Carnegie Mellon
School of Computer Science

Deep Reinforcement Learning and Control

Monte Carlo Tree Search

CMU 10703

Katerina Fragkiadaki

Part of slides inspired by Sebag, Gaudel

Finite Markov Decision Process

A **Finite Markov Decision Process** is a tuple $(\mathcal{S}, \mathcal{A}, T, r, \gamma)$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- T is a state transition probability function

$$T(s'|s, a) = \mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a]$$

- r is a reward function

$$r(s, a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a]$$

- γ is a discount factor $\gamma \in [0, 1]$

Definitions

Definitions

Learning: the acquisition of knowledge or skills through experience, study, or by being taught.

Planning: any computational process that uses a model to create or improve a policy

$$\text{Model} \xrightarrow{\text{Planning}} \text{Policy}$$

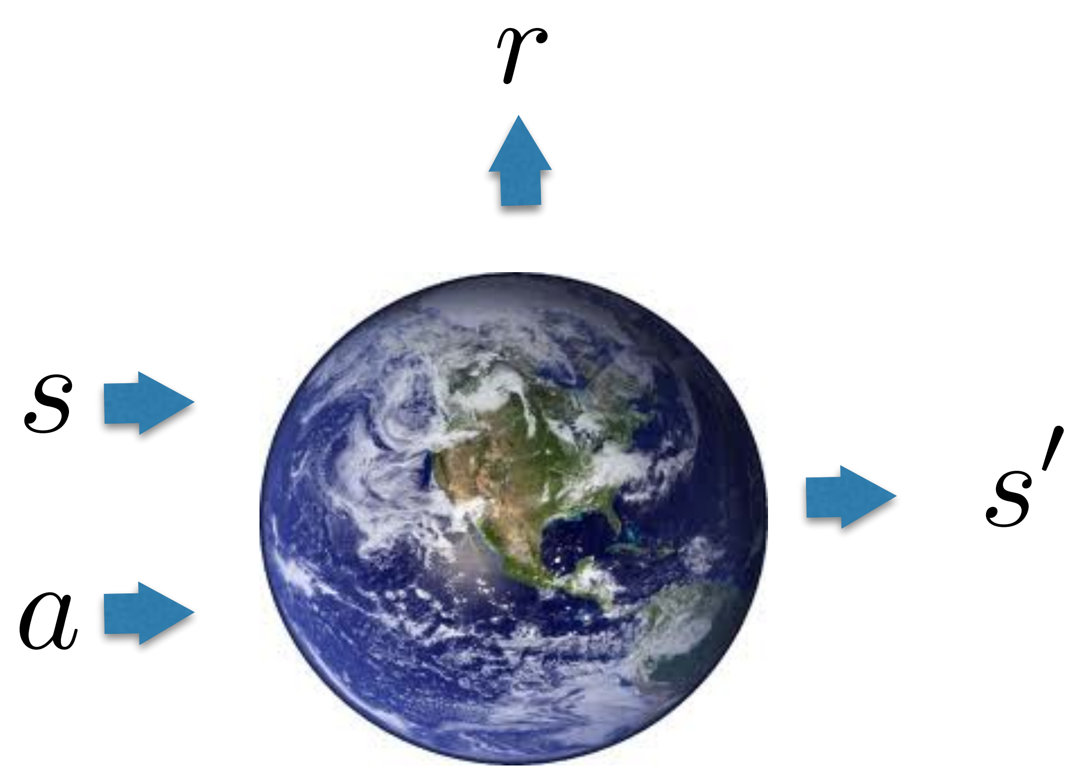
This lecture

Computing value functions combining learning and planning using Monte Carlo Tree Search

Computing value functions combining learning and planning in other ways will be revisited in later lectures

Model

Anything the agent can use to predict how the environment will respond to its actions, concretely, the state transition T(s’|s,a) and reward R(s,a).

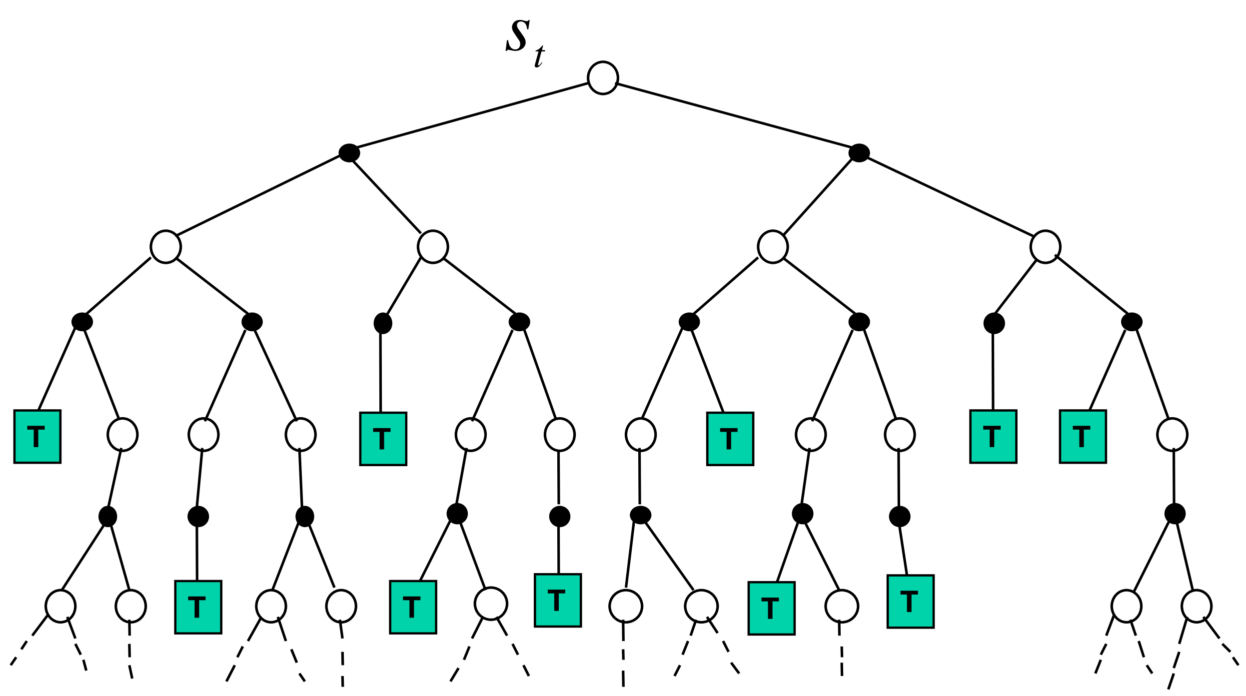


Ground-truth

this includes transitions of the state of the environment and the state of the agent

Online Planning with Search

1. Build a search tree **with the current state of the agent** at the root
2. Compute value functions using simulated episodes (reward usually only on final state, e.g., win or loose)
3. Select the next move to execute
4. Execute it
5. GOTO 1



Why online planning?

Why don't we learn a value function directly for every state offline, so that we do not waste time online?

- Because the environment has many many states (consider Go 10^170, Chess 10^48, real world ….)
- Very hard to compute a good value function for each one of them, most you will never visit
- Thus, condition on the current state you are in, try to estimate the value function of the relevant part of the state space online
- Focus your resources on sub-MDP starting from now, often dramatically easier than solving the whole MDP

Any problems with online tree search?

Curse of dimensionality

- The sub-MDP rooted at the current state the agent is in may still be very large (too many states are reachable), despite much smaller than the original one.
- Too many actions possible: large tree branching factor
- Too many steps: large tree depth

I cannot exhaustively search the full tree

Curse of dimensionality

Consider hex on an NxN board.

branching factor $\leq N^2$

$2N \leq \text{depth} \leq N^2$

board size	max branching factor	min depth	tree size	depth of 10^{10} nodes
6x6	36	12	$>10^{17}$	7
8x8	64	16	$>10^{28}$	6
11x11	121	22	$>10^{44}$	5
19x19	361	38	$>10^{96}$	4

Goal of HEX: to make a connected line that links two antipodal points of the grid

How to handle curse of dimensionality?

Intelligent search instead of exhaustive search:

A. The depth of the search may be reduced by **position evaluation**: truncating the search tree at state s and replacing the subtree below s by an approximate value function v(s)=v*(s) that predicts the outcome from state s.

B. The breadth of the search may be reduced by sampling actions from a **policy** p(als) that is a probability distribution over possible moves a in position s, instead of trying every action.

Position evaluation

We can estimate values for positions in two ways:

- Engineering them using human experts (DeepBlue)
- Learning them from self-play (TD-gammon)

Problems with human engineering:

- tiring
- non transferrable to other domains.

YET: that's how Kasparov was first beaten.

http://stanford.edu/~cpiech/cs221/apps/deepBlue.html

Position evaluation using sampled rollouts

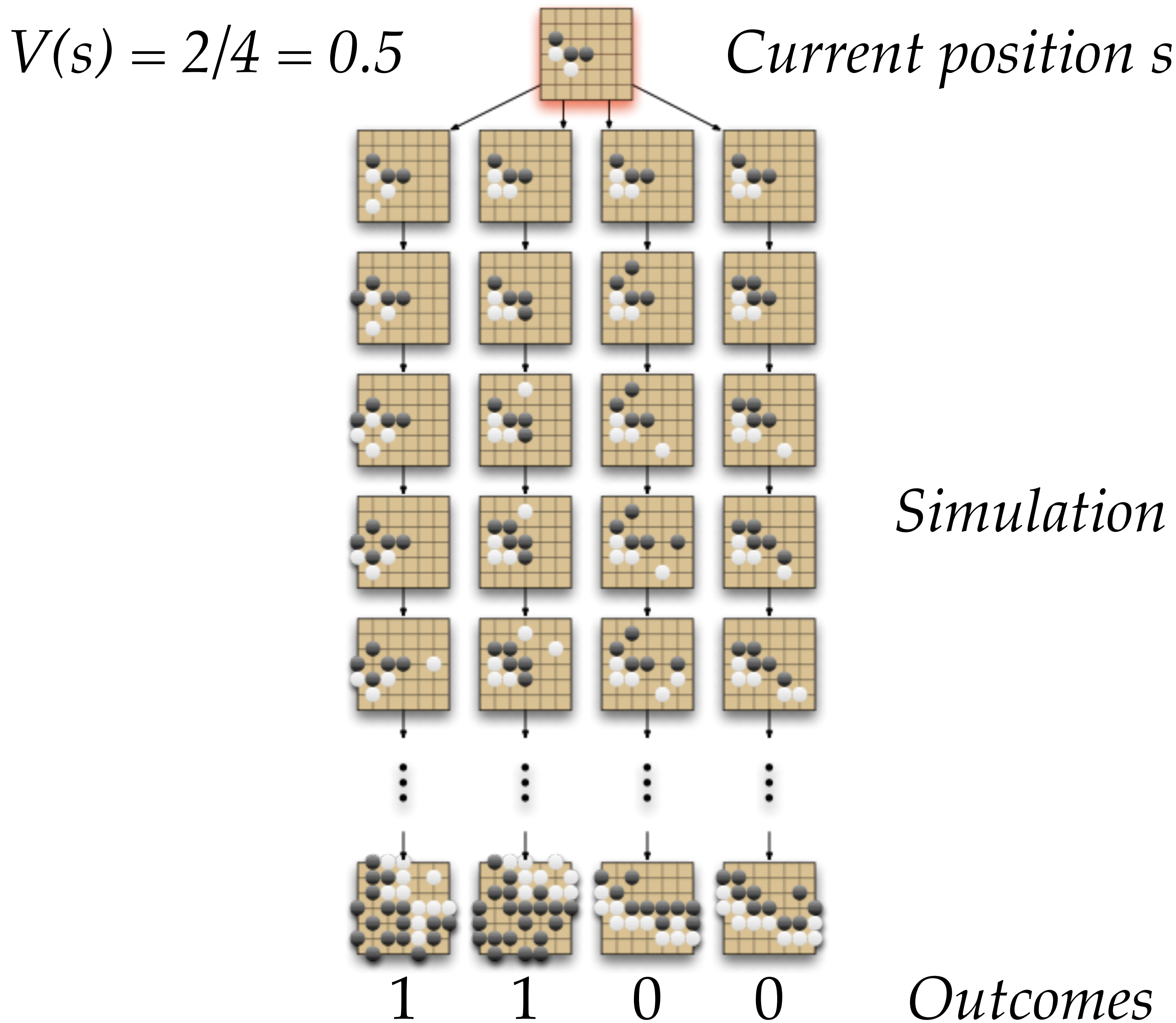
a.k.a. Monte Carlo

```
function MC_BoardEval(state):
    wins = 0
    losses = 0
    for i=1:NUM_SAMPLES
        next_state = state
        while non_terminal(next_state):
            next_state = random_legal_move(next_state)
        if next_state.winner == state.turn: wins++
        else: losses++ #needs slight modification if draws possible
    return (wins - losses) / (wins + losses)
```

What policy shall we use to draw our simulations?

The cheapest one is random..

Monte-Carlo Position Evaluation in Go



Simplest Monte-Carlo Search

- Given a model $\mathcal{M}_\nu$, a root state s_t, and a most of the times random policy π
- For each action $a \in \mathcal{A}$
  - Q(s_t,a) = MC_boardEval(s'), s'=T(s,a)
- Select root action:

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

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:

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

  - Evaluate action value function **<u>of the root</u>** 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 improve our simulation policy the more simulations we obtain?
- Yes we can. We can keep track of action values Q not only for the root but also for nodes internal to a tree we are expanding!

In MCTS the simulation policy improves

- How should we select the actions inside the tree?

- This doesn't work: $$a_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \; Q(s_t, a)$$

Why?

K-armed Bandit Problem

You are faced repeatedly with a choice among k different options, or actions. After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected. Your objective is to maximize the expected total reward over some time period, for example, over 1000 action selections, or time steps.

Each action has an expected reward:

$$q_*(a) = \mathbb{E}[R_t \,|\, A_t = a]$$

If we knew what it was, we would always pick the action with the highest expected reward, obviously. Those q values is exactly what we care to estimate.

Note that the state is not changing…

that is a big difference than what we have seen so far…

K-armed Bandit Problem

You are faced repeatedly with a choice among k different options, or actions. After each choice you receive a numerical reward chosen from a stationary probability distribution that depends on the action you selected. Your objective is to maximize the expected total reward over some time period, for example, over 1000 action selections, or time steps.

Let $Q_t(a)$ denote our estimates of q* at time t.

There are two things we can do each time-step:

- Exploit: Pick the action with the highest
- Explore: Pick a different action

$\epsilon - \mathrm{greedy}(Q)$ is a simple policy that balances in some way exploitation/exploration. However, it does not differentiate between suboptimal, clearly suboptimal or marginally suboptimal actions, or actions that have been tried often or not, and thus have unreliable Q values.

Upper-Confidence Bound

Sample actions according to the following score:

$$\underbrace{v_i}_{\text{value estimate}} + \underbrace{C}_{\text{tunable parameter}} \times \sqrt{\frac{\ln(\underbrace{N}_{\text{parent node visits}})}{\underbrace{n_i}_{\text{number of visits}}}}$$

- score is decreasing in the number of visits (explore)
- score is increasing in a node's value (exploit)
- always tries every option once

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

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

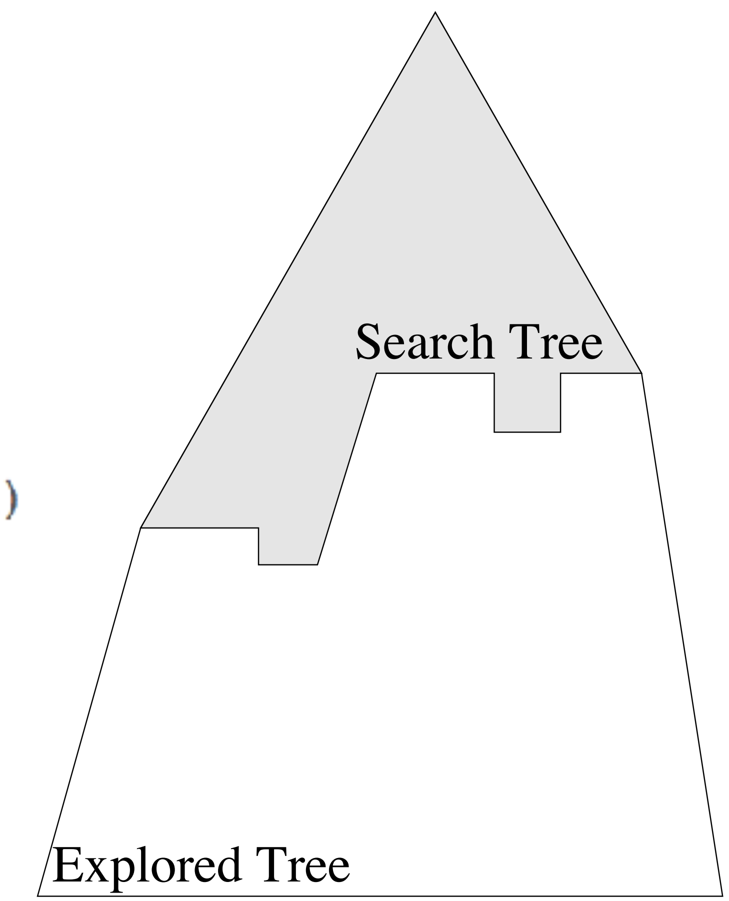
```
function expand(state):
    state.visits = 1
    state.value = 0


function update value(state, winner):

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

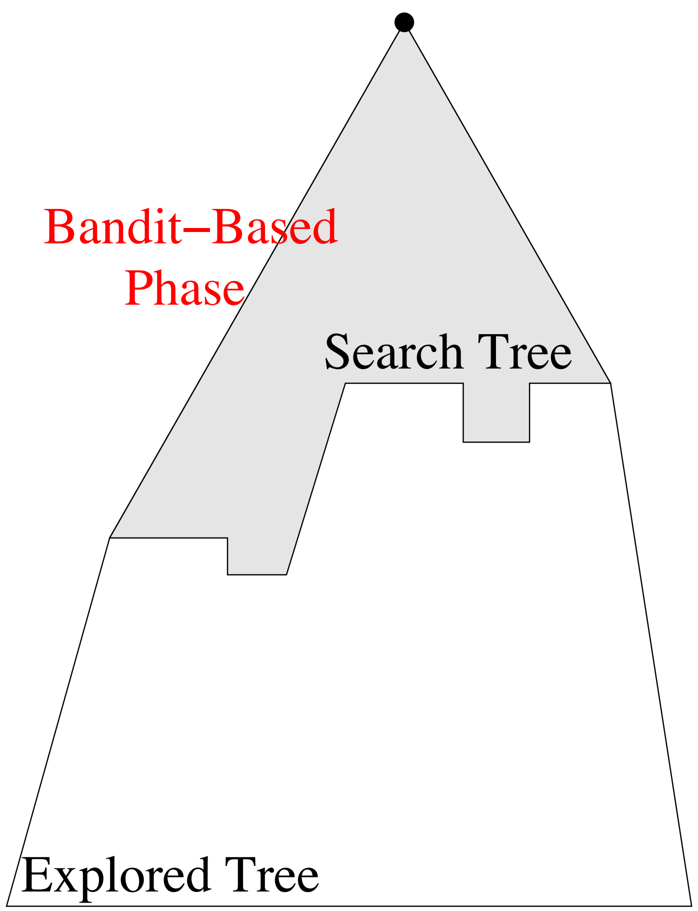
Basic MCTS pseudocode

```
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)
```



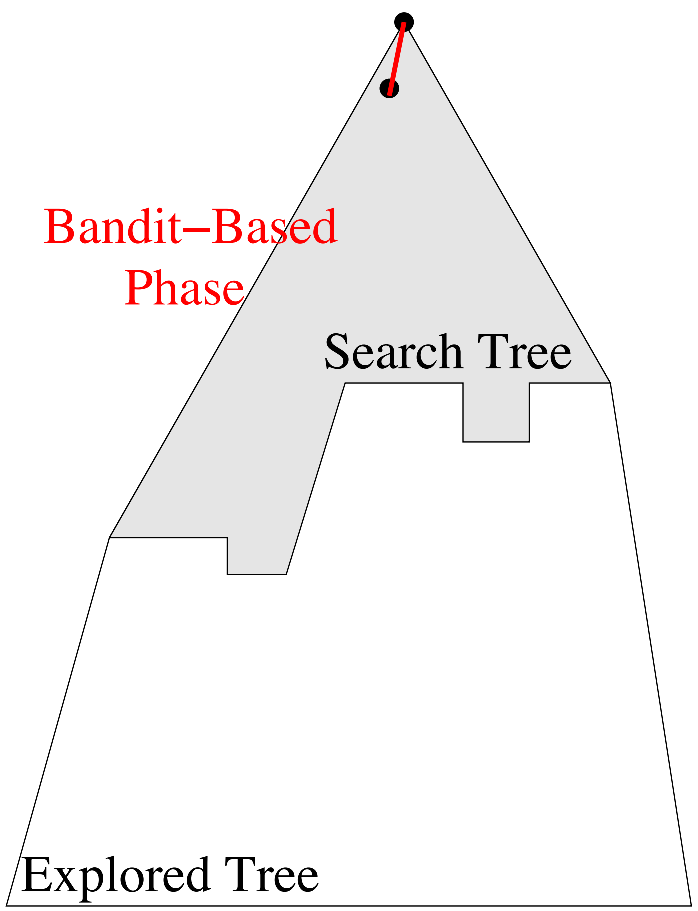
Basic MCTS pseudocode

```
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)
```



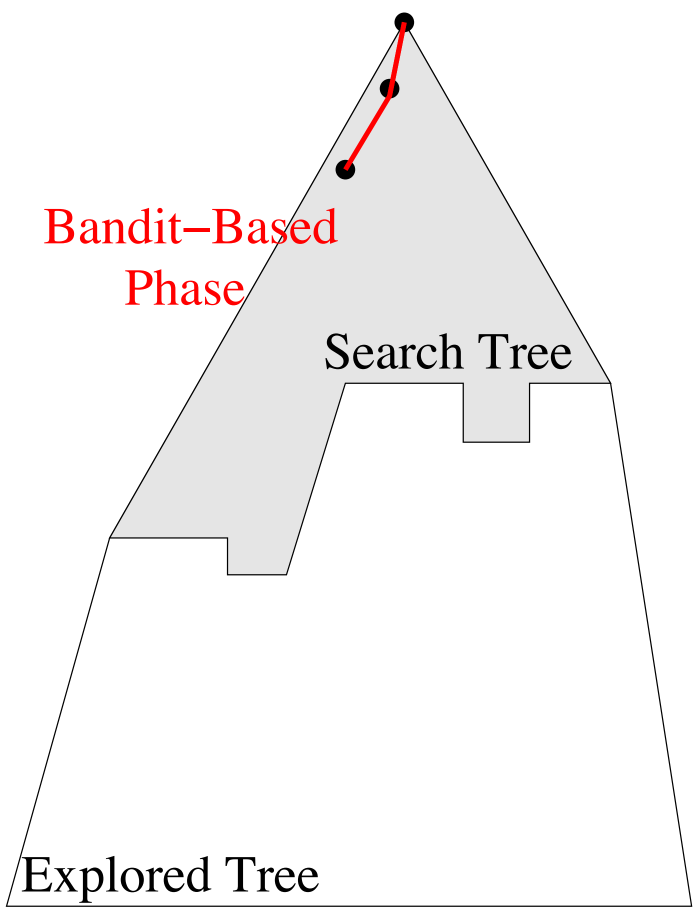
Basic MCTS pseudocode

```
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)
```



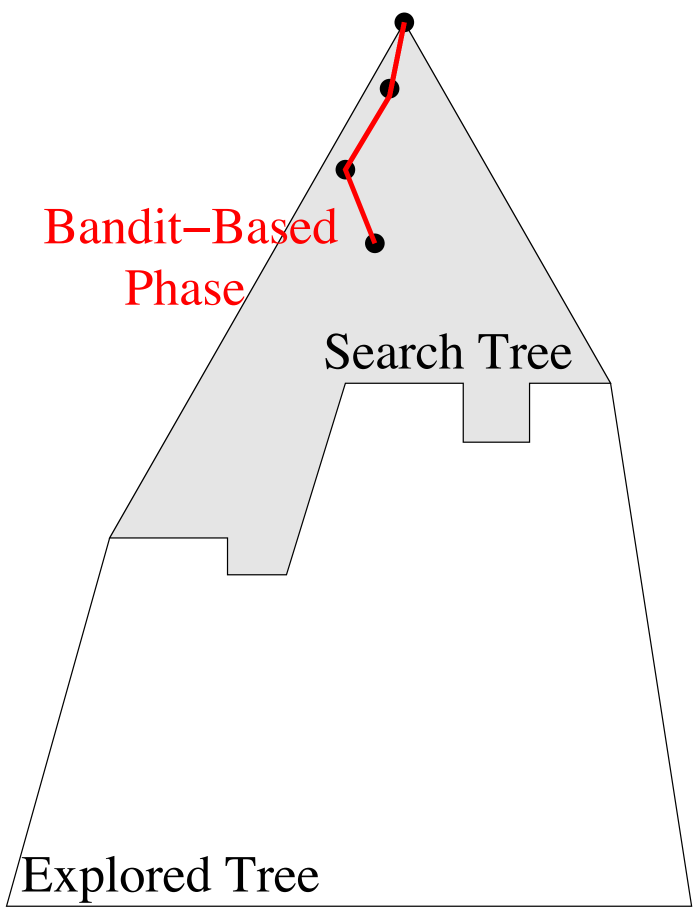
Basic MCTS pseudocode

```
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)
```



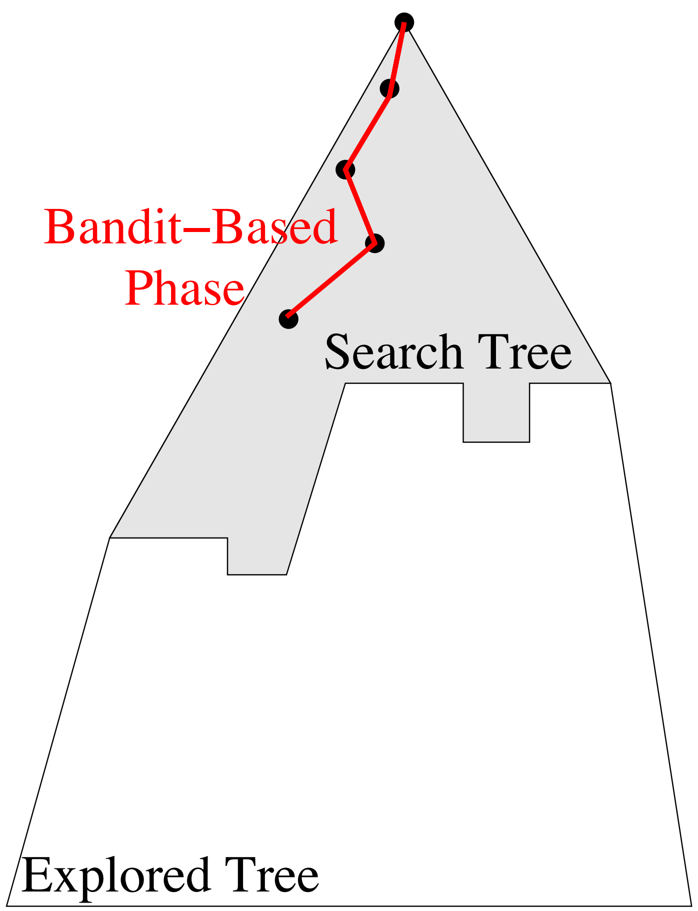
Basic MCTS pseudocode

```
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)
```



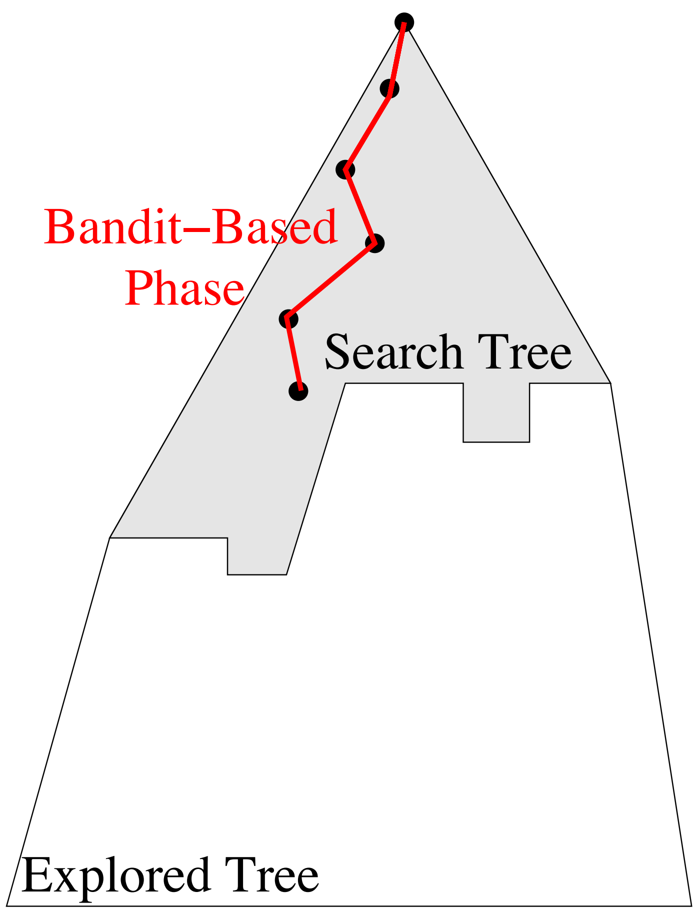
Basic MCTS pseudocode

```
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)
```



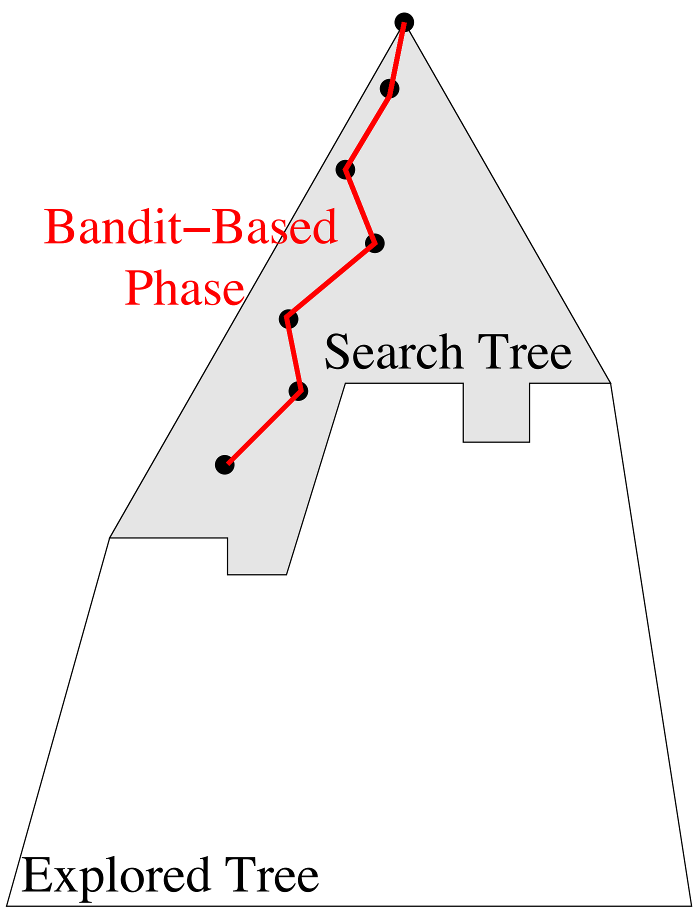
Basic MCTS pseudocode

```
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)
```



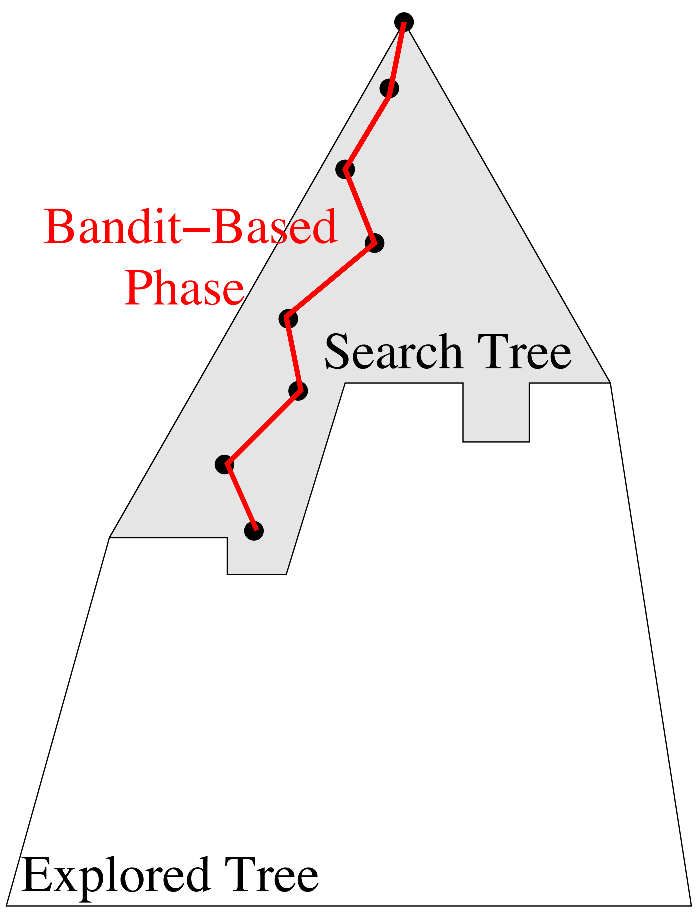
Basic MCTS pseudocode

```
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)
```



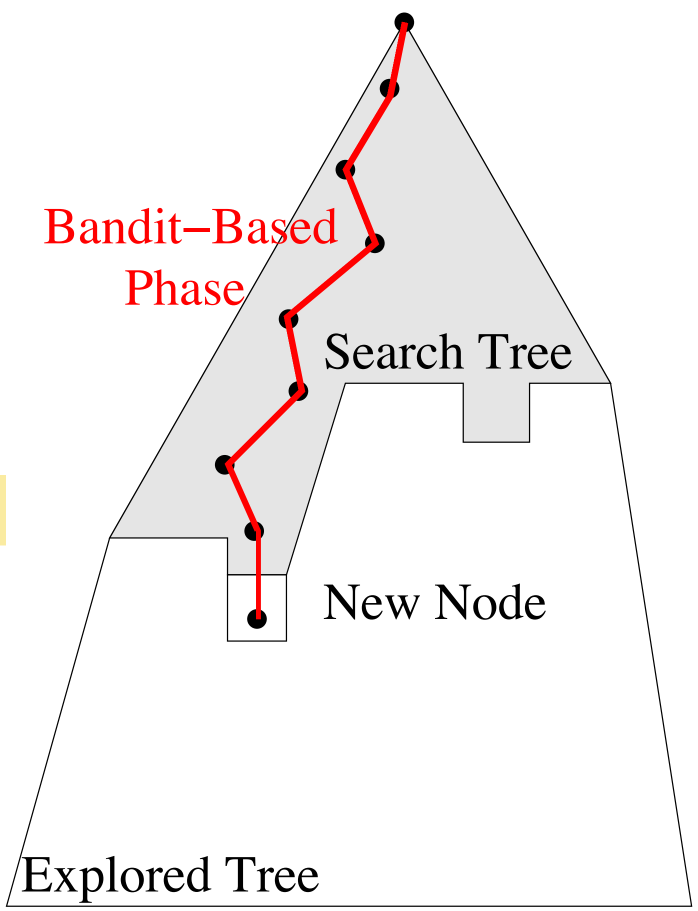
Basic MCTS pseudocode

```
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

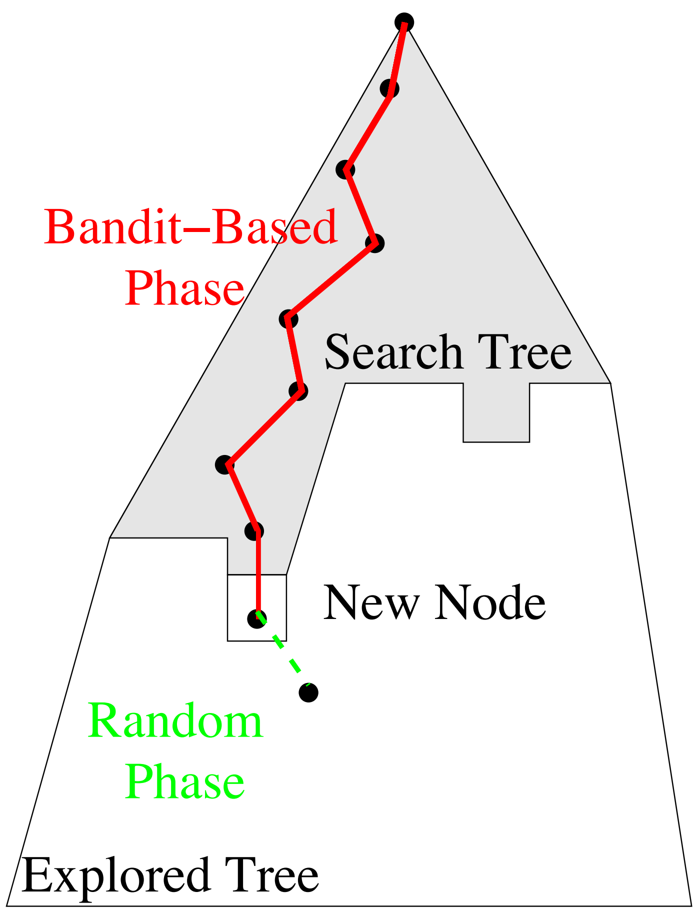
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
     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



Basic MCTS pseudocode

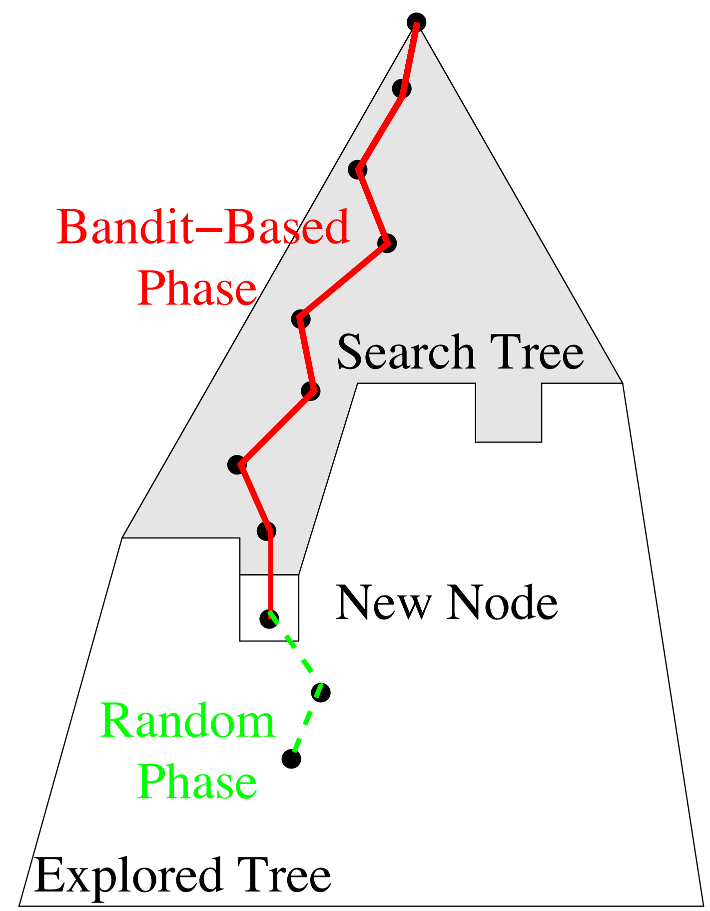
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
     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

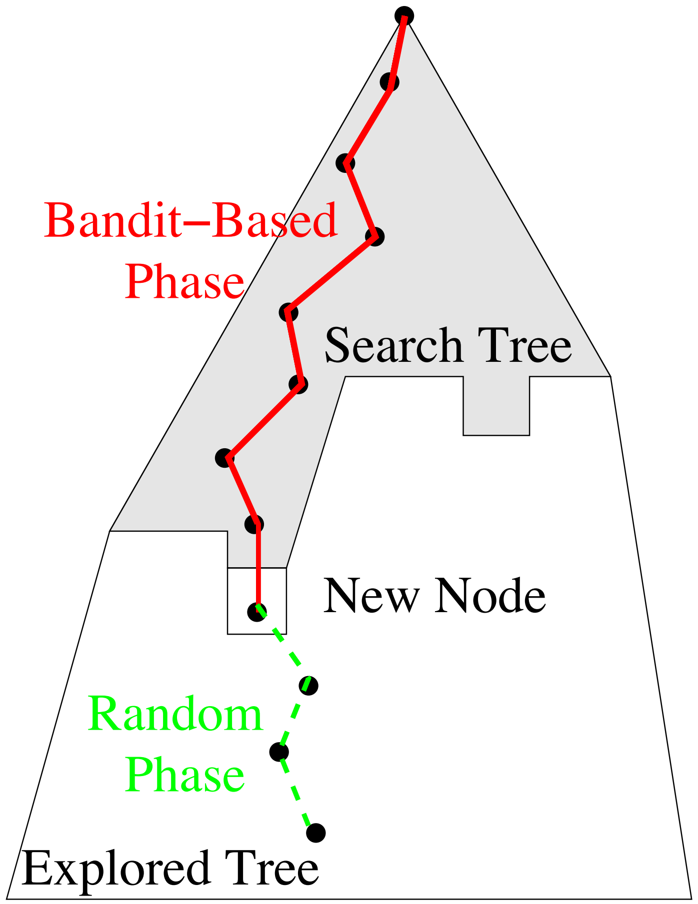
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
     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

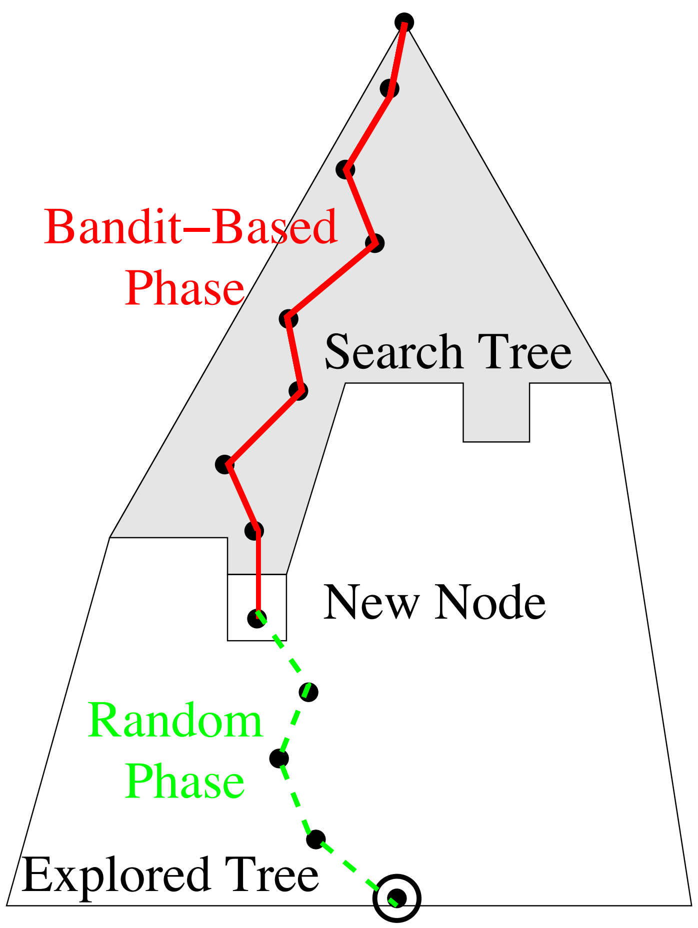



```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

Basic MCTS pseudocode

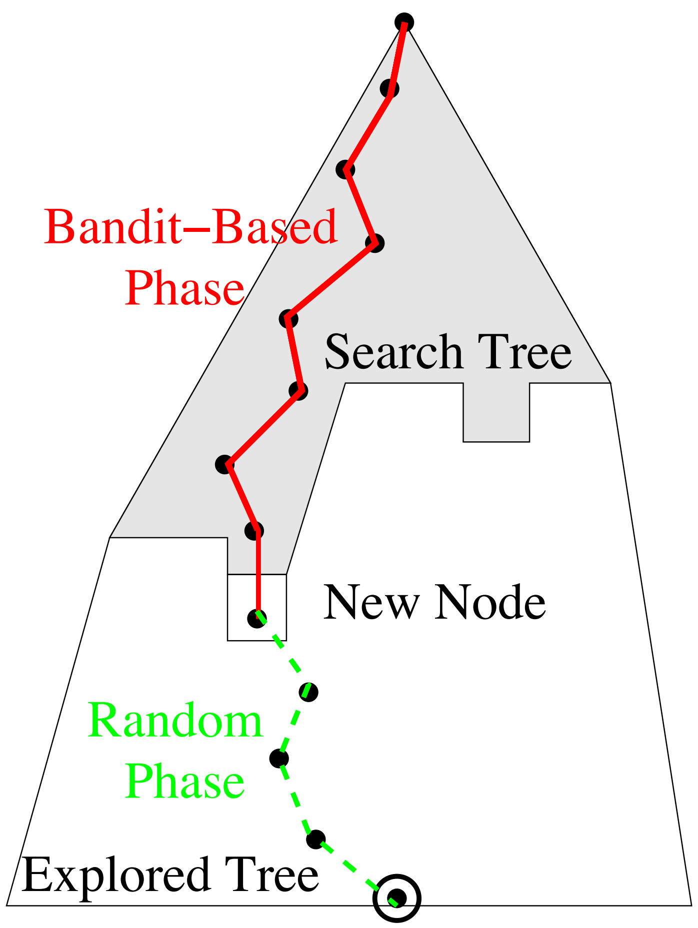
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
     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

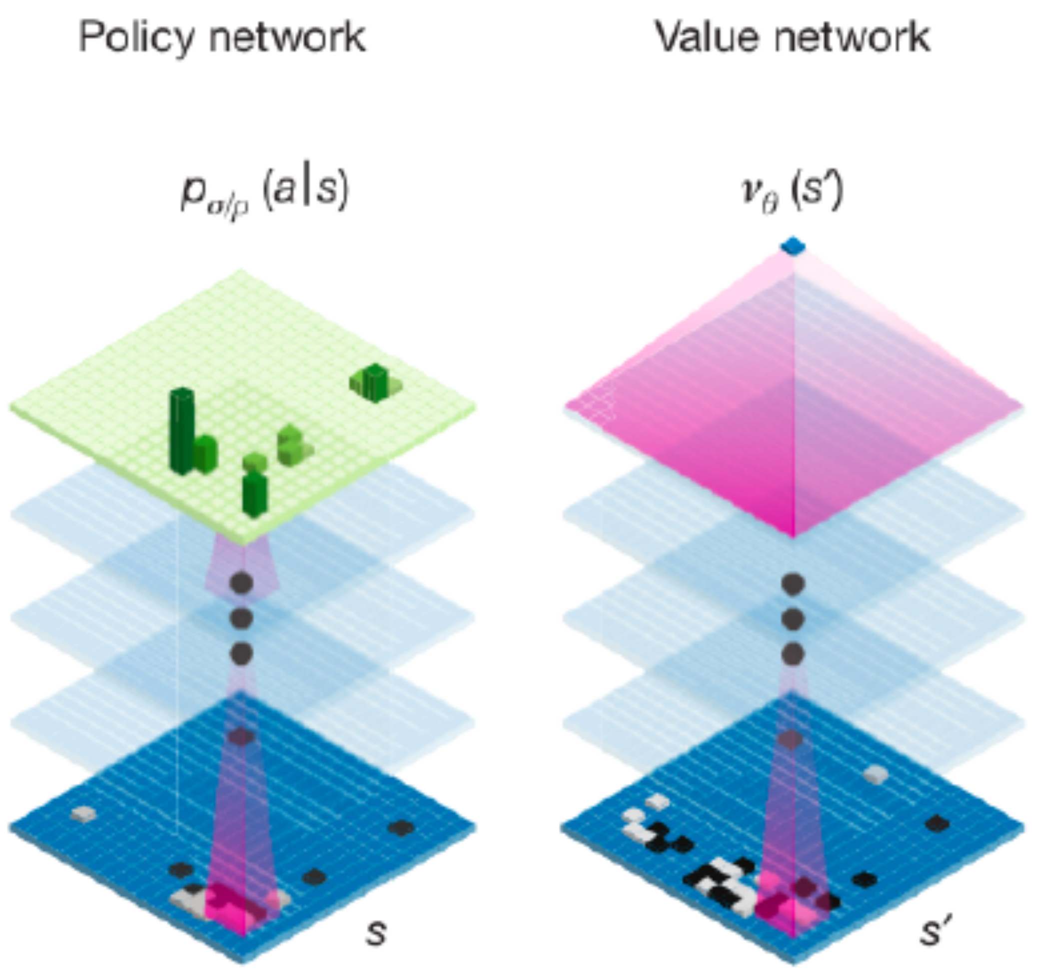


Can we do better?

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

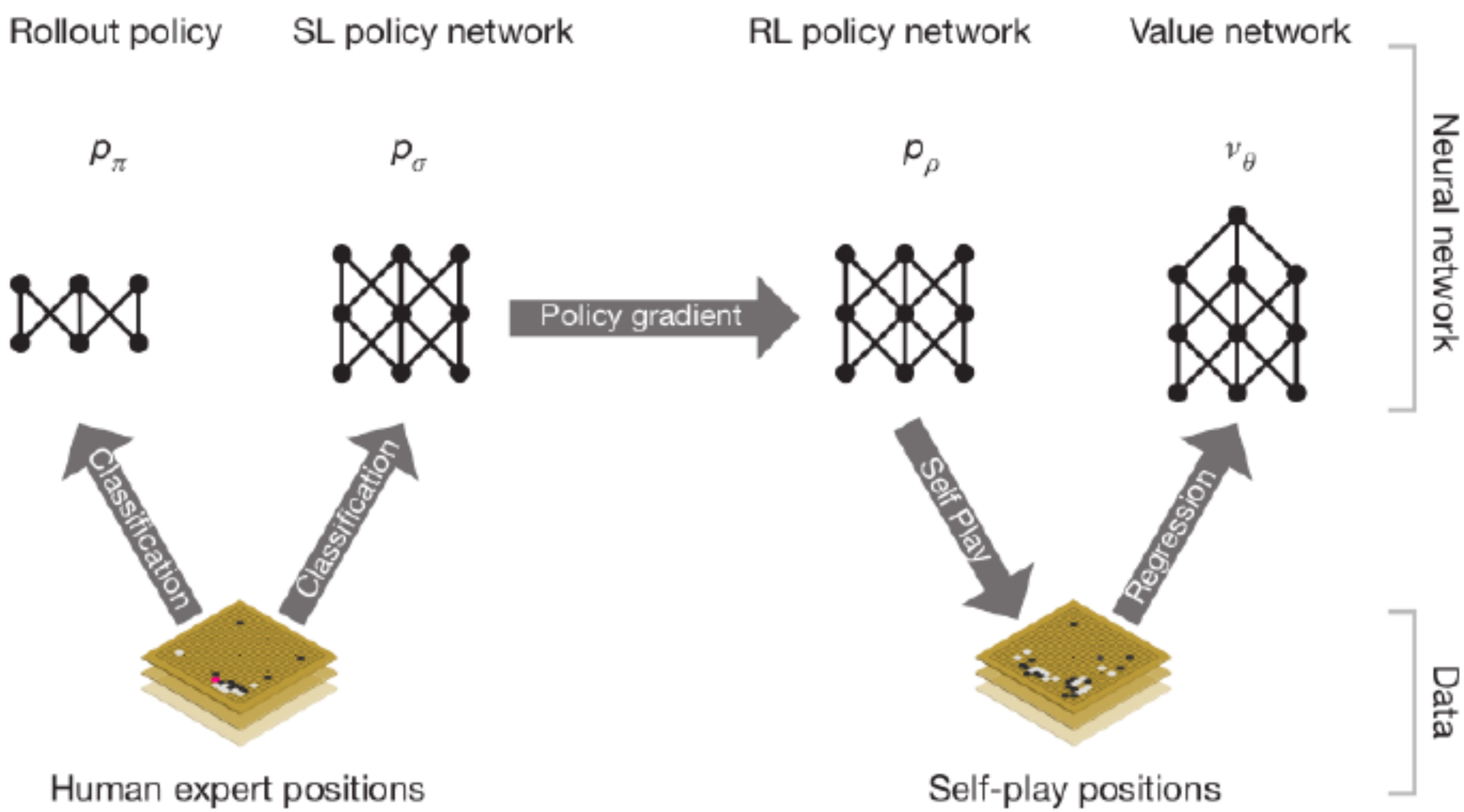
AlphaGo: Learning-guided MCTS

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



AlphaGo: Learning-guided search

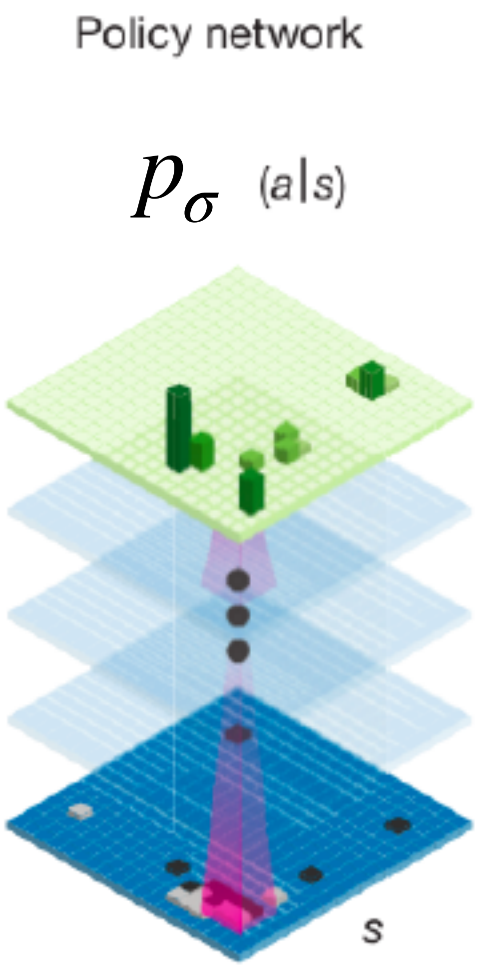
1. Train two action policies, one cheap (rollout) policy p_π and one expensive policy p_σ by mimicking expert moves (standard supervised learning).
2. Then, train a new policy p_ρ with RL and self-play p_σ initialized from SL policy.
3. Train a value network that predicts the winner of games played by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

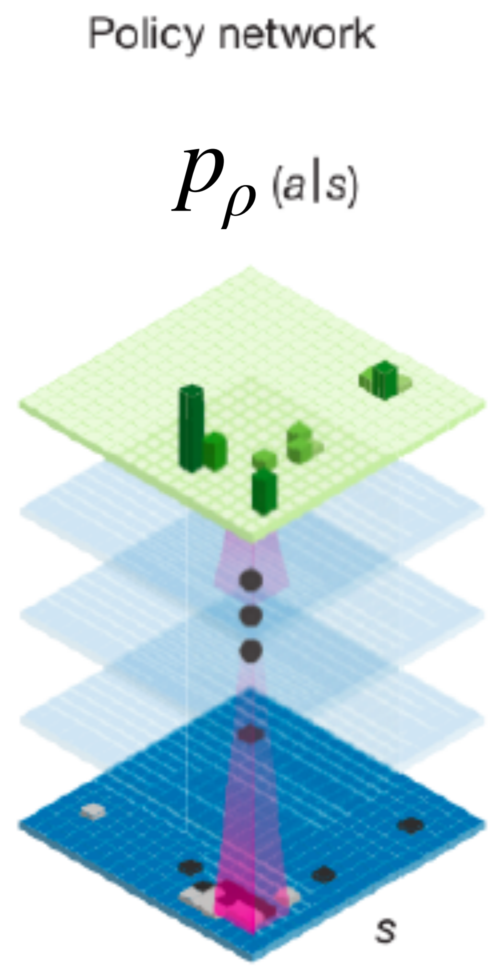


Reinforcement learning of policy networks

- 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

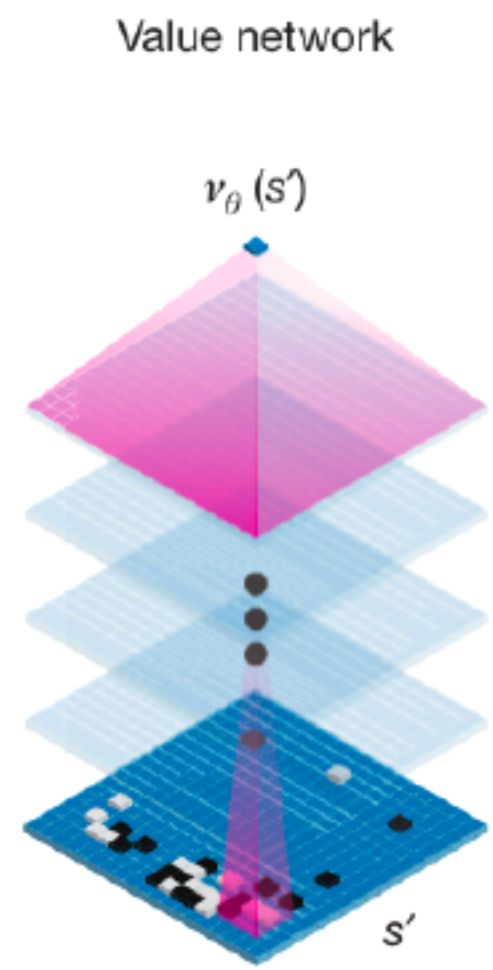
$$\Delta\rho \propto \frac{\partial \log p_\rho(a_t \mid s_t)}{\partial \rho} z_t$$



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

Reinforcement learning of value networks

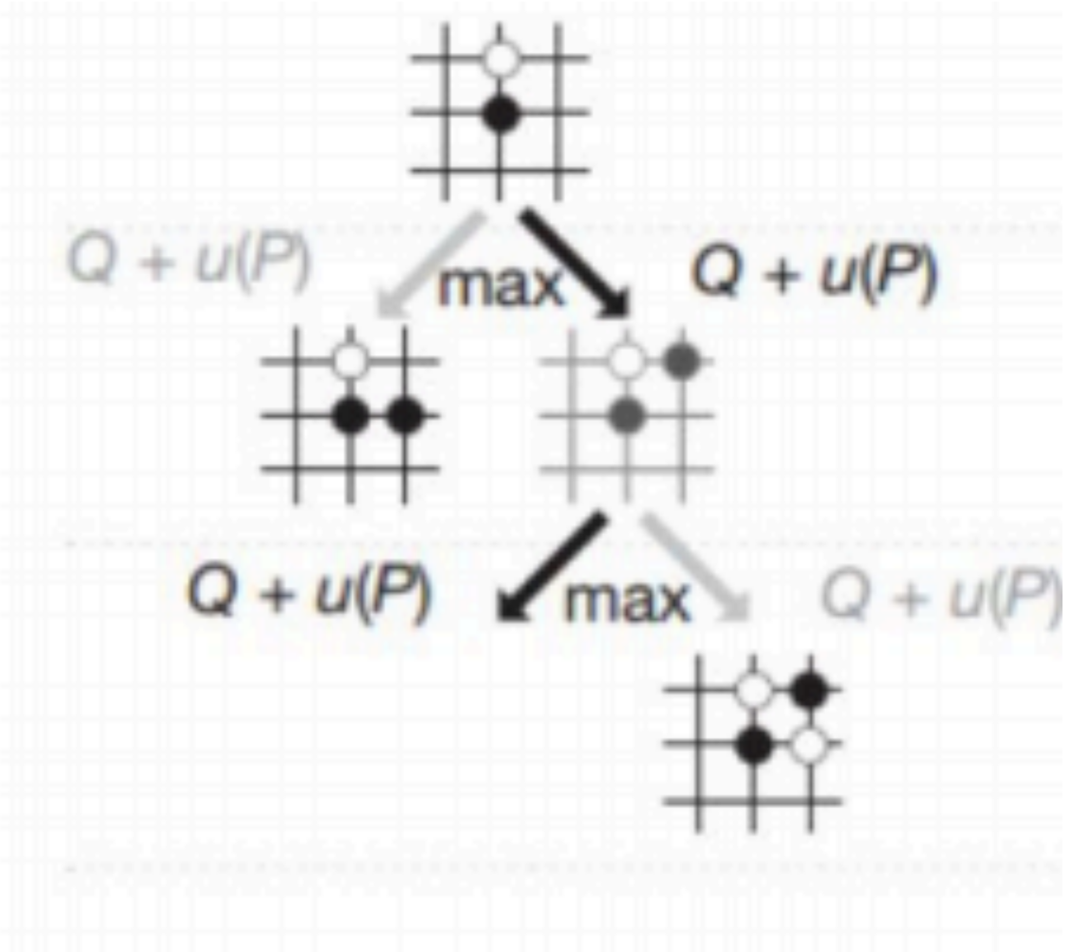
- Objective: Estimating a value function $v_p(s)$ that predicts the outcome from position s of games **played by using RL policy p for both players** (in contrast to min-max search)
- Input: Sampled states during self-play, 30 million distinct positions, each sampled from a separate game, played by the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

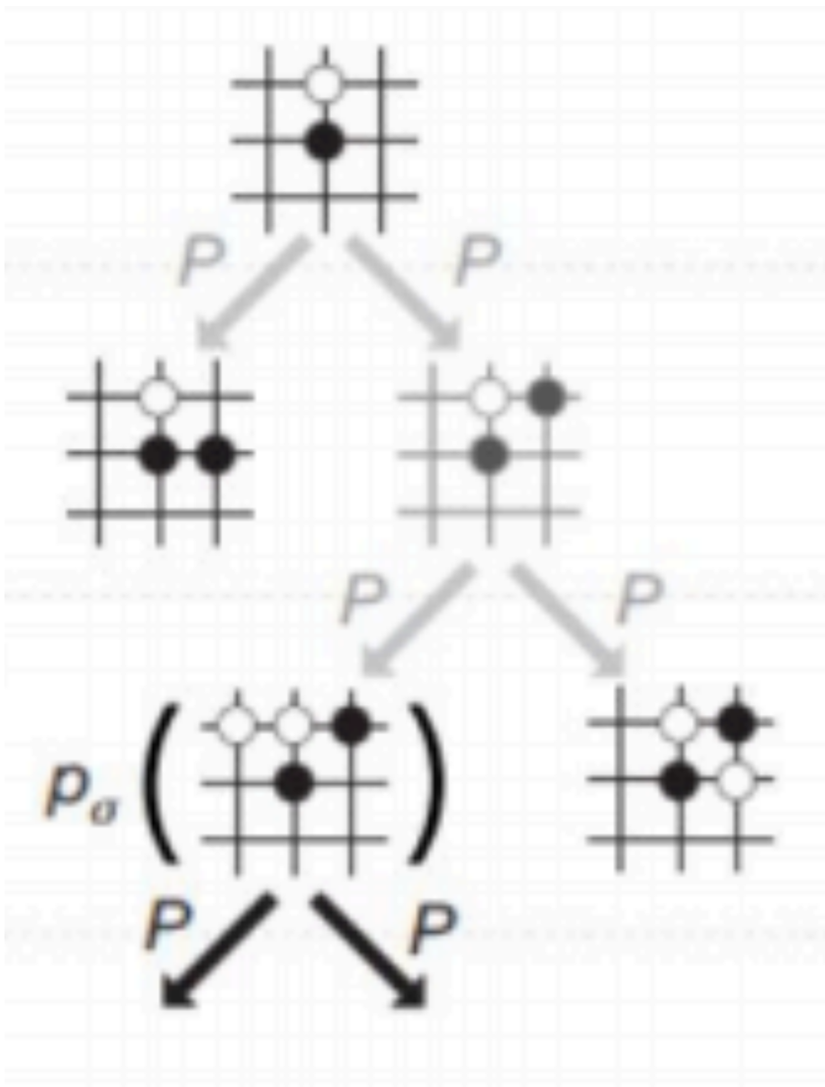
$$a_t = \text{argmax}_a(Q(s_t, a) + u(s_t, a))$$

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

- a_t - action selected at time step t from board s.
- $Q(s_t, a)$ - 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

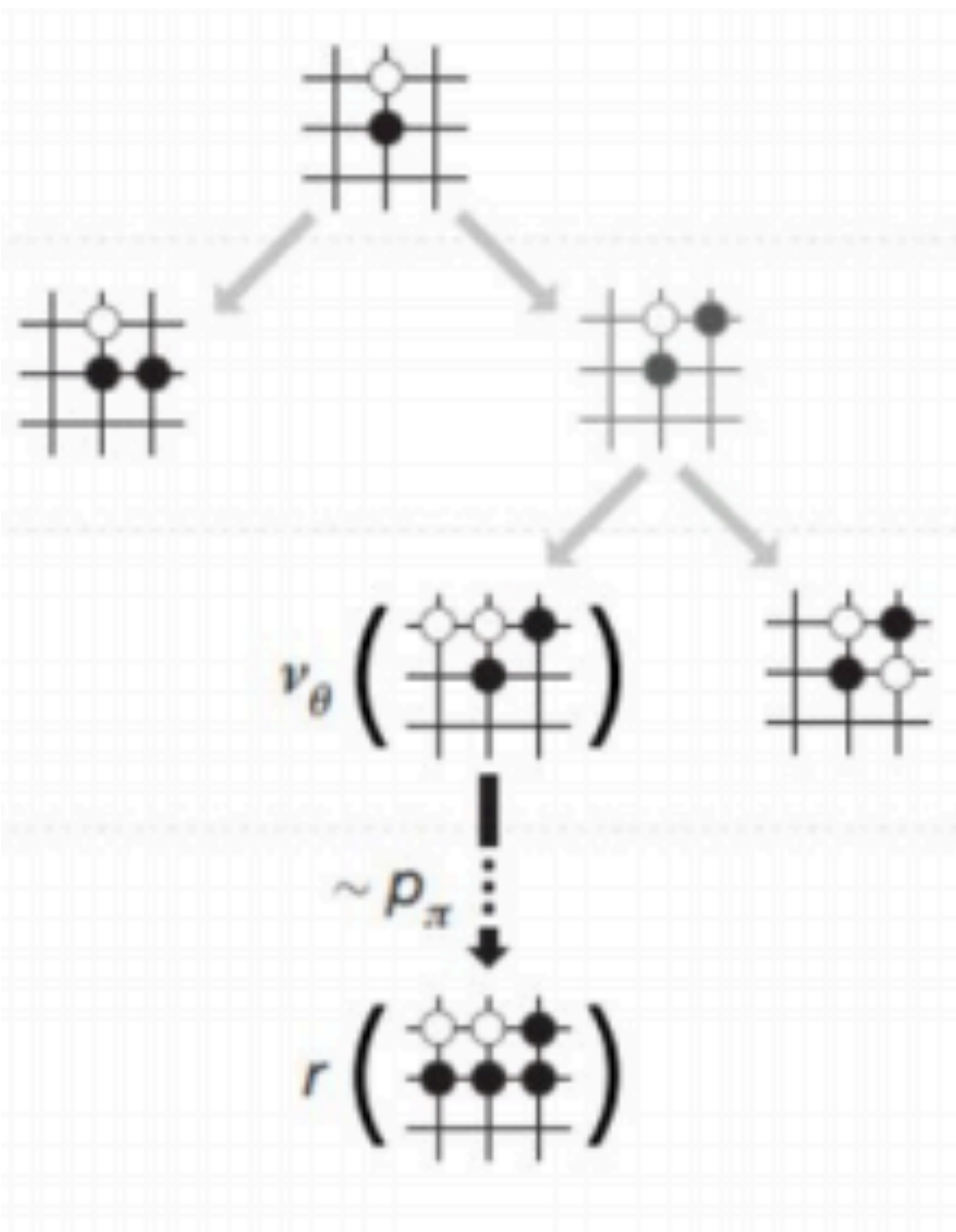
Expansion: when reaching a leaf, play the action with highest score from p_σ



- 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



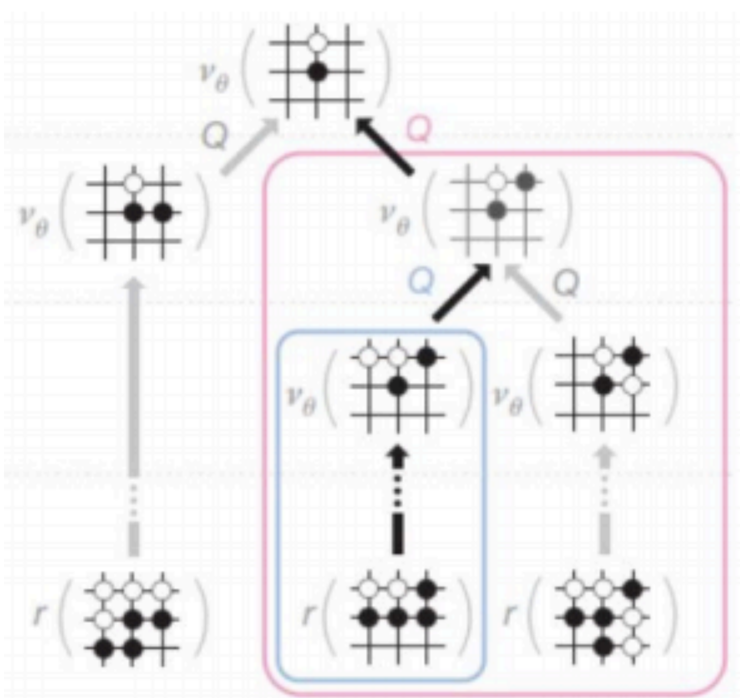
- From the selected leaf node, run multiple simulations in parallel using the rollout policy
- Evaluate the leaf node as:

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

- 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

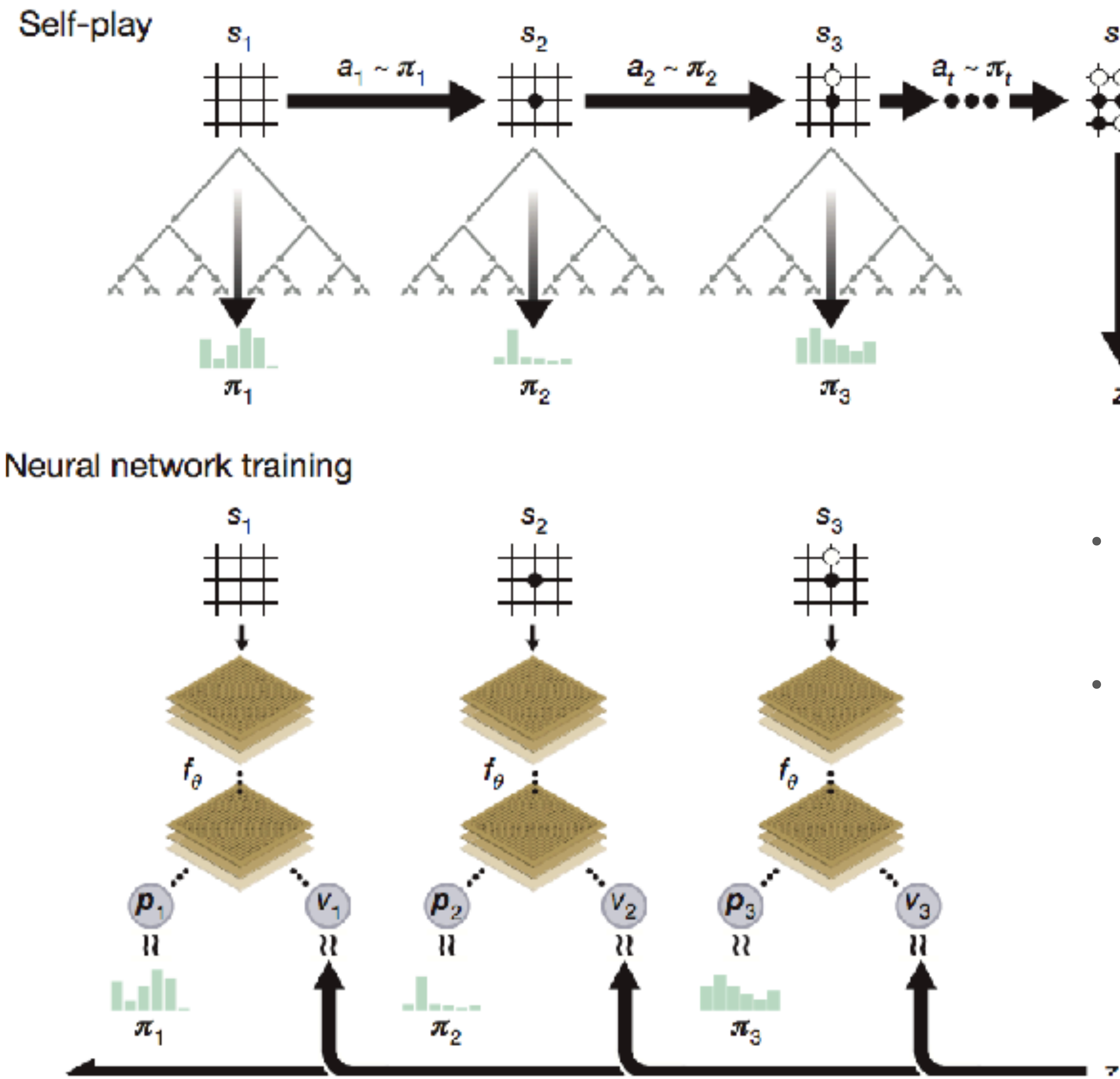
AlphaGoZero: Lookahead search during training!

- 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 need for human supervision.

AlphaGoZero: Lookahead search during training!

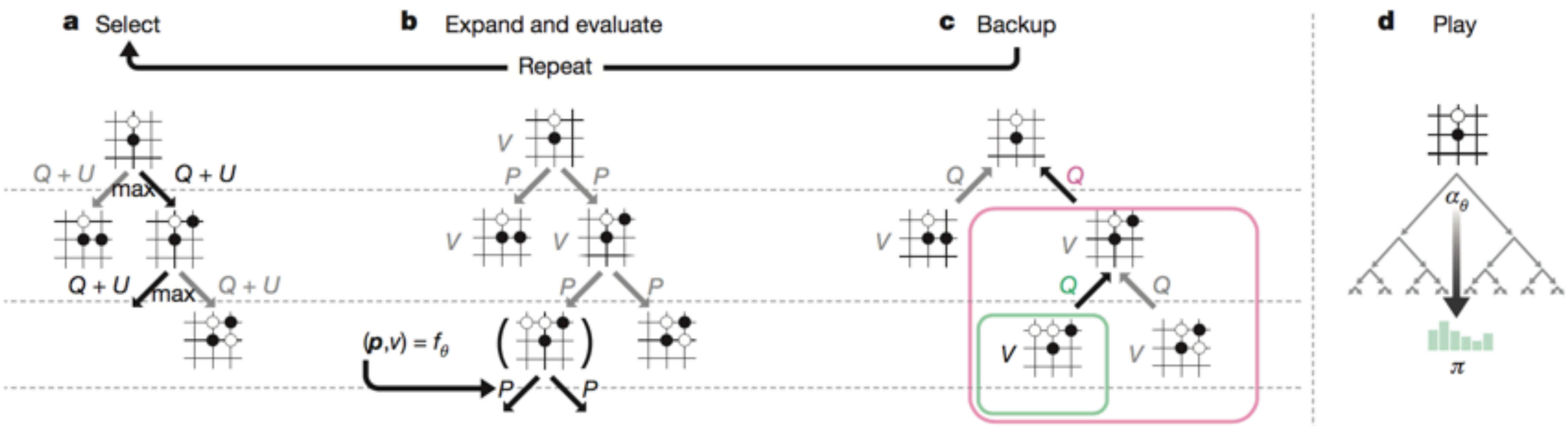
- Given any policy, a MCTS guided by this policy will produce an improved policy (policy improvement operator)
- Train to mimic such improved policy

MCTS as policy improvement operator



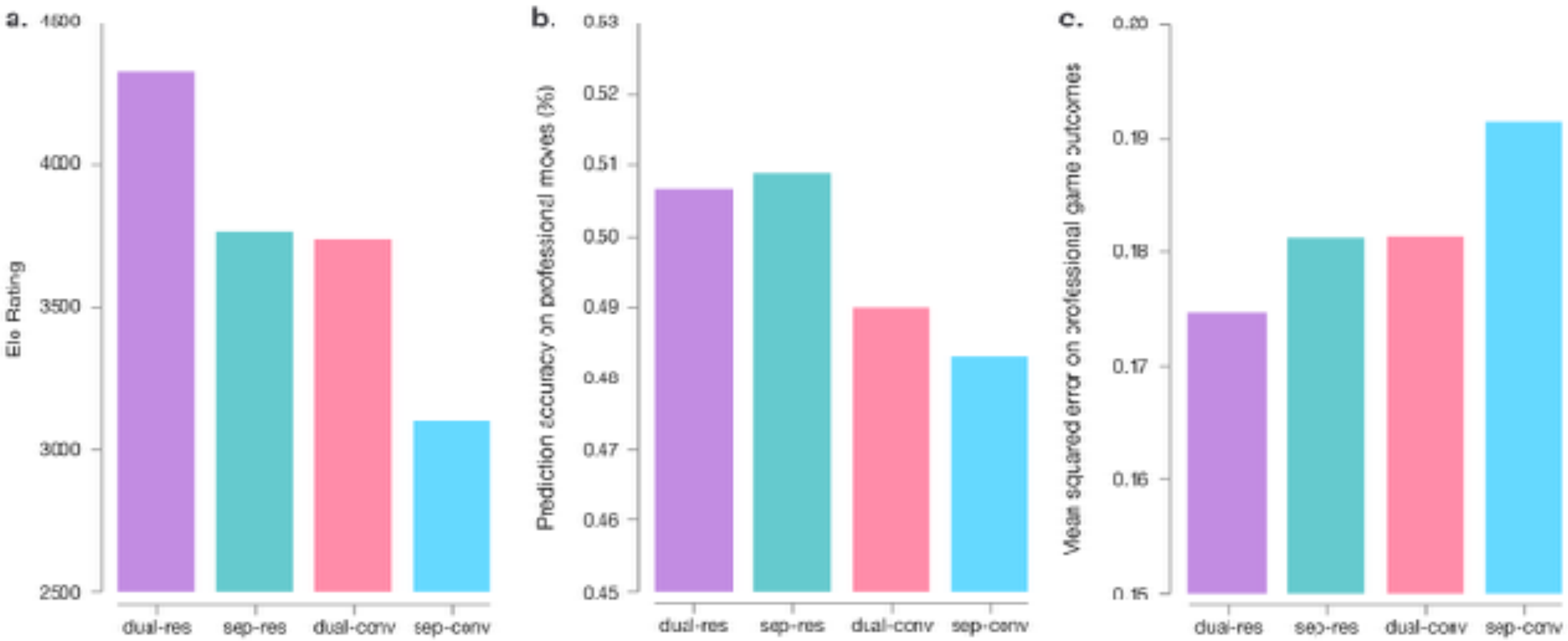
- Train so that the policy network mimics this improved policy
- Train so that the position evaluation network output matches the outcome (same as in AlphaGo)

MCTS: no MC rollouts till termination

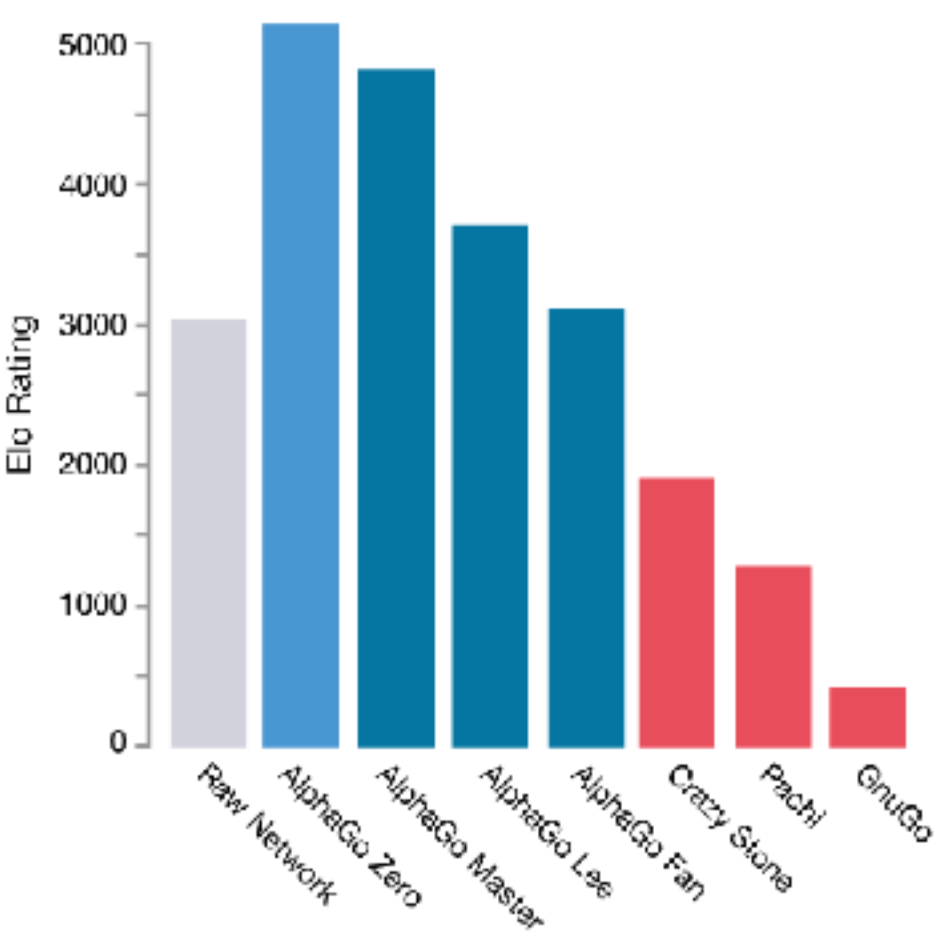


MCTS: using always value net evaluations of leaf nodes, no rollouts!

Architectures

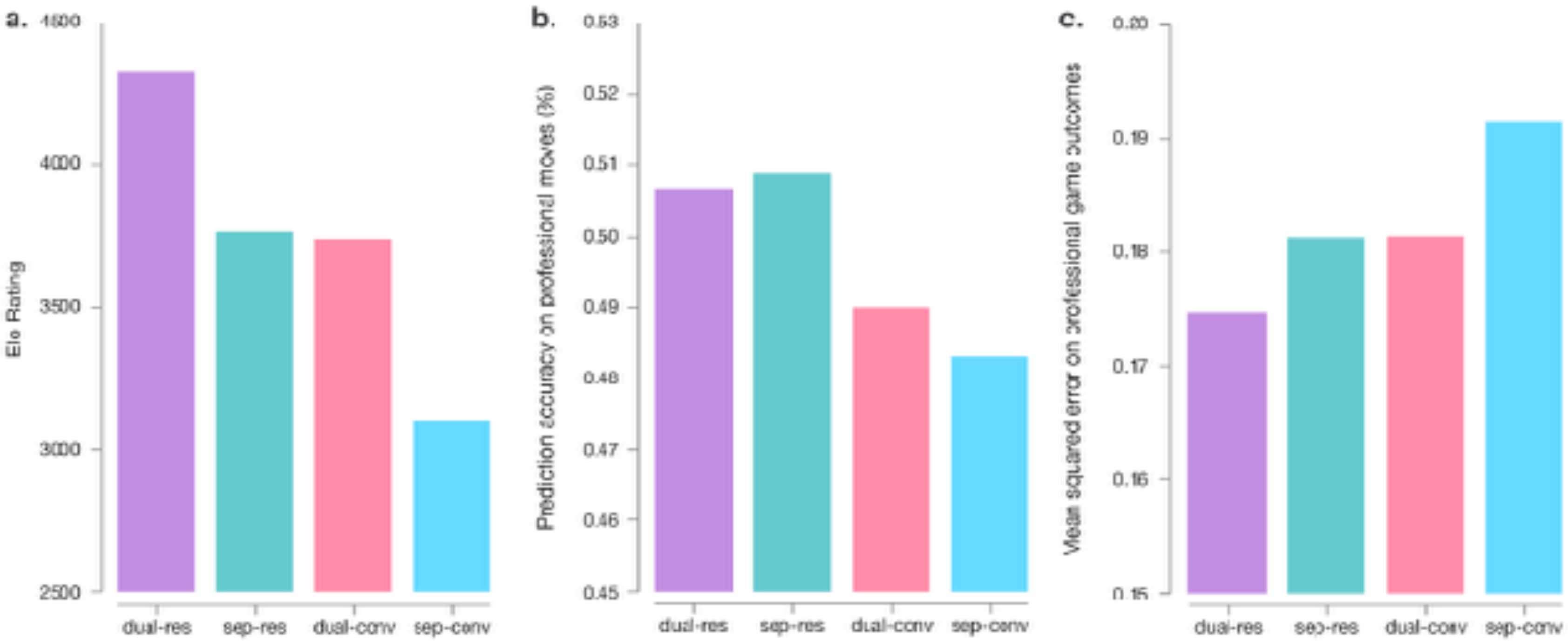


- Resnets help
- Jointly training the policy and value function using the same main feature extractor helps



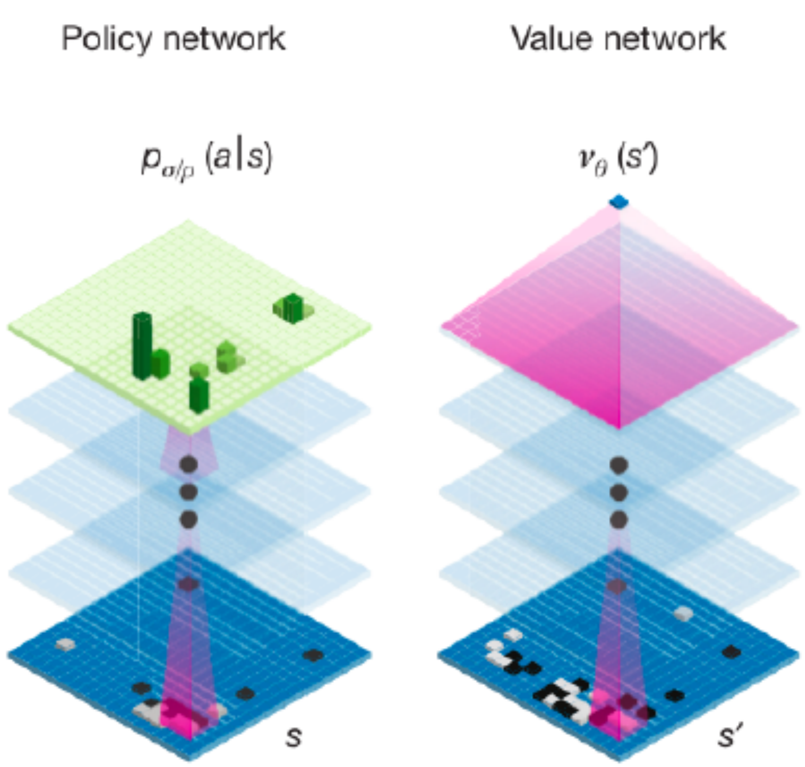
- Lookahead tremendously improves the basic policy

Architectures

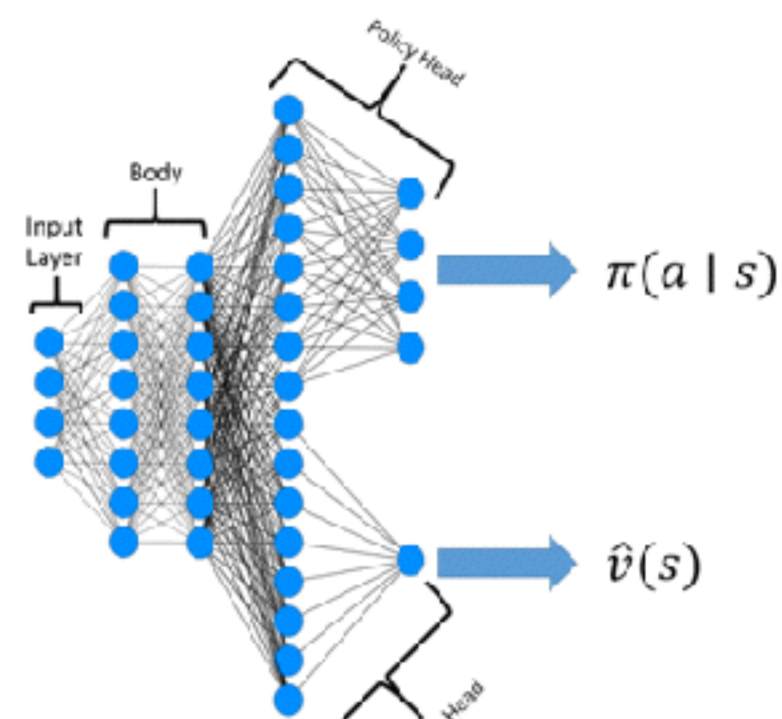


- Resnets help
- Jointly training the policy and value function using the same main feature extractor helps

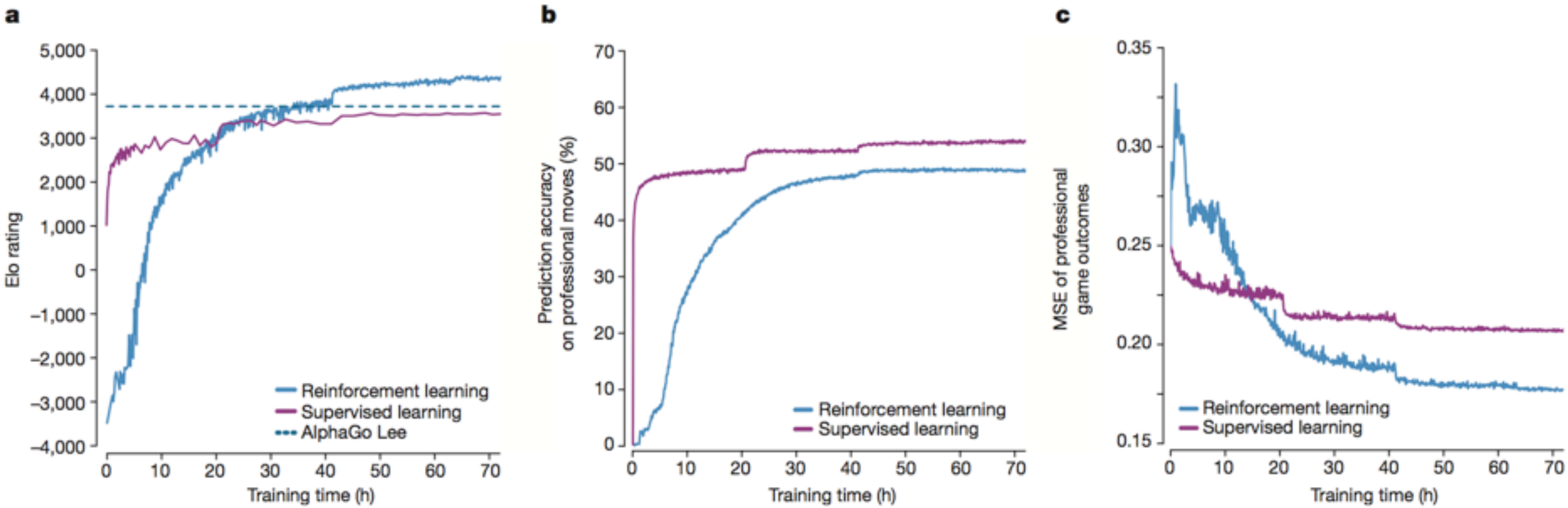
Separate policy/value nets



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