Lecture 10: Classic Games

David Silver

Outline

Why Study Classic Games?

- Simple rules, deep concepts
- Studied for hundreds or thousands of years
- Meaningful IQ test
- *Drosophila* of artificial intelligence

e.g. chess game - very easy k study, learning lots of experiment, ...

- Microcosms encapsulating real world issues

simplified by very simple rules

- Games are fun!

AI in Games: State of the Art

This is the level of play achieved by AI.

Program	Level of Play	Program to Achieve Level
Checkers	Perfect	*Chinook*
Chess	Superhuman	*Deep Blue*
Othello	Superhuman	*Logistello*
Backgammon	Superhuman	*TD-Gammon*
Scrabble	Superhuman	*Maven*
Go	Grandmaster	*MoGo*[1], *Crazy Stone*[2], *Zen*[3]
Poker[4]	Superhuman	*Polaris*

"Perfect" means that it is totally solved.
"Superhuman" means that it defeated the world best human player.
...

[1] 9×9
[2] 9×9 and 19×19
[3] 19×19
[4] Heads-up Limit Texas Hold'em

RL in Games: State of the Art

Program	Level of Play	RL Program to Achieve Level
Checkers	Perfect	*Chinook*
Chess	International Master	*KnightCap* / *Meep*
Othello	Superhuman	*Logistello*
Backgammon	Superhuman	*TD-Gammon*
Scrabble	Superhuman	*Maven*
Go	Grandmaster	*MoGo*[1], *Crazy Stone*[2], *Zen*[3]
Poker[4]	Superhuman	*SmooCT*

What would the method be involved in reinforcement learning?
Success stories...

[1] 9×9
[2] 9×9 and 19×19
[3] 19×19
[4] Heads-up Limit Texas Hold'em

Optimality in Games

- What is the optimal policy π^i for *i*th player?
- If all other players fix their policies π^{-i}
- Best response $\pi^i_*(\pi^{-i})$ is optimal policy against those policies
- Nash equilibrium is a joint policy for all players

basically the main game theory concept

$$\pi^i = \pi^i_*(\pi^{-i})$$

- such that every player's policy is a best response
- i.e. no player would choose to deviate from Nash

Single-Agent and Self-Play Reinforcement Learning

- Best response is solution to single-agent RL problem
  - Other players become part of the environment
  - Game is reduced to an MDP
  - Best response is optimal policy for this MDP

The way to understand the Nash equilibrium is how could we find this by reinforcement learning.

- Nash equilibrium is fixed-point of self-play RL

Consider that one player can control all the other agents in the game.

  - Experience is generated by playing games between agents

$$a_1 \sim \pi^1, a_2 \sim \pi^2, \dots$$

  - Each agent learns best response to other players
  - One player's policy determines another player's environment
  - All players are adapting to each other

Q: is the fixed-point of Nash equilibrium unique?

Ans: In general, the answer is NO (poker game, ...), but for the classical game we considered, there is a unique Nash equilibrium.

Two-Player Zero-Sum Games

We will focus on a special class of games:

- A two-player game has two (alternating) players
  - We will name player 1 *white* and player 2 *black*
- A zero sum game has equal and opposite rewards for black and white

$$R^1 + R^2 = 0$$

We consider methods for finding Nash equilibria in these games

- Game tree search (i.e. planning) 바둑
- Self-play reinforcement learning

Perfect and Imperfect Information Games

- A perfect information or Markov game is fully observed
    - Chess
    - Checkers
    - Othello
    - Backgammon
    - Go
- An imperfect information game is partially observed
    - Scrabble
    - Poker
- We focus first on perfect information games

Minimax

- A value function defines the expected total reward given joint policies $\pi = \langle \pi^1, \pi^2 \rangle$

$$v_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right]$$

- A minimax value function maximizes white's expected return while minimizing black's expected return

This is like a fundamental concept of both game theory and in practice building any high performance game system (sustained by this tool).

$$v_*(s) = \max_{\pi^1} \min_{\pi^2} v_\pi(s)$$

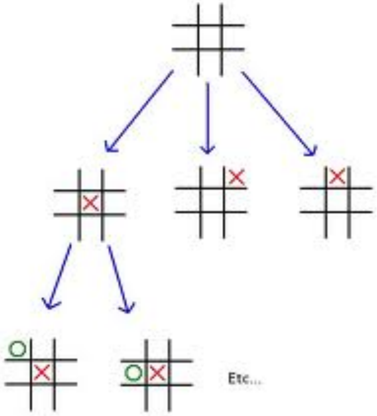
We really want to know what are these v_*s.
Again, just like usual, if we have the optimal value function, we're pretty much done in the game. This means that we know how to play the game optimally

- A minimax policy is a joint policy $\pi = \langle \pi^1, \pi^2 \rangle$ that achieves the minimax values
- There is a unique minimax value function
- A minimax policy is a Nash equilibrium

Minimax Search

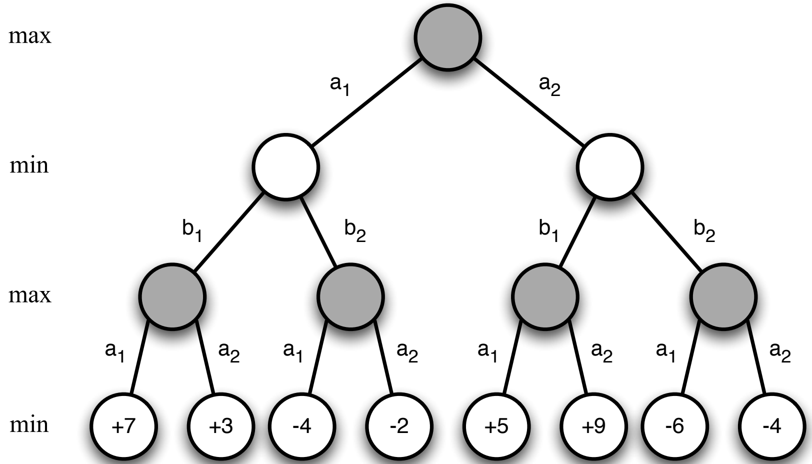
The way to do this is to build the familiar search tree.

- Minimax values can be found by depth-first game-tree search
- Introduced by Claude Shannon: *Programming a Computer for Playing Chess*
- Ran on paper!



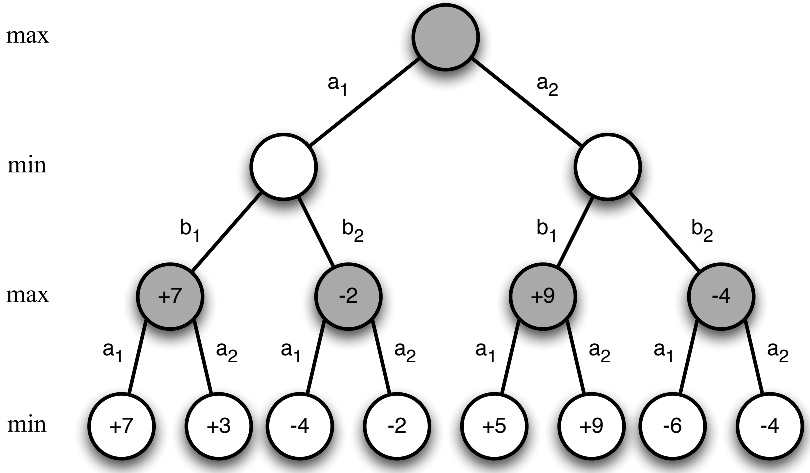
Minimax Search Example

two players

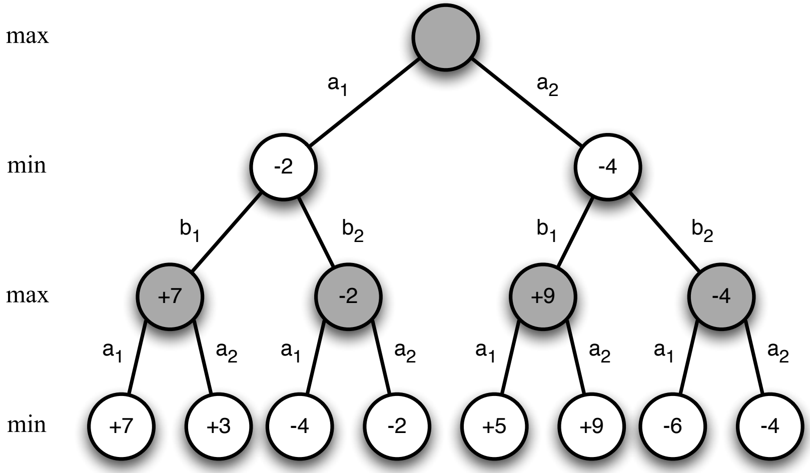


scores at the end of the game : the question is "what's the real score from the top node (root node)?"

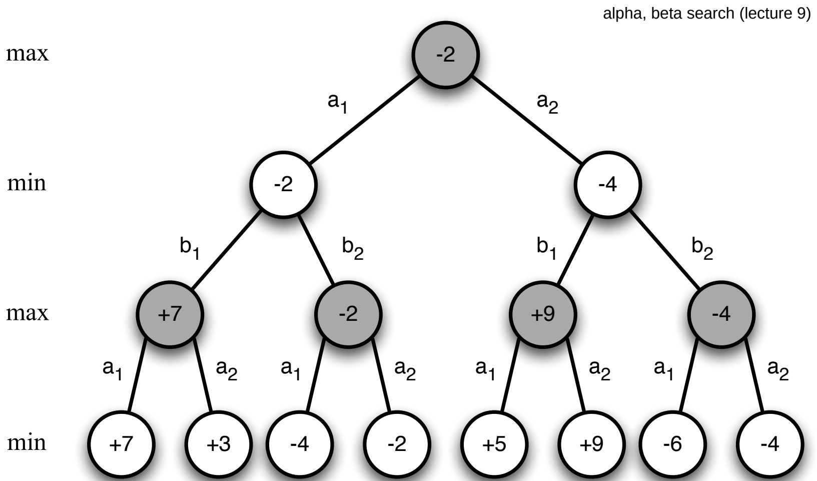
Minimax Search Example



Minimax Search Example



Minimax Search Example



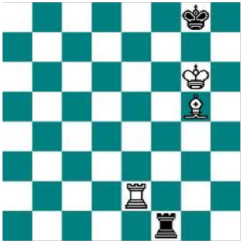
Value Function in Minimax Search

In practice, what we do is that we, instead of going all the way to the end of the game, we basically truncate the search after a small number of steps, and we use a value function to summarize the estimate the minimax value that node onwards.
So, if we have value function which is in game tree search refer to evaluation function or heuristic function, David think we in this course correctly understanded as value function, if we have that estimated minimax value, then what we can do is basically is to search down, consider the first actions we might take, build search tree over the first action, and then each of leaf node after three actions we truncate, and we have value functions at the end of node.

- Search tree grows exponentially
- Impractical to search to the end of the game
- Instead use value function approximator $v(s, \mathbf{w}) \approx v_*(s)$ linear combination features, neural network, ...
    - aka *evaluation function*, *heuristic function*
- Use value function to estimate minimax value at leaf nodes
- Minimax search run to fixed depth with respect to leaf values

Binary-Linear Value Function

- Binary feature vector $\mathbf{x}(\mathbf{s})$: e.g. one feature per piece
- Weight vector $\mathbf{w}$: e.g. value of each piece
- Position is evaluated by summing weights of active features



dot product

$$v(s,\mathbf{w}) = \mathbf{x}(s)\cdot\mathbf{w} = \begin{bmatrix}1\\1\\0\\1\\0\\0\\\vdots\end{bmatrix} \cdot \left[\begin{array}{c}+5\\+3\\+1\\-5\\-3\\-1\\\vdots\end{array}\right]\begin{array}{c}♖\\♗\\♙\\♜\\♝\\♟\\ \end{array}$$

weight vector

binary feature vector

We can make this feature vector very very large as we want.

$$v(s,\mathbf{w}) = 5+3-5 = 3$$

Deep Blue

Deep Blue is possibly still the most famous game play program.

- Knowledge
  - 8000 handcrafted chess features
  - Binary-linear value function
  - Weights largely hand-tuned by human experts
- Search
  - High performance parallel alpha-beta search
  - 480 special-purpose VLSI chess processors
  - Searched 200 million positions/second
  - Looked ahead 16-40 ply
- Results
  - Defeated human champion Garry Kasparov 4-2 (1997)
  - Most watched event in internet history

David emphasize the importance of background when we use RL (41:30~)

Chinook

It is really important to have that background (of search). When we talk about machine learning or reinforcement learning, David thinks that people have tendency to forget the importance of search and planning. There are domains, particularly the game domains.

- Knowledge algorithms achieving better performance. David thinks it's easy to go back into ...
  - Binary-linear value function
  - 21 knowledge-based features (position, mobility, ...)
  - x4 phases of the game
- Search
  - High performance alpha-beta search
  - Retrograde analysis
    - Search backward from won positions
    - Store all winning positions in lookup tables
    - Plays perfectly from last n checkers
- Results
  - Defeated Marion Tinsley in world championship 1994
    - won 2 games but Tinsley withdrew for health reasons
  - Chinook *solved* Checkers in 2007
    - perfect play against God

Self-Play Temporal-Difference Learning

Can we just apply these in a self-play context now?
the answer is YES, there is always no change necessary to take an RL algorithm and apply in a self-play context.
We just change the definition of the game to be self-play's. So, we estimate the return G_t and so force.

- Apply value-based RL algorithms to games of self-play
- MC: update value function towards the return G_t

$$\Delta \mathbf{w} = \alpha(G_t - v(S_t, \mathbf{w}))\nabla_{\mathbf{w}} v(S_t, \mathbf{w})$$

- TD(0): update value function towards successor value $v(S_{t+1})$

$$\Delta \mathbf{w} = \alpha(v(S_{t+1}, \mathbf{w}) - v(S_t, \mathbf{w}))\nabla_{\mathbf{w}} v(S_t, \mathbf{w})$$

- TD(λ): update value function towards the λ-return G_t^λ

function

$$\Delta \mathbf{w} = \alpha(G_t^\lambda - v(S_t, \mathbf{w}))\nabla_{\mathbf{w}} v(S_t, \mathbf{w})$$

So, all of our familiar machinery from previous reinforcement learning lectures is applicable in a self-play context.
We just try to estimate minimax values instead of the value functions. We're doing this in the setting when we improve the policy both player. So, the policy improvement process becomes policy improvement of both black and white. minimax & Nash equilibrium
But, function is still same algorithm you don't need to change the code from the single agent problem.

Policy Improvement with Afterstates

One thing which is often different is "afterstates".

- For deterministic games it is sufficient to estimate $v_*(s)$
- This is because we can efficiently evaluate the afterstate

which action is best

$$q_*(s,a) = v_*(succ(s,a))$$

All we need to do is to look at the successor position (state) we call that after state and evaluate the afterstates. And now, we just need to look at the afterstates of all the actions we can take and pick the one with highest value. So, we can do this because the game is deterministic. We have the successor function.

- Rules of the game define the successor state $succ(s,a)$
- Actions are selected e.g. by min/maximising afterstate value

slightly different part is that instead of just maximizing when we do a policy improvement step, we maximize for white and minimize for black. So, this is just like minimax search tree.

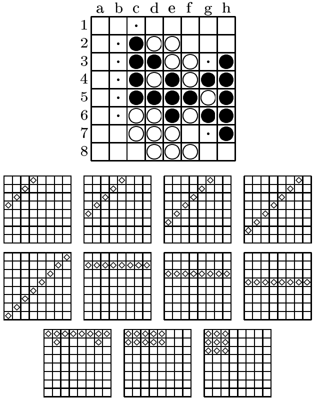
$$A_t = \operatorname*{argmax}_a v_*(succ(S_t,a)) \qquad \text{for white}$$

$$A_t = \operatorname*{argmin}_a v_*(succ(S_t,a)) \qquad \text{for black}$$

- This improves joint policy for both players

Self-Play TD in Othello: *Logistello*

- Logistello created its own features
- Start with raw input features, e.g. "black stone at C1?"
- Construct new features by conjunction/disjunction binary feature vector
- Created 1.5 million features in different configurations
- Binary-linear value function using these features



Reinforcement Learning in Logistello

Logistello used generalised policy iteration

Logistello plays itself few thousand of games using minimax search tree.

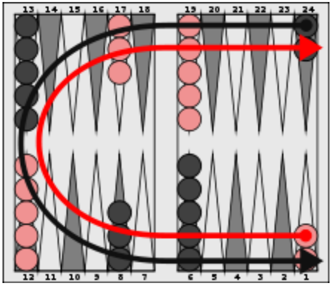
- Generate batch of self-play games from current policy
- Evaluate policies using Monte-Carlo (regress to outcomes)
- Greedy policy improvement to generate new players

Results

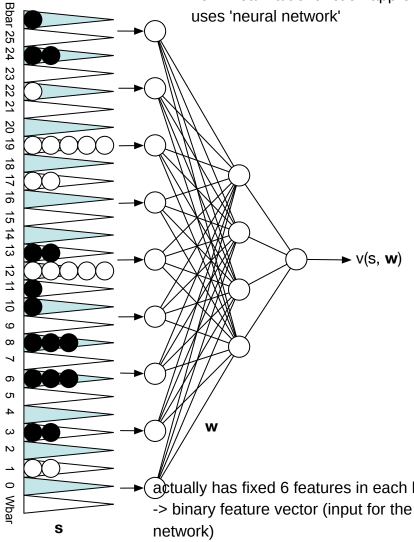
- Defeated World Champion Takeshi Murukami 6-0

TD Gammon: Non-Linear Value Function Approximation

Self-play RL



non-linear value function approximators uses 'neural network'


Bbar 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1 0 Wbar
v(s, **w**)
w
s

actually has fixed 6 features in each locations -> binary feature vector (input for the neural network)

Self-Play TD in Backgammon: *TD-Gammon*

- Initialised with random weights
- Trained by games of self-play
- Using non-linear temporal-difference learning

$$\delta_t = v(S_{t+1}, \mathbf{w}) - v(S_t, \mathbf{w})$$ TD error

$$\Delta \mathbf{w} = \alpha \delta_t \nabla_{\mathbf{w}} v(S_t, \mathbf{w})$$

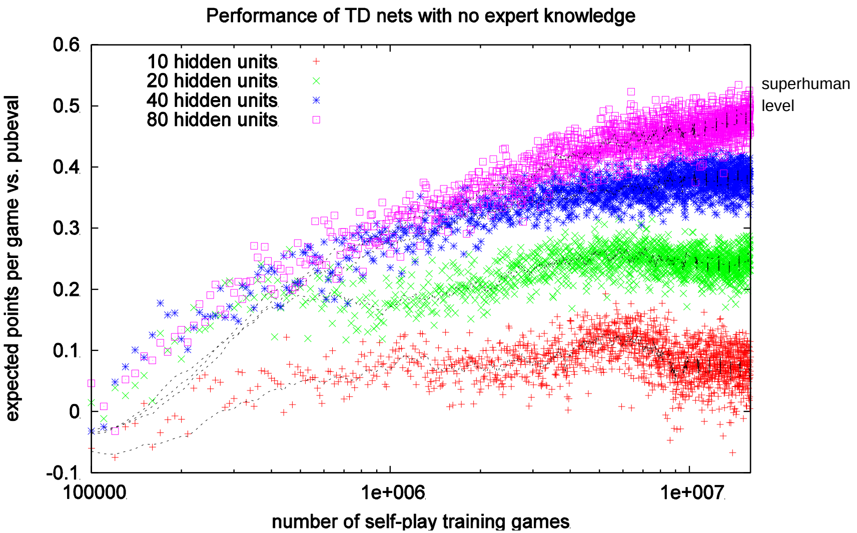
- Greedy policy improvement (no exploration)
- Algorithm always converged in practice to very effective solution
- Not true for other games

Why does this converge well?
- Backgammon has stochasticity.
In this kind of situation, it's not strictly necessary to explore.
Dice has stochasticity -> this gives randomness and naturally it has smooth value function which is much more stable in value function approximation.

TD Gammon: Results

- Zero expert knowledge $\implies$ strong intermediate play
- Hand-crafted features $\implies$ advanced level of play (1991)
- 2-ply search $\implies$ strong master play (1993)
- 3-ply search $\implies$ superhuman play (1998)
- Defeated world champion Luigi Villa 7-1 (1992)

New TD-Gammon Results

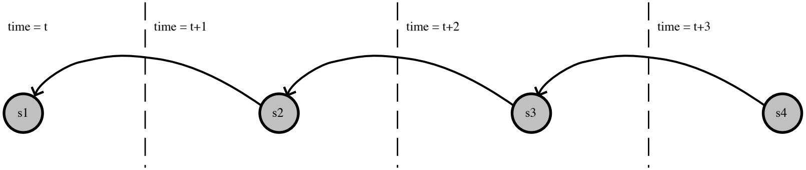


Simple TD

Now, we're going to focus on more sophisticated mixture of reinforcement learning research.

Let's start with sort of the most familiar method, basic TD algorithm that we called simple TD.

- TD: update value towards successor value



- Value function approximator $v(s, \mathbf{w})$ with parameters $\mathbf{w}$
- Value function backed up from raw value at next state

$$v(S_t, \mathbf{w}) \leftarrow v(S_{t+1}, \mathbf{w})$$

- First learn value function by TD learning
- Then use value function in minimax search (no learning)

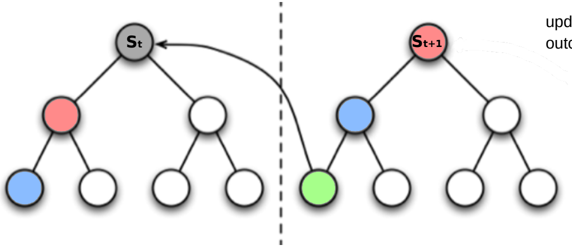
$$v_+(S_t, \mathbf{w}) = \operatorname*{minimax}_{s \in leaves(S_t)} v(s, \mathbf{w})$$

Simple TD: Results

- Othello: superhuman performance in *Logistello*
- Backgammon: superhuman performance in *TD-Gammon*
- Chess: ~~poor performance~~
- Checkers: ~~poor performance~~
- In chess tactics seem necessary to find signal in position
- e.g. hard to find checkmates without search
- Can we learn directly from minimax search values?

TD Root

- TD root: update value towards successor search value



update value functions towards the outcome deep search

- Search value is computed at root position S_t

$$v_+(S_t, \mathbf{w}) = \underset{s \in leaves(S_t)}{\text{minimax}}\ v(s, \mathbf{w})$$

- Value function backed up from *search value* at next state

$$v(S_t, \mathbf{w}) \leftarrow v_+(S_{t+1}, \mathbf{w}) = v(l_+(S_{t+1}), \mathbf{w})$$

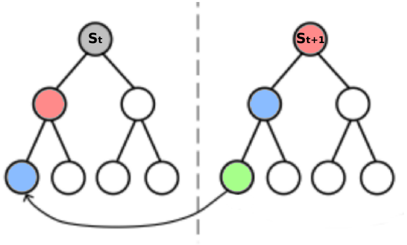
- Where $l_+(s)$ is the leaf node achieving minimax value from s

TD Root in Checkers: *Samuel's Player*

- First ever TD learning algorithm (*Samuel 1959*)
- Applied to a Checkers program that learned by self-play
- Defeated an amateur human player
- Also used other ideas we might now consider strange

TD Leaf

- TD leaf: update search value towards successor search value



- Search value computed at current and next step

$$v_+(S_t, \mathbf{w}) = \underset{s \in leaves(S_t)}{\text{minimax}}\, v(s, \mathbf{w}), \quad v_+(S_{t+1}, \mathbf{w}) = \underset{s \in leaves(S_{t+1})}{\text{minimax}}\, v(s, \mathbf{w})$$

- Search value at step t backed up from *search value* at $t+1$

$$v_+(S_t, \mathbf{w}) \leftarrow v_+(S_{t+1}, \mathbf{w})$$
$$\implies v(l_+(S_t), \mathbf{w}) \leftarrow v(l_+(S_{t+1}), \mathbf{w})$$

TD leaf in Chess: *Knightcap*

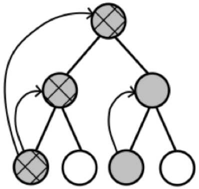
- Learning
  - *Knightcap* trained against expert opponent
  - Starting from standard piece values only
  - Learnt weights using TD leaf
- Search
  - Alpha-beta search with standard enhancements
- Results
  - Achieved master level play after a small number of games
  - Was not effective in self-play
  - Was not effective without starting from good weights

TD leaf in Checkers: *Chinook*

- Original Chinook used hand-tuned weights
- Later version was trained by self-play
- Using TD leaf to adjust weights
    - Except material weights which were kept fixed
- Self-play weights performed $\geq$ hand-tuned weights
- i.e. learning to play at superhuman level

TreeStrap

- TreeStrap: update search values towards deeper search values



- Minimax search value computed at *all* nodes $s \in nodes(S_t)$
- Value backed up from search value, at same step, for all nodes

$$\begin{aligned} & v(s, \mathbf{w}) \leftarrow v_+(s, \mathbf{w}) \\ \Longrightarrow \; & v(s, \mathbf{w}) \leftarrow v(l_+(s), \mathbf{w}) \end{aligned}$$

Treestrap in Chess: *Meep*

- Binary linear value function with 2000 features
- Starting from random initial weights (no prior knowledge)
- Weights adjusted by TreeStrap
- Won 13/15 vs. international masters
- Effective in self-play
- Effective from random initial weights

Simulation-Based Search

- Self-play reinforcement learning can replace search
- Simulate games of self-play from root state S_t
- Apply RL to simulated experience
    - Monte-Carlo Control $\implies$ Monte-Carlo Tree Search
    - Most effective variant is UCT algorithm
        - Balance exploration/exploitation in each node using UCB
    - Self-play UCT converges on minimax values

    - Perfect information, zero-sum, 2-player games
    - Imperfect information: see next section

Performance of MCTS in Games

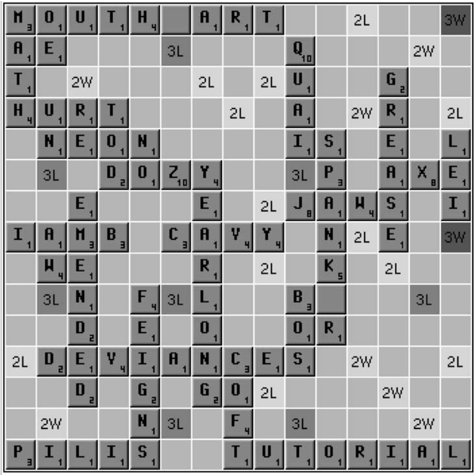
- MCTS is best performing method in many challenging games
  - Go (last lecture)
  - Hex
  - Lines of Action
  - Amazons
- In many games simple Monte-Carlo search is enough
  - Scrabble
  - Backgammon

Simple Monte-Carlo Search in Maven

- Learning
  - Maven evaluates moves by $score + v(rack)$
  - Binary-linear value function of rack
  - Using one, two and three letter features
  - Q??????, QU?????, III????
  - Learnt by Monte-Carlo policy iteration (cf. Logistello)
- Search
  - Roll-out moves by imagining n steps of self-play
  - Evaluate resulting position by $score + v(rack)$
  - Score move by average evaluation in rollouts
  - Select and play highest scoring move
  - Specialised endgame search using B*

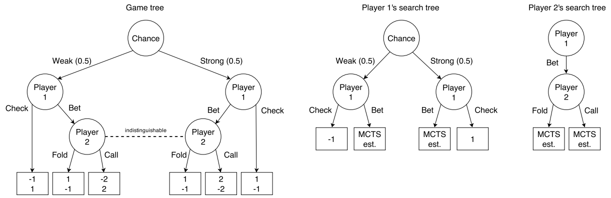
Maven: Results

- Maven beat world champion Adam Logan 9-5
- Here Maven predicted endgame to finish with MOUTHPART
- Analysis showed Maven had error rate of 3 points per game



Game-Tree Search in Imperfect Information Games

- Players have different information states and therefore separate search trees



- There is one node for each information state
  - summarising what a player knows
  - e.g. the cards they have seen
- Many real states may share the same information state
- May also aggregate states e.g. with similar value

Solution Methods for Imperfect Information Games

Information-state game tree may be solved by:

- Iterative forward-search methods
  - e.g. Counterfactual regret minimization
  - “Perfect” play in Poker (heads-up limit Hold’em)
- Self-play reinforcement learning
- e.g. Smooth UCT
  - 3 silver medals in two- and three-player Poker (limit Hold’em)
  - Outperformed massive-scale forward-search agents

Smooth UCT Search

- Apply MCTS to information-state game tree
- Variant of UCT, inspired by game-theoretic Fictitious Play
  - Agents learn against and respond to opponents' average behaviour
- Extract average strategy from nodes' action counts, $\pi_{avg}(a|s) = \frac{N(s,a)}{N(s)}$.
- At each node, pick actions according to

$$A \sim \begin{cases} \text{UCT}(S), & \text{with probability } \eta \\ \pi_{avg}(\cdot|S), & \text{with probability } 1 - \eta \end{cases}$$

- Empirically, in variants of Poker:
  - Naive MCTS diverged
  - Smooth UCT converged to Nash equilibrium

RL in Games: A Successful Recipe

Program	Input features	Value Fn	RL	Training	Search
Chess *Meep*	Binary *Pieces, pawns, ...*	Linear	TreeStrap	Self-Play / Expert	$\alpha\beta$
Checkers *Chinook*	Binary *Pieces, ...*	Linear	TD leaf	Self-Play	$\alpha\beta$
Othello *Logistello*	Binary *Disc configs*	Linear	MC	Self-Play	$\alpha\beta$
Backgammon *TD Gammon*	Binary *Num checkers*	Neural network	TD(λ)	Self-Play	$\alpha\beta$ / MC
Go *MoGo*	Binary *Stone patterns*	Linear	TD	Self-Play	MCTS
Scrabble *Maven*	Binary *Letters on rack*	Linear	MC	Self-Play	MC search
Limit Hold'em *SmooCT*	Binary *Card abstraction*	Linear	MCTS	Self-Play	-