# Advanced Games Programming (AI)
# Part 4: Adversarial Search

Michael Buro

December 23, 2024

[Under Construction]

# Change Log

- [Oct 10] Created
- [Nov 22] Added reference [7]

# Outline

# [AI-Lec 10 L17] Game Theory Introduction

Historically, AI research has been driven by the urge to construct systems that compete with human expert game players - starting with A. Turing's pencil and paper Chess program in the 1950s

The achieved results have been remarkable:

Checkers World Champion Don Lafferty lost one game and drew 31 against UofA Checkers program Chinook in 1995

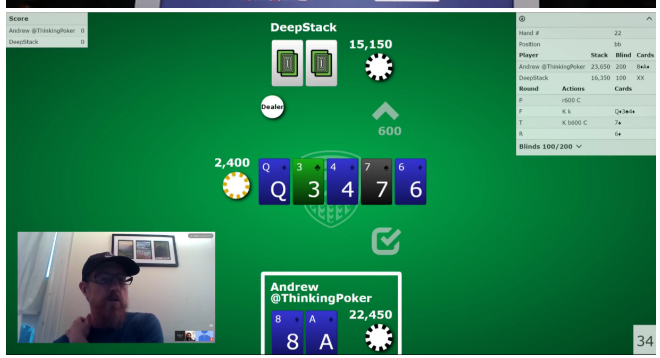
Chess World Champion Gary Kasparov lost 2.5-3.5 against IBM's Deep Blue in 1997

# Game Theory Introduction (continued)

Takeshi Murakami — the reigning Othello World Champion — lost 0-6 against Logistello the same year

Lee Sedol — a 9-dan professional Go player — lost 1-4 against DeepMind's AlphaGo system in 2016


UofA's DeepStack Poker system won against a group of professional players in 2017



# Game Theory Introduction (continued)

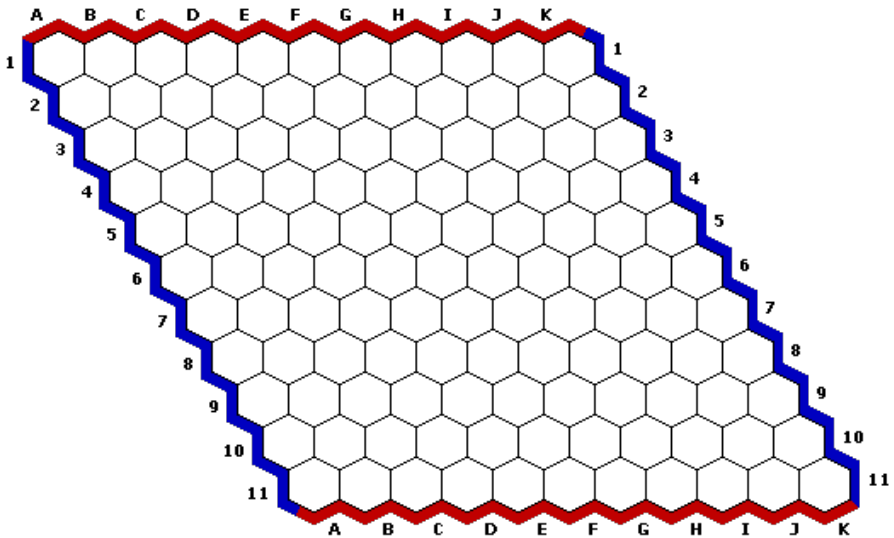
In this part we will discuss fundamental game theoretic concepts and algorithms for optimizing actions for different game types that were instrumental to some of the milestones mentioned above

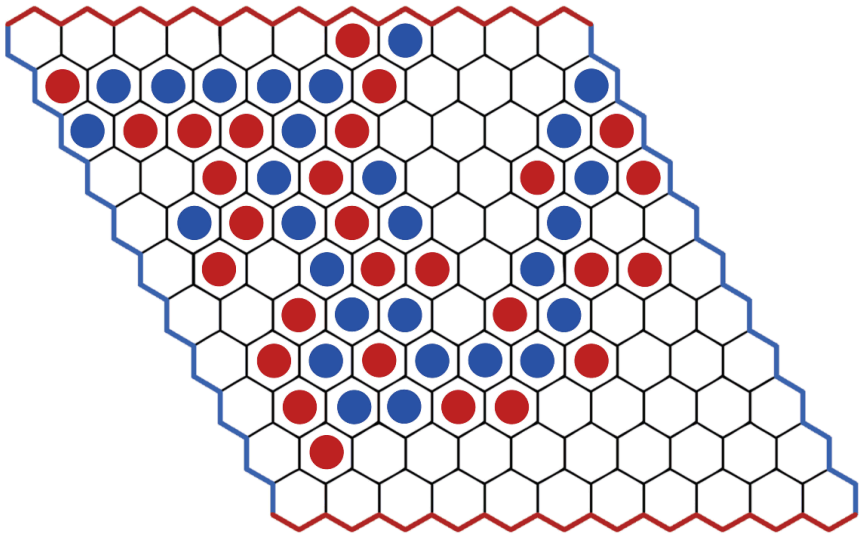
# Game Definition

A game is defined by the following components:

- $P$: the set of players which act
- $S$: the set of game states
- $s_0 \in S$: the start state (including player to move)
- $M : S \to P$: the player-to-move function that maps states to the unique player which acts next
- $T : S \to 2^S$: the move transition function which defines for each state what states are directly reachable. During game play, in state $s$, player $M(s)$ chooses the successor state from $T(s)$
- $E \subseteq S$: the set of end (or terminal) states. When reaching a state $s \in E$, the game ends. In such states, $T(s) = \emptyset$
- $R : E \to \mathbb{R}^{|P|}$: the reward (or payoff) function which assigns a reward value to each player in terminal states
- $I$: the information set structure, which essentially defines what players know about the state of the game (details below)

# Game Example: Hex



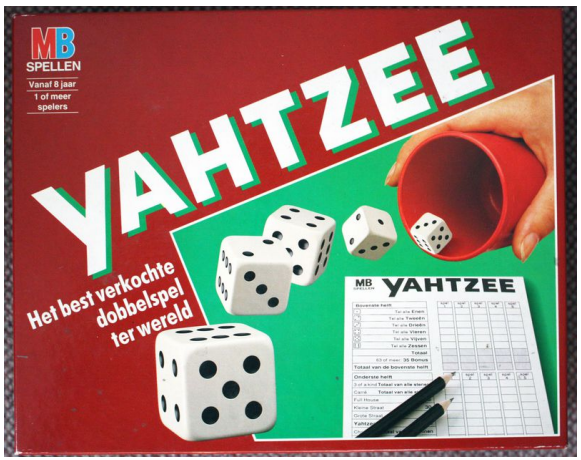
Objective: connect base lines by alternately placing stones on the initially empty board. The first player to do so wins (blue in the example on the right)

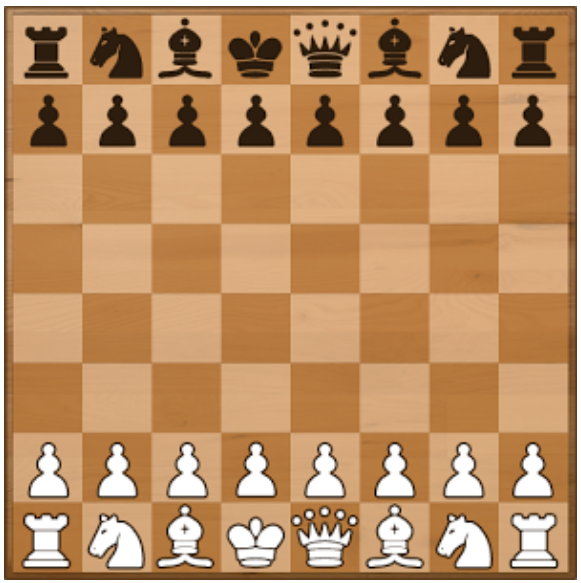
# Hex Formalization

- $P = \{\text{blue,red}\}$
- $S =$ set of all stone configurations on $k \times k$ hexagonal board, annotated with player to move
- $s_0 =$ empty board, blue to move
- $M$ : blue to move in $s_0$, alternating thereafter
- $E$ : set of states in which either blue or red connected their base lines
- $T$ : represents legal Hex moves: placing a stone on empty cell and changing the player to move, if the the state isn't terminal
- $R$ : $+1$ for player who connected his base lines first, $-1$ for the other player
- $I$: the game state is known to all players at all times

# Deterministic vs. Stochastic Games

- Stochasticity: there is a state where the environment (a special player) makes a random move (e.g., dice roll, card shuffle). The move probability distribution is usually known (e.g., Backgammon, Yahtzee)
- Deterministic = not stochastic (e.g., Chess)

# Game Strategies

- Strategy: describes how a player acts in any situation in which he is to move
- Pure strategy: in each non-terminal state a player chooses a fixed move
- Mixed strategy: a player chooses moves according to a probability distribution
- Nash-equilibrium strategy profile: players selecting their strategies so that noone has an incentive to change his strategy unilaterally

  Example: Playing Rock-Paper-Scissors – a popular children's game featuring simultaneous moves – with probabilities $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$

  Nash-equilibria exist for any finite game (famously proved by J. Nash in 1950)

# Payoff Structure

E.g., zero-sum, general, adversarial vs. cooperative

A game is a zero-sum game iff the payoff vector entries add up to 0 in each terminal state

For two-player games, this means that whatever one player wins the other loses

```
  E.g. Rock-Paper-Scissors          vs.        Prisoners Dilemma
       (zero-sum)                                (non-zero-sum)

   Simultaneous moves:          Both Cooperate: both go to jail for 1 year
   R beats S, S beats P,        Both Defect:    both go to jail for 2 years
   P beats R ...                One Defects:    defector goes free, other
                                                goes to jail for 3 years

Nash: probability for playing    Nash: Defect,Defect  OUCH! Not optimal
  R,P,S is 1/3 for both players
```

# Perfect vs. Imperfect Information

- Information set: set of possible game states which the player to move cannot distinguish; determined by the player's private knowledge about the game state, and the observable move history

  Example: Card Games
  You know your hand and the public move sequence. If you are to move, this defines your information set, which consists of all deals that are consistent with your cards and the observed move history

- Perfect information: all information sets have size 1, i.e., every player knows the state the game is in (e.g., Chess, Checkers, Go)

- Imperfect information: there exists an information set with size $> 1$ (e.g., Poker, Contract Bridge right after the initial deal)

  Mixed strategies are often required for playing imperfect information games well (e.g., R-P-S $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$ strategy, playing any other strategy can lose in the long run. How?)

# Modeling R-P-S

Rock-Paper-Scissors is a simultaneous move game

Against a robot with a high-speed camera you'd always lose
(E.g., https://www.youtube.com/watch?v=3nxjjztQKtY)

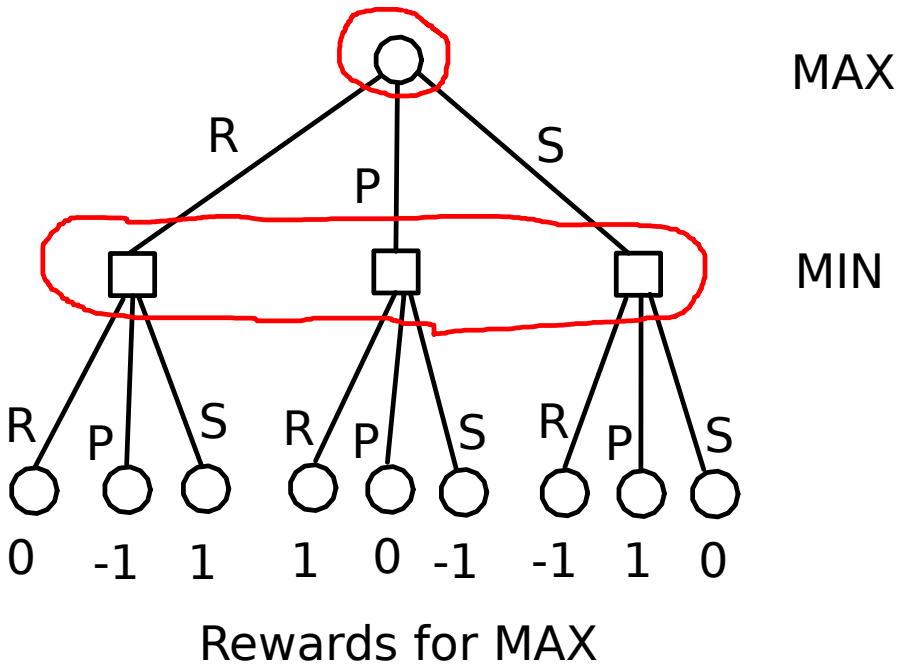
How can we model R-P-S as a fair alternating move game using a game tree in which states are represented by nodes and moves by edges?

We need something more, because it is easy to see that the first player would always lose if the second player knows the move

# Modeling R-P-S (continued)

Solution: we hide the first move (e.g., by writing it on a piece of paper and only revealing it after the second player made his choice)

Formally, information hiding can be accomplished with information sets which consist of game states the player to move cannot distinguish



# Modeling R-P-S (continued)

The top information set contains the start state, and the player to move knows that the game is in this state

The bottom information set contains three possible game states the second player cannot distinguish — he is confused and has to find a strategy that works equally well in all three scenarios

It turns out that pure strategies do not work well in R-P-S — whatever move you announce to play everytime, the opponent can always defeat

Thus, we are left with unpredictable mixed strategies

The only mixed Nash-strategy profile in R-P-S is both players using move distribution $(\frac{1}{3}, \frac{1}{3}, \frac{1}{3})$. We'll see how to compute such Nash-strategy profiles in Part 5 (covered in R11)

Note: above discussion shows that our game model needs to be refined: strategies actually map information sets to moves in case of imperfect information games

# Other Game Dimension

Number of players $n$ (excluding the environment)

- $n = 1$: Single player game (or puzzle); e.g., Rubic's cube
- $n = 2$: Two-player game; e.g., Chess
- $n > 2$: Multi-player game; e.g. DOTA-2 (5 vs. 5)

Finite or Infinite Game?

- In infinite games winning depends on property of infinite move sequence
- Can be used for proving properties of concurrent software systems such as liveness, i.e., provably making progress

Here, we only consider finite games

# Game Type Dependent Solution Methods

Most studied: deterministic two-player perfect information zero-sum games

- MiniMax search and improvements (Alpha-Beta search)
- Computing the best strategy possible in linear time in the game tree size
- Recently also Monte Carlo Tree Search (MCTS) based on sampling

Stochasticity + perfect information (e.g., Backgammon)

- ExpectiMax (MiniMax variant for games wtih change nodes)
- *-MiniMax (Alpha-Beta-like improvement)

# Game Type Dependent Solution Methods (continued)

Two-player zero-sum imperfect information games (e.g., Poker)

- Much harder to compute good strategies based on solving linear programs (i.e., maximizing linear functions subject to linear constraints)
- Computation still polynomial in the game tree size

More than two-players or non-zero-sum games

- Much harder still
- Known algorithms for computing Nash-equilibria run in time that is exponential in the game tree size

# Game Category Matrix

| | Perfect Information | Imperfect Information |
|---|---|---|
| deterministic | Checkers, Chess, Go, Othello, ... | Rock-Paper-Scissors, some RTS games, Kriegspiel, Stratego, ... |
| stochastic | Backgammon, Monopoly, ... | Contract Bridge, Hearts, Spades, Scrabble, Risk, Bluff, many video games ... |

# Two-Player Zero-Sum Perfect Information Games

E.g., Chess, Checkers, Othello, Backgammon, Hex, ...

Players MAX and MIN

Zero-sum means: the amount one player gains at the end of a game the other one loses, i.e., payoffs add up to 0
I win/you lose, I lose/you win, draw-draw ... sum=0

MAX tries to maximize his score, MIN tries to minimize MAX's score

Perfect information: at any time each player knows the state of the game (all information sets have cardinality 1)

# [AI-Lec 11 L19] Game Trees

The search space is usually represented by a game tree (or directed acyclic graph)

Sample game tree (directed edges pointing downwards):

```
MAX      o          root (height=2, depth=0) [start position]
        / \
MIN    x   x        height=1, depth=1
      / \   \
MAX o   o   o       leaves (height=0, depth=2) [finished]
   -1  +2   0
```

- Nodes: decision points, one player (or environment) to choose move
- Directed edges: move decisions – going from one state to the next
- Leaves: terminal positions labeled with game value for player MAX

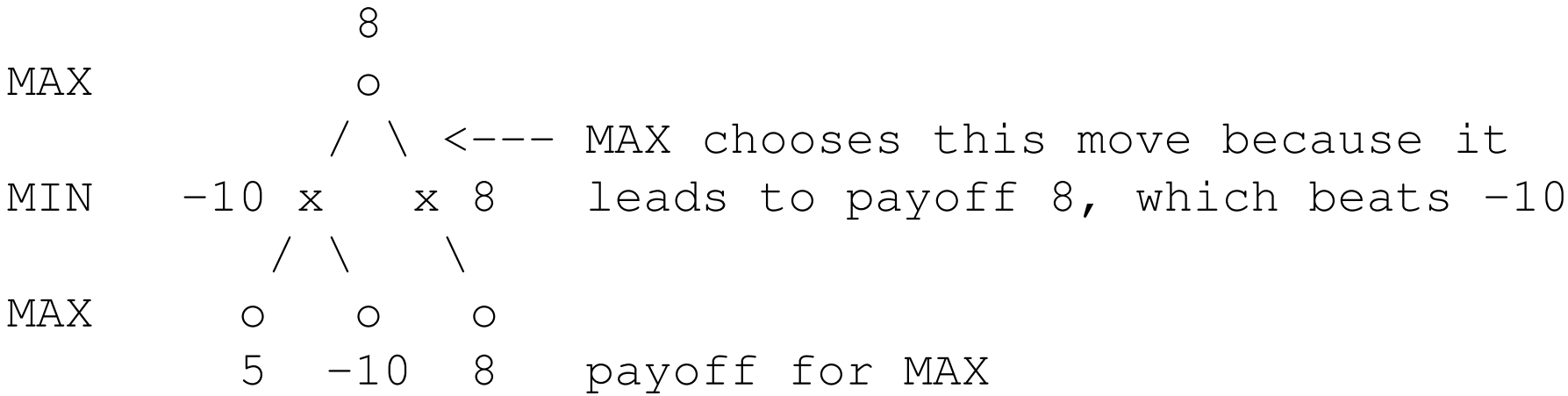
# Game Tree (continued)

- Depth of a node: number of state transitions (moves) from the start position to a given position
- Height of a node: maximum number of moves to reach a leaf from current node
- Branching factor: average or maximum number of successor nodes

# MiniMax Algorithm

The MiniMax algorithm determines the value of a state in view of player MAX assuming optimal play by recursively generating all move sequences, evaluating reached terminal states, and backing up values using the minimax rule (maximizing values for MAX, minimizing for MIN)

Example:




Our implementation stops searching after a certain number of steps ("height"), at which it approximates the state value. This is because interesting games are usually too big to be searched completetly. MiniMax is a depth-first-search algorithm

# MiniMax Pseudo Code

```
// returns value of state s at height h in view of MAX
int MiniMax(State s, height)
{
  if (height == 0 || terminal(s)) { // base case: value of state s in
    return Value(s, MAX)            // view of MAX; returns exact state
  }                                 // value in leaves; approx. otherw.
  if (toMove(s) == MAX) {
    score ← -∞                      // smaller than any state evaluation
    // compute move with maximum value for MAX
    for (i ← 0; i < numChildren(s); ++i) {
      value ← MiniMax(child(s, i), height-1)
      if (value > score) { score ← value } // found better move
    }
  } else {
    score ← ∞                       // bigger than any state evaluation
    // compute move with minimum value for MAX
    for (i ← 0; i < numChildren(s); ++i) {
      value ← MiniMax(child(s, i), height-1)
      if (value < score) { score ← value } // found better move
    }
  }
  return score
}
```

# Calling MiniMax

Call `MiniMax(s, height)` returns best approximate value achievable by MAX within height moves

`MiniMax(s, ∞)` returns exact value of *s*

Player distinction is awkward — it leads to code duplication which is error prone

# NegaMax Algorithm

NegaMax formulation : always evaluate states in view of player to move

```
// return value of state s in view of player to move
int NegaMax(State s, height)
{
  if (height == 0 || terminal(s)) {  // base cases
    return Value(s, toMove(s))       // value in view of player to move
  }                                  // approximation if not leaf
  score ← -∞
  for (i ← 0; i < numChildren(s); ++i) {
    // this assumes players alternate moves
    //        |
    value ← - NegaMax(child(s, i), height-1)
    if (value > score) {
      score ← value
    }
  }
  return score
}
```

Call: `result ← NegaMax(s, height)`

Shorter, less code to debug, fewer if statements => faster

What if players do not alternate moves, e.g. MAX plays twice in a row?

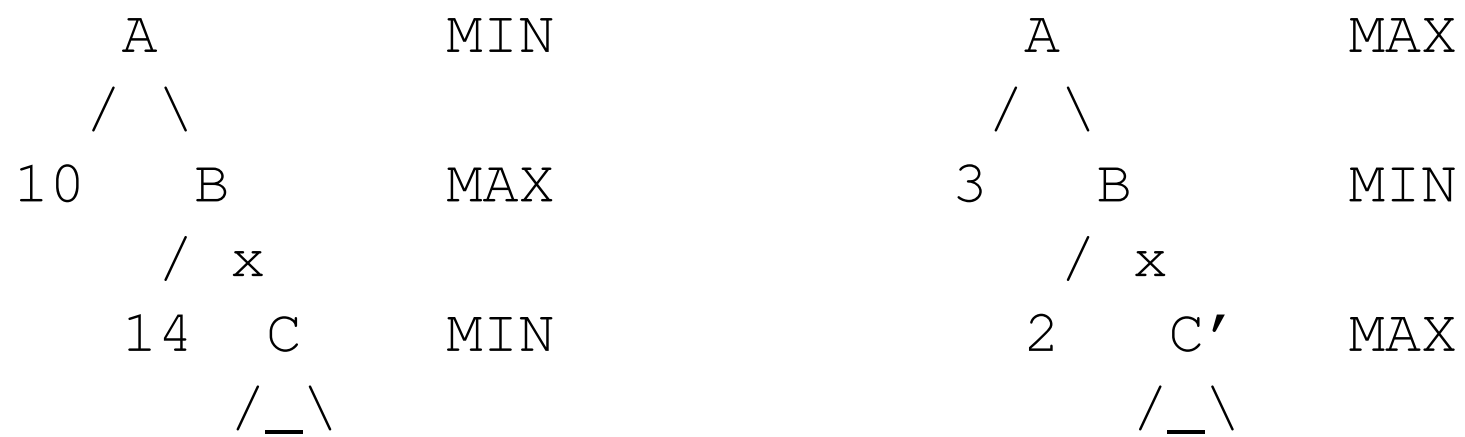
# MiniMax Analysis

Assume fixed uniform branching factor $b$ and search depth $d$

Number of visited leaves is $b^d$

Can we do better? Yes, some nodes in the search can be proved to be irrelevant to the search result

Examples (values in view of MAX):




The value of subtrees rooted at $C$ and $C'$ is irrelevant. Thus, those parts do not have to be searched

# Alpha-Beta Algorithm

This observation can be generalized:

The Alpha-Beta algorithm maintains two bounds:

- alpha: lower bound on what player to move can achieve
- beta: upper bound on what player to move can achieve

Whenever alpha $\geq$ beta, this node's exact value is irrelevant to move decision higher up in the tree, and thus its subtree can be pruned

Again, we'll use the NegaMax formulation which means that

- When descending, we negate and switch search bounds
- When ascending, we negate the return value

# Alpha-Beta Algorithm (continued)

```
// Input: s: state, height: steps to go, alpha/beta bounds
// For V = (heuristic) value of s: (see return value theorem)
//  If V ∈ (alpha, beta)  returns val = V
//  If V ≤ alpha          returns val with V ≤ val ≤ alpha
//  If V ≥ beta           returns val with beta ≤ val ≤ V
int AlphaBeta(State s, height, alpha, beta) {
  if (height == 0 || terminal(s)) { // base cases
    return Value(s, toMove(s))      // value in view of player to move
  }
  score ← -∞                        // current best value
  for (i ← 0; i < numChildren(s); ++i) {
    // assumes alternating moves: v ∈ (x, y) <=> -v ∈ (-y, -x)
    value ← - AlphaBeta(child(s, i), height-1, -beta, -alpha)
    if (value > score) {
      score ← value                 // better move found: update max
      if (score ≥ alpha) { alpha ← score }
      if (score ≥ beta) { break }  // beta cut
  } }
  return score
}
```

Call: `result ← AlphaBeta(s, height, -∞, ∞)`

# Illustration

Showing values and Alpha-Beta windows in MAX's view:

```
   | (0,10)   means: if value(A) is ≤ 0 or ≥ 10, then A does not
   A   MAX           influence decision higher up in the tree
  / \
 o   B (2,10)
 2  / \
   o   o
```

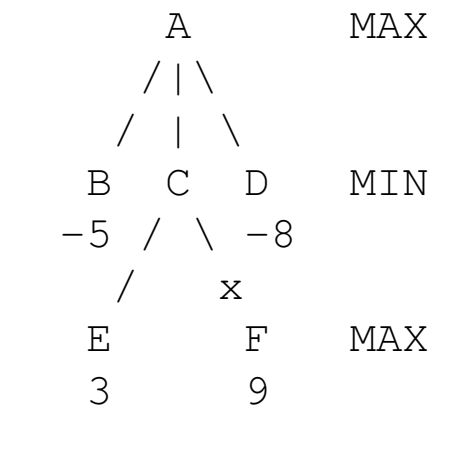
The first move in $A$ leads to value 2 $\Rightarrow$ subtree rooted at $B$ only makes a difference if $2 < \text{value}(B) < 10$

Thus, we can narrow the Alpha-Beta window to $(2, 10)$. The smaller the window, the more cuts may happen, i.e., fewer nodes have to be visited

In each node, the alpha bound gets bigger, and when it reaches beta, the value of the remaining children is irrelevant

# Another Example

Example: (left-to-right, Depth-First-Search like traversal)




1. initial call: AlphaBeta(A, 2, $-\infty$, $\infty$)
2. call AlphaBeta(B, 1, $-\infty$, $\infty$) = -5
   at node A: value = - (-5) = 5
   score = alpha = 5
   score < $\infty$ => no cut
3. call AlphaBeta(C, 1, $-\infty$, -5)
4. call AlphaBeta(E, 0, 5, $\infty$) = 3
   at node C: value = - (3) = -3
   score = alpha = -3
   score $\geq$ -5 => cut (F not visited)
   (MAX would never go to C, B is better)
5. call AlphaBeta(D, 1, 5, $\infty$) = -8
   at node A: value = - (-8) = 8
   score = al = 8

   done; returns 8 (move A->D is best for MAX)

# [AI-Lec 12 L21] Alpha-Beta Analysis

## Alpha-Beta Return Value Theorem

Let $V(s)$ be the true NegaMax value of state $s$ and $h$ the height of s (maximal distance to a leaf). Let $L$ = AlphaBeta$(s, h, \alpha, \beta)$

Then:

1. $L \leq \alpha \Rightarrow V(s) \leq L \leq \alpha$
2. $\alpha < L < \beta \Rightarrow V(s) = L$
3. $L \geq \beta \Rightarrow \beta \leq L \leq V(s)$
4. AlphaBeta$(s, h, -\infty, \infty) = V(s)$

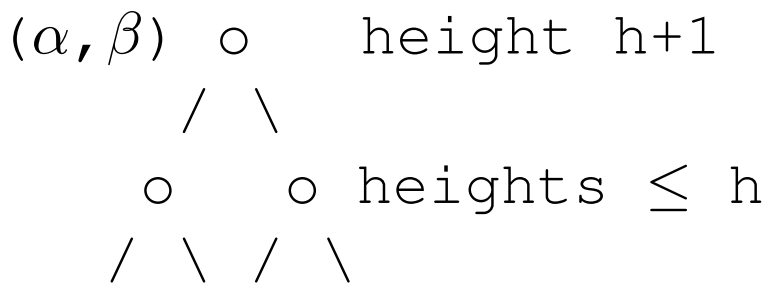
# Alpha-Beta Analysis (continued)

Proof by induction on node height – exercise (see [1] for details)

Idea:

- Prove claims 1,2,3) for height = 0 (leaves)
- Assuming claims 1,2,3) hold for nodes with height $\leq h$, prove that they also hold for height $h + 1$

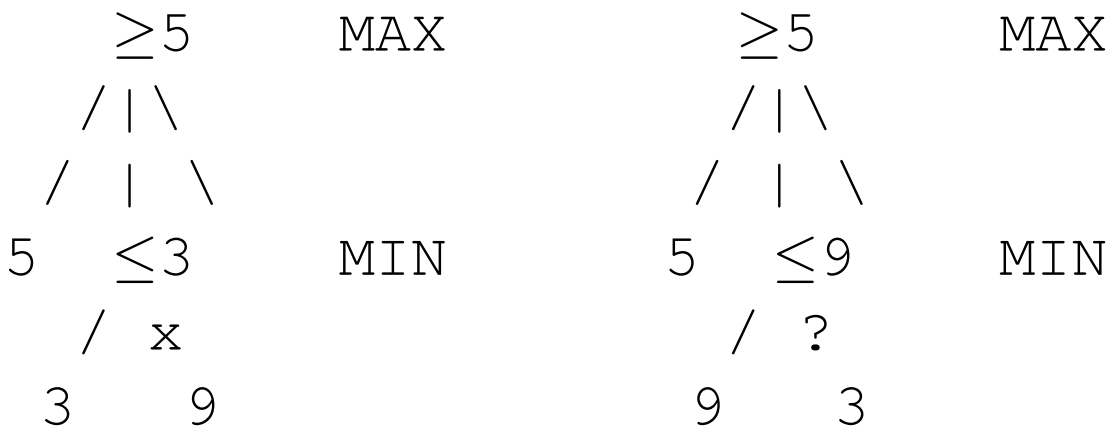
$\Rightarrow$ By the induction principle, claims 1,2,3) hold for all heights




Claim 4) follows from 2)

# Alpha-Beta Analysis (continued)

What is the best case for Alpha-Beta?
Consider two cases (values in view of player MAX)




Successor ordering matters!

- Alpha-Beta's performance depends on getting cut-offs as quickly as possible
- At nodes where cut-offs are possible, we ideally want to search one of the best moves first, and cut-off immediately

# Minimal Number of Leaves Visited by Alpha-Beta

Theorem:

In homogeneous game trees with branching factor $b$ and depth $d$ the minimum number of leaves to be visited to establish the MiniMax value of the root is

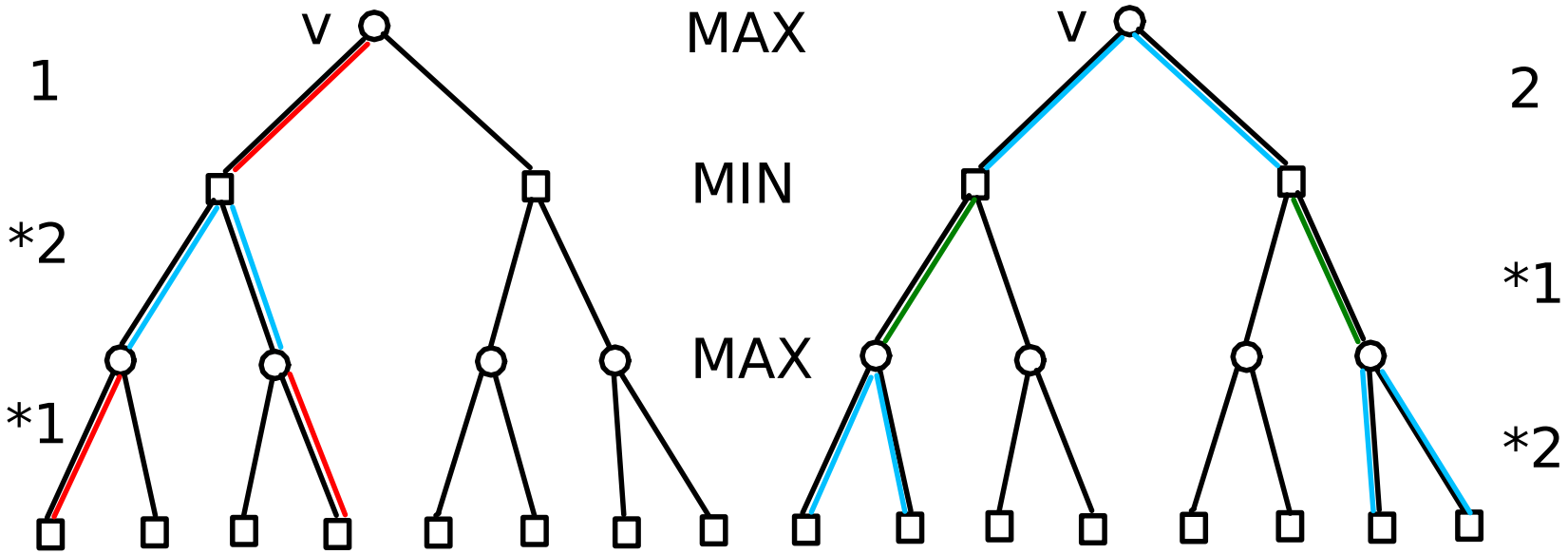
$$b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1 \quad (*)$$

$\lceil x \rceil$ = ceil($x$) = smallest integer $\geq x$
$\lfloor x \rfloor$ = floor($x$) = biggest integer $\leq x$

Proof:

To show that the root value $v$ is exactly $x$ we have to prove it to be $\geq x$ and $\leq x$ by presenting a strategy for MAX ($N$ leaves) and for MIN ($M$ leaves), resp. The minimal possible $N$ and $M$ values only depend on the tree parameters. It'll turn out that all minimal MAX (MIN) strategies have $N$ ($M$) leaves, and they always have one leaf in common. Therefore, $N + M - 1$ is a lower bound on the number of leaves

# Minimal Number of Leaves (continued)




MAX strategy
(relevant moves)

MIN strategy
(relevant moves)

All relevant opponent moves

proves v >=
min of reached leaf values

proves v <=
max of reached leaf values

# Minimal Number of Leaves (continued)

Assuming $d$ is even, the leaf count is

$$\underbrace{[(1 * b) * (1 * b)..(1 * b)]}_{\text{MAX strategy}} + \underbrace{[(b * 1) * (b * 1)..(b * 1)]}_{\text{MIN strategy}} - 1$$

$= b^{d/2} + b^{d/2} - 1$ [ OK: $\lfloor d/2 \rfloor = \lceil d/2 \rceil = d/2$ ]

(the leaf reached when following both strategies is counted twice)

For odd $d = 2k + 1$ the leaf count is

$$[(1 * b) * (1 * b)..(1 * b) * 1] + [(b * 1) * (b * 1)..(b * 1) * b] - 1$$

$= b^k + b^{k+1} - 1$ [also OK: $k = \lfloor d/2 \rfloor$, $k + 1 = \lceil d/2 \rceil$ ]

□

# Alpha-Beta Search Best Case

Theorem:

In homogeneous game trees with branching factor $b$ and depth $d$ Alpha-Beta search visits

$$b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$$

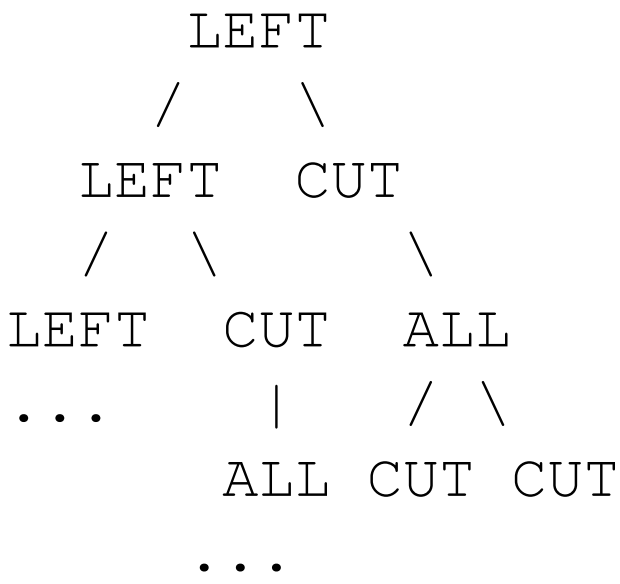
leaves if it considers the best move first in each node, i.e., Alpha-Beta search in this case is runtime-optimal

Proof Idea:

A node type system can be devised to describe the subtree Alpha-Beta search traverses (see below), which when best moves are searched first happens to have $b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$ leaves (see [1] for details) □

Note: in inhomogeneous trees it may not always be best to search the best move first (can you give an example?)

# Minimal Alpha-Beta Trees




Rules for minimal trees:

- Root is of type LEFT
- First successor of LEFT has type LEFT
- Remaining successors of LEFT have type CUT
- The CUT successor has type ALL
- All ALL successors have type CUT

# Impact

Approximation ($d$ even): $(*) \approx 2b^{d/2} = 2\sqrt{b^d} = 2\sqrt{b}^d$

Effective branching factor $b \to \sqrt{b}$ (E.g., Chess $\approx 36 \to \approx 6$)

Huge impact: given the same number of leaves, search depth is roughly doubled, assuming we use a good move ordering

Minimax: $10^9$ = 1,000,000,000 leaves

Alpha-Beta: $10^5 + 10^4 - 1$ ≈110,000 leaves $\to$ ≈9,100 times faster

Also, in homogeneous trees [all leaves at maximum depth, every interior node has $b$ children] searching the best move first minimizes the number of leaves Alpha-Beta has to visit. Thus, sorting moves well can have a big impact on the runtime

For more on minimum proof graphs and move ordering see [7]

# Are We Done?

Isn't this good enough, exponential gain already? No!

Still exponential ($b^d \rightarrow \sqrt{b}^d$ though)

Searching deeper often leads to much better decisions. Therefore, we can try to improve performance by

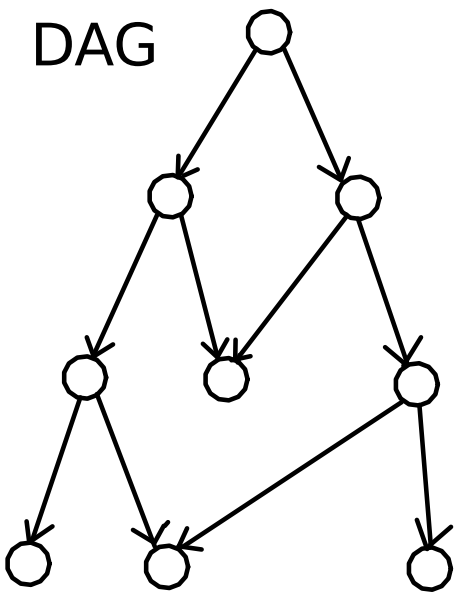
- Using storage to avoid repeating computations (transposition detection)
- Better successor ordering
- Increasing accuracy of heuristic evaluation functions (tradeoff between search and knowledge)
- Playing with search windows (smaller windows often lead to smaller trees. E.g., NegaScout)
- Playing with search depth (search reductions and extensions)
- Distribute work over multiple cores/processors or computers

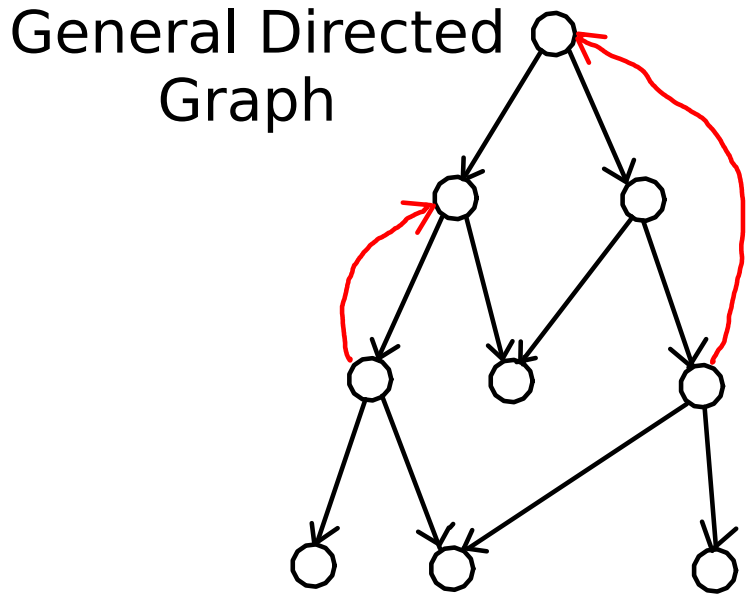
# [AI-Lec 13 L23] Search in Graphs

In many applications search is performed in directed acyclic graphs (DAGs) or general directed graphs

Often beneficial to detect and eliminate cycles

Detecting transpositions can also reduce search effort



# Cycles

A path from the root to the current node can contain a repeated position

Often, searching repeated nodes is unproductive and can be eliminated

Value of repeated state is application dependent:

- In many games scored as a draw (e.g., 3-fold repetition in Chess)
- Sometimes a loss for player that moved last
- Sometimes such a move is illegal (super-ko rule in Go)

Cycles can easily occur in games with reversible/commutative moves:

```
S1 -> S2 -> S3 -> S4 -> S1
   M1    M2   undo  undo
              M1    M2
```

At this point, if repetitions are scored as a draw, we do not explore S1 any further, assign value 0 to it and continue searching at S4

# Cycle Detection

Use a stack of states: make move $\rightarrow$ push, undo move $\rightarrow$ pop

Before pushing, check whether state already exists on stack

Optimization possible when irreversible moves exist (such as captures and pawn moves in Chess):

$\rightarrow$ Starting from the top of the stack, only search that far

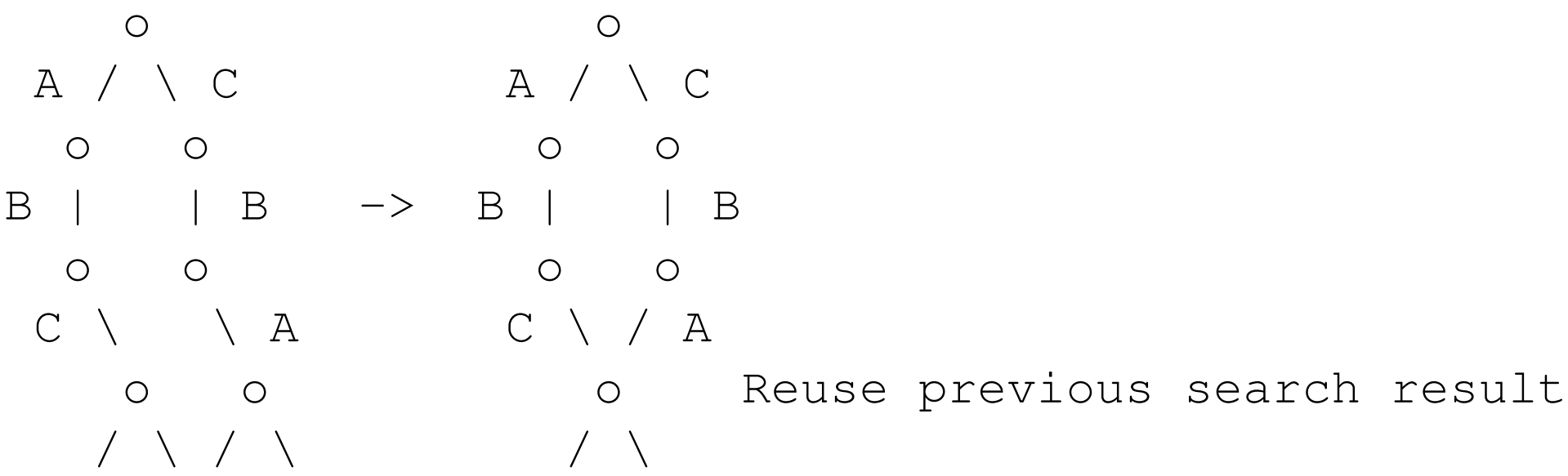
Little storage requirements, but can be slow in deep trees

Improvement: maintain a hash table that stores nodes on the stack

# Transpositions

It may be possible to reach the same state via two different paths (called transpositions)

We want to detect this and eliminate redundant search




For this we need to keep the history of previously visited nodes, not just along the current search path. Also, payoffs must not depend on history

# Transposition Table (TT) [ not covered - begin ]

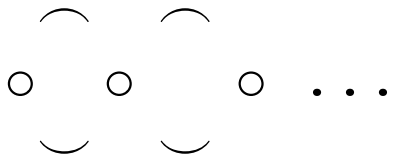
Cache of recently visited states and their exact values (or value bounds) and previously computed best moves

Usually implemented as hash table (fast!)

When visiting a node

- Check TT if state already encountered at a height $\geq$ current height
  If so, try to reuse stored value (or bound)
  Possibly resulting in immediate backtrack by producing a cut. E.g., if the TT entry indicates that the value is $\geq 5$ and beta = 4
- Otherwise, search subtree starting with the previous best move
- Save result in TT (exact value or bound, height, best move)

Savings of TTs can be exponential. E.g.



$2^i$ paths lead to node at depth $i$

# Transposition Table (continued)

What information do we need when revisiting a state?

We wish to decide whether search results can be reused or we need to search again

## Best previously found move *m*:

In case we need to search again, *m* should be tried first. It is likely that this move will also be best during the next search, leading to considerable savings in AlphaBeta search

## Search result *v*:

According to the Alpha-Beta Return Value Theorem, the previous result is either exact, a lower bound for the true state value, or an upper bound. To decide what to do when we reach the state a second time, we need to store *v* and a flag *f* that indicates the type of *v*

# Transposition Table (continued)

## Search height $h$:

If the game tree cannot be traversed completely, heuristic state evaluations need to be used, and deeper search can overcome their deficiencies

It is usually true that deeper search results are more accurate

If we reach a state a second time via a move transposition, the remaining search height might be different

If it is equal or lower, we can use the stored $v$, $f$ values. If not, we do not trust the stored value and have to search again

# Transposition Table (continued)

The last question we have to answer is how the current $(\alpha, \beta)$ window at the time we revisit a state influences the decision of whether to re-traverse the subtree or not depending on stored value $v$ and its type

Assuming the height test passes, there are three cases:

1. $v$ is exact

   We can return $v$ immediately, without searching at all

2. $v$ is a lower bound of the true state value

   We know the true state value is $\geq v$ and the subtree can only influence the decision at the root if its value lies in $(\alpha, \beta)$. Both conditions can be taken into account by changing the search window to $(\max(v, \alpha), \beta)$. This potentially narrows the window leading to smaller searches, and in case $\max(v, \alpha) \geq \beta$ to an immediate beta cut. In this case, we return $v$ as a lower bound

# Transposition Table (continued)

3. $v$ is an upper bound of the true state value

   In this case we know the true state value is $\leq v$ and, again, the subtree can only influence the decision at the root if its value lies in $(\alpha, \beta)$

   Both conditions can be combined by changing the search window to $(\alpha, \min(v, \beta))$

   If $\min(v, \beta) \leq \alpha$, we know the value cannot reach $\alpha$, and we therefore can return $v$ immediately, without further search

   Otherwise, the search window may have narrowed which can save time in the subsequent search

The following code implements these ideas:

# Alpha-Beta Search with TT

```
int AlphaBeta(State s, height, alpha, beta)
{
  if (height == 0 || terminal(s)) {
    return Value(s , toMove(s))
  }
  ttEntry ← checkTT(s, height, alpha, beta)
  if (alpha ≥ beta) { return ttEntry.value }
  score ← -∞
  // try heuristically best move first
  children ← GenerateAndSortChildren(s, ttEntry.bestMove)
  for (i ← 0; i < children.size(); i++) {
    value ← - AlphaBeta(children[i], height-1, -beta , -alpha)
    if (value > score) {
      bestMove ← move i
      score ← value
      if (score ≥ alpha) { alpha ← score }
      if (score ≥ beta) { break } // beta cut
    }
  }
  saveTT(ttEntry, s, score, bestMove, height, alpha, beta)
  return score
}
```

# Checking TT Entries

```
// retrieves TT entry and adjusts alpha,beta according
// to stored value bounds
TTEntry checkTT(State s, height, ref alpha, ref beta)
{
  ttEntry ← TranspositionTableLookup(s)
  if (ttEntry.valid() && ttEntry.height >= height) {
    // entry has been written to and stored value
    // is result of deeper or equal height search
    // which we trust to be at least as accurate
    if (ttEntry.flag == EXACT) {
      alpha ← beta ← ttEntry.value
    } else if (ttEntry.flag == LOWERBOUND) {
      alpha ← max(alpha, ttEntry.value)
    } else if (ttEntry.flag == UPPERBOUND) {
      beta ← min(beta, ttEntry.value)
    }
  }
  return ttEntry
}
```

# Saving TT Entries [ not covered - end ]

```
// save search results in TT entry
void saveTT(TTEntry ref ttEntry, score, bestMove, height, alpha, beta)
{
  ttEntry.value ← score
  ttEntry.bestMove ← bestMove
  ttEntry.height ← height

  // using the Alpha-Beta return value theorem:
  //    score ≥ beta  =>  true state value ≥ score
  //    score ≤ alpha =>  true state value ≤ score
  //    otherwise, true state value = score

  if (score ≤ alpha) {
    ttEntry.flag ← UPPERBOUND
  } else if (score ≥ beta) {
    ttEntry.flag ← LOWERBOUND
  } else {
    ttEntry.flag ← EXACT
  }
}
```

# Iterative Deepening

TTs are also very useful for move sorting when using iterative deepening:

```
h = 0
while (have time) {
  // using TT for storing values and best moves
  v = AlphaBeta(s, h, -∞, ∞)
  ++h
}
```

What looks like a waste of time, is in fact often faster than going to the maximum height right away because best moves in the previous iteration are likely good in the next. So we can store them in the TT and try them first next time. This improves the performance of Alpha-Beta search considerably

Iterative deepening also turns Alpha-Beta search into an anytime algorithm, which is good, because the maximum depth that can be reached given a certain time budget is hard to predict upfront

# Iterative Deepening (continued)

What is the total number of leaf visits when using iterative deepening MiniMax search using depths $0..d$ on a homogeneous search tree with branching factor $b > 1$?

At level $h$ we have $b^h$ nodes, so the total number is

$$\sum_{h=0}^{d} b^h = \frac{b^{d+1} - 1}{b - 1} \quad (b \neq 1)$$

To prove this geometric sum equation multiply by $(b - 1)$ and see how all but two terms on the left hand side cancel each other out

A regular MiniMax search to depth $d$ visits $b^d$ leaves. Thus, the fraction of leaves iterative deepening visits compared to regular search is

$$(b^{d+1} - 1)/(b - 1)/b^d \approx b/(b - 1)$$

which is $\leq 2$ for $b \geq 2$

# Iterative Deepening (continued)

To measure the overhead of iterative deepening Alpha-Beta search, we could assume perfect move ordering and relate the sum

$$\sum_{h=0}^{d} (b^{\lceil h/2 \rceil} + b^{\lfloor h/2 \rfloor} - 1)$$

to $b^{\lceil d/2 \rceil} + b^{\lfloor d/2 \rfloor} - 1$, like before

We'll leave this as an exercise, and instead approximate the ratio by applying the previous derivation to trees with the Alpha-Beta branching factor $b' = \sqrt{b}$, resulting in search overhead factor

$$\frac{b'}{b' - 1} = \frac{\sqrt{b}}{\sqrt{b} - 1}$$

which is similarly small. In practice, when using transposition tables, iterative deepening Alpha-Beta search is often faster than the deep one-shot search because of better move sorting

# Evaluation Functions

Evaluation functions assign values to non-terminal leaf nodes when the search algorithm decides to backtrack early

What value do you assign to such leaf nodes?

If you have a perfect evaluation function, then life is easy (the leaf node essentially becomes a terminal node)

Otherwise, we need to assign a heuristic value to the node

Heuristic values must be correlated with the true state value

The stronger the correlation, the more useful the heuristic

# Typical Evaluation Function

To evaluate a state we can simply add up weighted feature values:

$$\text{Eval}(s) = \sum_{i=1}^{n} w_i \cdot f_i(s),$$

where $w_i$ are real valued constant weights and features $f_i$ measure some properties deemed important indicators for the final outcome of the game

Non-linear functions can also be a good choice, but historically they weren't often, since weights can be harder to optimize because potentially many local minima exist. Also, evaluation speed may be slow (however, see recent successes of deep neural networks below)

Two important questions:

- How to select features?
- For a given feature set, how to assign weights?

# Typical Chess Evaluation Function

What is important in a Chess game?

- $f_1$: material balance
- $f_2$: piece development (mobility) balance
- $f_3$: King safety balance
- $f_4$: pawn structure balance

A simple evaluation function – returning the value of state $s$ in view of player $p$ – could look like this:

$$\begin{aligned} V(s,p) = & \; w_1 \cdot (\text{material}(s,p) - \text{material}(s,\text{opp}(p))) \\ & + w_2 \cdot (\text{pieceDevelopment}(s,p) - \text{pieceDevelopment}(s,\text{opp}(p))) \\ & + w_3 \cdot (\text{kingSafety}(s,p) - \text{YourKingSafety}(s,\text{opp}(p))) \\ & + w_4 \cdot (\text{pawnStructure}(s,p) - \text{pawnStructure}(s,\text{opp}(p))) \end{aligned}$$

where opp($p$) is the opponent of player $p$

# Typical Chess Evaluation Function (continued)

Usually, for the material balance different piece types are considered which are weighted by the empirical piece strength:

E.g.,

$$\begin{aligned} \text{material}(s,p) &= 10 \cdot \#\text{queens}(s,p) + 5 \cdot \#\text{rooks}(s,p) \\ &+ 3 \cdot \#\text{bishops}(s,p) + 3 \cdot \#\text{knights}(s,p) \\ &+ \#\text{pawns}(s,p) \end{aligned}$$

($s$: state, $p$: player)

# Example

You have 1 Queen and 2 pawns and your opponent has 2 rooks and 1 pawn

$$\begin{aligned} \text{material}(s, p) &: 10 \cdot 1 + 2 \cdot 1 = 12 \\ \text{material}(s, \text{opp}(p)) &: 5 \cdot 2 + 1 \cdot 1 = 11 \end{aligned}$$

So, you are slightly ahead in material and

$$V(s) = w_1 \cdot (12 - 11) + w_2 \cdot (...) + ...$$

# Interpretation of Values

To be able to compare values of states from different game stages (which often occurs when the search depth varies due to selective search extensions or reductions) evaluation values need a fixed interpretation, such as

- Probability of winning

  This is even useful for deterministic perfect information games. Probabilities become relevant when realizing that there are many positions with identical feature vectors. What matters then is the fraction of winning positions among them

- Expected perfect play game score

  In some games it matters how big wins actually are. E.g., achieving a gammon or even a backgammon counts more in Backgammon. In this case we are interested in maximizing the expected game score (which is a generalization of winning probability)

# Determining Weights

Historically done by manual tuning

$\rightsquigarrow$ tedious, time-consuming, error-prone

Nowadays, we use large-scale numerical optimization

Supervised learning (e.g., statistical regression)

- produce/observe $m$ training positions $s_i$ with game result label $v_i$
- fit weights $w_i$ of parameterized evaluation function $V_w(s)$ so that sum of squared errors is minimized

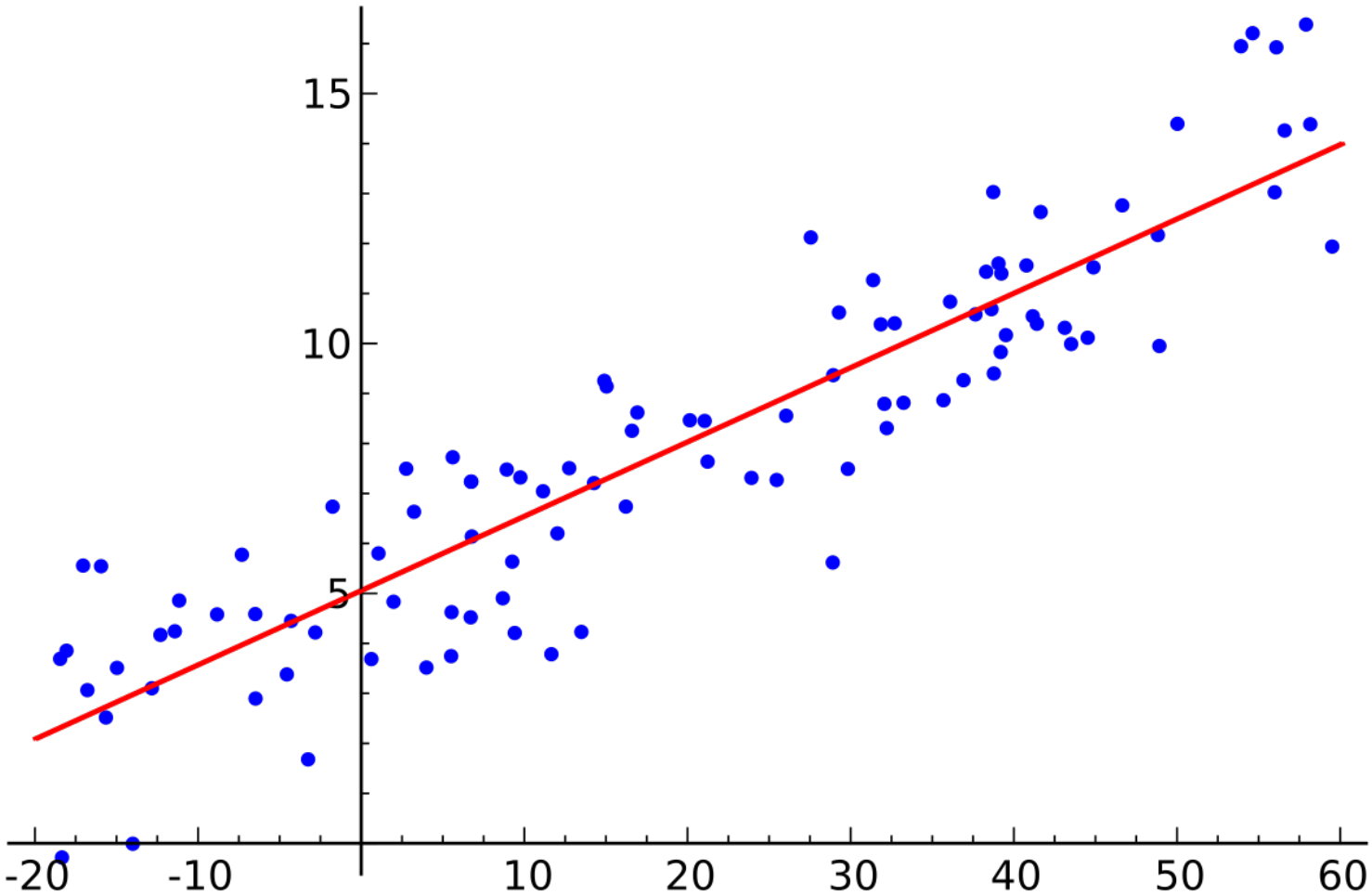
  I.e., find vector $w$ such that the error (or "loss") function

$$e(w) = \sum_{i=1}^{m} (v_i - V_w(s_i))^2$$

  is minimized w.r.t. parameter vector $w$

# [AI-Lec 14 L25] Example: Linear Regression

If $V_w$ is a linear function in $w$, this is called linear regression



# Determining Weights (continued)

In this case, optimal w vectors can be found directly by inverting a matrix for low-dimensional vectors $w$, or using iterative numerical optimization, such as gradient descent, if the number of parameters is large or $V_w$ is non-linear in $w$

```
feature          weight
 matrix error  estimate inverse  transposed
     |     |         |         |    /
```

$$y = Xw + e \quad \Rightarrow \quad \hat{w} = (X^t X)^{-1} X^t y$$

Here, $y$ denotes the label vector which represents the values we want to predict, and $X$ denotes the matrix consisting of feature vector rows

The linear model states that labels can be computed by multiplying the feature vector with weight vector $w$ and adding noise, which is assumed to be normally distributed with mean 0 and constant variance $\sigma^2$

# 2-Dimensional Example

Linear model: using two features ($x$, and the constant function 1)

$$y = a \cdot x + b \cdot 1 + e, \quad a, b \in \mathbb{R}, \; e \sim \text{Normal}(0, \sigma^2)$$

Given data $(x_i, y_i)$, determine $a, b$ such that the sum of squared errors

$$e(a, b) = \sum_i (a \cdot x_i + b - y_i)^2$$

is minimized

# Determining Weights (continued)

To find $a, b$ for which the error is minimized take the partial derivatives of $e(a, b)$ (assuming other variables are constant and using the chain rule), set them to 0 (necessary condition for local extrema), and solve for $a$ and $b$. I.e.,

$$\frac{\partial e(a,b)}{\partial a} = \sum_i 2(a \cdot x_i + b - y_i) \cdot x_i = a(2\sum_i x_i^2) + b(2\sum_i x_i) - 2\sum_i x_i y_i \stackrel{!}{=} 0$$

$$\frac{\partial e(a,b)}{\partial b} = \sum_i 2(a \cdot x_i + b - y_i) \cdot 1 = a(2\sum_i x_i) + b(2\sum_i 1) - 2\sum_i y_i \stackrel{!}{=} 0$$

Two linear equations with two variables $\Rightarrow$ can be solved for $a, b$

Linear regression works quite well in some simple domains (e.g., World's best Othello program evaluations predict the perfect play disc differential [3], featuring thousands of parameters that are automatically optimized)

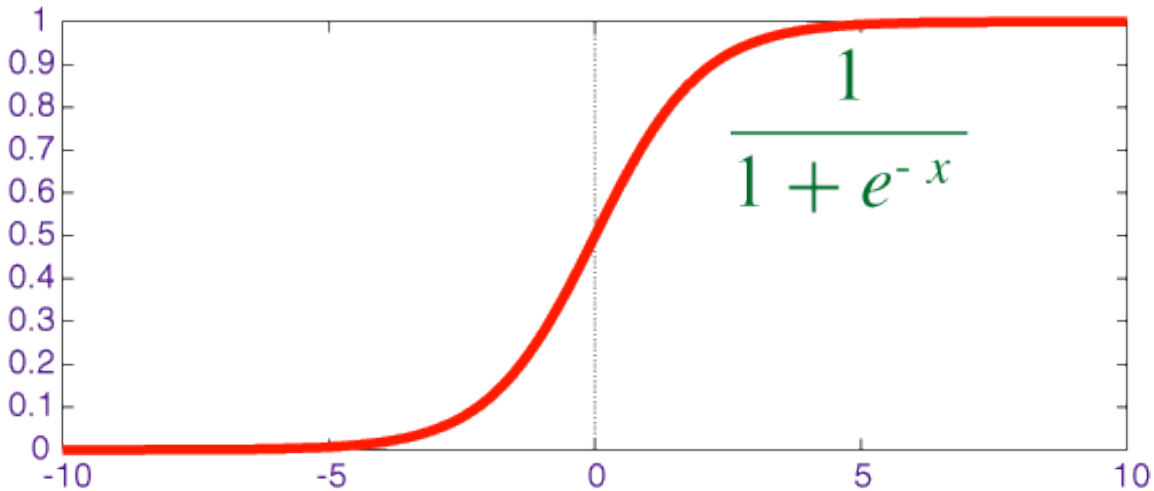
# Generalizing Linear Regression

Linear regression can be generalized in various ways. E.g.

**Logistic Regression:** modelling class membership probabilities. E.g.

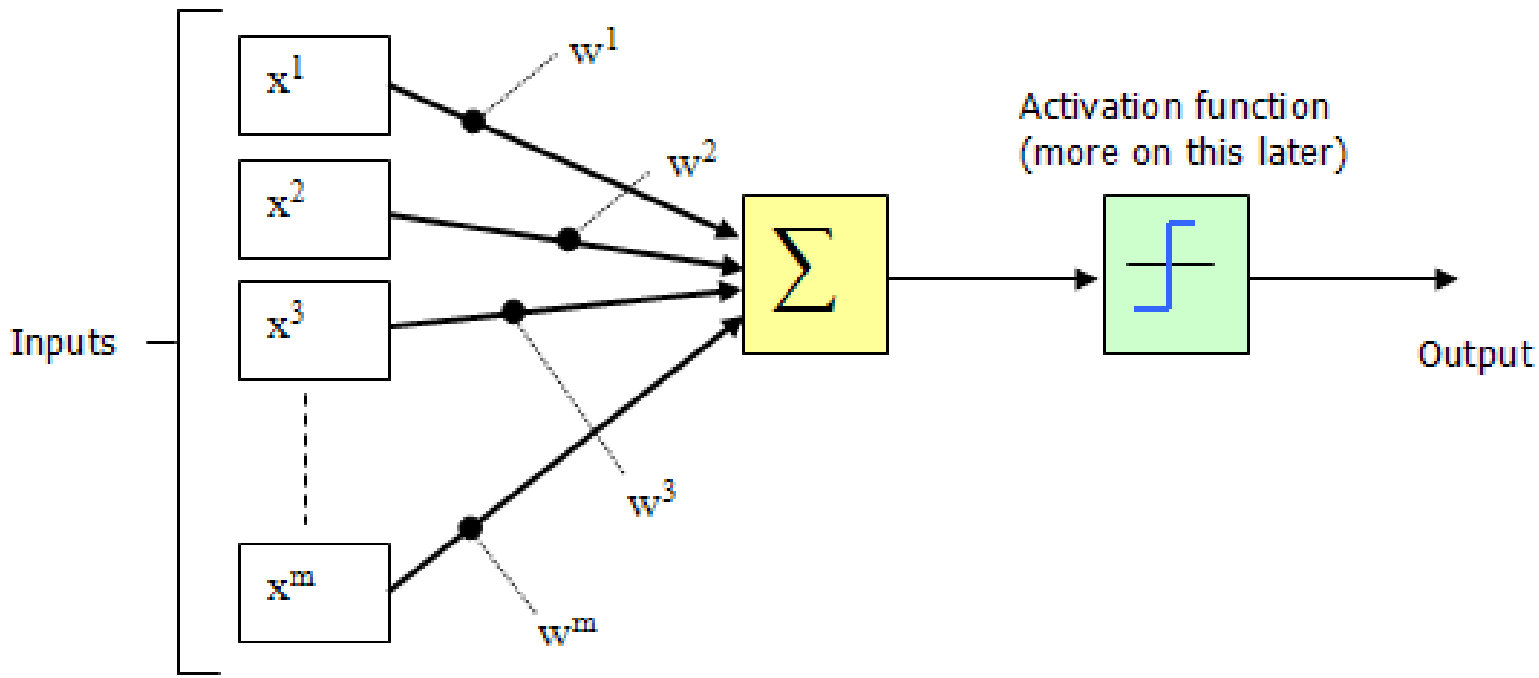
$$\text{Prob(player to move wins} \mid s) = \frac{1}{1 + exp(-\sum_i w_i \cdot f_i(s))} \quad (*)$$

This is called a generalized linear model because the evaluation core is linear and it is mapped using a so-called link or activation function, which is non-linear. For example:

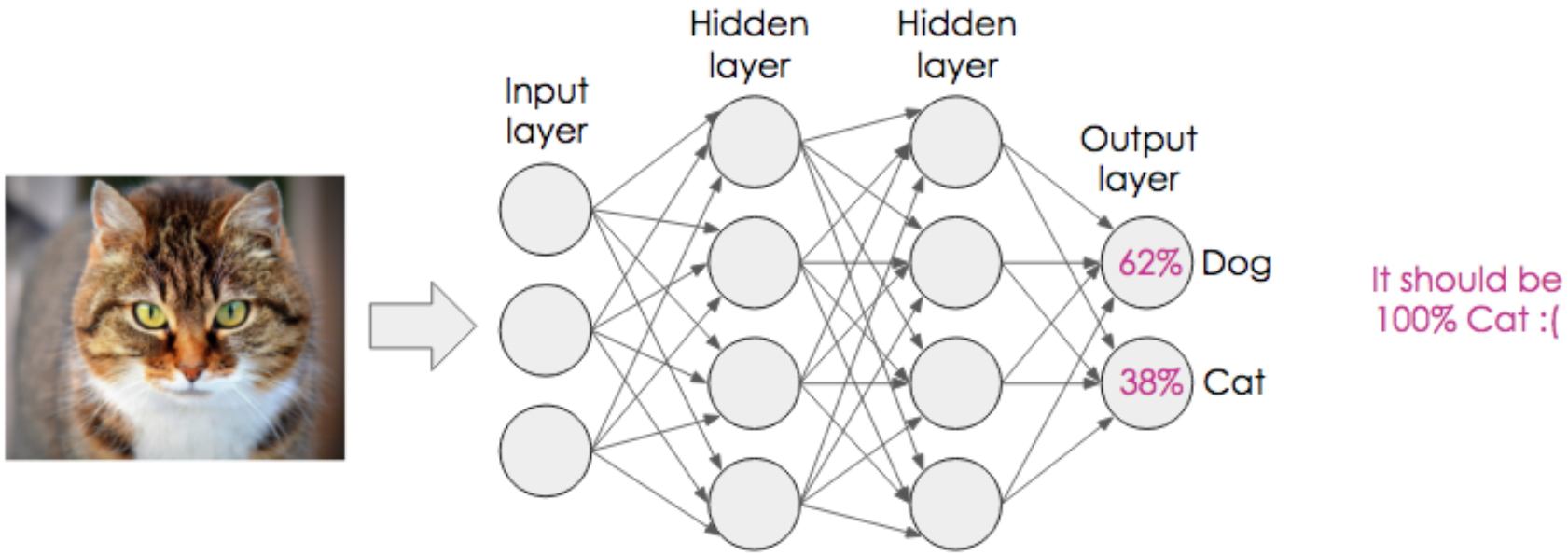


# Artificial Neural Networks

... or by using **artificial neural networks** which are non-linear function approximators composed of layers of hidden computation nodes that receive their inputs from previous layers, compute simple functions (like $(*)$), and propagate the output to subsequent layers



# Artificial Neural Network Example



# Artificial Neural Networks (continued)

Functions described by neural networks can be highly non-linear and optimizing weights is much harder in this case, because the error function is no longer convex (meaning that there could be many local minima)

Also, when applying back-propagation (iterative weight changes based on the chain rule) to networks with classic link functions such as $(*)$, gradients quickly vanish which makes it hard to train weights in deeper layers

But in the past few years new techniques have been developed that enable us to train large networks, sparking a revolution in image processing and AI (e.g., RELU ("rectified-linear-unit") activation functions, better weight initialization, etc.)

# The Power of Deep Networks

Why are deep networks so powerful?

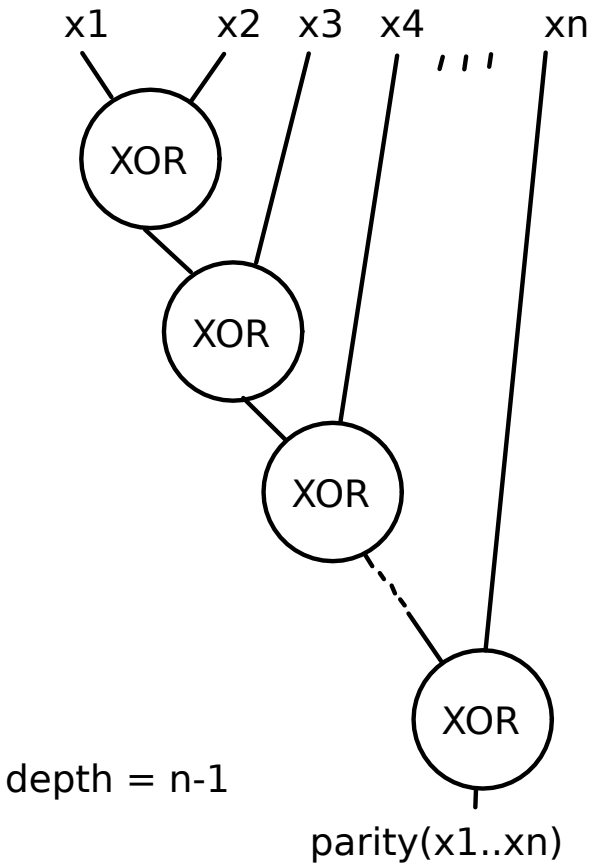
This can be illustrated by computing the parity function on $n$ bits:

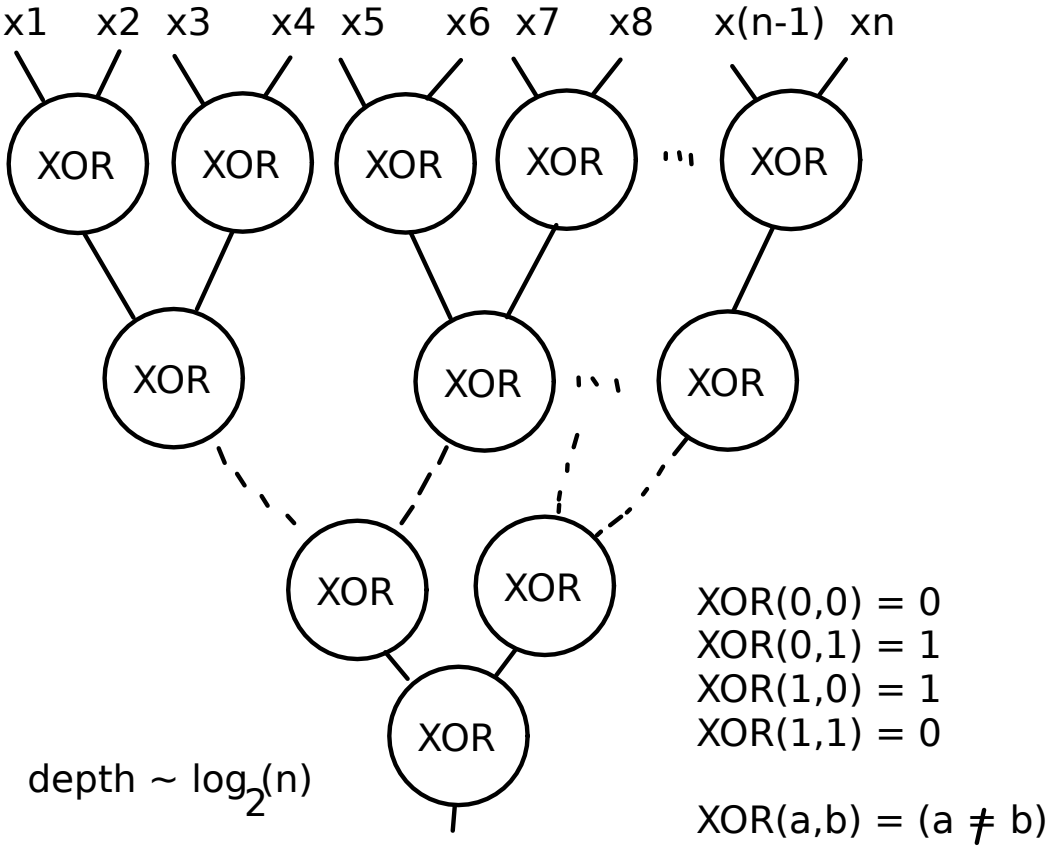
$$\text{parity}(x_1, ..., x_n) = (\sum_{i=1}^{n} x_i) \bmod 2$$

(i.e., result is 1 iff the number of 1 bits among $x_1, ..., x_n$ is odd)

Networks with one hidden layer require an exponential number of weights to approximate the parity function well, whereas $n$ or even $\log_2(n)$ layers can accomplish it with a linear number of weights by chaining XOR$(x, y)$ gate approximations:

# “Deep” Parity Networks



# Unsupervised Learning

E.g., Reinforcement Learning

- Have the program automatically interact with the environment (e.g., play games against other programs or itself)
- After playing a few episodes, modify model weights using gradient descent: change weights so that the evaluation becomes a better predictor of what actually happened (e.g., either approximating the end-game result, or the next state evaluation)
- Worked well in Backgammon (TD-gammon [2] and recently in Atari 2600 video games and Go [4,5]): Trained artificial neural-network to predict search results. Used this network for position evaluation

# Scaling Things Up

Good state evaluation functions are accurate and fast to evaluate

Chess programs are still getting better today by tuning evaluation parameters by statistical regression and reinforcement learning methods

Modern high-performance programs use functions that often contain millions of optimized parameters

# Example 1: Piece-Square Tables in Chess

Pieces in Chess are most effective when located close to the center of the board

So, instead of just counting how many pieces players have, we could construct a table for each piece type that contains values of pieces located on each square

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 1 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 1 | 2 | 3 | 3 | 3 | 3 | 2 | 1 |
| 1 | 2 | 3 | 3 | 3 | 3 | 2 | 1 |
| 1 | 2 | 3 | 3 | 3 | 3 | 2 | 1 |
| 1 | 2 | 3 | 3 | 3 | 3 | 2 | 1 |
| 1 | 2 | 2 | 2 | 2 | 2 | 2 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |

# Example 1 (continued)

The evaluation can still be linear at its core:

$$\text{eval-knights}(s, p) =$$

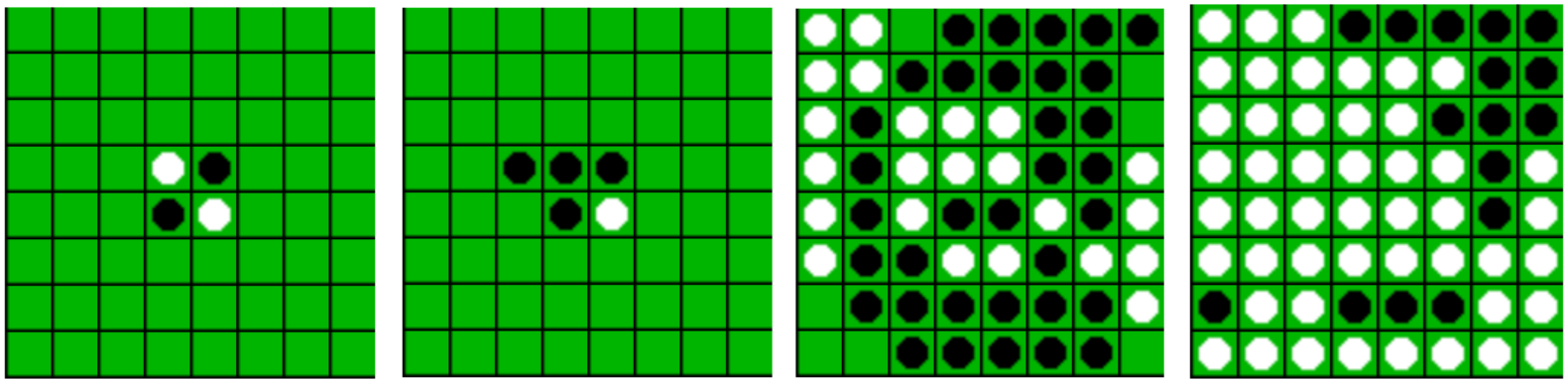
$$\sum_i (\text{knightOnSquare}(s, i, p) - \text{knightOnSquare}(s, i, \text{opp}(p))) \cdot \text{TKnight}[i]$$

Here, TKnight[$i$] is the value of a knight being placed on square $i$, and knightOnSquare($s, i, p$) is 1 if the player $p$ has a knight on square $i$ in state $s$, and 0 otherwise

Table values are regular feature weights and the evaluation function is still linear in the parameters. They therefore can be optimized with the methods described above

The resulting evaluation functions are often much more accurate

# Example 2: Pattern-Based Evaluation in Othello



[3] describes an evaluation function approach that bases state evaluation on Boolean conjunctions of atomic features

Example:

Edge configuration `OOXX-OOO` in Othello is good for `O` because `O` owns two corners and most interior discs for the rest of the game

One can read above configuration as

`s[A1]=O` and `s[B1]=O` and `s[C1]=X` and ...

# Example 2 (continued)

Edge pattern tables contain $3^8 = 6561$ entries which represent values of all possible edge configurations (square content: empty, $\mathrm{O}$, $\mathrm{X}$) in view of the player to move. To evaluate positions all matching table entries are added

Like piece-square tables, these weights can be optimized using statistical regression or reinformement learning

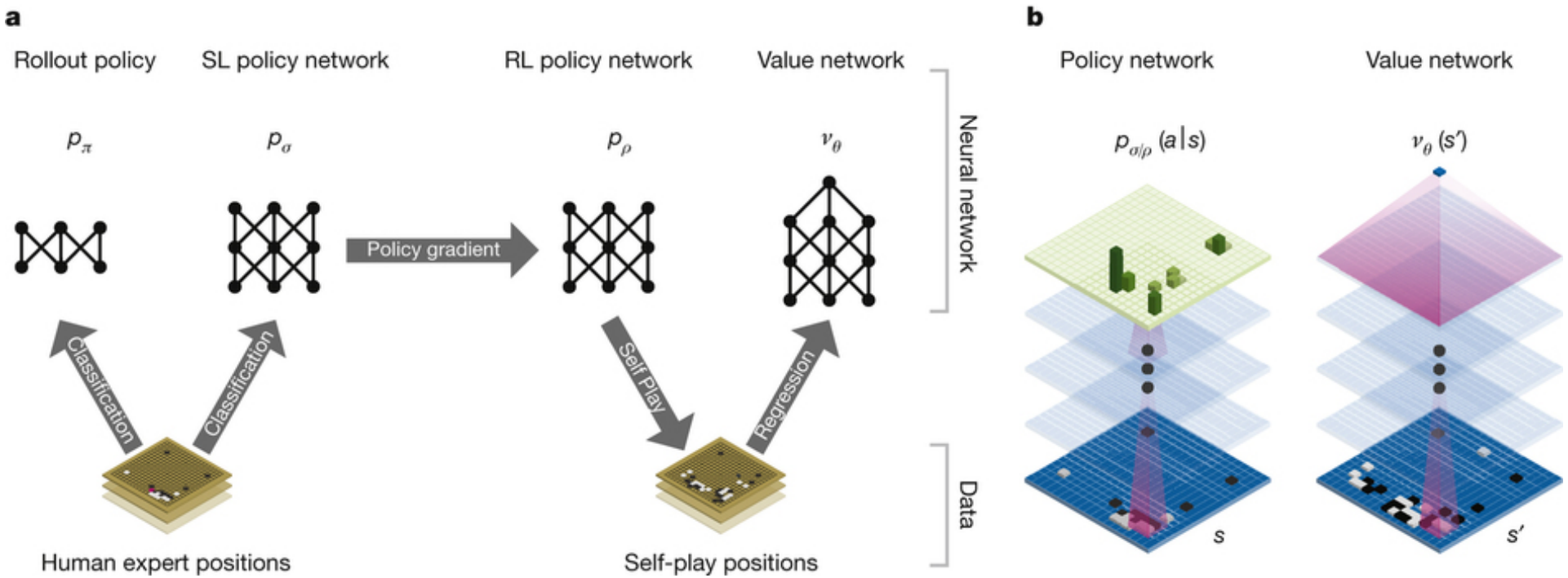
Evaluation functions built on various pattern tables such as all horizontals, verticals, diagonals, $3 \times 3$ corner regions, $2 \times 5$ corner regions, contain hundreds of thousands parameters and capture feature correlations well, unlike manual parameter tuning

The resulting evaluation functions are very accurate and fast to evaluate

Using such an evaluation function combined with deep selective search and opening-book learning, Logistello won against Takeshi Murakami – the reigning Othello Worldchampion – 6-0 in 1997

# Example 3: Deep Neural Networks Applied to Go

The first version of AlphaGo [4] was based on training large neural networks to predict moves and evaluate positions




They were trained based on 100 thousands of human expert games and through self-play

# Example 3 (continued)

With this an AI milestone was reached in March 2016:

The program defeated Lee Sedol, one of the best Go players, 4-1, a historical victory!

In 2017, AlphaGo-Zero [5] was trained solely based on self-play, and has reached a super-human playing level – no human expert understands anymore how the system plays

It won 100-0 against the version that won against Lee Sedol ...

# Where to go from here?

- Read tutorials on deep neural network learning (e.g., [6]). There is a lot of material out there now!
- Take statistics, machine learning, or reinforcement learning courses if you have not done so yet
- Get hands-on experience by launching your own machine learning project (e.g., guided by tutorials) using deep network learning software frameworks such as Tensorflow or PyTorch
- Work on video game AI, which is the new AI research challenge after Chess and Go. E.g., StarCraft 2 features a built-in AI programming API now, has a Python-based ML environment, and over a million human games available for training

Many companies are looking for CS students with data analytics and machine learning experience!

# References

[1] D.E. Knuth and R.W. Moore, An Analysis of Alpha-Beta Pruning. Artificial Intelligence 6 (4) (1975) pp. 293–326

[2] G. Tesauro, TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play, Journal Neural Computation archive Volume 6 Issue 2 (1994), pp. 215-219

[3] M. Buro, Improving Heuristic Mini-Max Search by Supervised Learning, Artificial Intelligence, Vol. 134 (1-2) (2002) pp. 85-99

[4] D. Silver et al., Mastering the game of Go with deep neural networks and tree search, Nature, Vol. 529 (2016), pp. 484–492

[5] D. Silver et al., Mastering the game of Go without human knowledge, Nature, Vol. 550 (2017), pp. 354–359

[6] Q.V. Le: A Tutorial on Deep Learning (Part 1 + Part 2)
`http://ai.stanford.edu/~quocle`

[7] T. Furtak and M. Buro, Minimum Proof Graphs and Fastest-Cut-First Search Heuristics, IJCAI, Pasadena USA, 2009, pp. 492-498 (on my webpage)

# References (continued)

Useful AlphaBeta search links:

https://en.wikipedia.org/wiki/Alpha-beta_pruning
https://www.chessprogramming.org/Main_Page