Computing Science (CMPUT) 455

Search, Knowledge, and Simulations

Martin Müller

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

Fall 2022

455 Today - Lecture 18

- Machine Learning with simple features
- Minorisation-Maximisation for optimizing feature weights
- Using learned knowledge in UCT

Coursework

- Reading: Maddison et al., *Move Evaluation in Go using Deep Convolutional Neural Networks*
- Quiz 10: Machine learning intro; Learning with simple features. *(Double length)*
- Activities
- Python code `nn.py`, `nn3.py` - neural network and learning demos

Machine Learning with Simple Features

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

Simple Features

- 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,...)`

Simple Features Implementation in `Go4` and `Go5`

- 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

Basic Features

```
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
}
```

Distance Features

- 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)
  - $dx = 1, \quad dy = 2$
  - $d(dx, dy) = 1 + 2 + \max(1, 2) = 5$

Distance Metric Discussion

- 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 than Euclidean
  - Example: Euclidean distance
  - $d(2, 1) = sqrt(5) = 2.236....$

Types of Distance Features

- 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

Pass and Tactics

- 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

Pattern features

- 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

Evaluation Function from Simple Features

- 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) = $\sum w_i f_i$

Evaluation Function (2)

- 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) = $\prod_{f_i=1} w_i$

Move Prediction using Features

- 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

- 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

Overview of the Feature Learning Process

- 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

Game Data for 19 $\times$ 19 Go Move Prediction

- Which data to learn from?
  1. Games between professional players
     - Can get about 100,000 games
  2. Games between amateur players
     - Can get around 1 million games
  3. 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

Data for 7×7 Go Move Prediction

- 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

Getting Features from Game Data

Process for 19×19 Go:

- Foreach game g (tens of thousands of games)
- Foreach position p in game g ($\approx$150-300 per game)
- Foreach legal move m in position p ($\approx$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

Move Prediction as Classification Problem

Classification problem:

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

Move Prediction as Classification Problem

Classification problem:

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

Coulom's Feature Learning and Minorization/Maximization Algorithm

- 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

Represent Move as Group of Features

- 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

Main Advantage of Learning with Features

- 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

Feature Strength and Bradley-Terry Model

- 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) $= \frac{w_i}{w_i + w_j}$

Example

- f_1 = capture, $w_1 = 30.68$
- f_2 = extension, $w_2 = 11.37$

Example

- 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) = $11.37/(30.68 + 11.37) \approx 0.27$

Example

- 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...) = $30.68/(30.68 + 1.58) \approx 0.95$

From Single Features to Groups - Generalized Bradley-Terry Model

- A move has more than 1 feature (about 5-10 is typical)
  - Coulom refers to these combinations as "teams"
- 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

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

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

- 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) =

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

Comparing Multiple Moves

- Similarly, we can compare all legal moves in a Go position
- $P(\text{move } m_i \text{ wins}) = \frac{\text{strength}(m_i)}{\sum_{j \in legalmoves} \text{strength}(m_j)}$
- Assumptions:
  - Strength can be measured on totally ordered 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...

Learning Weights with Generalized Bradley-Terry Model

- 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)

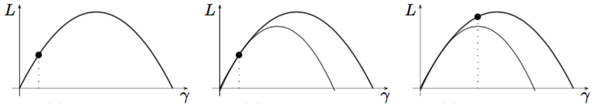
Learning Weights with Generalized Bradley-Terry Model

- 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)

Question: What do we mean by "move i beats move j"?

Minorization-Maximization (MM) Algorithm

- 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

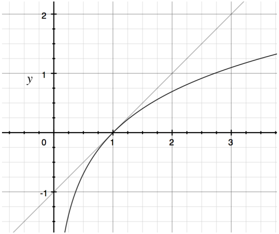


(a) Initial guess. (b) Minorization. (c) Maximization.

Fig. 1. Minorization-maximization.

Minorization-Maximization (MM) Algorithm

- 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 $1 - x \leq -logx$
- $1 - x$ minorizes $-logx$



Minorization-Maximization Iteration

- 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/`

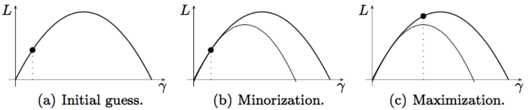


Fig. 1. Minorization-maximization.

Review - Summary of the Learning Process

- 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

How to use the Learned Model

Two main applications

1. In-tree knowledge for better move selection during MCTS
   - Three ideas:
   - Node initialization, additive knowledge, multiplicative knowledge
   - We'll cover these topics in the last part of these slides
2. Better probabilistic simulation policies
   - Lecture 14, Go4 program

From Move Weights to Move Probabilies

- 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 (1) - LFR

- 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

LFR Continued

- 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

Extensions to the MM Model (2) - FBT

- 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

Limits of Learning from Game Records

- 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?

Move Prediction - What to Expect?

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?

Strong Move Prediction vs Playing Well

- 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

Limits of Move Prediction

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

Equally Good Moves

- 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

Different Definitions of “Best” Move

- 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

- Discussed learning with simple features
- Coulom's approach:
  - Generalized Bradley-Terry model for strength of moves
  - MM algorithm for learning weights

Using Knowledge in UCT

- 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

How to Use Knowledge

Three ways:

1. Initialization of node statistics
2. Additive knowledge term
3. Multiplicative knowledge term

Decay Knowledge over Time

- At the beginning, we have only few simulations
  - Win rate $\hat{\mu}_i$ 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

1. Initialization of Node Statistics

- 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 imaginary initial “wins”
    - Give bad moves some imaginary initial “losses”

1. Initialization of Node Statistics (2)

- 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

2. Additive Knowledge

- 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: $1/(n_i + 1)$, $\sqrt{1/(n_i + 1)}$,...

3. Multiplicative Knowledge, Probabilistic UCT (PUCT)

- 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

- 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
  1. Initialization of node statistics by knowledge
  2. Additive term
  3. Multiplicative term, PUCT