Carnegie Mellon
School of Computer Science

Deep Reinforcement Learning and Control

Natural Policy Gradients (cont.)

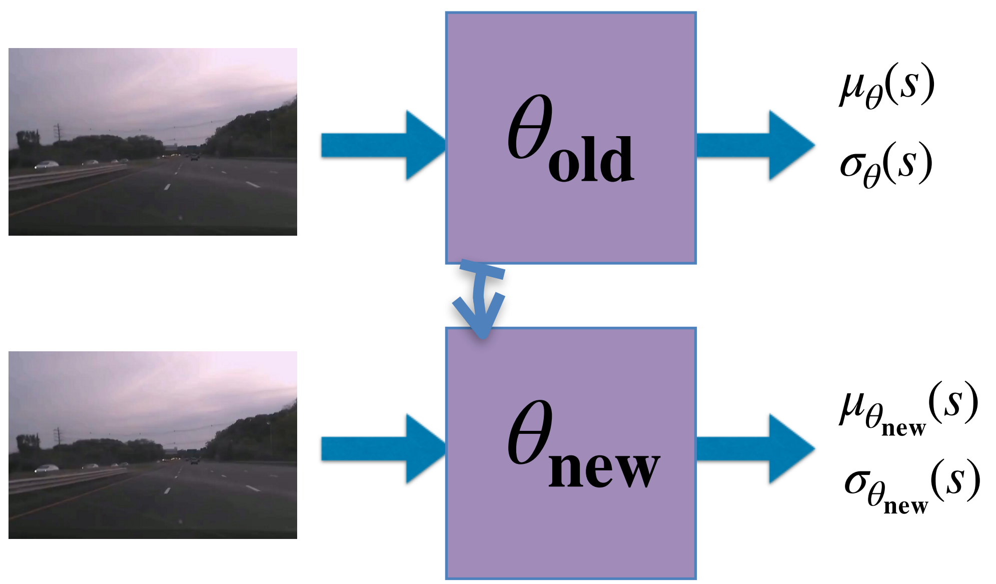
Katerina Fragkiadaki

Revision

Policy Gradients

1. Collect trajectories for policy π_θ
2. Estimate advantages A
3. Compute policy gradient $\hat{g}$
4. Update policy parameters $\theta_{new} = \theta + \epsilon \cdot \hat{g}$
5. GOTO 1

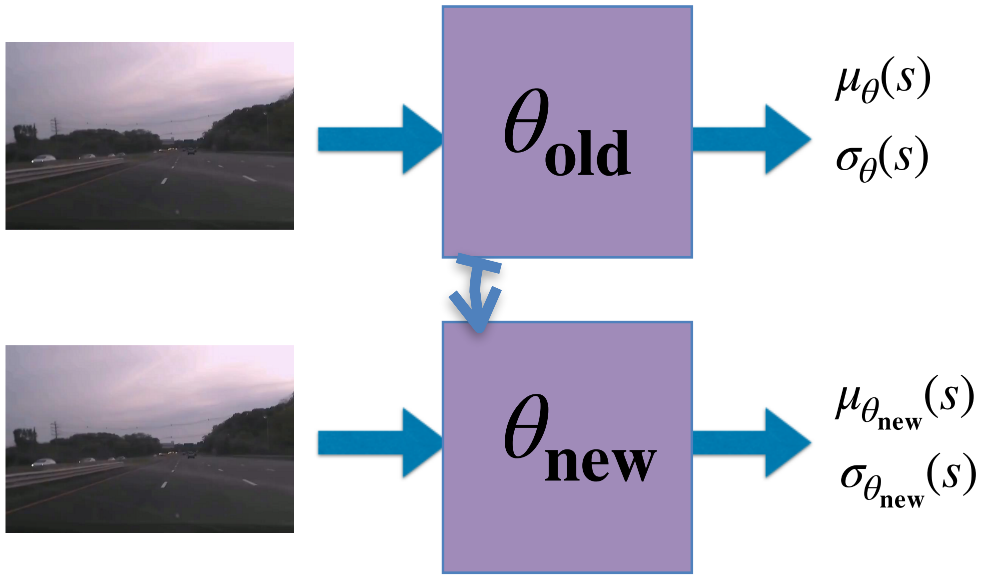
How to estimate this gradient



Policy Gradients

1. Collect trajectories for policy π_θ
2. Estimate advantages A
3. Compute policy gradient $\hat{g}$
4. Update policy parameters $\theta_{new} = \theta + \epsilon \cdot \hat{g}$
5. GOTO 1

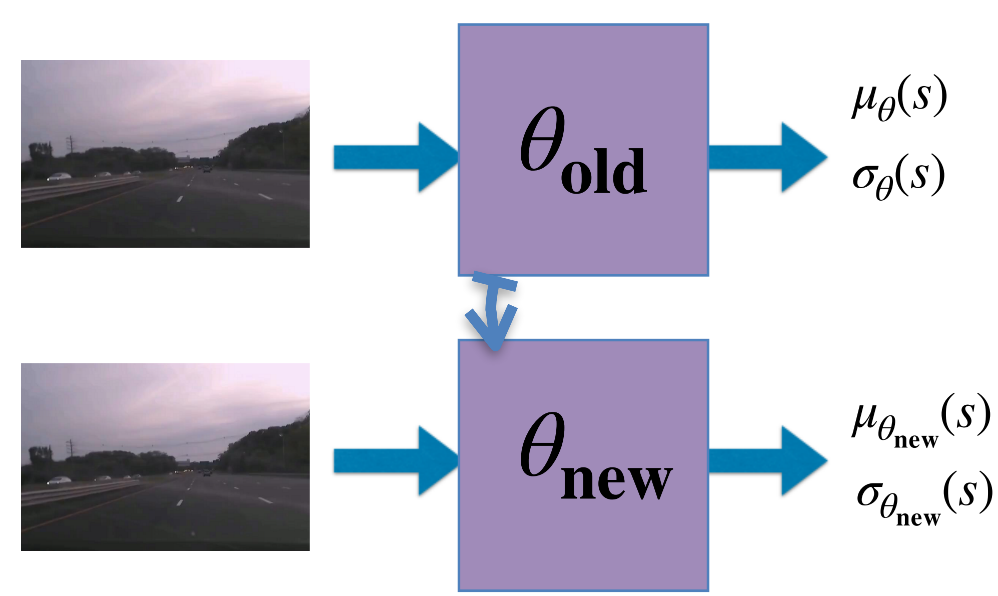
How to estimate the stepsize



Policy Gradients

1. Collect trajectories for policy π_θ
2. Estimate advantages A
3. Compute policy gradient $\hat{g}$
4. Update policy parameters $\theta_{new} = \theta + \epsilon \cdot \hat{g}$
5. GOTO 1

- Step too big
  Bad policy->data collected under bad policy-> we cannot recover
  (in Supervised Learning, data does not depend on neural network weights)
- Step too small
  Not efficient use of experience
  (in Supervised Learning, data can be trivially re-used)



What is the underlying optimization problem?

We started here: $\max_{\theta} . \quad U(\theta) = \mathbb{E}_{\tau \sim P(\tau;\theta)}[R(\tau)] = \sum_{\tau} P(\tau;\theta)R(\tau)$

Policy gradients: $\hat{g} \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\alpha_t^{(i)} \mid s_t^{(i)})A(s_t^{(i)}, a_t^{(i)}), \quad \tau_i \sim \pi_\theta$

$$\hat{g} = \mathbb{E}_t\left[\nabla_\theta \log \pi_\theta(\alpha_t \mid s_t)A(s_t, a_t)\right]$$

This result from differentiating the following objective function:

$$U^{PG}(\theta) = \mathbb{E}_t\left[\log \pi_\theta(\alpha_t \mid s_t)A(s_t, a_t)\right]$$

$\max_{\theta} . \quad U^{PG}(\theta)$

This is not the right objective: we can't optimize too far (as the advantage values become invalid), and this constraint shows up nowhere in the optimization:

Compare this to supervised learning using expert actions $\tilde{a} \sim \pi^*$ and a maximum likelihood objective:

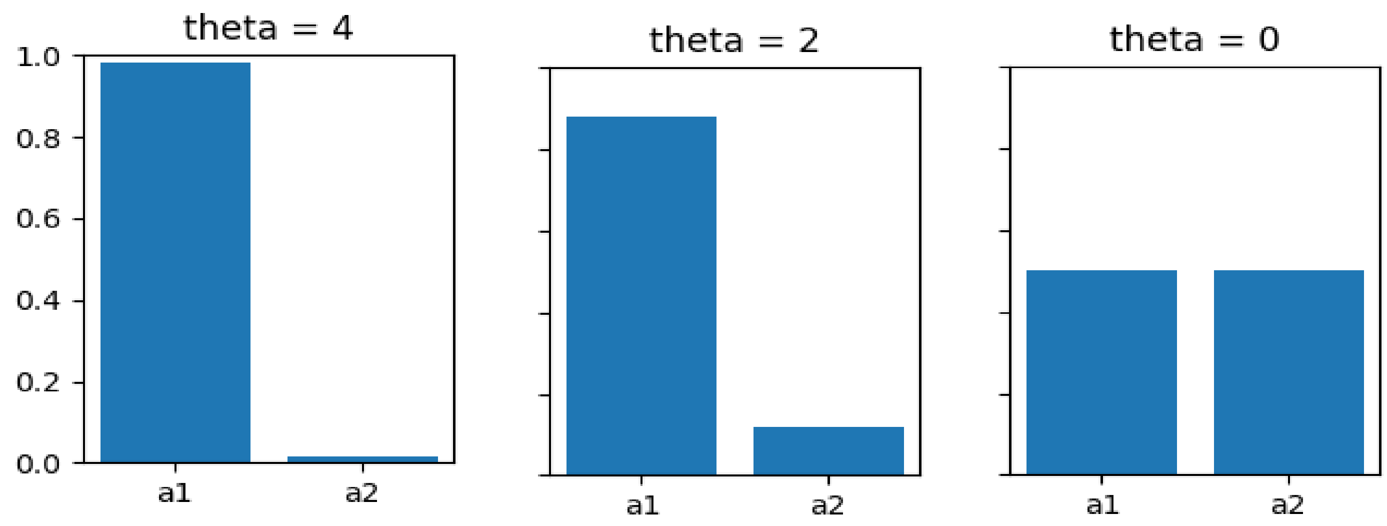
$U^{SL}(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \log \pi_\theta(\tilde{\alpha}_t^{(i)} \mid s_t^{(i)}), \quad \tau_i \sim \pi^*$ (+regularization)

Hard to choose stepsizes

1. Collect trajectories for policy π_θ
2. Estimate advantages A
3. Compute policy gradient $\hat{g}$
4. Update policy parameters $\theta_{new} = \theta + \epsilon \cdot \hat{g}$
5. GOTO 1

Consider a family of policies with parametrization:

$$\pi_\theta(a) = \begin{cases} \sigma(\theta) & a = 1 \\ 1 - \sigma(\theta) & a = 2 \end{cases}$$



The same parameter step $\Delta\theta = -2$ changes the policy distribution more or less dramatically depending on where in the parameter space we are.

Notation

We will use the following to denote values of parameters and corresponding policies before and after an update:

$$\theta_{old} \rightarrow \theta_{new}$$

$$\pi_{old} \rightarrow \pi_{new}$$

$$\theta \rightarrow \theta'$$

$$\pi \rightarrow \pi'$$

Gradient Descent in Distribution Space

The stepwise in gradient descent results from solving the following optimization problem, e.g., using line search:

$$d* = \arg\max_{||d|| \leq \epsilon} U(\theta + d)$$

SGD: $\theta_{new} = \theta_{old} + d*$

Euclidean distance in parameter space

It is hard to predict the result on the parameterized distribution.. hard to pick the threshold epsilon

Natural gradient descent: the stepwise in parameter space is determined by considering the KL divergence in the distributions before and after the update:

$$d* = \arg\max_{d,\ s.t.\ \mathrm{KL}(\pi_\theta || \pi_{\theta+d}) \leq \epsilon} U(\theta + d)$$

KL divergence in distribution space

Easier to pick the distance threshold (and we made the constraint explicit of ``don't optimize too much")

$$D_{\mathrm{KL}}(P\|Q) = \sum_i P(i) \log\left(\frac{P(i)}{Q(i)}\right)$$

$$D_{\mathrm{KL}}(P\|Q) = \int_{-\infty}^{\infty} p(x) \log\left(\frac{p(x)}{q(x)}\right) dx$$

Solving the KL Constrained Problem

$$U(\theta) = \mathbb{E}_t \left[\log \pi_\theta(\alpha_t \mid s_t) A(s_t, a_t)\right]$$

Unconstrained penalized objective:

$$d^* = \arg\max_d U(\theta + d) - \lambda(\mathrm{D}_{\mathrm{KL}}\left[\pi_\theta \| \pi_{\theta+d}\right] - \epsilon)$$

Let's solve it: first order Taylor expansion for the loss and second order for the KL:

$$d^* \approx \arg\max_d U(\theta_{old}) + \nabla_\theta U(\theta)\,|_{\theta=\theta_{old}} \cdot d - \frac{1}{2}\lambda(d^\top \nabla_\theta^2 \mathrm{D}_{\mathrm{KL}}\left[\pi_{\theta_{old}} \| \pi_\theta\right] |_{\theta=\theta_{old}} d) + \lambda\epsilon$$

Q: How will you compute this?

KL Taylor expansion

$$\mathrm{D_{KL}}(p_{\theta_{old}}|p_\theta) \approx \cancel{\mathrm{D_{KL}}(p_{\theta_{old}}|p_{\theta_{old}})} + d^\top \nabla_\theta \cancel{\mathrm{D_{KL}}(p_{\theta_{old}}|p_\theta)|_{\theta=\theta_{old}}} + \frac{1}{2} d^\top \nabla_\theta^2 \mathrm{D_{KL}}(p_{\theta_{old}}|p_\theta)|_{\theta=\theta_{old}} d$$

KL Taylor expansion

$$
\begin{aligned}
\mathrm{D}_{\mathrm{KL}}(p_{\theta_{old}} | p_\theta) &\approx \frac{1}{2} d^\top \nabla^2_\theta \mathrm{D}_{\mathrm{KL}}(p_{\theta_{old}} | p_\theta) |_{\theta=\theta_{old}} d \\
&= \frac{1}{2} d^\top \mathbf{F}(\theta_{old}) d \\
&= \frac{1}{2} (\theta - \theta_{old})^\top \mathbf{F}(\theta_{old}) (\theta - \theta_{old})
\end{aligned}
$$

Fisher Information matrix:

$$
\mathbf{F}(\theta) = \mathbb{E}_\theta \left[\nabla_\theta \log p_\theta(x) \nabla_\theta \log p_\theta(x)^\top \right]
$$

$$
\mathbf{F}(\theta_{old}) = \nabla^2_\theta \mathrm{D}_{\mathrm{KL}}(p_{\theta_{old}} | p_\theta) |_{\theta=\theta_{old}}
$$

Since KL divergence is roughly analogous to a distance measure between distributions, Fisher information serves as a **local distance metric between distributions**: how much you change the distribution if you move the parameters a little bit in a given direction.

Solving the KL Constrained Problem

Unconstrained penalized objective:

$$d* = \arg\max_{d} U(\theta + d) - \lambda(\mathrm{D}_{\mathrm{KL}}\left[\pi_\theta \| \pi_{\theta+d}\right] - \epsilon)$$

First order Taylor expansion for the loss and second order for the KL:

$$\approx \arg\max_{d} U(\theta_{old}) + \nabla_\theta U(\theta)\,|_{\theta=\theta_{old}} \cdot d - \frac{1}{2}\lambda(d^\top \nabla^2_\theta \mathrm{D}_{\mathrm{KL}}\left[\pi_{\theta_{old}} \| \pi_\theta\right]|_{\theta=\theta_{old}} d) + \lambda\epsilon$$

Substitute for the information matrix:

$$= \arg\max_{d} \nabla_\theta U(\theta)\,|_{\theta=\theta_{old}} \cdot d - \frac{1}{2}\lambda(d^\top \mathbf{F}(\theta_{old}) d)$$

$$= \arg\min_{d} - \nabla_\theta U(\theta)\,|_{\theta=\theta_{old}} \cdot d + \frac{1}{2}\lambda(d^\top \mathbf{F}(\theta_{old}) d)$$

Natural Gradient Descent

Setting the gradient to zero:

$$0 = \frac{\partial}{\partial d}\left(-\nabla_\theta U(\theta)\,|_{\theta=\theta_{old}} \cdot d + \frac{1}{2}\lambda(d^\top \mathbf{F}(\theta_{old})d)\right)$$

$$= -\nabla_\theta U(\theta)\,|_{\theta=\theta_{old}} + \frac{1}{2}\lambda(\mathbf{F}(\theta_{old}))d$$

$$d = \frac{2}{\lambda}\mathbf{F}^{-1}(\theta_{old})\nabla_\theta U(\theta)\,|_{\theta=\theta_{old}}$$

The natural gradient: $g_N = \mathbf{F}^{-1}(\theta_{old})\nabla_\theta U(\theta)$

$$\theta_{new} = \theta_{old} + \alpha \cdot g_N$$

Let's solve for the stepzise along the natural gradient direction:

$$\mathrm{D}_{\mathrm{KL}}(\pi_{\theta_{old}}|\pi_\theta) \approx \frac{1}{2}(\theta - \theta_{old})^\top \mathbf{F}(\theta_{old})(\theta - \theta_{old})$$

$$\frac{1}{2}(\alpha g_N)^\top \mathbf{F}(\alpha g_N) = \epsilon$$

$$\alpha = \sqrt{\frac{2\epsilon}{\ldots}}$$

Stepsize along the Natural Gradient direction

The natural gradient: $g_N = \mathbf{F}^{-1}(\theta_{old}) \nabla_\theta U(\theta)$

$$\theta_{new} = \theta_{old} + \alpha \cdot g_N$$

Let's solve for the stepzise along the natural gradient direction!

$$\mathrm{D}_{\mathrm{KL}}(\pi_{\theta_{old}} | \pi_\theta) \approx \frac{1}{2}(\theta - \theta_{old})^\top \mathbf{F}(\theta_{old})(\theta - \theta_{old}) = \frac{1}{2}(\alpha g_N)^\top \mathbf{F}(\alpha g_N)$$

I want the KL between old and new policies to be \epsilon:

$$\frac{1}{2}(\alpha g_N)^\top \mathbf{F}(\alpha g_N) = \epsilon$$

$$\alpha = \sqrt{\frac{2\epsilon}{(g_N^\top \mathbf{F} g_N)}}$$

Natural Gradient Descent

Algorithm 1 Natural Policy Gradient

Input: initial policy parameters θ_0
for $k = 0, 1, 2, \ldots$ **do**
Collect set of trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$
Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm
Form sample estimates for

- policy gradient $\hat{g}_k$ (using advantage estimates)
- and KL-divergence Hessian / Fisher Information Matrix $\hat{H}_k$

Compute Natural Policy Gradient update:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\,\epsilon}{\hat{g}_k^T \hat{H}_k^{-1} \hat{g}_k}} \hat{H}_k^{-1} \hat{g}_k$$

end for

Both use samples from the current policy $\pi_k = \pi(\theta_k)$

Natural Gradient Descent

Algorithm 1 Natural Policy Gradient

Input: initial policy parameters θ_0

for $k = 0, 1, 2, \ldots$ **do**

Collect set of trajectories $\mathcal{D}_k$ on policy $\pi_k = \pi(\theta_k)$

Estimate advantages $\hat{A}_t^{\pi_k}$ using any advantage estimation algorithm

Form sample estimates for

- policy gradient $\hat{g}_k$ (using advantage estimates)
- and KL-divergence Hessian / Fisher Information Matrix $\hat{H}_k$

Compute Natural Policy Gradient update:

$$\theta_{k+1} = \theta_k + \sqrt{\frac{2\epsilon}{\hat{g}_k^T \hat{H}_k^{-1} \hat{g}_k}} \hat{H}_k^{-1} \hat{g}_k$$

end for

very expensive to compute for a large number of parameters!

What is the underlying optimization problem?

We started here: $$\max_{\theta}. \quad U(\theta) = \mathbb{E}_{\tau \sim P(\tau;\theta)}[R(\tau)] = \sum_{\tau} P(\tau;\theta)R(\tau)$$

Policy gradients: $$\hat{g} \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\alpha_t^{(i)} \mid s_t^{(i)}) A(s_t^{(i)}, a_t^{(i)}), \quad \tau_i \sim \pi_\theta$$

$$\hat{g} = \mathbb{E}_t \left[\nabla_\theta \log \pi_\theta(\alpha_t \mid s_t) A(s_t, a_t) \right]$$

This result from differentiating the following objective function:

$$U^{PG}(\theta) = \mathbb{E}_t \left[\log \pi_\theta(\alpha_t \mid s_t) A(s_t, a_t) \right]$$

``don't optimize too much" constraint:

$$\max_{d}. \quad \mathbb{E}_t \left[\log \pi_{\theta+d}(\alpha_t \mid s_t) A(s_t, a_t) \right] - \lambda \mathrm{D}_{\mathrm{KL}} \left[\pi_\theta \| \pi_{\theta+d} \right]$$

We used the 1st order approximation for the 1st term, but what if d is large??

Alternative derivation

$$U(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[R(\tau) \right]$$
$$= \sum_\tau \pi_\theta(\tau) R(\tau)$$
$$= \sum_\tau \pi_{\theta_{old}}(\tau) \frac{\pi_\theta(\tau)}{\pi_{\theta_{old}}(\tau)} R(\tau)$$
$$= \mathbb{E}_{\tau \sim \pi_{\theta_{old}}} \frac{\pi_\theta(\tau)}{\pi_{\theta_{old}}(\tau)} R(\tau)$$

$$\max_\theta . \quad \mathbb{E}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} A(s_t, a_t) \right] - \lambda \mathrm{D}_{\mathrm{KL}} \left[\pi_{\theta_{old}} \| \pi_\theta \right]$$

$$\nabla_\theta U(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta_{old}}} \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_{\theta_{old}}(\tau)} R(\tau)$$

$$\nabla_\theta U(\theta) |_{\theta=\theta_{old}} = \mathbb{E}_{\tau \sim \pi_{\theta_{old}}} \nabla_\theta \log \pi_\theta(\tau) |_{\theta=\theta_{old}} R(\tau)$$

<-Gradient evaluated at theta_old is unchanged

Trust region Policy Optimization

Constrained objective:

$$\max_{\theta}. \quad \mathbb{E}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} A(s_t, a_t) \right]$$

$$\text{subject to } \mathbb{E}_t \left[\mathrm{D}_{\mathrm{KL}} \left[\pi_{\theta_{old}}(\cdot \mid s_t) \| \pi_\theta(\cdot \mid s_t) \right] \right] \leq \delta$$

Or unonstrained objective:

$$\max_{\theta}. \quad \mathbb{E}_t \left[\frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)} A(s_t, a_t) \right] - \beta \mathbb{E}_t \left[\mathrm{D}_{\mathrm{KL}} \left[\pi_{\theta_{old}}(\cdot \mid s_t) \| \pi_\theta(\cdot \mid s_t) \right] \right]$$

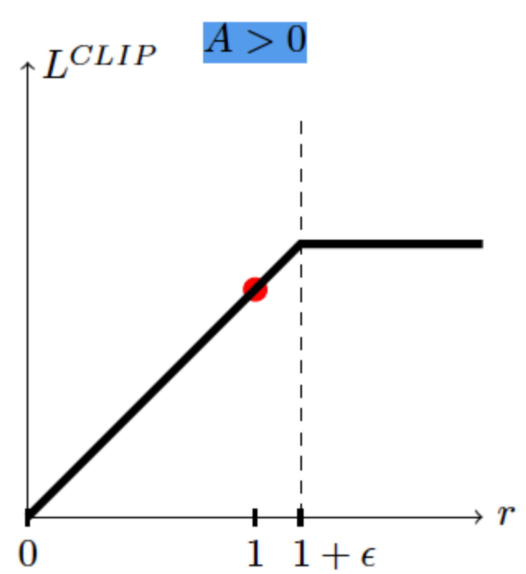
J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel. "Trust Region Policy Optimization".

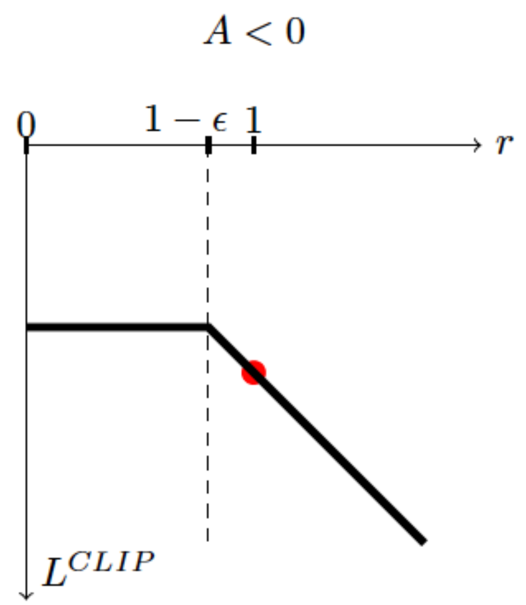
Proximal Policy Optimization

Can I achieve similar performance without second order information (no Fisher matrix!)

$$r_t(\theta) = \frac{\pi_\theta(a_t \mid s_t)}{\pi_{\theta_{old}}(a_t \mid s_t)}$$

$$\max_{\theta} . \quad L^{CLIP} = \mathbb{E}_t \left[\min \left(r_t(\theta) A(s_t, a_t), \text{clip} \left(r_t(\theta), 1 - \epsilon, 1 + \epsilon \right) A(s_t, a_t) \right) \right]$$





J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. "Proximal Policy Optimization Algorithms". (2017)

PPO: Clipped Objective

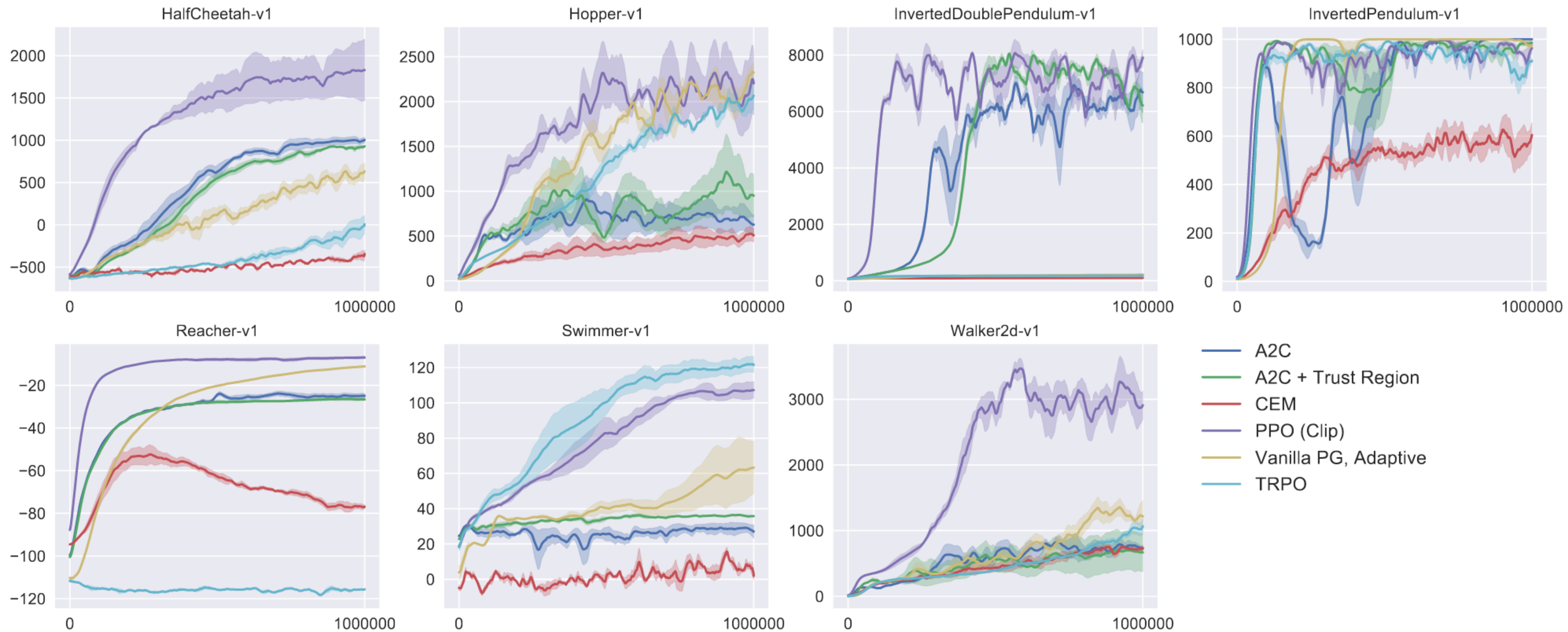


Figure: Performance comparison between PPO with clipped objective and various other deep RL methods on a slate of MuJoCo tasks. [10]

Towards Generalization and Simplicity in Continuous Control

Aravind Rajeswaran* **Kendall Lowrey*** **Emanuel Todorov** **Sham Kakade**
University of Washington Seattle
{ aravraj, klowrey, todorov, sham } @ cs.washington.edu

Training linear policies to solve control tasks with natural policy gradients

https://youtu.be/frojcskMkkY

Algorithm 1 Policy Search with Natural Gradient

Initialize policy parameters to θ_0

for $k = 1$ **to** K **do**

  Collect trajectories $\{\tau^{(1)}, \ldots \tau^{(N)}\}$ by rolling out the stochastic policy $\pi(\cdot; \theta_k)$.

  Compute $\nabla_\theta \log \pi(a_t | s_t; \theta_k)$ for each (s, a) pair along trajectories sampled in iteration k.

  Compute advantages A_k^π based on trajectories in iteration k and approximate value function V_{k-1}^π.

  Compute policy gradient according to (2).

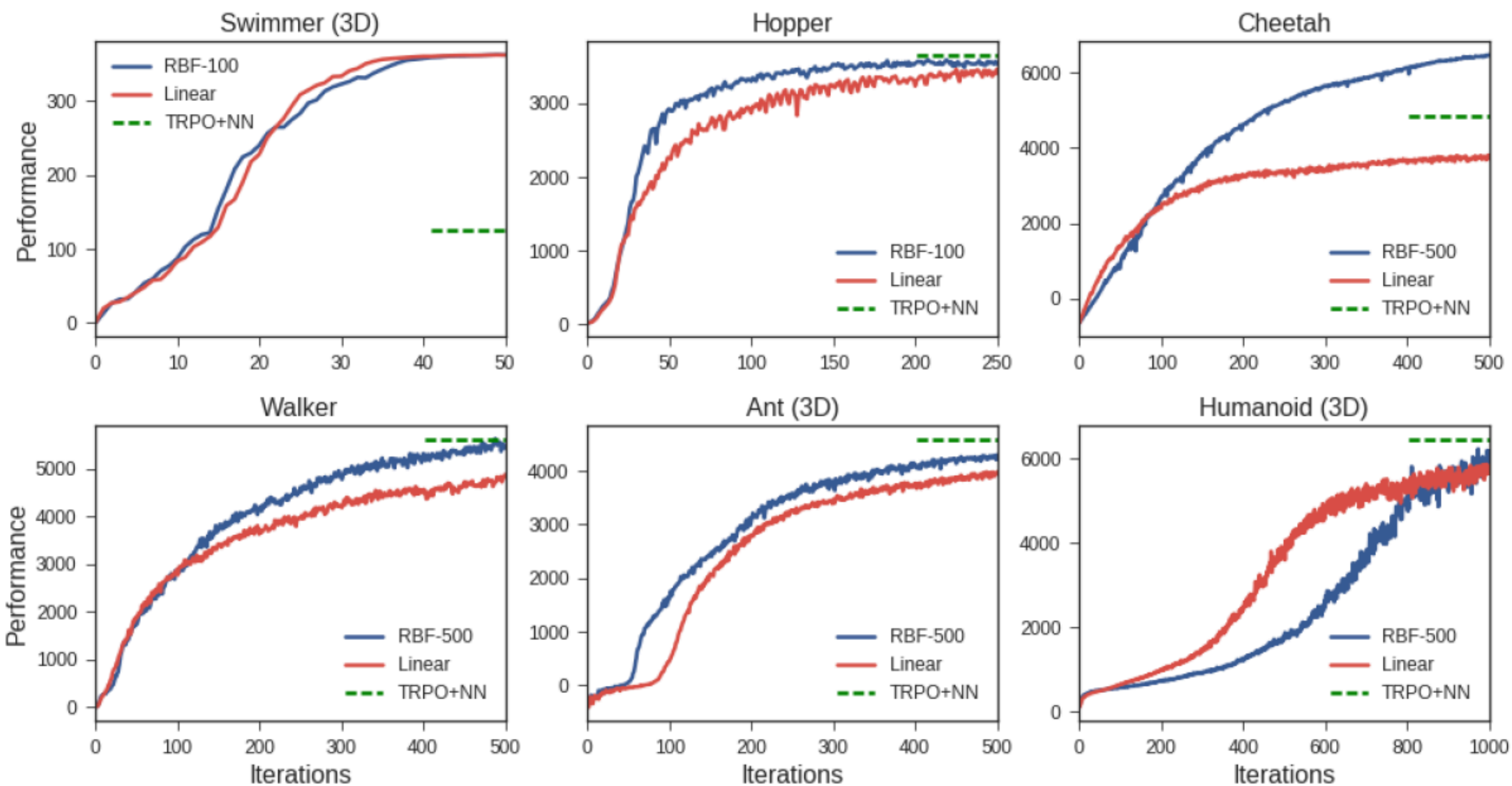
  Compute the Fisher matrix (4) and perform gradient ascent (5).

  Update parameters of value function in order to approximate $V_k^\pi(s_t^{(n)}) \approx R(s_t^{(n)})$, where $R(s_t^{(n)})$ is the empirical return computed as $R(s_t^{(n)}) = \sum_{t'=t}^{T} \gamma^{(t'-t)} r_t^{(n)}$. Here n indexes over the trajectories.

end for

State s: joint positions, joint velocities, contact info

$$a_t \sim \mathcal{N}(W s_t + b, \sigma),$$



Swimmer (3D)
Hopper
Cheetah
Walker
Ant (3D)
Humanoid (3D)
Performance
Iterations
RBF-100
RBF-500
Linear
TRPO+NN

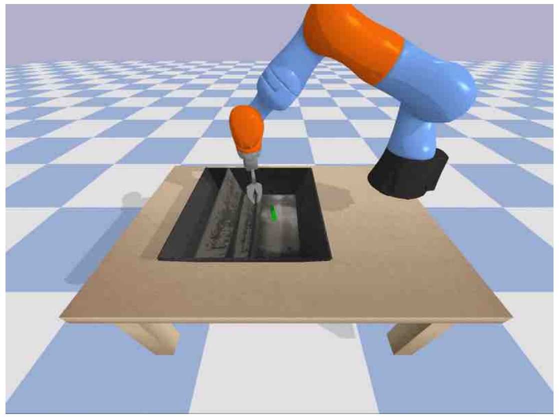
Carnegie Mellon
School of Computer Science

Deep Reinforcement Learning and Control

Multigoal RL

Katerina Fragkiadaki

So far we train one policy/value function per task, e.g., win the game of Tetris, win the game of Go, reach to a *particular* location, put the green cube inside the gray bucket, etc.



Universal value function Approximators

$$V(s;\theta) \rightarrow V(s,g;\theta)$$

$$\pi(s;\theta) \rightarrow \pi(s,g;\theta)$$

- All methods we have learnt so far can be used.
- At the beginning of an episode, **we sample not only a start state but also a goal g**, which stays constant throughout the episode
- The experience tuples should contain the goal.

$$(s,a,r,s') \rightarrow (s,g,a,r,s')$$

Universal Value Function Approximators, Schaul et al.

Universal value function Approximators

$$V(s, \theta) \rightarrow V(s, \theta, g)$$

$$\pi(s; \theta) \rightarrow \pi(s, g; \theta)$$

What should be my goal representation?

(not an easy question, same as your state representation)

- **Manual**: 3d centroids of objects, robot joint angles and velocities, 3d location of the gripper, etc.
- **Learnt**: We supply **a target image as the goal**, and an autoencoder learns to map it to an embedding vector by minimizing reconstruction loss

Hindsight Experience Replay

Marcin Andrychowicz*, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel†, Wojciech Zaremba†
OpenAI

Main idea: use failed executions under one goal g, as successful executions under an alternative goal g’ (which is where we ended spat the end of the episode)

Hindsight Experience Replay

Marcin Andrychowicz[*], Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel[†], Wojciech Zaremba[†]
OpenAI

Main idea: use failed executions under one goal g, as successful executions under an alternative goal g' (which is where we ended spat the end of the episode)

Hindsight Experience Replay

Algorithm 1 Hindsight Experience Replay (HER)

Given:

- an off-policy RL algorithm $\mathbb{A}$, ▷ e.g. DQN, DDPG, NAF, SDQN
- a strategy $\mathbb{S}$ for sampling goals for replay, ▷ e.g. $\mathbb{S}(s_0, \ldots, s_T) = m(s_T)$
- a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$. ▷ e.g. $r(s, a, g) = -[f_g(s) = 0]$

Initialize $\mathbb{A}$ ▷ e.g. initialize neural networks
Initialize replay buffer R
for episode $= 1, M$ **do**
  Sample a goal g and an initial state s_0.
  for $t = 0, T-1$ **do**
    Sample an action a_t using the behavioral policy from $\mathbb{A}$:
      $a_t \leftarrow \pi_b(s_t || g)$ ▷ || denotes concatenation
    Execute the action a_t and observe a new state s_{t+1}
  end for
  for $t = 0, T-1$ **do**
    $r_t := r(s_t, a_t, g)$
    Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in R ▷ standard experience replay
    Sample a set of additional goals for replay $G := \mathbb{S}($**current episode**$)$
    for $g' \in G$ **do**
      $r' := r(s_t, a_t, g')$
      Store the transition $(s_t||g', a_t, r', s_{t+1}||g')$ in R ▷ HER
    end for
  end for
  for $t = 1, N$ **do**
    Sample a minibatch B from the replay buffer R
    Perform one step of optimization using $\mathbb{A}$ and minibatch B
  end for
end for

Usually as additional goal we pick the goal that this episode achieved, and the reward becomes non zero

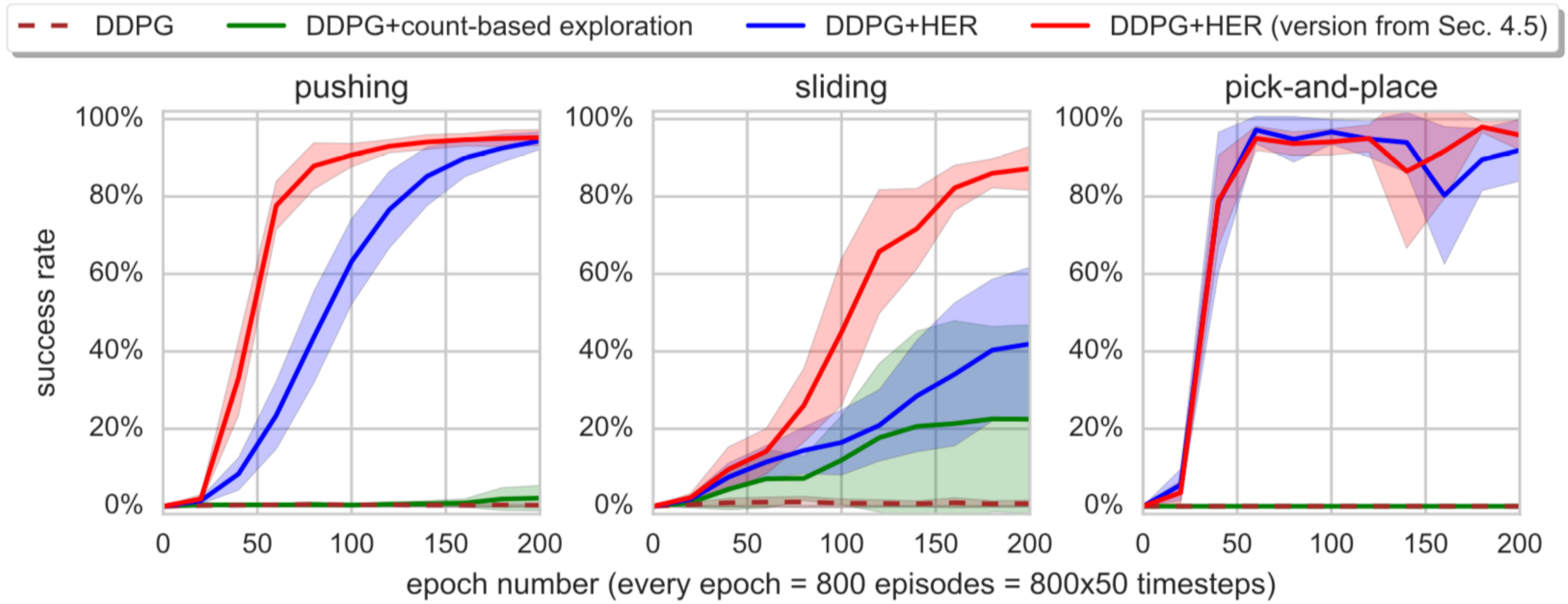
Hindsight Experience Replay

Reward shaping: instead of using binary rewards, use continuous rewards, e.g., by considering Euclidean distances from goal configuration

HER does not require reward shaping! :-)

The burden goes from designing the reward to designing the goal encoding.. :-(

Hindsight Experience Replay



Carnegie Mellon
School of Computer Science

Deep Reinforcement Learning and Control

MCTS with neural networks

Katerina Fragkiadaki

Simplest Monte-Carlo Search

- Given a model $\mathcal{M}_\nu$ and a most of the times random policy π
- For each action $a \in \mathcal{A}$
  - Simulate K episodes from current (real) state s:

$$\{s_t, a, R^k_{t+1}, S^k_{t+1}, A^k_{t+1}, ..., S^k_{\mathcal{T}}\}_{k=1}^K \sim \mathcal{M}_\nu, \pi$$

  - Evaluate action value function **of the root** by mean return

$$Q(s_t, a) = \frac{1}{K}\sum_{k=1}^{K} G_t \xrightarrow{P} q_\pi(s_t, a)$$

- Select current (real) action with maximum value

$$a_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \; Q(s_t, a)$$

Can we do better?

- Could we be **improving our simulation policy** the more simulations we obtain?
- Yes we can! We can have two policies:
  1. Internal to the tree: keep track of action values Q not only for the root but also for nodes internal to a tree we are expanding, and (maybe) use \epsilon-greedy(Q) to improve the simulation policy over time
  2. External to the tree: we do not have Q estimates and thus we use a random policy

In MCTS, the simulation policy improves

- Any better ideas for the simulation policy?

Monte-Carlo Tree Search

We will allocate samples more efficiently!

- **In MCTS, the simulation policy improves**
- Each simulation consists of two phases (in-tree, out-of-tree)
  - Tree policy (improves): pick actions to maximize $Q(s, a)$
  - Default policy (fixed): pick actions often randomly
- Repeat (each simulation)
  - Evaluate states $Q(s, a)$ by Monte-Carlo evaluation
  - **Improve there policy,** e.g. by $\epsilon - \text{greedy}(Q)$
- Converges on the optimal search tree assuming each action in the tree is tried infinitely often.

Monte-Carlo Tree Search

Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```

The state is inside the tree

The state is in the frontier

expansion

Monte-Carlo Tree Search

MCTS helper functions

Sample actions based on UCB score

```
function UCB_sample(state):
    weights = []
    for child of state:
        w = child.value + C * sqrt(ln(state.visits) / child.visits)
        weights.append(w)
    distribution = [w / sum(weights) for w in weights]
    return child sampled according to distribution
```

unrolling

```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

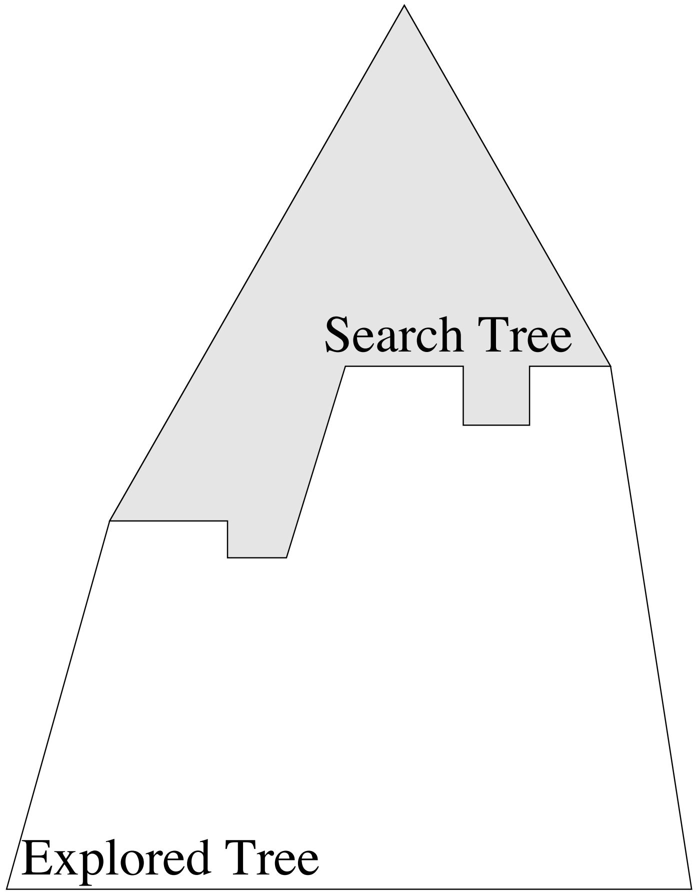
Monte-Carlo Tree Search

MCTS helper functions

```
function expand(state):
    state.visits = 1
    state.value = 0


function update value(state, winner):

    if winner == state.turn:
        state.value += 1
    else:
        state.value -= 1
```

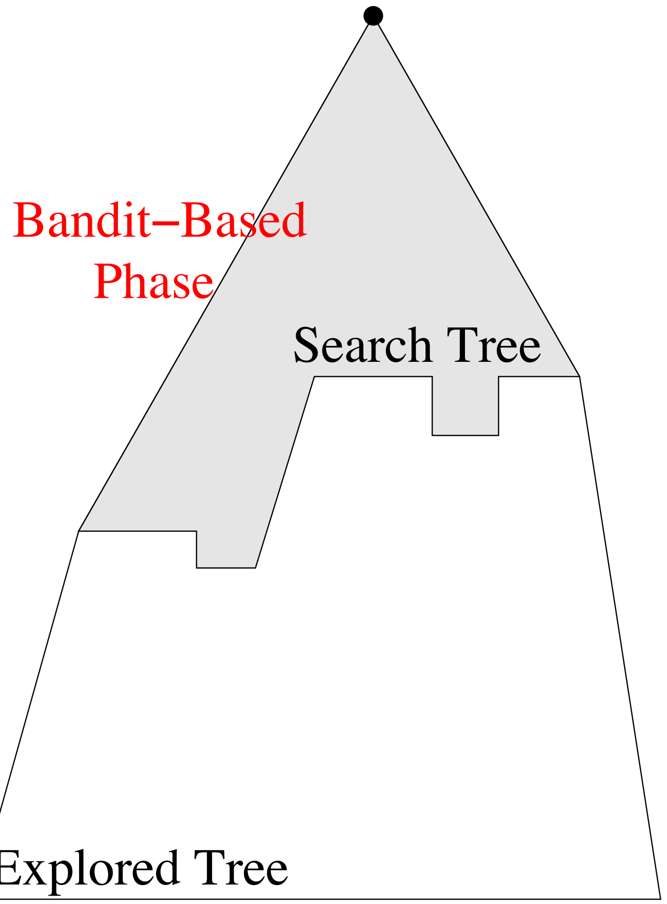
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



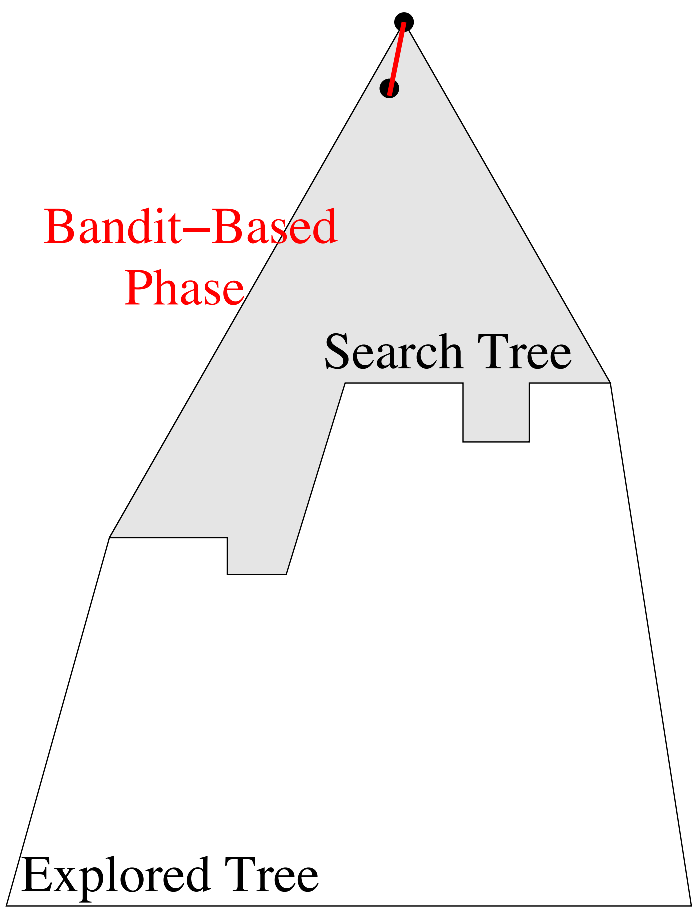
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



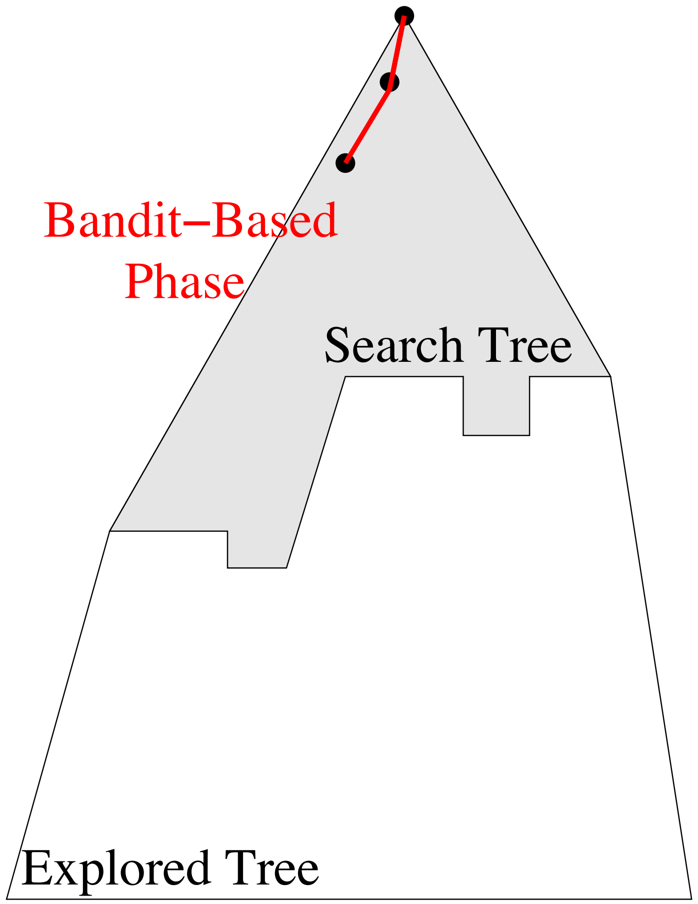
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



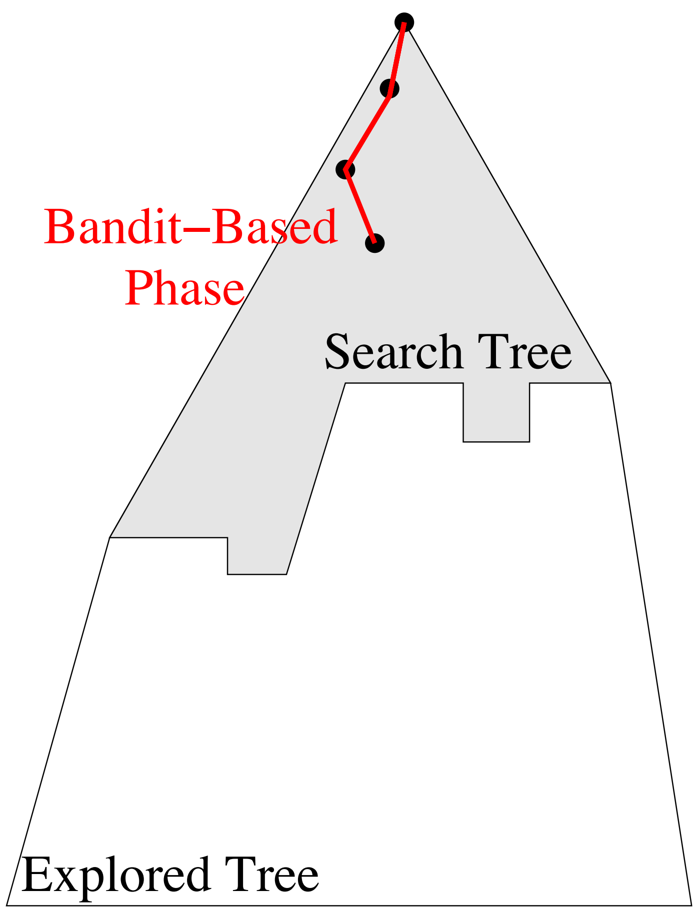
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



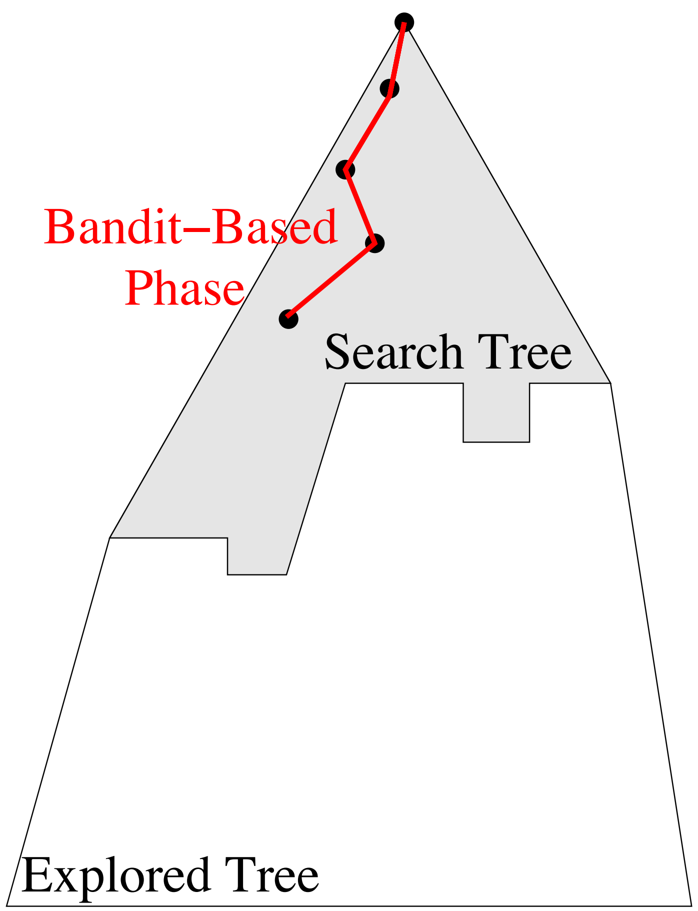
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



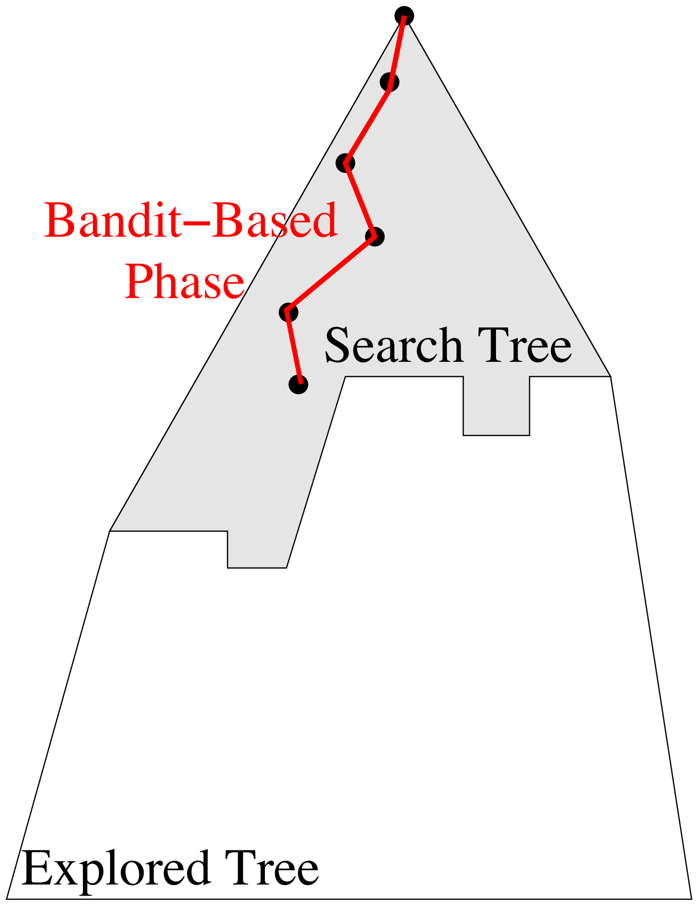
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



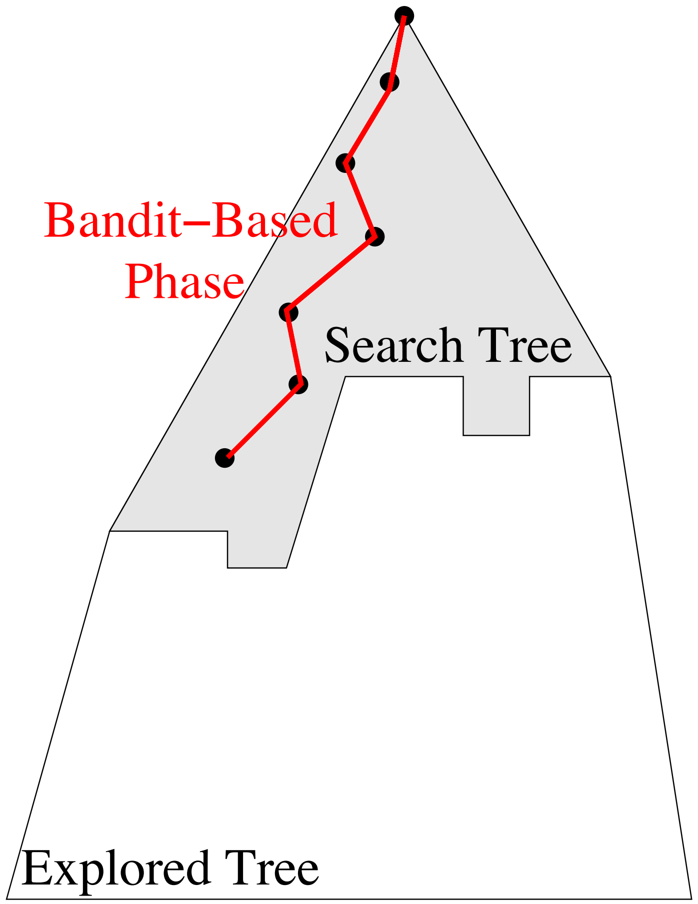
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



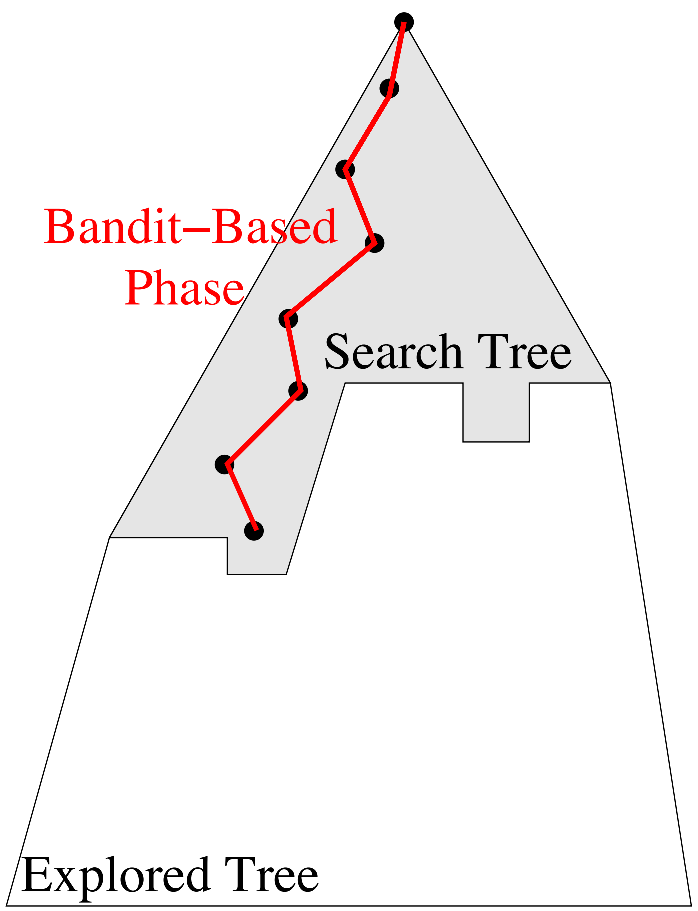
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



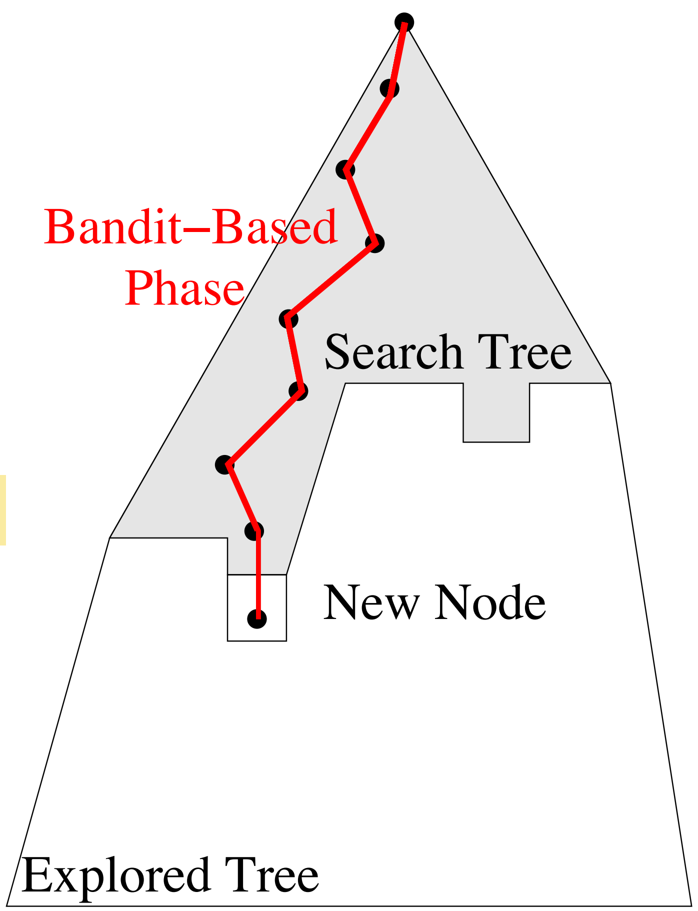
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



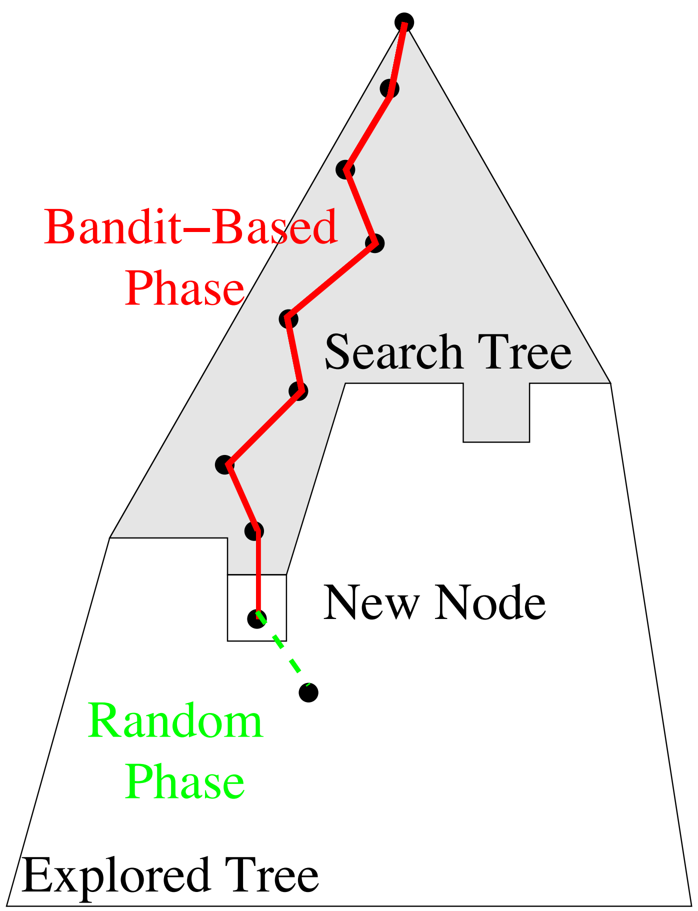
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



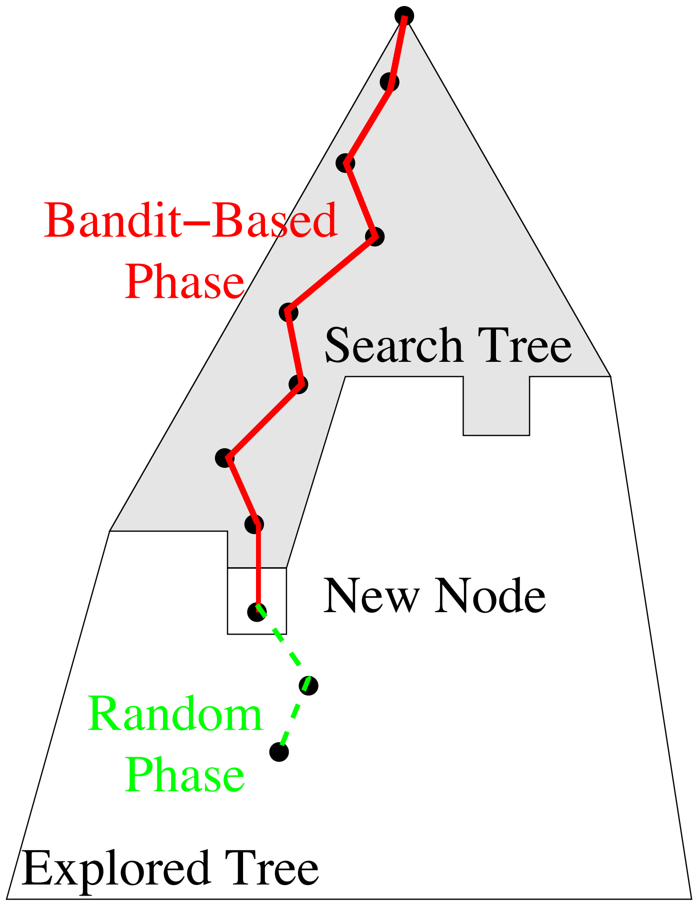
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)

function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```



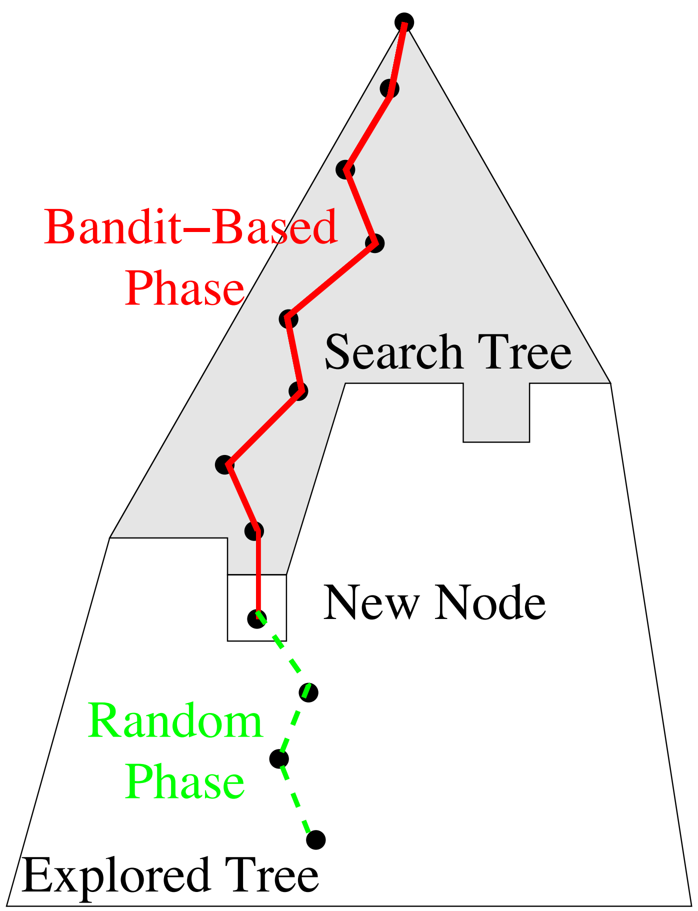
Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)


   function random_playout(state):
       if is_terminal(state):
           return winner
       else: return random_playout(random_move(state))
```



Basic MCTS pseudocode

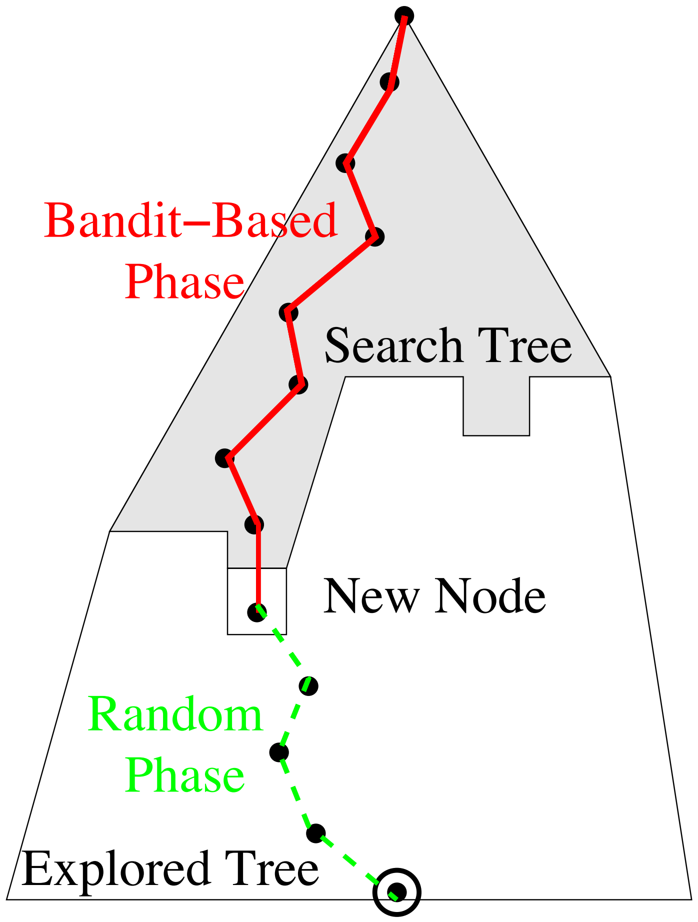
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```




```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

Basic MCTS pseudocode

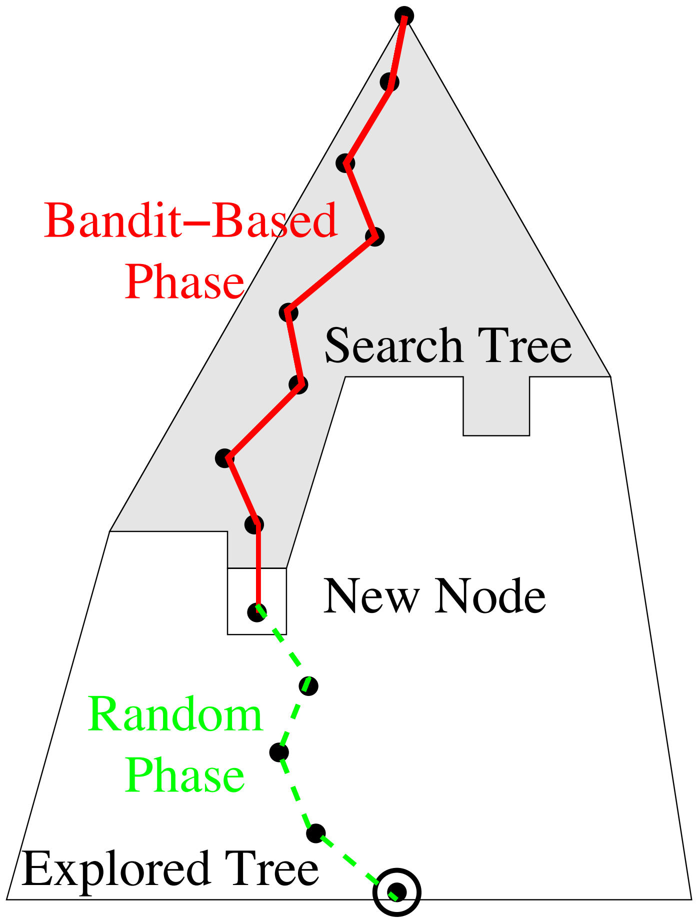
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```




```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

Basic MCTS pseudocode

```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
            next_state = state
        winner = random_playout(next_state)
    update_value(state, winner)
```



Can we do better?

Can we inject prior knowledge into value functions to be estimated and actions to be tried, instead of initializing uniformly?

Monte-Carlo Tree Search

1. **Selection**
   - Used for nodes we have seen before
   - Pick according to UCB
2. **Expansion**
   - Used when we reach the frontier
   - Add one node per playout
3. **Simulation**
   - Used beyond the search frontier
   - Don’t bother with UCB, just play randomly
4. **Backpropagation**
   - After reaching a terminal node
   - Update value and visits for states expanded in selection and expansion

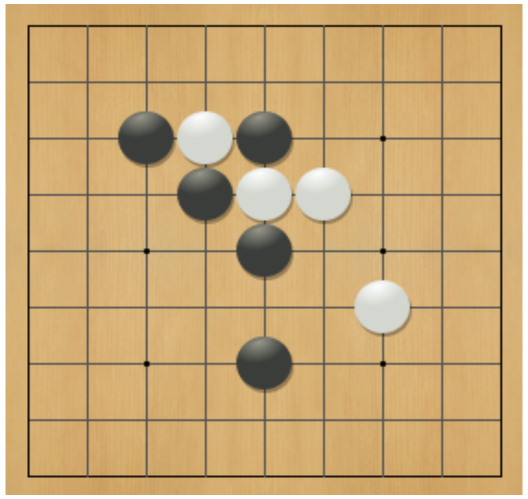
Bandit based Monte-Carlo Planning, Kocsis and Szepesvari, 2006

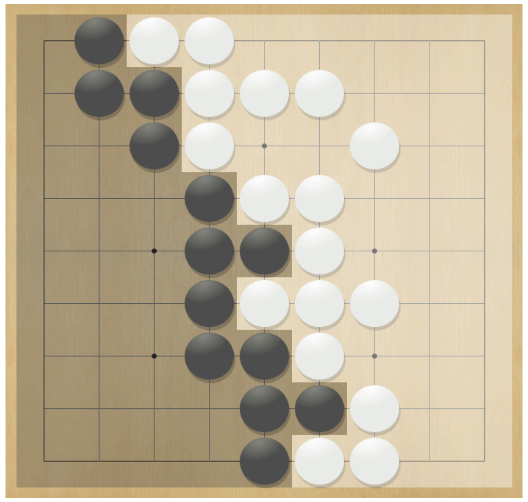
Case Study: the Game of Go

- The ancient oriental game of Go is 2500 years old
- Considered to be the hardest classic board game
- Considered a grand challenge task for AI (*John McCarthy*)
- Traditional game-tree search has failed in Go

Rules of Go

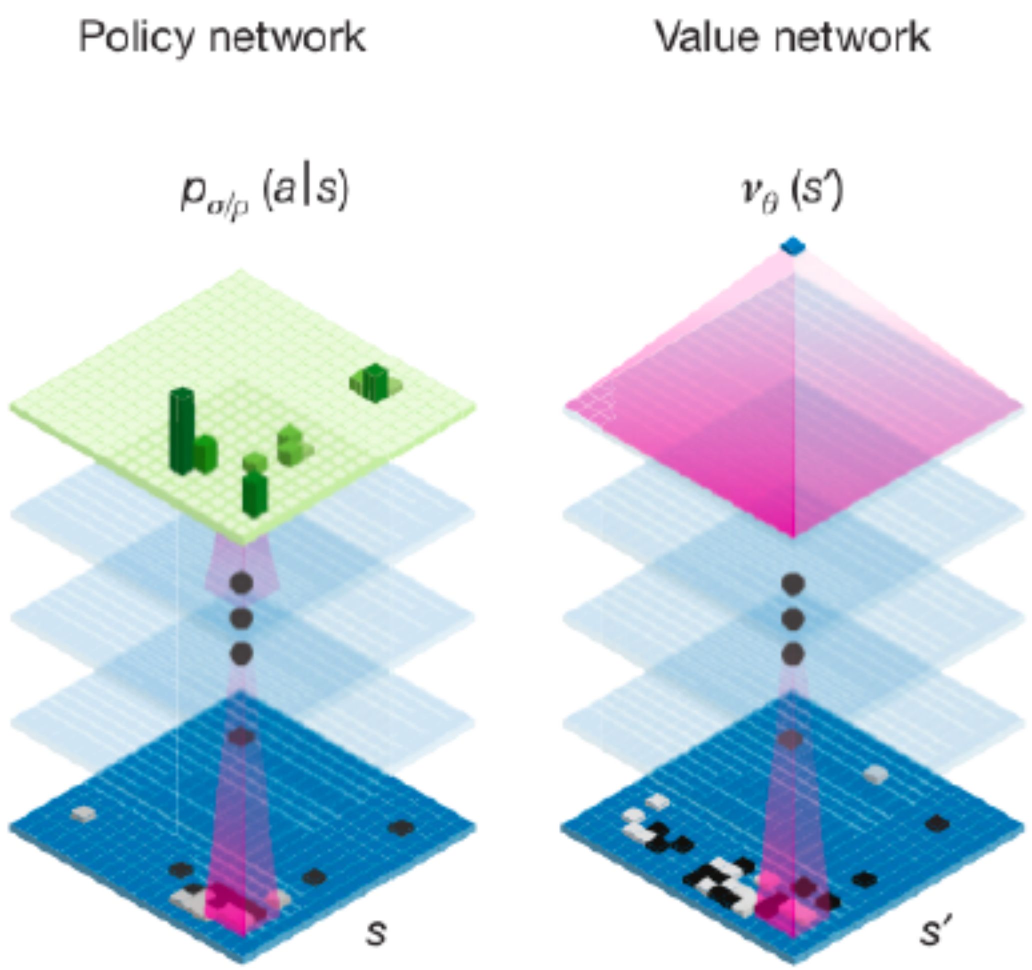
- Usually played on 19x19, also 13x13 or 9x9 board
- Simple rules, complex strategy
- Black and white place down stones alternately
- Surrounded stones are captured and removed
- The player with more territory wins the game





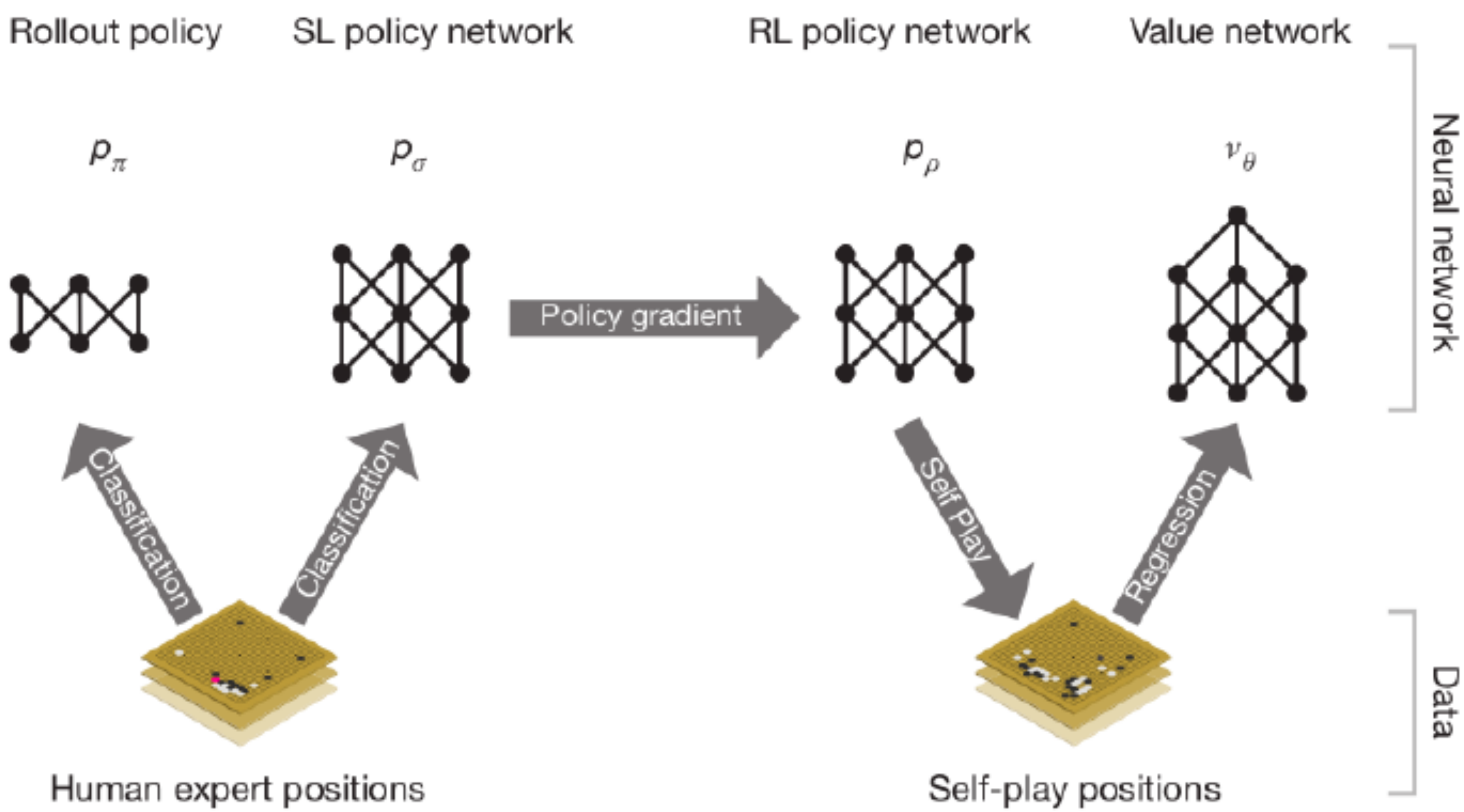
AlphaGo: Learning-guided MCTS

- Value neural net to evaluate board positions
- Policy neural net to select moves
- Combine those networks with MCTS



AlphaGo: Learning-guided search

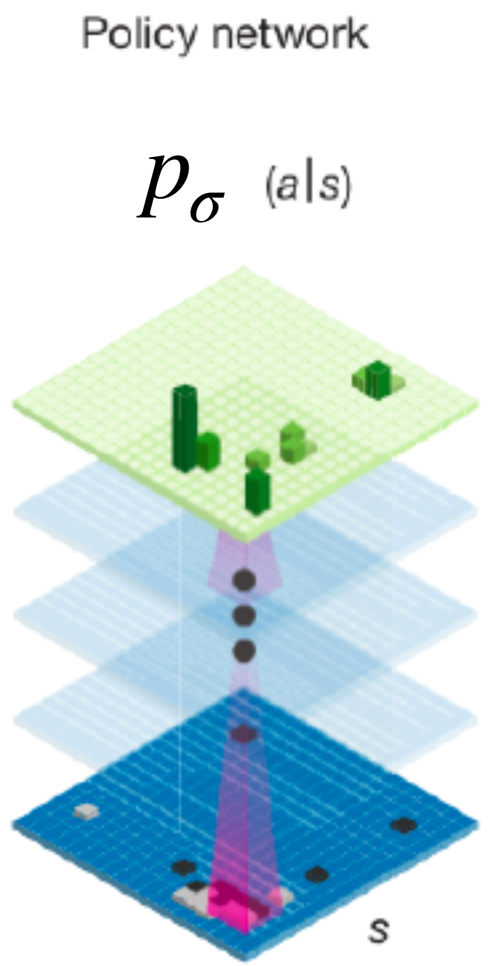
1. Train two action policies, one cheap (rollout) policy and one expensive policy by mimicking expert moves (standard supervised learning).
2. Then, train a new policy p_ρ with RL and self-play initialized from SL policy.
3. Train a value network that predicts the winner of games played by p_ρ against itself.



Supervised learning of policy networks

- Objective: predicting expert moves
- Input: randomly sampled state-action pairs (s, a) from expert games
- Output: a probability distribution over all legal moves a.

SL policy network: 13-layer policy network trained from 30 million positions. The network predicted expert moves on a held out test set with an accuracy of 57.0% using all input features, and 55.7% using only raw board position and move history as inputs, compared to the state-of-the-art from other research groups of 44.4%.

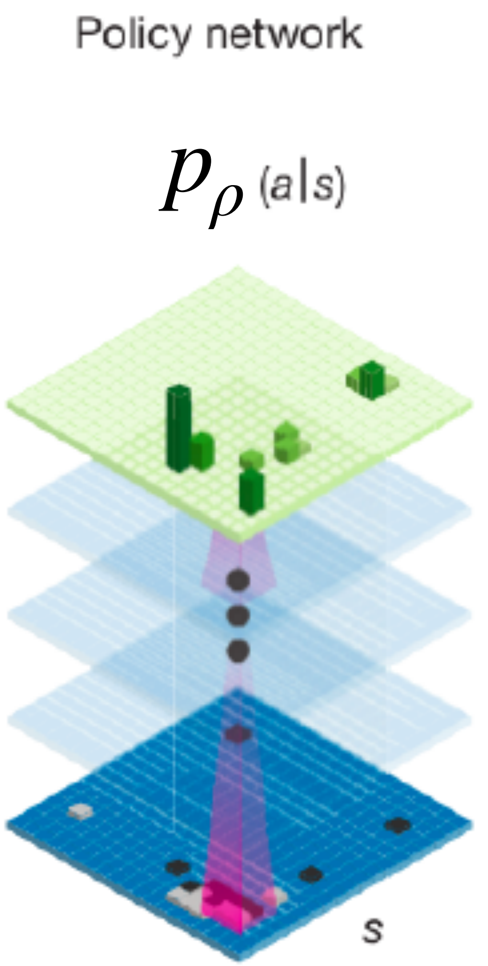


Reinforcement learning of policy networks

- Objective: improve over SL policy
- Weight initialization from SL network
- Input: Sampled states during self-play
- Output: a probability distribution over all legal moves a.

Rewards are provided only at the end of the game, +1 for winning, -1 for loosing

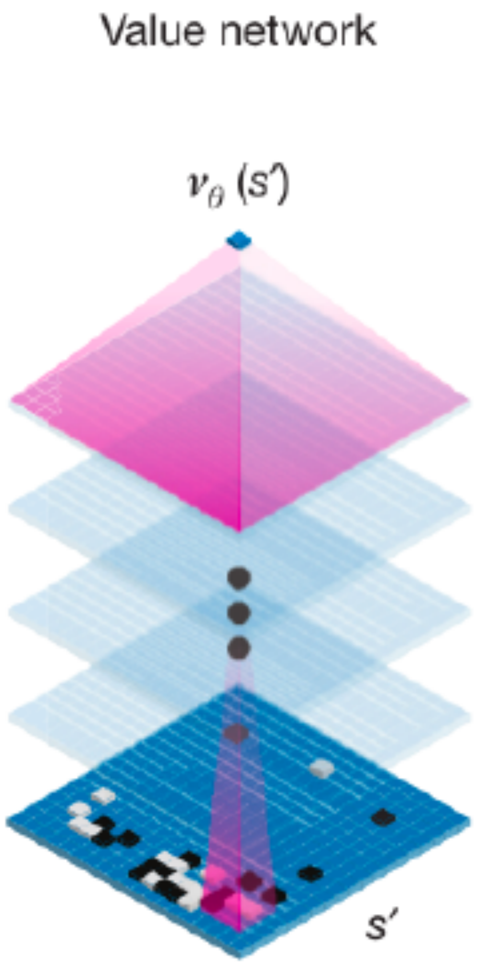
$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t \mid s_t)}{\partial \rho} z_t$$



The RL policy network won more than 80% of games against the SL policy network.

Reinforcement learning of value networks

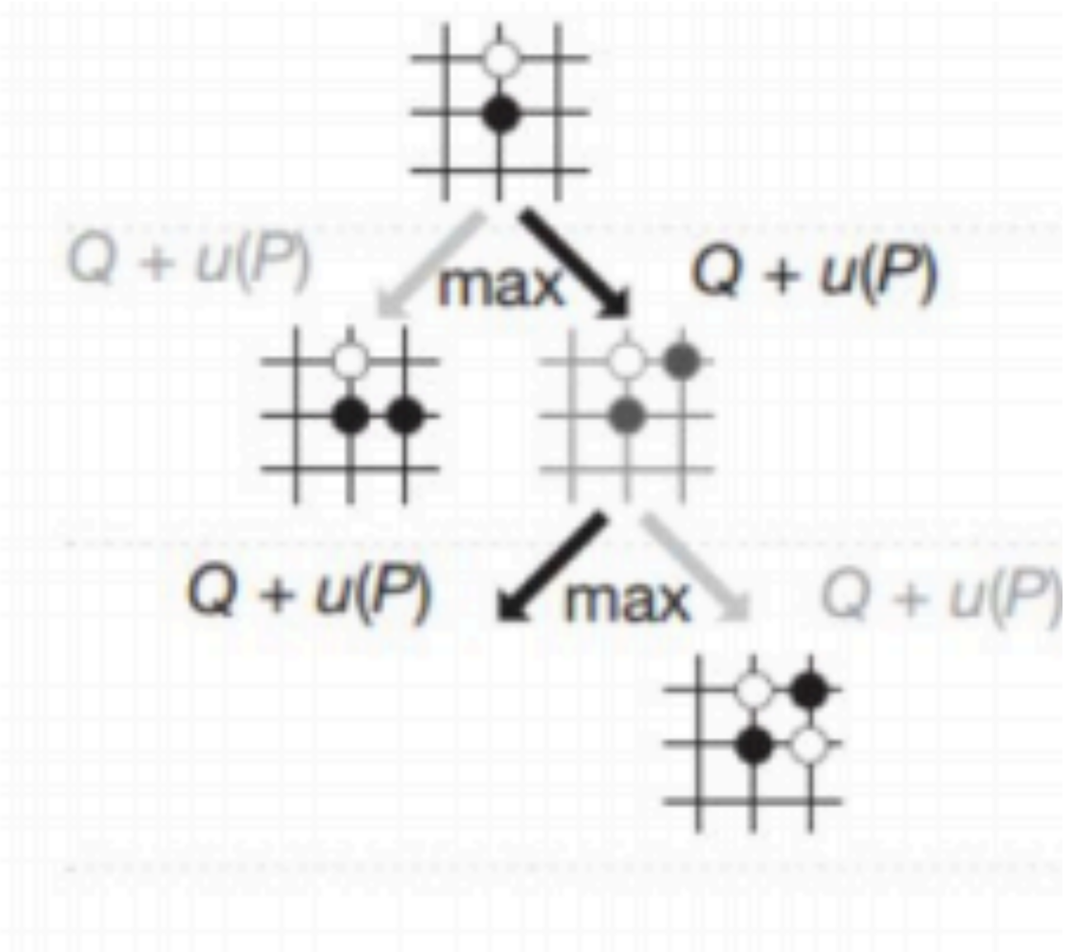
- Objective: Estimating a value function $v_p(s)$ that predicts the outcome from position s of games **played by using RL policy p for both players** (in contrast to min-max search)
- Input: Sampled states during self-play, 30 million distinct positions, each sampled from a separate game, played by the RL policy against itself.
- Output: a scalar value



Trained by regression on state-outcome pairs (s, z) to minimize the mean squared error between the predicted value v(s), and the corresponding outcome z.

MCTS + Policy/ Value networks

Selection: selecting actions within the expanded tree



Tree policy

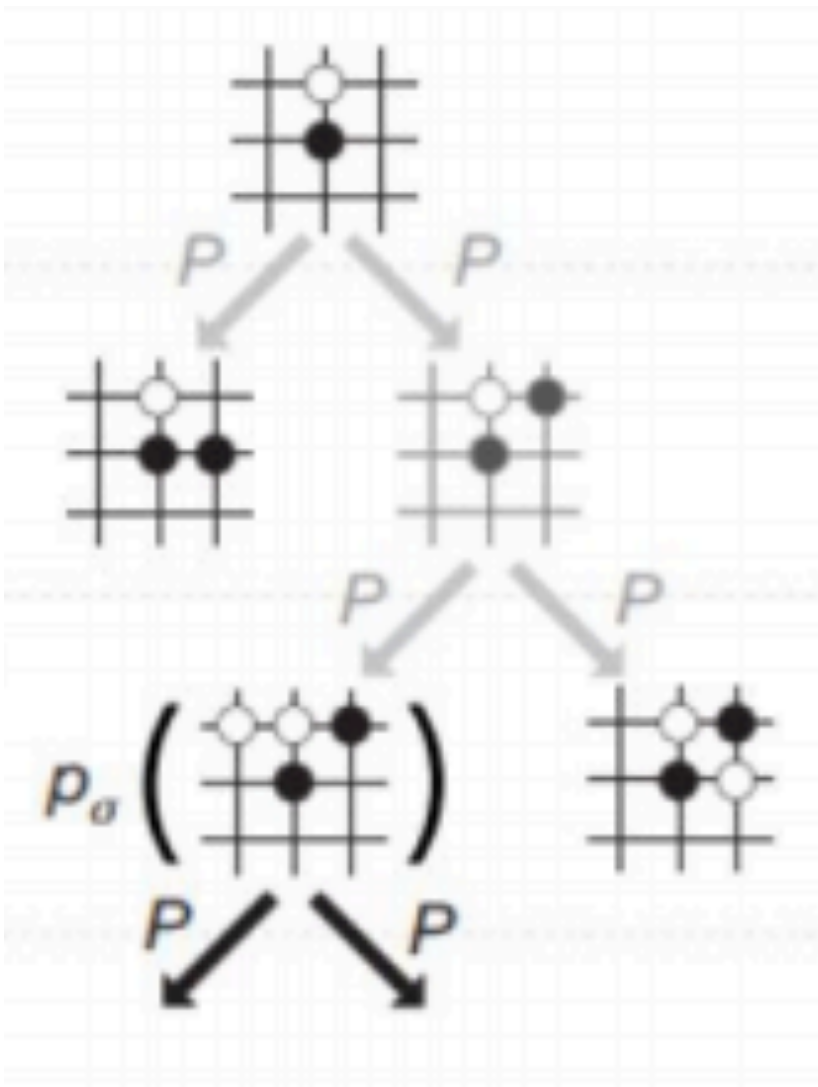
$$a_t = \text{argmax}_a(Q(s_t, a) + u(s_t, a))$$

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

- a_t - action selected at time step *t* from board s_t
- $Q(s_t, a)$ - average reward collected so far from MC simulations
- $P(s, a)$ - prior expert probability of playing moving *a* provided by SL policy
- $N(s, a)$ - number of times we have visited parent node
- *u* acts as a bonus value
    - Decays with repeated visits

MCTS + Policy/ Value networks

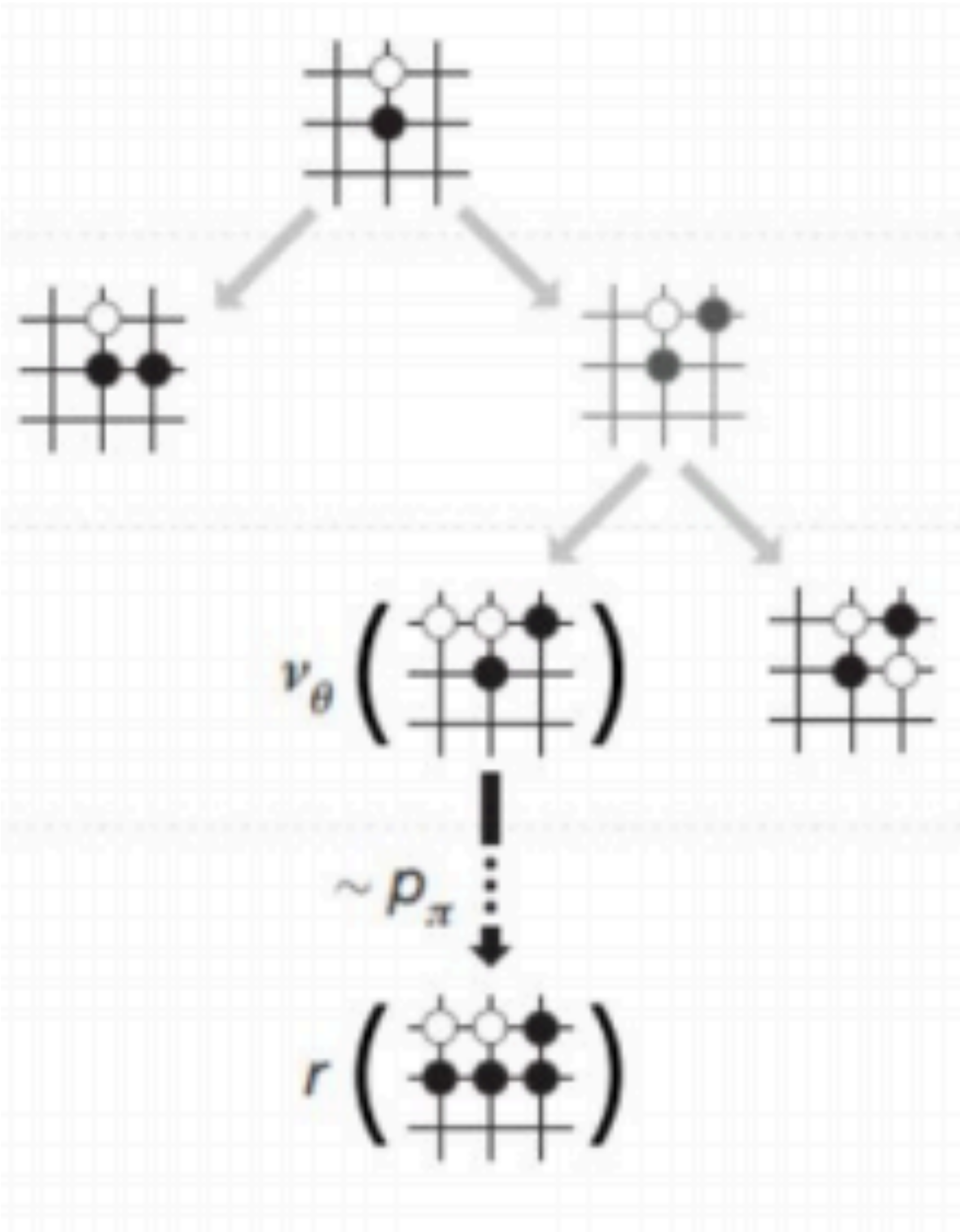
Expansion: when reaching a leaf, play the action with highest score from p_σ



- When leaf node is reached, it has a chance to be expanded
- Processed once by SL policy network (p_σ) and stored as prior probs *P(s, a)*
- Pick child node with highest prior prob

MCTS + Policy/ Value networks

Simulation/Evaluation: use the rollout policy to reach to the end of the game



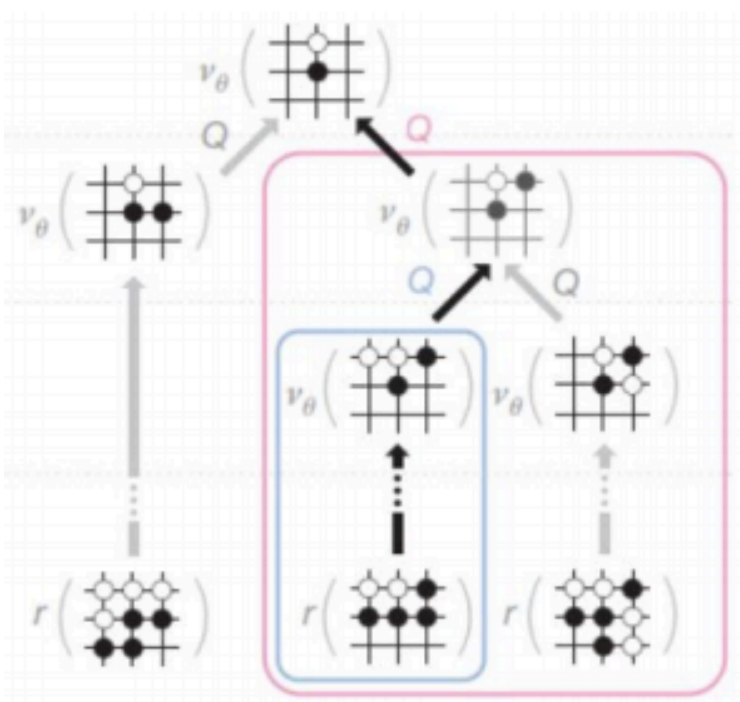
- From the selected leaf node, run multiple simulations in parallel using the rollout policy
- Evaluate the leaf node as:

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

- v_θ - value from value function of board position s_L
- z_L - Reward from fast rollout p_π
  - Played until terminal step
- λ - mixing parameter
  - Empirical

MCTS + Policy/ Value networks

Backup: update visitation counts and recorded rewards for the chosen path inside the tree:



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

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

- Extra index *i* is to denote the i^{th} simulation, *n* total simulations
- Update visit count and mean reward of simulations passing through node
- Once search completes:
  - Algorithm chooses the most visited move from the root position

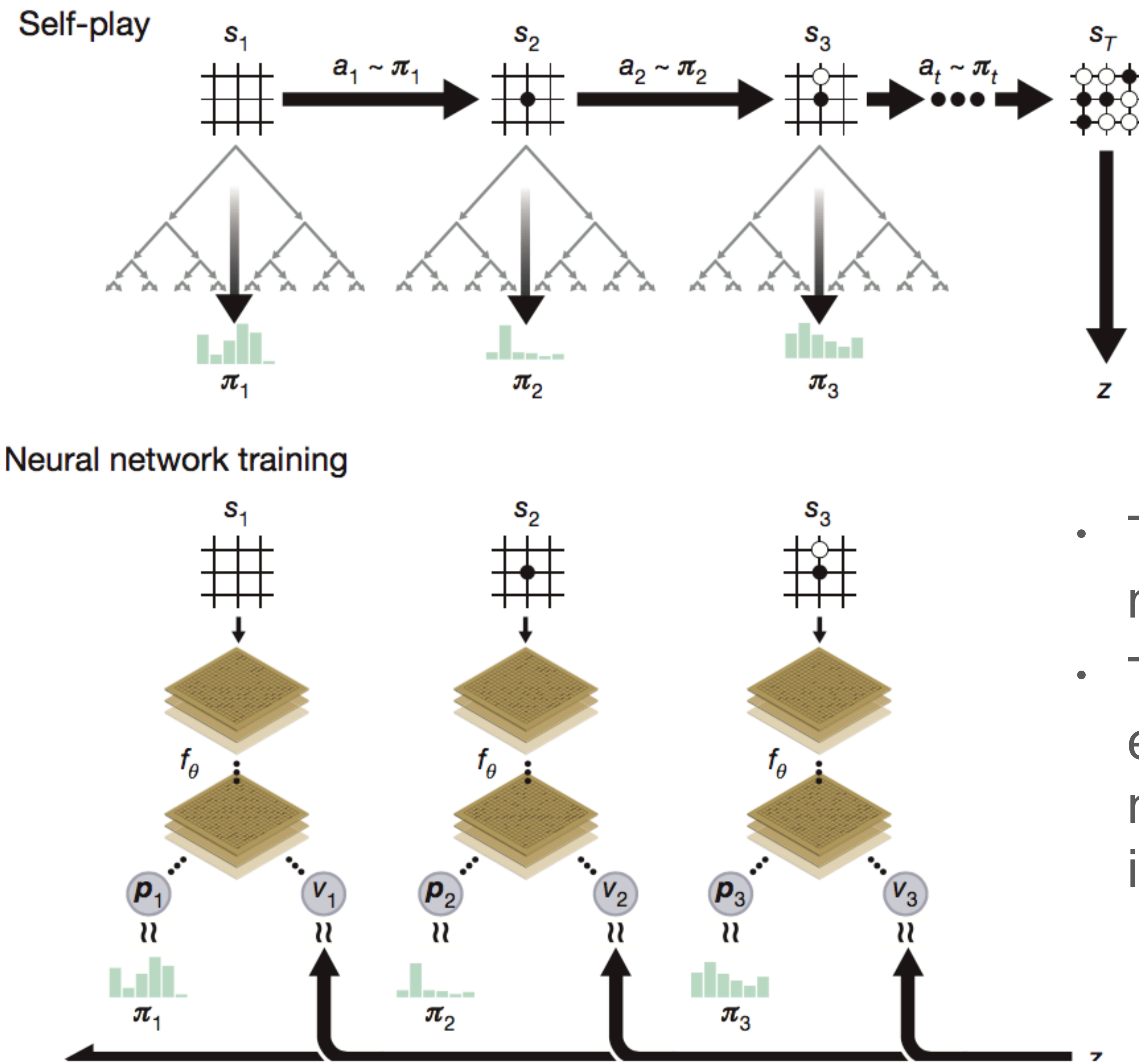
AlphaGoZero: Lookahead search during training!

- So far, look-ahead search was used for online planning at test time!
- AlphaGoZero uses it during training instead, for improved exploration during self-play
- AlphaGo trained the RL policy using the current policy network p_ρ and a randomly selected previous iteration of the policy network as opponent (for exploration).
- The intelligent exploration in AlphaGoZero gets rid of human supervision.

AlphaGoZero: Lookahead search during training!

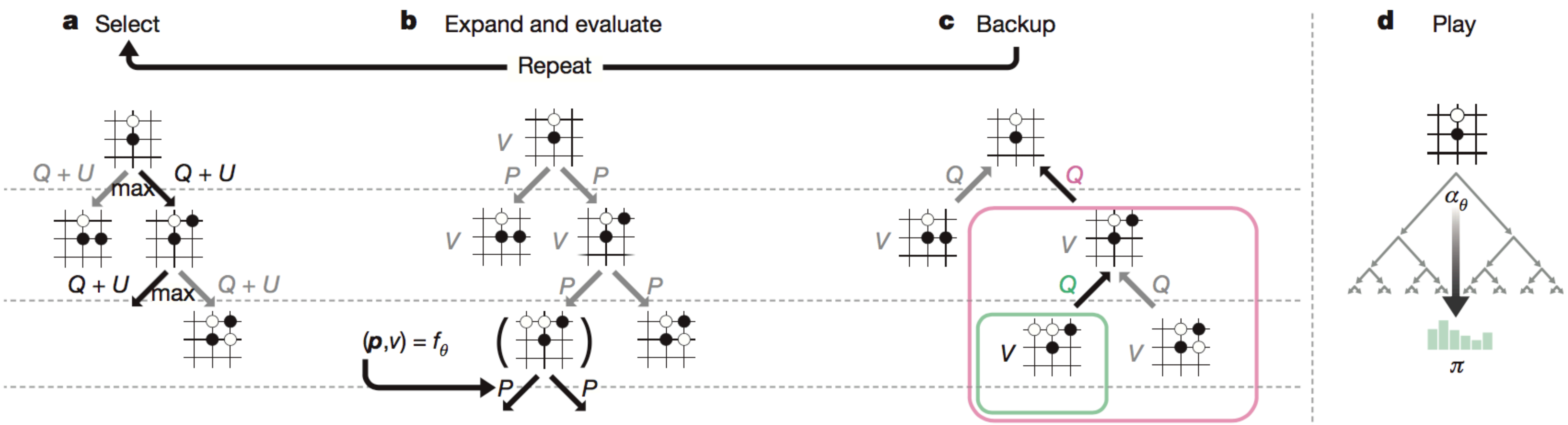
- Given any policy, a MCTS guided by this policy will produce an improved policy (policy improvement operator)
- Train to mimic such improved policy

MCTS as policy improvement operator



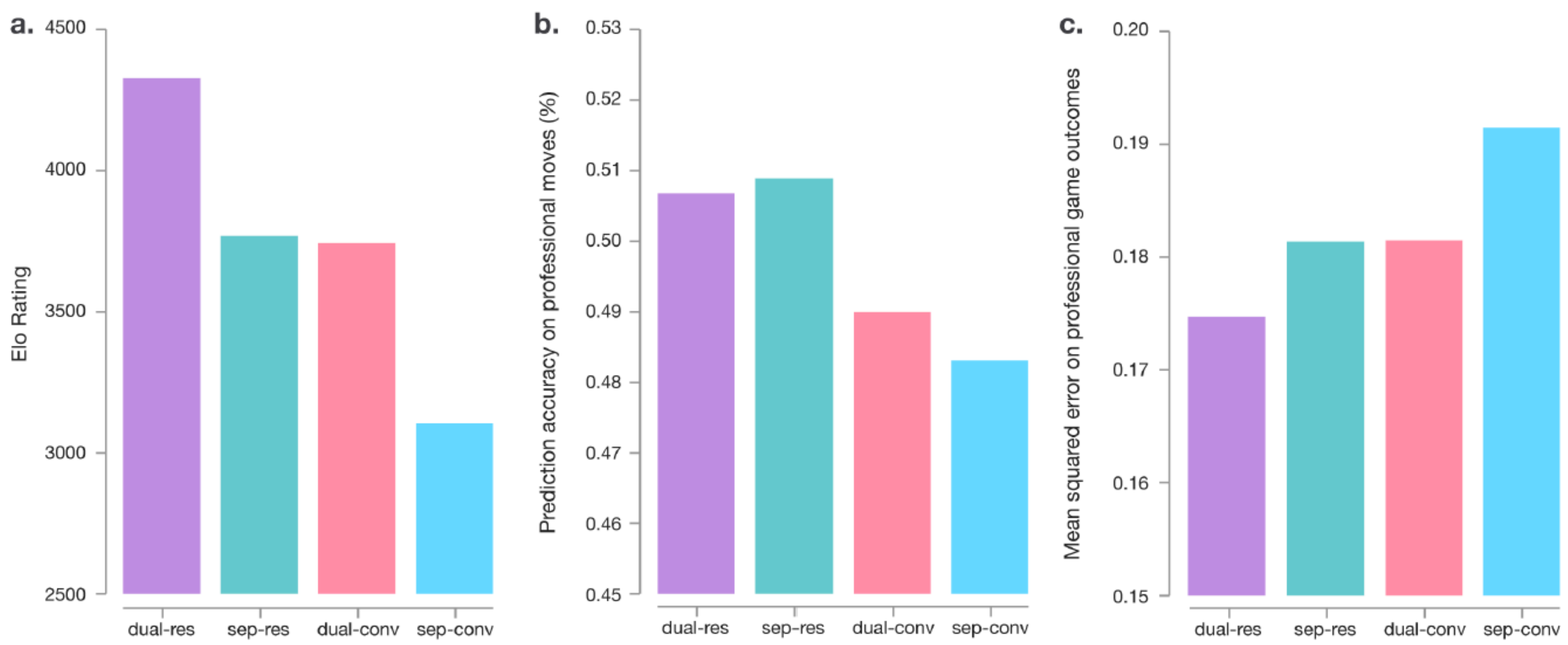
- Train so that the policy network mimics this improved policy
- Train so that the position evaluation network output matches the outcome (same as in AlphaGo)

MCTS: no MC rollouts till termination

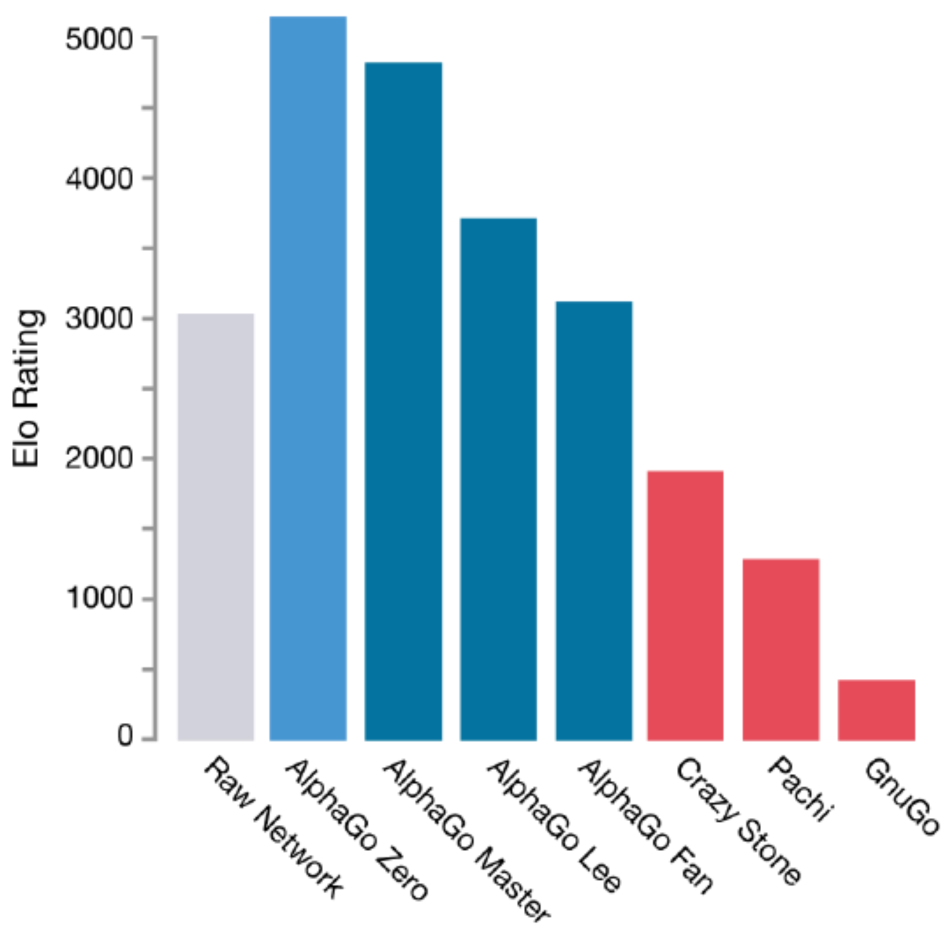


MCTS: using always value net evaluations of leaf nodes, no rollouts!

Architectures

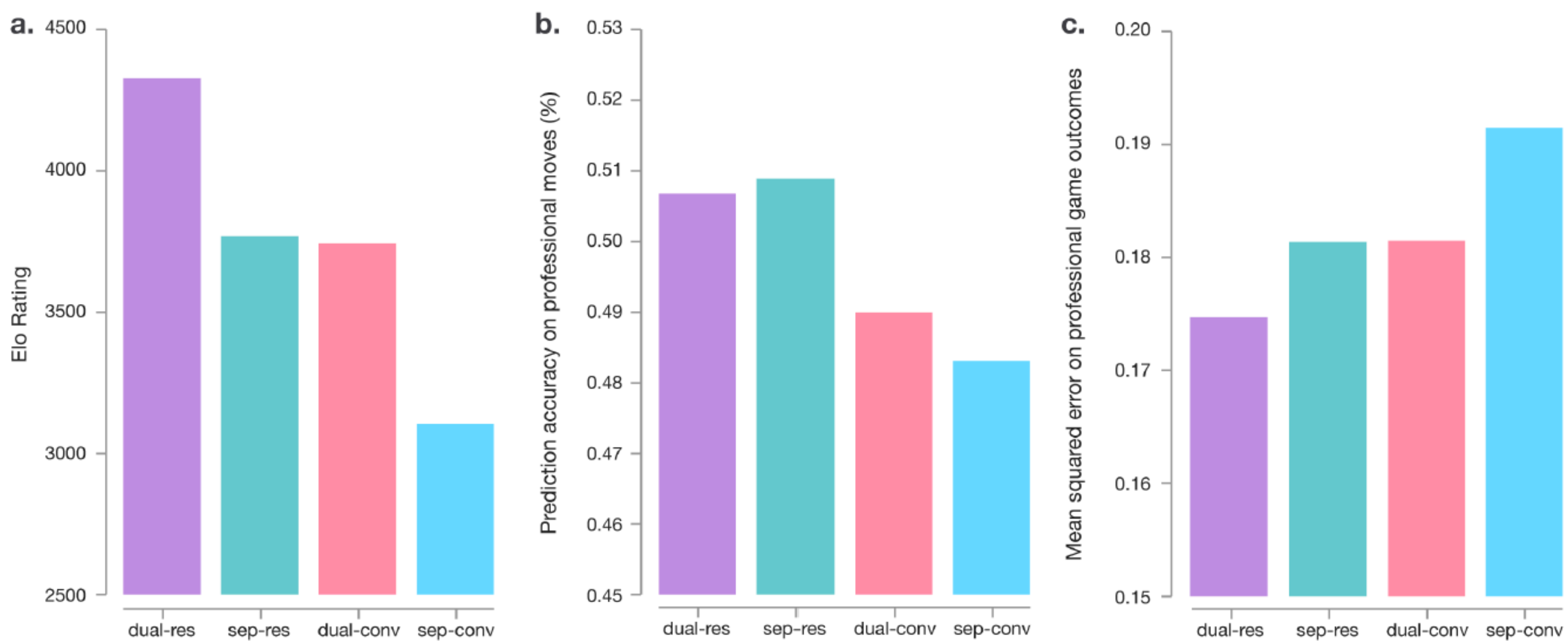


- Resnets help
- Jointly training the policy and value function using the same main feature extractor helps



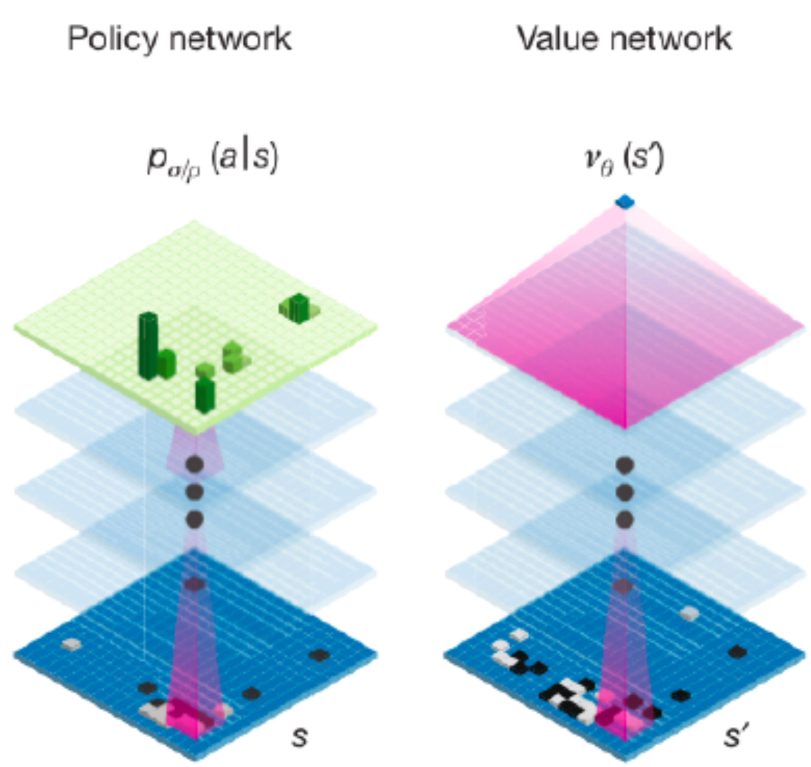
- Lookahead tremendously improves the basic policy

Architectures

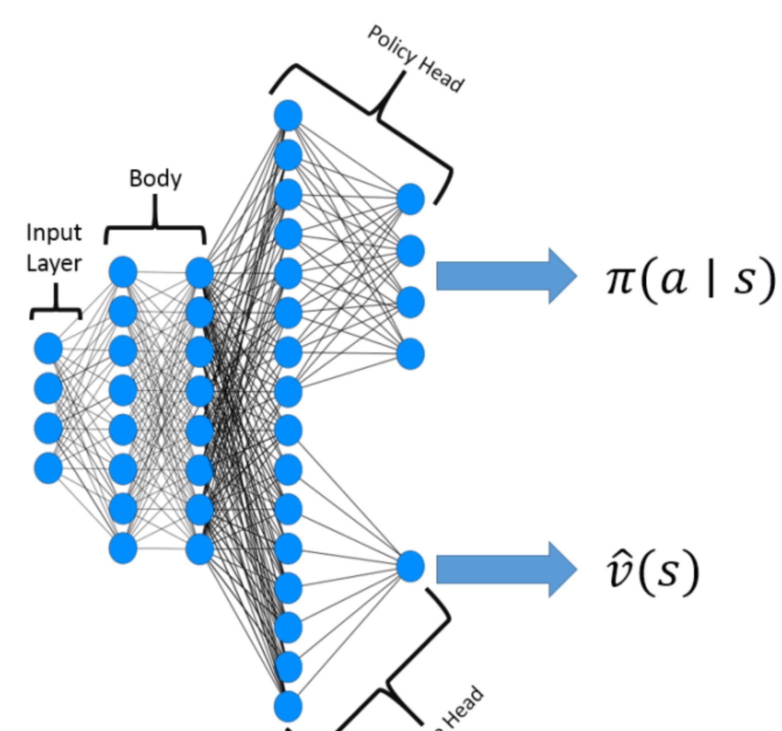


- Resnets help
- Jointly training the policy and value function using the same main feature extractor helps

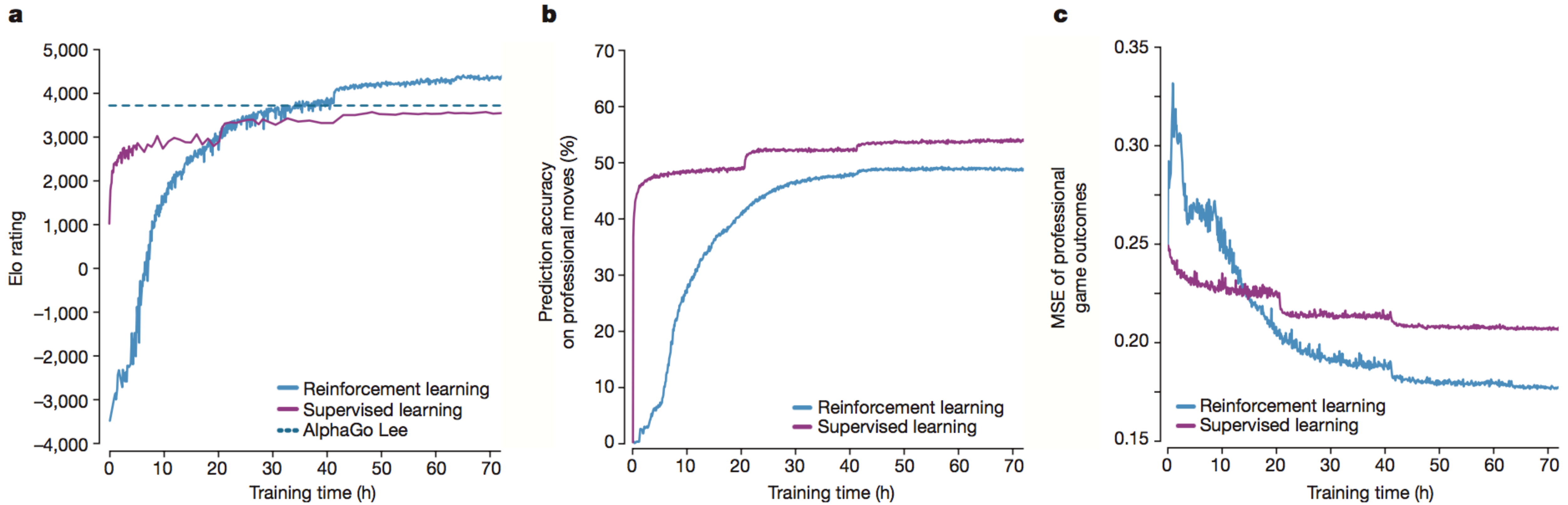
Separate policy/value nets



Joint policy/value nets



RL VS SL



a
5,000
4,000
3,000
2,000
1,000
0
–1,000
–2,000
–3,000
–4,000
Elo rating
0 10 20 30 40 50 60 70
Training time (h)
Reinforcement learning
Supervised learning
AlphaGo Lee
b
70
60
50
40
30
20
10
0
Prediction accuracy on professional moves (%)
0 10 20 30 40 50 60 70
Training time (h)
Reinforcement learning
Supervised learning
c
0.35
0.30
0.25
0.20
0.15
MSE of professional game outcomes
0 10 20 30 40 50 60 70
Training time (h)
Reinforcement learning
Supervised learning