CSE 592: Foundations of AI — University of Michigan

# Lecture 18: Games



Gabriel Poesia
Assistant Professor, CSE
University of Michigan, Ann Arbor
poesia@umich.edu
https://gpoesia.com

# Plan

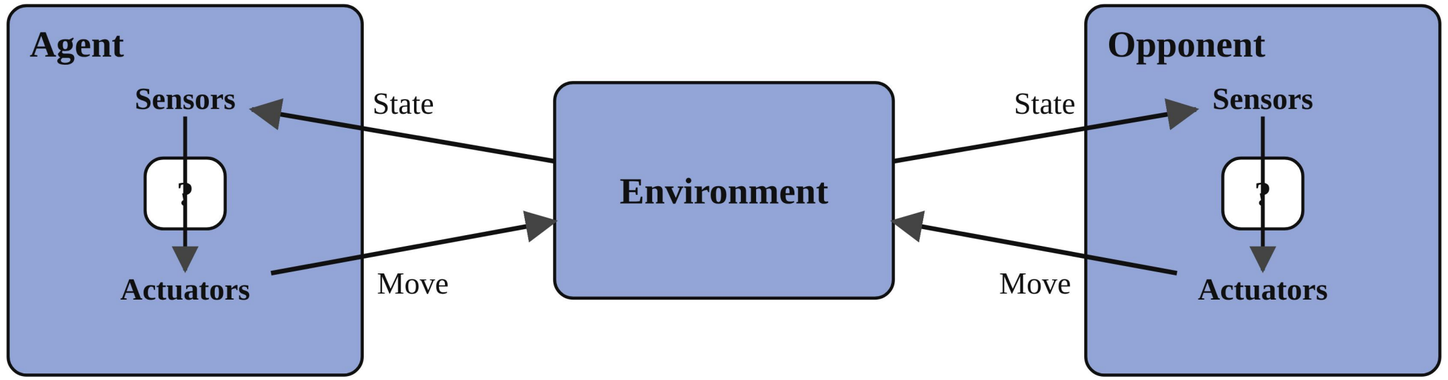
1. Games
2. Minimax
3. Alpha–beta pruning
4. Cutting off search
5. Monte Carlo tree search
6. Learning to search

# Where we left off

> 🧠 **Recall**
> Lecture 1: environments differ in several ways. One of them: single-agent vs. multi-agent.

So far: one agent. The environment may be stochastic.

Today: **another agent** in competition with us.




Still fully observable and deterministic. Players take turns.

New problem: we don't know opponent's moves.

But different from stochastic: they are trying to win!

# Games in AI



Source: IBM

Games have been AI benchmarks from the start.

Easy to state, hard to play well.

1950: Shannon describes how a computer could play chess.

1950s: Samuel's checkers program **learns** to evaluate positions by playing against itself.

1997: Deep Blue beats Kasparov at chess.

2016: AlphaGo beats Lee Sedol at Go.

2017: AlphaGo Zero beats AlphaGo, learning from scratch.

2017: AlphaZero beats both AlphaGo Zero and AlphaGo, learning from scratch.

Today: the algorithms behind each of these.

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# Games

# Two-player zero-sum games

Today's games:

- **Two players**, taking turns
- **Perfect information**
- **Deterministic**
- **Zero-sum**

| **Examples** | **Non-examples** |
| --- | --- |
| Tic-tac-toe, Connect 4, checkers, chess, Go, Hex | Poker, backgammon, football |

Some non-examples covered in the book (Ch 5.5-5.6) if you're curious.

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# Formalizing a game

Two players: **MAX** (moves first) and **MIN**.

**Definition (Game)**

- $s_0$: initial state
- $\text{To-Move}(s)$: whose turn it is in $s$
- $\text{Actions}(s)$: legal moves in $s$
- $\text{Result}(s, a)$: the state after playing $a$ in $s$
- $\text{Is-Terminal}(s)$: is the game over?
- $\text{Utility}(s)$: final score of terminal $s$, from MAX's point of view

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# Why not just search?

> 🧠 **Recall**
> Lecture 2: search returns a **plan**, a sequence of actions that reaches a goal.

Q: Can we search for a sequence of moves that ends in a win?

A: No, half of the moves aren't ours!

Instead, we need a **strategy**: our move for every possible reply.

Like a policy in RL: action given state.

Should assume the opponent is trying to beat us.

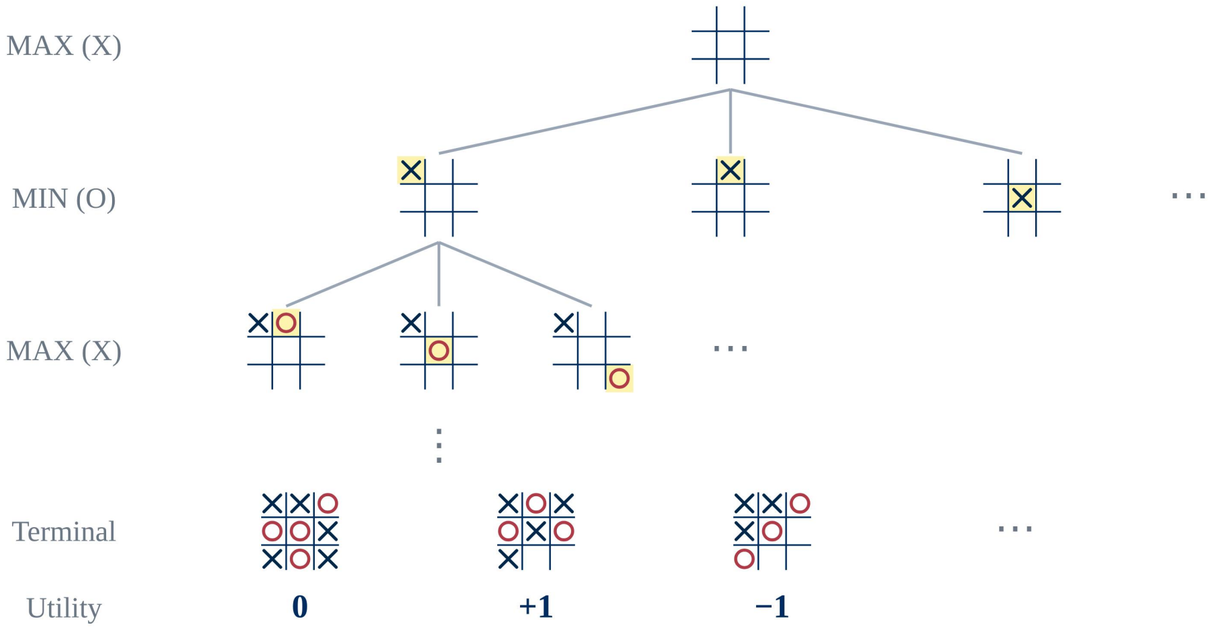
Will find a strategy via search, just different.

# Game trees

Our key data structure here will be the **game tree**.

Nodes: states. Edges: moves. Levels alternate between MAX and MIN.

Example: Tic-tac-toe, X playing first.




255,168 possible games, but only 5,478 distinct positions.

# Minimax

# Minimax value

How do we play optimally against an unknown opponent?

Idea: assume the opponent also plays optimally.

We'll assign a value to each state in the game tree.

For terminal states, the value is a "utility": e.g. 1 for MAX wins, -1 for MIN wins, 0 for tie.

For non-terminal states, we optimize over possible moves:

$$\text{Minimax}(s) = \begin{cases} \text{Utility}(s) & \text{if } s \text{ is terminal} \\ \max_a \ \text{Minimax}(\text{Result}(s,a)) & \text{if MAX to move} \\ \min_a \ \text{Minimax}(\text{Result}(s,a)) & \text{if MIN to move} \end{cases}$$

Result: What MAX gets if both play perfectly from $s$.

Strategy: MAX picks a move with the highest value. MIN, the lowest.

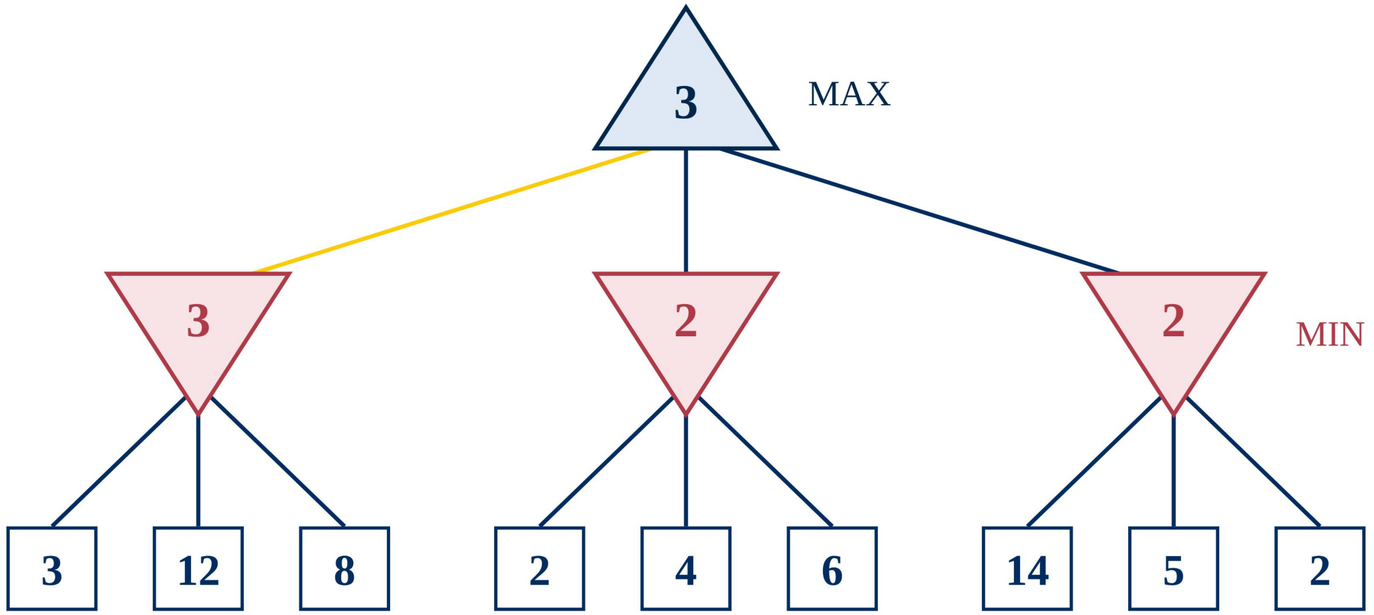
Somewhat similar to the Bellman optimality equations! But $\max$ / $\min$ instead of $\mathbb{E}$.

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# A two-ply game

Suppose a simple game with just one move per player: MAX, then MIN, then game ends.

Game tree might look like this:




Q: What will MIN do at each node?

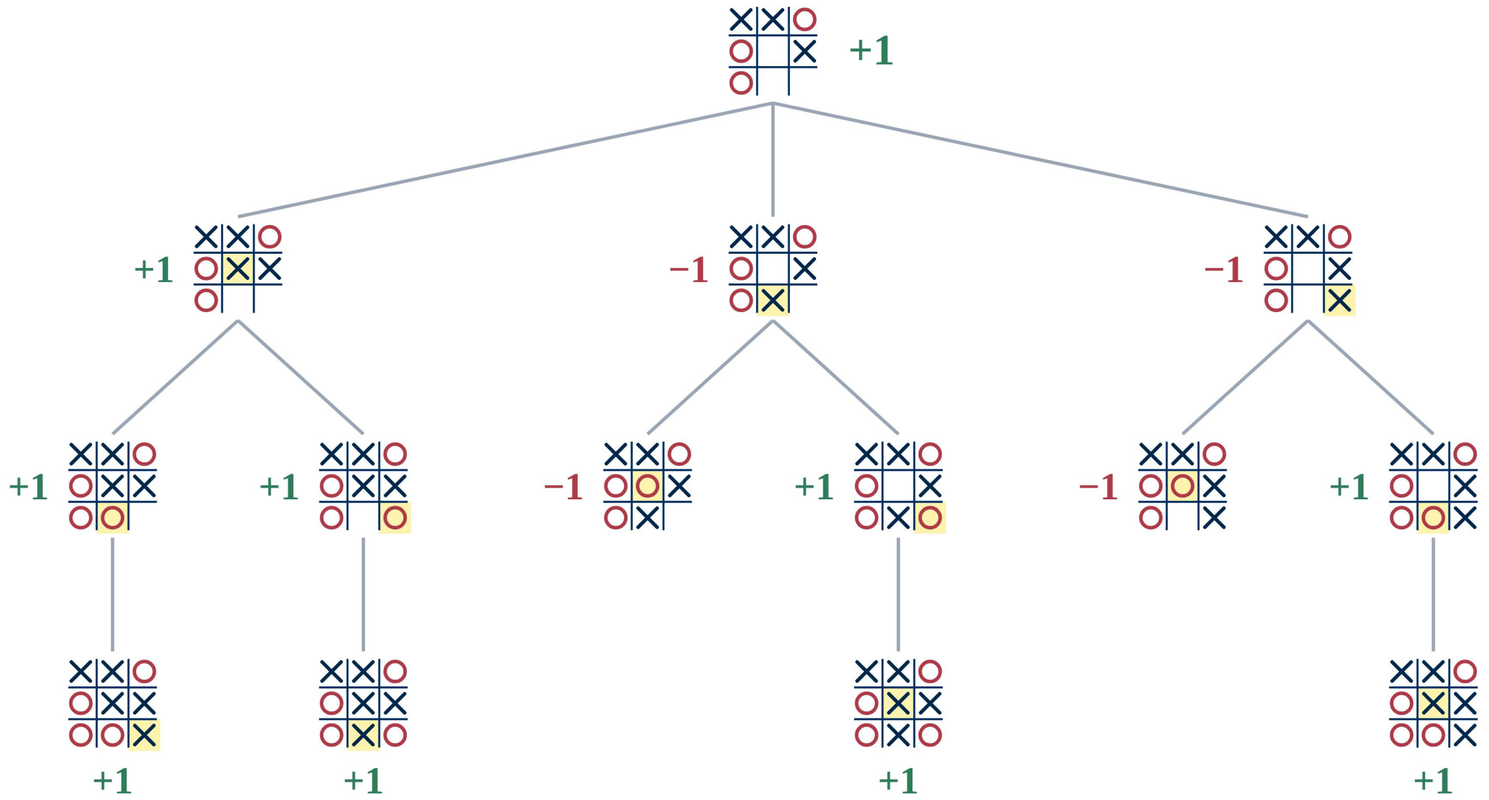
Q: And MAX?

Note: MAX doesn't go for the 14. No "optimism in the face of uncertainty" here!

# Tic-tac-toe

Games might not end in one move, but same idea generalizes.

Example: tic-tac-toe, X to move. Q: Where should X play?



# Minimax search

We can use this idea to compute minimax values exactly, + optimal move.

Modified depth-first search over the game tree:

```
def minimax(s):
    if is_terminal(s): return utility(s)
    values = [minimax(result(s, a)) for a in actions(s)]
    if to_move(s) == MAX: return max(values)
    else: return min(values)

def best_move(s):   # s: MAX to move
    return argmax(actions(s), key=lambda a: minimax(result(s, a)))
```

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# Solving a game

What if we compute the minimax value of the initial position of a game?

That's called **solving** the game: answering what happens under perfect play.

For tic-tac-toe, minimax visits 549,946 nodes.

Value: 0, so optimal play $\rightarrow$ draw.

Checkers solved in 2007: value also 0.

Chess and Go are still theoretically open.

# Is minimax optimal?

Against an optimal opponent: yes. Best guaranteed outcome.

Q: What if the opponent makes mistakes?

A: We still get **at least** the minimax value.

Modeling choice: assume the worst-case opponent.

Assumes no computational constraints, so not necessarily realistic

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# How expensive?

Minimax explores the whole tree, depth-first.

Assume $b$ moves per position, games last $m$ moves. Time complexity $O(b^m)$, memory $O(bm)$.

| Game | $b$ | $m$ | $b^m$ |
|---|---|---|---|
| Tic-tac-toe | $\leq 9$ | $\leq 9$ | $< 10^6$ |
| Chess | $\approx 35$ | $\approx 80$ | $\approx 10^{123}$ |
| Go | $\approx 250$ | $\approx 150$ | $\approx 10^{360}$ |

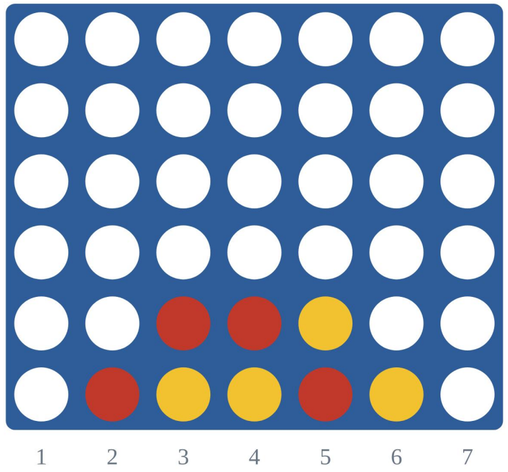
Problem: exact minimax is hopeless for interesting games.

Two ideas next: skip parts of the tree **exactly** (pruning), and stop early with an **estimate**.

Silver et al. Mastering the game of Go with deep neural networks and tree search. Nature, 2016.

# Alpha–beta pruning

# Connect 4




New game so we have bigger trees: Connect 4.

7 columns. Move: drop a piece on a column.

Four in a row ("connected") wins (any direction).

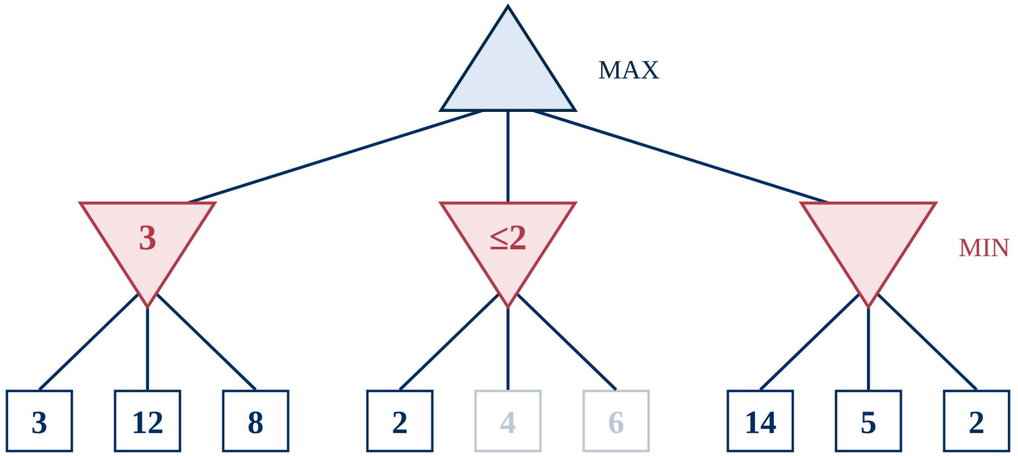
At most 7 moves per turn, up to 42 moves.

$\approx 4.5 \times 10^{12}$ positions.

Solved in 1988: the first player wins, by starting in the center column.

Allis. A knowledge-based approach of Connect-Four, 1988; position count: J. Tromp.

# Pruning leaves




Say we look at leaves left to right. First MIN node: $3$. So MAX can get **at least** $3$.

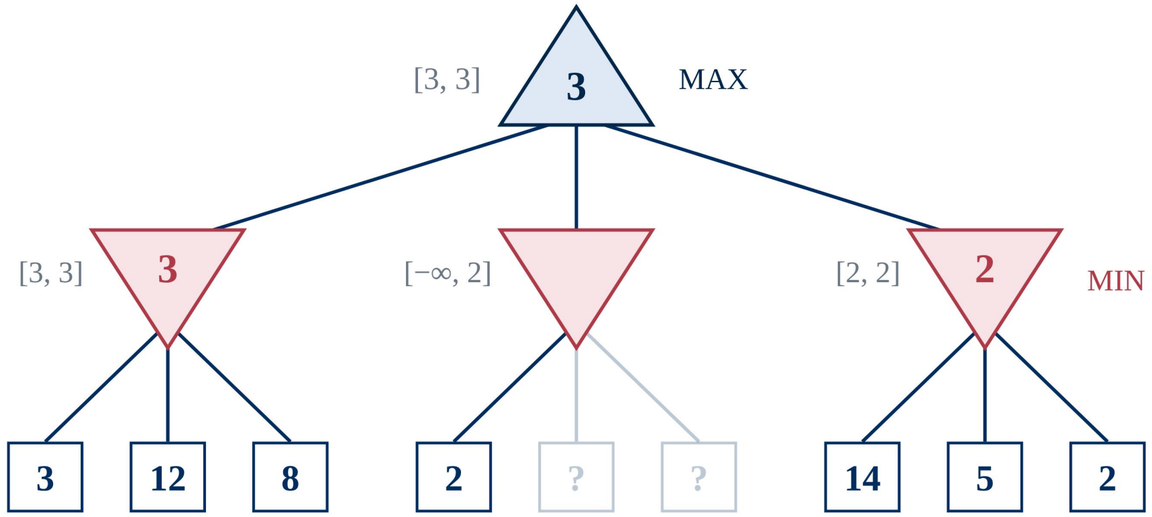
Second MIN node: first leaf is $2$. So this node is worth **at most** $2$.

Q: Do we need to look at the $4$ and the $6$?

A: No! MAX already has $3$ elsewhere.

$$\max(\min(3, 12, 8),\ \min(2, x, y),\ \min(14, 5, 2)) = \max(3,\ z,\ 2) = 3 \quad (z \leq 2)$$

# Value bounds




Let's track the range of values each node could still have.

MAX is guaranteed $\geq 3$.

Middle node $\leq 2$: prune.

Same value as full minimax.

# α and β

That's the key idea in alpha-beta pruning:

$\alpha$: the best value MAX can already guarantee on this path. (Lower bound)

$\beta$: the best value MIN can already guarantee on this path. (Upper bound)

Only values in $(\alpha, \beta)$ can still affect the root.

At a MIN node: once $v \leq \alpha$, stop. MAX above will never choose this node.

At a MAX node: once $v \geq \beta$, stop. MIN above will never choose this node.

# Alpha–beta search

We can keep and update these bounds down the tree, prune when we can.

That's the idea in **alpha-beta search**:

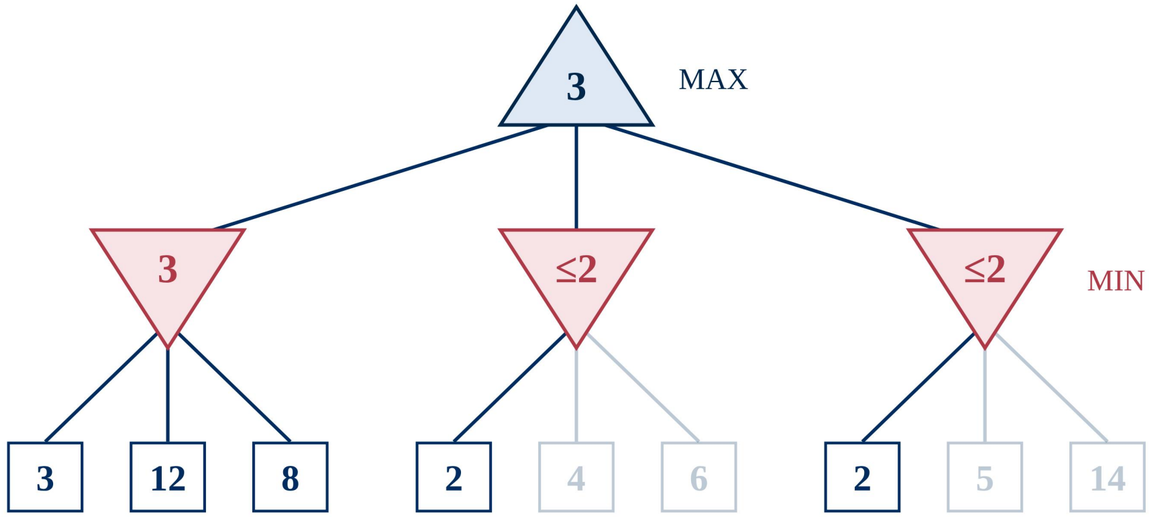
```
def max_value(s, α, β):
   if is_terminal(s): return utility(s)
   v = -∞
   for a in actions(s):
      v = max(v, min_value(result(s, a), α, β))
      if v ≥ β: return v      # MIN above won't allow this
      α = max(α, v)
   return v

def min_value(s, α, β):
   if is_terminal(s): return utility(s)
   v = +∞
   for a in actions(s):
      v = min(v, max_value(result(s, a), α, β))
      if v ≤ α: return v      # MAX above won't allow this
      β = min(β, v)
   return v
```

Start with `max_value(s0, -∞, +∞)`. Same root value as minimax.

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# Move ordering




Q: In the right MIN node, what if we had seen 2 first?

A: We'd have pruned!

How much we prune depends on the order we try moves.

Best case (best moves first): $O(b^{m/2})$ instead of $O(b^m)$.

Random order: roughly $O(b^{3m/4})$.

Like in CSPs, we can use move order heuristics

# Ordering in practice

Some experiments on Connect 4, max depth 7 , with an evaluation heuristic (we'll see this next):

| Search | Columns tried in order | Nodes visited |
|---|---|---|
| Minimax | any | 960,793 |
| Alpha–beta | edges first: 1, 7, 2, 6, 3, 5, 4 | 253,158 |
| Alpha–beta | left to right: 1, 2, ..., 7 | 56,551 |
| Alpha–beta | center first: 4, 3, 5, 2, 6, 1, 7 | 6,256 |

All four return the same value.

Tic-tac-toe (whole game): 549,946 → 18,297 nodes; center and corners first: 7,275 .

Node counts and MCTS runs: raw/cse592/lec18/games.py.

# Finding good orders

Can also use several domain-general heuristics.

**Iterative deepening**: search to depth 1, 2, 3, ... Try the previous iteration's best move first.

**Killer moves**: moves that caused cutoffs elsewhere in the tree. Try them early.

**Transposition table**: same position, different move orders. Just memoize values.

# Cutting off search

# Still too deep

Chess with perfect ordering: $35^{80/2} \approx 10^{62}$. Still hopeless.

Idea (Shannon!): stop at depth $d$, and **estimate** who's winning.

```
1   def h_minimax(s, d):
2       if is_terminal(s): return utility(s)
3       if d == 0: return eval(s)       # cutoff
4       values = [h_minimax(result(s, a), d - 1) for a in actions(s)]
5       return max(values) if to_move(s) == MAX else min(values)
```

$\text{Eval}(s)$: **evaluation function**, estimating the minimax value of $s$.

Same change for alpha–beta.

Caveat: not exact anymore, depends on how good is $\text{Eval}$.

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# Evaluation functions

What makes a good $\mathrm{Eval}$?

Ideally agrees with the utility on terminal states.

Ideally fast.

Ideally higher when the chances of winning are higher.

Common design: weighted sum of features.

$$\mathrm{Eval}(s) = w_1 f_1(s) + w_2 f_2(s) + \cdots + w_n f_n(s)$$

E.g., chess: count material. Pawn $1$, knight / bishop $3$, rook $5$, queen $9$ (our minus opponent).

Just a linear model on features!

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# Evaluation = value function

> 🧠 **Recall**
> Value functions $V(s)$, the value of a state (expected return).

$\mathrm{Eval}(s)$ plays the same role: how good is $s$ for us?

Q: Can we **learn** it instead of hand-tuning the weights?

A: Yes! Samuel's checkers (1959) adjusted its weights by playing against itself.

TD-Gammon (1990s): a neural network $V(s)$ for backgammon, trained with TD learning from self-play.

Reached the level of top human players.

Tesauro. Temporal difference learning and TD-Gammon. Commun. ACM, 1995.

# Search vs. lookup

Sometimes we don't need to search at all.

Openings: well-studied lines, stored in an **opening book**.

Endgames: solve exactly, searching backwards from checkmates.

E.g., in chess: all positions with up to 7 pieces are solved and stored in tables.

Can use search + Eval for the middle of the game.

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# Deep Blue




Source: Wikipedia, Deep Blue vs Garry Kasparov

1997: Deep Blue beats world champion Garry Kasparov, best of 7.

Alpha–beta search, with many extensions.

Hand-tuned evaluation function.

Custom chess chips to search fast.

Opening book and endgame tables.

Everything in this section, engineered at scale.

Huge milestone in AI!

But not the last board game milestone: Go was still out of reach.

Campbell, Hoane & Hsu. Deep Blue. Artificial Intelligence, 2002.

# Monte Carlo tree search

# Go

$19 \times 19$ board. $b \approx 250$, games of $\approx 150$ moves.

Alpha–beta can't go deep with $b \approx 250$.

And it's hard to write a good $\mathrm{Eval}$.

Q: Can we estimate how good a position is, without an evaluation function?

Silver et al. Mastering the game of Go with deep neural networks and tree search. Nature, 2016.

# Playouts

Idea: play the game to the end with random moves. Many times.

Win rate over these **playouts**: estimate of how good the position is.

> 🧠 **Recall**
>
> Monte Carlo evaluation: estimate a value by averaging sampled returns!

Pure Monte Carlo search: for each legal move, run $N$ playouts. Play the best win rate.

Caveat: it's the win rate for random players, not for good ones.

Q: How should we divide our playouts between moves? Uniformly?

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# Bandits strike back

A: No. Would waste playouts on "obviously" bad moves.

> 💡 **Insight**
> We can see each move as an arm in a bandit!

Each playout is like a pull, with reward 1 (win) or 0 (loss).

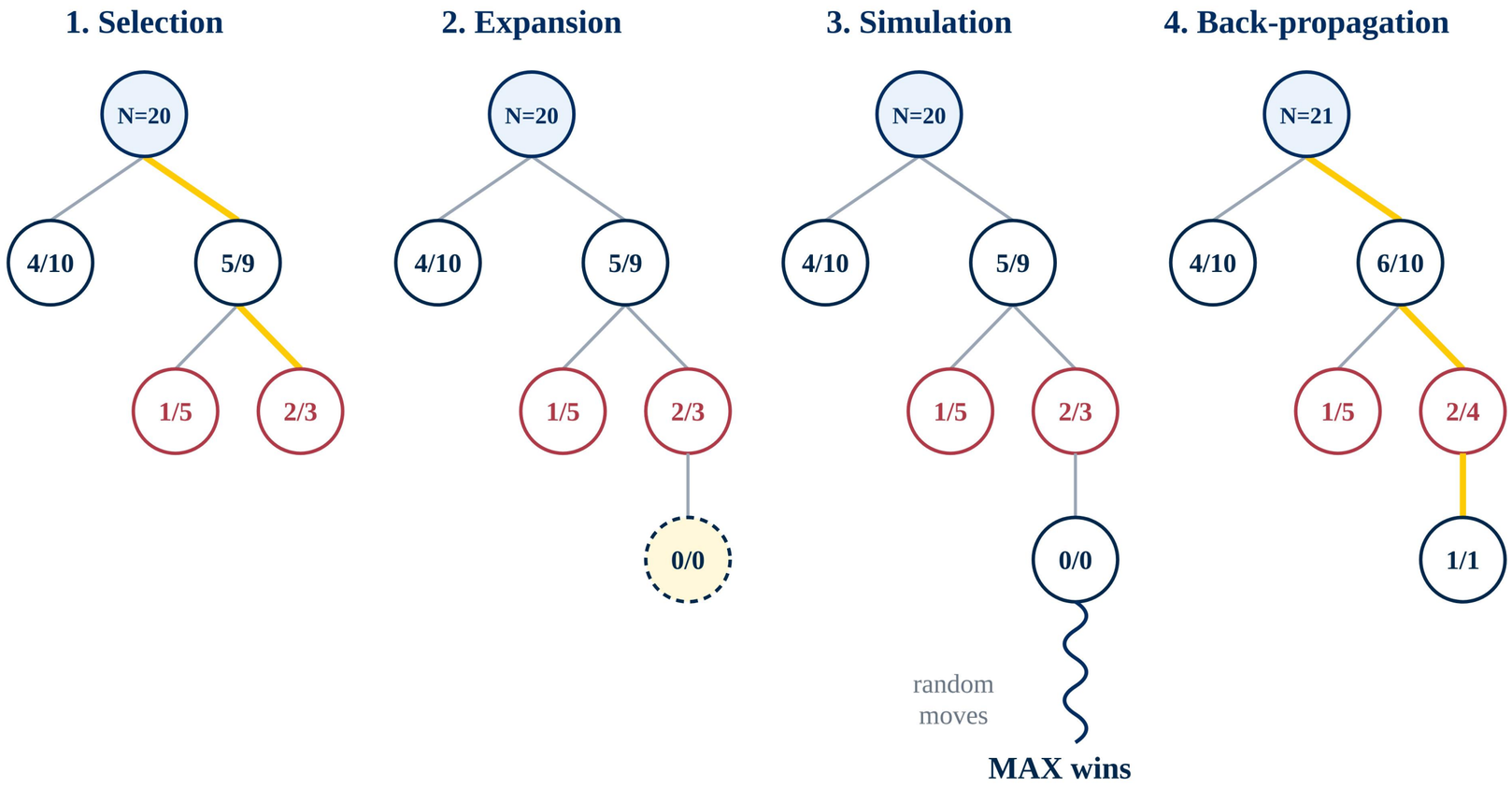
> 🧠 **Recall (UCB1)**
> UCB1: pull the arm with the best average + bonus.
>
> $$A_t = \arg\max_a \left[ \hat{\mu}_t(a) + \sqrt{\frac{2 \log t}{N_t(a)}} \right]$$

Spend playouts on moves that look good, or that we're unsure about.

Problem: after the first move, the playout is still random.

Idea: do this at every node we visit often. A tree of bandits!

# Monte Carlo tree search




Each node: wins / visits, for the player who made the move into it.

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# UCT

Selection rule: UCB applied to trees (**UCT**).

$$\text{UCB1}(n) = \underbrace{\frac{W(n)}{N(n)}}_{\text{win rate}} + \underbrace{C\sqrt{\frac{\log N(\text{parent}(n))}{N(n)}}}_{\text{exploration}}$$

With $C = \sqrt{2}$: exactly like UCB1.

Each player picks by **their own** win rate.

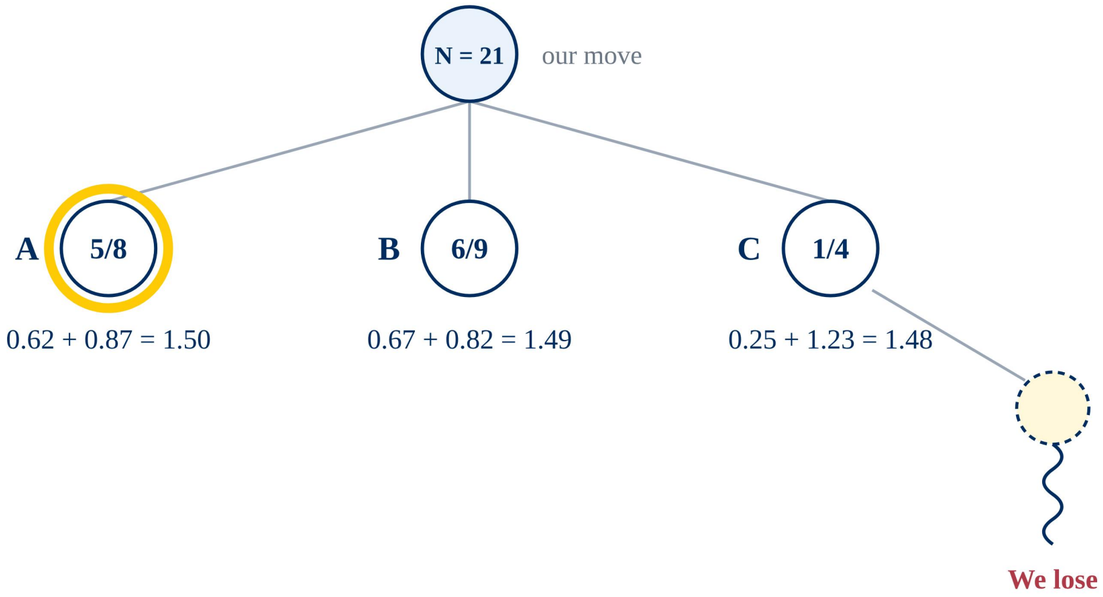
Unvisited children go first ($N(n) = 0$, so exploration $= \infty$).

With enough iterations, UCT's values **provably converge** to minimax values.

Kocsis & Szepesvári. Bandit based Monte-Carlo planning. ECML, 2006.

# One iteration

Example: say it's our move, root visited 20 times so far:




Q: Which child does selection pick?

A: C! Lowest win rate, but fewest visits.

Expand, play out: we lose. Q: What's next?

Then A, C's bonus lowers.

# MCTS: algorithm

```
def mcts(s0, iterations):
    tree = Tree(root=s0)
    for _ in range(iterations):
        leaf = select(tree)        # follow UCT down the tree
        child = expand(leaf)       # add one new node
        u = playout(child)         # random moves until the game ends
        backup(child, u)           # update W, N on the path to the root
    return most_visited_child(tree.root)
```

To play: the **most visited** move, not the best win rate. Few visits = noisy average.

Anytime: stop whenever we run out of time.

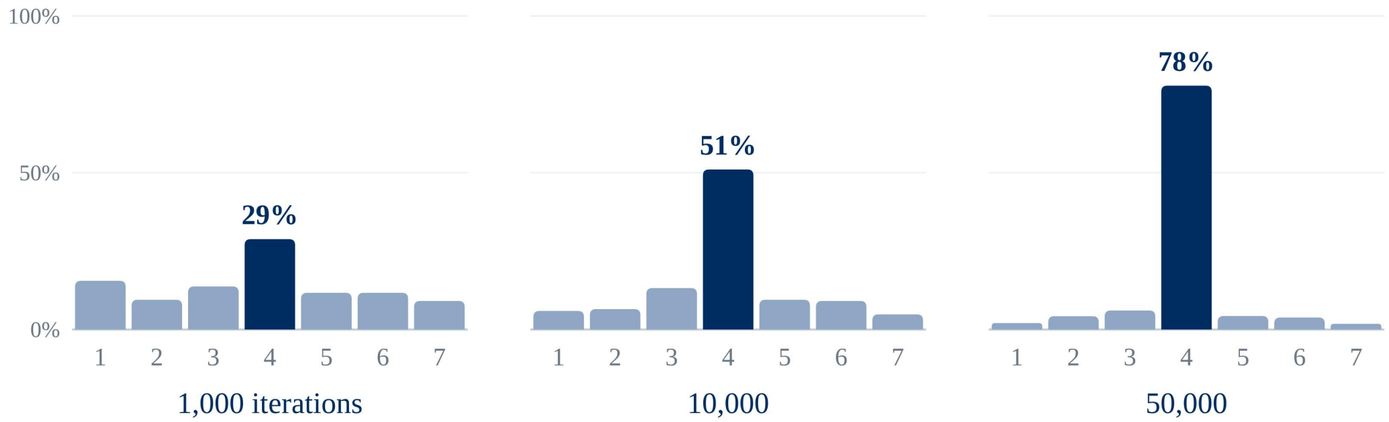
The tree grows unevenly: deep along promising lines, shallow elsewhere.

Russell & Norvig, Artificial Intelligence: A Modern Approach, 4th ed., ch. 5.

# MCTS on Connect 4

Experiment: UCT with random playouts on Connect 4 from the empty board.

Here's how the visit counts evolve:




More iterations: more and more visits to the center column.

Recall: Connect 4's best first move is the center!

No specific heuristic or domain knowledge here.

Node counts and MCTS runs: raw/cse592/lec18/games.py.

# MCTS vs. alpha–beta

| | Alpha–beta | MCTS |
|---|---|---|
| Position estimate | Hand-written Eval | Random playouts |
| Tree shape | Every move, to depth $d$ | Deep where it looks promising |
| Stop any time? | With iterative deepening | Yes |
| Exact? | Without a cutoff | In the limit |

Chess: alpha–beta engines dominated. Go: MCTS took over.

Before 2016, the best Go programs were MCTS + hand-designed playout policies (so not random).

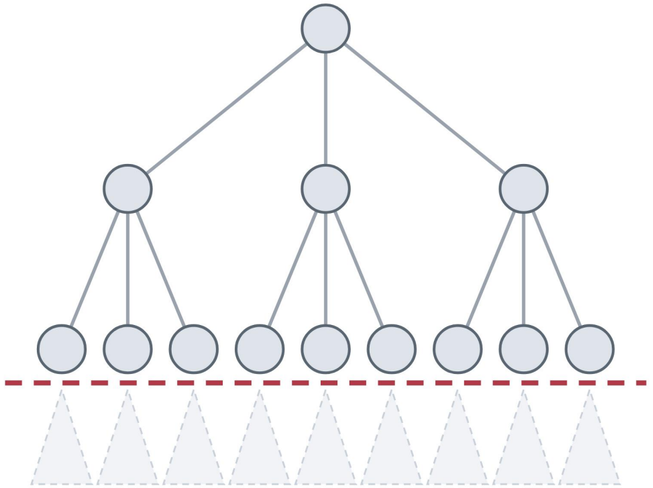
Even then, only strong amateur level!

Silver et al. Mastering the game of Go with deep neural networks and tree search. Nature, 2016.

# Learning to search

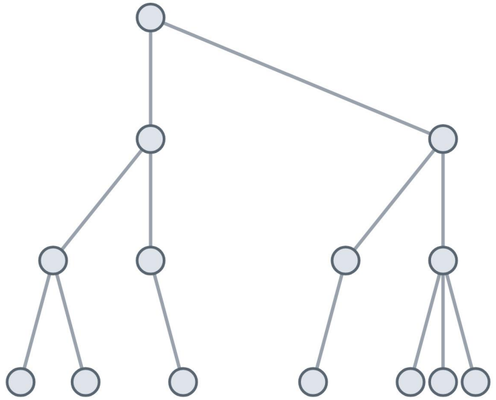
# Two ways to shrink the tree

**Reduce depth**



stop at depth d; estimate with v(s)

**Reduce breadth**



only expand moves likely under p(a|s)

**Depth**: a value function $v(s)$. Like Eval.

**Breadth**: a policy $p(a \mid s)$. Like move ordering.

Q: For Go, where could we get $v$ and $p$?

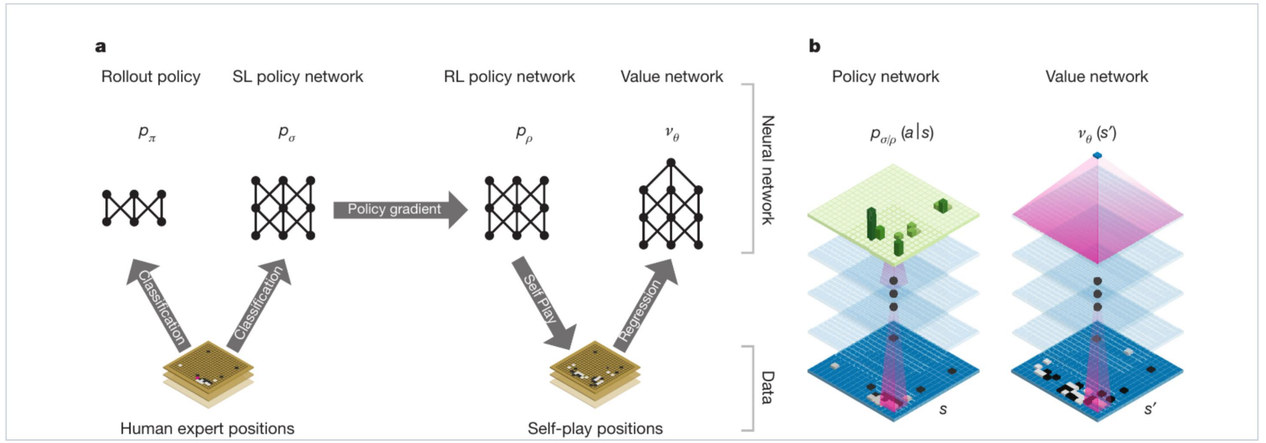
A: First idea: learn them (with neural nets)!

Silver et al. Mastering the game of Go with deep neural networks and tree search. Nature, 2016.

# AlphaGo (2016)

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

David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]




Policy network: predict expert human moves (trained on 30M positions from human games).

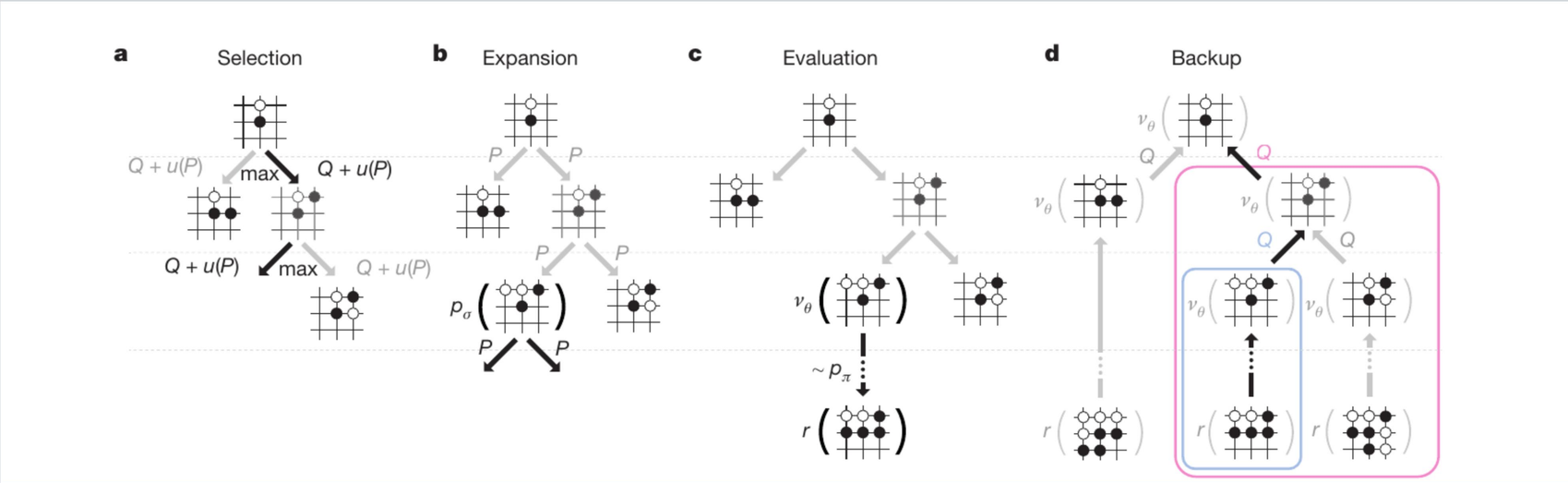
Accuracy: 57%.

Improve it by policy gradient on self-play games. Wins $>$ 80% against the original.

Value network: regression, predict the winner of self-play games (30M positions).

Silver et al. Mastering the game of Go with deep neural networks and tree search. Nature, 2016.

# Search in AlphaGo




MCTS, with the learned networks plugged in.

Selection: $Q(s,a) + u(s,a)$, with $u \propto \frac{P(s,a)}{1+N(s,a)}$. $P$: the policy network's probability.

Leaf evaluation: average of the value network and a fast playout: $V = \frac{1}{2} v_\theta(s) + \frac{1}{2} z$.

Beat Fan Hui (European champion) 5–0 in 2015, Lee Sedol 4–1 in 2016.

Silver et al. Mastering the game of Go with deep neural networks and tree search. Nature, 2016.

# AlphaGo Zero (2017)

**Mastering the Game of Go without Human Knowledge**

David Silver*, Julian Schrittwieser*, Karen Simonyan*, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, Demis Hassabis.

DeepMind, 5 New Street Square, London EC4A 3TW.

AlphaGo used supervised learning on human games, then self-play.

AlphaGo Zero: can we do just self-play?

One network, two outputs: $(p, v) = f_\theta(s)$. Move probabilities and value.

MCTS guided by $f_\theta$. No random playouts: the leaf's value is just $v$.

Selection rule (PUCT, UCT + Policy):

$$Q(s,a) + c\,P(s,a)\,\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$$

Q: Compared to UCT, what does the prior $P$ do different?

A: No denominator 0 anymore, so from the beginning we weigh using the policy.

Silver et al. Mastering the game of Go without human knowledge. Nature, 2017.

# Search as a teacher

Suppose we have some $f_\theta$, and we play $f_\theta$ vs. $f_\theta$ with MCTS. Who wins?

> 💡 **Insight**
> MCTS guided by $f_\theta$ plays better than $f_\theta$ alone.

Search output: visit counts at the root, $\pi(a) \propto N(s_0, a)^{1/\tau}$.

Idea: self-play, then train $p$ to imitate $\pi$, and $v$ to predict the winner $z$.

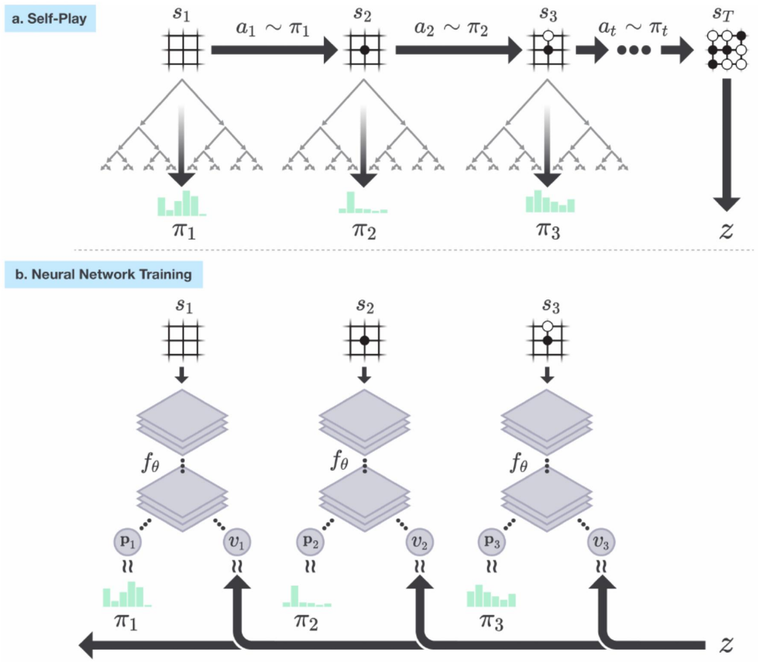
$$\ell = (z - v)^2 - \pi^\top \log p + c\|\theta\|^2$$

Better network → better search → better targets → better network → ...

The policy is trained by **supervised learning** on the search's output.

Silver et al. Mastering the game of Go without human knowledge. Nature, 2017.

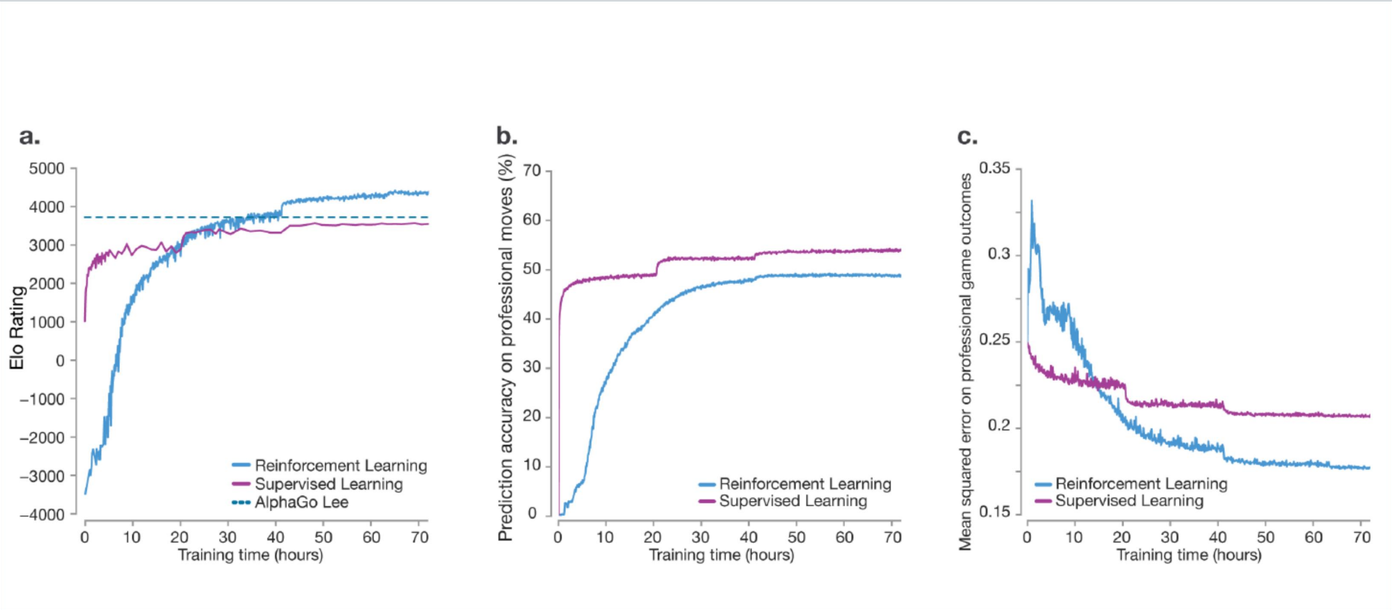
# Self-play training




```
θ = random_init()
loop:
   for each self-play game:
      for each position s:
         π = mcts(s, f_θ)
         store (s, π); play a ~ π
      z = who won (for each s)
   train f_θ on (s, π, z): p → π, v → z
```

Silver et al. Mastering the game of Go without human knowledge. Nature, 2017.

# AlphaGo Zero: results




3 days, 4.9 million self-play games, 1,600 simulations per move.

Passed AlphaGo Lee after 36 hours. Beat it 100–0 after 72.

(b): worse than supervised learning at predicting human moves. But it plays better!

Silver et al. Mastering the game of Go without human knowledge. Nature, 2017.

# Expert Iteration

**Thinking Fast and Slow with Deep Learning and Tree Search**

**Thomas Anthony**[1,✉], **Zheng Tian**[1], and **David Barber**[1,2]

Same idea, developed independently (NeurIPS 2017).

"Thinking fast and slow": fast intuition (System 1) vs. slow deliberation (System 2).

**Apprentice**

Neural network policy. Fast, no lookahead.

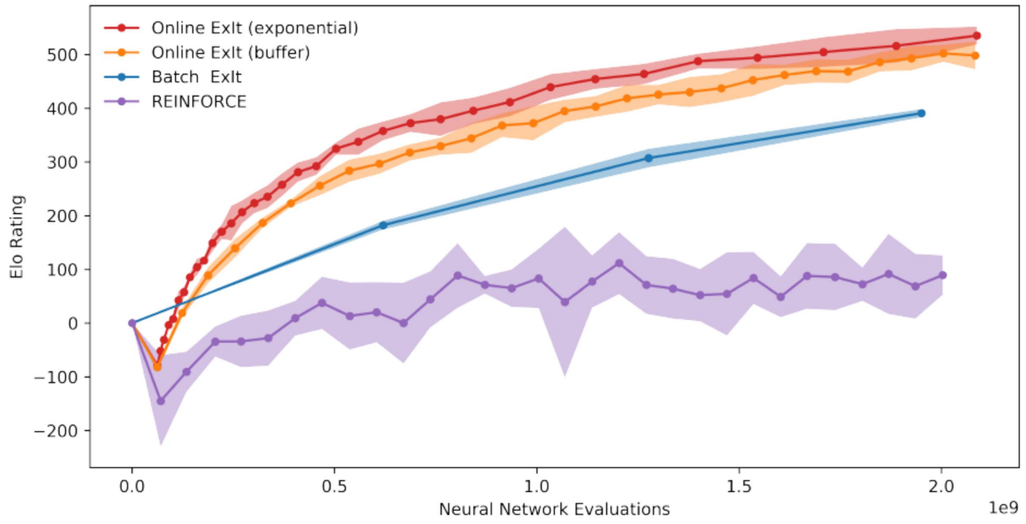
**Expert**

MCTS guided by the apprentice. Slow, stronger.

**Algorithm 1** Expert Iteration

1: $\hat{\pi}_0$ = initial_policy()
2: $\pi^*_0$ = build_expert($\hat{\pi}_0$)
3: **for** i = 1; i $\leq$ max_iterations; i++ **do**
4: $S_i$ = sample_self_play($\hat{\pi}_{i-1}$)
5: $D_i = \{(s, \text{imitation_learning_target}(\pi^*_{i-1}(s)))|s \in S_i\}$
6: $\hat{\pi}_i$ = train_policy($D_i$)
7: $\pi^*_i$ = build_expert($\hat{\pi}_i$)
8: **end for**

# ExIt on Hex




Hex, $9 \times 9$: connect your two sides of the board.

Same network, trained with REINFORCE (lecture 17): purple.

ExIt learns much faster, and steadily.

Why? Search gives a much better target than one game's outcome.

Final: beat MoHex 1.0, Hex Olympiad champion, in $75\%$ of games.

Anthony, Tian & Barber. Thinking fast and slow with deep learning and tree search. NeurIPS, 2017.

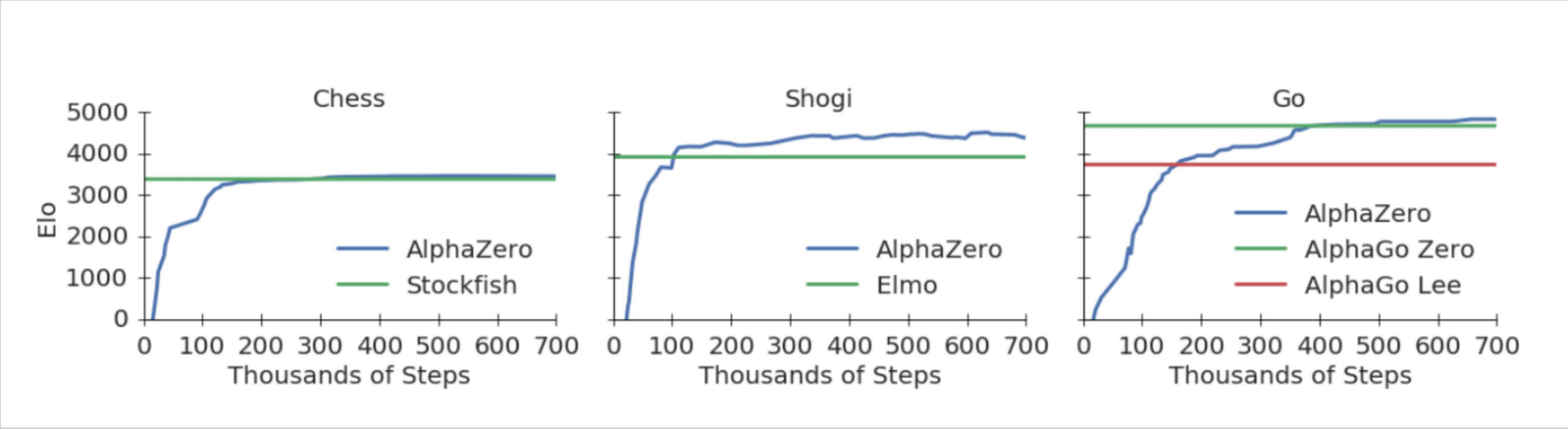
# AlphaZero (2017)

Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm

David Silver,[1*] Thomas Hubert,[1*] Julian Schrittwieser,[1*]
Ioannis Antonoglou,[1] Matthew Lai,[1] Arthur Guez,[1] Marc Lanctot,[1]
Laurent Sifre,[1] Dharshan Kumaran,[1] Thore Graepel,[1]
Timothy Lillicrap,[1] Karen Simonyan,[1] Demis Hassabis[1]

[1]DeepMind, 6 Pancras Square, London N1C 4AG.

Same algorithm and settings for chess, shogi, and Go. Only the rules change.




Chess: passed Stockfish after 4 hours of training.

100-game match vs. Stockfish: no losses. As White: 25 wins, 25 draws. As Black: 3 wins, 47 draws.

# Search less, evaluate better

AlphaZero searches 80 thousand chess positions per second. Stockfish: 70 million.

Roughly 1000× fewer positions, with a much better $p$ and $v$.

| | Search | Position estimate | Which moves first |
|---|---|---|---|
| Deep Blue, Stockfish | Alpha–beta | Hand-written | Heuristics |
| MCTS (Go, pre-2016) | MCTS | Playouts | UCT (+ hand-made playouts) |
| AlphaGo | MCTS | Value net + playouts | Policy net (humans, then RL) |
| AlphaGo Zero, AlphaZero | MCTS | Value net | Policy net (self-play only) |

Silver et al. Mastering chess and shogi by self-play with a general RL algorithm. arXiv:1712.01815, 2017 (Science, 2018).

# Summary

Games: the opponent controls half the moves. We need a strategy, not a plan.

Minimax: assume a perfect opponent. Exact, but $O(b^m)$.

Alpha–beta: same answer, skips what can't matter. Good move ordering: twice as deep.

Cutting off: stop at depth $d$, estimate with $\mathbf{Eval}$, a value function.

MCTS: estimate with playouts; a bandit at every node.

Learning: policy saves breadth, value saves depth. Expert Iteration, self-play.

End of module 4 (RL)!

Next and last module: modern AI systems (e.g., LLMs, reasoning, AI agents)