Carnegie Mellon

School of Computer Science

Deep Reinforcement Learning and Control

Monte Carlo Tree Search with Prior Knowledge

Spring 2021, CMU 10-403

Katerina Fragkiadaki

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

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

Simplest Monte-Carlo Search

Given a deterministic transition function T, a root state s and a simulation policy π (potentially random)

Simulate K episodes from current (real) state:

$$\{s, a, R_1^k, S_1^k, A_1^k, R_2^k, S_2^k, A_2^k, \ldots, S_T^k\}_{k=1}^K \sim \mathrm{T}, \pi$$

Evaluate action value function of the root by mean return:

$$Q(s, a) = \frac{1}{K} \sum_{k=1}^{K} G_k \rightarrow q_\pi(s, a)$$

Select root action: $a = \operatorname{argmax}_{a \in \mathcal{A}} Q(s, a)$

Can we do better?

- Could we be improving our simulation policy the more simulations we obtain?
- Yes we can! We can have two policies:
    - 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 use to improve the simulation policy over time
    - External to the tree: we do not have Q estimates and thus we use a random policy

In MCTS, the simulation policy improves

- Can we think anything better than $\epsilon -$ greedy?

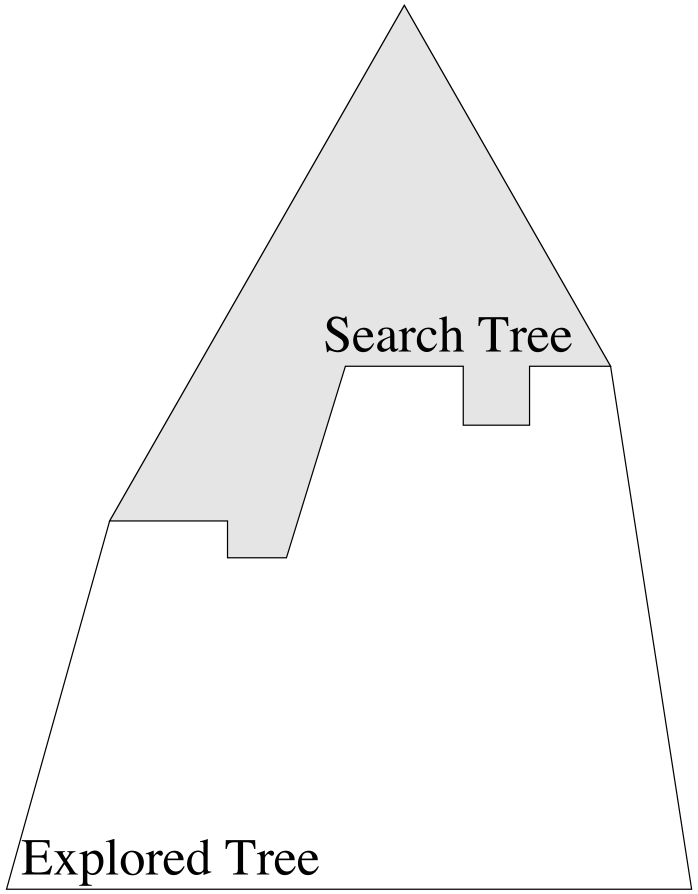
Monte-Carlo Tree Search

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. **Back-propagation**
   - 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*

Basic MCTS pseudocode

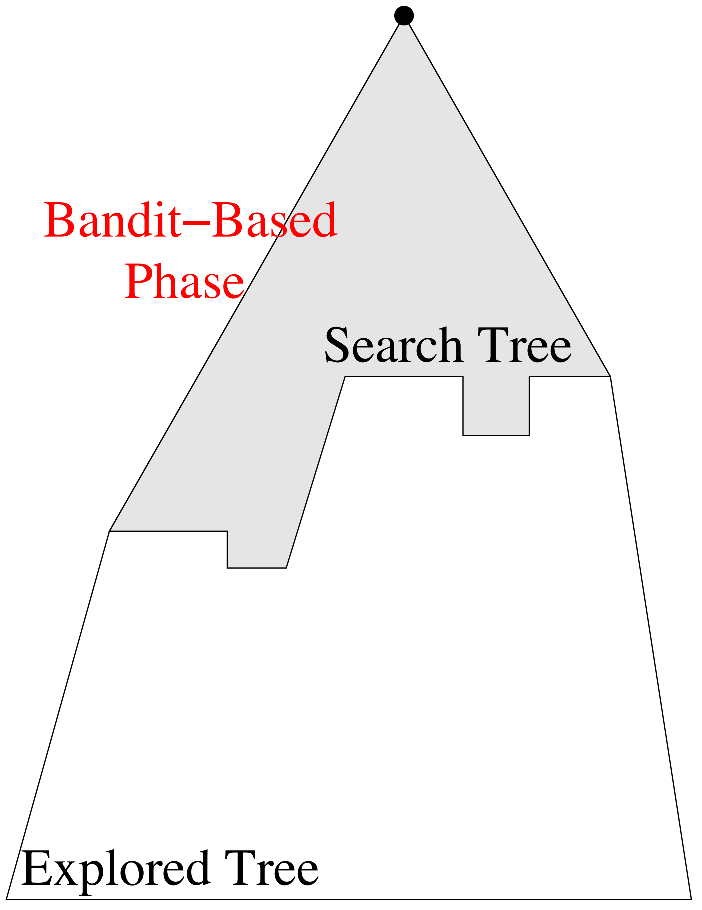
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
     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



Search tree contains states whose all children have been tried at least once

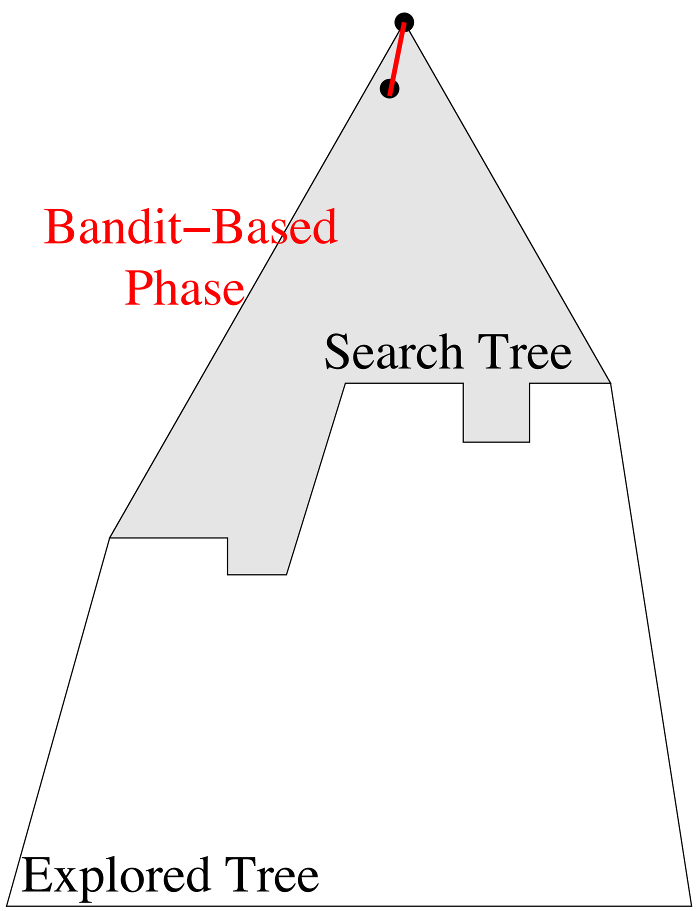
Basic MCTS pseudocode

```
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)
```



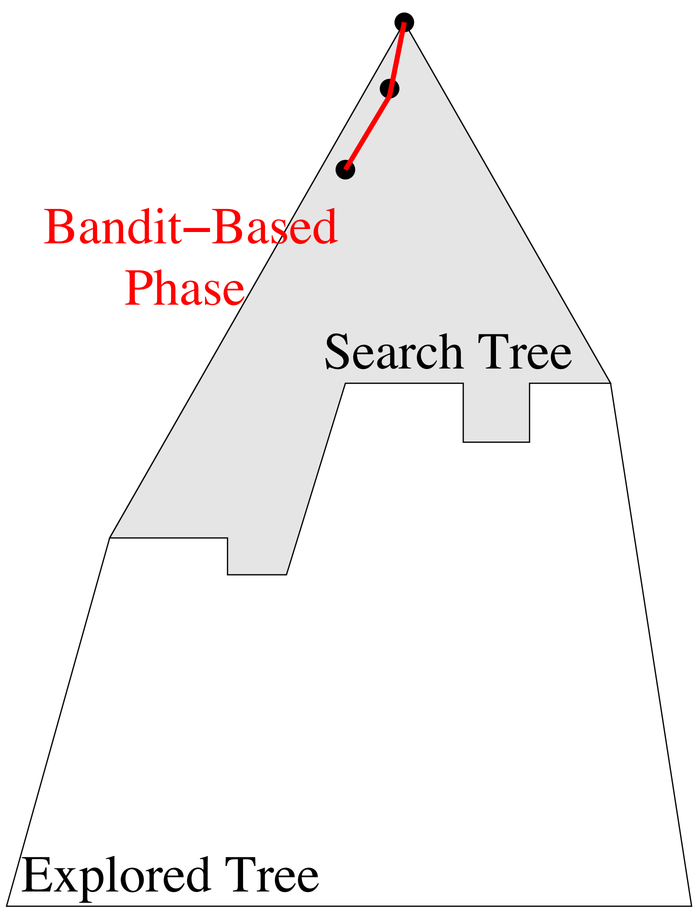
Basic MCTS pseudocode

```
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)
```



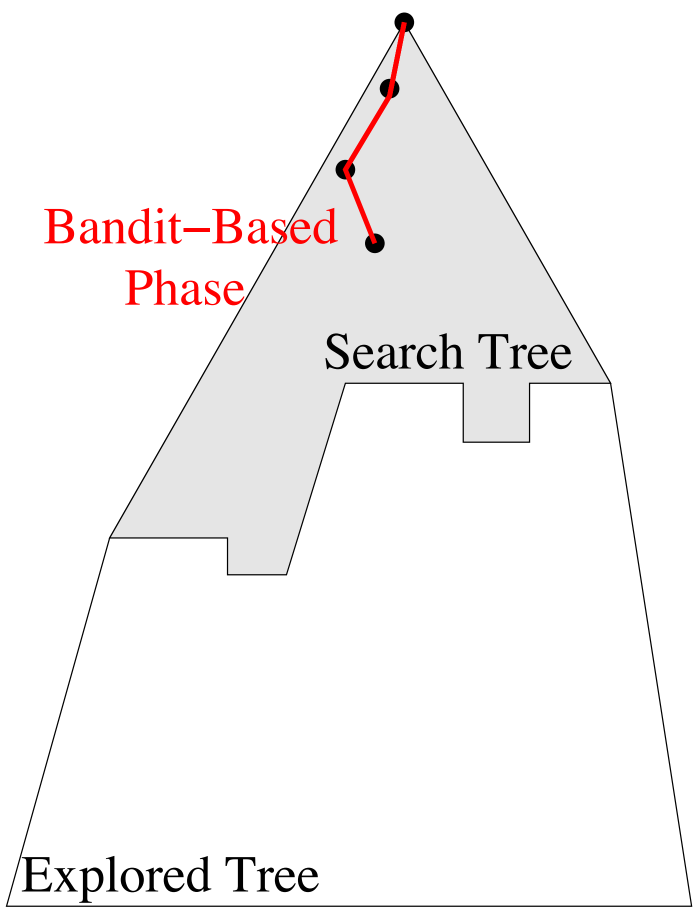
Basic MCTS pseudocode

```
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)
```



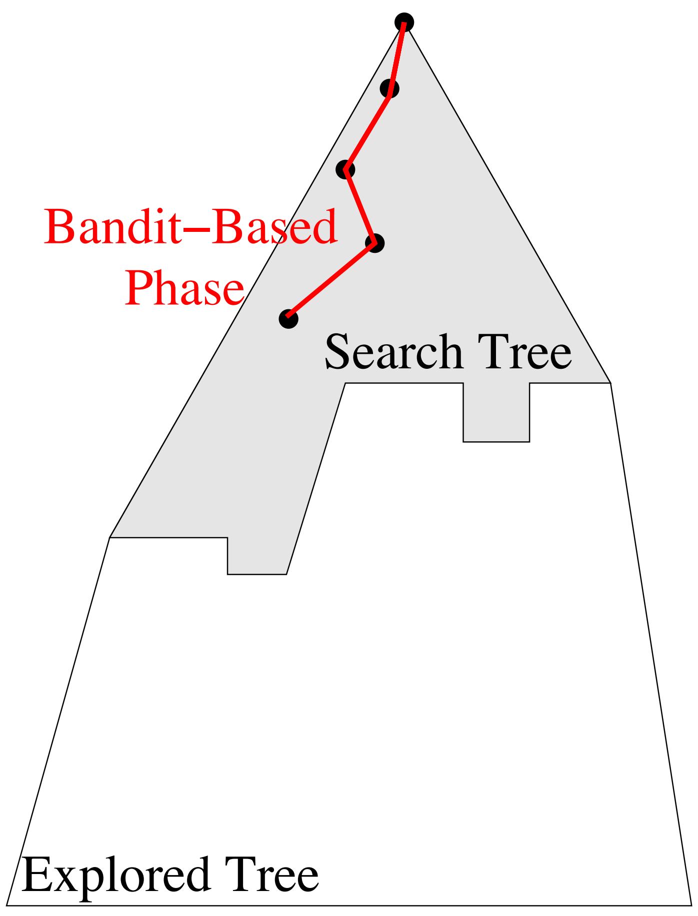
Basic MCTS pseudocode

```
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)
```



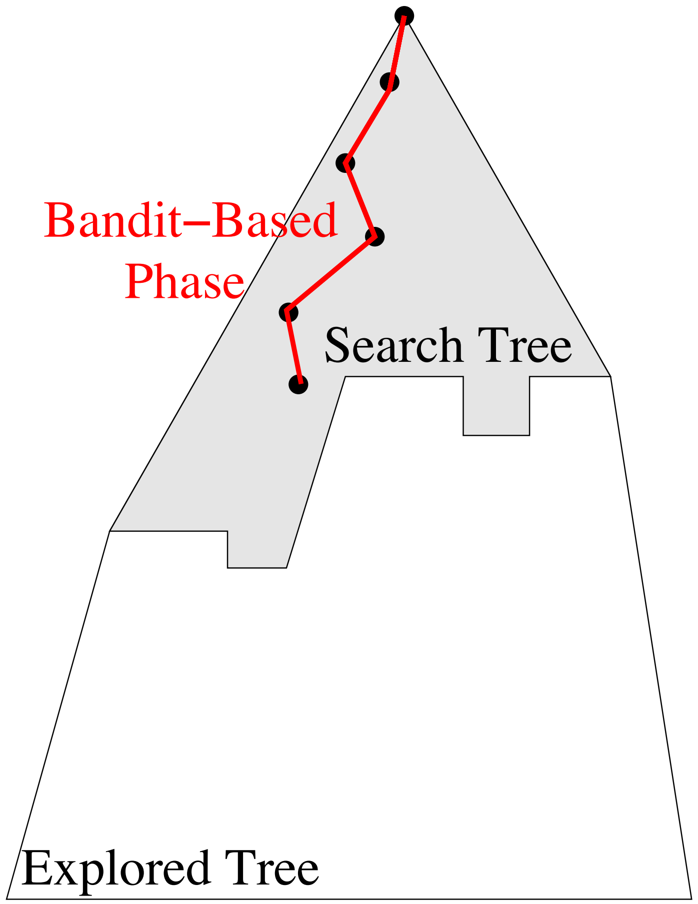
Basic MCTS pseudocode

```
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)
```



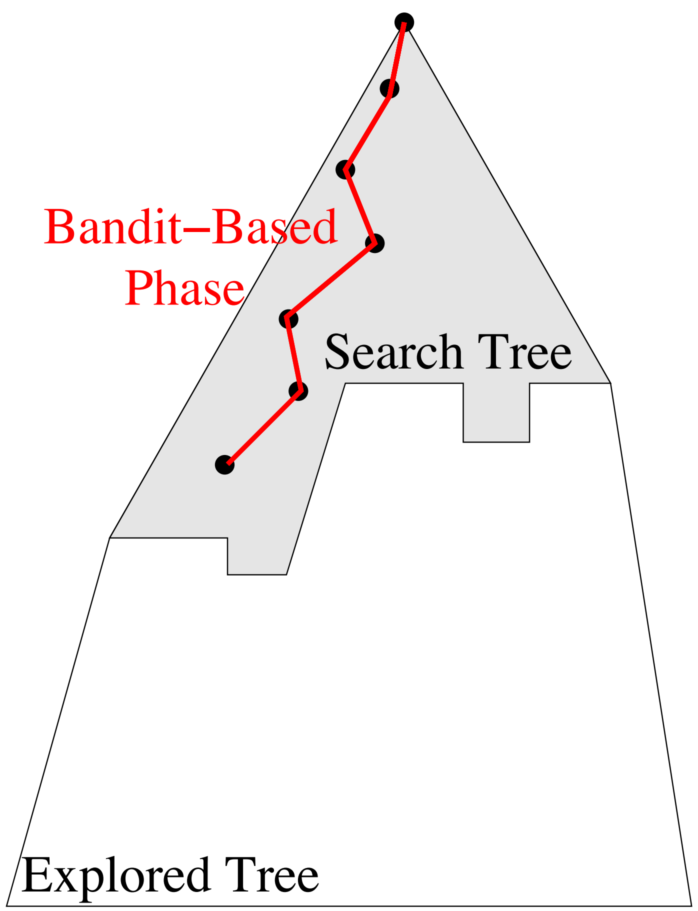
Basic MCTS pseudocode

```
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)
```



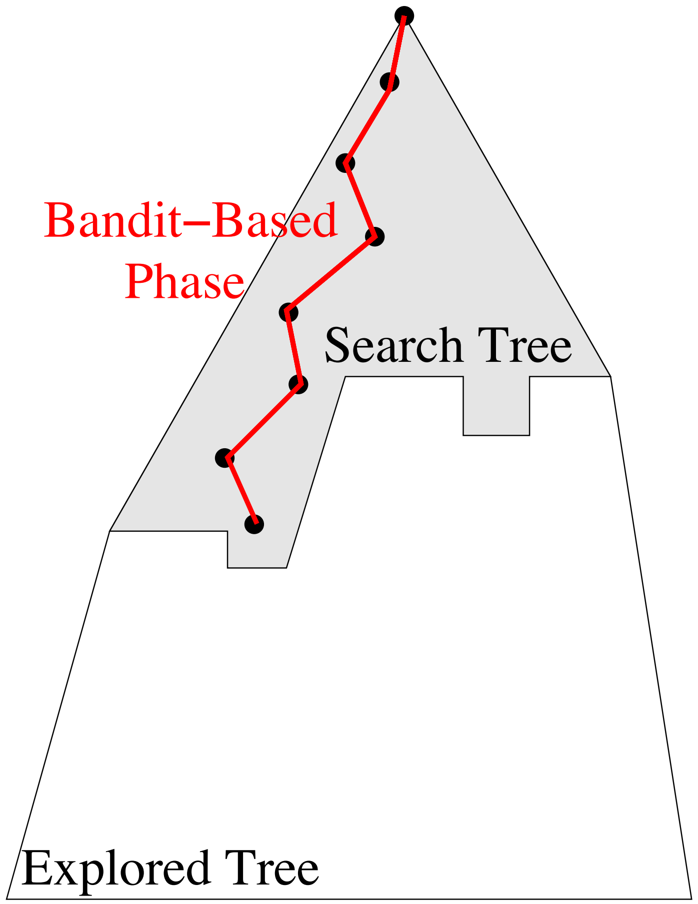
Basic MCTS pseudocode

```
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)
```



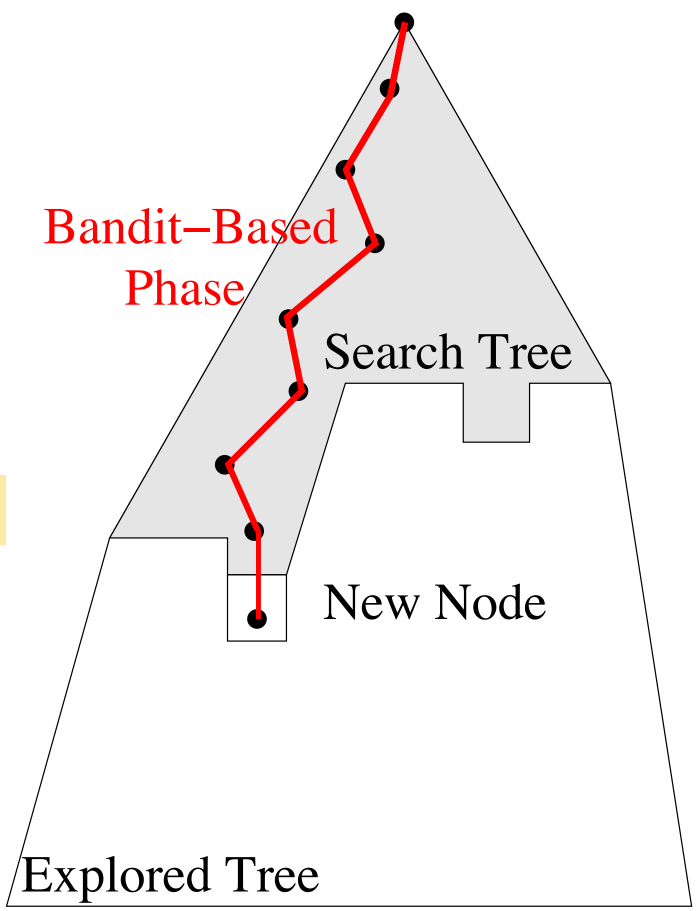
Basic MCTS pseudocode

```
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

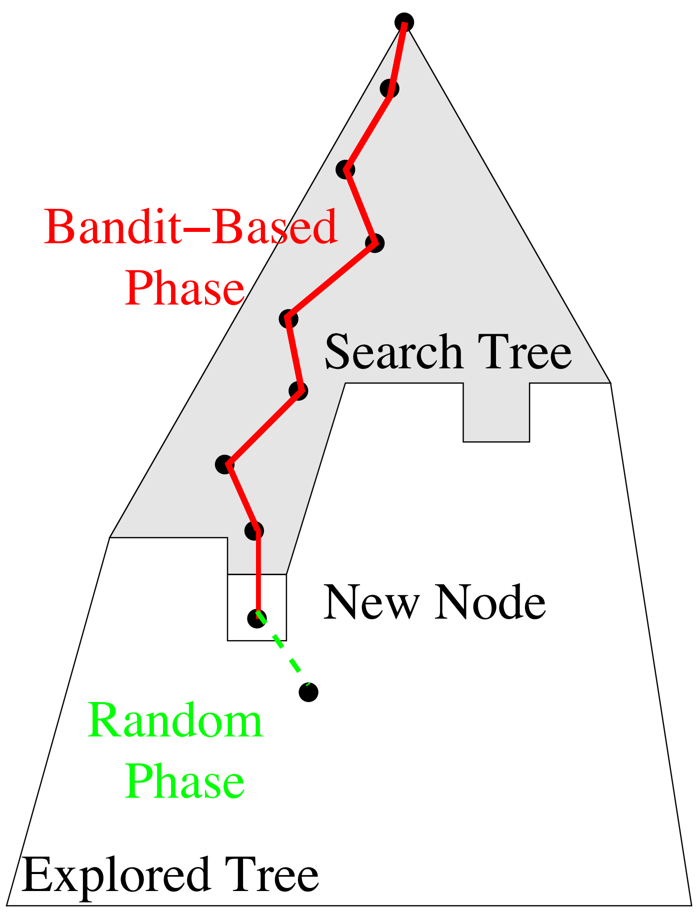
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
     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



Basic MCTS pseudocode

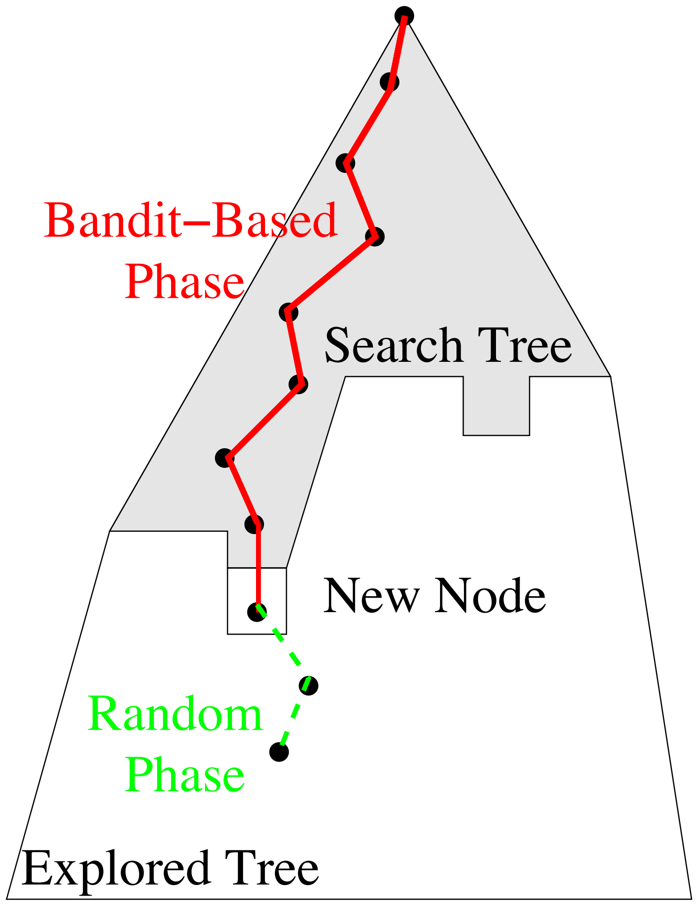
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
     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

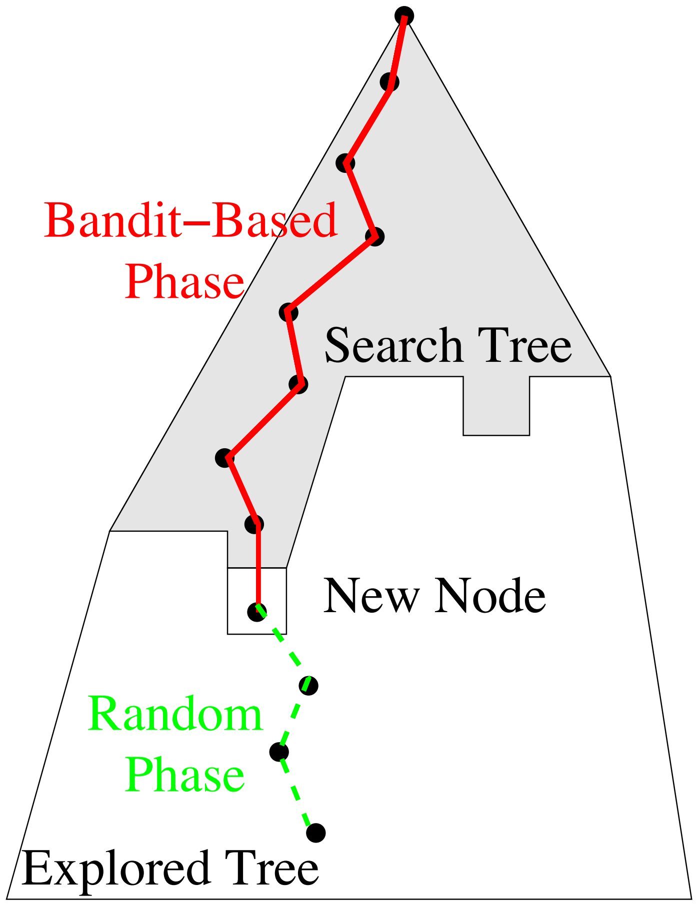
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
     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

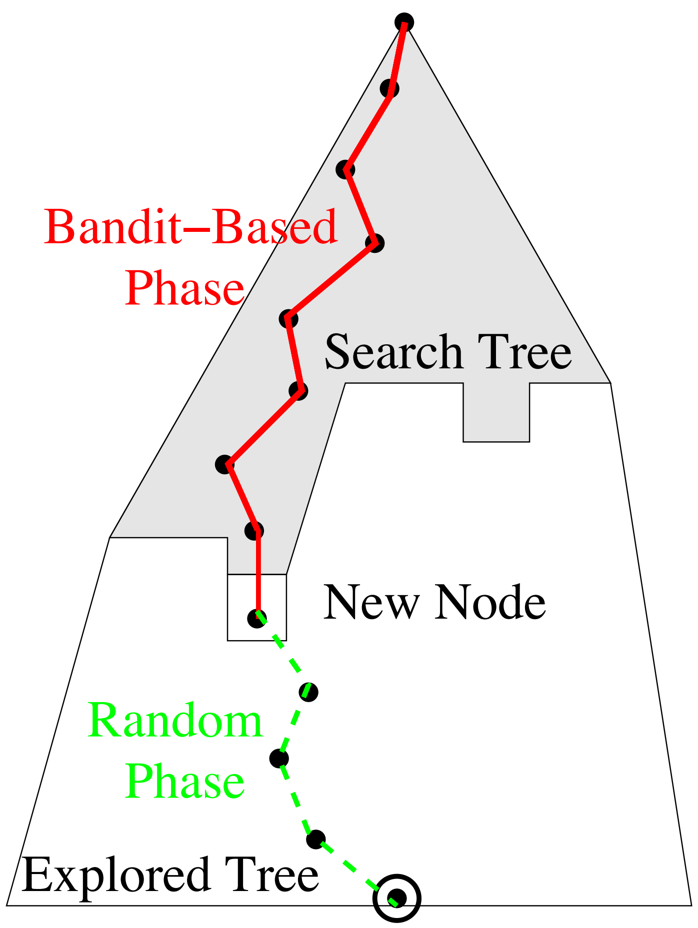
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
     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

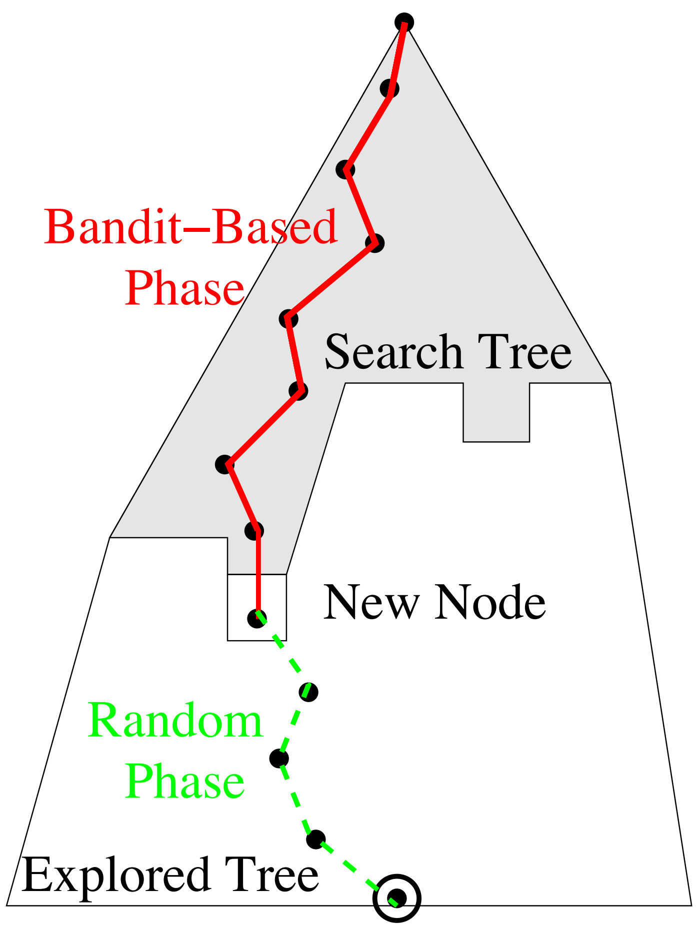
```
function MCTS_sample(state)
    state.visits++
    if all children of state expanded:
        next_state = UCB_sample(state)
        winner = MCTS_sample(next_state)
    else:
        if some children of state expanded:
            next_state = expand(random unexpanded child)
        else:
     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



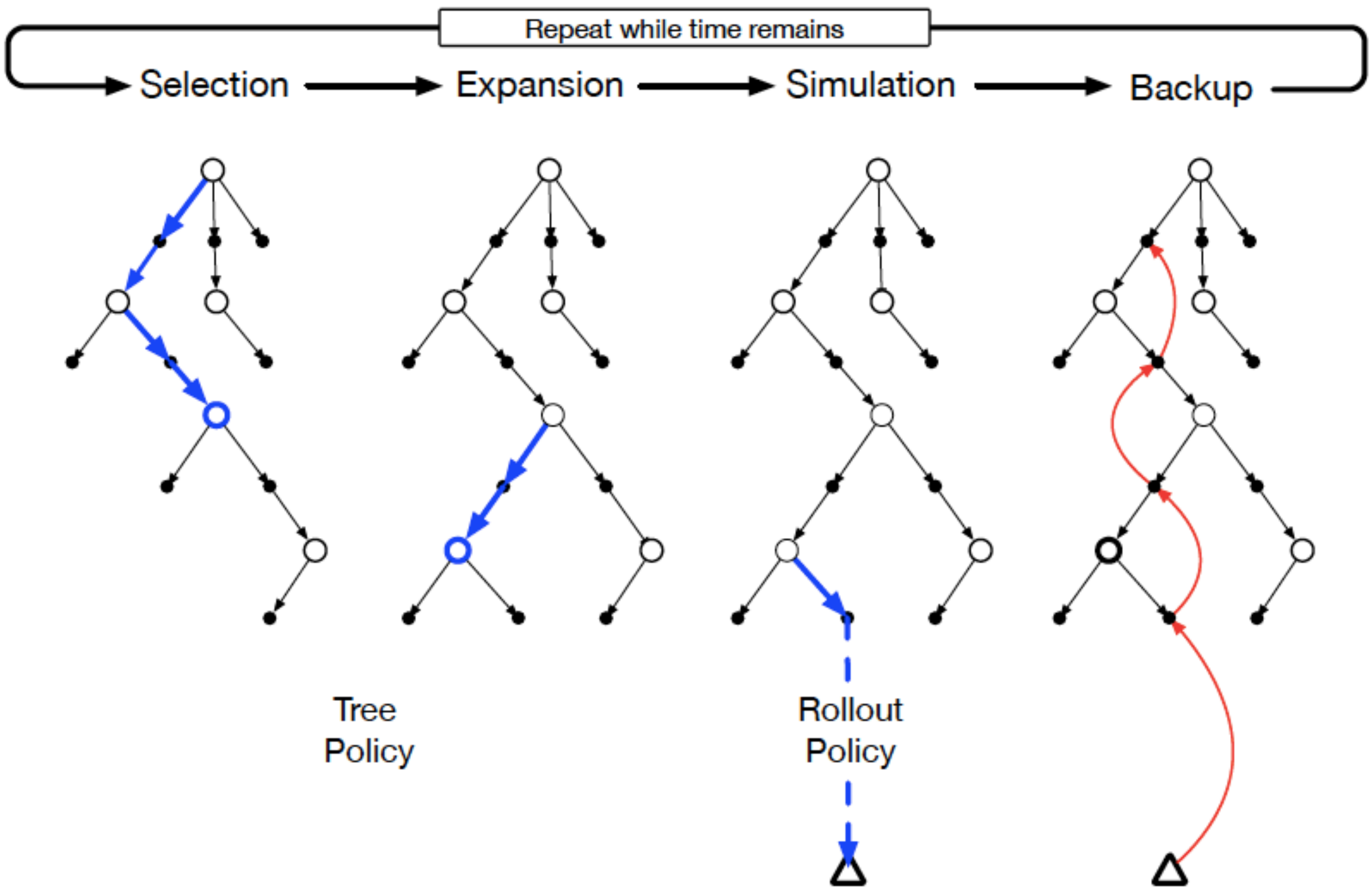

```
function random_playout(state):
    if is_terminal(state):
        return winner
    else: return random_playout(random_move(state))
```

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)
```



Monte-Carlo Tree Search



Monte-Carlo Tree Search

Use cases:

- As online planner for selecting the next move
- For state-action value estimation at training time. At test time just use the reactive policy network, without any lookahead planning.
- In combination with policy and value networks at test time (AlphaGo)
- In combination with policy and value networks at both train and test time (AlphaGoZero)

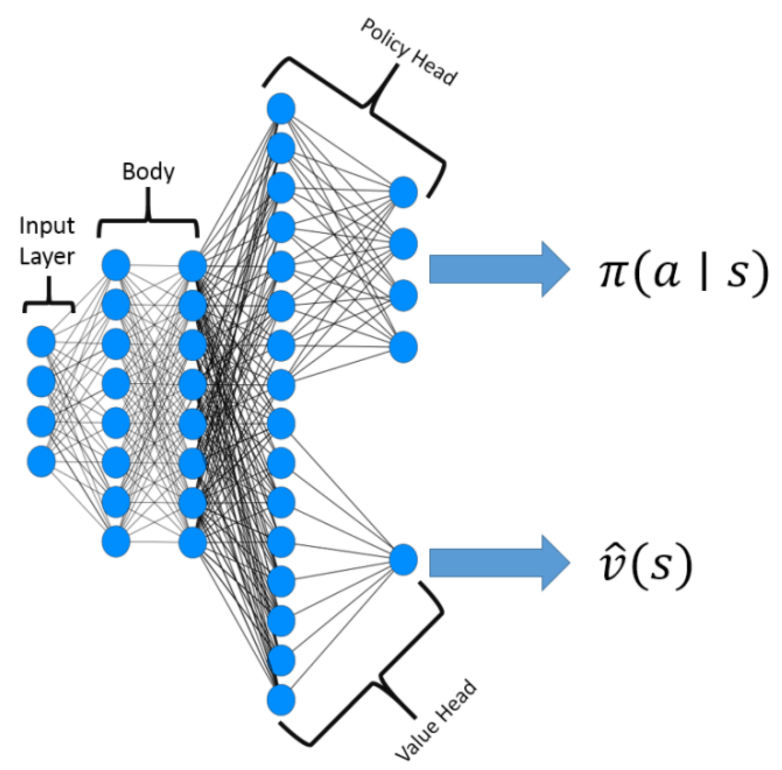
Can we do better?

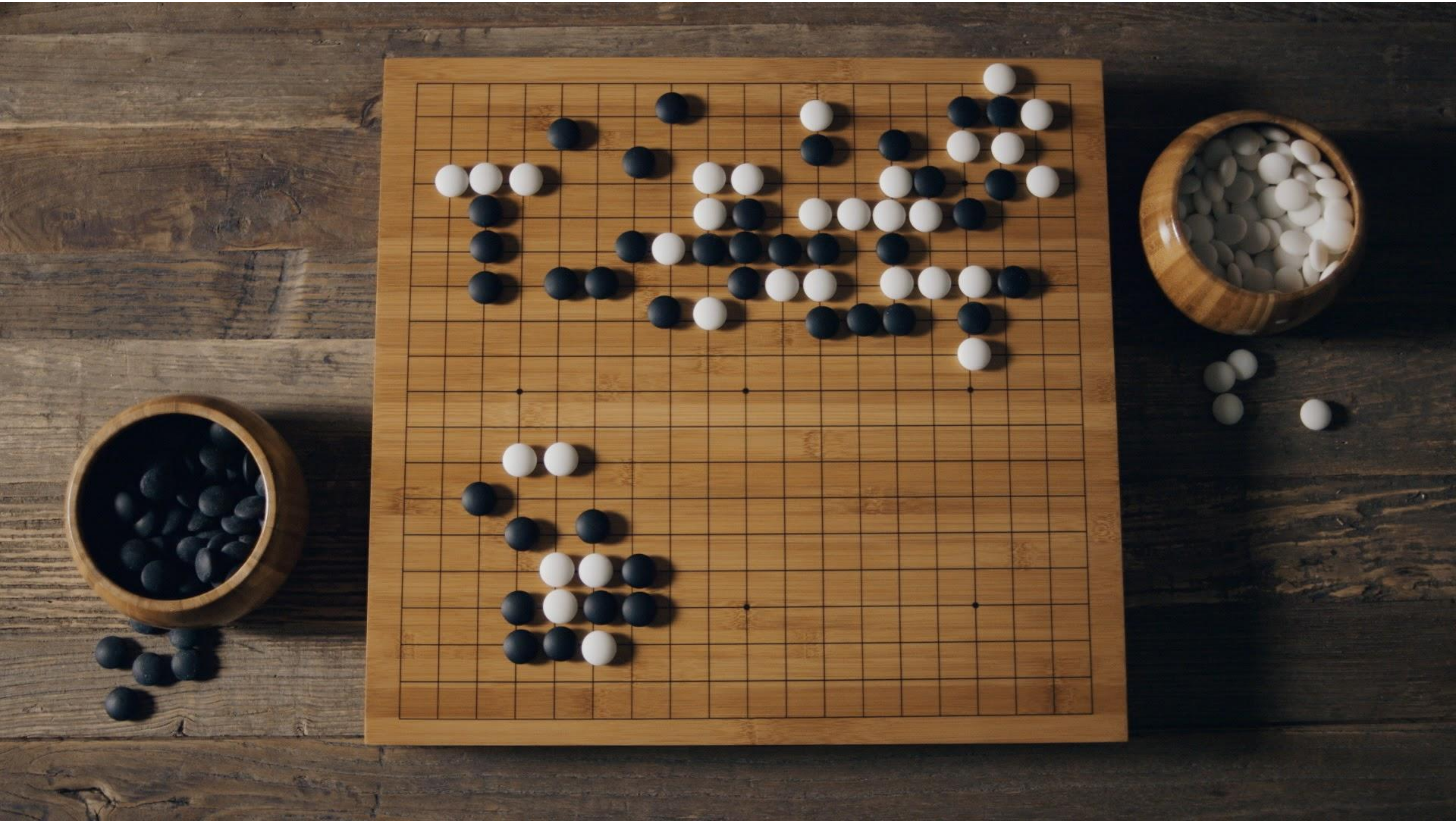
Can we inject prior knowledge into state and action values instead of initializing them uniformly?

MCTS + Policy/ Value networks

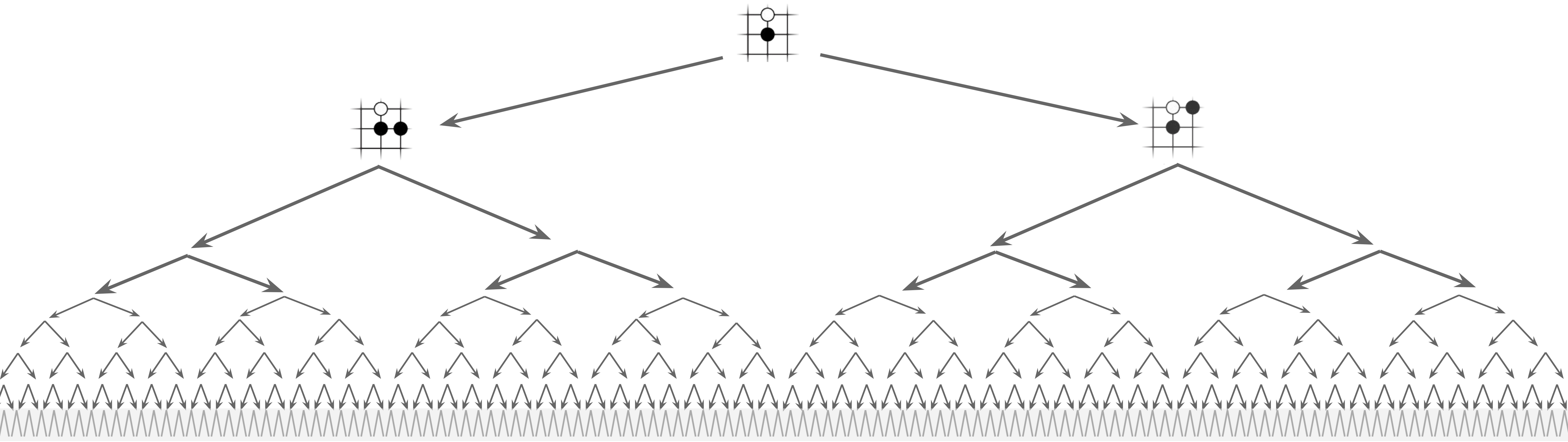
- Value neural net to evaluate board positions to help prune the tree depth.

- Policy neural net to select moves to help prune the tree breadth.



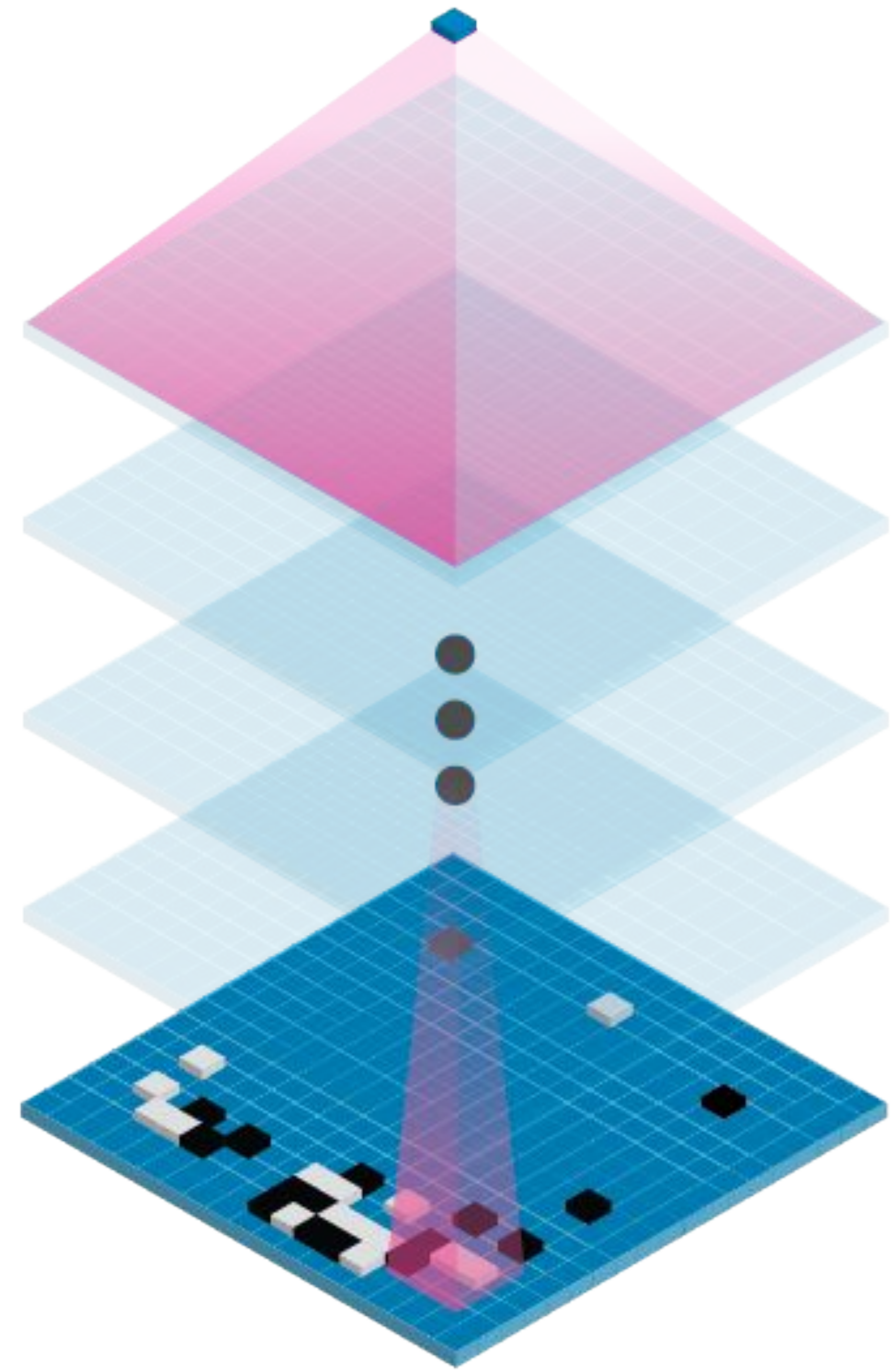


Exhaustive Search

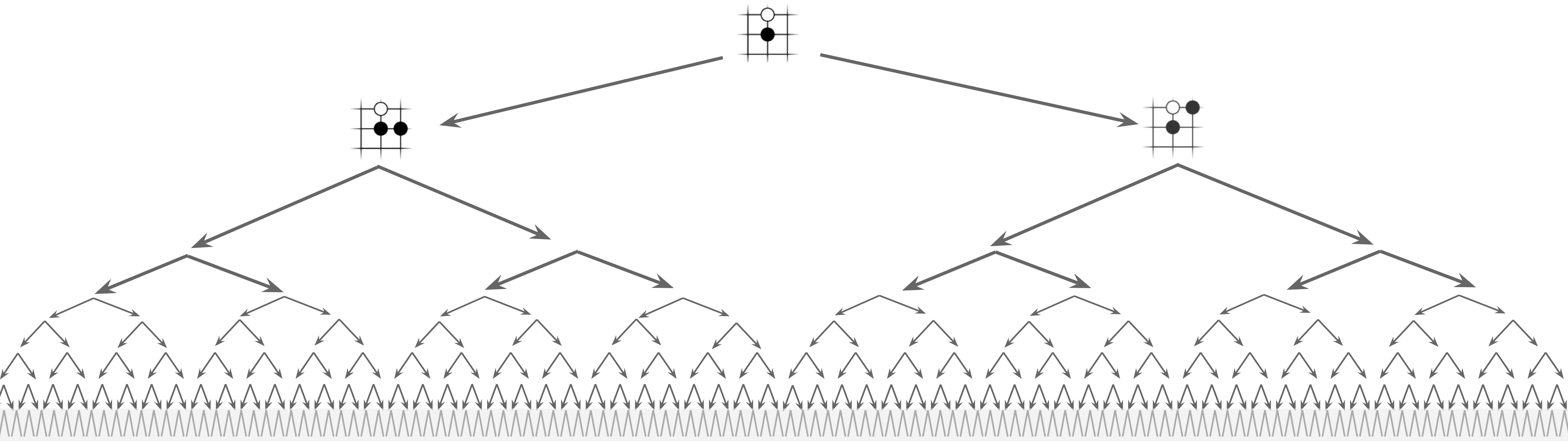


Value Network

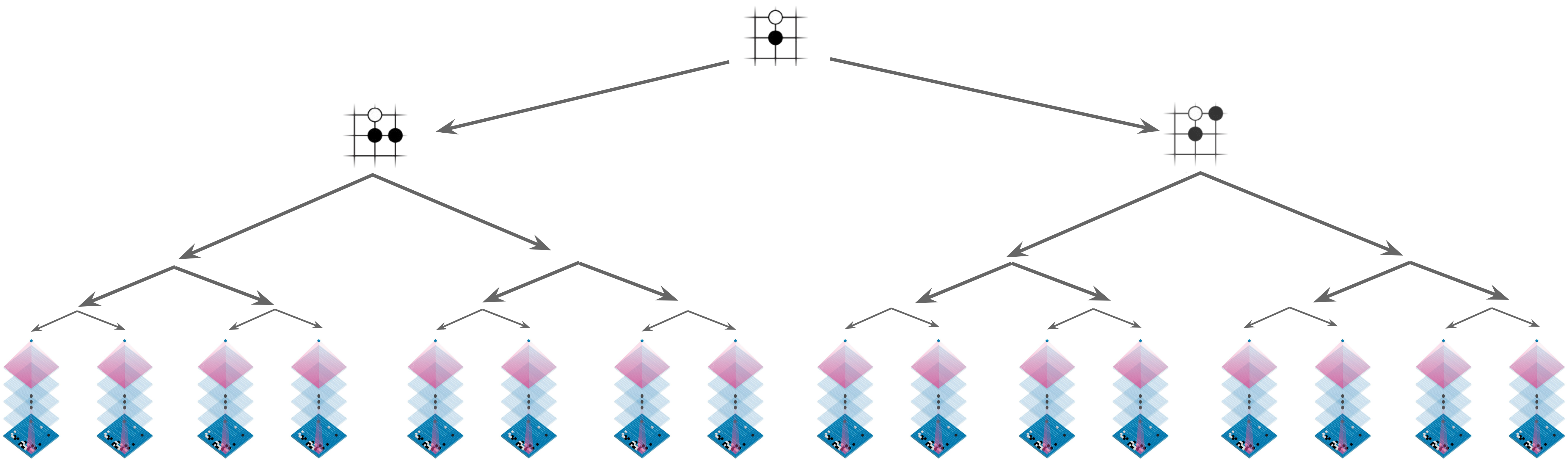
How well am I doing?



Reduce Depth with Value Network

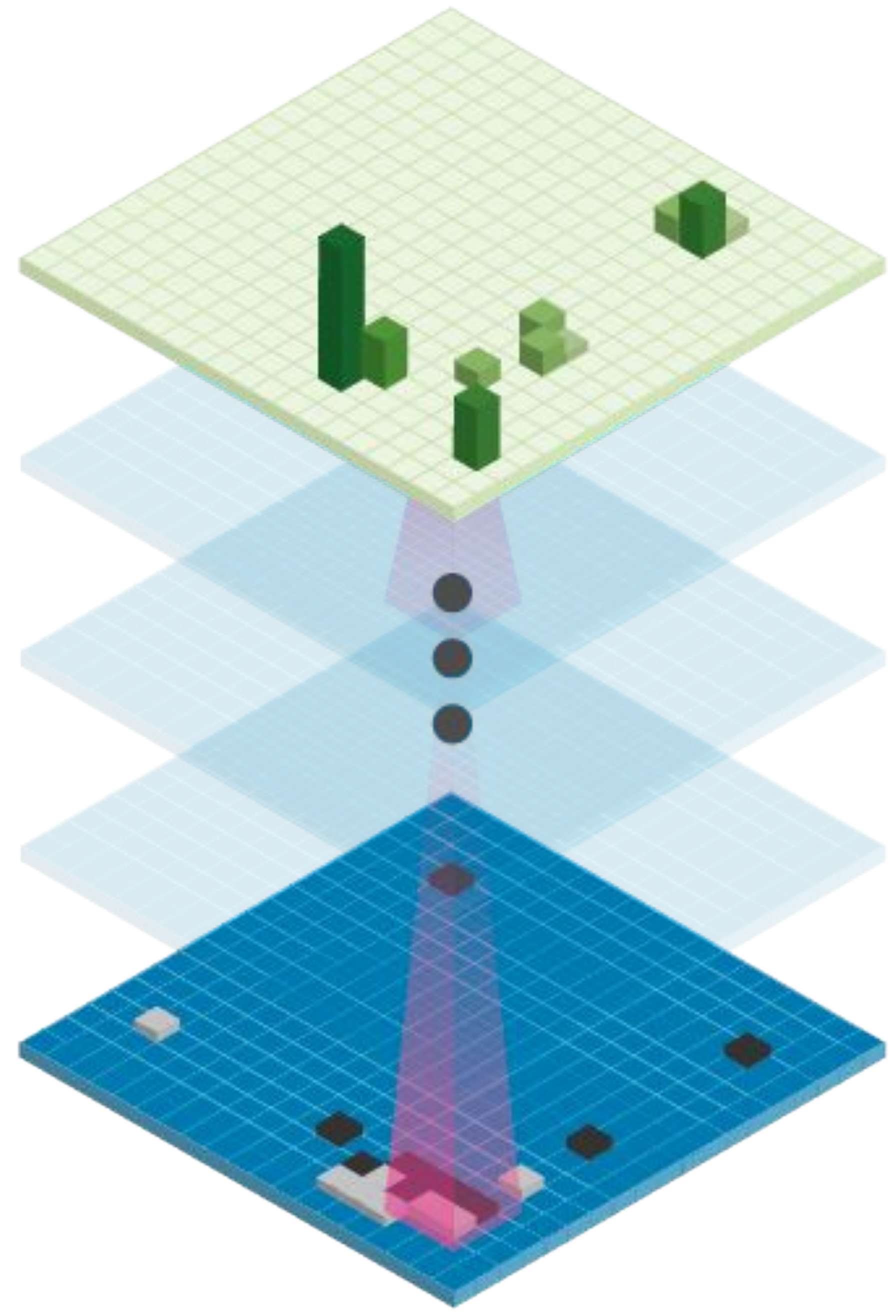


Reduce Depth with Value Network

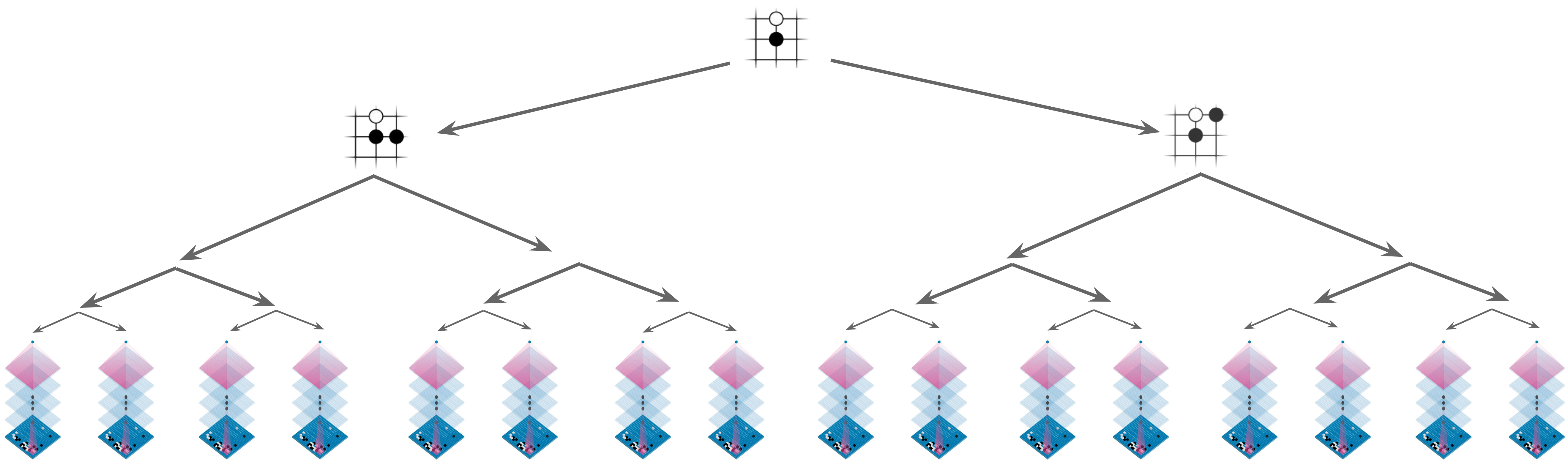


Prior Network

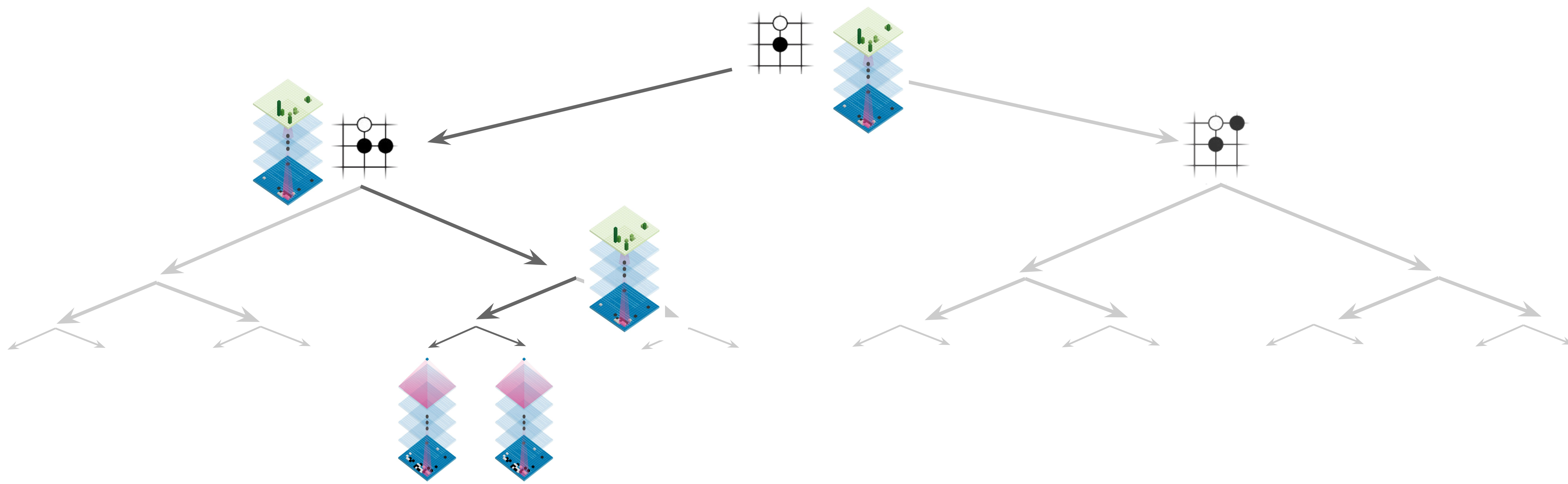
What are the most likely actions?



Reduce Breadth with Policy Network



Reduce Breadth with Policy Network



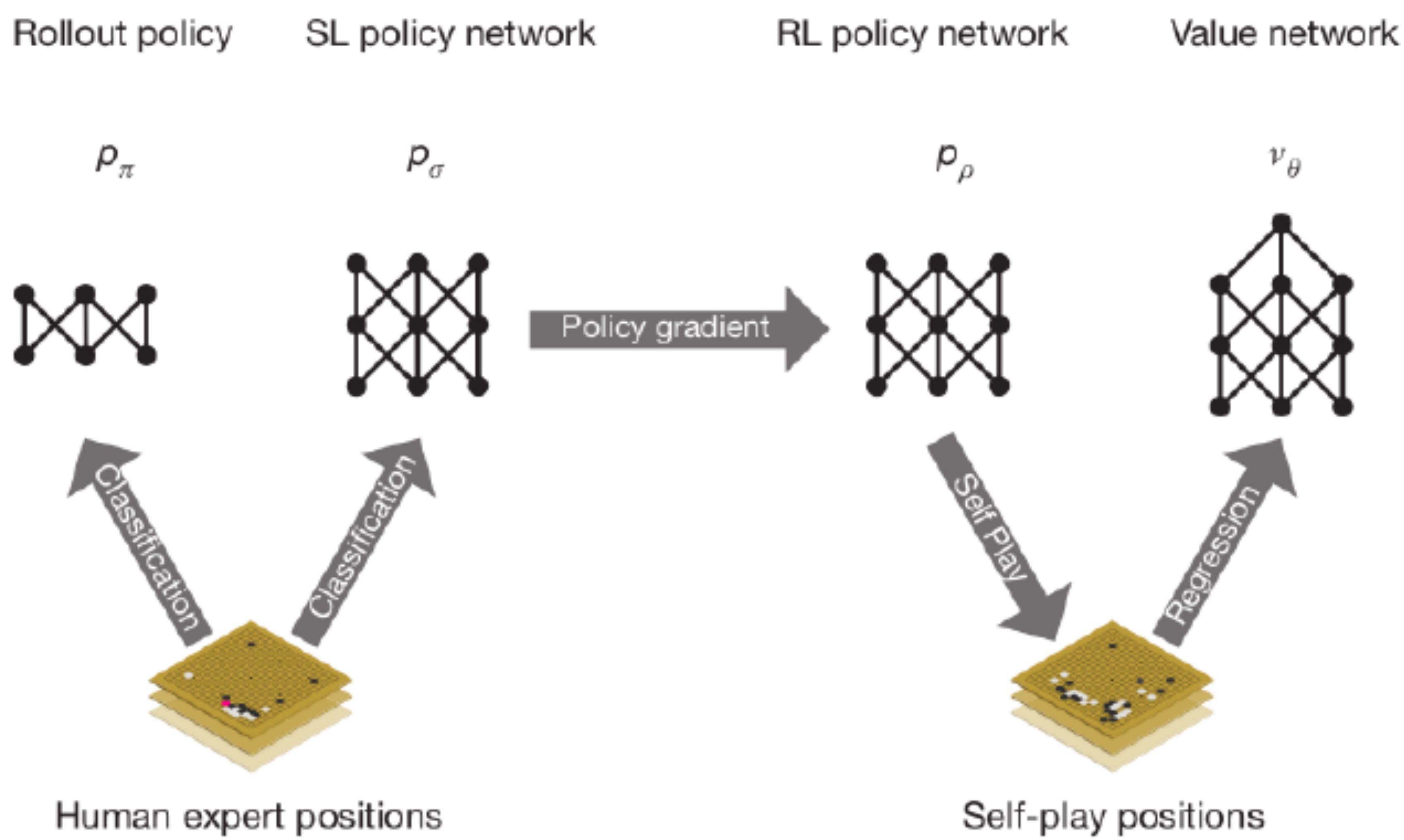
AlphaGo

1. Train two policies, one cheap policy p_π and one expensive p_σ by mimicking expert moves.

2. Train a new policy p_ρ with RL and self-play p_ρ initialized from the p_σ policy.

3. Train a value network that predicts the winner of games played by p_ρ against itself.

4. Combine the policy and value networks with MCTS at test time.



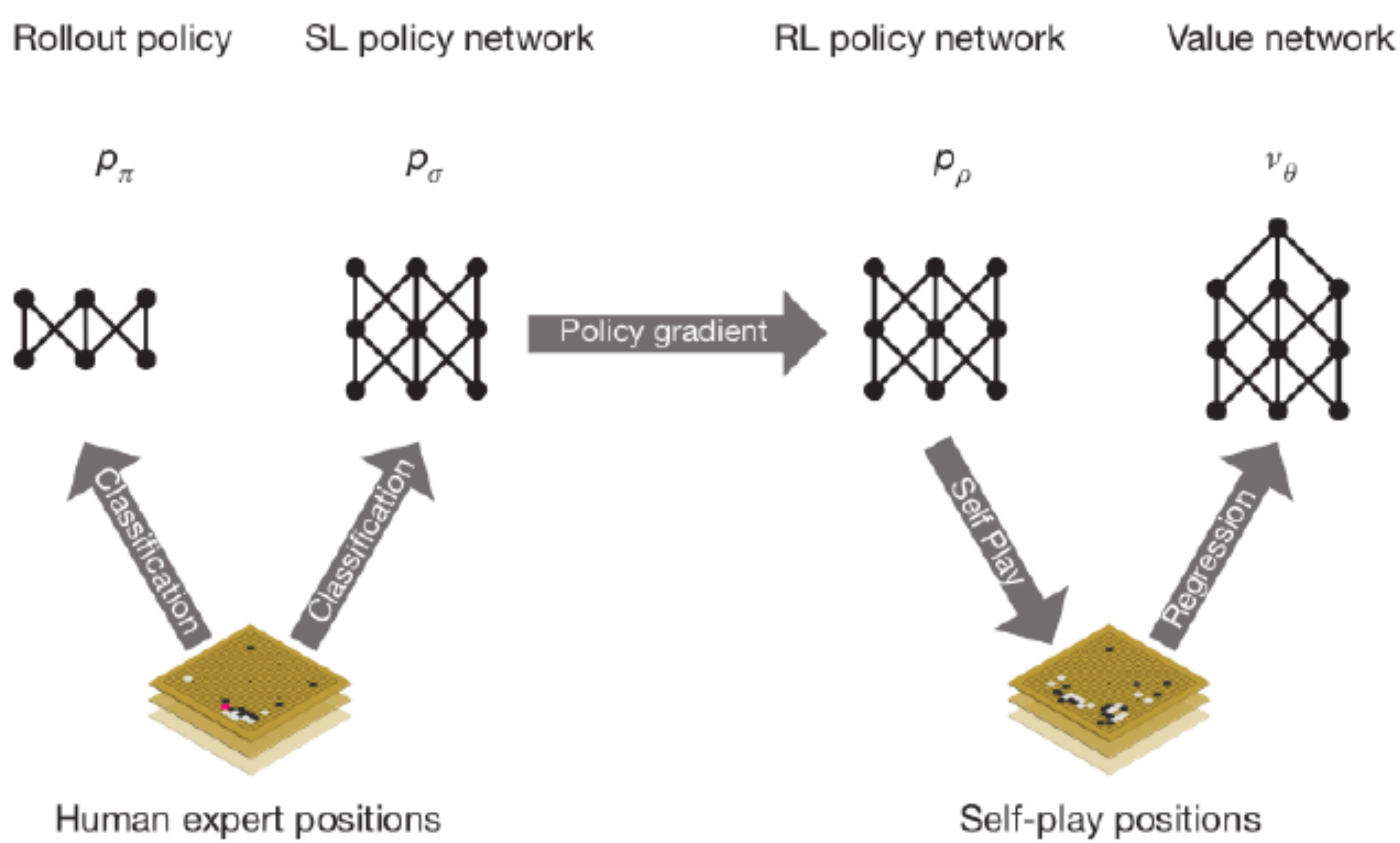
AlphaGo

1. Train two policies, one cheap policy p_π and one expensive p_σ by mimicking expert moves.

2. Train a new policy p_ρ with RL and self-play p_ρ initialized from the p_σ policy.

3. Train a value network that predicts the winner of games played by p_ρ against itself.

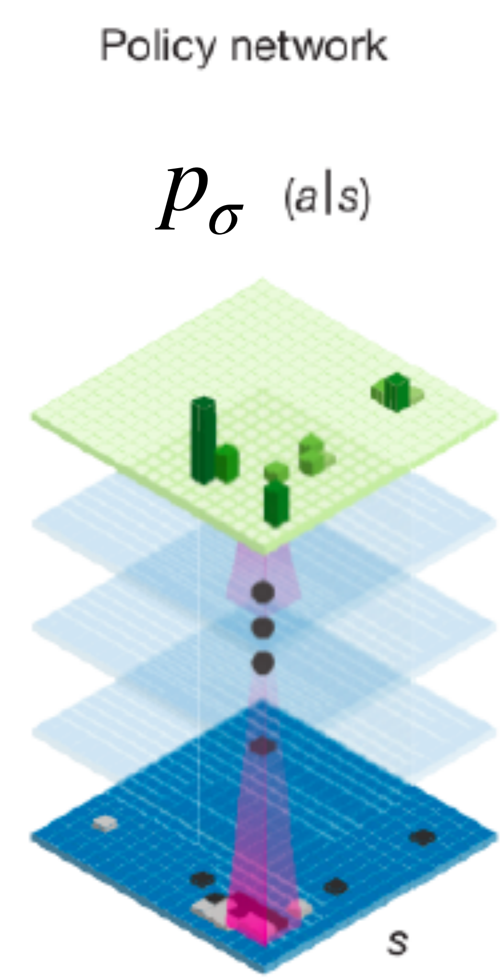
4. Combine the policy and value networks with MCTS at test time.



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%.



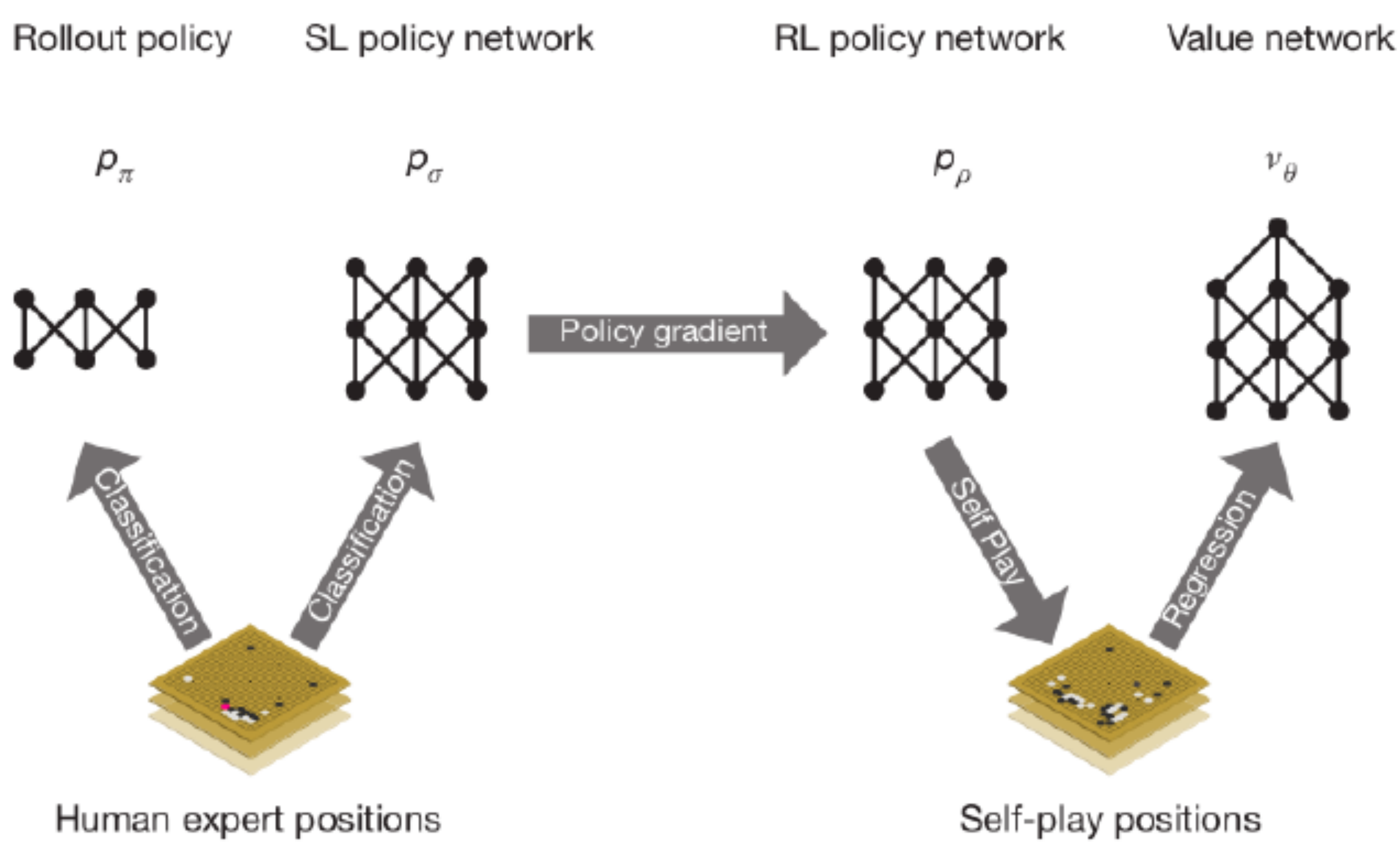
AlphaGo

1. Train two policies, one cheap policy p_π and one expensive p_σ by mimicking expert moves.

2. Train a new policy p_ρ with RL and self-play p_ρ initialized from the p_σ policy.

3. Train a value network that predicts the winner of games played by p_ρ against itself.

4. Combine the policy and value networks with MCTS at test time.



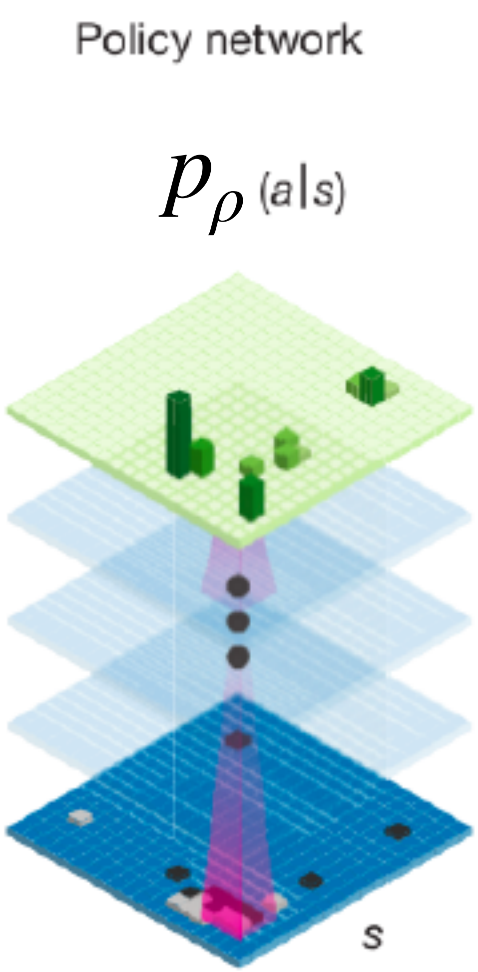
Reinforcement learning of policy networks

- 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

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

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



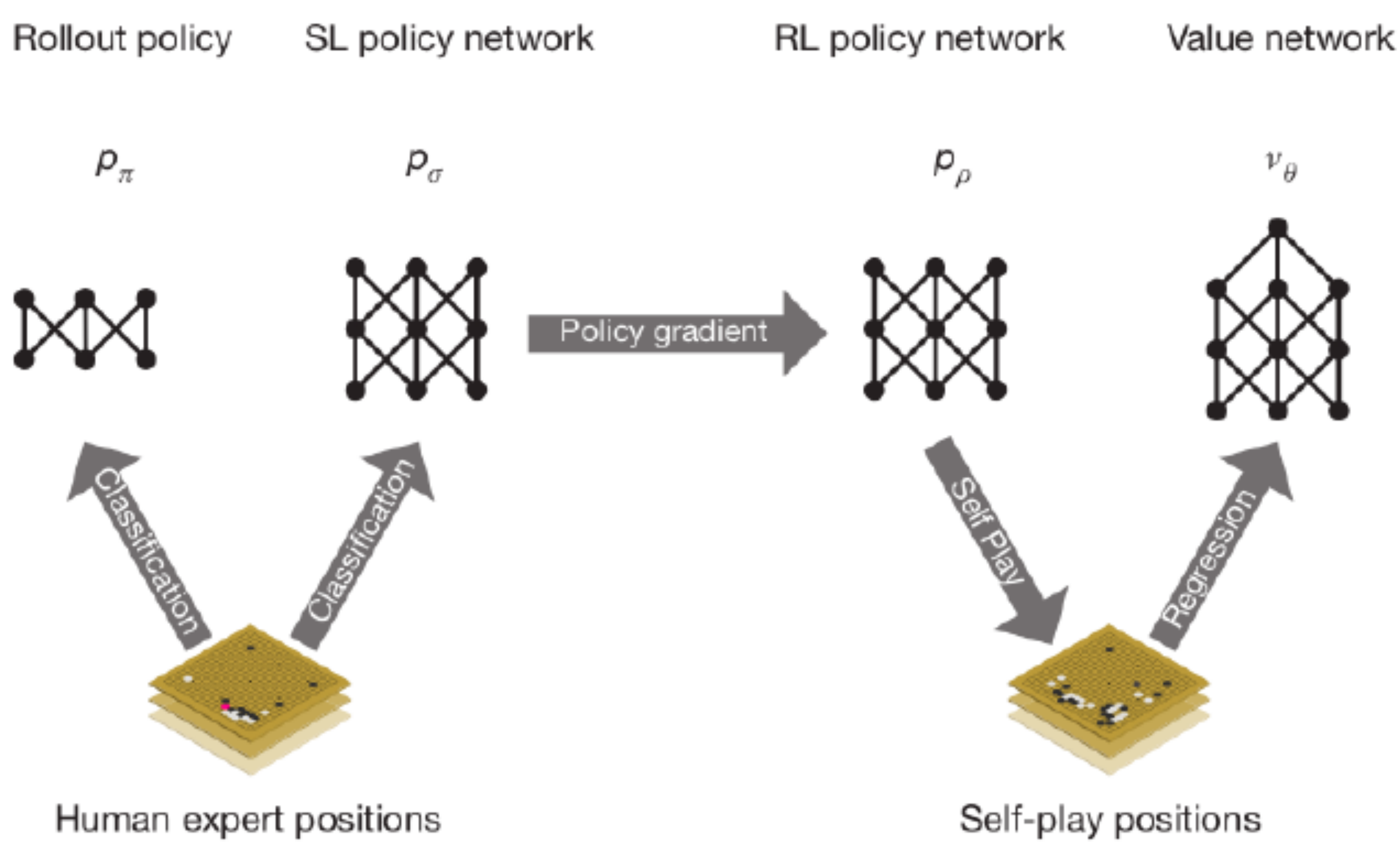
AlphaGo

1. Train two policies, one cheap policy p_π and one expensive p_σ by mimicking expert moves.

2. Train a new policy p_ρ with RL and self-play p_ρ initialized from the p_σ policy.

3. Train a value network that predicts the winner of games played by p_ρ against itself.

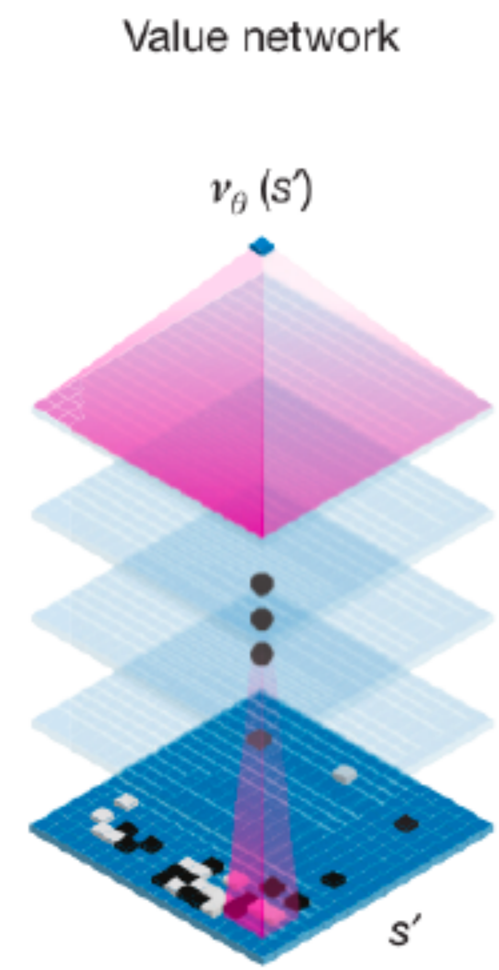
4. Combine the policy and value networks with MCTS at test time.



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.
- Input: Sampled states during self-play, 30 million distinct positions, each sampled from a separate game.
- 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.



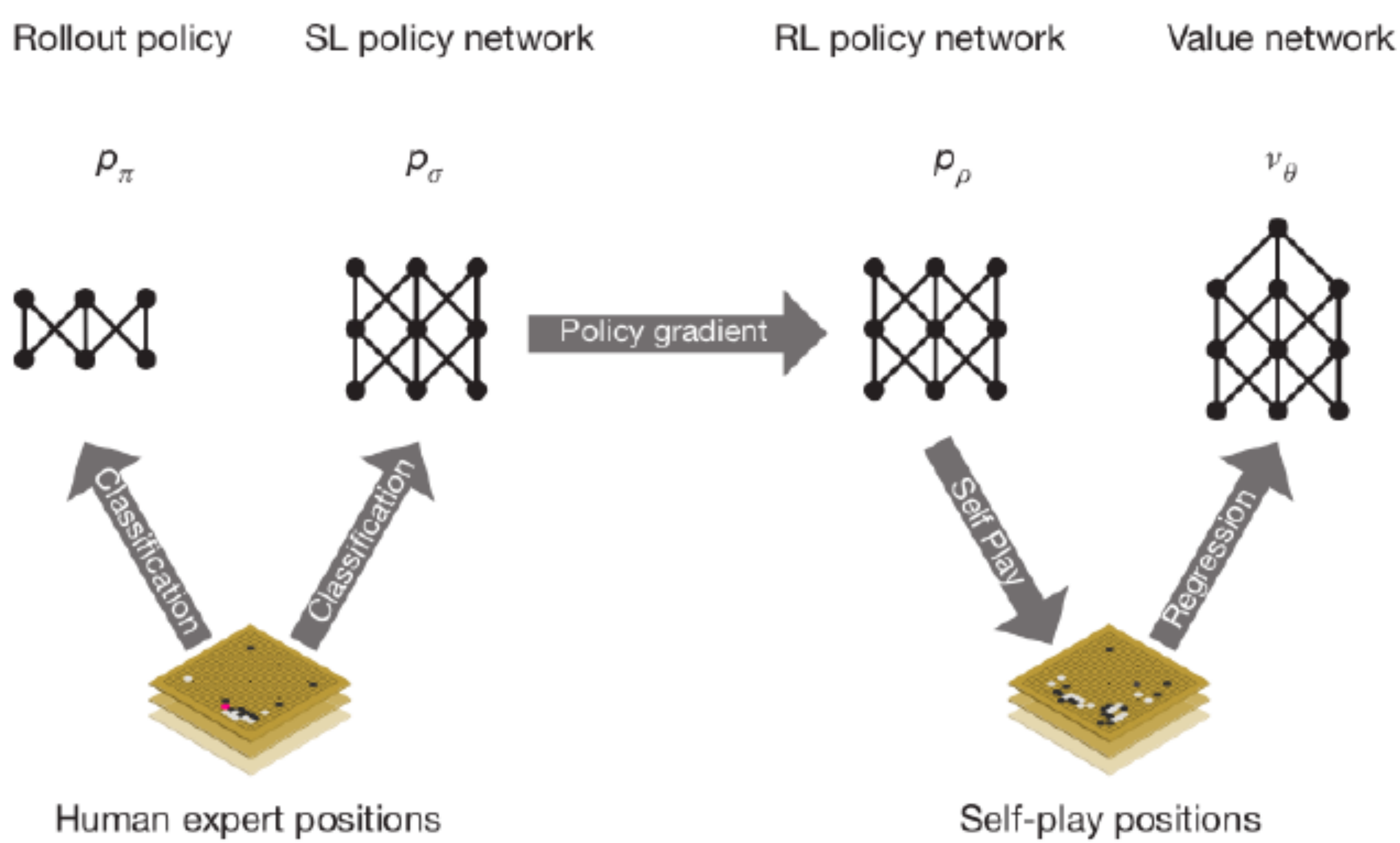
AlphaGo

1. Train two policies, one cheap policy p_π and one expensive p_σ by mimicking expert moves.

2. Train a new policy p_ρ with RL and self-play p_ρ initialized from the p_σ policy.

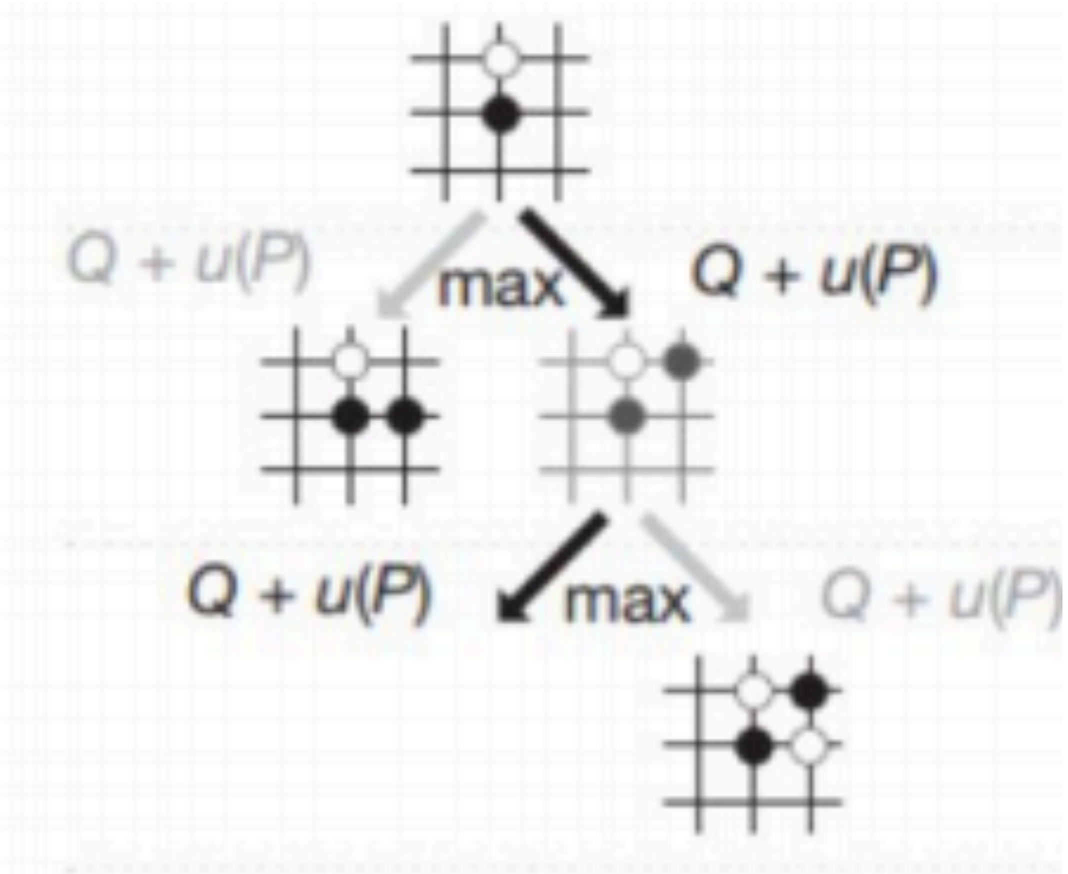
3. Train a value network that predicts the winner of games played by p_ρ against itself.

4. Combine the policy and value networks with MCTS at test time.



MCTS + Policy/ Value networks

Selection: selecting actions within the expanded tree



Tree policy

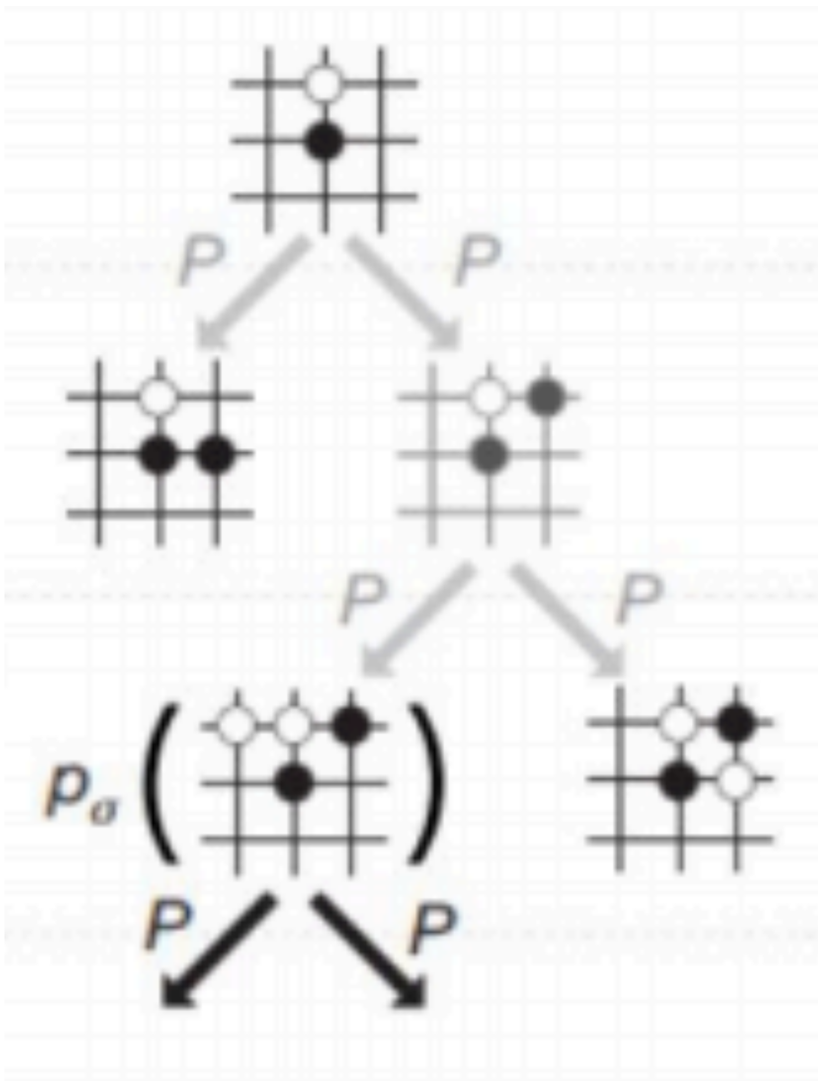
$$a_t = \underset{a}{\operatorname{argmax}} \left(Q\left(s_t, a\right) + u\left(s_t, a\right)\right)$$

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

- a_t - action selected at time step t from state s_t
- $Q\left(s_r, a\right)$- average reward collected so far from MC simulations
- $P(s, a)$ - prior expert probability provided by the SL policy p_σ
- $N(s, a)$ - number of times we have taken action a from state s from MC simulations
- u acts as a bonus value

MCTS + Policy/ Value networks

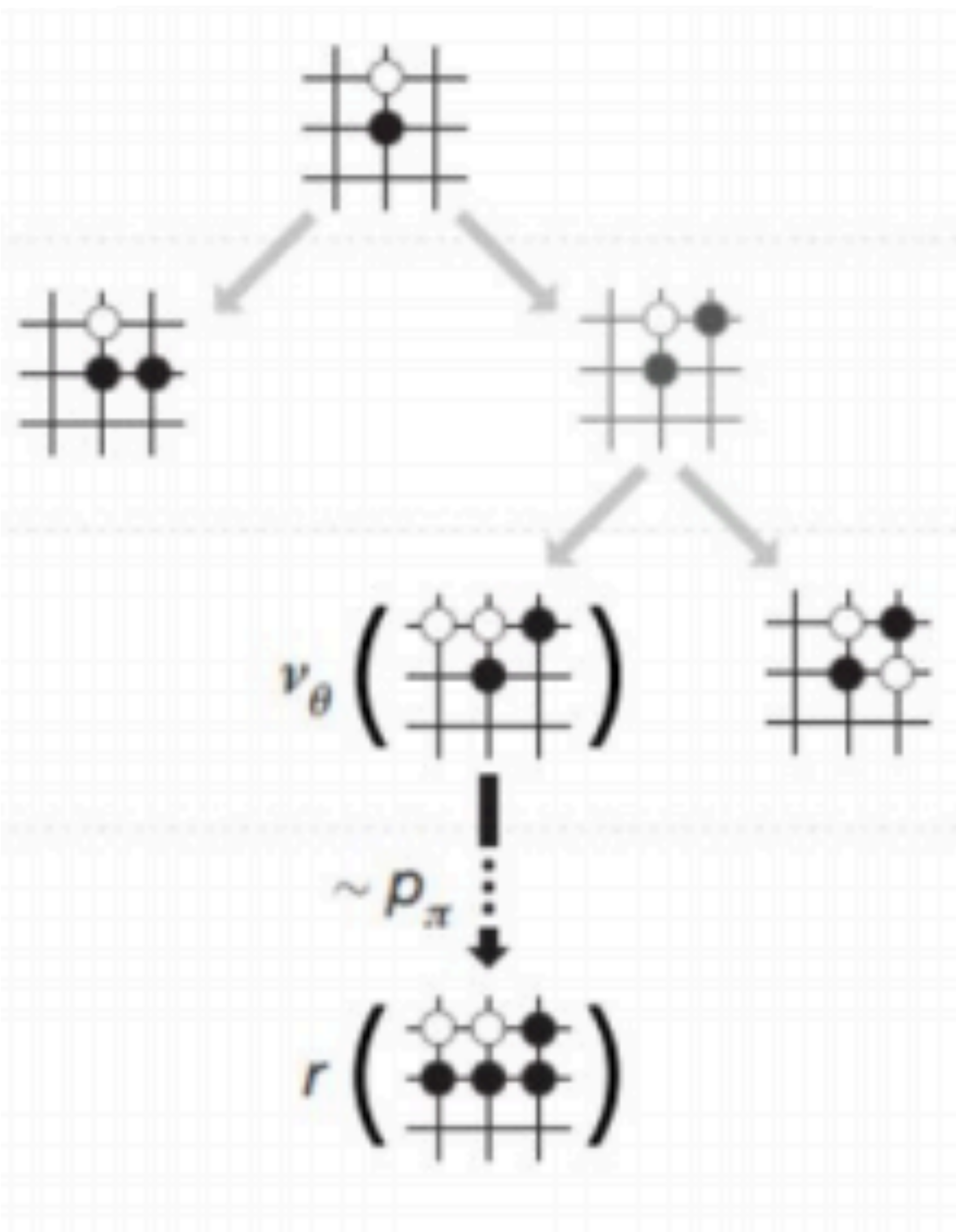
Expansion: when reaching a leaf, play the action with highest score from p_σ



- 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



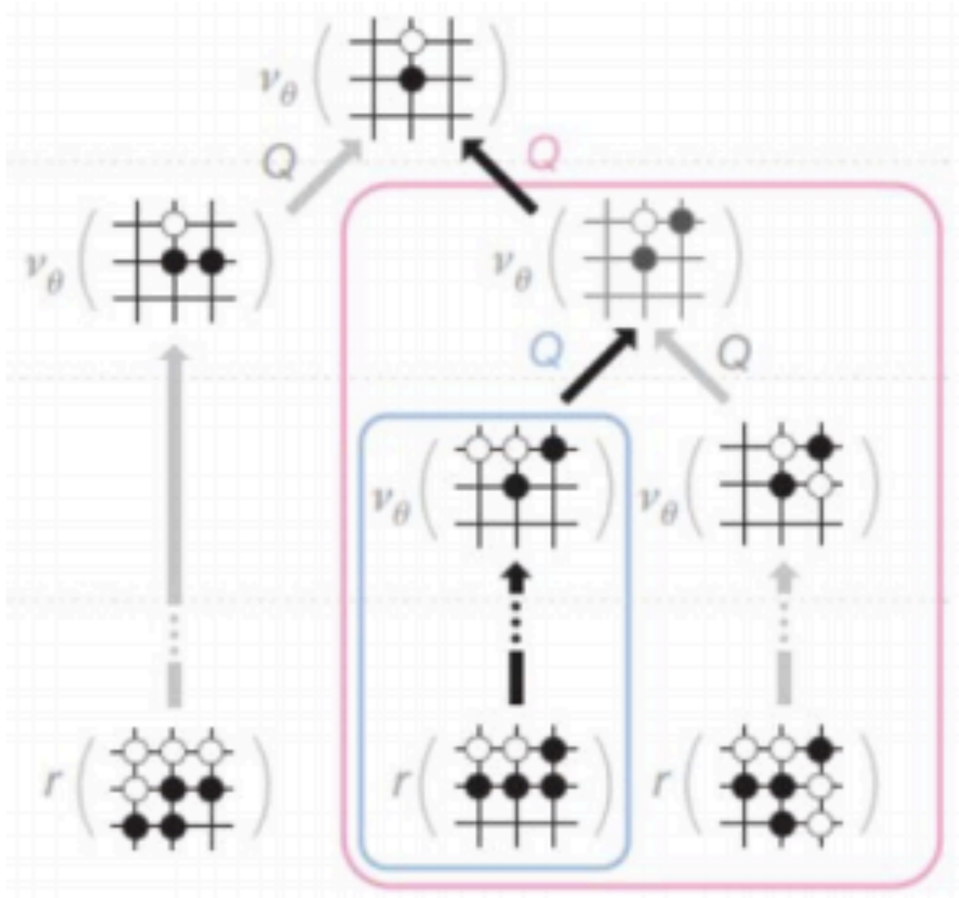
- From the selected leaf node, run multiple simulations in parallel using the rollout policy
- Evaluate the leaf node as:

$$V\left(s_L\right) = (1-\lambda)v_\rho\left(s_L\right) + \lambda z_L$$

- v_ρ : value from the trained value function for board position s_L
- z_L : Reward from fast rollout p_x
  - Played until terminal step
- λ - mixing parameter

MCTS + Policy/ Value networks

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



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

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} \mathbf{1}_{(s,a)\in\tau_i} V\left(s_L^i\right)$$

- Extra index is to denote the i simulation, n total simulations
- Update visit count and mean reward of simulations passing through node
- Once MCTS completes, the 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!
- We saw in the last lecture that MCTS is also useful at training time: it in fact reaches superior Q values that vanilla model-free RL.
- AlphaGoZero uses MCTS during training instead.
- AlphaGoZero gets rid of human supervision.

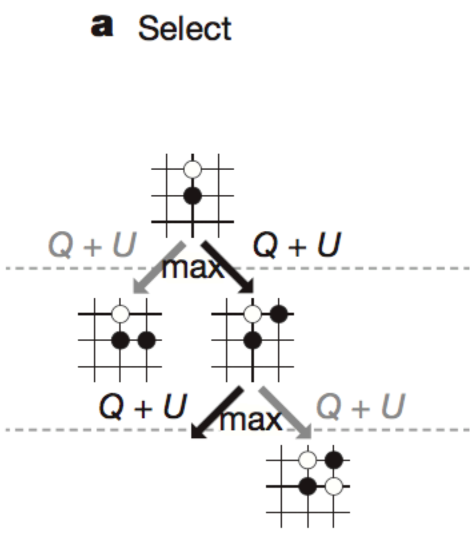
AlphaGoZero: Lookahead search during training!

- So far, look-ahead search was used for online planning at test time!
- We have seen that MCTS is useful at training time: it in fact reaches superior Q values that vanilla model-free RL.
- AlphaGoZero uses MCTS during training instead.
- AlphaGoZero does not use any human supervision and outperforms human players while trained only by self-play.

AlphaGoZero: Lookahead search during training!

- Given any policy, a MCTS guided by this policy for action selection (as described earlier), will produce an improved policy for the root node (policy improvement operator)
- Train to mimic such improved policy

MCTS + Policy/ Value networks



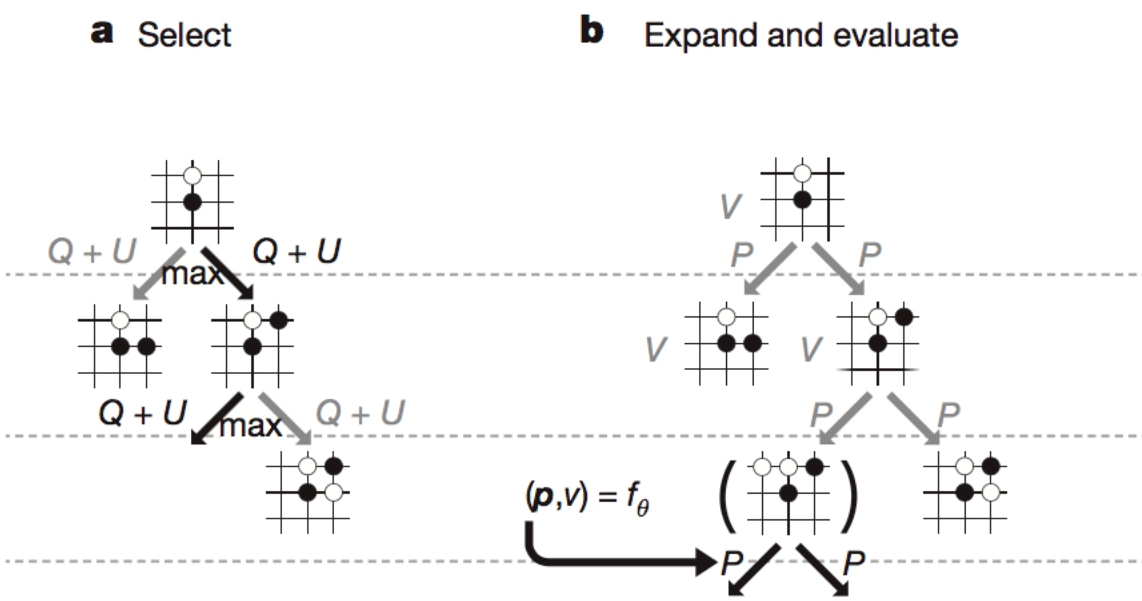
Tree policy

$$a_t = \underset{a}{\operatorname{argmax}} \left(Q\left(s_t, a\right) + U\left(s_t, a\right) \right)$$

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

- a_t - action selected at time step t from state s_t
- $Q\left(s_r, a\right)$- average reward collected so far from MC simulations
- $P(s, a)$ - prior expert probability provided by the policy π_θ
- $N(s, a)$ - number of times we have taken action a from state s from MC simulations
- U acts as a bonus value

MCTS + Policy/ Value networks

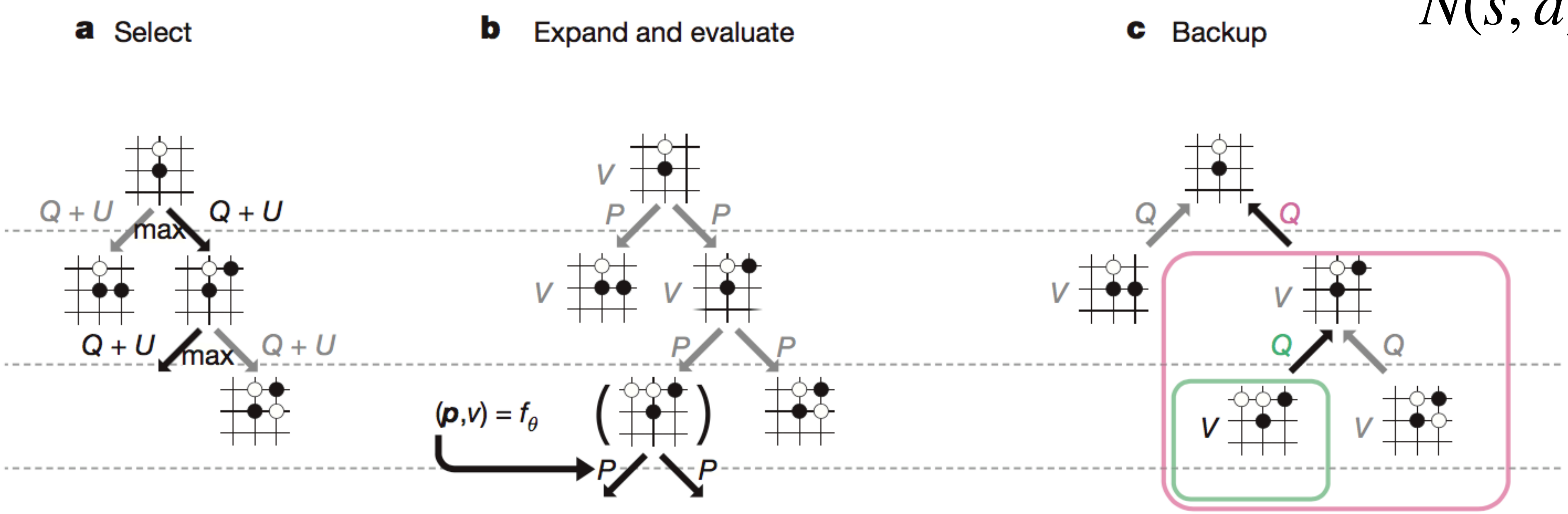


- When leaf node is reached, its value is computed $v_\theta(s)$ and the prior probs $P(s, a)$ for all its legal action-children are computed and stored.

MCTS + Policy/ Value networks

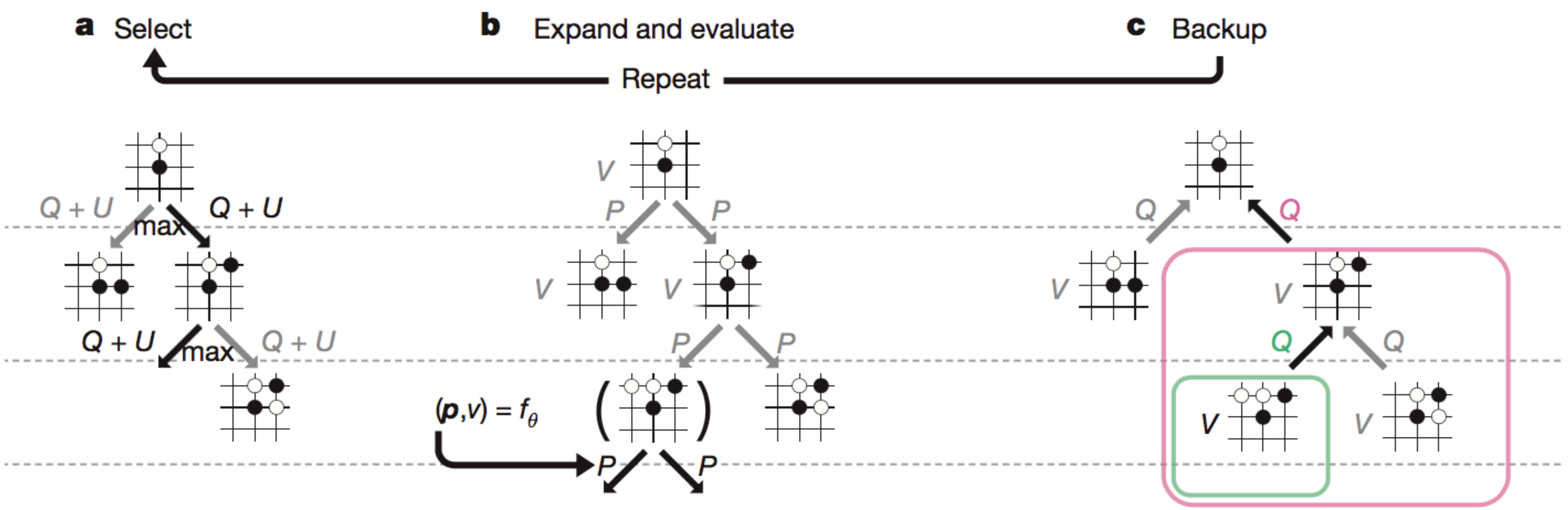
$$N(s,a) = \sum_{i=1}^{n} \mathbf{1}_{(s,a)\in\tau_i}$$

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} \mathbf{1}_{(s,a)\in\tau_i} V\left(s_L^i\right)$$



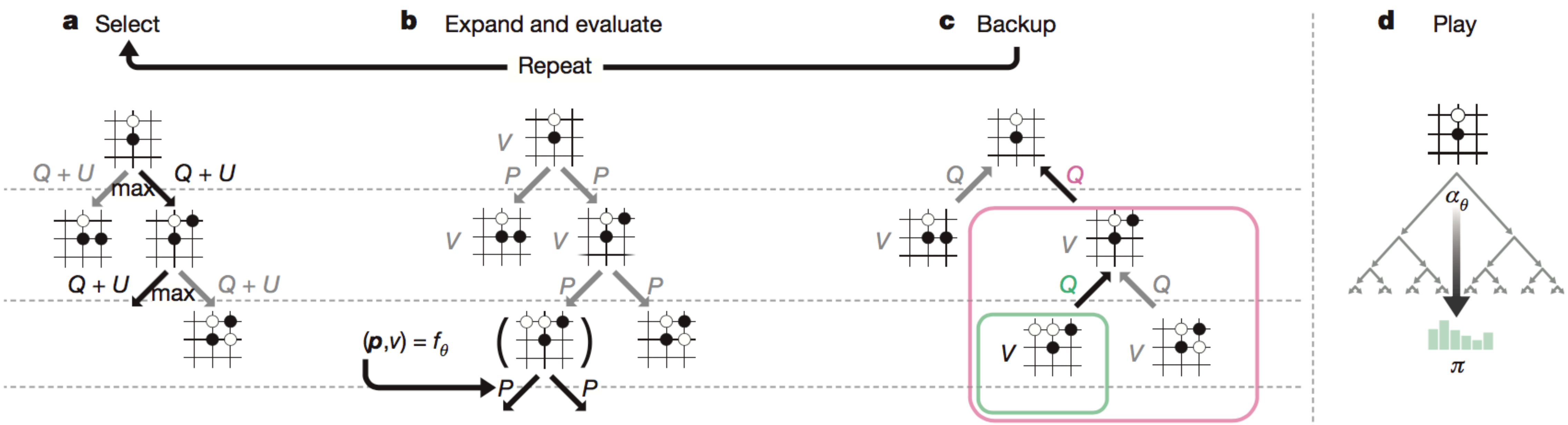
- No full rollouts till game termination!
- Update visit counts, total reward and mean reward for the actions used in the current rollout.
- n: the # of MC simulations so far

MCTS + Policy/ Value networks



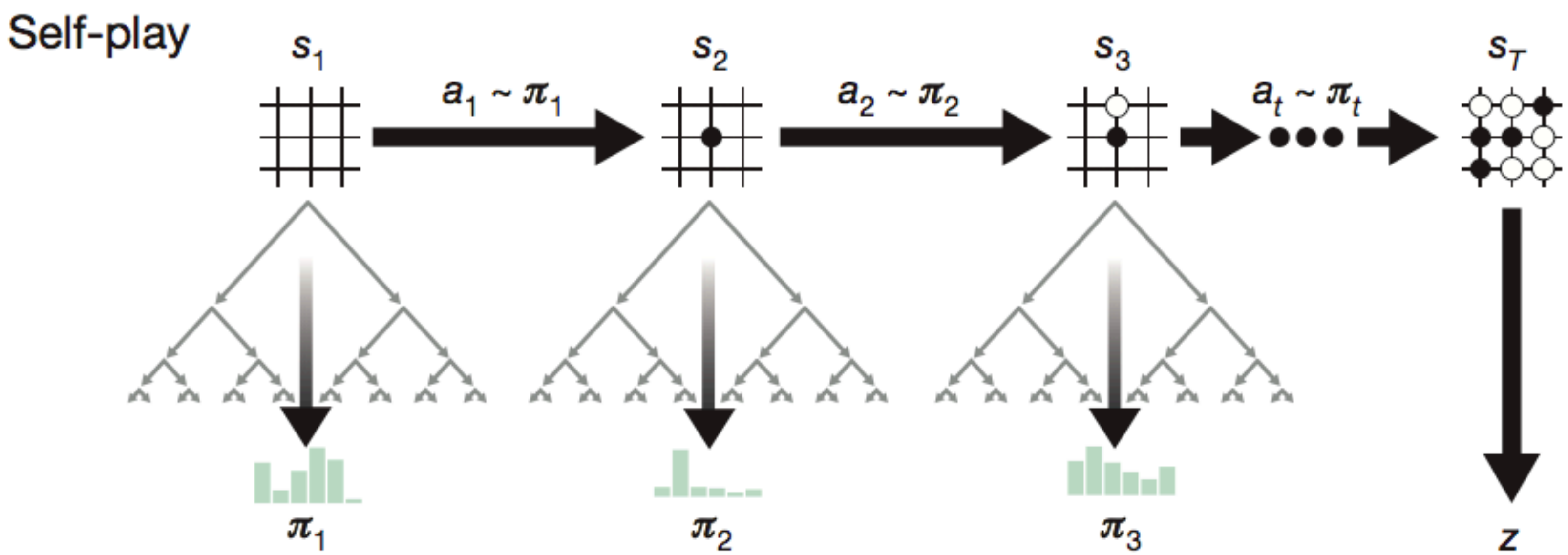
1600 MC rollouts were used to select each root action.

MCTS + Policy/ Value networks



Once MCTS completes, the algorithm chooses the most visited move from the root position.

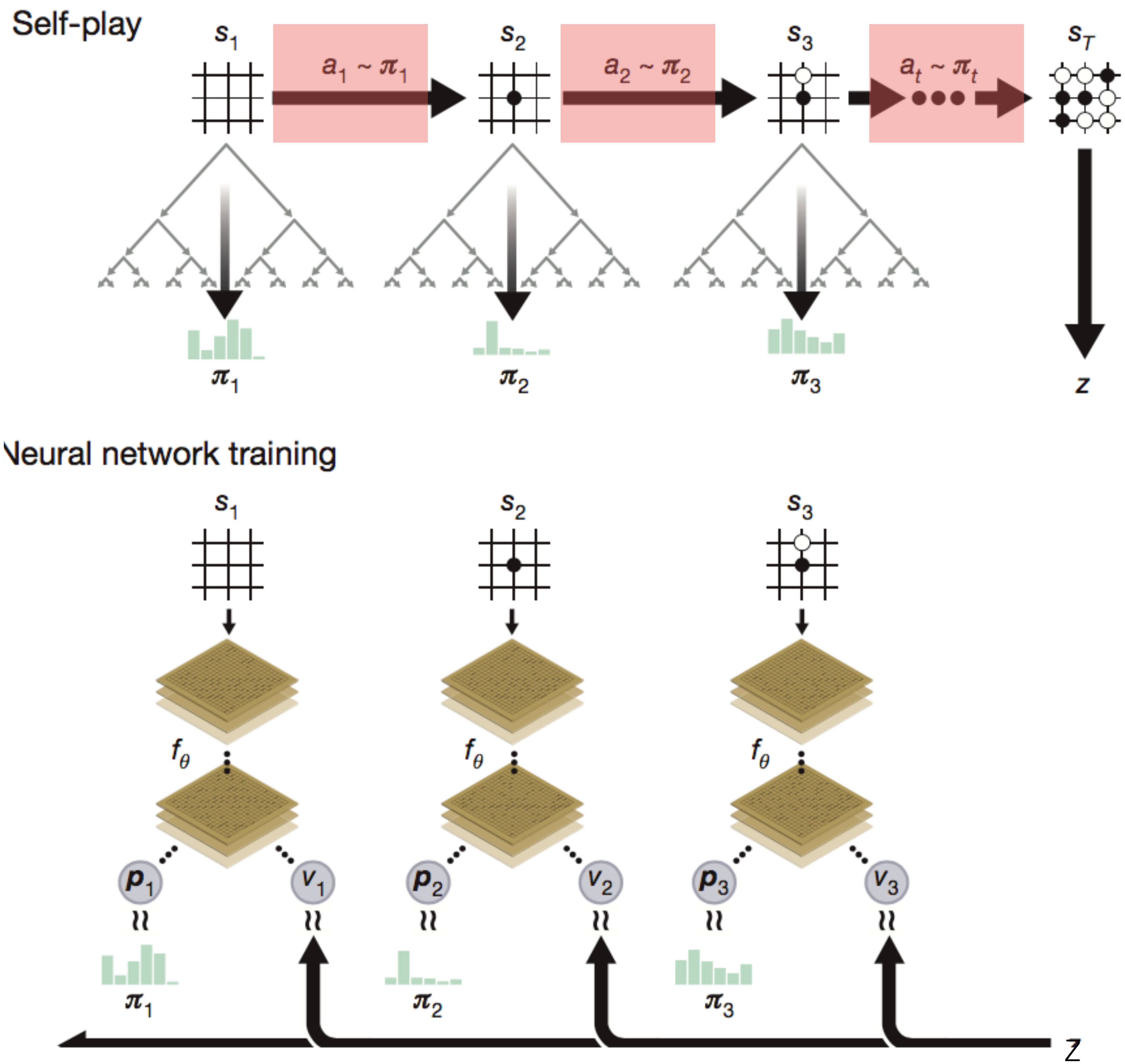
Self-play


Self-play
s_1
s_2
s_3
s_T
$a_1 \sim \pi_1$
$a_2 \sim \pi_2$
$a_t \sim \pi_t$
π_1
π_2
π_3
z

Z

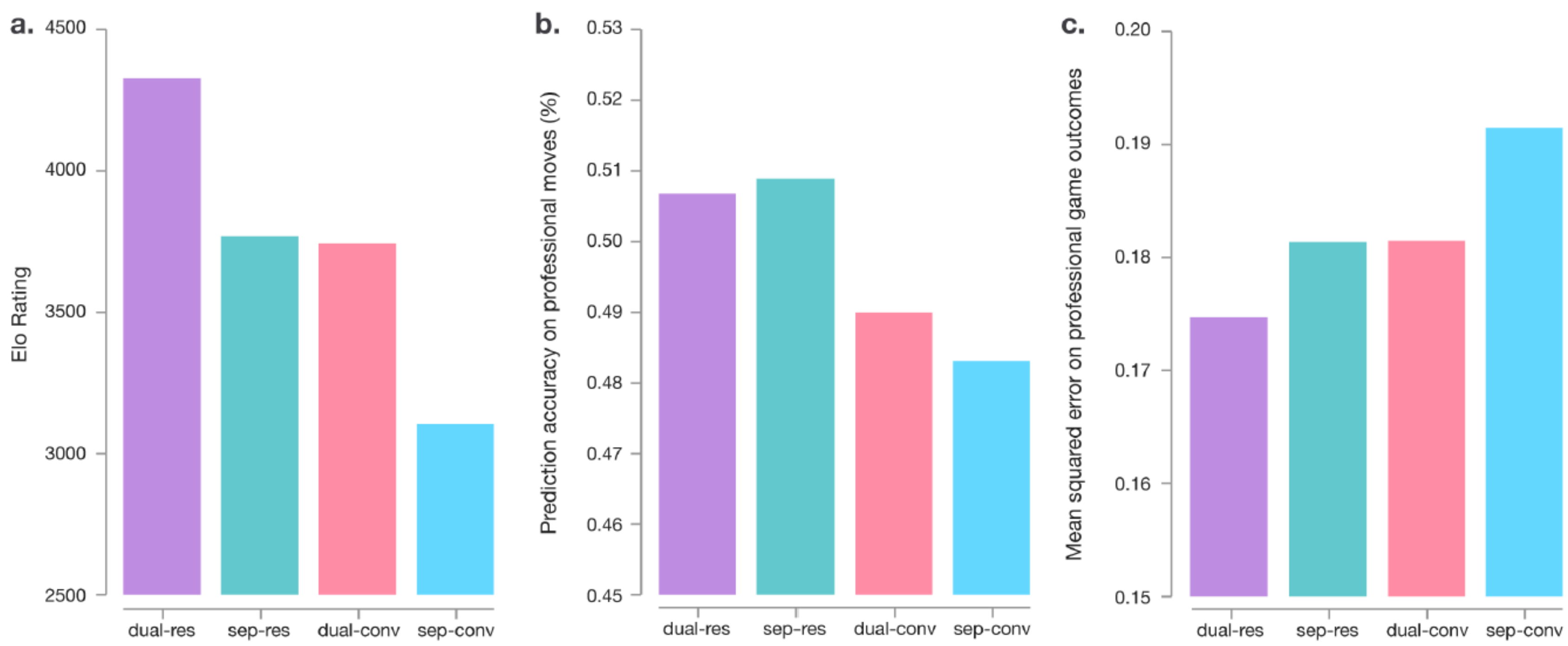
MCTS as policy improvement operator

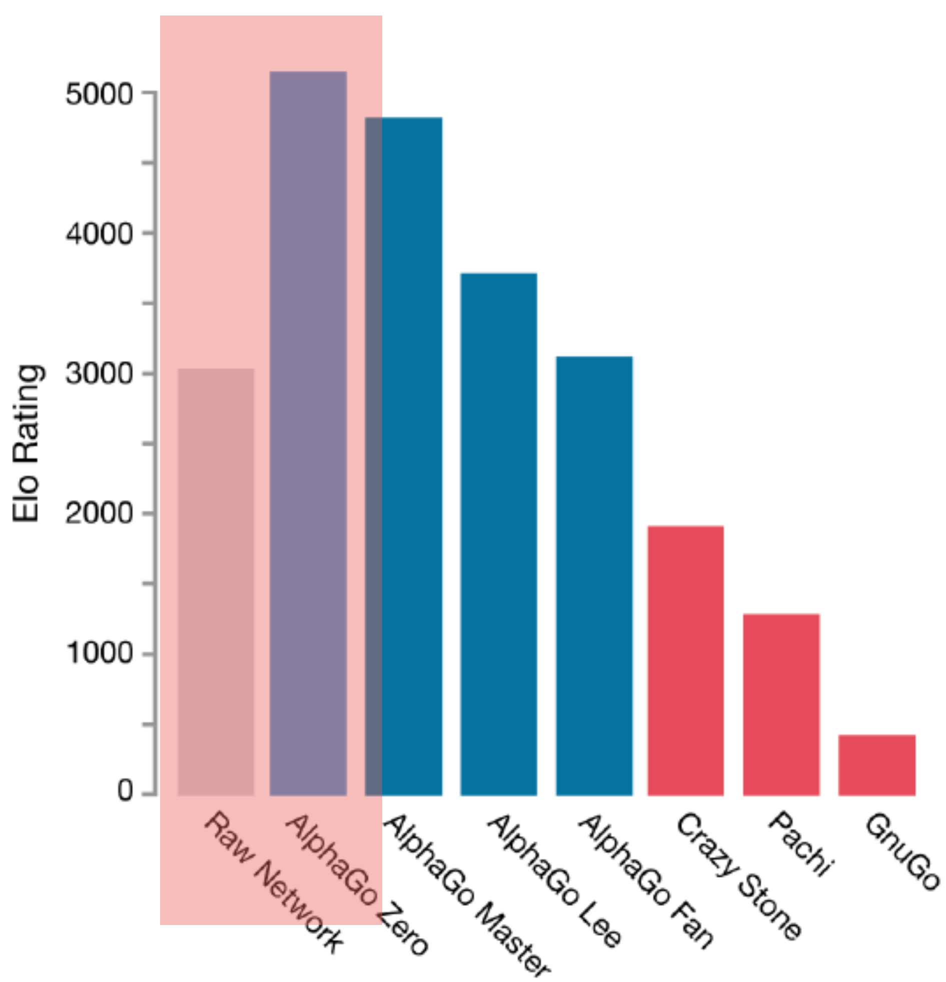
Each of those requires 1600 MC rollouts, i.e., about 0.4 secs thinking time per move.



- Given a policy π_θ, MC rollouts can provide for the root state an action distribution that is better than the initial policy π_θ.
- Train so that the policy network mimics this improved policy
- Train so that the position evaluation network (value function approximator) output matches the outcome

Architectures





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