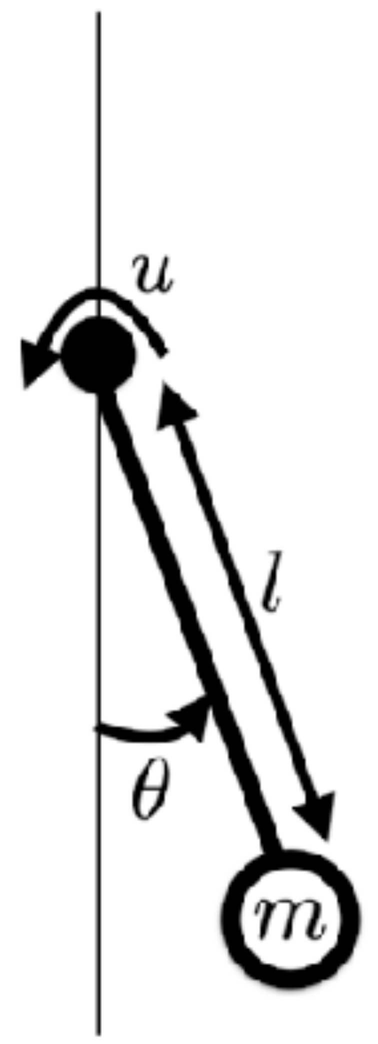
ROB-GY 6323
reinforcement learning and optimal control for robotics

Lecture XIII

Learning to play Go

Project #2 (due December 21st - no deadline extension)

Goal: implement Q-learning to control an inverted pendulum



Deliverables:

1) report in pdf format answering all the questions that do not request code. DO NOT include code in the report
2) One (or several) Jupyter notebook(s) containing all the code used to answer the questions. The notebook(s) should be runnable as is.

Paper report (due December 21st - no deadline extension)

Goal: read one scientific paper and understand it

Pick one paper from the list of papers markers with a * on brightspace

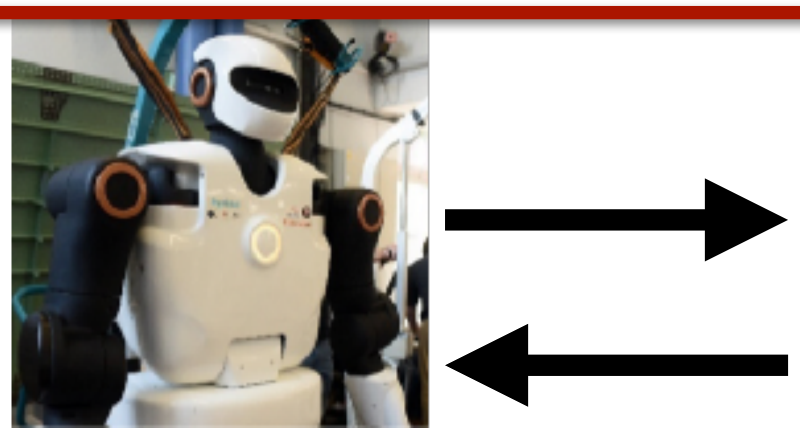
Report (<u>maximum 2 pages</u> - IEEE format double column)

It should contain 3 sections:

1. A section that summarizes the paper:
   What was done? How was it done?
   Why was it worth doing? What are the results?
2. A section explaining how the paper relates to the algorithms seen in class.
   Which algorithms? What is different?
3. A section containing a critical discussion on the paper: pros and cons.
   What seems to work and what convinces you about the result What are the issues/ limitations? What could be done better? What should be done next?

Do not copy equations or figures from the paper - keep your explanations to the point

Please fill the evaluation form!



$$\mathbf{x}_{n+1} = \mathbf{f}(\mathbf{x}_n, \mathbf{u}_n, \boldsymbol{\omega}_n)$$

given a dynamical system

$$\min_{u_k} \sum_k g_k(x_k, u_k)$$

minimize a performance cost
(measuring how well a task is performed)

Bellman's principle of optimality: $J_n(x_n) = \min_{u_n} g_n(x_n, u_n) + J_{n+1}(f(x_n, u_n))$

f(x,u) is known

Global Methods
=> optimal value and policy functions

- Dynamic Programming
- Shortest path algorithms
- Linear Quadratic Regulators
- Linear Model Predictive Control
- Value and Policy Iteration

Monte-Carlo Tree Search

Local Methods
=> (locally) optimal policies / trajectories

- LQ problems with constraints
- DDP / iLQR
- Nonlinear trajectory optimization

f(x,u) is unknown

Learn the value function (or Q-function)

- TD learning and Q-learning
- deep Q-learning

Actor-critic (Deep RL)

Learn the dynamic model

- Model-based RL

Learn the policy

- Policy gradients

Policy gradient methods

$$\nabla_\theta J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} \Psi_n \nabla_\theta \log \pi(u_n|x_n, \theta)\right]$$

$$\Psi_n = \sum_{k=0}^{N} \alpha^k g(x_k, u_k)$$

$$\Psi_n = g(x_n, u_n) + \alpha V(x_{n+1}) - V(x_n)$$

$$\Psi_n = \sum_{k=n}^{N} \alpha^k g(x_k, u_k)$$

$$\Psi_n = Q_\pi(x_n, u_n)$$

$$\Psi_n = \sum_{k=n}^{N} \alpha^k g(x_k, u_k) - b(x_n)$$

$$\Psi_n = A_n = Q(x_n, u_n) - V(x_n)$$

Policy gradient methods

$$\min_{\theta} J(\theta) = \mathbb{E}\left[\sum_{n=0}^{N} \Psi_n \log \pi(u_n|x_n, \theta)\right]$$

$$\Psi_n = \sum_{k=0}^{N} \alpha^k g(x_k, u_k)$$

$$\Psi_n = g(x_n, u_n) + \alpha V(x_{n+1}) - V(x_n)$$

$$\Psi_n = \sum_{k=n}^{N} \alpha^k g(x_k, u_k)$$

$$\Psi_n = Q_\pi(x_n, u_n)$$

$$\Psi_n = \sum_{k=n}^{N} \alpha^k g(x_k, u_k) - b(x_n)$$

$$\Psi_n = A_n = Q(x_n, u_n) - V(x_n)$$

Proximal policy optimization (PPO)

"Clip" the total scaling

$$\min_{\theta} \mathbb{E}\left[\min\left(A_n \frac{\pi(u_n|x_n,\theta)}{\pi(u_n|x_n,\theta_{old})}, clip\left(\frac{\pi(u_n|x_n,\theta)}{\pi(u_n|x_n,\theta_{old})}, 1-\epsilon, 1+\epsilon\right) A_n\right)\right]$$

Run a lot of episodes in <u>parallel</u> (in simulation) to improve the estimation of the gradient and expectation

Model-based reinforcement learning

We can learn:

- a value function
- a policy
- a model?

Model-based RL
=> learn a model + do optimal control with the model

Model-based reinforcement learning

How do we learn a model?

If the dynamics is linear

$$x_{n+1} = Ax_n + Bu_n$$

we can find A and B from data
=> regression problem

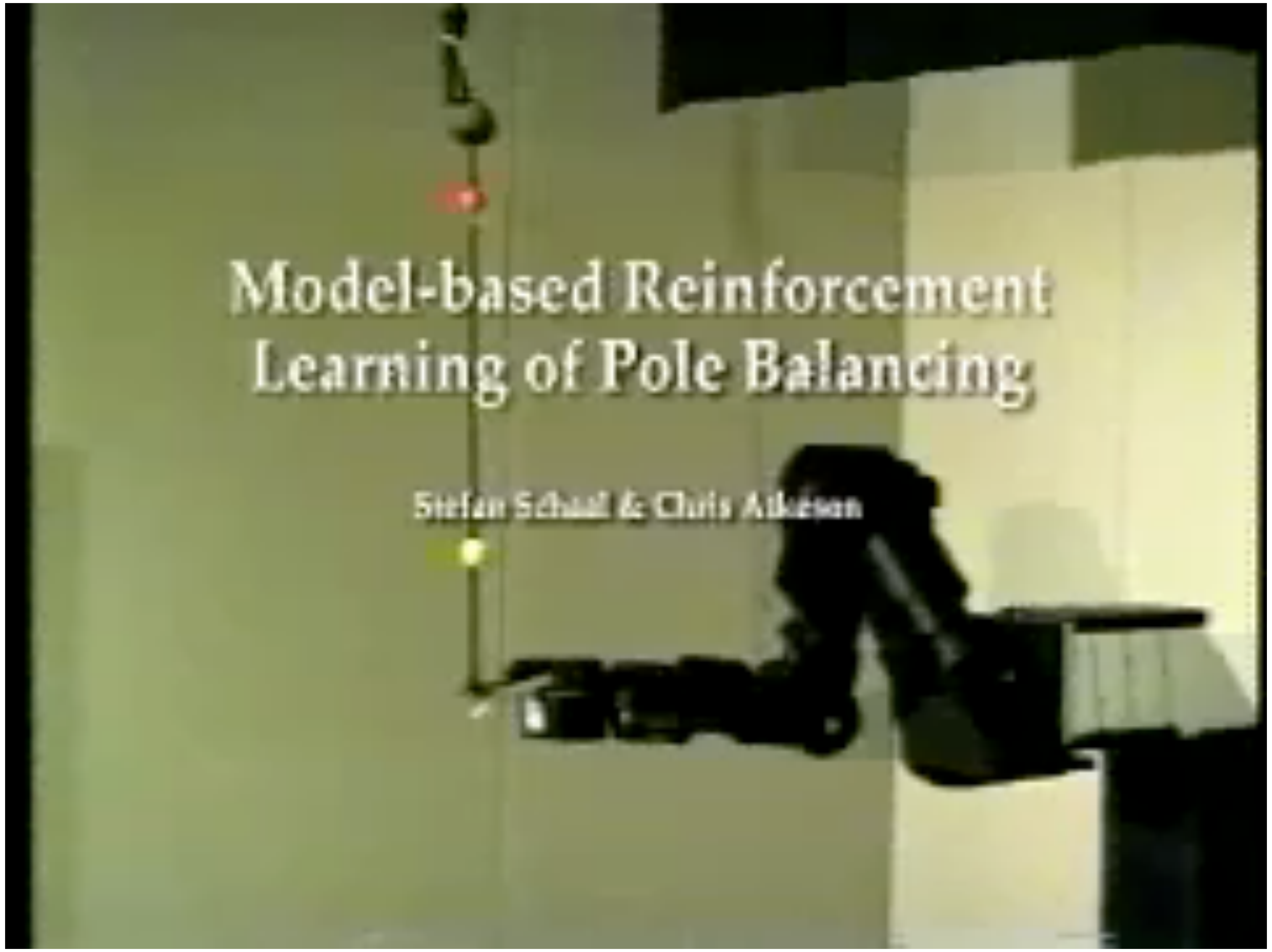
Nonlinear models

1. Use a function approximator for nonlinear functions that is "linearization" friendly
   (e.g. locally weighted regression, Schaal et al. 1997 or Gaussian Processes, Deisenroth et al. 2011)
   => good to do LQR and related, exploit linearity

2. Learn a nonlinear model
   => typically linearization is problematic - might need other techniques to solve OC problems (e.g. cross-entropy methods)

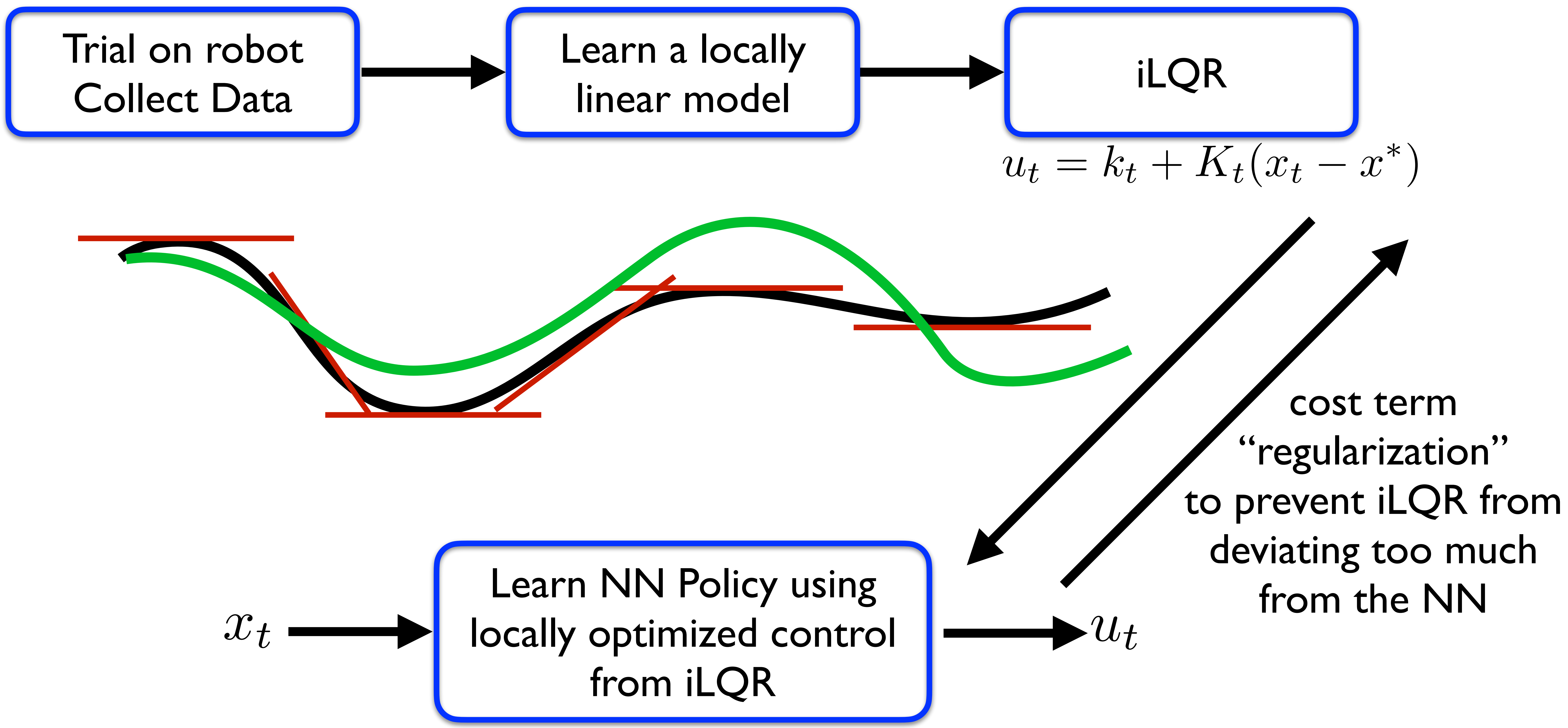
Model-based reinforcement learning

[Schaal and Atkeson ~1995]

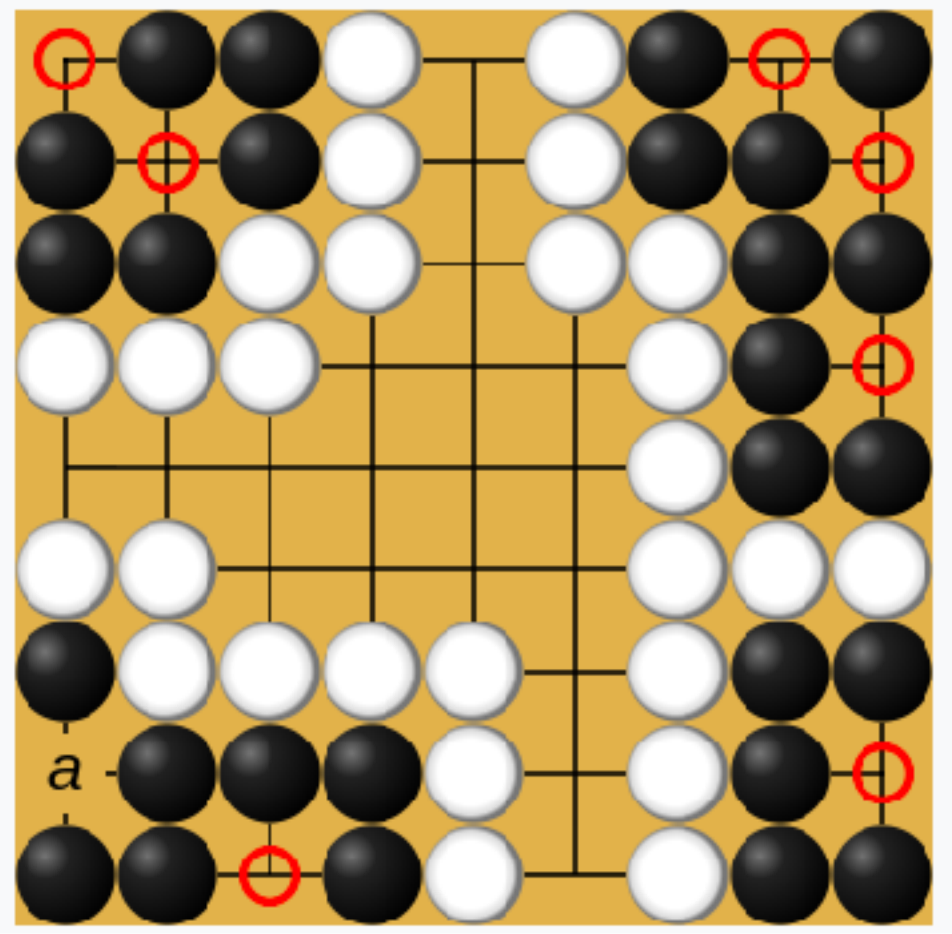


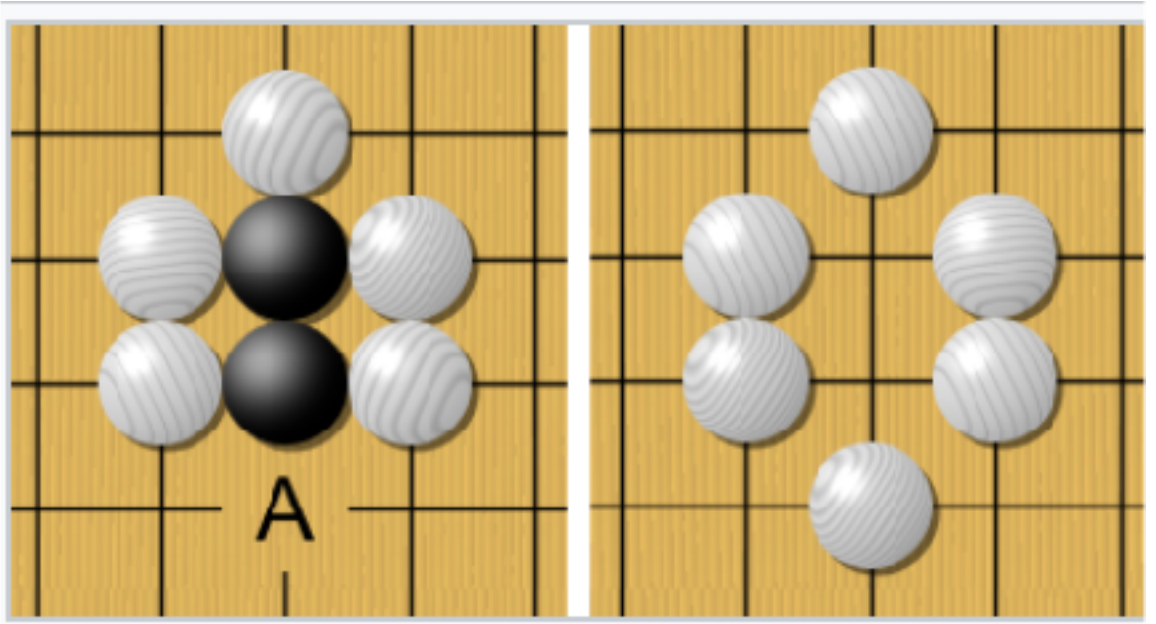
Guided Policy Search

[Levine et al. 2015]

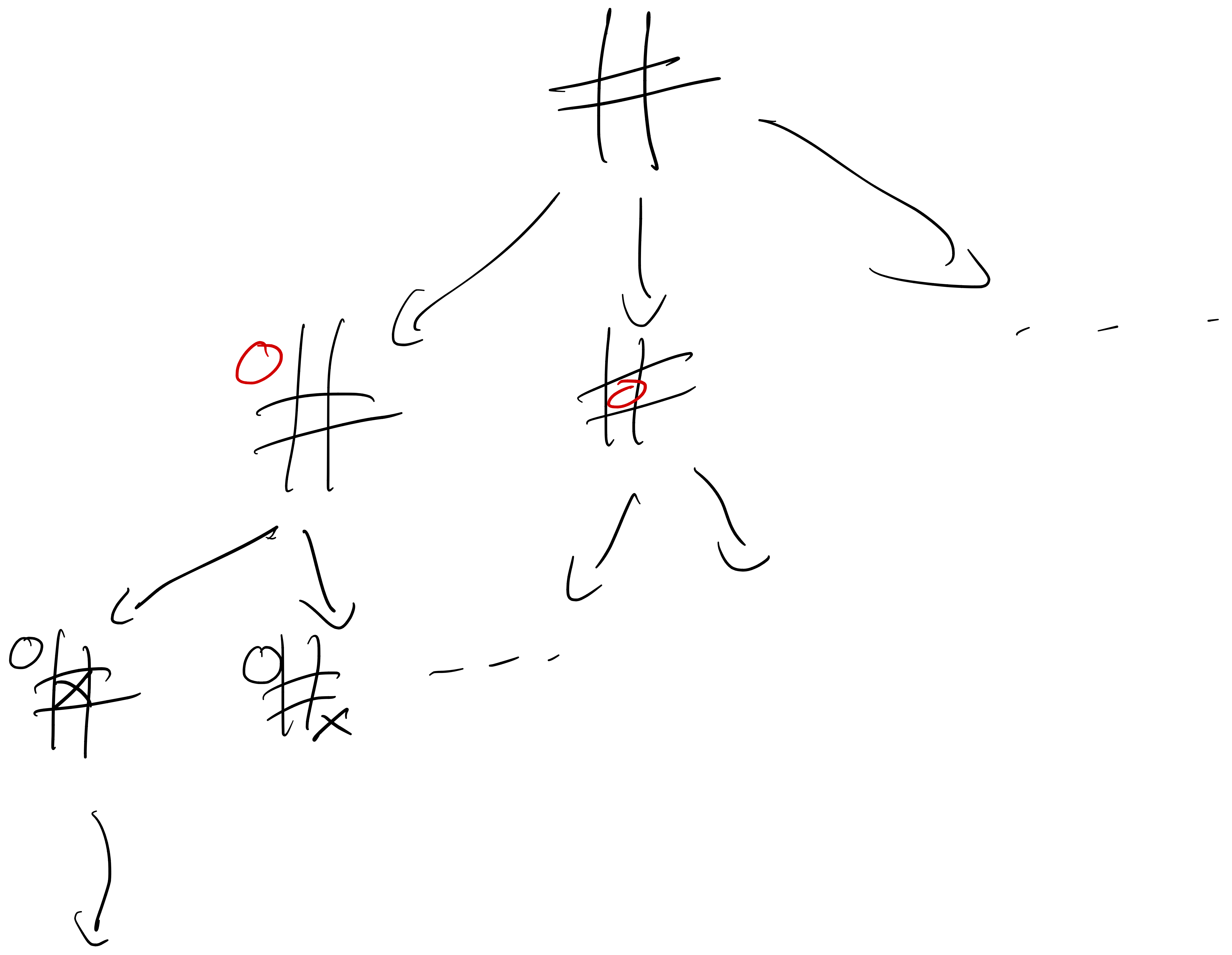


Playing the game of Go
A mixture of OC and RL


a


A

Deciding how to play with tree search



W / L / D

Go branching factor

For a game typically b^d number of moves to test
b is "breadth", i.e. number of legal moves at each turn
d is the "depth", i.e. the game length

For Go b~250 and d~150

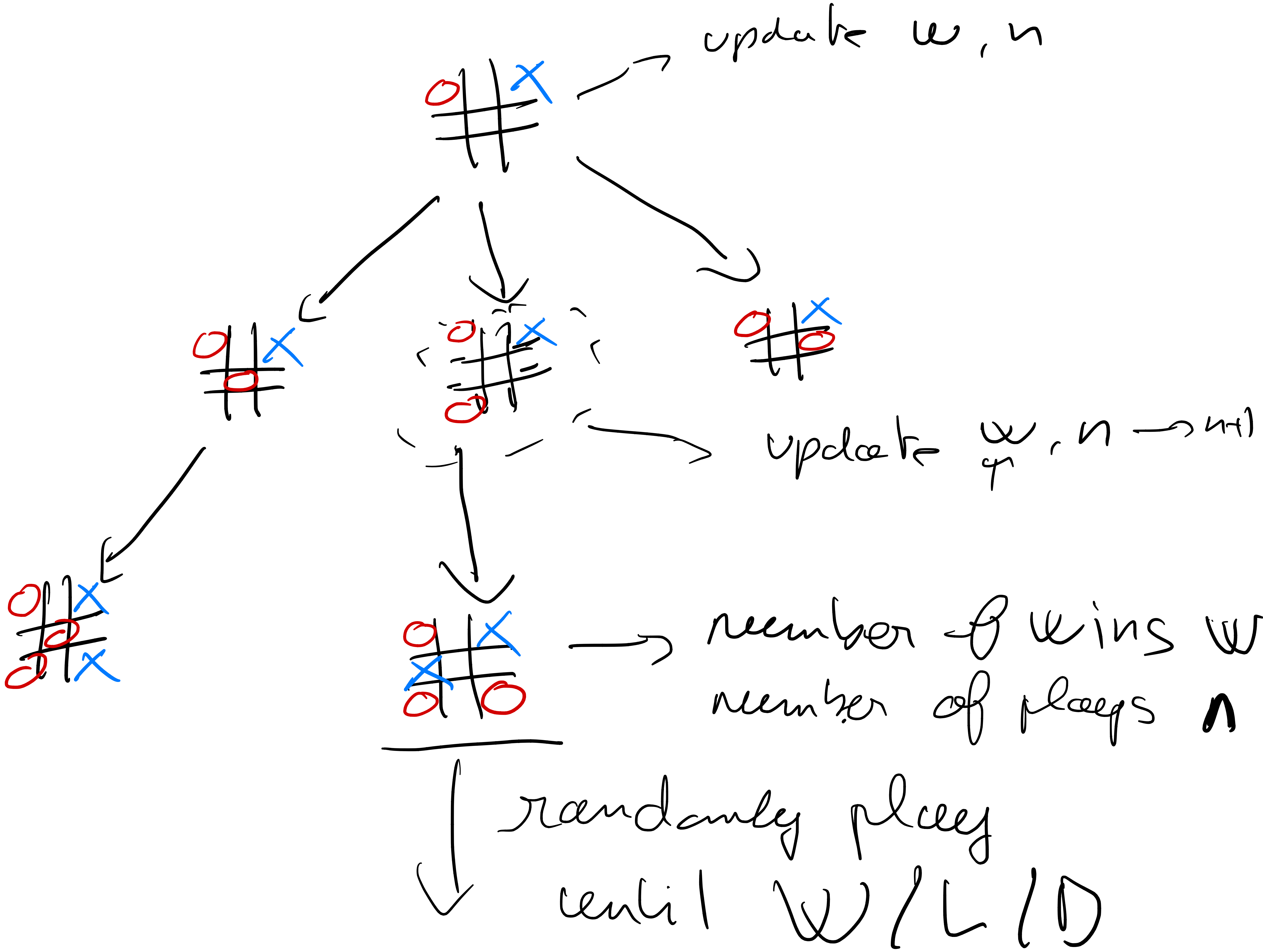
$\boxed{250^{150}}$

Speeding up search: Monte-Carlo Tree Search

Invented by R. Coulom in 2006 (not new!)

4 steps to be repeated N times

1. Selection
2. Expansion
3. Simulation
4. Backup



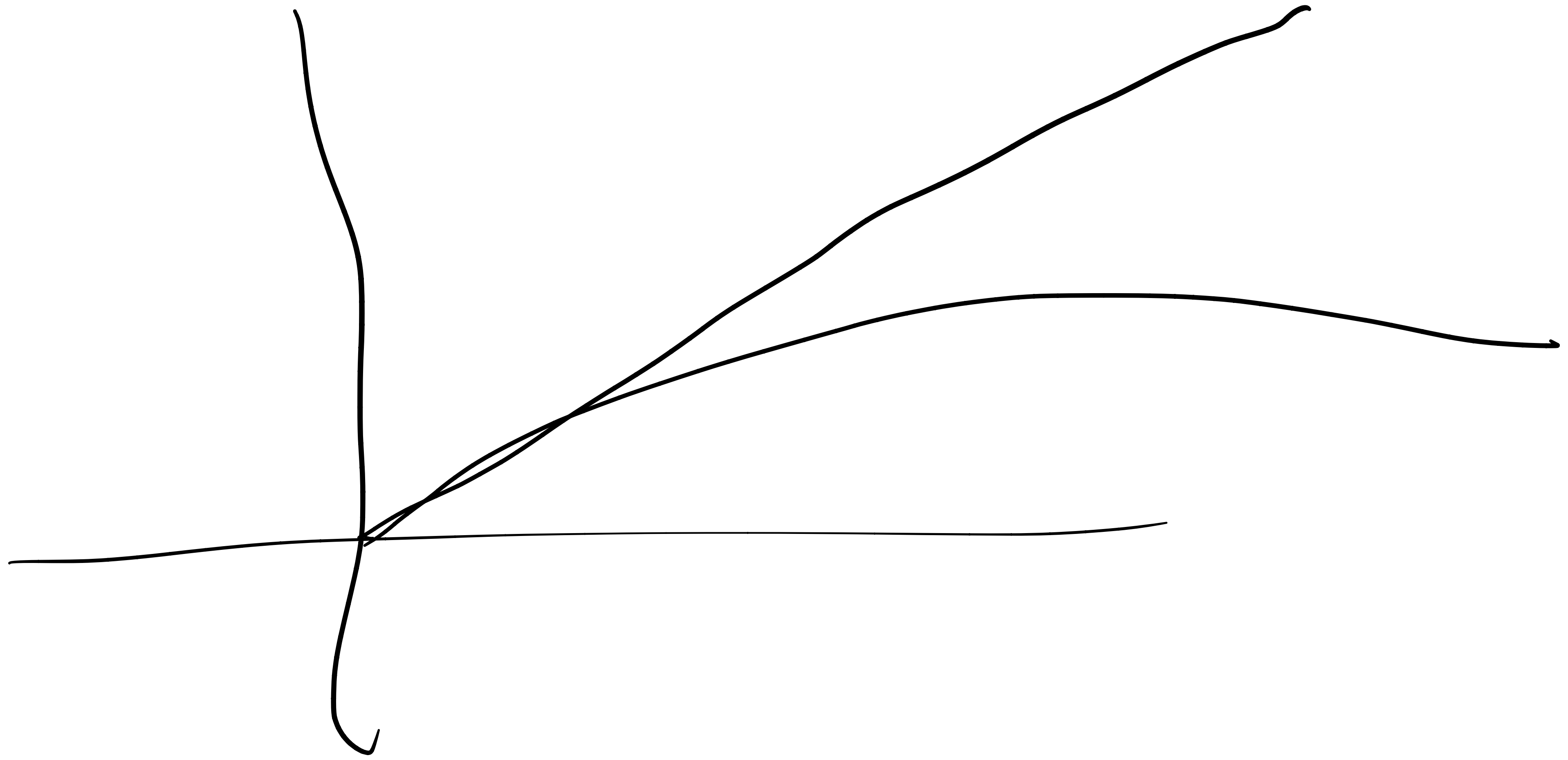
Exploration vs. Exploitation (UCT)

Upper Confidence Bound 1 for trees

UCB1

$$\underbrace{\frac{w}{n}}_{\text{exploitation}} + \underbrace{c \cdot \sqrt{\frac{\log N}{n}}}_{\text{exploration}}$$

N = # total plays



AlphaGo 2016

Reduce the "breadth" and the "depth" of the search using MCTS
In addition, improve the sampling efficiency by:

- Learning a policy
- Learning a value function

Defining the states

Feature	# of planes	Description
Stone colour	3	Player stone / opponent stone / empty
Ones	1	A constant plane filled with 1
Turns since	8	How many turns since a move was played
Liberties	8	Number of liberties (empty adjacent points)
Capture size	8	How many opponent stones would be captured
Self-atari size	8	How many of own stones would be captured
Liberties after move	8	Number of liberties after this move is played
Ladder capture	1	Whether a move at this point is a successful ladder capture
Ladder escape	1	Whether a move at this point is a successful ladder escape
Sensibleness	1	Whether a move is legal and does not fill its own eyes
Zeros	1	A constant plane filled with 0
Player color	1	Whether current player is black

Step 1: Use supervised learning using human plays to <u>learn a policy</u>

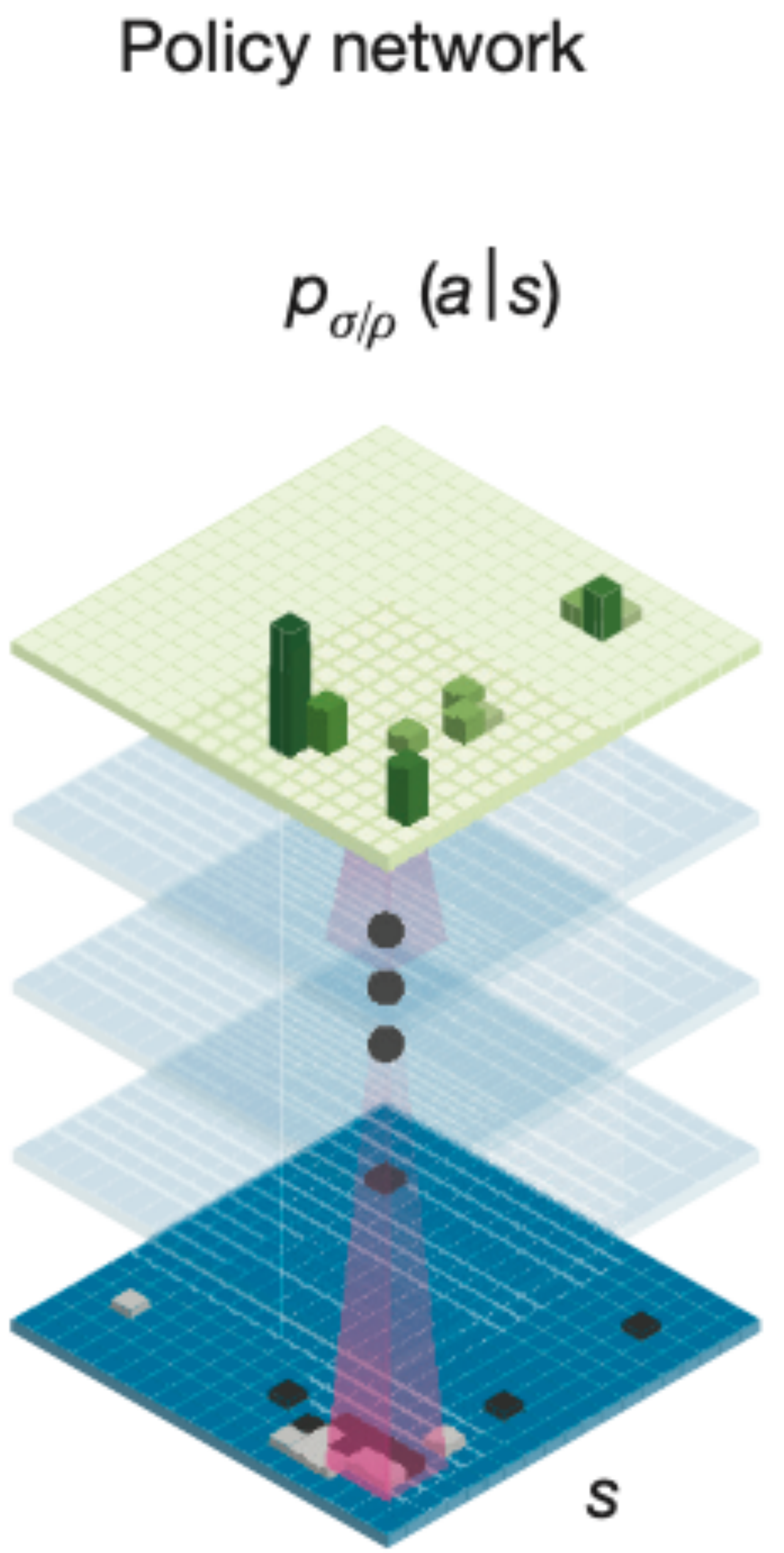
Step 2: Use policy gradients to <u>improve policy</u> (using self-play)

Step 3: Use RL to compute <u>value function</u> of policy

Step 4: <u>Monte-Carlo Tree Search</u> using previously learned policy and value function to direct exploration

Stage 1: learn a policy from Human players

Imitation learning



$p_\sigma(a|s)$

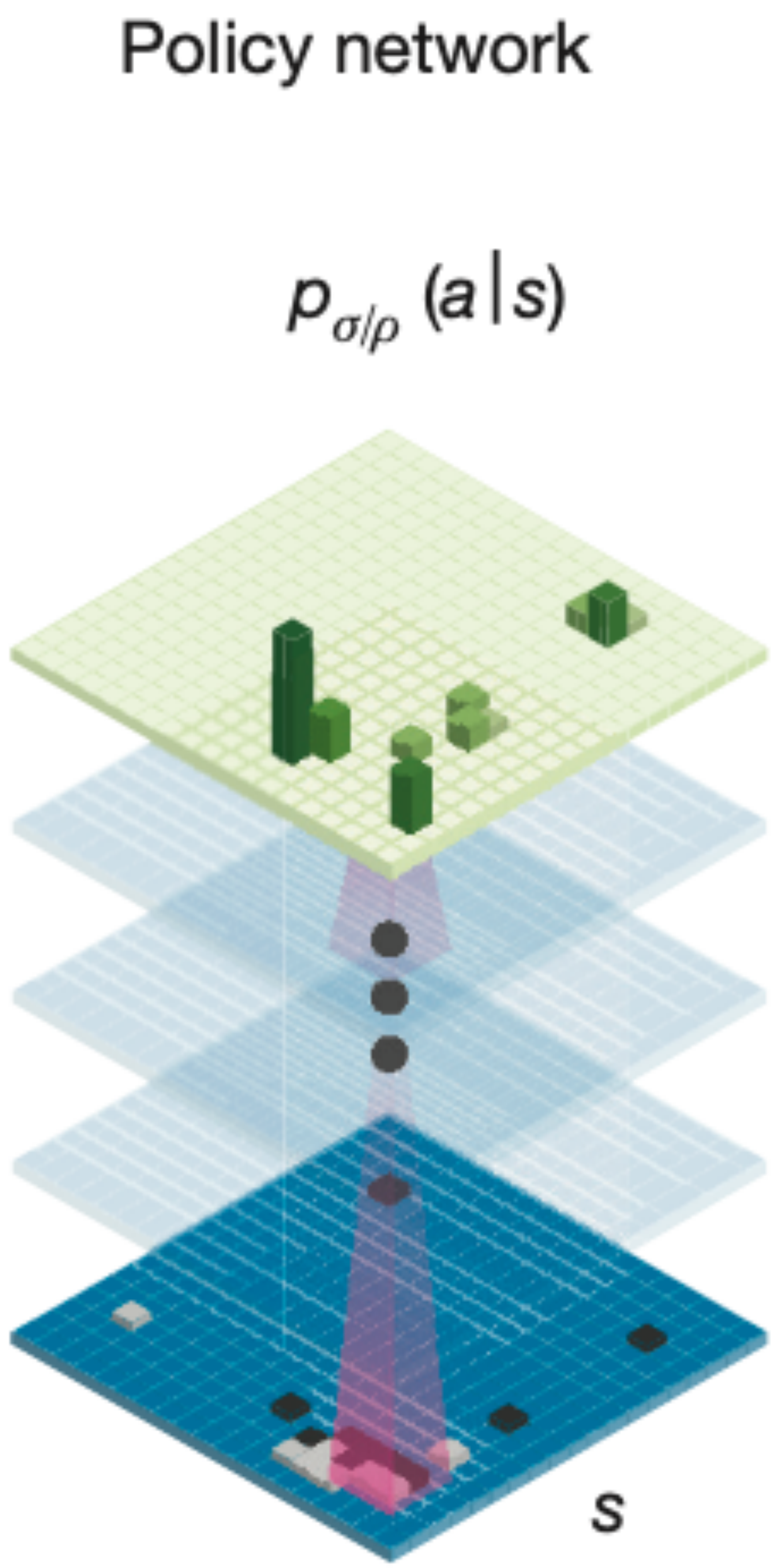
Policy learned with supervised learning SL-policy

13 layers neural network - accurate (57% / 55%) but slow to evaluate (3ms)

$p_\pi(a|s)$

Policy with smaller network - less accurate (24%) but fast to evaluate (2us)

Stage 2: improve policy using RL policy gradient



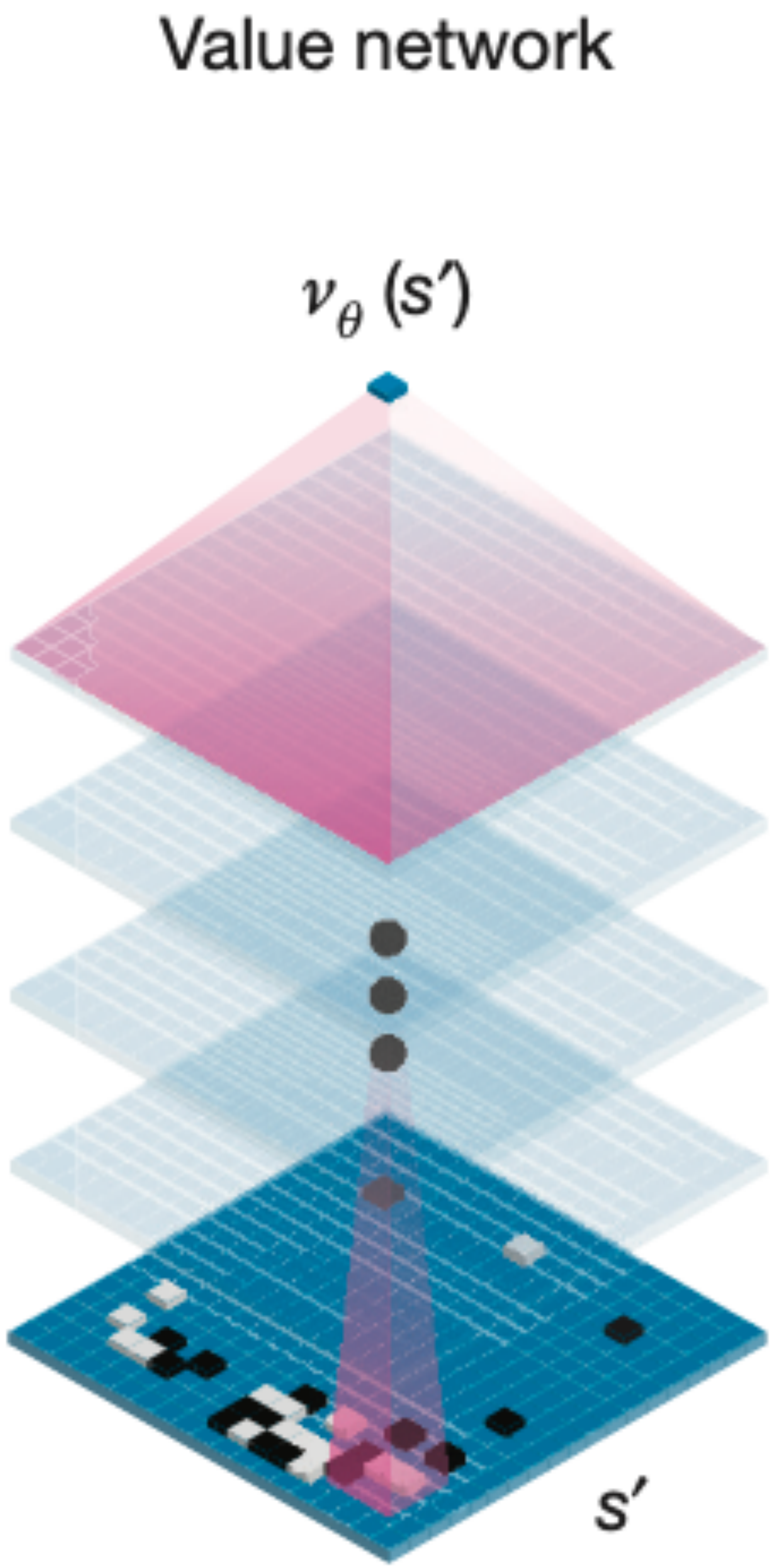
p_ρ Policy learned using RL

$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t$$

$$z_t = \pm r(s_T)$$

RL policy won 80% games agains SL policy

Stage 3: learning a value function



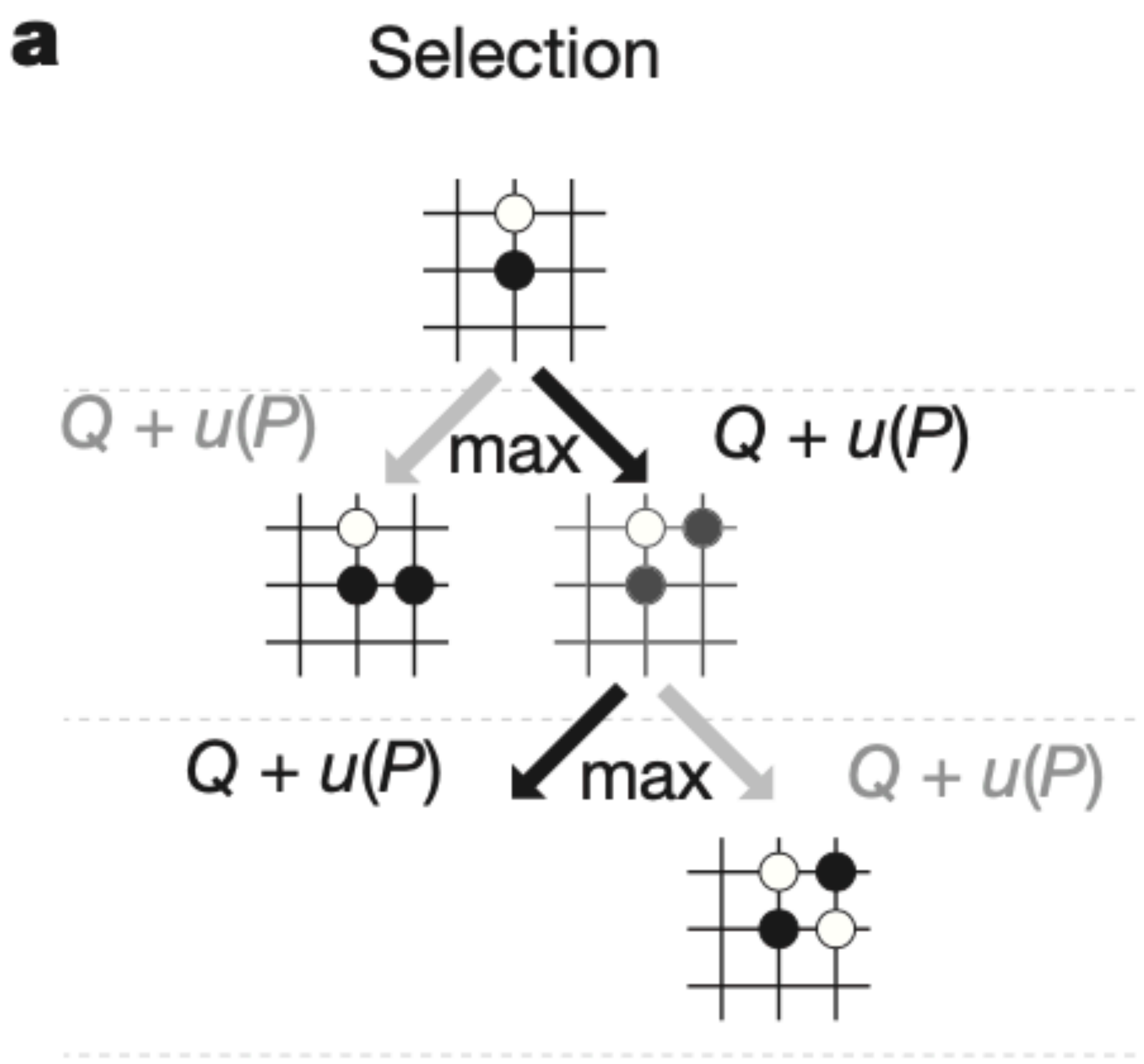
Self-play and randomization

$$v^p(s) = \mathbb{E}[z_t | s_t = s,\ a_{t\ldots T} \sim p]$$

$$\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}(z - v_\theta(s))$$

Stage 4: (modified) Monte-Carlo Tree Search

Each edge of the tree (action/state pair) stores an action value Q(s,a), visit count N(s,a) and prior probability P(s,a)

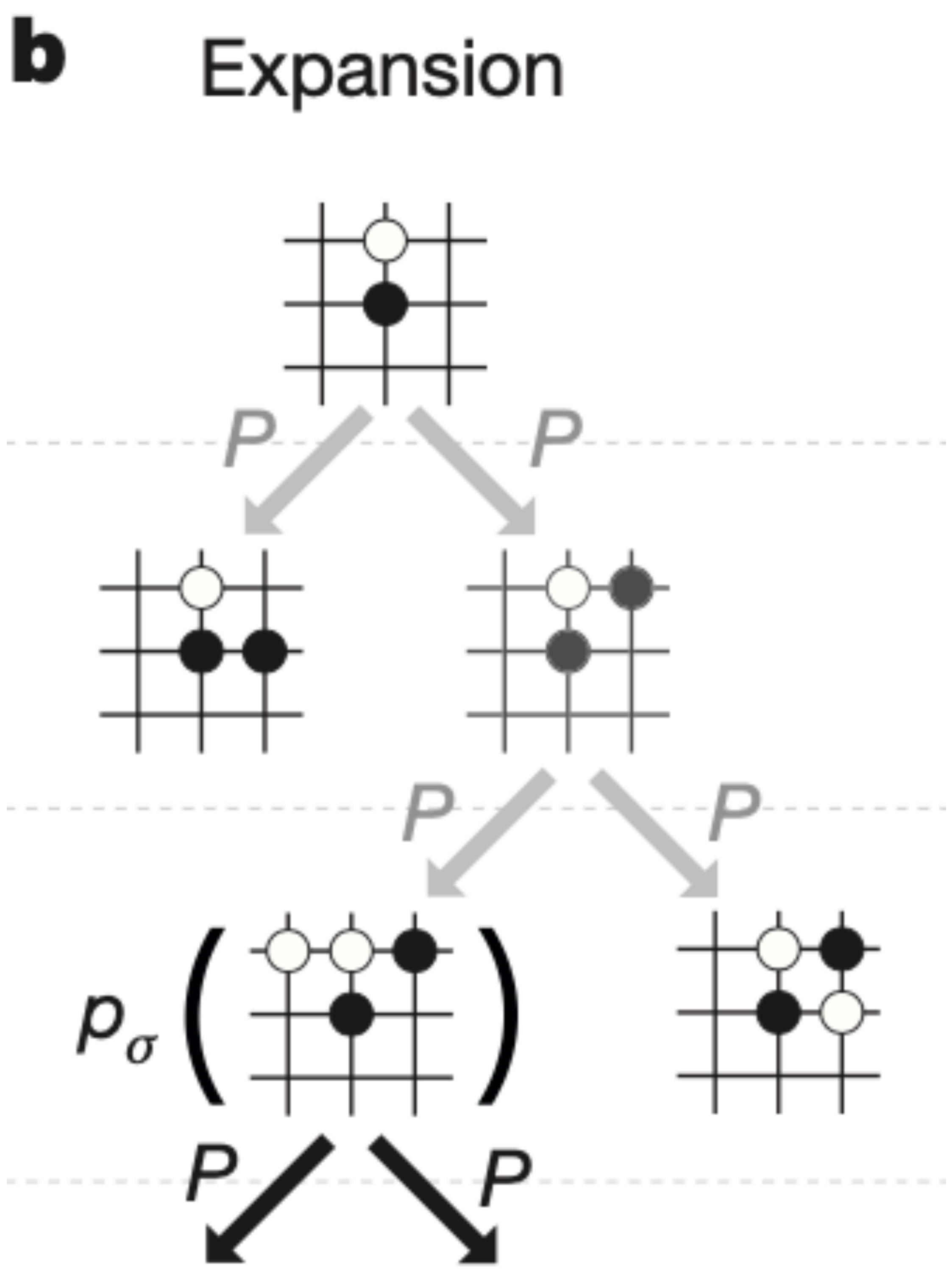


Go down the (partial) tree using

$$a_t = \operatorname{argmax}(Q(s_t, a) + u(s_t, a))$$

$$u(s,a) \propto \frac{P(s,a)}{1 + N(s,a)}$$

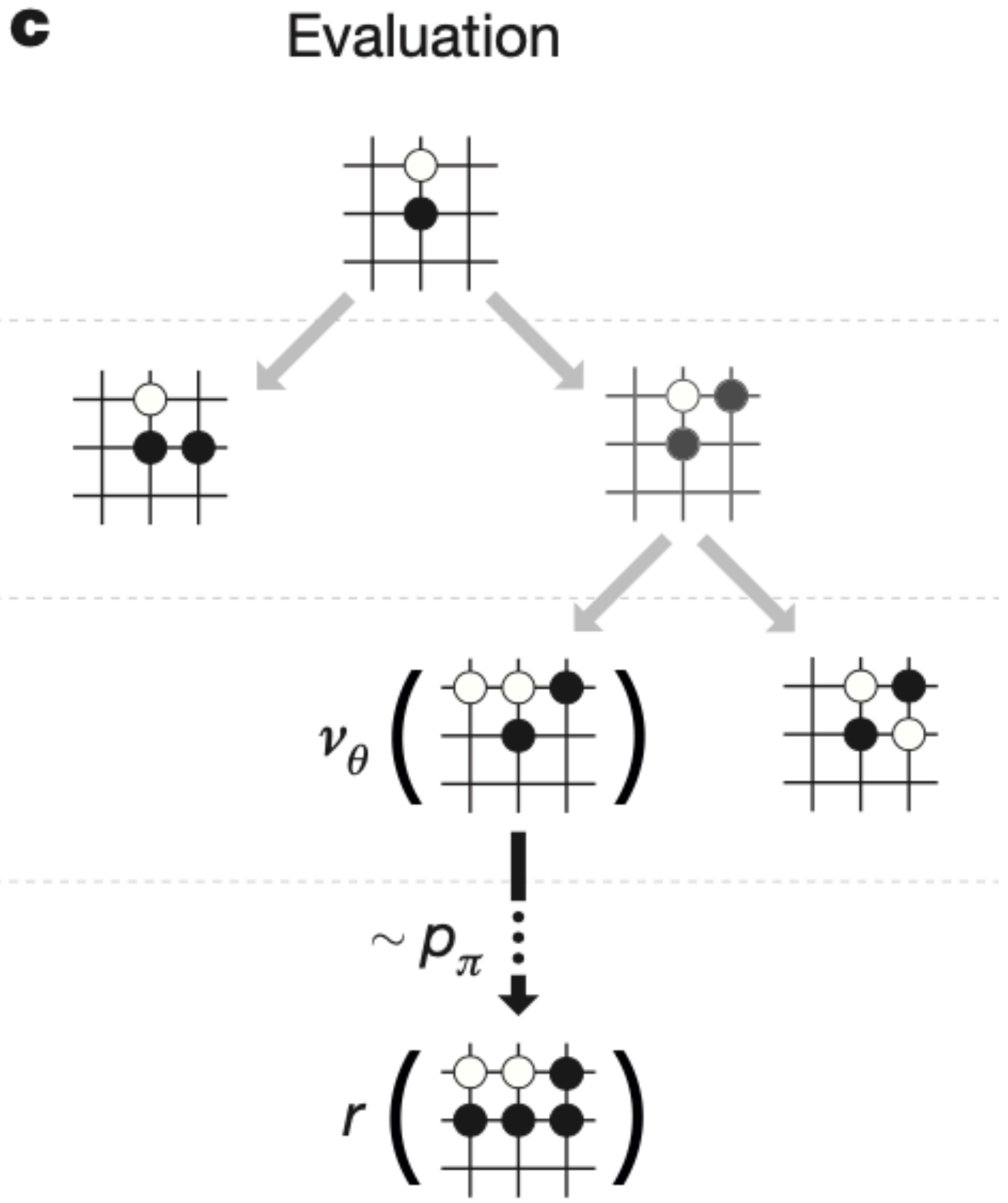
Stage 4: (modified) Monte-Carlo Tree Search



When a leaf is reached it can be expanded using the SL policy network

$$P(s,a) = p_\sigma(a|s)$$

Stage 4: (modified) Monte-Carlo Tree Search

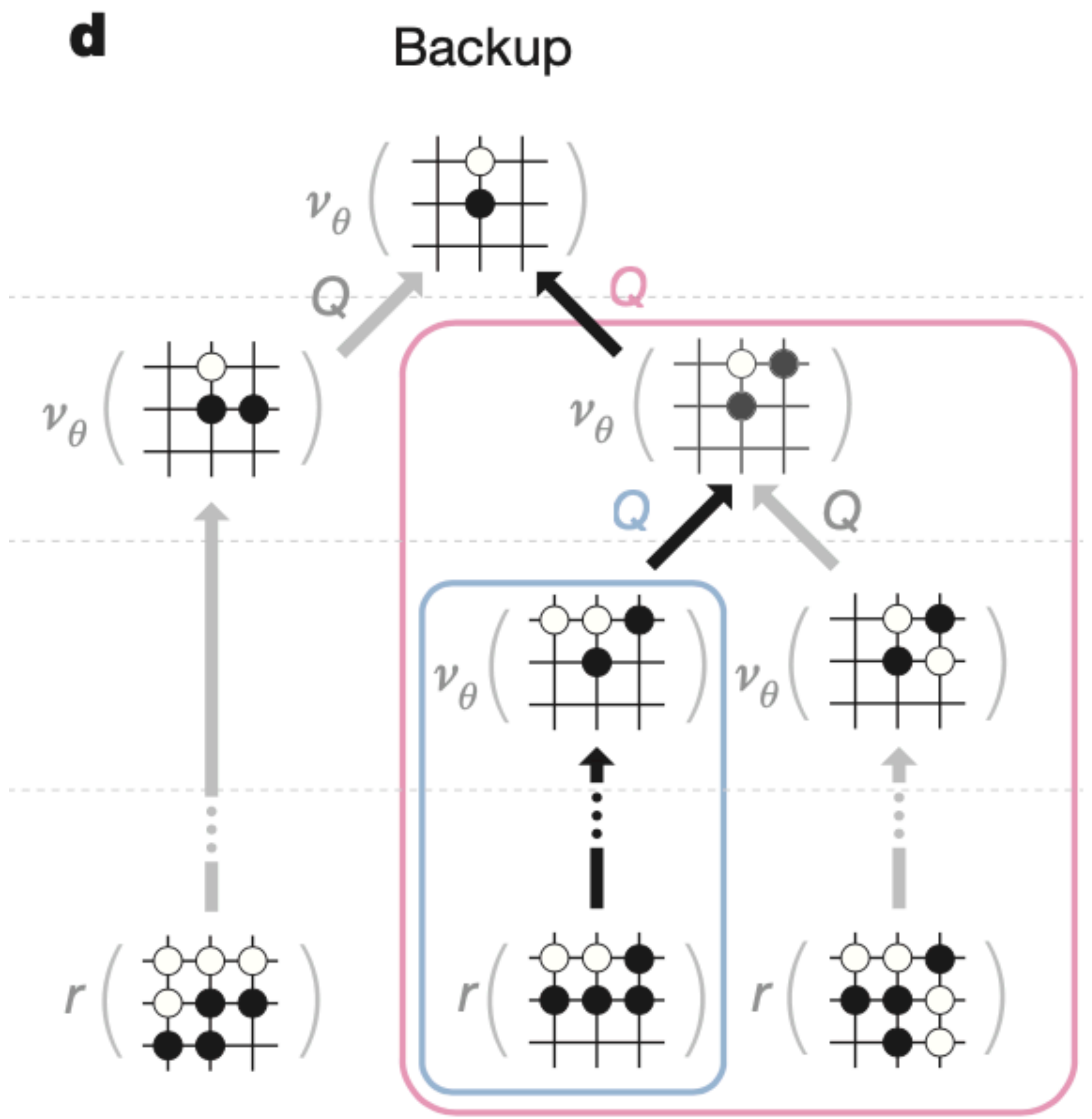


Evaluate the leaf node V(s) using:

1. The learned value function $v^p(s)$
2. The outcome of a random simulated "play"

$$V(s_L) = (1-\lambda)v_\theta(s_L) + \lambda z_L$$

Stage 4: (modified) Monte-Carlo Tree Search



$$N(s,a)=\sum_{i=1}^{n} 1(s,a,i)$$

$$Q(s,a)=\frac{1}{N(s,a)}\sum_{i=1}^{n} 1(s,a,i)V(s_L^i)$$

AlphaGoZero (2017)

Use self-play to learn a policy and value function (no SL)

Input features are only black and white stones (no other features)

Single NN for both policy and value

Simpler tree search - no evaluation through simulated play

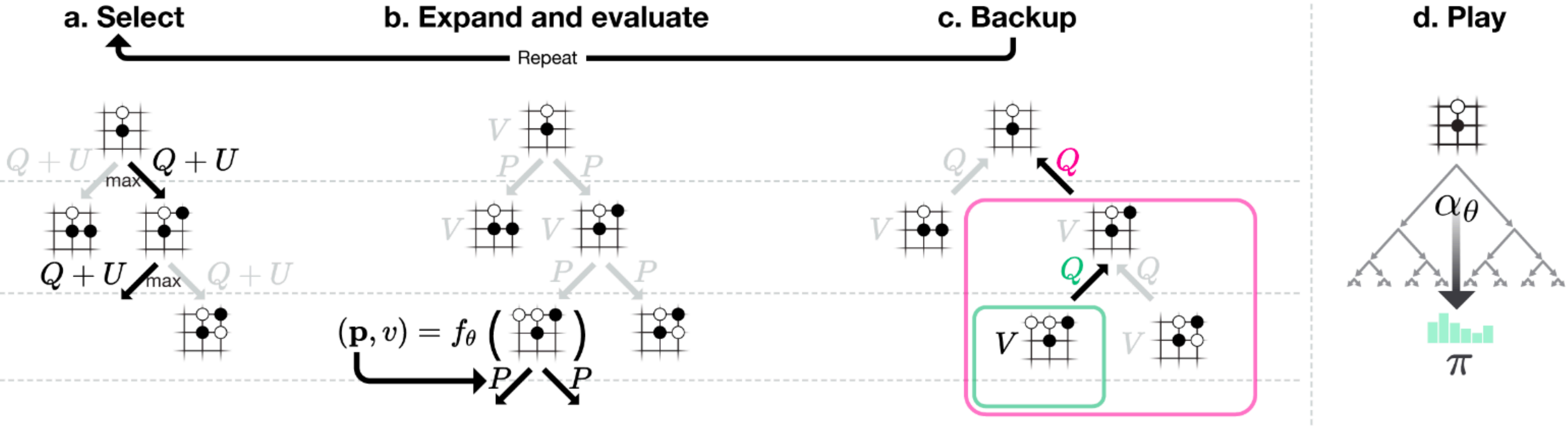
AlphaGoZero (2017)

Use self-play to learn a policy and value function (no SL)

Input features are only black and white stones (no other features)

Single NN for both policy and value

Simpler tree search - no evaluation through simulated play



AlphaZero (2017)

Similar to AlphaGoZero but to play also Chess and Shogi

MuZero (2019)

Similar to AlphaGoZero but also learns the game model