Two-Player Games

Lucas Janson

CS/Stat 184(0): Introduction to Reinforcement Learning
Fall 2024

Today

- Feedback from last lecture
- Recap
- Game Playing: AlphaBeta Search/Rule Based Systems
- MCTS
- AlphaZero and Self-Play

Feedback from feedback forms

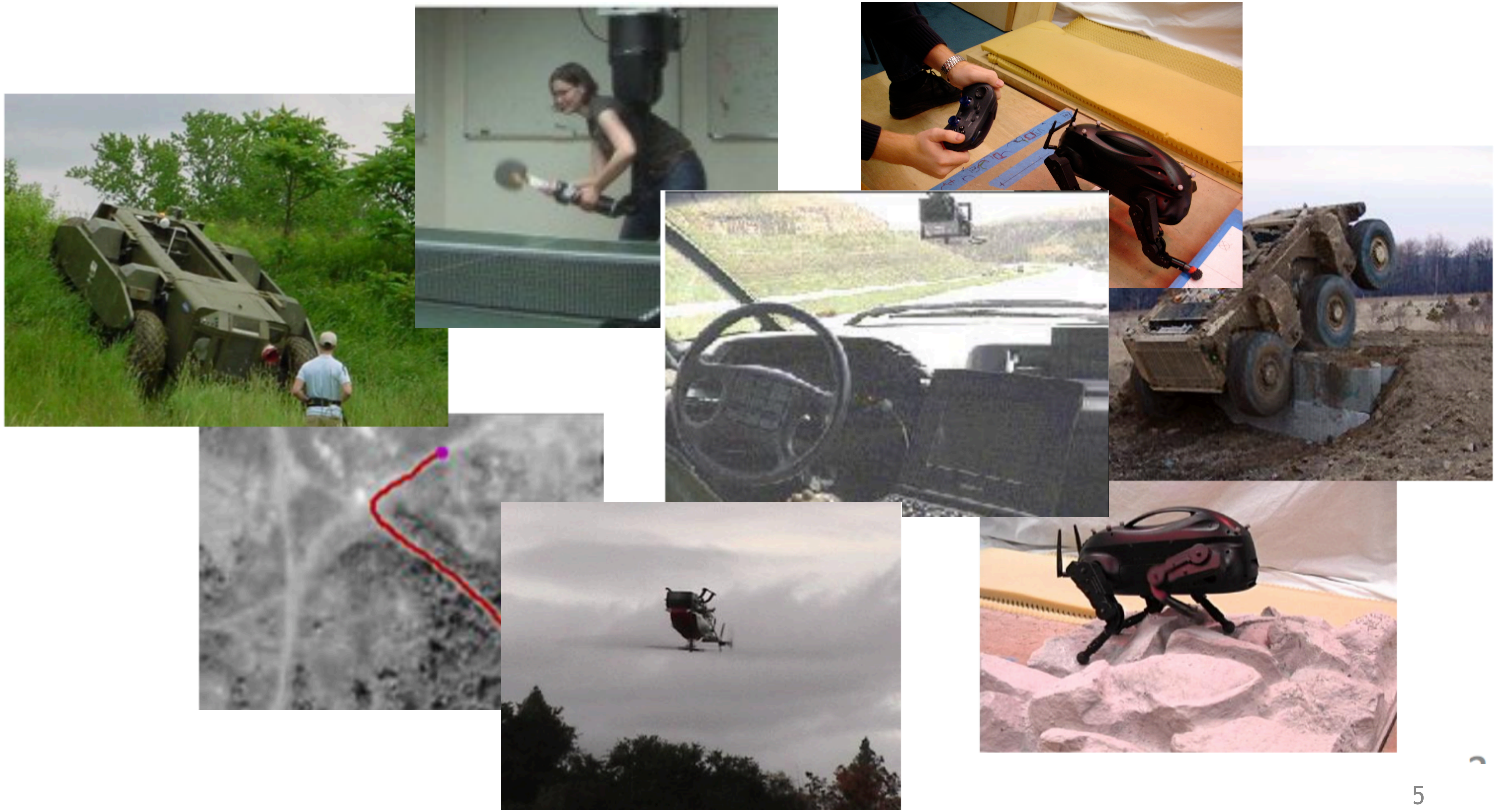
Feedback from feedback forms

1. Thank you to everyone who filled out the forms!

Today

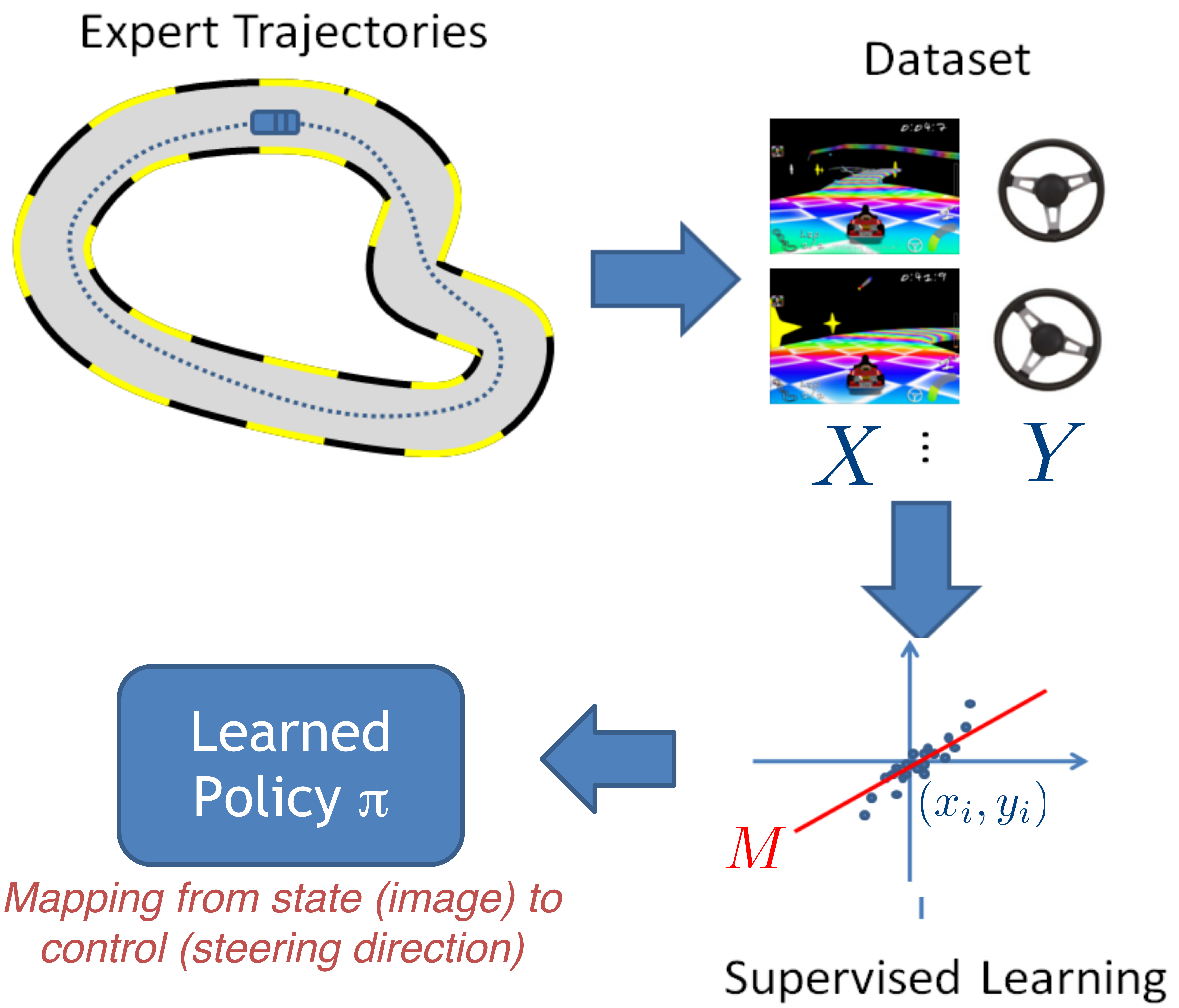
- Feedback from last lecture
- Recap
- Game Playing: AlphaBeta Search/Rule Based Systems
- MCTS
- AlphaZero and Self-Play

Imitation Learning

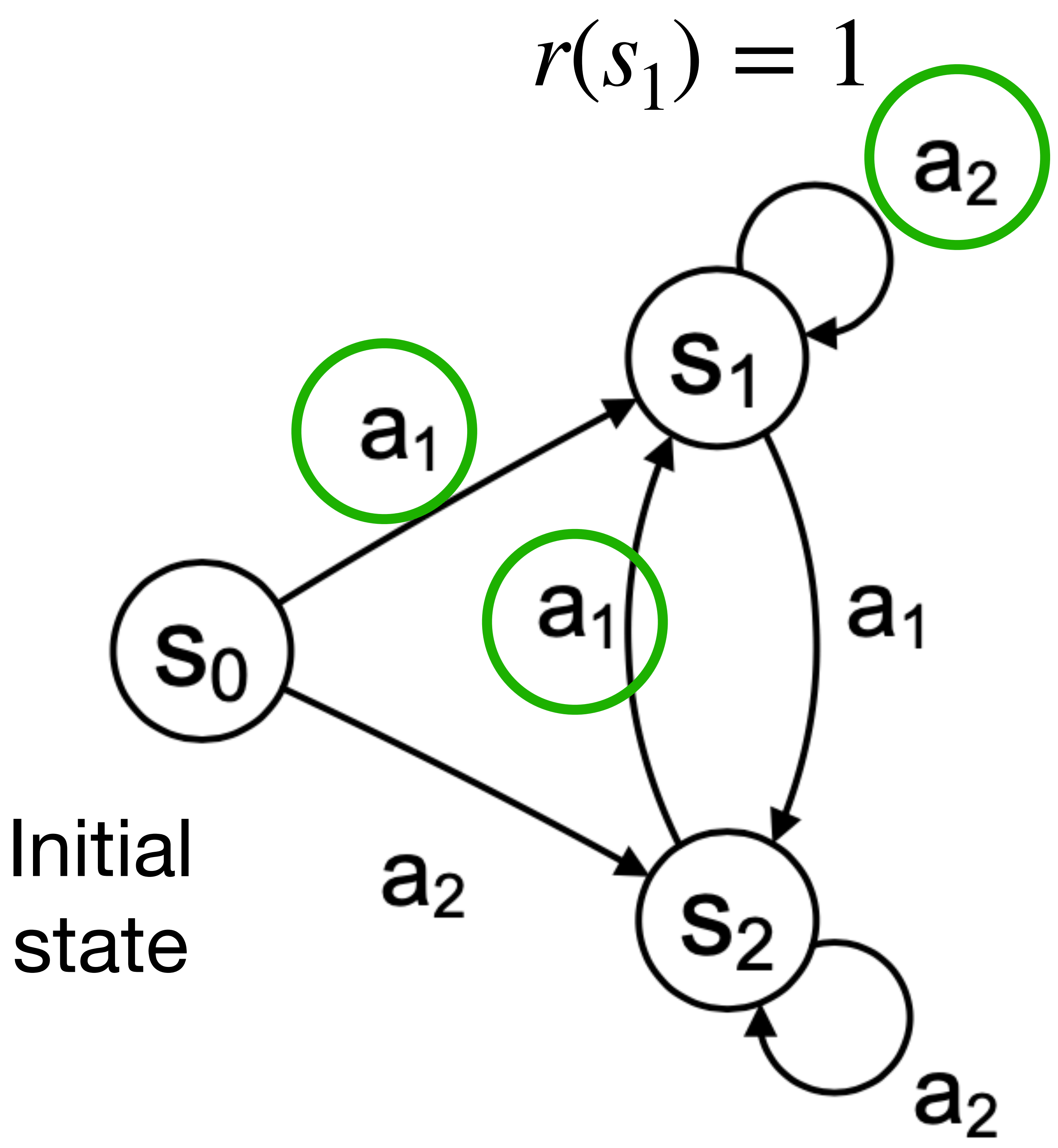


[Widrow64,Pomerleau89]

Supervised Learning Approach: Behavior Cloning



Distribution Shift Example ($|V^{\pi^\star} - V^{\widehat{\pi}}| \leq H^2\epsilon$)



Opt policy: $\pi^\star(s_0) = \pi^\star(s_2) = a_1,$
$\pi^\star(s_1) = a_2$

Under $\rho_{\pi^\star}$, trajectory is $s_0, s_1, s_1, \ldots$

$\rho_{\pi^\star}(s_h = s_2) = 0$

$V_0^{\pi^\star}(s_0) = H - 1$

Assume SL returns the policy $\widehat{\pi}$:

$$\widehat{\pi}(s_0) = \begin{cases} a_1 & \text{w/ prob } 1 - H\epsilon \\ a_2 & \text{w/ prob } H\epsilon \end{cases}, \quad \widehat{\pi}(s_1) = a_2, \widehat{\pi}(s_2) = a_2$$

This policy has good supervised learning error:

$$\mathbb{E}_{\tau \sim \rho_{\pi^\star}} \left[\frac{1}{H} \sum_{h=0}^{H-1} \mathbf{1}\left[\widehat{\pi}(s_h) \neq \pi^\star(s_h)\right] \right] = \epsilon$$

note: while $\widehat{\pi}(s_2) \neq \pi^\star(s_2)$, state s_2 is never visited under $\pi^\star$

We have quadratic degradation (in H):

$$V_0^{\widehat{\pi}}(s_0) = (1 - H\epsilon) \cdot V_0^{\pi^\star}(s_0) + H\epsilon \cdot 0 = V_0^{\pi^\star}(s_0) - \epsilon H(H-1)$$

Intuition: once we make a mistake at s_0, we end up in s_2 which is not in the training data!

The DAgger algorithm

Initialize π^0, and dataset $\mathscr{D} = \emptyset$

For $t = 0 \to T - 1$:

1. W/ π^t, generate dataset of trajectories $\mathscr{D}^t = \{\tau_1, \tau_2, \ldots\}$
where for all trajectories $s_h \sim \rho_{\pi^t}$, $a_h = \pi^\star(s_h)$
2. **Data aggregation:** $\mathscr{D} = \mathscr{D} \cup \mathscr{D}^t$
3. **Update policy via Supervised-Learning**: $\pi^{t+1} = \text{SL}\left(\mathscr{D}\right)$

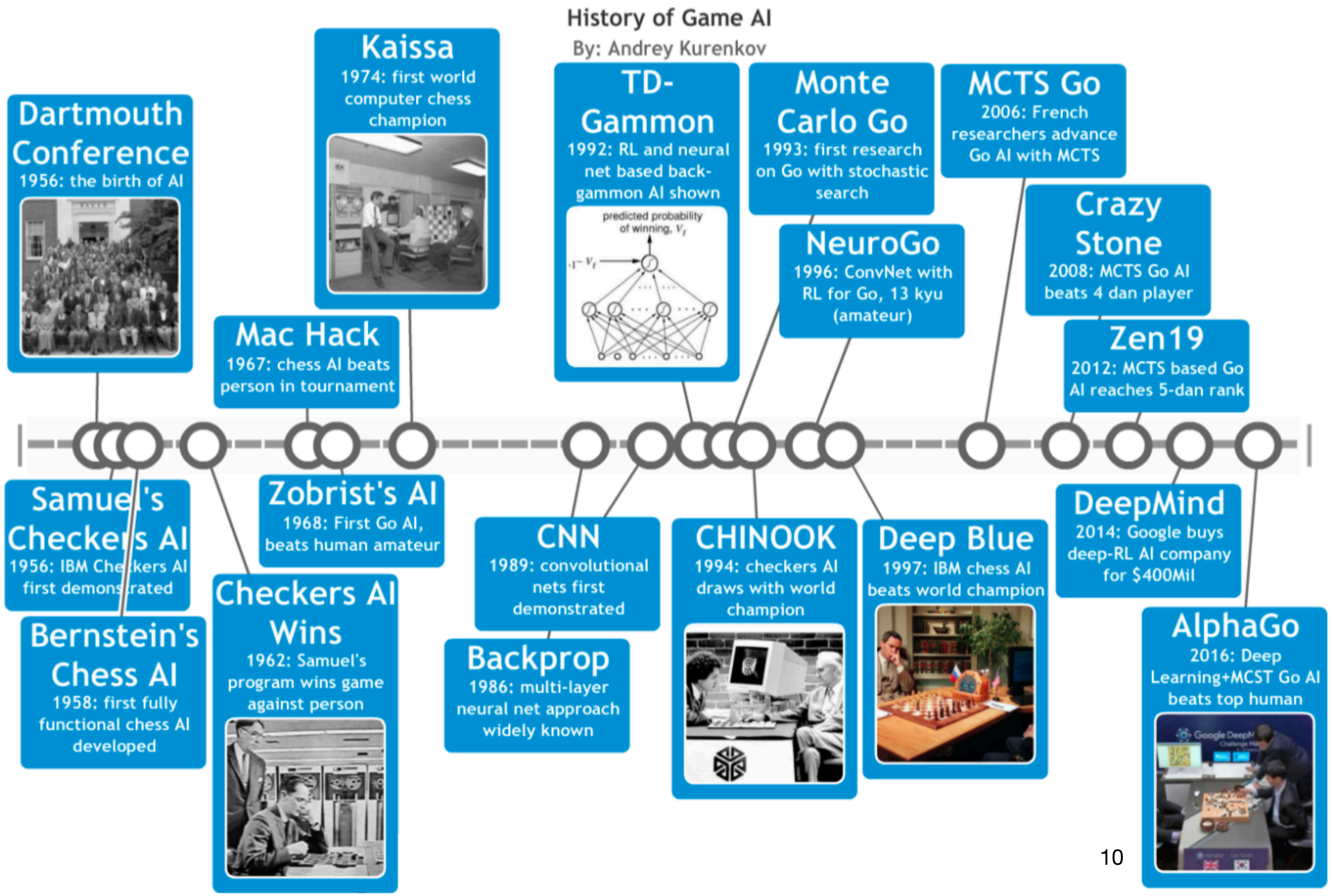
In practice, the DAgger algorithm requires less human labeled data than BC.

[Informal Theorem] Under more assumptions + assuming ϵ SL error is achievable,
the DAgger algorithm has error: $|V^{\pi^\star} - V^{\widehat{\pi}}| \leq H\epsilon$

Today

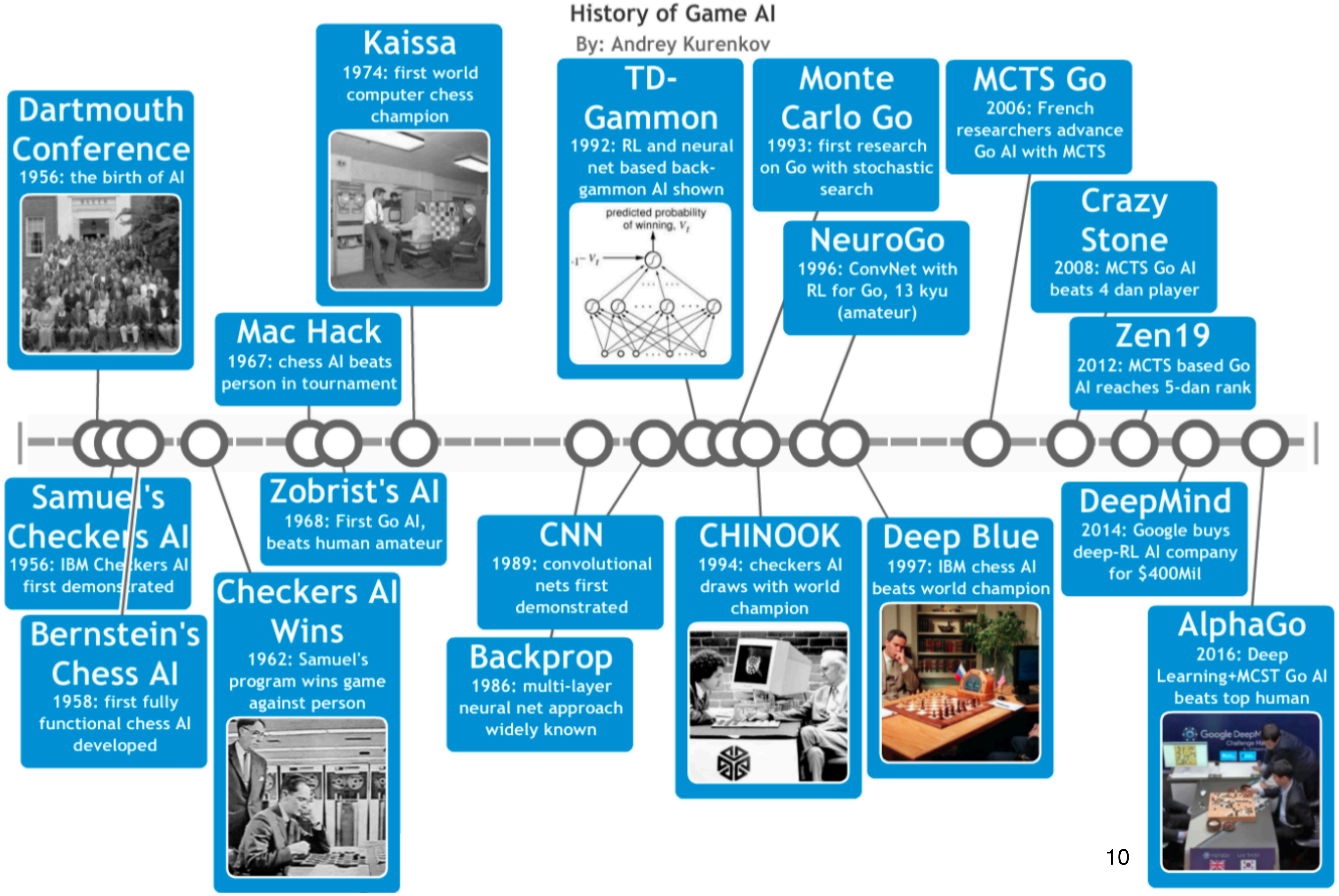
- Feedback from last lecture
- Recap
- Game Playing: AlphaBeta Search/Rule Based Systems
- MCTS
- AlphaZero and Self-Play

Fascination with AI and Games…



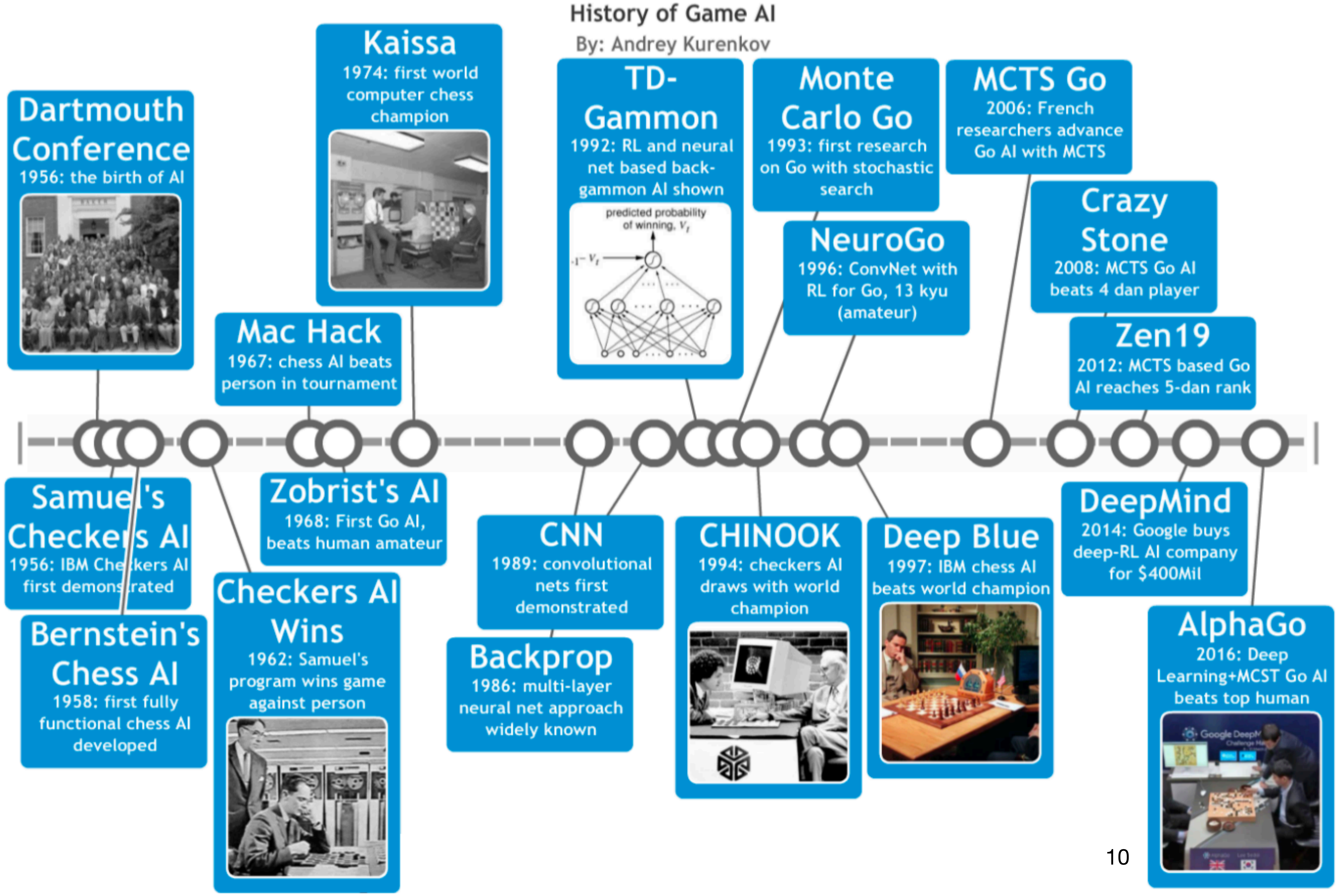
Fascination with AI and Games…

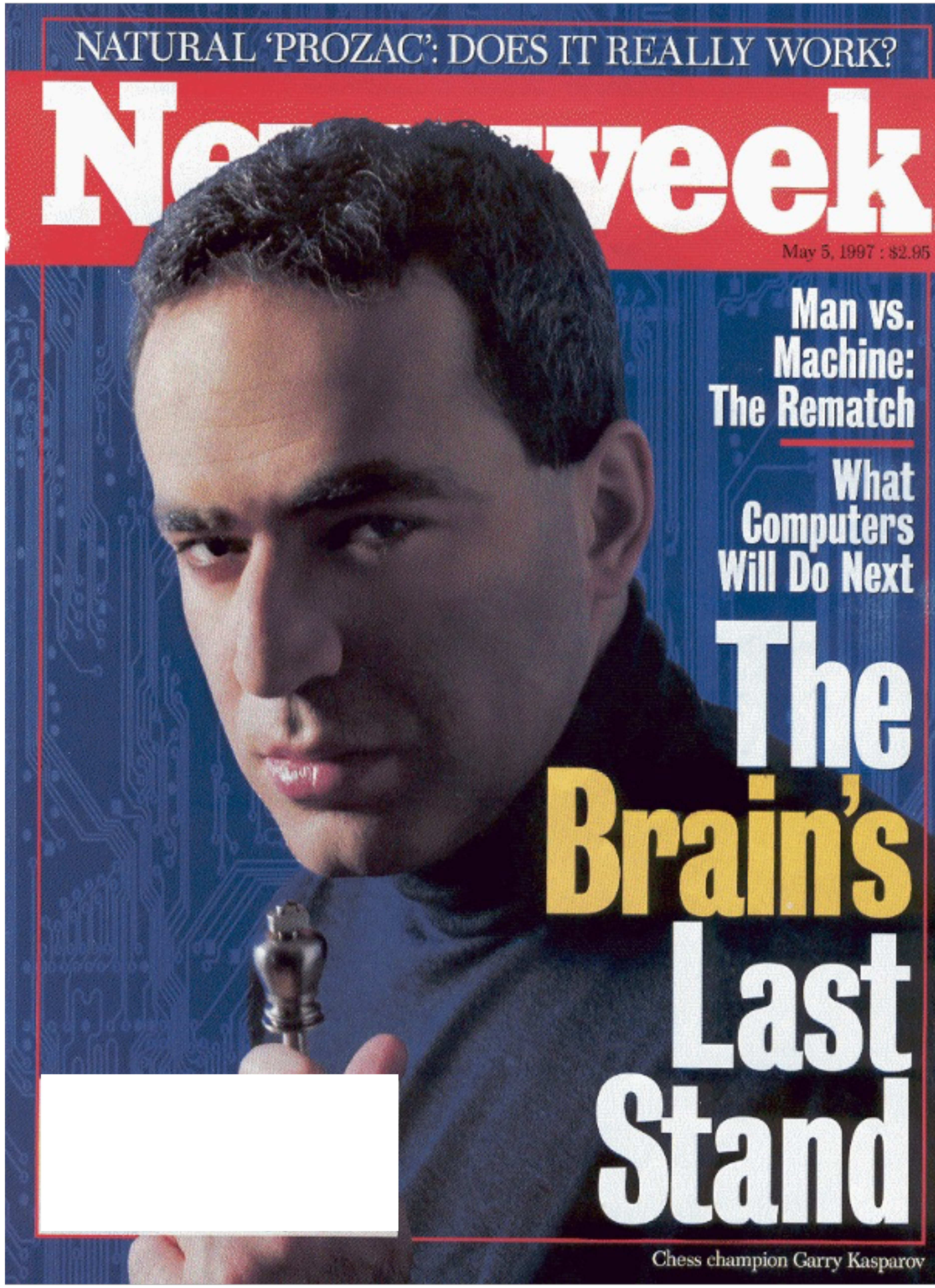
- DeepBlue v. Kasparov (1997)
  - winning in chess wasn't a good indicator of "progress in AI"



Fascination with AI and Games…

- DeepBlue v. Kasparov (1997)
  - winning in chess wasn't a good indicator of "progress in AI"





Two-player, deterministic games

We will focus on games that are:

Two-player, deterministic games

We will focus on games that are:

- deterministic
- two-player (alternating turns)
- zero sum (one player wins and the other loses)
- fully observable (by both players)
- stationary (only game state and whose turn it is matters)

Two-player, deterministic games

We will focus on games that are:

- deterministic
- two-player (alternating turns)
- zero sum (one player wins and the other loses)
- fully observable (by both players)
- stationary (only game state and whose turn it is matters)

E.g.,
- Tic-tac-toe
- Chess
- Go

Two-player, deterministic games

We will focus on games that are:

- deterministic
- two-player (alternating turns)
- zero sum (one player wins and the other loses)
- fully observable (by both players)
- stationary (only game state and whose turn it is matters)

E.g.,

- Tic-tac-toe
- Chess
- Go

Notation:

- Game states S, initial state $s_0 \in S$
- Set of actions available in state s: $A(s)$
- Dynamics $P(s, a) \in S$
- Maximum game length H
- Score at terminal state $r(s)$ (sign determines winner)

Two-player, deterministic games

We will focus on games that are:

- deterministic
- two-player (alternating turns)
- zero sum (one player wins and the other loses)
- fully observable (by both players)
- stationary (only game state and whose turn it is matters)

E.g.,

- Tic-tac-toe
- Chess
- Go

Notation:

- Game states S, initial state $s_0 \in S$
- Set of actions available in state s: $A(s)$
- Dynamics $P(s, a) \in S$
- Maximum game length H
- Score at terminal state $r(s)$ (sign determines winner)

Still an MDP, but two competing players make it a bit different than earlier RL setup

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C
then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C
then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C
then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C
then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C
then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C
then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C
then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C
then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C
then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C
then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C
then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Simple example of min-max search

$H = 2$, player 1 takes action A, B, or C
then player 2 takes action D, E, F

Outcome r(s)	D	E	F
A	4	-2	5
B	-3	3	1
C	0	3	-1

Game tree:

Basically dynamic programming! Numbers in boxes are value function $V(s)$

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^{\star}(s)$
- $\beta(s)$ is upper-bound for $V^{\star}(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^{\star}(s)$
- $\beta(s)$ is upper-bound for $V^{\star}(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^{\star}(s)$
- $\beta(s)$ is upper-bound for $V^{\star}(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^{\star}(s)$
- $\beta(s)$ is upper-bound for $V^{\star}(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^{\star}(s)$
- $\beta(s)$ is upper-bound for $V^{\star}(s)$
- Bounds sometimes allow pruning

Alpha-beta search

Pruning can speed up search without losing exactness

- $\alpha(s)$ is lower-bound for $V^\star(s)$
- $\beta(s)$ is upper-bound for $V^\star(s)$
- Bounds sometimes allow pruning

The order that actions are considered can matter a lot

Today

- ✓ Feedback from last lecture
- ✓ Recap
- ✓ Game Playing: AlphaBeta Search/Rule Based Systems
- MCTS
- AlphaZero and Self-Play

Monte Carlo Tree Search (MCTS)

Monte Carlo Tree Search (MCTS)

For now, assume game outcome just win or lose: $r(s) \in \{-1, 1\}$

Monte Carlo Tree Search (MCTS)

For now, assume game outcome just win or lose: $r(s) \in \{-1, 1\}$

- Alpha-beta search evaluates non-leaf nodes via a min-max approach

Monte Carlo Tree Search (MCTS)

For now, assume game outcome just win or lose: $r(s) \in \{-1,1\}$

- Alpha-beta search evaluates non-leaf nodes via a min-max approach
  - Even with pruning, requires searching a LOT of paths down tree

Monte Carlo Tree Search (MCTS)

For now, assume game outcome just win or lose: $r(s) \in \{-1,1\}$

- Alpha-beta search evaluates non-leaf nodes via a min-max approach
  - Even with pruning, requires searching a LOT of paths down tree
- Idea of MCTS is evaluate non-leaf nodes via sampling (Monte Carlo)

Monte Carlo Tree Search (MCTS)

For now, assume game outcome just win or lose: $r(s) \in \{-1, 1\}$

- Alpha-beta search evaluates non-leaf nodes via a min-max approach
    - Even with pruning, requires searching a LOT of paths down tree
- Idea of MCTS is evaluate non-leaf nodes via sampling (Monte Carlo)
- High-level: at each iteration, **MCTS** does the following

Monte Carlo Tree Search (MCTS)

For now, assume game outcome just win or lose: $r(s) \in \{-1,1\}$

- Alpha-beta search evaluates non-leaf nodes via a min-max approach
  - Even with pruning, requires searching a LOT of paths down tree
- Idea of MCTS is evaluate non-leaf nodes via sampling (Monte Carlo)
- High-level: at each iteration, **MCTS** does the following
  - Defines a game-playing strategy (policy for both players) that is a simple function of a set of statistics computed from existing samples

Monte Carlo Tree Search (MCTS)

For now, assume game outcome just win or lose: $r(s) \in \{-1,1\}$

- Alpha-beta search evaluates non-leaf nodes via a min-max approach
  - Even with pruning, requires searching a LOT of paths down tree
- Idea of MCTS is evaluate non-leaf nodes via sampling (Monte Carlo)
- High-level: at each iteration, **MCTS** does the following
  - Defines a game-playing strategy (policy for both players) that is a simple function of a set of statistics computed from existing samples
  - Plays the game to completion via this strategy and records outcome

Monte Carlo Tree Search (MCTS)

For now, assume game outcome just win or lose: $r(s) \in \{-1,1\}$

- Alpha-beta search evaluates non-leaf nodes via a min-max approach
  - Even with pruning, requires searching a LOT of paths down tree
- Idea of MCTS is evaluate non-leaf nodes via sampling (Monte Carlo)
- High-level: at each iteration, **MCTS** does the following
  - Defines a game-playing strategy (policy for both players) that is a simple function of a set of statistics computed from existing samples
  - Plays the game to completion via this strategy and records outcome
  - Updates statistics used to define game-playing strategy

Monte Carlo Tree Search (MCTS)

For now, assume game outcome just win or lose: $r(s) \in \{-1, 1\}$

- Alpha-beta search evaluates non-leaf nodes via a min-max approach
  - Even with pruning, requires searching a LOT of paths down tree
- Idea of MCTS is evaluate non-leaf nodes via sampling (Monte Carlo)
- High-level: at each iteration, **MCTS** does the following
  - Defines a game-playing strategy (policy for both players) that is a simple function of a set of statistics computed from existing samples
  - Plays the game to completion via this strategy and records outcome
  - Updates statistics used to define game-playing strategy
- Strategy gradually improves with more iterations/samples, so can fit in any computational budget

Monte Carlo Tree Search (MCTS)

For now, assume game outcome just win or lose: $r(s) \in \{-1,1\}$

- Alpha-beta search evaluates non-leaf nodes via a min-max approach
  - Even with pruning, requires searching a LOT of paths down tree
- Idea of MCTS is evaluate non-leaf nodes via sampling (Monte Carlo)
- High-level: at each iteration, **MCTS** does the following
  - Defines a game-playing strategy (policy for both players) that is a simple function of a set of statistics computed from existing samples
  - Plays the game to completion via this strategy and records outcome
  - Updates statistics used to define game-playing strategy
- Strategy gradually improves with more iterations/samples, so can fit in any computational budget
- Samples are concentrated around more promising strategies

“Pure” MCTS Algorithm

“Pure” MCTS Algorithm

Input: game state (“root node” R), #iterations N, exploration constant C

“Pure” MCTS Algorithm

Input: game state (“root node” R), #iterations N, exploration constant C

For iteration $t = 1, \ldots, N$

"Pure" MCTS Algorithm

Input: game state ("root node" R), #iterations N, exploration constant C

For iteration $t = 1, \ldots, N$

1. Obtain the t-th sample trajectory: Starting at R, while current state $s \notin$ {win, lose}

"Pure" MCTS Algorithm

Input: game state ("root node" R), #iterations N, exploration constant C

For iteration $t = 1, \ldots, N$

1. Obtain the t-th sample trajectory: Starting at R, while current state $s \notin$ {win, lose}
   a. For player $X \in \{0,1\}$, at current state s, let $s' = P(s, a)$ and define:

$$\text{UCBscore}_t(s, a) = \frac{\text{\#wins for player } X \text{ from } s'}{\text{\#visits to } s'} + C\sqrt{\frac{\log(\text{\#visits to } s)}{\text{\#visits to } s'}}$$

“Pure” MCTS Algorithm

Input: game state (“root node” R), #iterations N, exploration constant C

For iteration $t = 1, \ldots, N$

1. Obtain the t-th sample trajectory: Starting at R, while current state $s \notin$ {win, lose}
   a. For player $X \in \{0,1\}$, at current state s, let $s' = P(s, a)$ and define:
   $$\text{UCBscore}_t(s, a) = \frac{\text{\#wins for player } X \text{ from } s'}{\text{\#visits to } s'} + C\sqrt{\frac{\log(\text{\#visits to } s)}{\text{\#visits to } s'}}$$
   b. “Take” action:
   $$\hat{a} = \arg\max_a \text{UCBscore}(s, a)$$

“Pure” MCTS Algorithm

Input: game state (“root node” R), #iterations N, exploration constant C

For iteration $t = 1, \ldots, N$

1. Obtain the t-th sample trajectory: Starting at R, while current state $s \notin$ {win, lose}
   a. For player $X \in \{0,1\}$, at current state s, let $s' = P(s, a)$ and define:

   $$\text{UCBscore}_t(s, a) = \frac{\text{\#wins for player } X \text{ from } s'}{\text{\#visits to } s'} + C\sqrt{\frac{\log(\text{\#visits to } s)}{\text{\#visits to } s'}}$$

   b. “Take” action:

   $$\hat{a} = \arg\max_a \text{UCBscore}(s, a)$$

2. Update stats: For all visited states s' in this trajectory,

"Pure" MCTS Algorithm

Input: game state ("root node" R), #iterations N, exploration constant C

For iteration $t = 1, \ldots, N$

1. Obtain the t-th sample trajectory: Starting at R, while current state $s \notin$ {win, lose}
   a. For player $X \in \{0,1\}$, at current state s, let $s' = P(s, a)$ and define:
   $$\text{UCBscore}_t(s, a) = \frac{\#\text{wins for player } X \text{ from } s'}{\#\text{visits to } s'} + C\sqrt{\frac{\log(\#\text{visits to } s)}{\#\text{visits to } s'}}$$
   b. "Take" action:
   $$\hat{a} = \arg\max_a \text{UCBscore}(s, a)$$
2. Update stats: For all visited states s' in this trajectory,
   c. update visit counts:
   $$[\#\text{visits to } s'] = [\#\text{visits to } s'] + 1$$

“Pure” MCTS Algorithm

Input: game state (“root node” R), #iterations N, exploration constant C

For iteration $t = 1, \ldots, N$

1. Obtain the t-th sample trajectory: Starting at R, while current state $s \notin$ {win, lose}
   a. For player $X \in \{0,1\}$, at current state s, let $s' = P(s, a)$ and define:

$$\text{UCBscore}_t(s, a) = \frac{\text{\#wins for player } X \text{ from } s'}{\text{\#visits to } s'} + C\sqrt{\frac{\log(\text{\#visits to } s)}{\text{\#visits to } s'}}$$

   b. “Take” action:

$$\hat{a} = \arg\max_a \text{UCBscore}(s, a)$$

2. Update stats: For all visited states s' in this trajectory,
   c. update visit counts:

$$[\text{\#visits to } s'] = [\text{\#visits to } s'] + 1$$

   d. for winner X and if s was visited by X:

$$[\text{\#wins for X at } s'] = [\text{\#wins for X at } s'] + 1$$

“Pure” MCTS Algorithm

Input: game state (“root node” R), #iterations N, exploration constant C

For iteration $t = 1,\ldots,N$

1. Obtain the t-th sample trajectory: Starting at R, while current state $s \notin$ {win, lose}
   a. For player $X \in \{0,1\}$, at current state s, let $s' = P(s, a)$ and define:
   $$\text{UCBscore}_t(s, a) = \frac{\text{\#wins for player } X \text{ from } s'}{\text{\#visits to } s'} + C\sqrt{\frac{\log(\text{\#visits to } s)}{\text{\#visits to } s'}}$$
   b. “Take” action:
   $$\hat{a} = \arg\max_a \text{UCBscore}(s, a)$$
2. Update stats: For all visited states s' in this trajectory,
   c. update visit counts:
   $$[\text{\#visits to } s'] = [\text{\#visits to } s'] + 1$$
   d. for winner X and if s was visited by X:
   $$[\text{\#wins for X at } s'] = [\text{\#wins for X at } s'] + 1$$

(data structure: only need to keep track of stats at visited states)

“Pure” MCTS Algorithm

Input: game state (“root node” R), #iterations N, exploration constant C

For iteration $t = 1, \ldots, N$

1. Obtain the t-th sample trajectory: Starting at R, while current state $s \notin$ {win, lose}
   a. For player $X \in \{0,1\}$, at current state s, let $s' = P(s, a)$ and define:
   $$\text{UCBscore}_t(s, a) = \frac{\text{\#wins for player } X \text{ from } s'}{\text{\#visits to } s'} + C\sqrt{\frac{\log(\text{\#visits to } s)}{\text{\#visits to } s'}}$$
   b. “Take” action:
   $$\hat{a} = \arg\max_a \text{UCBscore}(s, a)$$
2. Update stats: For all visited states s' in this trajectory,
   c. update visit counts:
   $$[\text{\#visits to } s'] = [\text{\#visits to } s'] + 1$$
   d. for winner X and if s was visited by X:
   $$[\text{\#wins for X at } s'] = [\text{\#wins for X at } s'] + 1$$

(data structure: only need to keep track of stats at visited states)

Output: return the action $\hat{a} = \arg\max_a \text{UCBscore}_N(R, a)$

Improving MCTS

Improving MCTS

- MCTS re-runs at every game step (root node gets updated to current state)

Improving MCTS

- MCTS re-runs at every game step (root node gets updated to current state)
- “Pure” MCTS can work well for small games, but what can go wrong?

Improving MCTS

- MCTS re-runs at every game step (root node gets updated to current state)
- “Pure” MCTS can work well for small games, but what can go wrong?
- For large games, most states never visited…
  so UCB basically just samples trajectories randomly after a certain point!

Improving MCTS

- MCTS re-runs at every game step (root node gets updated to current state)
- “Pure” MCTS can work well for small games, but what can go wrong?
- For large games, most states never visited…
  so UCB basically just samples trajectories randomly after a certain point!
- **Solution**:

Improving MCTS

- MCTS re-runs at every game step (root node gets updated to current state)
- "Pure" MCTS can work well for small games, but what can go wrong?
- For large games, most states never visited…
  so UCB basically just samples trajectories randomly after a certain point!
- **Solution**:
  - Fix a strategy π and a look-ahead horizon T

Improving MCTS

- MCTS re-runs at every game step (root node gets updated to current state)
- "Pure" MCTS can work well for small games, but what can go wrong?
- For large games, most states never visited…
  so UCB basically just samples trajectories randomly after a certain point!
- **Solution**:
  - Fix a strategy π and a look-ahead horizon T
  - Only use UCB strategy for choosing actions for T steps, use π after

Improving MCTS

- MCTS re-runs at every game step (root node gets updated to current state)
- "Pure" MCTS can work well for small games, but what can go wrong?
- For large games, most states never visited…
  so UCB basically just samples trajectories randomly after a certain point!
- **Solution**:
  - Fix a strategy π and a look-ahead horizon T
  - Only use UCB strategy for choosing actions for T steps, use π after
  - Note since MCTS re-runs at every game step, π's use gets later and later

Improving MCTS

- MCTS re-runs at every game step (root node gets updated to current state)
- "Pure" MCTS can work well for small games, but what can go wrong?
- For large games, most states never visited…
  so UCB basically just samples trajectories randomly after a certain point!
- **Solution**:
  - Fix a strategy π and a look-ahead horizon T
  - Only use UCB strategy for choosing actions for T steps, use π after
  - Note since MCTS re-runs at every game step, π's use gets later and later
- Need a good strategy π… or, a good value function approximation $\hat{V}(s)$:
  After T steps, instead of using π, stop and record $\hat{V}(s)$ as game outcome

Improving MCTS

- MCTS re-runs at every game step (root node gets updated to current state)
- "Pure" MCTS can work well for small games, but what can go wrong?
- For large games, most states never visited…
  so UCB basically just samples trajectories randomly after a certain point!
- **Solution**:
  - Fix a strategy π and a look-ahead horizon T
  - Only use UCB strategy for choosing actions for T steps, use π after
  - Note since MCTS re-runs at every game step, π's use gets later and later
- Need a good strategy π… or, a good value function approximation $\hat{V}(s)$:
  After T steps, instead of using π, stop and record $\hat{V}(s)$ as game outcome
- $\hat{V}(s)$ could be learned from offline/expert data and improved online

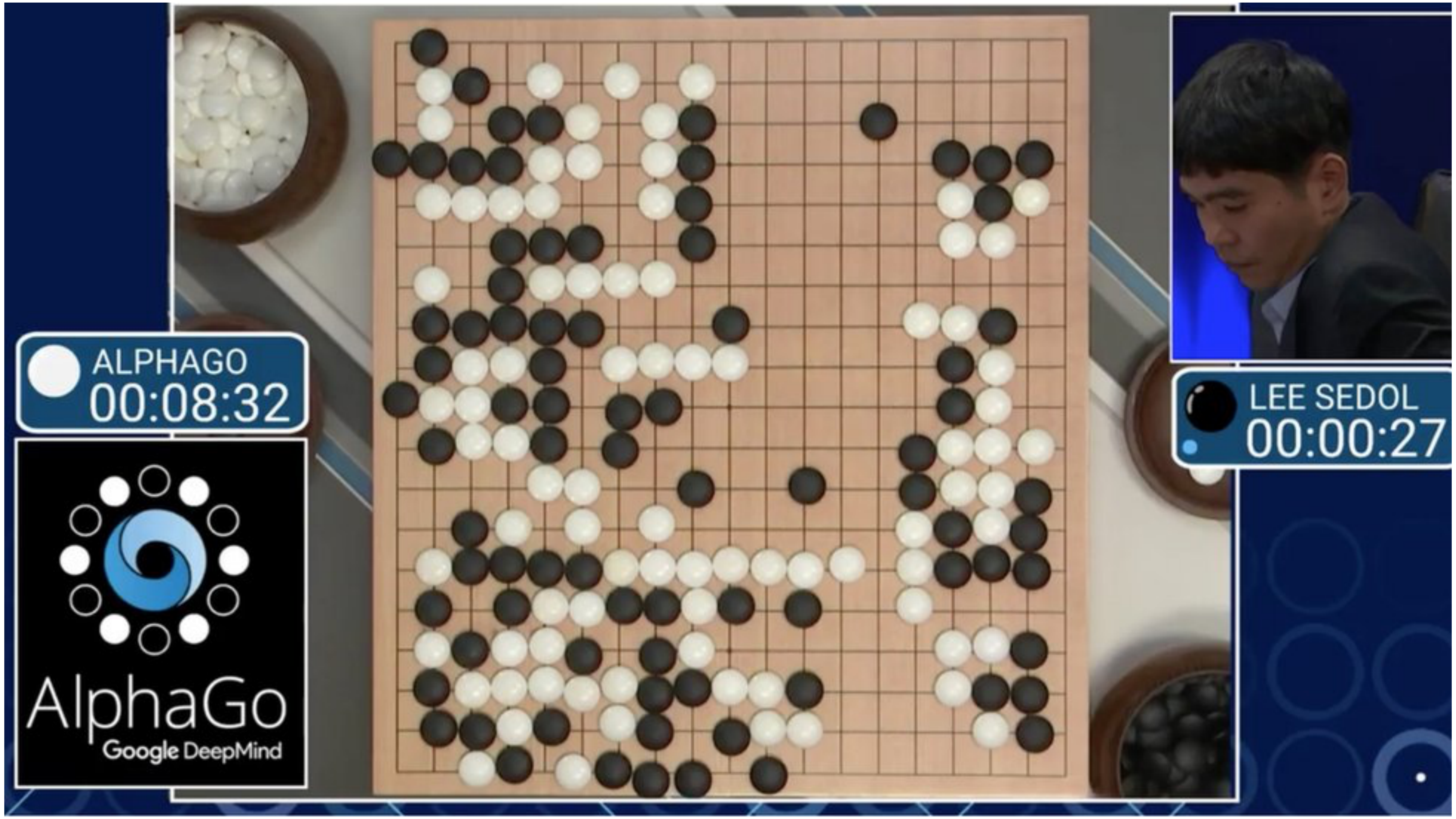
Today

- ✓ Feedback from last lecture
- ✓ Recap
- ✓ Game Playing: AlphaBeta Search/Rule Based Systems
- ✓ MCTS
- AlphaZero and Self-Play

AlphaGo

AlphaGo versus Lee Sedol
4–1

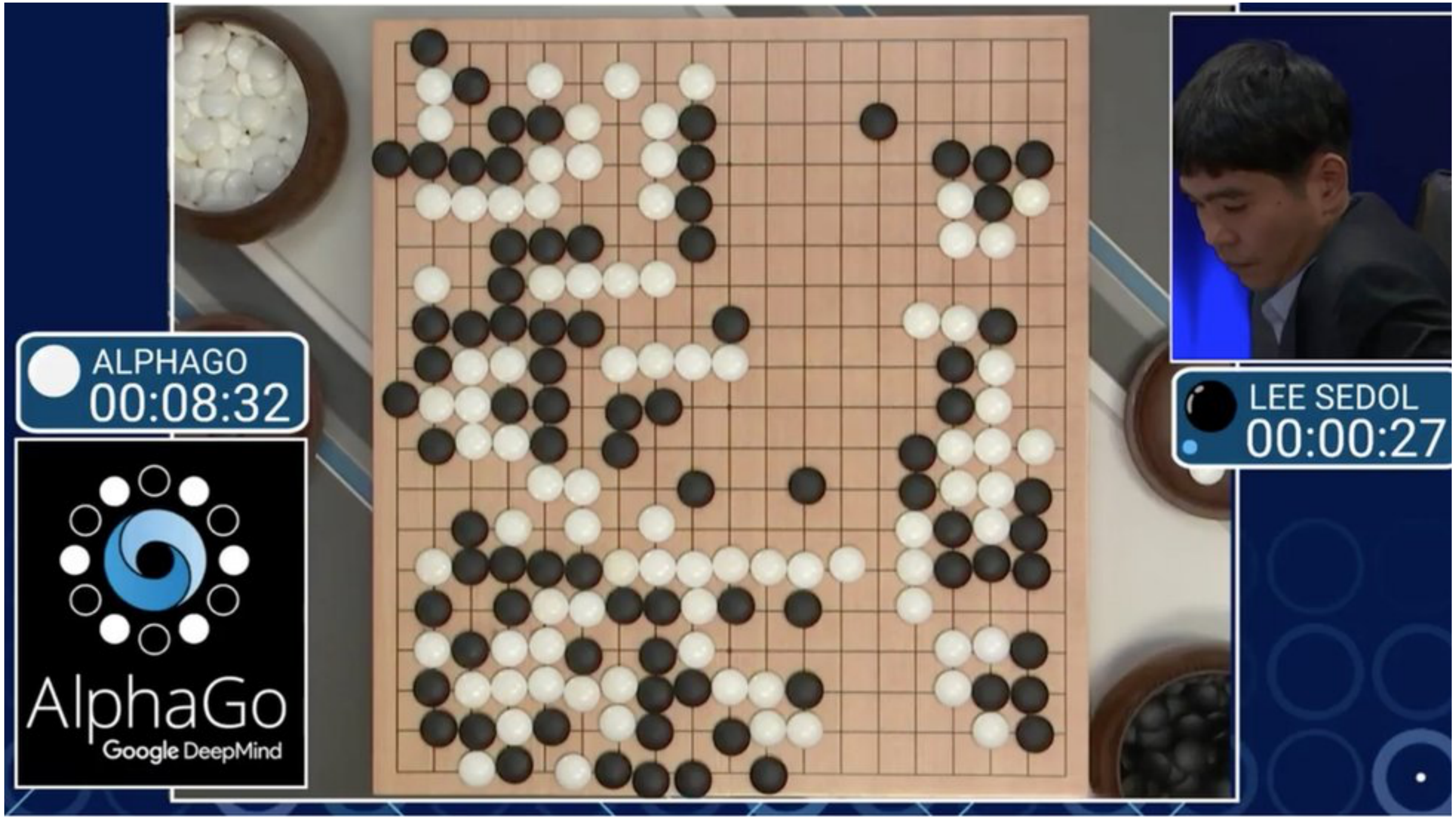
Seoul, South Korea, 9–15 March 2016	
Game one	AlphaGo W+R
Game two	AlphaGo B+R
Game three	AlphaGo W+R
Game four	Lee Sedol W+R
Game five	AlphaGo W+R



AlphaGo

AlphaGo versus Lee Sedol
4–1

Seoul, South Korea, 9–15 March 2016	
Game one	AlphaGo W+R
Game two	AlphaGo B+R
Game three	AlphaGo W+R
Game four	Lee Sedol W+R
Game five	AlphaGo W+R



- Lots of moving parts:
  - **Imitation Learning:** first, the algo estimates the values from historical games.
  - It then uses an **MCTS-stye lookahead** with **learned value functions**.
- **AlphaZero** (2017) is a simpler more successful approach that uses self-play

AlphaZero

AlphaZero

- AlphaZero: MCTS + DeepLearning + self-play

AlphaZero

- AlphaZero: MCTS + DeepLearning + self-play
  - MCTS subroutine has a value network and policy network

AlphaZero

- AlphaZero: MCTS + DeepLearning + self-play
  - MCTS subroutine has a value network and policy network
    - a **value network** estimating the value for the state of the board $\hat{V}_\theta(s)$

AlphaZero

- AlphaZero: MCTS + DeepLearning + self-play
  - MCTS subroutine has a value network and policy network
    - a **value network** estimating the value for the state of the board $\hat{V}_\theta(s)$
    - A **policy network** $\pi_\theta(a \mid s)$ that is a probability vector over all possible actions

AlphaZero

- AlphaZero: MCTS + DeepLearning + self-play
  - MCTS subroutine has a value network and policy network
    - a **value network** estimating the value for the state of the board $\hat{V}_\theta(s)$
    - A **policy network** $\pi_\theta(a \mid s)$ that is a probability vector over all possible actions
  - Use these for MCTS, then play agent against self and use self-play data to learn better θ; iterate

AlphaZero MCTS subroutine (without self-play)

AlphaZero MCTS subroutine (without self-play)

Input: game state ("root node" R), #iterations N, exploration constant C, look-ahead horizon T, **value network** $\hat{V}_\theta(s)$, **policy network** $\pi_\theta(a \mid s)$

AlphaZero MCTS subroutine (without self-play)

Input: game state ("root node" R), #iterations N, exploration constant C, look-ahead horizon T, **value network** $\hat{V}_\theta(s)$, **policy network** $\pi_\theta(a \mid s)$

For iteration $t = 1 : N$

AlphaZero MCTS subroutine (without self-play)

Input: game state ("root node" R), #iterations N, exploration constant C, look-ahead horizon T, **value network** $\hat{V}_\theta(s)$, **policy network** $\pi_\theta(a \mid s)$

For iteration $t = 1 : N$

1. Obtain the t-th sample trajectory: For T steps starting from R,

AlphaZero MCTS subroutine (without self-play)

Input: game state ("root node" R), #iterations N, exploration constant C, look-ahead horizon T, **value network** $\hat{V}_\theta(s)$, **policy network** $\pi_\theta(a \mid s)$

For iteration $t = 1 : N$

1. Obtain the t-th sample trajectory: For T steps starting from R,
   a. For player $X \in \{0,1\}$, at current state s, define $s' = P(s, a)$ and define:

$$\text{UCBscore}_t(s, a) = \overline{\hat{V}}(s') \cdot (-1)^X + C \cdot \pi_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{\#visits to s})}{\text{\#visits to } s'}}$$

AlphaZero MCTS subroutine (without self-play)

Input: game state ("root node" R), #iterations N, exploration constant C, look-ahead horizon T, **value network** $\hat{V}_\theta(s)$, **policy network** $\pi_\theta(a \mid s)$

For iteration $t = 1 : N$

1. Obtain the t-th sample trajectory: For T steps starting from R,
   a. For player $X \in \{0,1\}$, at current state s, define $s' = P(s, a)$ and define:

   $$\text{UCBscore}_t(s, a) = \overline{\hat{V}}(s') \cdot (-1)^X + C \cdot \pi_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{\#visits to s})}{\text{\#visits to } s'}}$$

   b. "Take" action:

   $$\hat{a} = \arg\max_a \text{UCBscore}_t(s, a)$$

AlphaZero MCTS subroutine (without self-play)

Input: game state ("root node" R), #iterations N, exploration constant C, look-ahead horizon T, **value network** $\hat{V}_\theta(s)$, **policy network** $\pi_\theta(a \mid s)$

For iteration $t = 1 : N$

1. Obtain the t-th sample trajectory: For T steps starting from R,
   a. For player $X \in \{0,1\}$, at current state s, define $s' = P(s, a)$ and define:

      $$\text{UCBscore}_t(s, a) = \overline{\hat{V}}(s') \cdot (-1)^X + C \cdot \pi_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{\#visits to s})}{\text{\#visits to } s'}}$$

   b. "Take" action:

      $$\hat{a} = \arg\max_a \text{UCBscore}_t(s, a)$$

2. Update stats: For all visited states s' in this "roll-out", letting s_T be the last sampled state

AlphaZero MCTS subroutine (without self-play)

Input: game state (“root node” R), #iterations N, exploration constant C, look-ahead horizon T, **value network** $\hat{V}_\theta(s)$, **policy network** $\pi_\theta(a \mid s)$

For iteration $t = 1 : N$

1. Obtain the t-th sample trajectory: For T steps starting from R,
   a. For player $X \in \{0,1\}$, at current state s, define $s' = P(s, a)$ and define:

      $$\text{UCBscore}_t(s, a) = \overline{\hat{V}}(s') \cdot (-1)^X + C \cdot \pi_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{\#visits to s})}{\text{\#visits to } s'}}$$

   b. “Take” action:

      $$\hat{a} = \arg\max_a \text{UCBscore}_t(s, a)$$

2. Update stats: For all visited states s' in this “roll-out”, letting s_T be the last sampled state
   c. Update counts: $[\text{\#visits to } s'] = [\text{\#visits to } s'] + 1$

AlphaZero MCTS subroutine (without self-play)

Input: game state ("root node" R), #iterations N, exploration constant C, look-ahead horizon T, **value network** $\hat{V}_\theta(s)$, **policy network** $\pi_\theta(a \mid s)$

For iteration $t = 1 : N$

1. Obtain the t-th sample trajectory: For T steps starting from R,
   a. For player $X \in \{0,1\}$, at current state s, define $s' = P(s, a)$ and define:
   $$\text{UCBscore}_t(s, a) = \overline{\hat{V}}(s') \cdot (-1)^X + C \cdot \pi_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{\#visits to s})}{\text{\#visits to } s'}}$$
   b. "Take" action:
   $$\hat{a} = \arg\max_a \text{UCBscore}_t(s, a)$$
2. Update stats: For all visited states s' in this "roll-out", letting s_T be the last sampled state
   c. Update counts: $[\text{\#visits to } s'] = [\text{\#visits to } s'] + 1$
   d. Update average value estimate: $\overline{\hat{V}}(s') \leftarrow \frac{[\text{\#visits to } s']}{[\text{\#visits to } s'] + 1} \overline{\hat{V}}(s') + \frac{1}{[\text{\#visits to } s'] + 1} \hat{V}_\theta(s_T)$

AlphaZero MCTS subroutine (without self-play)

Input: game state ("root node" R), #iterations N, exploration constant C, look-ahead horizon T, **value network** $\hat{V}_\theta(s)$, **policy network** $\pi_\theta(a \mid s)$

For iteration $t = 1 : N$

1. Obtain the t-th sample trajectory: For T steps starting from R,
   a. For player $X \in \{0,1\}$, at current state s, define $s' = P(s, a)$ and define:
   $$\text{UCBscore}_t(s,a) = \overline{\hat{V}}(s') \cdot (-1)^X + C \cdot \pi_\theta(a \mid s) \cdot \sqrt{\frac{\log(\text{\#visits to s})}{\text{\#visits to } s'}}$$
   b. "Take" action:
   $$\hat{a} = \arg\max_a \text{UCBscore}_t(s,a)$$
2. Update stats: For all visited states s' in this "roll-out", letting s_T be the last sampled state
   c. Update counts: $[\text{\#visits to } s'] = [\text{\#visits to } s'] + 1$
   d. Update average value estimate: $\overline{\hat{V}}(s') \leftarrow \frac{[\text{\#visits to } s']}{[\text{\#visits to } s'] + 1} \overline{\hat{V}}(s') + \frac{1}{[\text{\#visits to } s'] + 1} \hat{V}_\theta(s_T)$

Output: return the action $\hat{a} = \arg\max_a \text{UCBscore}_N(R, a)$

Self-play

Self-play

- Iterate the following:

Self-play

- Iterate the following:
  - Self-play: Play against self M times using current MCTS strategy

Self-play

- Iterate the following:
  - Self-play: Play against self M times using current MCTS strategy
  - Supervised Learning: Use M self-play game trajectories to update:

Self-play

- Iterate the following:
  - Self-play: Play against self M times using current MCTS strategy
  - Supervised Learning: Use M self-play game trajectories to update:
    - $\hat{V}_\theta$ with squared error loss wrt game outcomes (similar to in fitted VI or baseline estimation)

Self-play

- Iterate the following:
  - Self-play: Play against self M times using current MCTS strategy
  - Supervised Learning: Use M self-play game trajectories to update:
    - $\hat{V}_\theta$ with squared error loss wrt game outcomes (similar to in fitted VI or baseline estimation)
    - π_θ with negative log likelihood loss wrt actions taken in game (similar to in BC)

Self-play

- Iterate the following:
  - Self-play: Play against self M times using current MCTS strategy
  - Supervised Learning: Use M self-play game trajectories to update:
    - $\hat{V}_\theta$ with squared error loss wrt game outcomes (similar to in fitted VI or baseline estimation)
    - π_θ with negative log likelihood loss wrt actions taken in game (similar to in BC)
    - In practice, combine loss functions into single SL problem with shared θ

Self-play

- Iterate the following:
  - Self-play: Play against self M times using current MCTS strategy
  - Supervised Learning: Use M self-play game trajectories to update:
    - $\hat{V}_\theta$ with squared error loss wrt game outcomes (similar to in fitted VI or baseline estimation)
    - π_θ with negative log likelihood loss wrt actions taken in game (similar to in BC)
    - In practice, combine loss functions into single SL problem with shared θ

- AlphaZero uses no historical data, only self-play

Self-play

- Iterate the following:
  - Self-play: Play against self M times using current MCTS strategy
  - Supervised Learning: Use M self-play game trajectories to update:
    - $\hat{V}_\theta$ with squared error loss wrt game outcomes (similar to in fitted VI or baseline estimation)
    - π_θ with negative log likelihood loss wrt actions taken in game (similar to in BC)
    - In practice, combine loss functions into single SL problem with shared θ

- AlphaZero uses no historical data, only self-play
- Performance improvement was pretty astronomical!

Chess [edit]

In AlphaZero's chess match against Stockfish 8 (2016 TCEC world champion), each program was given one minute per move. Stockfish was allocated 64 threads and a hash size of 1 GB,[1] a setting that Stockfish's Tord Romstad later criticized as suboptimal.[7][note 1] AlphaZero was trained on chess for a total of nine hours before the match. During the match, AlphaZero ran on a single machine with four application-specific TPUs. In 100 games from the normal starting position, AlphaZero won 25 games as White, won 3 as Black, and drew the remaining 72.[8] In a series of twelve, 100-game matches (of unspecified time or resource constraints) against Stockfish starting from the 12 most popular human openings, AlphaZero won 290, drew 886 and lost 24.[1]

Shogi [edit]

AlphaZero was trained on shogi for a total of two hours before the tournament. In 100 shogi games against elmo (World Computer Shogi Championship 27 summer 2017 tournament version with YaneuraOu 4.73 search), AlphaZero won 90 times, lost 8 times and drew twice.[8] As in the chess games, each program got one minute per move, and elmo was given 64 threads and a hash size of 1 GB.[1]

Go [edit]

After 34 hours of self-learning of Go and against AlphaGo Zero, AlphaZero won 60 games and lost 40.[1][8]

Chess [edit]

In AlphaZero's chess match against Stockfish 8 (2016 TCEC world champion), each program was given one minute per move. Stockfish was allocated 64 threads and a hash size of 1 GB,[1] a setting that Stockfish's Tord Romstad later criticized as suboptimal.[7][note 1] AlphaZero was trained on chess for a total of nine hours before the match. During the match, AlphaZero ran on a single machine with four application-specific TPUs. In 100 games from the normal starting position, AlphaZero won 25 games as White, won 3 as Black, and drew the remaining 72.[8] In a series of twelve, 100-game matches (of unspecified time or resource constraints) against Stockfish starting from the 12 most popular human openings, AlphaZero won 290, drew 886 and lost 24.[1]

Shogi [edit]

AlphaZero was trained on shogi for a total of two hours before the tournament. In 100 shogi games against elmo (World Computer Shogi Championship 27 summer 2017 tournament version with YaneuraOu 4.73 search), AlphaZero won 90 times, lost 8 times and drew twice.[8] As in the chess games, each program got one minute per move, and elmo was given 64 threads and a hash size of 1 GB.[1]

Go [edit]

After 34 hours of self-learning of Go and against AlphaGo Zero, AlphaZero won 60 games and lost 40.[1][8]

Cup

Event	Year	Time Controls	Result	Ref
Cup 1	2018	30+10	1st	[63]
Cup 2	2019	30+5	2nd[note 1]	[64]
Cup 3	2019	30+5	2nd	[65]
Cup 4	2019	30+5	1st	[66]
Cup 5	2020	30+5	1st	[67]
Cup 6	2020	30+5	3rd	[68]
Cup 7	2020	30+5	1st	[69]
Cup 8	2021	30+5	1st	[70]
Cup 9	2021	30+5	1st	[71]
Cup 10	2022	30+3	1st	[72]
Cup 11	2023	30+3	2nd	[73]

Today

- ✓ Feedback from last lecture
- ✓ Recap
- ✓ Game Playing: AlphaBeta Search/Rule Based Systems
- ✓ MCTS
- ✓ AlphaZero and Self-Play

Summary:

1. Search is powerful: MCTS
2. Search + learning is better: AlphaZero

Attendance:

bit.ly/3RcTC9T

Feedback:

bit.ly/3RHtlxy