many more simulations soon, stabilize value
- If B’s recent wins were a fluke:
  - Its winrate and upper confidence bound will drop quickly with more simulations
- Extending search in this way is called “Max-Robust child” in the paper

Improving MCTS

CMPUT 496

- Many ways to improve:
- Adding knowledge in tree or in simulation
- Modify in-tree selection
- Modify or replace simulations
- We will discuss several good options when we talk about machine learning and AlphaGo

Go5

CMPUT 496

- MCTS-based Go player
- Options similar to `Go4`, see `Go5.py`
- MCTS implementation in `Go5/mcts.py`

Summary

CMPUT 496

- Overview of game tree search and simulation
- Discussed Monte Carlo Tree Search
- After all the preparation, MCTS mostly combines previously discussed concepts
- 4+1 steps of MCTS
  - Repeat: select, expand, simulate, backpropagate
  - Finally: select move to play

Memory-Augmented Monte Carlo Tree Search

CMPUT 496

- Paper by Chenjun Xiao, Jincheng Mei and Martin Müller
- An improvement of MCTS
- Outstanding paper award at the 2018 AAAI conference
- Here: short, nontechnical summary of the ideas
  - Credit: most pictures and some bullet points taken from Chenjun's AAAI talk
- Interested in technical details?
  - Read the paper on our publications page `http://webdocs.cs.ualberta.ca/~mmueller/publications.html`
  - Look at the technical talk on our talks page `https://webdocs.cs.ualberta.ca/~mmueller/talks.html`

Go4

CMPUT 496

- Simulation-based player
- 1-ply search + simulations as in `Go3`
- Probabilistic simulation policy, similar to Lecture 14 and Coulom's paper
- It works but is very slow
- Experiment: empty 5 $\times$ 5 board, default settings, `genmove`
  - `Go3`, almost-random simulations: 1.8 seconds
  - `Go3`, rule-based simulations: 9.9 seconds
  - `Go4`, probabilistic simulations: 50 seconds

Main Idea

CMPUT 496

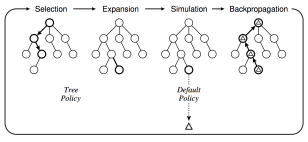


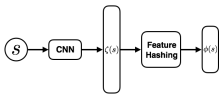
Fig. 2. One iteration of the general MCTS approach.

- Problem of MCTS:
- Most nodes are leaves or near leaf
- Most nodes have few simulations
- Evaluation is noisy

- Can we improve it?
- Approach: find similar states
- Use values of similar states to improve evaluation

Feature Representation for States

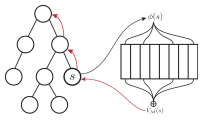
CMPUT 496



- How to define *similar* states?
- Represent state as vector of features
- States are similar if they share lots of features
- In this paper, features are defined by
  - Using a layer of a neural net
  - Using an unbiased hashing technique to reduce number of features

Using Memory with MCTS

CMPUT 496



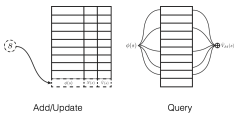
- Selection: compute state value by linear combination of state value $\hat{V}_s$ and memory value $\hat{V}_{\mathcal{M}}$

$$V(s) = (1 - \lambda_s)\hat{V}_s + \lambda_s \hat{V}_{\mathcal{M}}$$

- Evaluation: evaluate state by both Monte Carlo and memory
- Backup: update MC value and memory value in tree

Memory

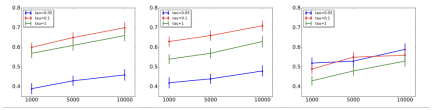
CMPUT 496



- Store for state s:
  - Feature vector of s
  - Pointer back to s to lookup its value (wins / visits)
  - As we do more search, value becomes better
- Lookup new state s:
  - Compute memory value as weighted sum of similar states in memory

Experiment 1

CMPUT 496



- Play games Fuego + M-MCTS against normal Fuego
- Vary neighbourhood size M
- τ is a "temperature" parameter in the algorithm
- X-axis: number of simulations/move
- Y-axis: winrate against Fuego

Finding Similar States in Memory

CMPUT 496

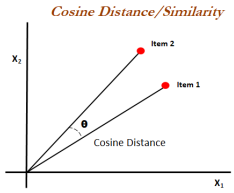
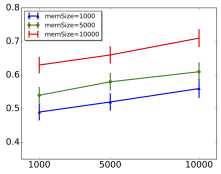


Image source: `https://www.safaribooksonline.com/library/view/statistics-for-machine`

- Compare two feature vectors
- Similar if they "point in similar direction"
- Measure: cosine similarity
- A standard similarity measure in machine learning
- Larger is better, similarity 1 if they have same direction
- Math: see `https://en.wikipedia.org/wiki/Cosine_similarity`

Experiment 2

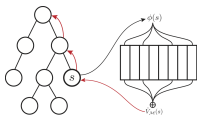
CMPUT 496



- Varying Memory Size
- Keep neighborhood size M and τ constant

Summary of M-MCTS

CMPUT 496



- MCTS has very few samples on most nodes near the leaves
- We can “interpolate” the value of similar nodes
- This gives a better evaluation
- Not in this summary (read the paper...):
- Math. framework and proof that this gives better values with high probability

Part IV

Machine Learning for Heuristic Search

Machine Learning for Heuristic Search - Introduction

CMPUT 496

- Main Concepts of machine learning (ML)
- Use of ML in heuristic search
- Learning as function approximation
- Overfitting problem
- Example: linear regression and least squares
- Representation and models for ML

Which Kinds of ML are Most Useful for Heuristic Search?

CMPUT 496

- **Learn an evaluation function**
- **Learn a move generation policy**
- Learn search control for the tree search
  - Which parts of a tree to search (first)?
- Learn a filter - which moves to cut
- Learn time control
  - How much time to use on each move?
  - Spend more time on more important decisions

What is Machine Learning?

CMPUT 496

- Wikipedia: Give computers the ability to learn without being explicitly programmed.
- Domingos article: Algorithms that figure out how to perform important tasks by generalizing from examples.
- Amii: Machine learning enables a computer system to independently learn from, and continuously adapt to, data without being explicitly programmed for that data.

Simplest Learning - Remembering Facts

CMPUT 496

- Rote learning in humans: remember facts
- Not an issue in computers - databases can store massive amounts of facts
- This is (mostly) a solved problem
- Cache and re-use results of previous computations

Examples of Machine Learning Applications

CMPUT 496

- Ad placement on web pages
- Spam filters for email
- High-frequency trading
- Image, speech and text recognition
- Robotics
- Games
- Thousands of other applications, increasing rapidly

Remembering Facts - Examples in Heuristic Search and Learning

CMPUT 496

- Transposition table
- Compute and store winning strategy for whole game, then follow it
  - Endgame database, as in checkers
- Tabular learning - simple kind of reinforcement learning
  - Learn a table of all states and their expected reward

Learning as Function Approximation and Generalisation

CMPUT 496

Large, complex problems: cannot learn all the facts

- What can we learn?
- Learn abstract concepts
- Learn general knowledge from (many) examples
- Find interesting trends, correlations in your data
- Learn evaluation function
- Predict moves in the game of Go

Learning with a Model in Games

CMPUT 496

- In games we have fast and perfect simulators -
- Truthful: exact implementation of rules and actions
- Fast, efficient
- Cheap, can repeat as often as needed
- Games are ideal test beds for learning
- Used for much groundbreaking work on learning

Supervised vs Unsupervised Learning

CMPUT 496

- Supervised: learn from *labeled* training data - labeled with the correct result
  - Example: learn from master moves in game records. Label of position = move played by the master
- Unsupervised: unlabeled training data, learn from *reward*, which is often delayed
  - Example: learn from playing games. Reward = win/loss at the end of the game
- Semi-supervised learning:
  - Some (often small) amount of labeled data
  - Some (large) amount of unlabeled data
  - Not discussed further in this course

Input, Output, and Representation

CMPUT 496

- Key questions before we start a machine leaning task:
- What is the input? How is it represented?
- What is the output? How is it represented?
- What is learned? How is that represented?
- We get to choose representations.
- Good choices can make a huge difference in the results

Learning with a Model vs Learning from Experience

CMPUT 496

- Learning with a Model: allows us to try out actions and observe results in simulator
- Learning from Experience: real world, learn from observations of the effects of real actions
- Simulator: learn from simulated experience from experiments within the simulator
- Most domains:
  - Must worry about the error from modelling
  - Garbage in - garbage out

Learn a Function from Data

CMPUT 496

- Given training examples: data points (x_i, y_i)
- Learn a function $y = f(x)$
- Goal: approximate the data as well as possible
- Use function for *prediction*:
  - Given a new x-value
  - What should the y-value be?

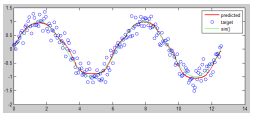


Image source: http://stackoverflow.com/questions/1565115/

Goals:

- Good approximation
- Robust against noisy data
- Avoid overfitting

Training Data vs Test Data

CMPUT 496

- Training data
  - Data used for the machine learning process
- Test data
  - Data used to evaluate the quality of the learned function
- Training data and test data should be *independent* samples from the same learning problem
- One rule of thumb:
  split data into 90% training, 10% test
- Example: given 100,000 master-level Go games:
  - Use 90,000 games for training
  - 10,000 games for test

Error from Overfitting

CMPUT 496

Image source: https://en.wikipedia.org/wiki/Overfitting

- x-axis = training time
  y-axis = error of the learned function
- Blue line: error on training data
- Red line: error on test data
- In the beginning, learning from training data works well on the test data
- At some point, it begins to overfit to the training data
- The generalization performance on the test data becomes worse

Overfitting

CMPUT 496

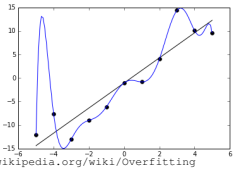


Image source: https://en.wikipedia.org/wiki/Overfitting

- Problem: learning the noise as well as the data
- If you fit the noisy data too closely,
  it will not generalize well to new data
- Example: fit exact polynomial through noisy data points
- Depending on the noise level in the data,
  a simple linear function may be better
- Opposite problem: underfitting - cannot see the regularity in the data from the learned function

Linear Regression

CMPUT 496

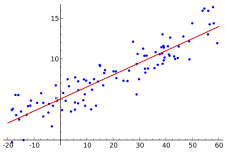


Image source:

https://en.wikipedia.org/wiki/

https://en.wikipedia.org/wiki/

Regression_analysis

- Find best linear function
  $y = ax + b$ to approximate the data
- Only two parameters a, b to optimize
- Standard approach to solve:
  least squares

More Overfitting

CMPUT 496

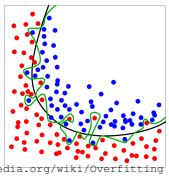


Image source: https://en.wikipedia.org/wiki/Overfitting

- Complicated green line:
  separates red from blue data points exactly
- What if there is some noise in the data?
  The green line overfits to the outlier data points
- Likely the simpler black line is the better separator
- More examples in the Domingos paper

Least Squares

CMPUT 496

- Given data points (x_i, y_i)
- Find a linear function $y = ax + b$ which approximates the data as well as possible
- Error for point i = difference between y_i and $ax_i + b$
- $|y_i - (ax_i + b)|$
- Difficult to optimize with absolute values, so *squared error* is much more popular
- Find a, b which minimize $\sum_i (y_i - (ax_i + b))^2$

Least Squares (2)

CMPUT 496

- Find a, b which minimize $\sum_i (y_i - (ax_i + b))^2$
- Closed-form solution using simple calculus
  - Best values for a and b computed as functions of x_i and y_i
- Many python sample codes on the net, e.g. `http://machinelearningmastery.com/implement-simple-linear-regression-scratch-`
- Details: `https://en.wikipedia.org/wiki/Simple_linear_regression`

Representation and Models for Machine Learning

CMPUT 496

- Main questions:
- How is the input represented?
- **How is the learned model represented?**
- What is the format of the output?

Generalizations

CMPUT 496

- Linear regression is the most basic model
- Many more general statistical models exist
- Fit nonlinear functions, e.g. polynomials
- **Linear functions of more than one variable**
  - Examples in next class
- **Nonlinear functions of more than one variable**
  - Example: neural networks
- Minimize error functions other than least squares

Representation of the Input

CMPUT 496

- Raw input
  - Location of stones on the board, or sequence of moves
- *Features* which represent (hopefully) useful concepts that will facilitate learning
  - Go examples: selfatari, liberties, proximity to last move, local pattern
  - Simplest case: *binary* features, only two values 0 (off, false) and 1 (on, true)
- Popular for learning in heuristic search:
  - Machine-learned weights
  - Hand-designed features
  - Example: Coulom's MM

Example for Different Error Function

CMPUT 496

- Move prediction problem in Go
- Learning task: Learn a value for how good each move is
- Pick the move with highest value
- We do not care about the function itself
- We only care about the move ordering it produces
- We do not directly have a target function to approximate
- Just count the number of correctly predicted moves
- Evaluate the set of function values of all legal moves:
  - Value 1 if master move has highest evaluation among all moves, 0 otherwise
  - This is a *classifier* as discussed in Domingos' article
  - "is best move" vs "is not best move"

Representation of the Model

CMPUT 496

- What kind of functions are we trying to learn?
- It is our choice, we can use some general principles
- Principle 1: simple is good - helps avoid overfitting
- Principle 2: as complex as needed to represent what we need
  - Example: Linear functions are not enough to give a good evaluation for Go
- Principle 3: functions which represent general assumptions about our world
  - See detailed discussion in Domingos' paper
- Principle 4: functions for which we have efficient learning algorithms
- Choice is closely tied to choice of input representation

Examples of Models

CMPUT 496

- Linear model
  - Features f_i given as input
  - Learn weights w_i
  - Evaluation: $\sum_i w_i f_i$
- Neural network
  - Simple features or raw data as input
  - (Many) layers of neurons and nonlinear *activation functions*
  - Weights represent strength of connection between neurons

Developing the Model

CMPUT 496

In practice, we do (many) iterations of:

- Create data
- Develop model
- Use machine learning to learn weights
- Evaluate model
- Find problems or weaknesses with data and/or model
- Repeat

Output of Learning Process

CMPUT 496

Result of learning in games:

- Classifiers: good/bad move, filter/don't filter for search,...
- Move evaluation or move probabilities
- State, position evaluation
- Local evaluation, e.g. territory maps
  - How likely is point p going to be Black/White/neutral?

Machine Learning for Heuristic Search

Machine Learning with Simple Features

CMPUT 496

- Review - Simple features in Go
- Implementation in `Go4` and `Go5`
- Evaluation with Features
- Learning feature weights
- `Go4` used features in simulation policy

Basic Features

CMPUT 496

```
FeBasicFeatures = {
    "FE_PASS_NEW": 0,
    "FE_PASS_CONSECUTIVE": 1,
    "FE_CAPTURE": 2,
    "FE_ATARI_KO": 3,
    "FE_ATARI_OTHER": 4,
    "FE_SELF_ATARI": 5,
    "FE_LINE_1": 6,
    "FE_LINE_2": 7,
    "FE_LINE_3": 8,
    "FE_DIST_PREV_2": 9,
    "FE_DIST_PREV_3": 10,
    ...
    "FE_DIST_PREV_9": 16,
    "FE_DIST_PREV_OWN_0": 17,
    "FE_DIST_PREV_OWN_2": 18,
    ...
    "FE_DIST_PREV_OWN_9": 25
}
```

Simple Features

CMPUT 496

- We discussed simple features in Lecture 11 as an example of knowledge
- We also saw simple features in Remi Coulom's paper
- Here: review with focus on implementation in `Go4` and `Go5`
- Feature: boolean-valued statement about a move
- Fixed set of features f_i
  - $f_i = 1$ means feature i is true for a move - **active feature**
  - $f_i = 0$ - feature i is false for a move - inactive
- Describe each move by its feature vector $F = (f_i)$
  - Example: `(0,0,1,1,0,1,0,0,0,1,...)`
- Alternative: list of indices of active features
  - `(2, 3, 5, 9,...)`

Distance Features

CMPUT 496

- Measure distance between two points on board
- Points $(x1, y1)$ and $(x2, y2)$
- $dx = |x1 - x2|, \quad dy = |y1 - y2|$
- Distance metric $d(dx, dy) = dx + dy + \max(dx, dy)$
- Example:
  - Points (3,5) and (4,3)
  - $dx = 1, \quad dy = 2$
  - $d(dx, dy) = 1 + 2 + \max(1, 2) = 5$

Simple Features Implementation in `Go4` and `Go5`

CMPUT 496

- Implementation in `go4/feature.py`
- 26 basic features, plus about 950 small pattern features
- Similar to features in Coulom's paper and in our Fuego program
- Each legal move has a small set of active features
- Features form groups of *mutually exclusive* features
  - In each group, at most one feature is active
  - Example: area around each move matches exactly one of the about 950 patterns
  - All the other pattern features are inactive, do not match

Distance Metric Discussion

CMPUT 496

- Distance metric $d(dx, dy) = dx + dy + \max(dx, dy)$
- Why not just use Manhattan or Euclidean distance?
- This metric is more fine-grained than Manhattan
- Can distinguish more cases
  - Example: (2,1) and (3,0) have different distances from (0,0)
  - $d(2, 1) = 5$, $d(3, 0) = 6$
- This metric is integer-valued, easier to use then Euclidean
  - Example: Euclidean distance
  - $d(2, 1) = sqrt(5) = 2.236...$

Types of Distance Features

CMPUT 496

- Feature group: Distance to previous stone (last move by opponent)
  - FE_DIST_PREV_2 .. FE_DIST_PREV_9
- Feature group: Distance to previous own stone (our move before that)
  - FE_DIST_PREV_OWN_0, FE_DIST_PREV_OWN_2, FE_DIST_PREV_OWN_9
  - FE_DIST_PREV_OWN_0: play again at same point after opponent's capture
- Feature group: Line on the board (counting from edge)
  - Line 1, or Line 2, or Line 3 ...
  - FE_LINE_1, FE_LINE_2, FE_LINE_3

Evaluation Function from Simple Features

CMPUT 496

- Evaluate one move m
- Which features f_i are *active* for m?
- About 1000 features
- Only about 5-10 are active for any given move
- Different moves have different active features
- Simplest evaluation function: linear combination
- Learn a weight w_i for each feature
- eval(m) = $\sum w_i f_i$

Pass and Tactics

CMPUT 496

- Feature group: pass move
  - FE_PASS_NEW: previous move was not a pass
  - FE_PASS_CONSECUTIVE: previous move was also a pass
- Feature group: atari move
  - FE_ATARI_KO, FE_ATARI_OTHER
- Other simple tactics (not a group, not mutually exclusive)
  - FE_CAPTURE
  - FE_SELF_ATARI

Evaluation Function (2)

CMPUT 496

- This is a sum of about 1000 terms
- Most terms are 0
- Only need to sum the active features
- $\sum w_i f_i = \sum_{f_i=1} w_i$
- Example: $f_0 = 0, f_1 = 1, f_2 = 0, f_3 = 0, f_4 = 1$
- eval(m) = $0 \times w_0 + 1 \times w_1 + 0 \times w_2 + 0 \times w_3 + 1 \times w_4 = w_1 + w_4$
- Compare: in Coulom's approach, evaluation is the product of active feature weights
- eval(m) = $\prod_{f_i=1} w_i$

Pattern features

CMPUT 496

- Feature group: 3×3 area centered on candidate move
- Move can also be on edge of board
- About 950 different cases
  - By far the biggest feature group in `Go4`
  - Implementation from `michi` program: see `go4/pattern.py`
  - Review discussion of patterns in Lecture 13

Move Prediction using Features

CMPUT 496

- What is move prediction?
  - Predict which move a master player would choose in a given position
  - Example of supervised learning - position is labeled by the master move
- Why move prediction?
  - Use for move ordering in search
  - Use for better moves in simulation policies (`Go4` policy)

Fast vs Slow Move Prediction

CMPUT 496

- Fast: use simple features
- Slow: use deep neural network
- Tradeoffs:
  - Deep neural networks are much better move predictors
  - Simple features are several orders of magnitude faster, especially on normal CPU without custom hardware

Data for 7 × 7 Go Move Prediction

CMPUT 496

- For `Go4`, we learned simple features for a 7 × 7 board
- No human master games available on this small board
- We created thousands of training games by self-play using the strong program Fuego
  - First 5 moves of game were chosen at random ...
  - ... to ensure diversity of training data
  - Only learned from the remaining moves in each game

Overview of the Feature Learning Process

CMPUT 496

- Collect training/test data
  - Game records with master moves
- Label each move in each position by its features
- Run an algorithm to learn feature weights
  - Example:
    Coulom's Minorization/Maximization algorithm
- Use the learned weights as knowledge in your program to select good moves

Getting Features from Game Data

CMPUT 496

Process for 19 × 19 Go:

- Foreach game g (tens of thousands of games)
- Foreach position p in game g (≈150-300 per game)
- Foreach legal move m in position p (≈20-362 per position)
- One data point: all the active features for this move
- One of these moves is $\mathbf{m}^*$, the move played by the master

Game Data for 19 × 19 Go Move Prediction

CMPUT 496

- Which data to learn from?
  - Games between professional players
    - Can get about 100,000 games
  - Games between amateur players
    - Can get around 1 million games
  - Games between computer programs
    - Unlimited, if enough time/hardware to generate them
- For learning simple concepts,
  more variety/weaker players may be better
- One option: learn only from stronger player/winner

Move Prediction as Classification Problem

CMPUT 496

Classification problem:

- Compute score for each legal move
- Two classes of moves:
- class 1 = {highest scoring move}
- class 2 = {all other moves}
- When is classification problem solved?
- When score of m^* is highest

Coulom's Feature Learning and Minorization/Maximization Algorithm

CMPUT 496

- Paper by Remi Coulom, Computing Elo Ratings of Move Patterns in the Game of Go
- You already read it for the "knowledge" topic
- Now we discuss the machine learning part
- Main topics:
  - Represent move as group of active features
  - Bradley-Terry model to evaluate strength of a group of features
  - Minorization-Maximization algorithm to learn weight for each feature
  - How to use in Go program

Feature Strength and Bradley-Terry Model

CMPUT 496

- Each individual feature f_i has a strength
  - We call it the weight w_i
  - In the paper it is called Gamma value, γ_i.
  - Larger weight means better feature
- How do two features compare: probabilistic model
- P(feature f_i beats f_j) $= \frac{w_i}{w_i + w_j}$

Represent Move as Group of Features

CMPUT 496

- For each move, about 10 features are active (less for the simple features in Go4)
- In learning, we represent each move *only* by its group of features
- Learning objective:
- Group of features representing the master move...
  ... **is stronger than**...
  ... Feature group representing any other legal move

Example

CMPUT 496

- f_1 = capture, $w_1 = 30.68$
- f_2 = extension, $w_2 = 11.37$
- P(capture beats extension) = $30.68/(30.68 + 11.37) \approx 0.73$
- P(extension beats capture) = $11.37/(30.68 + 11.37) \approx 0.27$
- f_3 = distance 5 to previous move, $w_3 = 1.58$
- P(capture beats distance 5...) = $30.68/(30.68 + 1.58) \approx 0.95$

Main Advantage of Learning with Features

CMPUT 496

- Tabular learning of moves for full states:
  - Just memorizes which particular moves were good in particular positions
  - No generalization
- Learning with features:
  - Learn which features are generally good or bad
  - Learn which features work in many examples
  - This approach provides *generalization* to new positions, not seen before
  - Much more useful in practice, each new game has different positions

From Single Features to Groups - Generalized Bradley-Terry Model

CMPUT 496

- A move has more than 1 feature (about 5-10 is typical)
- How to combine them?
- Generalized Bradley-Terry model: multiply them
- Example: move m has active features f_2, f_5 and f_6
- strength(m) = $w_2 \times w_5 \times w_6$

Comparing Two Moves

CMPUT 496

- To compare moves, we estimate their win probabilities as before.
- P(move m_1 beats move m_2) =

$$\frac{\text{strength}(m_1)}{\text{strength}(m_1) + \text{strength}(m_2)} \quad (1)$$

- Example:
  - m_1 has features f_1, f_2, strength $w_1 \times w_2$
  - m_2 has features f_2, f_5, f_6, strength $w_2 \times w_5 \times w_6$
  - P(m_1 beats m_2) =

$$\frac{w_1 \times w_2}{w_1 \times w_2 + w_2 \times w_5 \times w_6} \quad (2)$$

Minorization-Maximization (MM) Algorithm

CMPUT 496

- Problem: it is difficult to maximize L directly
- Approach: find a simpler formula m which minorizes L:
  - m approximates L
  - $m(x) < L(x)$
- We can directly compute the maximum of m with respect to each weight w_i

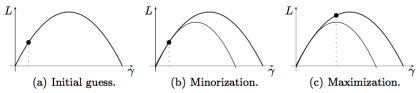


Fig. 1. Minorization-maximization.

Comparing Multiple Moves

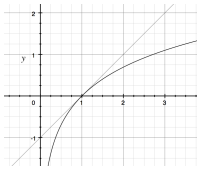
CMPUT 496

- Similarly, we can compare all legal moves in a Go position
- P(move m_i wins) $= \frac{\text{strength}(m_i)}{\sum_{j \in \text{legalmoves}} \text{strength}(m_j)}$
- Assumptions:
  - Strength can be measured on a linear scale
    - Not true for rock-paper-scissors like scenarios, A beats B beats C beats A
  - Strength of combination of features can be measured by product
    - Not clear why it should be true in general
    - Not true if features are strongly dependent
  - Strong assumptions, but it seems to work anyway...

Minorization-Maximization (MM) Algorithm

CMPUT 496

- Idea: $-\log L$ is a sum of simpler log terms
- Can approximate log function:
- For x close to 1, $\log x \approx x - 1$
- Also, $\log x \leq x - 1$, so $1 - x \leq -\log x$
- $1 - x$ minorizes $-\log x$



Learning Weights with Generalized Bradley-Terry Model

CMPUT 496

- Goal: find weights w_i for all features...
- ...such that probability of playing the master moves is maximized
- Maximize $L = \prod_{j=1}^{N} P(R_j)$
- Where $P(R_j)$ is probability of playing master move in test case j
- $P(R_j)$ can be expressed as a function of the weights w_i (details in paper)

Minorization-Maximization Iteration

CMPUT 496

- Start with some weights settings, e.g. $w_i = 1$ for all i
- Do one step of MM for each weight w_i
- This brings us closer to the maximum of L
- Repeat the process from here
- Each repetition brings closer approximation
- Remi's C++ implementation of MM: `https://www.remi-coulom.fr/Amsterdam2007/`

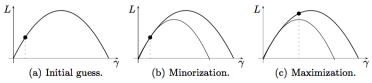


Fig. 1. Minorization-maximization.

Review - Summary of the Learning Process

CMPUT 496

- Collect training data (game records with master moves)
- Label each move in each position by its features
- Run MM to compute feature weights
- Use the weights as knowledge in your program to select good moves

Extensions to the MM Model (1) - LFR

CMPUT 496

- Wistuba et al (2013) Latent Factor Ranking (LFR) algorithm
- Main idea: take *interactions* of features into account
- Two features may *reinforce* or *cancel* each other's effects
- Taking the sum $w_1 + w_2$ of feature weights does not work well in such cases
- Learn *interaction terms* as well as individual feature weights

How to use the Learned Model

CMPUT 496

Two main applications

- In-tree knowledge for better move selection during MCTS
  - Discussed last lecture
  - Three ideas:
  - Node initialization, additive knowledge, multiplicative knowledge
- Better probabilistic simulation policies
  - Lecture 14, Go4 program

LFR Continued

CMPUT 496

- Problem: for n features there are
  - $\binom{n}{2}$ pairwise interactions
  - $\binom{n}{3}$ interactions of three features
  - $\binom{n}{k}$ interactions of k features
- Example: $n = 2000$, $\binom{n}{2} \approx 2000000$, $\binom{n}{3} > 1.3$ billion
- Solution: develop smart algorithm to learn only the most important interactions
- Achieves better move prediction than MM

From Move Weights to Move Probabilies

CMPUT 496

- Some applications require probabilities, not just weights
  - Probabilistic simulation policies
  - Multiplicative in-tree knowledge
- Now we finally have a way to learn such probabilities
- Idea: run MM to learn feature weights w_i
- Compute the strength of each move as product of its features' weights
- Choose each move with probability proportional to its strength

Extensions to the MM Model (2) - FBT

CMPUT 496

- Factorization Bradley-Terry (FBT) model (Xiao 2016)
- Problem with LFR algorithm:
- The weights it computes are "just numbers"
- Larger weights are better, but...
- ... no interpretation as probabilities
- Harder to use in a program than MM weights
- FBT adds interaction terms in a probabilistic model
- Achieves better move prediction than MM and LFR
- Your TA Chenjun Xiao's MSc thesis and AAAI publication

Limits of Learning from Game Records

CMPUT 496

- First main limit:
  - Can only learn what is in the data
  - New situation may require different moves not seen before
- Second main limit:
  - Can only learn what can be represented in our model
  - Simple features cannot represent high-level concepts
  - Neural nets are **much** more powerful
- Important question for any learning algorithm:
  - How well can it pick up the knowledge that is "hidden" in the data and transfer it into a learned model?

Limits of Move Prediction

CMPUT 496

- Can never reach 100% prediction
- Two main reasons
  - Multiple equally good moves
  - Different definitions of "best" move

Move Prediction - What to Expect?

CMPUT 496

Prediction of master moves in Go

- What is a good prediction score?
- Random prediction on 19×19: under 0.5%
- Simple features and algorithms (Go4, MM): maybe 20%
- Better features and algorithms (Fuego, FBT): 30-40%
- Human amateur master players: 40-50%
- AlphaGo neural net: 57%
- Professional human players: similar to AlphaGo?

Equally Good Moves

CMPUT 496

- Reasons why moves are equally good:
- Symmetry, e.g. in opening
- Same point value in endgame
  - Example: there may be five 2-point moves in the endgame
  - No reason to prefer one over the other
  - Even a perfect player has only a 20% chance in move prediction
- Forcing moves:
  - Opponent must answer such moves
  - Can often be played at different times without changing the result
  - Hard to predict when exactly a master will play it
- Moves may have different strong and weak features which balance each other
  - Choice is "matter of taste", playing style

Strong Move Prediction vs Playing Well

CMPUT 496

- A better move predictor does not necessarily make a better player
- Most Go games have some very specific, complex tactics
  - Often not covered by general learned knowledge
- Playing moves that are good "on average" may fail in such situations
- Need precise "reading" (lookahead, search)
- Move prediction can help focus the search
- It cannot find all good moves by itself
- This is still very much true in AlphaGo

Different Definitions of "Best" Move

CMPUT 496

- I think I am winning. What is the best move?
- In theory, any move which preserves a win (follows a winning strategy) is equally good
- In practice, neither me nor my opponent are perfect players
- One answer: maximize my probability of winning
- What does it mean? It depends on modeling myself and my opponent
  - Example: in TicTacToe, simulation player was better than perfect player against random opponent
- I think I'm losing. How do I best trick the opponent into a mistake?

Summary

CMPUT 496

- Discussed learning with simple features
- Coulom's approach:
- Generalized Bradley-Terry model for strength of moves
- MM algorithm for learning weights
- Use as in-tree knowledge or as simulation policy

Decay Knowledge over Time

CMPUT 496

- At the beginning, we have only few simulations
  - Win rate is very noisy
  - Knowledge may be more reliable, can help to guide search
- Later, we may have many simulations for a node
  - We should trust them more now
  - All knowledge is heuristic, may be wrong
  - Slowly phase out knowledge as more simulations accumulate

Using Knowledge in UCT

CMPUT 496

- Regular UCT: select best child by UCT formula
- UCT value of move i from parent p:

$$UCT(i) = \hat{\mu}_i + C\sqrt{\frac{\log n_p}{n_i}}$$

- This uses only information from simulations
  - Empirical winrate $\hat{\mu}_i$, number of simulations n_i, number of simulations for parent n_p
- We can improve move selection by using learned knowledge
  - Examples: simple features, neural networks
- Idea: give good moves a bonus before simulations start

Initialization of Node Statistics

CMPUT 496

- Normal UCT: count number of simulations and wins
- Initialize to 0
  - For all children i
  - Wins $w_i = 0$
  - Simulations $n_i = 0$
- We can initialize with other values to encode knowledge about moves
  - Give good moves some initial wins
  - Give bad moves some initial losses

How to Use Knowledge

CMPUT 496

Three ways

- Initialization of node statistics
- Additive knowledge term
- Multiplicative knowledge term

Initialization of Node Statistics (2)

CMPUT 496

- How to initialize n_i and w_i ?
- Size of n_i expresses how reliable the knowledge is
- Winrate w_i/n_i expresses how good or bad the move is, according to the knowledge
- Original work by Gelly and Silver (2007): knowledge worth up to 50 simulations
- Fuego program: simple feature knowledge converted into winrate/simulations
- Decay over time: yes
  - Over time, real simulation statistics dominate over initialization

Additive Knowledge

CMPUT 496

- Idea: add a term to UCT formula

$$UCT(i) = \hat{\mu}_i + \textbf{knowledgeValue}(i) + C\sqrt{\frac{\log n_p}{n_i}}$$

- `knowledgeValue(i)` computed e.g. from simple features or neural network
- Must scale it relative to other terms by tuning
  - Too small: little influence on search
  - Too big: too greedy, ignores winrate
- Decay over time: must be explicitly programmed
- Multiply knowledge term by some *decay factor*
  - Examples: $1/n_i$, $1/(n_i+1)$, $\sqrt{1/n_i}$,...

Multiplicative Knowledge, Probabilistic UCT (PUCT)

CMPUT 496

- Idea: explore promising moves more
- Knowledge used:
  - Probability p_i that move i is best
- Multiply exploration term by p_i

$$PUCT(i) = \hat{\mu}_i + \mathbf{p_i} \times C\sqrt{\frac{\log n_p}{n_i}}$$

- Decay over time: yes
  - Divide by n_i in the exploration term
- Exploration term smaller than before, because $p_i \leq 1$
  - May need to balance by increasing C
- AlphaGo: exploration term $p_i \times C/(n_i+1)$

Summary of Knowledge in UCT

CMPUT 496

- Knowledge can be used in an in-tree selection formula
- Independent from using knowledge during the simulation phase
- Can be (much) slower, used only in tree nodes, not in each simulation step
- Different approaches have been tried successfully
  - Initialization of node statistics by knowledge
  - Additive term
  - Multiplicative term, PUCT

CMPUT 496

- Introduction to Neural Networks (NN)
- Neural networks in biology
- Artificial neural networks in computing science
- NN structure and relation to biological neurons
- Small Python examples and demo of learning
- Neural networks as function approximators
- Learning weights for NN - Backpropagation

Neural Networks

CMPUT 496

- A neural network in Computing Science is a *function*
- It takes input(s) and produces output(s)
- It has many parameters (weights) which are determined by learning (training)
- Deep neural networks can approximate (almost) any function in practice
- Training NN:
  - Supervised learning
  - Reinforcement learning

Neural networks in Biology - Neurons

CMPUT 496

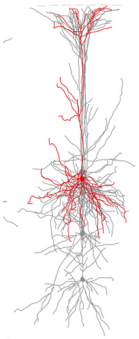


Image source:

`http://www.frontiersin.org/files/Articles/62984/`

- Neuron = nerve cell
- Found in:
  - Central nervous system (brain and spinal cord)
  - Peripheral nervous system (nerves connecting to limbs and organs)
- Involved in all sensing, movement, and information processing (thinking, reflexes)
- Very complex systems, function is still only partially understood

Neural networks in Biology - Neurons

CMPUT 496

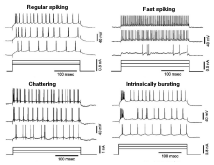


Image source:
http://ecee.colorado.edu/~ecen4831/cnsweb/cns0.html

- Neurons transmit information through electrical and chemical signals
- Transmission through synapses - connections between two neurons
- Complex behaviors:
  - In time
  - In space

Neural Networks (NN) in Computing Science

CMPUT 496

- Massively simplified, abstract model
- Used as a powerful function approximator for (almost) arbitrary functions
- We now have effective learning algorithms even for very large and deep networks
- Single (artificial) neuron: implements a simple mathematical function from its inputs to its output
- Connections between neurons:
  - Each connection has a *weight*
  - Expresses the strength of the connection

Synapses

CMPUT 496

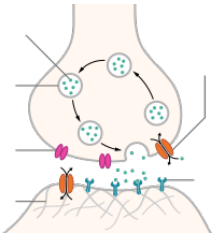


Image source:
https://en.wikipedia.org/wiki/Synapse

- Small (atom-scale) gap between neurons
- Information transmitted via
  - Chemicals (neurotransmitters, main mechanism)
  - Electric currents (faster)
- Human brain - about 150 trillion (1.5×10^{14}) synapses
- Some neurons have up to 100000 synapses

Components of a NN - Input, Output and Hidden Layers

CMPUT 496

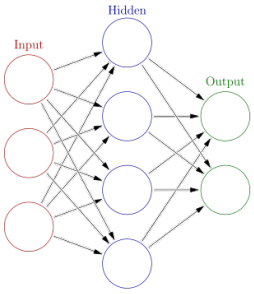


Image source: https://en.wikipedia.org/wiki/Artificial_neural_network

- Organized in layers of neurons
- Each layer is connected to the next
- Input layer
- One or more hidden layers
- Output layer
- Shallow vs Deep NN
  Main difference:
  Number of hidden layers

Neuron Count in Humans and Animals

CMPUT 496

- Elephant 251 billion
- **Human 86 billion**
- Gorilla 33 billion
- Baboon 14 billion
- Raven 2.2 billion
- Cat 760 million
- Rat 200 million
- Frog 16 million
- Cockroach 1 million
- Fruit fly 250,000
- Ant 250,000
- Jellyfish 5600
- Worm 300
- Sponge 0

Source: https://en.wikipedia.org/wiki/List_of_animals_by_number_of_neurons

(Optional, interesting story) Counting elephant neurons:
http://nautil.us/issue/35/boundaries/the-paradox-of-the-elephant-brain

How does a Neuron Work?

CMPUT 496

- Inputs $x_1 \ldots x_m$ from m neurons on previous layer
- Extra constant input $x_0 = 1$
- Each input x_i has a weight w_i
- Weighted sum of inputs $\sum_{i=0}^{m} w_i x_i$
- Nonlinear activation function (or transfer function) ϕ
- Output $y = \phi(\sum_{i=0}^{m} w_i x_i)$
- Output used as input for neurons on next layer

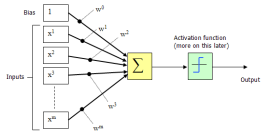


Image source: https://www.codeproject.com/KB/AI/NeuralNetwork_1/nn2.png

Supervised Training of a Network - Overview

CMPUT 496

- View the whole network as a function $y = f(x)$
- Both x and y are vectors of numbers
- Train by supervised learning from set of data (x_j, y_j)
- Compute errors - differences between y_j and $f(x_j)$
- Compute how error depends on each weight w_i in network
- Gradient descent - adjust weights w_i in network to reduce these errors
- Example now, details later

Properties of Sigmoid Function

CMPUT 496

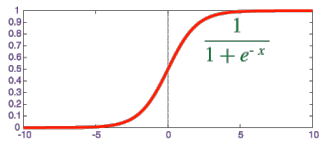


Image source: `https://qph.ec.quoracdn.net`

- x large negative number: e^{-x} very large, $\sigma(x)$ close to 0
- x large positive number: e^{-x} very small, $\sigma(x)$ close to 1
- $x = 0$: $\sigma(x) = 1/2$
- Nice property of $\sigma(x)$: derivative

$$\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$$

Software: NN Toy Examples in Python

CMPUT 496

- First example: `nn.py` in `python/code`
- Adapted from article at `http://iamtrask.github.io/2015/07/12/basic-python-network`
- 1 input layer, 1 hidden layer, 1 output node
- 3 input nodes - Each input x_i consists of three values
- Training data: 4 examples
- Input: 4 rows, 1 for each x_i, $i = 0, 1, 2, 3$
- Sigmoid activation function (see next slide)
- Output vector with 4 numbers y_i
- Demo now.

Backpropagation and Training - Error

CMPUT 496

- Same basic ideas as learning with simple features
- Let f be the function computed by the net
- Result of f depends on
  - input vector x
  - all weights w_j
- Output $y = f(x, w_0, ...w_n)$
- Error on data point (x_i, y_i):
  - Difference between $f(x_i)$ and y_i
  - Usual measure - squared error $(y_i - f(x_i))^2$
- Goal: minimize sum of square errors over training data
- Error $E = \sum_i (y_i - f(x_i))^2$

Sigmoid Function

CMPUT 496

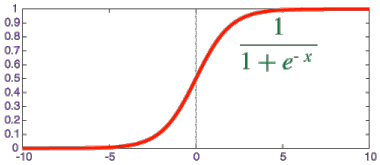


Image source: `https://qph.ec.quoracdn.net`

- Nonlinear function, popular for activation function
- Smoothly grows from 0 to 1
- Definition:

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

Backpropagation Concepts

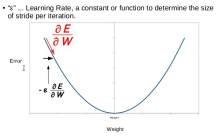
CMPUT 496

- How to reduce error?
- The only thing we can change are the weights w_i
- How does error E depend on all the weights?
- Simpler question: how does error E depend on a single weight w_i?
- Should we increase w_i, decrease it, or leave it the same?
- The *partial derivative* of E with respect to w_i gives the answer

$$\frac{\partial E}{\partial w_i}$$

Partial Derivative - Intuition

CMPUT 496



- Meaning of $\frac{\partial E}{\partial w_i}$
- Make a small change of w_i
- How does it affect the error E?
- Which change will *reduce* the error?
- Look at sign of derivative

- $\frac{\partial E}{\partial w_i} > 0$ - Small **decrease** in w_i will decrease E
- $\frac{\partial E}{\partial w_i} = 0$ - Small change in w_i will have no effect on E
- $\frac{\partial E}{\partial w_i} < 0$ - Small **increase** in w_i will decrease E

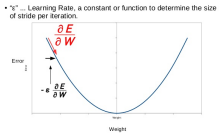
Chain Rule

CMPUT 496

- $z = f(x)$, $y = g(z) = g(f(x))$
- Then
$$\frac{\partial y}{\partial x} = \frac{\partial y}{\partial z} \times \frac{\partial z}{\partial x}$$
- Example:
- Neuron input
$z = \sum_{i=0}^{m} w_i x_i$
- Sigmoid activation function
$y = \sigma(z) = \sigma(\sum_{i=0}^{m} w_i x_i)$
- How does output y depend on some weight, say w_1?

Partial Derivative and Rate of Change

CMPUT 496



- Error E is a function of
all inputs x, all outputs y and all weights w
- Partial derivative quantifies the effect of
leaving everything else constant
and making a small change ϵ to w_i
- $E(\cdots, w_i + \epsilon, \cdots) \approx E(\cdots, w_i, \cdots) + \frac{\partial E}{\partial w_i}\epsilon$

Chain Rule Example Continued

CMPUT 496

- Example - compute derivative of y with respect to w_1, $\frac{\partial y}{\partial w_1}$
- By chain rule, $\frac{\partial y}{\partial w_1} = \frac{\partial y}{\partial z} \times \frac{\partial z}{\partial w_1}$
- First, derivative of z with respect to w_1, $\frac{\partial z}{\partial w_1}$
  - z is just a linear function of w_1
  - $z = w_1 x_1 +$ (terms that do not depend on w_1)
  - $\frac{\partial z}{\partial w_1} = x_1$
- Now, $\frac{\partial y}{\partial z} = \frac{\partial \sigma(z)}{\partial z}$
- Remember $\frac{d\sigma(x)}{dx} = \sigma(x)(1 - \sigma(x))$
- So $\frac{\partial y}{\partial z} = \sigma(z)(1 - \sigma(z))$
- Result: $\frac{\partial y}{\partial w_1} = \sigma(z)(1 - \sigma(z)) \times x_1 = y(1 - y)x_1$
- Final result is simple, easy to compute
- In practice, packages such as TensorFlow can do all of the math automatically

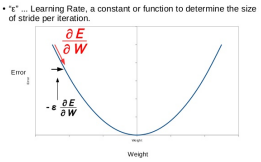
Derivative and Chain Rule

CMPUT 496

- How does the error E change if we change *any* single weight in the net?
- We can break down the computation layer by layer
- The error function is a simple function of the output
- The output is the result from the last layer in the net
- Each node implements a simple function of its inputs
- The inputs are again simple functions of the previous layer, etc.
- We can break down the computation of $\frac{\partial E}{\partial w_i}$ into a neuron-by-neuron computation using the chain rule

Backpropagation (Backprop) Step

CMPUT 496



- Apply chain rule to compute how changes to weights reduce error
- Go some distance ϵ along the *gradient* of E with respect to weights
- $w_i = w_i - \epsilon \frac{\partial E}{\partial w_i}$
- Choice of *step size* ϵ is important
- Go too far - overshoot the minimum
- Go too little - very slow improvement of E

Backprop Algorithms

CMPUT 496

- Developed starting in the 1960's
- Main ideas
- Define step size ϵ
- Compute backprop step for *all* weights
- Repeat until error on test set does not improve
- Huge number of variations of backprop algorithms
  - Momentum, adaptive step size, stochastic vs batch data, ...

Neural Networks as Universal Approximators

CMPUT 496

- NN with at least one hidden layer can *approximate* any *continuous* function arbitrarily well, given enough neurons in the hidden layer
- Given a continuous function $f(x)$
- Consider $f(x)$ in the range $0 \leq x \leq 1$
- Given an arbitrarily small $\epsilon > 0$
- Theorem (Cybenko 1989)
  There exists a 1-hidden-layer NN $g(x)$ such that

$$|f(x) - g(x)| < \epsilon \quad \text{for all} \quad 0 \leq x \leq 1$$

Network Types

CMPUT 496

- Feed-forward NN (all our examples)
  - Information flows in one direction from input to output
- Recurrent NN (RNN)
  - Directed cycles in the network
  - Popular in natural language processing, speech and handwriting recognition
  - Example of very successful deep RNN architecture: LSTM, "Long short-term memory"
    - Can be trained by backprop, like our feed-forward nets
- Autoencoder - learn representation for data with unsupervised learning
- Hundreds of other NN types, new ones each month

NN as Universal Approximators (2)

CMPUT 496

- How is that possible?
- Intuitively, it works by:
  - Having lots of neurons in the hidden layer
  - Two neurons together can approximate a *step function*
  - Their sum is very close to $f(x)$ in a tiny interval
  - Their sum is almost 0 everywhere else
- Now demo from
  `http://neuralnetworksanddeeplearning.com/chap4.html`
- Note: constant b in demo is what we called w_0

Building a Neural Network

CMPUT 496

- Important Questions
- How many layers?
- How to connect the layers
- How many neurons in each layer?
- What kind of functions can we represent in principle?
- What kind of functions can we learn efficiently?

NN as Universal Approximators (3)

CMPUT 496

Comments:

- The theorem does *not* mean that any network can approximate any function arbitrarily well (see `nn.py` counterexample last class)
- The theorem says that by *adding* more and more hidden neurons, we can make the error smaller and smaller
- The theorem is only about *continuous* function. But we can also approximate functions with discontinuous jumps pretty well (e.g. as in the demo)

NN as Universal Approximators (4)

CMPUT 496

More comments:

- Why are we using multilayer "deep" networks if 1 hidden layer is enough in theory?
- Short answers:
  - Efficiency of learning
  - Size of representation
- Details: `http://neuralnetworksanddeeplearning.com/chap5.html`

Summary

CMPUT 496

- Introduced neural networks
- Backprop algorithm
- Examples of networks
- Next time: convolutional networks, deep networks
- Move prediction in Go with deep convolutional networks

Network Architecture - fully connected

CMPUT 496

- Review - usually, connections are only from one layer to the next
- Some recent success with adding connections to layers "further up" (not discussed here)
- Simplest architecture: *fully connected*
  - Each neuron on layer n connected to each neuron on layer $n+1$

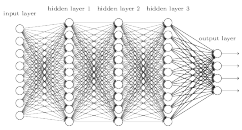


Image source: `http://neuralnetworksanddeeplearning.com/chap6.html`

Sparse Network Architectures

CMPUT 496

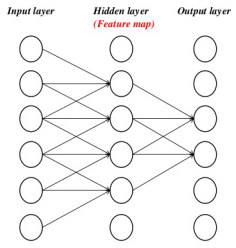


Image source: `https://www.slideshare.net/SeongwonHwang/presentations`

- Opposite of fully connected: *sparse*
- Neuron connected to only *some* neurons on next layer
- Important case for us: *Convolutional* NN (next lecture)

Convolutional Neural Networks

Convolutional Neural Networks

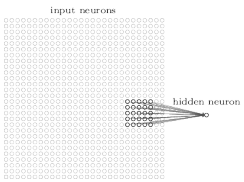


Image source: `http://neuralnetworksanddeeplearning.com/chap6.html`

- Hugely successful in image recognition tasks
- Later applied to move prediction in Go
- Main ideas:
  - start learning locally
  - Higher layers can learn successively more globally
- Each neuron depends only on a small, closely related group of neurons on the previous layer
  - "Local receptive field"

Feature Maps

- Consequence of weight sharing:
- Each layer can only learn a single feature
- To learn more, we use several copies of the layer
- Each copy is called a *feature map*
- Example in picture: 20 different features for image recognition task learned in first hidden layer

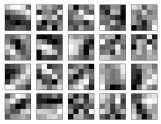


Image source: `http://neuralnetworksanddeeplearning.com/chap6.html`

Local Receptive Field (or Filter)

- Typical sizes 5×5, 3×3
- Compare with 3×3 simple pattern features in Go
- First hidden layer: represent many simple *local features*
- Main difference to learning with simple features:
  - The network itself decides what the features are!

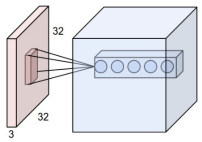


Image source: `http://cs231n.github.io/convolutional-networks`

Deep Convolutional NN Concepts

- Repeat many layers of local filters, processing pipeline
- Higher-level features built from simpler local features
- Pooling: reduce from e.g. 2×2 region to single neuron by averaging
- Padding: add artificial outside elements to prevent shrinking
- Often: fully connected layers at the end of pipeline

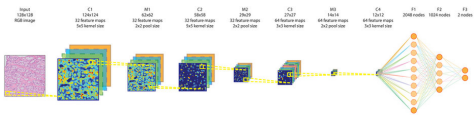


Image source: `http://www.nature.com/articles/srep26286/figures/1`

Weight Sharing

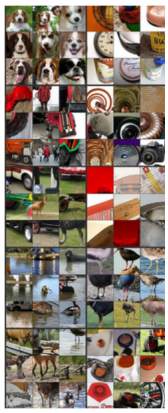
- For both image recognition and in Go:
- Want to learn a universally useful set of local features
- Idea: share weights across *all neurons*
- Each neuron on the same level computes the same function...
- ...but with different input variables (different receptive fields)
- Same idea in simple feature learning
  - Learned one weight for one 3×3 pattern
  - Same weight used everywhere on the board

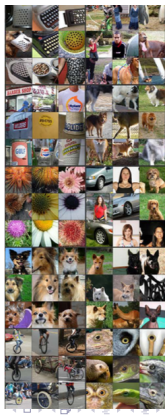
What Do the Learned Features Mean?

- Short answer: usually, we don't know
- But they work!!!
- Long answer: "it's an active area of research"
- Some successes with interpreting features from image recognition tasks
- Images on next few slides:
  Zeiler and Fergus (2013),
  Visualizing and Understanding Convolutional Networks,
  `https://arxiv.org/pdf/1311.2901.pdf`

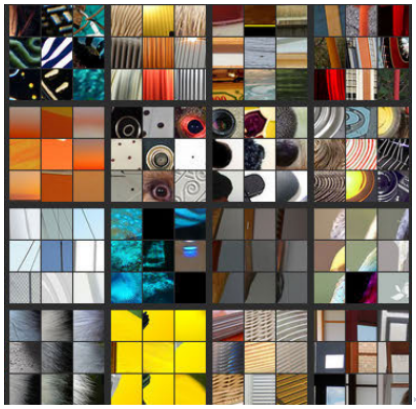
Learned Features, Hidden Layer 1

Learned Features, Hidden Layer 4 and 5





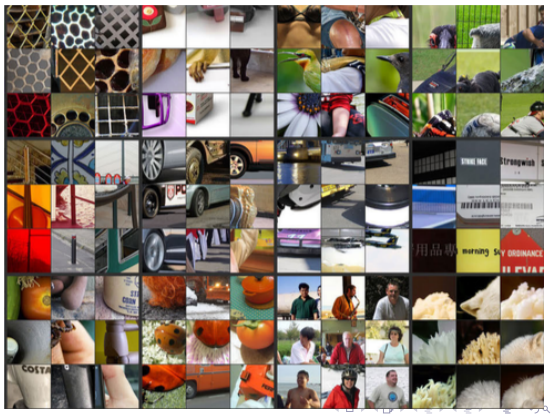
Learned Features, Hidden Layer 2



Summary of CNN - General

- Studied NN architectures
- Fully connected or sparse
- Convolutional NN (CNN) learn hierarchy starting with local features
- Deep CNN were breakthrough for image understanding

Learned Features, Hidden Layer 3



Deep Convolutional Neural Networks for Go

Deep Convolutional Neural Networks for Go

Deep Convolutional Networks for Go

CMPUT 496

Convolutional Neural Networks (CNN)

Deep Convolutional Neural Networks for Go

- Idea: image recognition is similar to recognizing high-level Go concepts
- Go is a very visual game
- Two papers in 2015 within weeks of each other
- Massive improvement in move prediction for Go From about 40% to 57%
- Both trained on full 19×19 board
- Clark and Storkey, Training Deep Convolutional Neural Networks to Play Go.
- Maddison, Huang, Sutskever, Silver. Evaluation in Go using deep convolutional neural networks. ICLR 2015.

ReLU Activation Function

CMPUT 496

Convolutional Neural Networks (CNN)

Deep Convolutional Neural Networks for Go

- ReLU stands for rectified linear unit
- Very popular simple nonlinear function
- $f(x) = \max(0, x)$
- Derivative is also very simple:
- 0 for $x < 0$
- 1 for $x > 0$
- Often more efficient learning than sigmoid, other continuous activation functions

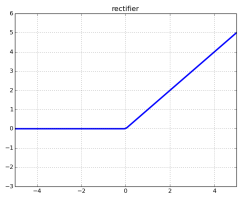


Image source:

https://i.stack.imgur.com/8CGlM.png

Clark and Storkey DCNN

CMPUT 496

Convolutional Neural Networks (CNN)

Deep Convolutional Neural Networks for Go

Studied two different types of input

- Raw input, three *channels*
  - Channel = 19×19 bitmap (boolean array)
  - Black stones
  - White stones
  - Point forbidden by simple ko rule
- Input with liberty information, seven channels
  - Split black, white stones into three separate channels by liberty count
  - 1 liberty (in atari)
  - 2 liberties
  - 3 or more liberties

More Design Features

CMPUT 496

Convolutional Neural Networks (CNN)

Deep Convolutional Neural Networks for Go

- Edge encoding
  - Extra channel to indicate edge of board
    - Avoid that the zero-padding looks like empty points
- Masked training
  - Do not backpropagate results for illegal moves
- Reflection preservation
  - Share weights to make sure output is same if we rotate or mirror the board

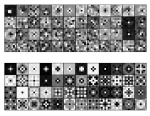


Image source: Clark and Storkey

Architecture

CMPUT 496

Convolutional Neural Networks (CNN)

Deep Convolutional Neural Networks for Go

- Many convolutional layers
- Many small convolutional filters
- Zero-padding to keep each layer at size 19×19
  - Add extra rows and columns with values of 0 around the edges after applying filters
- One fully connected layer at the top
- ReLu activation function

Training Data

CMPUT 496

Convolutional Neural Networks (CNN)

Deep Convolutional Neural Networks for Go

- Two data sets
- 81,000 games by professionals from Games of Go on Disk (GoGoD) collection
- 86,000 games by strong amateurs from KGS Go server
- About 16.5 million move-position pairs for each dataset
- Split into 88% training, 8% test, 4% validation data

Training Method

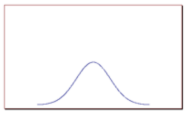
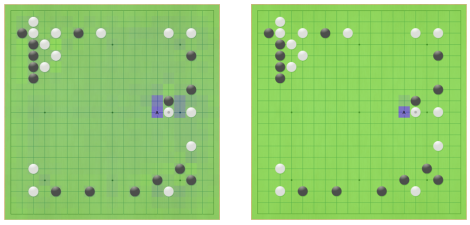


Image source: `http://images.tutorvista.com/cms/images/67/NORMAL-DISTRIBUTION.PNG`

- Basic gradient descent algorithm
- Initialize weights randomly, from *normal distribution* with mean zero, standard deviation 0.01
- 10 “epochs” of training over all data
- Measured three metrics:
  - Accuracy, Rank, Probability

DCNN vs Simple Features

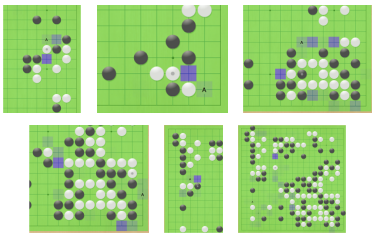


- Compare simple features with interactions (left) vs Clark and Storkey DCNN (right)
- Simple features: more generic knowledge, spread out
- DCNN: focused on very small number of moves

Accuracy, Rank, Probability

- Accuracy
  - How often is the network’s move equal to the master move?
- Rank
  - In the ordered list of moves as ranked by the net, where is the master move?
- Probability
  - What probability did the net assign to the master move?
- Best: 100% accuracy, rank 1, probability 100%
- As discussed before, this cannot be achieved in Go
  - Equally good moves
  - Different preferences in opening
  - Different ideas about the “safest” way to win, or best way to complicate game when losing

Strong and Weak Points of DCNN



- Many good moves, but not enough to play good Go
- About 10 bad moves per game, some real blunders
- Bad moves all from one randomly chosen game
- Need to combine it with search and simulations

Results

	Accuracy	Rank	Probability
Test Data	41.06%	5.91	0.1117
Train Data	41.86%	5.78	0.1158

Table 2. Results for the 8 layer DCNN on the train and test set of the GoGoD dataset. Rank refers to the average rank the expert’s move was given, Probability refers to the average probability assigned to the expert’s move.

	Accuracy	Rank	Probability
Test Data	44.37%	5.21	0.1312
Train Data	45.24%	5.07	0.1367

Table 3. Results for the 8 layer DCNN on the train and test set of the KGS dataset. Rank refers to the average rank the expert’s move was given, Probability refers to the average probability assigned to the expert’s move

Image source: Clark and Storkey

- About 4% improvement over previous best results
  - Using simple features and pairwise interactions
- Trained 4 days on single GPU
- Did *not* use the previous move as an input feature, which is very important with simple features
- Policy-only player, no search
- Wins most games vs GNU Go, some vs (old) Fuego
- Demo at `https://chrisc36.github.io/deep-go/`

Maddison et al Paper

- First Deepmind Go Paper
- Appeared soon after Clark and Storkey
- Very similar approach overall
- More input features, more training, deeper net
- Much stronger results

Input Feature Planes

- Total 36 input planes (Clark/Storkey had 7)

Feature	Planes	Description
Black / white / empty	3	Stone colour
Liberties	4	Number of liberties (empty adjacent points)
Liberties after move	6	Number of liberties after this move is played
Legality	1	Whether point is legal for current player
Turns since	5	How many turns since a move was played
Capture size	7	How many opponent stones would be captured
Ladder move	1	Whether a move at this point is a successful ladder capture
KGS rank	9	Rank of current player

Table 1: Features used as inputs to the CNN.

Architecture and Training

- 12 layers, ReLU activation function
- First layer has 5×5 filters, later layers 3×3
- Between 64 and 192 filters per layer
- Big network:
  - 2.3 million parameters
  - 630 million connections
  - 550,000 hidden units
- Asynchronous stochastic gradient descent
- 50 models trained in parallel with 1 GPU each
- Final training to create a single model, with reduced learning rate

Results (1)

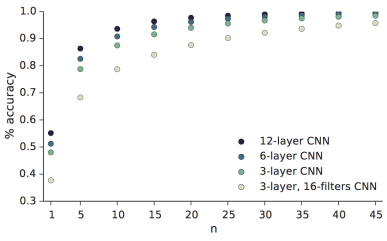


Figure 1: Probability that the expert's move is within the top-n predictions of the network. The 10 layer CNN was omitted for clarity, but it's performance is only slightly worse than 12 layer. Note y-axis begins at 0.30.

Image source: Maddison et al

Results (2)

Depth	Size	% Accuracy	% Wins vs. *GnuGo*	stderr
3 layer	16 filters	37.5	3.4	± 1.1
3 layer	128 filters	48.0	61.8	± 2.6
6 layer	128 filters	51.2	84.4	± 1.9
10 layer	128 filters	54.5	94.7	± 1.2
12 layer	128 filters	55.2	97.2	± 0.9
8 layer (Clark & Storkey, 2014)[4]	≤ 64 filters	44.4	86	± 2.5
Aya 2014		38.8	6	± 1.0
Human 6 dan		52 ±5.8	100	

Image source: Maddison et al

Results (3)

Opponent	Rollouts per move	Games won by CNN	stderr
GnuGo		97.2	± 0.9
MoGo	100,000	45.9	± 4.5
Pachi	100,000	11.0	± 2.1
Fuego	100,000	12.5	± 5.8
Pachi	10,000	47.4	± 3.7
Fuego	10,000	23.3	± 7.8

Image source: Maddison et al

Summary and Outlook

- Deep convolutional neural nets (DCNN) revolutionized image understanding
- They also work very well for move prediction in Go
- Best static (no search) method known
  - A very simple MCTS + DCNN in the Deepmind paper showed they could be combined in principle
- After these two papers, everyone started building DCNN for their Go programs
- Deepmind went quiet for a while
- Then the first AlphaGo paper appeared in January 2016

Part V

RL, AlphaGo and Beyond

Reinforcement Learning

We will learn about:
Reinforcement Learning (RL) introduction
Credit assignment problem
Learning from rewards and temporal differences
TD-gammon as early example
Training by RL
Deep RL

Basic concepts of RL

Basic Concepts of RL

CMPUT 496

- Observe input (state of game)
- Produce move, action
- Observe reward (quality of action)
- Often, the reward is *delayed*
  - Games: reward only at end of game

Credit Assignment Problem

Reward for (possibly long) sequence of decisions
No direct reward for each single move decision
How can we tell which moves are good or bad?
Distribute reward from end of game over all actions
Difficult problemRL provides the most popular answers
Main idea: if same action happens in many different sequences, we can learn if it leads to more wins or losses

Basic Concepts of RL in Games

CMPUT 496

- Play a game from some state *s* until the end
- Receive *reward* at the end
  - Win, loss, or numeric score
- Use that reward to learn about the quality of the individual moves played
  - Credit assignment problem
  - How can we learn which single moves were good or bad?
  - We only get reward for the whole sequence, not for individual moves

RL vs Supervised Learning

RL vs Supervised Learning in Games

CMPUT 496

Supervised Learning
- Label for each move
  - Good/bad, expert move/not expert move
- Learn - minimize prediction error on given data set
- Can use mathematical optimization techniques, e.g. gradient descent

Reinforcement Learning
- Reward for whole game sequence only
- Learn - try to improve gameplay by trial and error
- Need to solve the *credit assignment problem*

Review - Backgammon

CMPUT 496

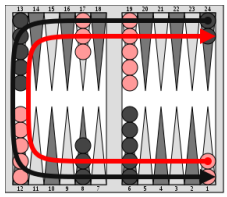


Image source: https://en.wikipedia.org/wiki/Backgammon

- Racing game played with dice
- Players race in opposite directions on the 24 *points*
- Single pieces can be captured and have to start from the beginning
- Doubling cube - play for double stakes, or resign
- Gammon and backgammon - win counts more if opponent is far behind

Tesauro's Neurogammon and TD-Gammon

CMPUT 496

- Neurogammon (Tesauro 1989)
- Plays backgammon using neural networks
- First program to reach "strong intermediate" human level, close to expert
- Beat all (non-learning) opponents at 1989 Computer Olympiad
- Beat many intermediate level humans, lost to an expert player

Neurogammon Architecture

CMPUT 496

- Six separate networks, for different phases of the game
- Fully connected feed-forward nets
- One hidden layer
- Trained with backprop
  - Supervised learning from 400 expert games
- One more network to make doubling cube decisions
  - Trained on 3000 positions, hand-labeled

Limitations of Neurogammon

CMPUT 496

- Hand-engineered features are difficult to create
- Human experts cannot explain much of what they are doing in a form that can be programmed
- Human expert games are difficult to collect, and are not perfect

TD-Gammon

CMPUT 496

- TD-Gammon (Tesauro 1992, 1994, 1995)
- Training by self-play
- Learns from the outcome of games
- Uses Temporal Difference (TD) Learning
  - A technique for function approximation by RL

TD-Gammon Architecture

CMPUT 496

- 198 inputs - 8 per point, 6 extra information (pieces off the board, toPlay)
- Single hidden layer, tried 10..80 hidden units
- Sigmoid activation function
- Output: one number, winning probability of input position
- Trained by $TD(\lambda)$ with $\lambda = 0.7$, learning rate $\alpha = 0.1$
- 200,000 training games, 2 weeks on high-end workstation
- Small (1-3-ply) Alphabeta search

TD-Gammon - Examples of Weights Learned

CMPUT 496

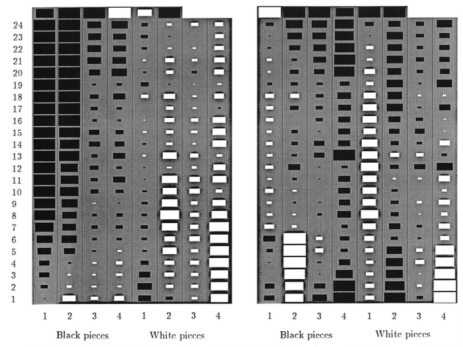


Image source: Tesauro, Practical Issues in Temporal Difference Learning, Machine Learning, 1992

Computer Backgammon Now

CMPUT 496

- Programs generally follow the TD-Gammon architecture
- Bigger, faster, longer training
- Endgame databases with exact winning probabilities
- Considered almost perfect
- Much stronger than humans

TD-Gammon - Examples of Weights Learned

CMPUT 496

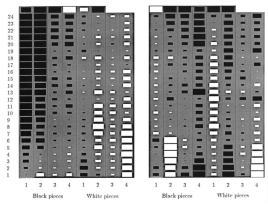


Image source: Tesauro, Practical Issues in Temporal Difference Learning, Machine Learning, 1992

- Weights from input to two of the 40 hidden units
- Both make sense to human expert players
- Top: corresponds to ho is ahead in the race
- Bottom: probability that attack will be successful

Temporal Difference (TD) Learning and $TD(\lambda)$

CMPUT 496

- Sutton (1988)
- Learn a model - a function from inputs to outputs
- Given only action sequences and rewards
- Learns a prediction (what is the best move?)
- Samples the environment (plays games)
- Compares learned estimate in each state with reward
- Learns from the difference
- Discount factor λ for future rewards
- The sooner after the current state the reward happens, the higher the effect

TD-Gammon Impact

CMPUT 496

- Much stronger than Neurogammon
- Close to top human players
- Changed opening theory
- Changed the way the game is played by human experts
- For many years, the most impressive application of RL

TD High-level Ideas

CMPUT 496

- Usually, predictions from states closer to the end are more reliable
- We can adjust earlier predictions, “trickle down”
- Bootstrapping - learn predictions from other predictions
- Whole process is grounded in the true final rewards
- This is one successful approach to solving the credit assignment problem in practice

Summary of RL Introduction

CMPUT 496

- Reinforcement learning for learning from self-play
- TD-Gammon as early success story
- Very small (for todays standard) net with 1 hidden layer
- World class performance
- Trained by RL, more specifically the $TD(\lambda)$ algorithm

AlphaGo Introduction

CMPUT 496

- High-level overview
- History of DeepMind and AlphaGo
- AlphaGo components and versions
- Performance measurements
- Games against humans
- Impact, limitations, other applications, future

Computing Science (CMPUT) 496

Search, Knowledge, and Simulations

Martin Müller

Department of Computing Science
University of Alberta
mmueller@ualberta.ca

Winter 2019

About DeepMind

CMPUT 496

DeepMind

- Founded 2010 as a startup company
- Bought by Google in 2014
- Based in London, UK, **Edmonton (from 2017)**, Montreal, Paris
- Expertise in Reinforcement Learning, deep learning and search

496 Today - Lecture 22

CMPUT 496

- AlphaGo - overview and early versions
- Coursework
- Work on Assignment 4
- Reading: AlphaGo Zero paper

DeepMind and AlphaGo

CMPUT 496

Image source:

https://www.nature.com

- A DeepMind team developed AlphaGo 2014-17
- Result: Massive advance in playing strength of Go programs
- Before AlphaGo: programs about 3 levels below best humans
- AlphaGo/Alpha Zero: far surpassed human skill in Go
- Now: AlphaGo is retired
- Now: Many other super-strong programs, including open source
- All are based on AlphaGo, Alpha Zero ideas

DeepMind and UAlberta

CMPUT 496

- UAlberta has deep connections
- Faculty who work part-time or on leave at DeepMind
  - Rich Sutton, Michael Bowling, Patrick Pilarski (all part time), Csaba Szepesvari (on leave)
- Many of our former students and postdocs work at DeepMind
  - David Silver - UofA PhD, designer of AlphaGo, lead of the DeepMind RL and AlphaGo teams
  - Aja Huang - UofA postdoc, main AlphaGo programmer
  - Many from the computer Poker group
- Some are starting to come back to Canada (DeepMind Edmonton and Montreal)

AlphaGo Versions and Publications

CMPUT 496

- Worked on Go program for about 2 years, mostly kept secret
- DCNN paper published in 2015 (see previous lecture)
- Fall 2015
  Match **AlphaGo Fan** vs
  European champion Fan Hui (2 dan professional)
  - AlphaGo won 5:0 in official games, lost some other games
  - Match kept secret until January 2016
- January 2016
  Article in Nature describes this version of AlphaGo

State of Computer Go before AlphaGo

CMPUT 496

Summary of previous work, and of our course so far

- Search - MCTS, quite strong
- Simulations - OK, hard to improve
- Knowledge
  - Good for move selection
  - Considered **hopeless for position evaluation**
- Strength: about 3 handicap levels below best humans
- Quick progress considered unlikely - see next slide

AlphaGo Versions and Publications (continued)

CMPUT 496

- March 2016
  Match **AlphaGo Lee** vs top-level human player Lee Sedol, wins 4 - 1
- January 2017
  AlphaGo Master plays 60 games on internet vs top humans, wins 60 - 0
- May 2017
  Match vs world #1 Ke Jie, wins 3 - 0
  AlphaGo retires from match play
- October 2017
  AlphaGo Zero article in Nature
  Learns without human game knowledge
- December 2017/December 2018
  Alpha Zero article preprint/final - chess and shogi

Online Betting Market - Computer Go Program Beats Human Champion

CMPUT 496

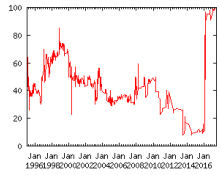


Image source:
http://www.ideosphere.com/fx-bin/Claim?claim=GoCh

- Online “play-money” betting market
- Claim: A machine will be the best Go player in the world, sometime before the end of 2020
- Before AlphaGo, chances were considered low, and falling
- First Nature paper changed it around completely

AlphaGo Project

CMPUT 496

Image source: http://sports.sina.com.cn/go/2016-12-19/

- AlphaGo project was “Big Science”
- Dozens of developers
- World experts in RL, game tree search and MCTS, neural nets, deep learning
- Many millions of dollars in hardware and computing costs
- Huge engineering effort

AlphaGo vs Fan Hui

CMPUT 496

Image source:

https://www.geekwire.com/2016/

- Fall 2015 - early AlphaGo version **AlphaGo Fan Hui**
- Fan Hui: trained in China, 2 dan professional
- Lives in Europe since 2000, French citizen
- European champion 2013 - 2015
- AlphaGo beat Fan Hui by 5:0 in official games
- Fan Hui won some faster, informal games

Alpha Go Fan Design (continued)

CMPUT 496

- **New: strong RL policy network**
- Learns by Reinforcement Learning (RL) from self-play
- **New: value network**
- Learns from labeled game data created by strong RL policy
- Main contributions
  - Reinforcement learning for training DCNN policy net
  - Value network for state evaluation
  - Large, extremely well-engineered learning and playing system

Alpha Go Article in Nature

CMPUT 496

Image source:

https://www.nature.com

- Nature and Science are the top two scientific journals
- Papers on computing science are rare
- Papers on games are very rare
- Games articles from our dept:
  - Checkers Is Solved, Science 2007
  - Heads-up limit hold'em poker is solved, Science 2015
- January 2016 AlphaGo on title page of Nature
- Describes version that beat Fan Hui

Rollout Policy Network

CMPUT 496

- Used for simulations in MCTS
- Designed to be simple and fast
- "Linear softmax of small pattern features"
- Probabilistic move selection as in Coulom's paper, Go4
- Weights trained by RL instead of MM
- Slightly larger patterns/extended features
- 24.2% move prediction rate
- 2μs per move selection (500,000 simulated moves/second)

Alpha Go Fan Design

CMPUT 496

As described in January 2016 article in Nature

- Search: MCTS (pretty standard, parallel MCTS)
  - Modified UCT combines simulation result and value network evaluation
- Simulation (rollout) policy: relatively normal
  - Uses small, fast network for policy
- Supervised Learning (SL) policy network
  - DCNN, similar to their 2015 paper
  - Learns move prediction from master games
  - Improved in details, more data

AlphaGo Fan Deep Network Architecture

CMPUT 496

- Three "big" networks
  - SL policy network, RL policy network, RL value network
- Same overall design for all three
- 13-layer deep convolutional NN
- 192 filters in convolutional layers
- 4.8ms evaluation on GPU
  - About 200 Evaluations/second/GPU
  - More than 2000x slower than simulation policy evaluation
  - Can learn and represent **much more complex** information

Input Features for NN

CMPUT 496

Extended Data Table 2 | Input features for neural networks

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Feature planes used by the policy network (all but last feature) and value network (all features).

Image source: AlphaGo paper

- Pretty similar input as previous nets

AlphaGo Fan Training Pipeline

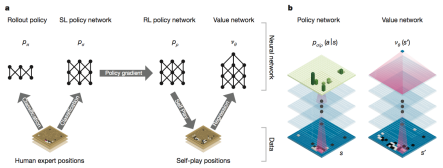
CMPUT 496

Supervised Learning (SL)

- Rollout policy
- SL policy network

Reinforcement Learning

- RL policy network
- Value network



Value Network

CMPUT 496

- Learn a *state evaluation function*
- Given a Go position
- Computes probability of winning
- No search, no simulation!
- Learns mapping
  - From state s
  - To expected outcome $\mathbb{E}[z]$ of the game
- Network architecture mostly same as policy nets
- Difference: output layer
- Value net outputs single number: evaluation of s
  - Compare policy nets: output = probability distribution

Supervised Learning (SL) Policy Network

CMPUT 496

- Trained from 30 million positions from the KGS Go Server
- Maximize likelihood of human move
- 57% move prediction when using all simple input features
- 55.7% when using only raw board input
- “Small improvements in accuracy led to large improvements in playing strength”
- MCTS with this network is already much better than all previous Go programs

AlphaGo Fan Learning Pipeline and Play

CMPUT 496

- Training Pipeline
- AlphaGo program
- Performance measurements
- Games against humans

RL Policy Network

CMPUT 496

- Learns from self play
- Learning: adjust weights of net to learn from winner of each game
- Exactly same network architecture as SL net
- Weights initialized from SL net
- Then plays many millions of games
- Keeps a pool of previous networks as opponents to avoid overfitting

Rewards and RL Training Procedure

CMPUT 496

- Goal: learn improved weights ρ for current network
- Play full game
  - Reward z only at end
  - +1 for winning, -1 for losing
- Update weights ρ by stochastic gradient ascent
- Ascent: go in the direction that maximizes expected outcome of game
- Formula from REINFORCE learning method (Williams 1992)

Training the Value Net

CMPUT 496

- Value net computes a function $v_\theta(s)$
- Function arguments:
  - input state s
  - Value net weights θ
- Output: $v_\theta(s)$ is a single real number
- Training: minimize squared error between
  - Value net prediction $v_\theta(s)$
  - True game outcome z_i

$$\text{Squared error} = \sum_i (v_\theta(s_i) - z_i)^2$$

- Measure mean squared error (MSE):
  - Squared error / number of training items

REINFORCE Update Formula

CMPUT 496

REINFORCE update formula

$$\Delta\rho = \alpha \frac{\partial \log p_\rho(a|s)}{\partial \rho} z$$

- $\Delta\rho$ .. change in weight ρ
- α step size for gradient ascent
- $p_\rho(a|s)$ .. probability of playing the move a which was played in the game from state s
- z .. game result (+1 or -1)

Training Data for Value Net

CMPUT 496

- Played 30 million self-play games using the RL policy player
- Randomly sample *one single* state s from each game
  - Why only one state? Because of overfitting problems
- Label s with the outcome z of the game
- Result: 30 million training points (s_i, z_i)
- Learn the value net by trying to predict z_i from s_i
- This was called "reinforcement learning of value networks" in the paper
- It is really supervised learning from game data
- Game data was produced by the RL policy

RL Policy Network vs Other Go Programs

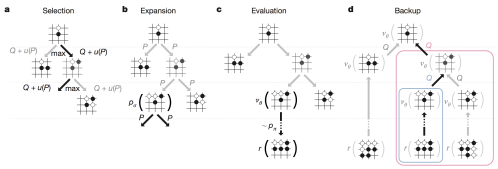
CMPUT 496

- Tested RL Policy Network as a standalone player
- No search, only one call to network
- Plays move by sampling from p_ρ
- Won 80% against SL policy net
- Won 85% vs MCTS Go program `pachi` which used 100,000 simulations/move (!)

MCTS in AlphaGo Fan Player

CMPUT 496

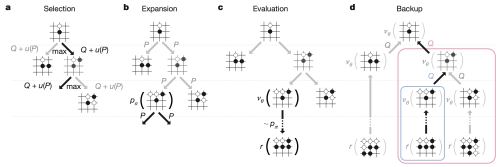
- Player uses MCTS - search and simulations
- Neural nets used as (very strong) in-tree knowledge
  - Policy net guides tree search
  - Value net helps evaluate leaf nodes
- 3 values stored on each edge (s,a) of game tree
- Winrate Q, visit count N, prior probability P from policy net



In-tree Move Selection

CMPUT 496

- Choose move which maximizes $Q + u$
- Winrate Q, exploitation term
- u exploration term, with multiplicative knowledge
- Decay by $u = C\frac{P}{1+N}$
- Exploration constant C



SL vs RL policy network in MCTS

CMPUT 496

- Search with RL policy was too narrow
- Quality of other candidate moves not as good as in SL network
- Result: they ended up using the SL policy network, not the RL policy network in AlphaGo Fan
- SL net worked better in MCTS, even though it is much weaker in move prediction
- SL net gave a better set of good moves to search, not just a single strong move
- In AlphaGo Zero, this problem was finally solved by defining a different learning target for the policy net

Computing $V(s_L)$

CMPUT 496

- Value estimate $V(s_L)$ of leaf node s_L
- $V(s_L)$ is weighted average of two different evaluations
- $v_\theta(s_L)$ = Value network evaluation of s_L
- z_L = win/loss result of single simulation from s_L
- Combined by $V(s_L) = (1-\lambda)v_\theta(s_L) + \lambda z_L$
  - $\lambda = 0.5$, equal weight

Evaluation and Parallel Search

CMPUT 496

- Evaluating policy and value networks is expensive!
- Asynchronous multi-threaded search
- Simulations run on CPUs
- Policy and value networks evaluated on GPUs
- In match vs Fan Hui:
  40 search threads, 1,202 CPUs and 176 GPUs

Which Prior Probability P to Use?

CMPUT 496

- Remember meaning of knowledge term in MCTS:
- Give prior to good moves that should be searched
- Problem of RL policy in AlphaGo Fan:
- It was optimized to play well
- It learned a very strong prior for a **single** move
- It was not optimized to produce a good **set** of moves to search in MCTS

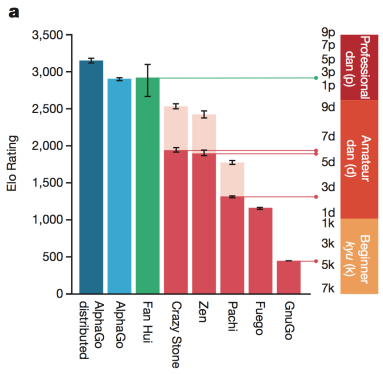
Elo Model of Playing Strength

CMPUT 496

- Numerical rating scale
- Developed for chess
- Works well for many games of skill
- Basic idea: difference in rating corresponds to winning probability
- In Go, 100 Elo is roughly 1 stone handicap
- Formula: Difference d, win probability $1/(1+10^{d/400})$
- Example: 200 Elo stronger = wins about 80% of games

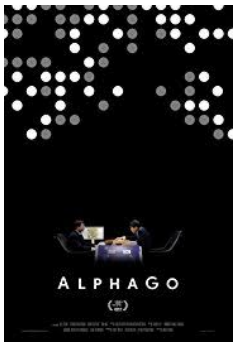
AlphaGo Fan Playing Strength - 2016 Nature Paper

CMPUT 496



AlphaGo Movie

CMPUT 496



- `https://www.alphagomovie.com`
- Story of AlphaGo development and match vs Lee Sedol
- Director Greg Kohs
- Very good movie, focus on human aspects

Image source: `http://www.imdb.com/title/tt6700846/`

Engineering and Technical Contributions

CMPUT 496

- Massive amounts of self-play training for the neural networks
- Massive amounts of testing/tuning
- Parallel training algorithms
- Large-scale parallel asynchronous search
- Large hardware:
- 1202 CPU, 176 GPU used vs Fan Hui

AlphaGo Lee

CMPUT 496

- Version used for Lee Sedol match, March 2016
- Much more RL training than for Fan Hui match
- Trained from **AlphaGo selfplay games**, not from RL policy games anymore
- Larger neural net
- Better hardware to evaluate neural nets
- About 2000 CPU, 48 TPU used
- 1 TPU: maybe 30x faster than 1 GPU
- Strength: 3 handicap stones stronger than AlphaGo Fan in self-play

AlphaGo vs Lee Sedol

CMPUT 496

Image source: `http://time.com/4257406/go-google-alphago-lee-sedol/`

- 5 game match in March 2016
- First ever match computer vs top player on 19 $\times$ 19 board, with no handicap
- Lee Sedol was the world's top Go player for more than a decade
- Won large number of international tournaments
- Still very strong at time of match, about world #3
- AlphaGo won the match 4:1
- Lee won game #4

AlphaGo Master

CMPUT 496

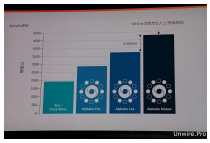


Image source: `https://unwire.pro/2017/05/25`

- Played online end of December 2016 - early January 2017
- 60 fast games, many against top human players
- Score 60 wins no losses

AlphaGo Master Architecture

CMPUT 496

- Mostly same architecture as AlphaGo Zero (next lecture)
- Some parts were still the same as in previous AlphaGo, different from Zero:
  - Still uses handcrafted input features
  - Still uses rollouts as part of evaluation
  - Still uses initialization by SL net
- Reduced hardware:
  4 TPU, “single machine”
- Strength:
  - Far surpasses humans
  - 3 handicap stones stronger than AlphaGo Lee in direct match

Computing Science (CMPUT) 496

Search, Knowledge, and Simulations

Martin Müller

Department of Computing Science
University of Alberta
mmueller@ualberta.ca

Winter 2019

AlphaGo vs Ke Jie

CMPUT 496

Image source:

- Match in May 2017 at “Future of Go Summit”
- AlphaGo played world #1 Ke Jie
- AlphaGo won 3:0
- Program: similar to AlphaGo Master
- Also on reduced hardware: 4 TPU, “single machine”

496 Today - Lecture 23

CMPUT 496

- Quiz 11 review
- AlphaGo Lee and Master versions
- AlphaGo Zero
- USRI Evaluations start today, open Apr 4 - 10
- Coursework:
- Finish Assignment 4 - due Apr 8

Summary

CMPUT 496

- Discussed the earlier versions of AlphaGo
- AlphaGo Fan, AlphaGo Lee, AlphaGo Master
- Main architecture: MCTS + neural networks for knowledge
- Policy net for biasing in-tree move selection
- Value net plus simulations for evaluating leaf nodes in tree
- Massively parallel implementation on CPU for simulations + GPU or TPU for nets
- Quantum leap in performance of Go-playing programs
- Reached, then surpassed level of best human Go players

USRI (Universal Student Ratings of Instruction)

CMPUT 496

- You should have received an email
- If not, use this link:
  `https://usri.srv.ualberta.ca/etw/ets/et.asp?nxappid=WCQ&nxmid=start`
- Important part of evaluating this course
- Part of instructor and TA’s annual evaluation
- Please fill it out!

AlphaGo Zero

CMPUT 496

- October 2017 article in Nature
- Mastering the game of Go without human knowledge
- New simplified architecture
- Learns entirely from self-play
- No human knowledge beyond the basics such as rules of game
- Stronger than previous AlphaGo versions
- Far super-human skill

Human Knowledge Not Used in Zero

CMPUT 496

Some examples of knowledge used in many other Go programs, but not in Zero

- Avoid eye-filling moves (as in `GoI`)
- Patterns
- Tactics, atari, selfatari
- Human game records
- Rules for simulation policy
  No simulations outside of tree used in Zero

Zero has to learn many of these basics, and then much much more.

Main Technical Changes in AlphaGo Zero

CMPUT 496

- New training method tailored for improving MCTS
- Self-learning from predicting searched moves and outcomes
- New network architecture: resnets
- New network architecture: combine policy and value nets into one net with two "heads"
- Does not need large distributed system anymore, strong performance on "one machine"

AlphaGo Zero's Search

CMPUT 496

- Search - still MCTS
- Used in two different ways:
  - **For learning (new)**
  - For playing
- The most impressive innovation in Zero is how search is used to improve learning, and learning in turn improves search

Human Knowledge in Zero

CMPUT 496

- Rules of Go, legal moves
- Hard limit of $19 \times 19 \times 2 = 722$ moves on game length
- Tromp-Taylor scoring
- Input has same 2-d grid structure as Go board
- Uses rotation and reflection invariance of Go rules for training
- MCTS search parameters optimized

Main Components of AlphaGo Zero - Knowledge

CMPUT 496

Knowledge

- All knowledge created by machine learning from self-play
- New network architecture
  - Knowledge represented by deep residual neural net
  - Combines policy and value nets into one net with two "heads"
  - Both move and position evaluation learned together
- No more simulations (rollouts) to end of game!
  - MCTS tree growth controlled **only** by neural net knowledge (plus real end of game states reached in the tree)

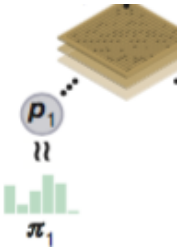
Knowledge Representation

CMPUT 496

- Deep residual neural network (He et al 2015)
- Learns two types of knowledge simultaneously
- Policy head
  - Learns good moves for the search
- Value head
  - Learns evaluation function - probability of winning
- Most of network is shared between both

Output 1 of Neural Network: Policy Head

CMPUT 496



- Learns to predict what the search would do
- How frequently should each move be tried in MCTS?
- Learning goal: minimize *cross-entropy* between
  - Predicted probability of move
  - Frequency of move as selected by MCTS
- Cross-entropy: measures how well one probability distribution can predict another

Deep Residual Neural Network (Resnet)

CMPUT 496

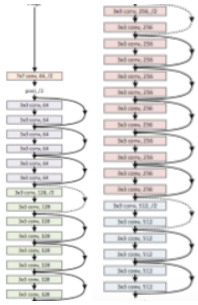
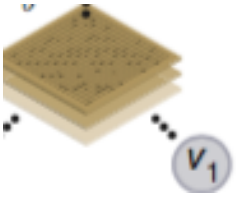


Image source: He et al 2015 article

- Main idea: pass output of previous "block" directly through
- Each block learns a "delta", a small change to previous output
- Learning small changes is easier
- Can train really deep nets efficiently
- $>$100 layers in image recognition
- In theory, no greater representational power than DCNN
- In practice, learns better

Output 2 of Neural Network: Value Head

CMPUT 496



- Given a Go position
- Computes probability of winning
- Static evaluation function
- Trained from selfplay
- Learning goal: Minimize squared error between:
  - Predicted value v
  - Final result z of game

Two-head Architecture

CMPUT 496

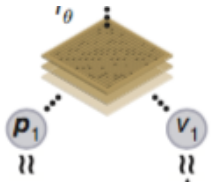


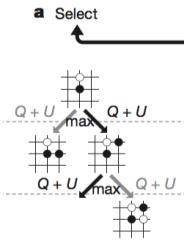
Image source: All further images from AlphaGo Zero paper, unless stated otherwise

$$(p, v) = f_\theta(s)$$

- Deep net
- Input Go position s
- Network weights θ
- Network computes function $f_\theta(s)$
- Two outputs: (p, v)
  - p vector of move probabilities $p(s, a)$ for each move a
  - v value of s

MCTS in AlphaGo Zero - Move Selection

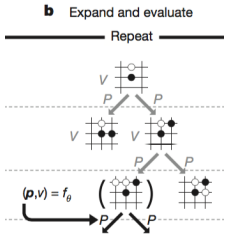
CMPUT 496



- In tree move selection
- Same formulas as in previous AlphaGo
  - Exploitation term Q
  - Exploration term u
- Meaning of Q is slightly changed
  - Value of simulation ending in in-tree state s = value head evaluation of s
  - No more simulation beyond the tree, no more evaluation component from rollouts

MCTS in AlphaGo Zero - Evaluation

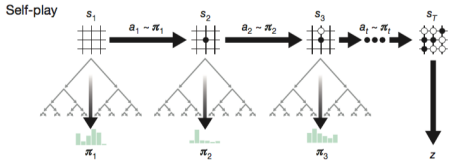
CMPUT 496



- Node s expanded
- Single call to neural net
- $(p, v) = f_\theta(s)$
- p = vector of move probabilities, $p(s, a)$ for all moves a from s
- v estimated value of s

Self-Play Games

CMPUT 496



- Play whole game
- For each state s_t in game:
- Run MCTS on s_t
- Sample move to play according to number of simulations it received
  - Note difference to regular MCTS: *exploration*!
  - Regular MCTS would always pick the most-simulated move (exploitation)
- Finish game, get outcome z (win = +1 or loss = -1)
- Store tuples (s_t, π_t, z_t) for learning after end of game

Training

CMPUT 496

- Error measured by *loss function*
- Combines three terms
  - Error of policy head (cross entropy)
  - Error of value head
  - Regularization term to keep size of weights in check

Meaning of Tuples

CMPUT 496

- (s_t, π_t, z_t)
- s_t = state at time step t
- Game = sequence of states s_1, s_2,...
- π_t = probability distribution derived from visit count of moves in MCTS of s_t
- z_t = result from current player's point of view (z or $-z$, negamax)

MCTS Visit Count and Policy π

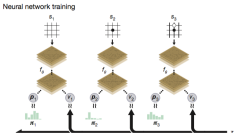
CMPUT 496

Meaning of policy π:

- Run MCTS from some state s
- If move a was played $N(s, a)$ times:
- $\pi_t(a) \propto N(s, a)^{1/\tau}$
- Probability is proportional to its "exponentiated visit count"
- Temperature parameter τ controls exploration of low-probability moves
- $\tau = 1$ for early game only, small for rest of game
- What does "proportional" mean?
- Compute values $N(s, a)^{1/\tau}$ for all actions a, then divide by their sum to make them into probabilities

Learning from Self-Play Games

CMPUT 496



- After each game
- Randomly sample tuple (s, π, z) from all tuples stored from the game
- Adjust net weights θ by gradient descent:
- $(p, v) = f_\theta(s)$
- Make policy p better match π
- Make value v better match z

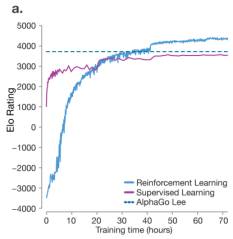
Training Process

CMPUT 496

- Most tests with 20 block resnet
- 4.9 million self-play games
- 1600 simulations / move in MCTS
- Update net in minibatches of 2048 game positions

Comparing Early Learning - RL vs SL and AlphaGo Lee

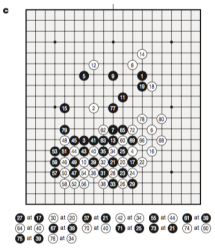
CMPUT 496



- After 72 hours of training
- Slow games vs AlphaGo Lee
- 2 hours per game per player
- Zero won 100 - 0

Zero After 3 Hours of Learning

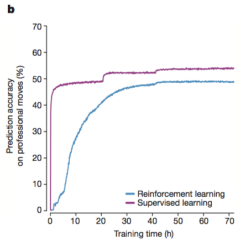
CMPUT 496



- Net learned for 3 hours
- Quick game, MCTS with 1600 simulations/move
- Learned about capturing stones
- Plays like human beginner
- Clearly better than random

Move Prediction

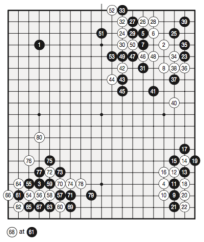
CMPUT 496



- Move prediction of human professional moves
- Learning improves, then plateaus
- Always stays below SL policy predictions
- Despite lower stats, most moves are very “human-like”

Zero After 70 Hours of Learning

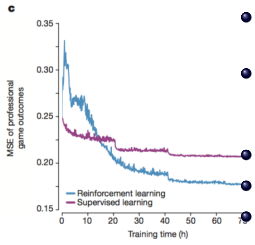
CMPUT 496



- 7 hours
- Quick game, MCTS with 1600 simulations/move
- Plays super-strong game of Go
- Complex strategies
- Exact score estimates, counting

Predicting the Outcome of Games

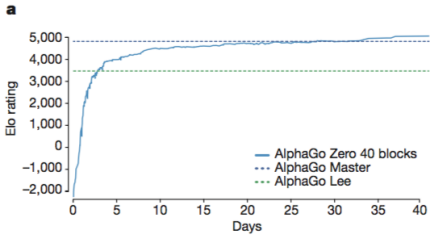
CMPUT 496



- Predict winner of human professional games
- MSE = mean square error between value net and real outcome
- Compare SL and RL value nets
- SL starts with big advantage
- RL becomes much better than SL with more training

Compare Strength with AlphaGo Lee and Master

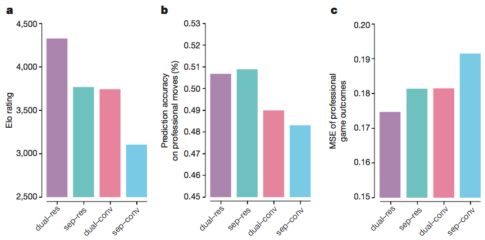
CMPUT 496



- Strongest version - 40 residual blocks instead of 20
- Trained 29 million games, 40 days
- Learning compared to Elo strength of AlphaGo Lee and AlphaGo Master
- Match Zero 40-block vs Master: 89 wins 11 losses in slow games

Comparing Network Architectures

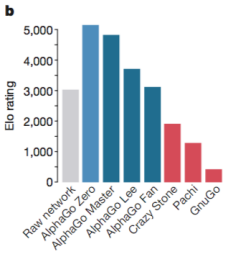
CMPUT 496



- Evaluate Elo strength, move prediction, MSE of game outcomes
- sep (separate networks) vs dual (one net, 2 heads)
- conv (DCNN) vs res (residual net)
- Clear benefit of dual architecture, sharing most of network
- Clear benefit of residual net over DCNN

Results for Different AlphaGo Versions

CMPUT 496



- Compares Elo for different versions of AlphaGo
- Fast games, 5 seconds / move
- Also compares Zero's "raw network" vs full Zero
- Raw network:
  - Evaluate current state s
  - Play highest probability move from policy head
  - No search
  - Almost as strong as AlphaGo Fan (with search)

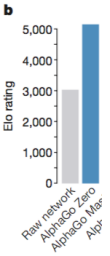
Some Limitations of AlphaGo

CMPUT 496

- AlphaGo only plays 19×19 Go with 7.5 komi
- AlphaGo is not perfect - no proofs, "mastering" vs solving the game
- 5×5 is still the largest solved square board size, since 2002 `http://erikvanderwerf.tengen.nl/5x5/5x5solved.html`
- AlphaGo is not open source - we do not know many of the details

The Importance of Search

CMPUT 496



- Raw network vs AlphaGo Zero, 5 seconds / move
- With search **2000 Elo stronger**
- That's 20 skill levels...
- Same gap as between top human professional and weak club player
- Stronger knowledge makes search even stronger
- No "diminishing returns", value of search remains very high

Limitations of an AlphaGo-like Approach to Problem-Solving

CMPUT 496

- AlphaGo relies on having a perfect model of the game
- Exact rules of game, perfect scoring of outcome, full state of game known
- Model is used for creating many millions of self-play games
- Learning relies on having these games
- Big challenge: how to learn without perfect model
- Much ongoing research, mostly unsolved problem

Impact of AlphaGo

CMPUT 496

- Huge impact in media, outside of core AI community
- Often described as a major step towards “machine intelligence”
- Remember main limitation - still needs an exact model to work well
- Impact on AI and machine learning in general
- Impact on heuristic search and computer game playing
- Impact on human Go community

Impact on Heuristic Search and Computer Game-playing

CMPUT 496

- Dramatic shift to much stronger knowledge
- With each major advance in one of the three areas - search, knowledge, simulations -
- Need to rethink all heuristic search systems
- AlphaGo is only the beginning
- Much work ahead to fully exploit the power of stronger knowledge
- Can we learn stronger knowledge for other, harder problems
  - Video games (e.g. Atari games, Starcraft)
  - More open real-world problems with less well-defined rules

Impact on AI and Machine Learning

CMPUT 496

- Prime example of how combination of search, simulation and knowledge can achieve spectacular results
- Using deep learning:
  - Search improve knowledge
  - Knowledge improves search
  - Virtuous cycle, positive feedback loop
- Simulation has changed dramatically - in-tree only, controlled completely by neural net evaluations, no more rollouts to end of game.
- AlphaGo (Zero) has dramatically shifted the landscape of what knowledge can do as part of a larger search-based system

Impact on Human Professional Go Community

CMPUT 496

- Human professionals study AlphaGo and other AI games intensely
- Try out many AlphaGo-inspired openings
- Some pros are worried for their jobs
- Less interest in human tournaments?
- Can pro-level phone replace human teachers?

Impact on AI and Machine Learning

CMPUT 496

- 10 years ago, with MCTS there was a similar shift in what search can do
- After current round of progress driven by knowledge, is it time for improving search methods again?
- Main questions for me:
  - Which other applications of deep learning can profit from adding search and simulation?
  - Which other applications of heuristic search can profit from deep learning?

Impact on Human Amateur Go Community

CMPUT 496

- Temporary boost in excitement and visibility for the game of Go
- Having strong computer opponents (and online play) helps individuals in small communities without a Go club
- Will cheating become a problem, as in chess?
- Goals:
  - Turn programs into tools for teaching Go
  - Explain programs’ moves in human terms

Impact on Computer Go Community

CMPUT 496

- Mission accomplished? Game over?
- Taking chess as example
  - Public interest in programming Go will fade
  - A core group of enthusiasts will keep going
  - Everyone will use programs as study tools
  - Level of humans will improve from studying with programs
- It is now possible for a single person to write a professional-level Go program in a year
- Recently, several such new programs

Leela Zero

CMPUT 496

Leela Zero

- Strongest open source Go program:
- Public reimplementation of AlphaGo Zero
- Smaller network for faster learning
- Reached top pro level in a few months, still improving rapidly
- Community effort, over 400 participants donate CPU and GPU cycles
- Over 13 million games played
- Improved by 14000 Elo from random play at beginning

Beyond AlphaGo

CMPUT 496

- Computer Go Today
- Applications to other games
- Other types of applications?
- Current research: improving the techniques

Leela Zero Learning Curve

CMPUT 496

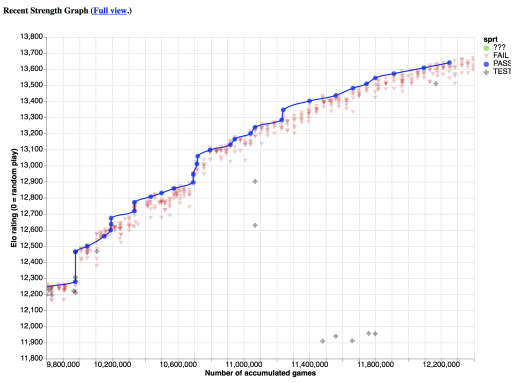


Image source: `http://zero.sjeng.org`

Computer Go Today

CMPUT 496

- Half a dozen professional level Go programs
- Strongest: Tencent's FineArt
  - Can give top human professionals two stones handicap (!) in fast games
- Computer Go Server for automated testing
  - `http://www.yss-aya.com/cgos/`
- Computer Go Tournaments: `http://www.computer-go.info/events/index.html`

Summary

CMPUT 496

- Reviewed AlphaGo Zero in detail
- Discussed impact, state of computer Go after AlphaGo

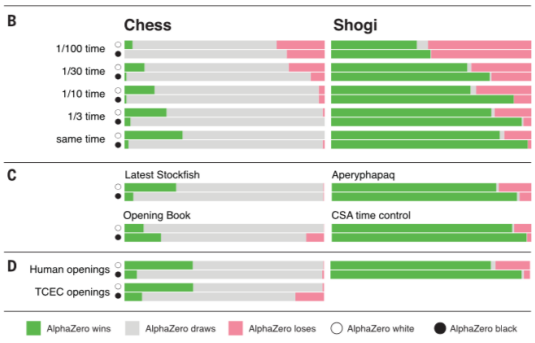
Alpha Zero Approach

CMPUT 496

- Same as in Alpha**Go** Zero:
  - Two-head deep network, with policy and value heads
  - $(p, v) = f_\theta(s)$
  - MCTS for learning from self-play and for playing
- Different from Alpha**Go** Zero:
  - Learns expected outcome, not winning probability
    - Chess and shogi have draws, needs to learn to estimate those
  - Go training and evaluation took advantage of board symmetries
  - Learns by continuous updates to a single network
    - Alpha**Go** Zero learned its networks in generations
    - Each network used games from the previous best net
    - Alpha Zero learns and updates the same single net

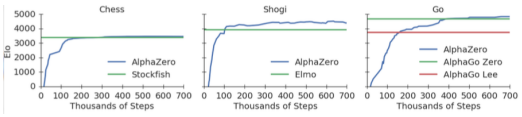
Alpha Zero: Go, Chess and Shogi Results Summary

CMPUT 496



Alpha Zero: Go, Chess and Shogi Learning

CMPUT 496



- Can learn Go, chess, shogi from scratch
- Beat top programs in matches
- In 2018 version, careful results against many versions of top other programs
- Alpha Zero wins even with large time handicap
- Hardware hard to compare - TPU vs parallel CPU

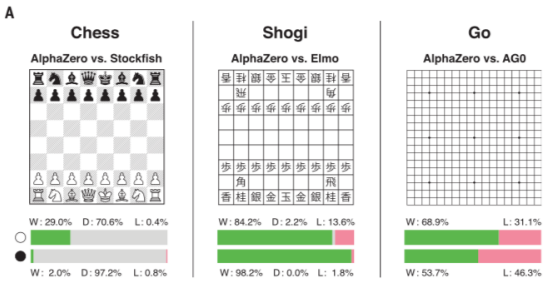
Alpha Zero Summary and Discussion

CMPUT 496

- Very strong result
- Generalizes work on Go to other classical board games
- Stronger than other top chess and shogi programs
- Approach often copied by others
- Examples: five in a row (gomoku), connect 4, other games

Alpha Zero: Go, Chess and Shogi Results Summary

CMPUT 496



Computer Poker

CMPUT 496

- Poker is a very popular family of card games
- Poker was one of the main models for development of mathematical game theory in the 1930s and 1940s
- Imperfect information game - players do not know opponent's cards, or cards that will be drawn in the future
- Our university is one of two major centers of computer poker research
- Carnegie-Mellon University (CMU) is the other
  - Programs: Tartanian, Claudico, Libratus

Recent Poker Publications involving U. of Alberta

CMPUT 496

- Bowling, Burch, Johanson and Tammelin. Heads-up limit hold'em poker is solved. Science, 2015
  - `http://poker.srv.ualberta.ca`
- Moravcik, Schmid, Burch, Lisy, Morrill, Bard, Davis, Waugh, Johanson and Bowling. DeepStack: Expert-level artificial intelligence in heads-up no-limit poker. Science, 2017
  - `https://www.deepstack.ai`

Mixed Strategies vs Probabilistic Simulation Policies

CMPUT 496

- In both cases, we pick an action according to a probability distribution
- However, the meaning behind it is very different
- In simulation policy, we want to sample from a huge game tree, get better exploration
- In mixed strategy, we want to avoid being predictable and being exploited by the opponent
- We need to protect our hidden information (e.g. cards) from being inferred too easily

Game Theory Background

CMPUT 496

- In perfect information games (e.g. Go), solving a game includes:
  - Computing its *minimax value*
  - Finding a winning strategy
  - Strategy includes one move at OR nodes (your turn), all moves at AND nodes (opponent's turn)
- Imperfect information games (e.g. poker):
  - For games with more than two players, there is no analog to minimax value
    - One player's result depends on what all other players do
  - The most important general concept is the *Nash equilibrium*
  - A player's strategy in general needs to be *mixed*

Nash Equilibrium

CMPUT 496

- Defined for multi-player, non-cooperative games
  - Two-player games included as special case
- A set of strategies for all players
- Assuming they behave "rationally" in a certain sense:
- In Nash equilibrium, no single player can profit from changing their strategy unilaterally
- Many Nash equilibria may exist, and they can be quite different from each other in a multiplayer (more than 2) setting

Pure vs Mixed Strategies

CMPUT 496

- Pure strategy: in the same state, always choose the same action
- Mixed strategy: choose action according to a probability distribution
- Example in poker
  Call 80% of the time
  Raise 20% of the time
- In imperfect information games, sometimes mixed strategy needed for optimal play (e.g. Rock-Paper-Scissors)
- In complete information games such as Go, an optimal pure strategy always exists. No need to mix

Imperfect Information Games - Two Player Zero-sum Case

CMPUT 496

Special case - only two players

- *Minimax value* still exists (in expectation)
- Minimax value is achieved by playing a Nash equilibrium strategy
- May *have* to use a mixed strategy to achieve it
- Example: Rock-Paper-Scissors
  - Choose action randomly, each with probability $1/3$
  - You will get an even result in the long run, no matter what the opponent does
  - Any other strategy could be exploited by the opponent

Imperfect Information and Beliefs

CMPUT 496

- Because of unknown information, reasoning about players' beliefs is very important
- Try to infer opponent's cards **and** beliefs from their actions
- Try to hide own cards and beliefs
- Search becomes much more complex
- Search now needs to reason over player beliefs, not just over completely known search states

Cepheus vs Previous Approaches

CMPUT 496

- Previous poker programs: finding (close to) optimum solutions for a simplified world
- Used abstraction to group many different poker states into the same abstract state
- Example: pair of kings treated the same as pair of aces
- They are both very strong hands. But in some important situations, they are very different
- Cepheus does not do that.
  - It approximates the full game of poker directly

Heads-up Limit Texas Hold'em Poker

CMPUT 496

- Limit on size of bets
- State space - 10^{14} decision points
- UofA program Cepheus "essentially solved" the game
- Main progress: different approach to satisficing than in previous poker programs
- Remember Herb Simon's quote from many weeks ago:

Cepheus Approach

CMPUT 496

- Learned a full strategy for all 10^{14} decision points
- Used a new approximation method called CFR+
- Used 900 CPU years of (parallel) computation
- The final result is basically one huge lookup table
  - Contains probability of each action in each state
- Error under 1 milli big blind per hand
- Note log scales in picture...

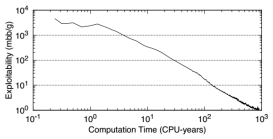


Figure 3: **Exploitability of the approximate solution with increasing computation.** The exploitability, measured in milli-big-blinds per game (mbb/g), is that of the current strategy measured after each iteration of CFR+. After 1579 iterations or 900 core-years of computation, it reaches an exploitability of 0.986 mbb/g.

Herb Simon on Satisficing

CMPUT 496

"Decision makers can satisfice either by finding optimum solutions for a simplified world, or by finding satisfactory solutions for a more realistic world.
Neither approach, in general, dominates the other, and both have continued to co-exist..."

DeepStack and Heads-up No-Limit Texas Hold'em Poker

CMPUT 496

- No limit poker - any size of bet is allowed
- Much larger state space, about 10^{160} decision points
- Program DeepStack announced in March 2017
- Strong heuristic player, beat professionals in a set of matches
- Uses techniques much more similar to heuristic search than previous poker programs

DeepStack Approach

CMPUT 496

- Activity: watch Mike Bowling's talk: `https://www.youtube.com/watch?v=qndXrHcV1sM`
- Previous poker programs reasoned over the whole state space at once
- In complete information games (e.g. Go) you can limit your search to a state and its subtree
- The DeepStack team found a way to decompose the poker computation in a similar way
- "Until DeepStack, no theoretically sound application of heuristic search was known in imperfect information games."

DeepStack Approach (2)

CMPUT 496

- Continuous re-solving of subtree with different vectors - one traversal of search tree is not enough
- Standard fully connected network, trained on 10 million random poker games, starting from the end of game, with random belief vectors
- Output of network is vector of values, not just single evaluation

Exploitability

CMPUT 496

Table 1. Exploitability bounds from Local Best Response. For all listed programs, the value reported is the largest estimated exploitability when applying LBR using a variety of different action sets. Table S2 gives a more complete presentation of these results (*10*).

Program	LBR (mbb/g)
Hyperborean (2014)	4675
Slumbot (2016)	4020
Act1 (2016)	3302
Always Fold	750
DeepStack	0*

*LBR was unable to identify a positive lower bound for DeepStack's exploitability.

Image source: DeepStack paper

DeepStack Neural Net

CMPUT 496

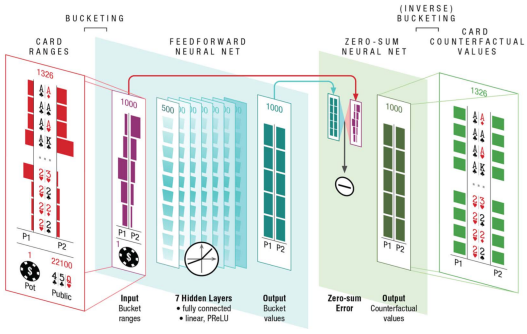


Fig. 3. Deep counterfactual value network. The inputs to the network are the pot size, public cards, and the player ranges, which are first processed into hand clusters. The output from the seven fully connected hidden layers is postprocessed to guarantee the values satisfy the zero-sum constraint, and then mapped back into a vector of counterfactual values.

DeepStack Approach

CMPUT 496

- Summarize history in two vectors, one for each player
  - Beliefs represented by "Range" of cards we could hold, plus "counterfactual values" for the opponent
- Depth-limited forward search using these vectors as state information
- Selective lookahead using only some of the possible bets
- At depth limit, use a deep neural network to evaluate states

DeepStack Approach (3) and Results

CMPUT 496

- Many other technical innovations to make heuristic search work in imperfect information games
- Amazing body of research, well worth deeper study
- Beat many human professionals over 3000 hand series

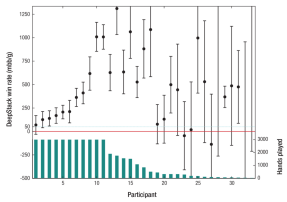


Fig. 4. Performance of professional poker players against DeepStack. Performance estimated with AIVAT along with a 95% confidence interval. The solid bars at the bottom show the number of games the participant completed.

Course Summary, Final Exam, Wrap-Up

CMPUT 496

- Three components of modern heuristic search: search, knowledge, simulations
- Fourth component - machine learning. Learn knowledge, simulation policy, in-tree guidance
- Recent strong focus on knowledge - deep learning, DCNN
- The combination of search and knowledge is much stronger than knowledge alone
  - E.g. first AlphaGo paper - Value net alone = 1600 Elo (weak amateur), Value net + policy net + simulations = 2800 Elo (strong professional) - 1200 Elo difference
  - AlphaGo Zero: 2000 Elo difference between full system and network-only
  - In AlphaGo Zero and Alpha Zero, search improves knowledge, and knowledge improves search - virtuous cycle

Single-Agent Heuristic Search with Simulations

CMPUT 496

- Single-agent MCTS
  - Used for many optimization problems, even for drug discovery in Chemistry
- Nested Monte Carlo Search (Cazenave)
  - Use rollouts, plus recursion to define more powerful rollouts which themselves are Monte Carlo searches
- Nested Rollout Policy Adaptation (Rosin)
  - Online machine learning to improve the simulation policy
  - Holds current world records for several difficult puzzles

If This Course Was Twice as Long ... Some Pointers for Further Studies

CMPUT 496

- In Cmput 496 I have given only a high-level overview of modern heuristic search
- I needed to skip over many very interesting and important topics
- I will now present some of these with one or two slides per topic
- Single agent search, planning, MCTS improvements, tuning, parallel search, Atari games, ...

Automated Planning with Random Walks

CMPUT 496

- In automated planning, you are given an abstract description of a planning problem, start state, and goal
  - Example: plan coordinated movements of airplanes in an airport
- Classical methods used single-agent search and (much) knowledge, exploitation only
- Much work in my research group over last ten years on adding exploration techniques
- Two PhD students Hootan Nakhost and Fan Xie developed algorithms to add simulations by random action sequences to planning
- Strong results, many publications in top venues, student thesis awards

Single-Agent Heuristic Search

CMPUT 496

- We only reviewed the basics of “traditional” single-agent search
  - Search only, or “blind search”
  - Search plus knowledge
- There are many exciting new single-agent methods that combine all three pillars - search, knowledge, simulations
- There are also exciting applications of machine learning for single-agent search

Machine Learning for Single-Agent Search

CMPUT 496

- Often, many different problem instances from same domain
  - Example: same airport, different airplanes landing and taking off at diffrent times
- General idea for many learning approaches:
- Solve small problems by a (quick) search
- Learn generally good actions or action sequences from the solutions for these small problems
- Use the learned knowledge to solve similar, but larger problems

Search Improvements, and Other Applications of Modern Heuristic Search

CMPUT 496

- Better algorithms for MCTS and for solving games
- Other games
- Performance optimization

Specialized Game Solvers

CMPUT 496

- There are algorithms that usually work better than alphabeta for solving games
- Example: Proof-number search, df-pn
- They try to find a small proof tree
- Sometimes, a narrow deep tree can be a smaller proof
- These algorithms are very good in finding such proofs

MCTS Enhancements

CMPUT 496

- Many improvements for MCTS have been developed
- Adaptive simulation policies
- More statistics over all simulation moves (RAVE), over single points on the board (territory map)
- Integrating an exact minimax solver within MCTS
- Adding 1 or 2-ply lookahead to simulations
- Many more...

Tuning and Optimizing

CMPUT 496

- Tuning and optimization are important steps in building high performance heuristic search programs
  - Remember experiment: Fuego 500x faster than our Python demo Go programs
- Better algorithms
- Better caching and re-use of solved subproblems
- Low-level optimizations and tuning
- Running on parallel computers
- Offline pre-computations (e.g. opening books, endgame databases)

Other Games where MCTS is Successful

CMPUT 496

- Hex - multiple World Champion MoHex developed in Prof. Hayward's group
- Amazons - most strong bots use MCTS
  - Interesting case - short 5 move simulations, then call an evaluation function works best
- General Game Playing - all strong programs use MCTS
- Havannah - best program by UofA graduate Timo Ewalds (now at DeepMind)
- Many more

Atari Games and General Game-Playing

CMPUT 496

- Everything we did was for a single game - Go
- AlphaGo is very specialized: cannot play on a different board size or even with a different komi
- Can we solve more general tasks with heuristic search?
- Automated planning is one example - one planner, many different planning problems
- Atari games is a famous example for learning to play many different games
  - ALE - Arcade Learning environment - simulator for Atari 2600 games console, developed in our dept.
  - Deep learning can learn to play many games from raw pixels, surpass human skills
- Many other learning platforms for developing "general AI", often using games

Implementation of Neural Networks

CMPUT 496

- Details, hands-on training and evaluation
- Algorithms for GPU and TPU
- Using software packages such as Tensorflow
- Creating, cleaning and preparing data for training and test
- Choices of architecture and input features

Finally...

CMPUT 496

- Thank you for taking 496 with me
- Thanks to the TA Chao, Chenjun, and Yunpeng for their work
- Please finish your quiz 13 and your USRI
- Good luck for all your final exams!

Parallel Heuristic Search

CMPUT 496

- Parallel MCTS on computer clusters
- How to scale to really big searches?
- Example: Fuego ran MCTS on 2000 cores on Hungabee system
- Example: AlphaGo Fan ran on 1,202 CPUs and 176 GPU, “similar hardware” plus TPU for AlphaGo Lee
- Example: ArvandHerd - our system for parallel planning, uses search, knowledge and simulations. Winner of the two most recent IPC planning competitions

Some Final Words of Wisdom...

CMPUT 496

- Many of you will graduate soon
- This is a time of unprecedented importance of AI, ML, heuristic search in real world applications
- 99.99% of users will not understand the potential and the limitations of the new technology. That is scary.
- Remember garbage in - garbage out principle
- Your responsibility is to use your understanding to be a leader in guiding applications of this technology

RL, AlphaGo and Beyond

Professor's Villains

Online Minion

Medium Website psychic, neutral evil

Armor Class Collectively 20
Hit Points 14 (1d10 + 9)
Speed 1 week

STR	DEX	CON	INT	WIS	CHA
29 (+9)	10 (+0)	17 (+3)	12 (+1)	11 (+0)	15 (+2)

Senses —
Languages Python
Challenge 1 (200 XP)

Coverage

Everything that is being discussed in class during the existence of the minion. Including the readings

Composition

Multiple Choice. There are anywhere between 10 - 20 MC question

Python Minion

Medium Website psychic, neutral evil

Armor Class Collectively 20
Hit Points 5
Speed $\tilde{3}$ week

STR	DEX	CON	INT	WIS	CHA
29 (+9)	10 (+0)	17 (+3)	12 (+1)	11 (+0)	15 (+2)

Senses —
Languages Python
Challenge 1 (200 XP)

Coverage

Everything that is being discussed in class during the existence of the minion. Including the readings

Composition

Programming. Python programming

Midterm Champion

Large paper psychic, neutral evil

Armor Class 25
Hit Points 50
Speed 90 minutes

STR	DEX	CON	INT	WIS	CHA
23 (+6)	14 (+2)	21 (+5)	16 (+3)	13 (+1)	20 (+5)

Senses —
Languages Python, Math
Challenge 8 (3,900 XP)

Coverage

Everything from the beginning to Lecture 10

Composition

Multiple Choice. Most of the exam is MC

Short Answer. Some short answer in a specific format

Final Foe

Gargantuan paper psychic, neutral evil

Armor Class 35
Hit Points ?
Speed 180 minutes

STR	DEX	CON	INT	WIS	CHA
30 (+10)	14 (+2)	29 (+9)	18 (+4)	17 (+3)	28 (+9)

Senses —
Languages Math, Python, something else
Challenge 23 (50,000 XP)

Coverage

Everything

Composition

?

Credits

General

- Created by u\DnD_Notes, March 2019
- Typesetting engine: LaTeX
- Dungeon and Dragon (5e) LaTeX Template

Art

- Owl for the cover art is from D&D Beyond
- Andy the D&D Ampersand is from Dungeon and Dragons
- Cover art formatting: Photoshop CC 2019

Disclaimer