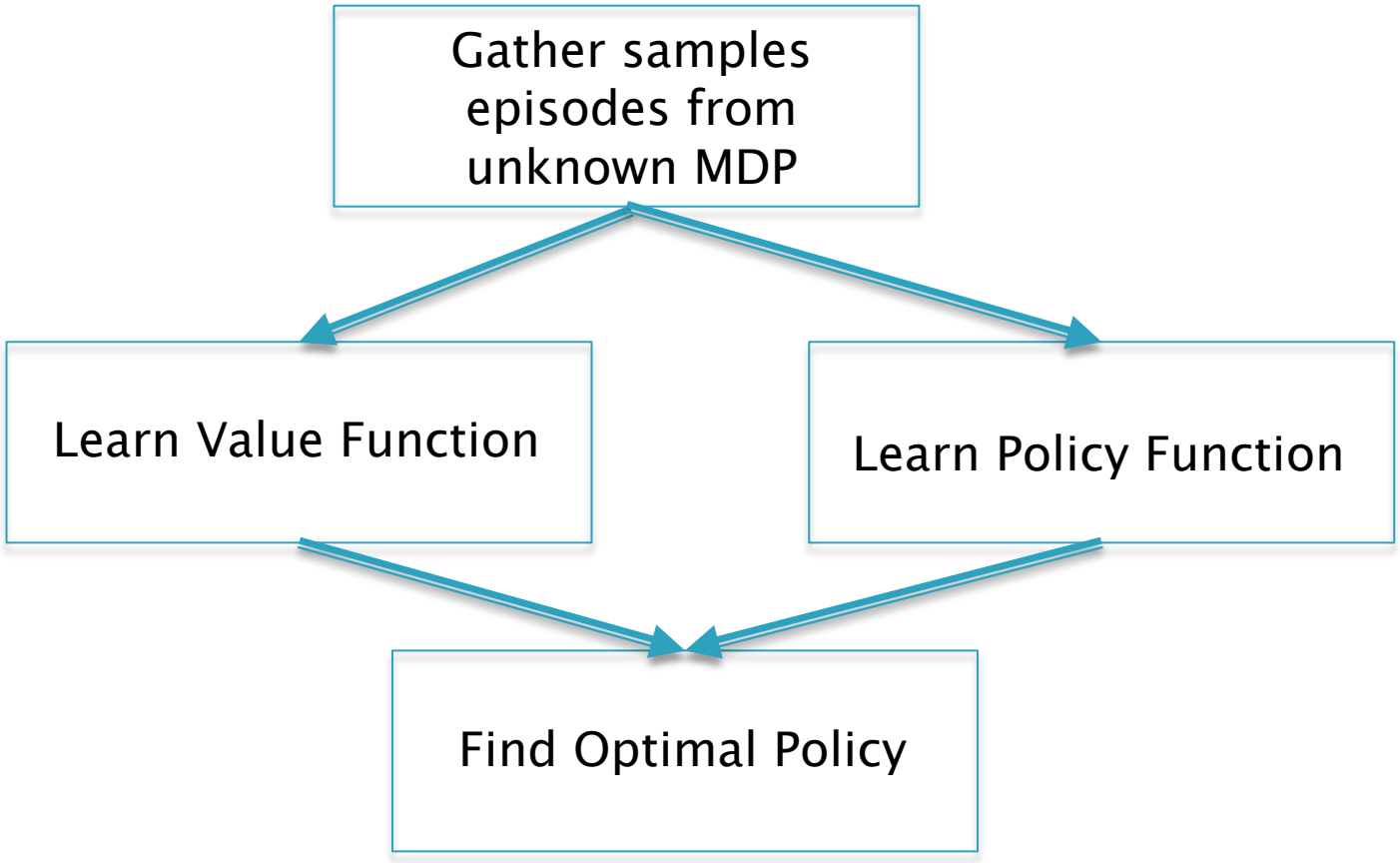
Model based Planning

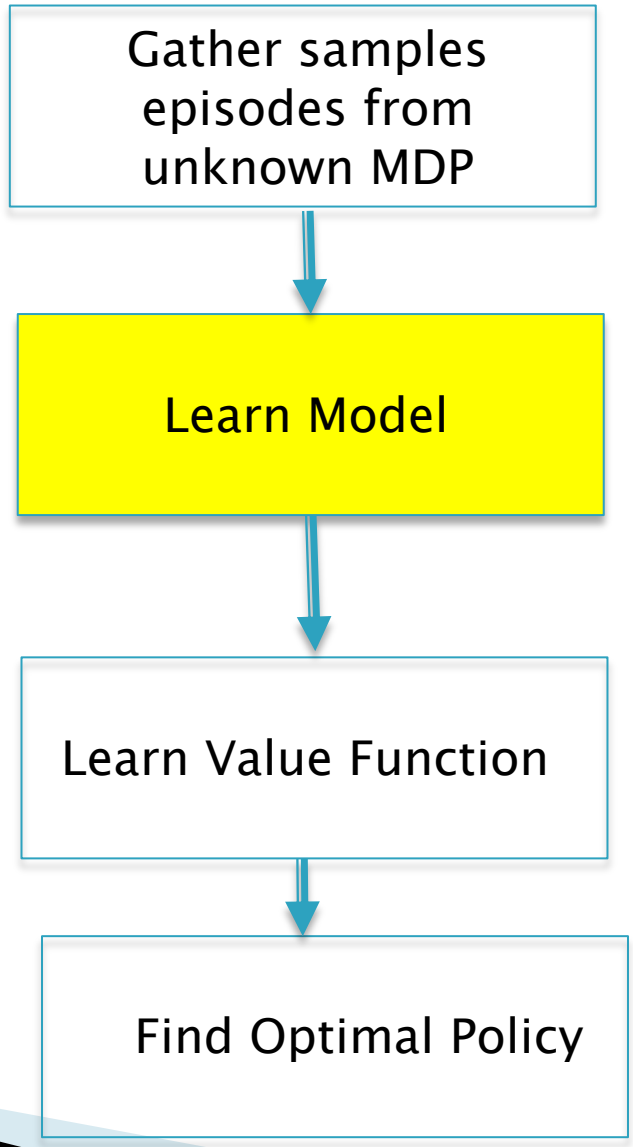
Lecture 9

Subir Varma

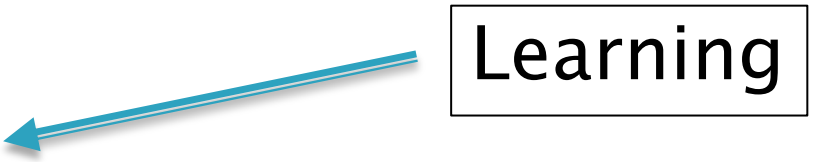
Last Few Lectures: Model Free



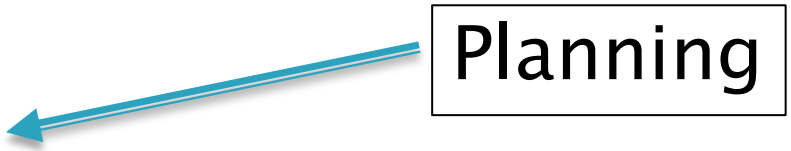
This Lecture: Model Based



Model Based and Model Free RL

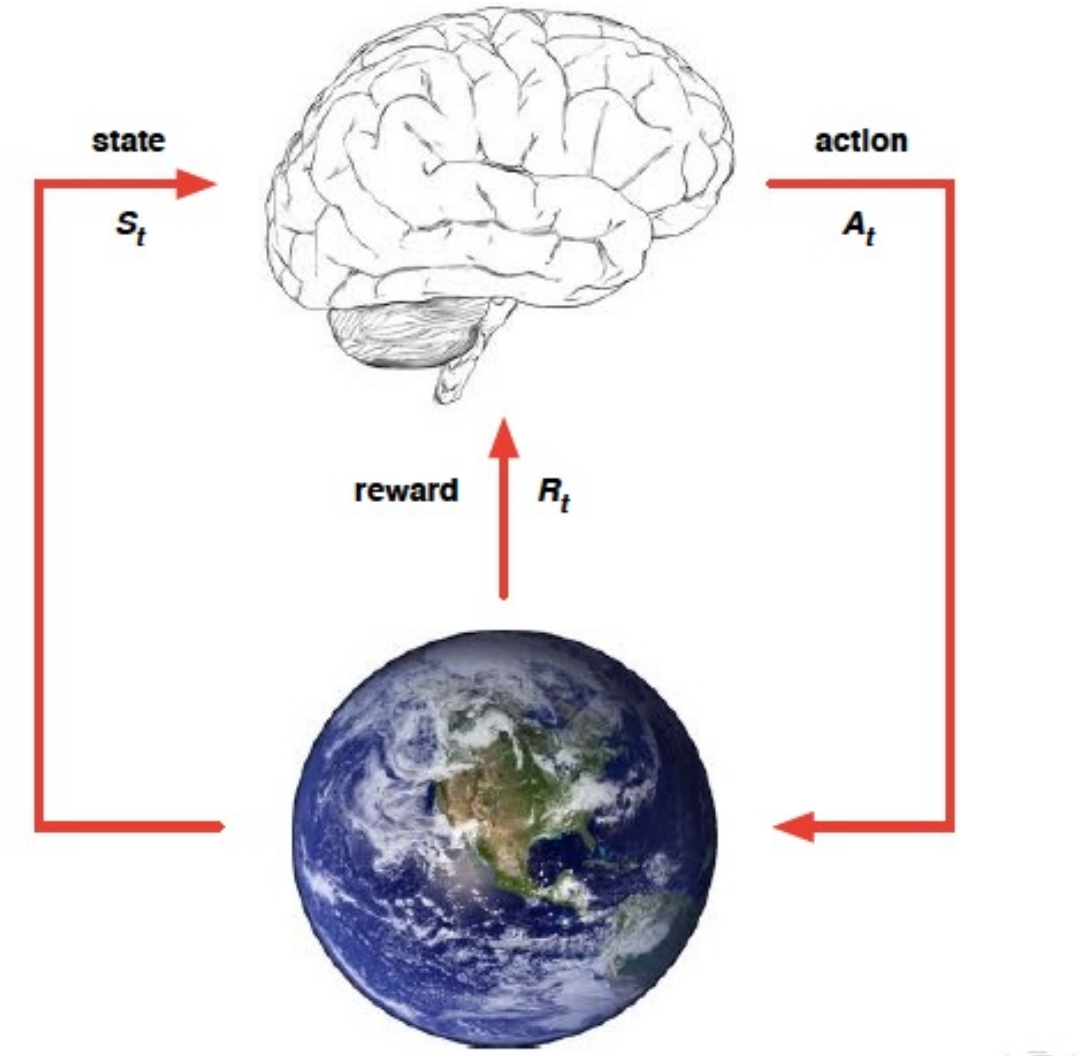


- Model-Free RL
  - No model
  - Learn value function (and/or policy) from experience

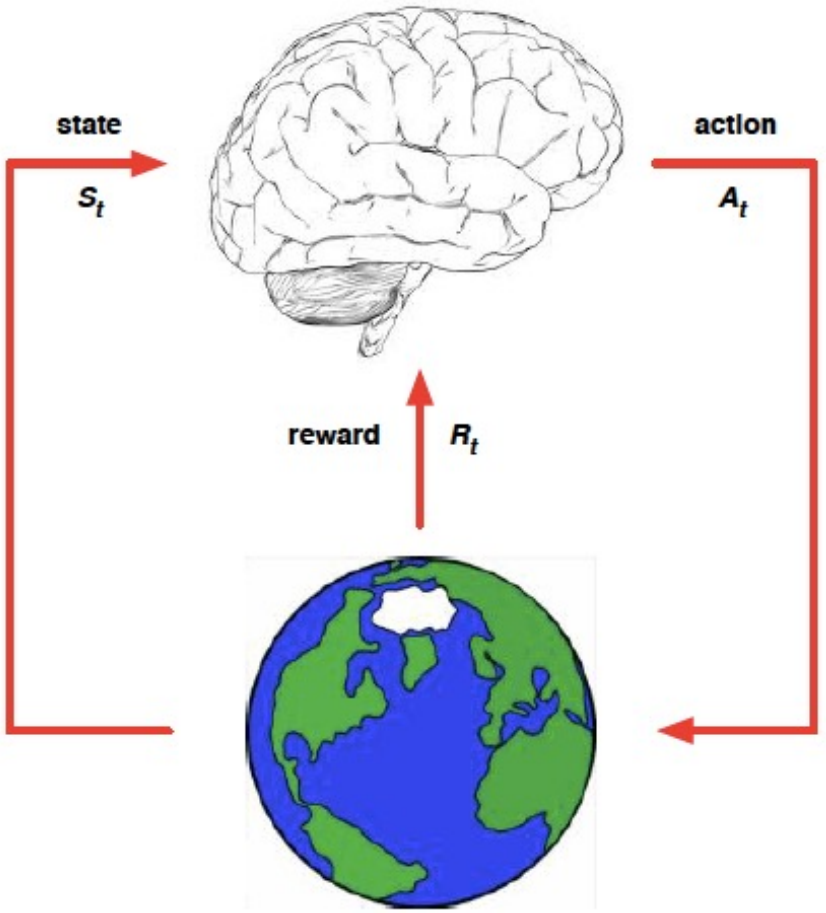


- Model-Based RL
  - Learn a model from experience
  - Plan value function (and/or policy) from model

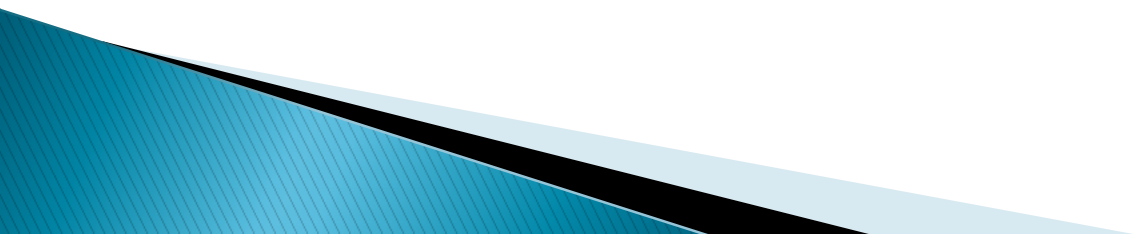
Model Free RL – Learning



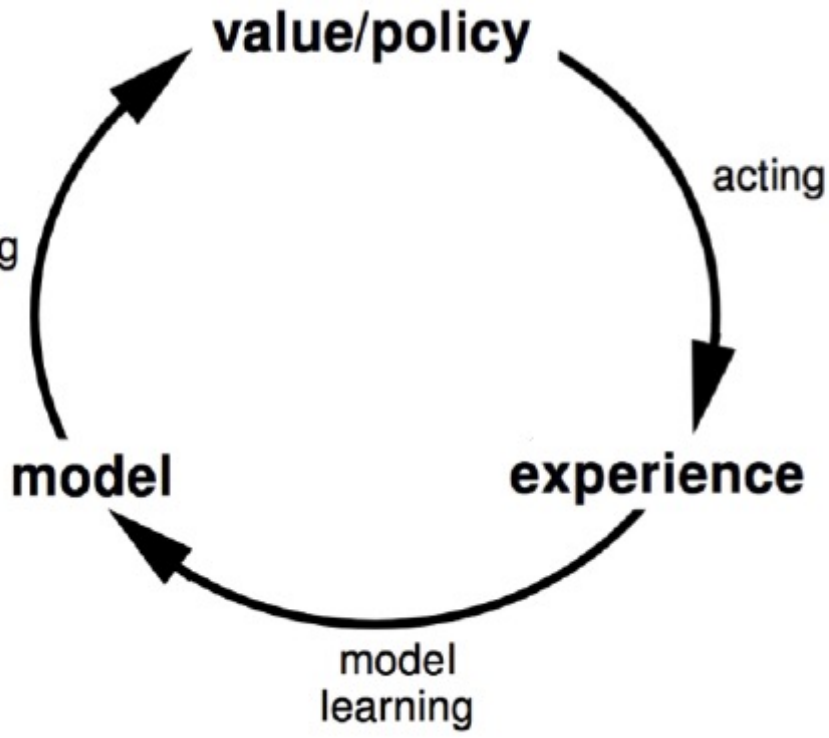
Model based RL – Planning



Model Learning



Model Based RL



Pros-Cons of Model Based RL

Advantages:

- Can efficiently learn model by supervised learning methods
- Can reason about model uncertainty

Disadvantages:

- First learn a model, then construct a value function
  $\Rightarrow$ two sources of approximation error

What is a Model?

- A *model* $\mathcal{M}$ is a representation of an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R} \rangle$, parametrized by η
- We will assume state space $\mathcal{S}$ and action space $\mathcal{A}$ are known
- So a model $\mathcal{M} = \langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$ represents state transitions $\mathcal{P}_\eta \approx \mathcal{P}$ and rewards $\mathcal{R}_\eta \approx \mathcal{R}$

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1} \mid S_t, A_t)$$
$$R_{t+1} = \mathcal{R}_\eta(R_{t+1} \mid S_t, A_t)$$

Model Learning

- Goal: estimate model $\mathcal{M}_\eta$ from experience $\{S_1, A_1, R_2, ..., S_T\}$
- This is a supervised learning problem

$$S_1, A_1 \rightarrow R_2, S_2$$
$$S_2, A_2 \rightarrow R_3, S_3$$
$$\vdots$$
$$S_{T-1}, A_{T-1} \rightarrow R_T, S_T$$

- Learning $s, a \rightarrow r$ is a *regression* problem
- Learning $s, a \rightarrow s'$ is a *density estimation* problem
- Pick loss function, e.g. mean-squared error, KL divergence, ...
- Find parameters η that minimise empirical loss

Table Lookup Model

- Model is an explicit MDP, $\hat{\mathcal{P}}, \hat{\mathcal{R}}$
- Count visits $N(s, a)$ to each state action pair

$$\hat{\mathcal{P}}^a_{s,s'} = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbf{1}(S_t, A_t, S_{t+1} = s, a, s')$$

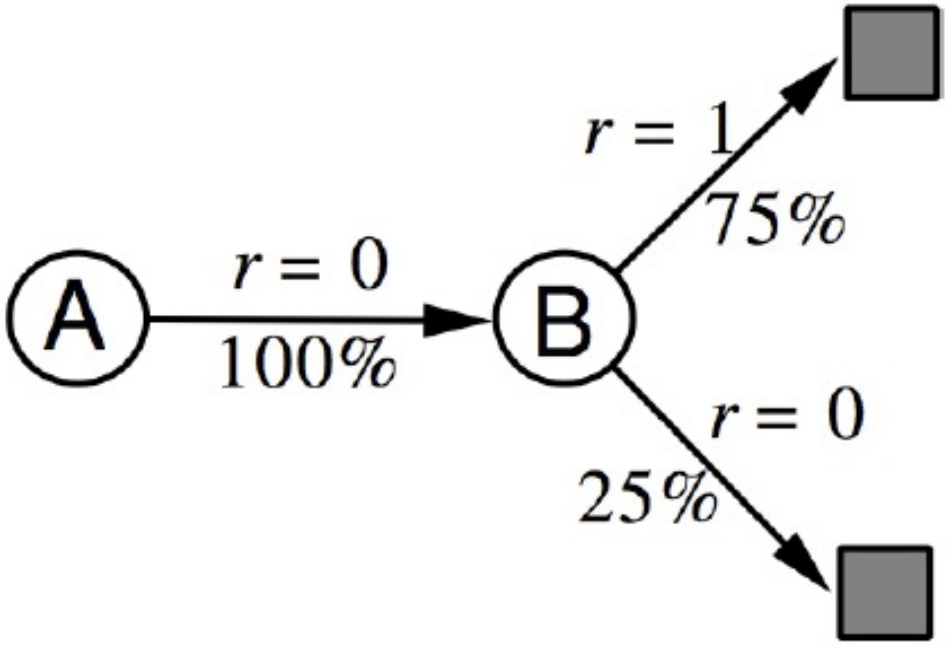
$$\hat{\mathcal{R}}^a_s = \frac{1}{N(s,a)} \sum_{t=1}^{T} \mathbf{1}(S_t, A_t = s, a) R_t$$

- Alternatively
  - At each time-step t, record experience tuple $\langle S_t, A_t, R_{t+1}, S_{t+1} \rangle$
  - To sample model, randomly pick tuple matching $\langle s, a, \cdot, \cdot \rangle$

AB Example

Two states A, B; no discounting; 8 episodes of experience

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0



We have constructed a table lookup model from the experience

Planning with a Model

- Given a model $\mathcal{M}_\eta = \langle \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$
- Solve the MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}_\eta, \mathcal{R}_\eta \rangle$
- Using favourite planning algorithm
  - Value iteration
  - Policy iteration
  - Tree search
  - ...

Lecture 3

Today's Lecture

Sample based Planning

Instead of Using Dynamic Programming

- A simple but powerful approach to planning
- Use the model only to generate samples
- Sample experience from model

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1} \mid S_t, A_t)$$
$$R_{t+1} = \mathcal{R}_\eta(R_{t+1} \mid S_t, A_t)$$

- Apply model-free RL to samples, e.g.:
  - Monte-Carlo control
  - Sarsa
  - Q-learning
- Sample-based planning methods are often more efficient

Tabular Q-Planning

Random-sample one-step tabular Q-planning

Loop forever:

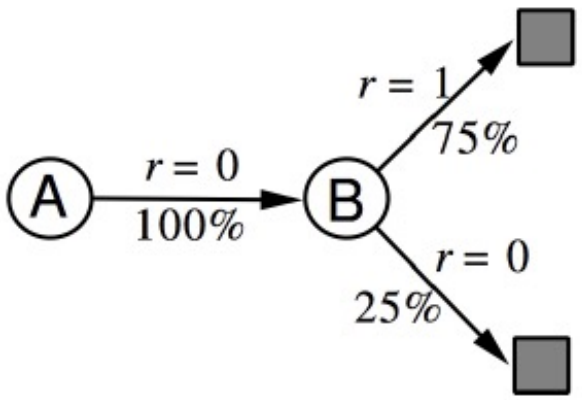
1. Select a state, $S \in \mathcal{S}$, and an action, $A \in \mathcal{A}(S)$, at random
2. Send S, A to a sample model, and obtain
   a sample next reward, R, and a sample next state, S'
3. Apply one-step tabular Q-learning to S, A, R, S':
   $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$

Back to AB Example

- Construct a table-lookup model from real experience
- Apply model-free RL to sampled experience

Real experience

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0



Sampled experience

B, 1
B, 0
B, 1
A, 0, B, 1
B, 1
A, 0, B, 1
B, 1
B, 0

e.g. Monte-Carlo learning: $V(A) = 1, V(B) = 0.75$

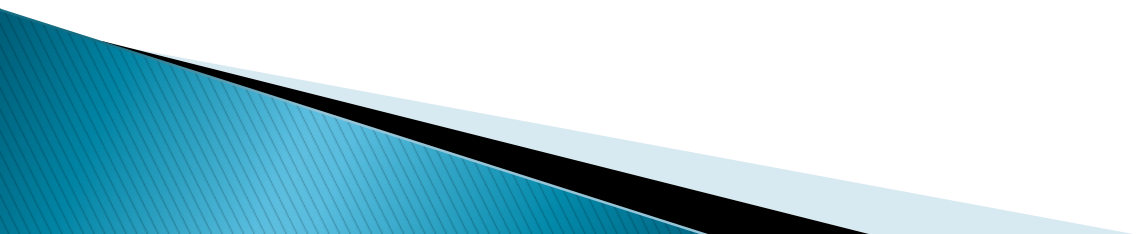
Why would we want to do this?

Types of Planning Algorithms

Planning can be used in two ways:

- Background Planning
  - Use Planning to gradually improve a policy or value function on the basis of simulated experience obtained from a model
  - Planning is not focused on any particular state
  - Best known algorithm: Dyna
- Decision Time Planning
  - Planning focused on finding the best action for a particular state
  - Algorithm run separately for each new state encountered
  - Best known algorithm: Monte Carlo Tree Search (MCTS)

Background Planning: Dyna Algorithm



Real and Simulated Experience

We consider two sources of experience

Real experience Sampled from environment (true MDP)

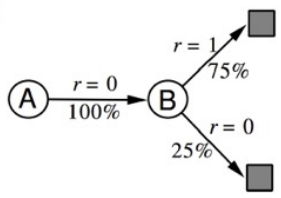
$$S' \sim \mathcal{P}^a_{s,s'}$$
$$R = \mathcal{R}^a_s$$

Simulated experience Sampled from model (approximate MDP)

$$S' \sim \mathcal{P}_\eta(S' \mid S, A)$$
$$R = \mathcal{R}_\eta(R \mid S, A)$$

Real experience

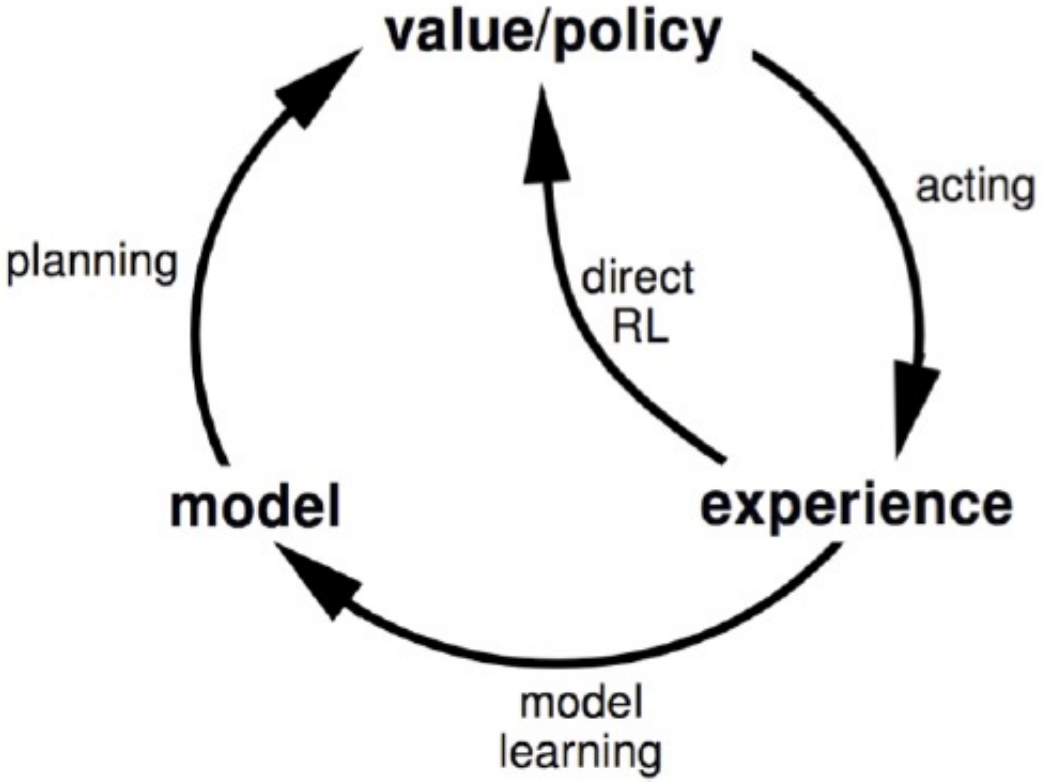
A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0



Sampled experience

B, 1
B, 0
B, 1
A, 0, B, 1
B, 1
A, 0, B, 1
B, 1
B, 0

Dyna Architecture



Dyna-Q Algorithm

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$

Do forever:

(a) $S \leftarrow$ current (nonterminal) state

(b) $A \leftarrow \varepsilon$-greedy(S,Q)

(c) Execute action A; observe resultant reward, R, and state, S'

(d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

(e) $Model(S,A) \leftarrow R, S'$ (assuming deterministic environment)

(f) Repeat n times:

$S \leftarrow$ random previously observed state

$A \leftarrow$ random action previously taken in S

$R, S' \leftarrow Model(S,A)$

$Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$

Dyna-Q on a Simple Maze

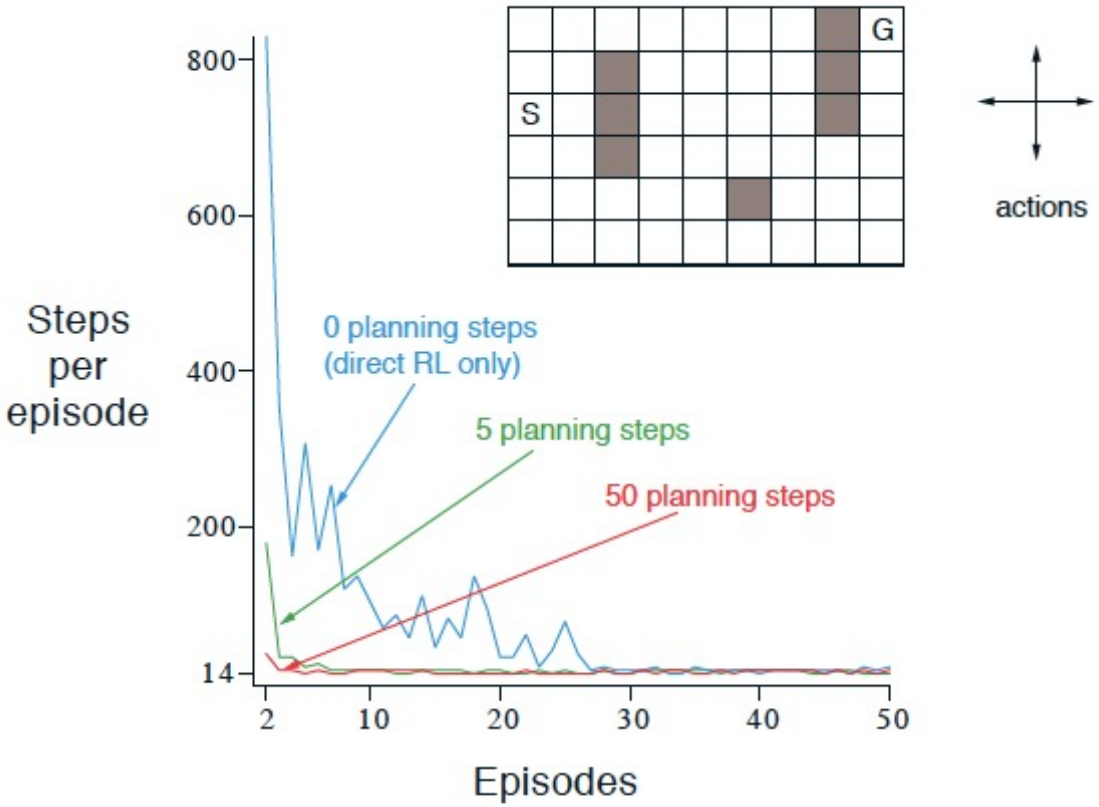


Figure 8.2: A simple maze (inset) and the average learning curves for Dyna-Q agents varying in their number of planning steps (n) per real step. The task is to travel from S to G as quickly as possible.

Dyna-Q on a Simple Maze

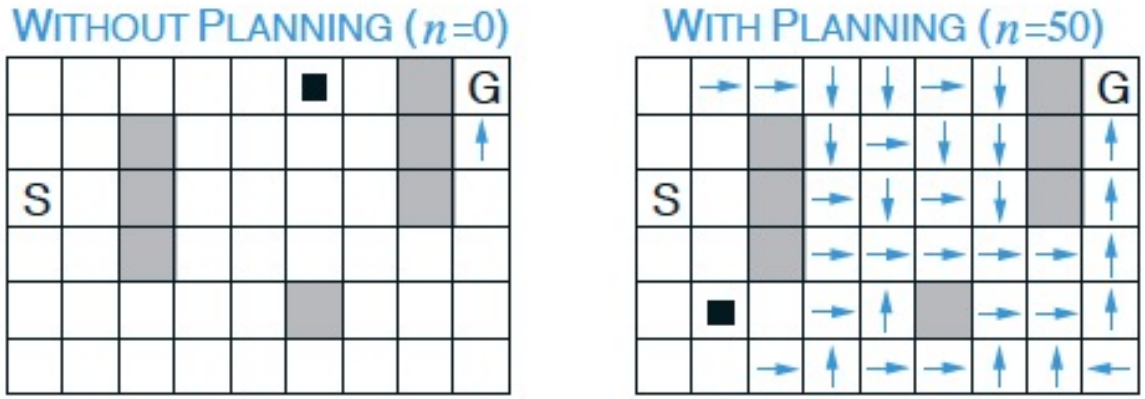
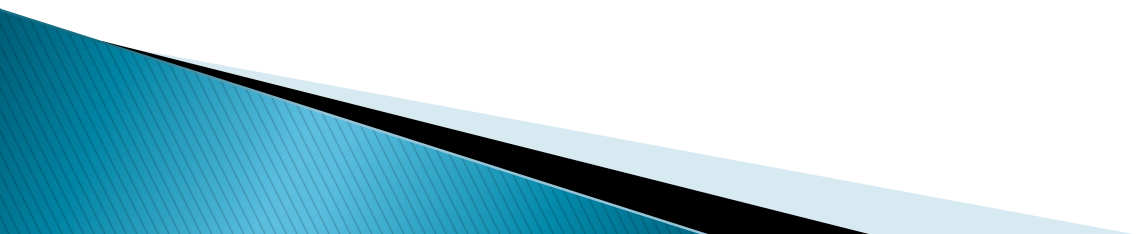


Figure 8.3: Policies found by planning and nonplanning Dyna-Q agents halfway through the second episode. The arrows indicate the greedy action in each state; if no arrow is shown for a state, then all of its action values were equal. The black square indicates the location of the agent. ■

Decision Time Planning
Monte Carlo Tree Search

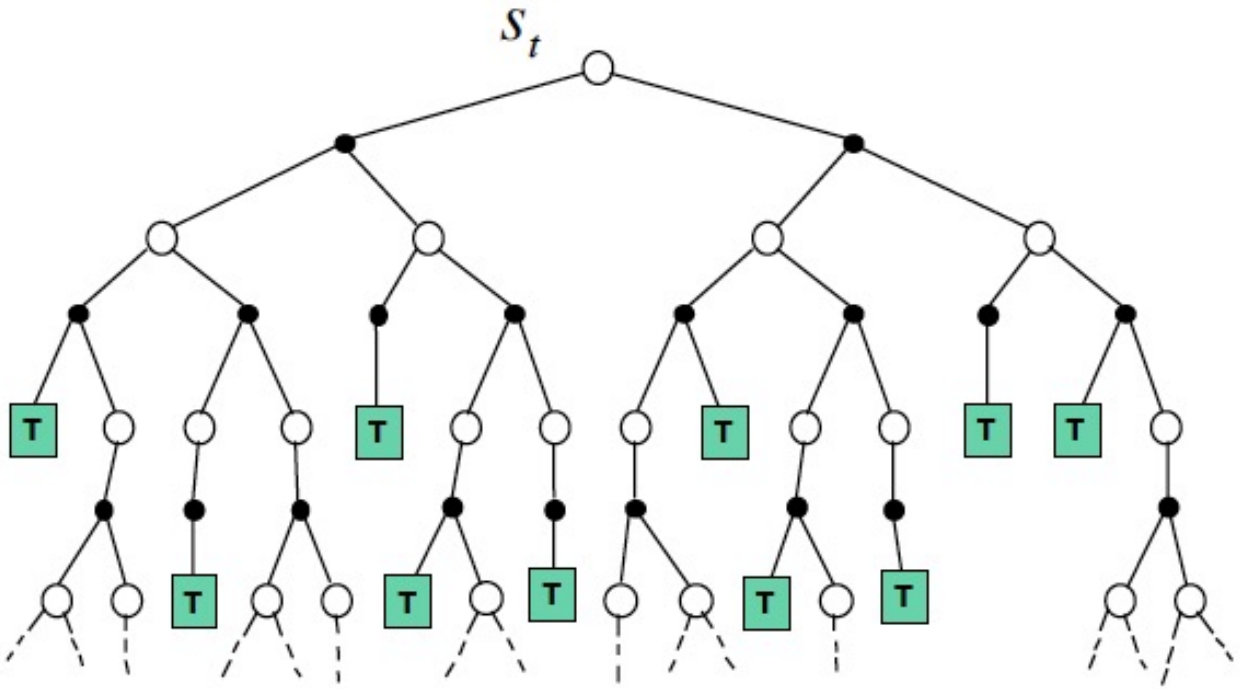


Decision Time Planning

- Decision Time Planning focuses on a particular state
- Agent finds itself in state S
  - Agent begins planning, with the objective of choosing a single best action to take
  - After the action is chosen, the planning process stops
- Planning process re-started for each new state

Forward Search

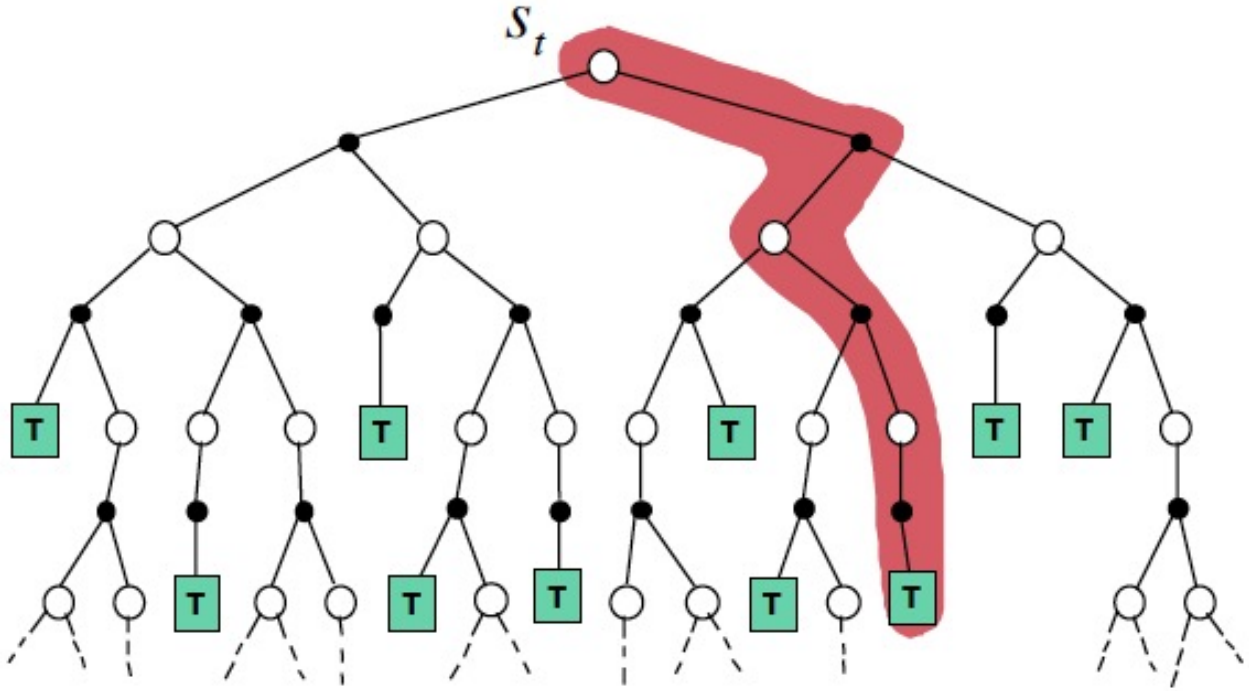
- Forward search algorithms select the best action by lookahead
- They build a search tree with the current state s_t at the root
- Using a model of the MDP to look ahead



- No need to solve whole MDP, just sub-MDP starting from now

Simulation Based Search

- Forward search paradigm using sample-based planning
- Simulate episodes of experience from now with the model
- Apply model-free RL to simulated episodes



Simulation Based Search (cont)

- Simulate episodes of experience from now with the model

$$\{s_t^k, A_t^k, R_{t+1}^k, \ldots, S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu$$

- Apply model-free RL to simulated episodes
  - Monte-Carlo control $\rightarrow$ Monte-Carlo search
  - Sarsa $\rightarrow$ TD search

We also want to make the process of finding the best action as efficient as possible, since we may have limited time to make a decision

Simple Monte Carlo Search

- Given a model $\mathcal{M}_\nu$ and a simulation policy π
- For each action $a \in \mathcal{A}$
  - Simulate K episodes from current (real) state s_t

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

  - Evaluate actions by mean return (Monte-Carlo evaluation)

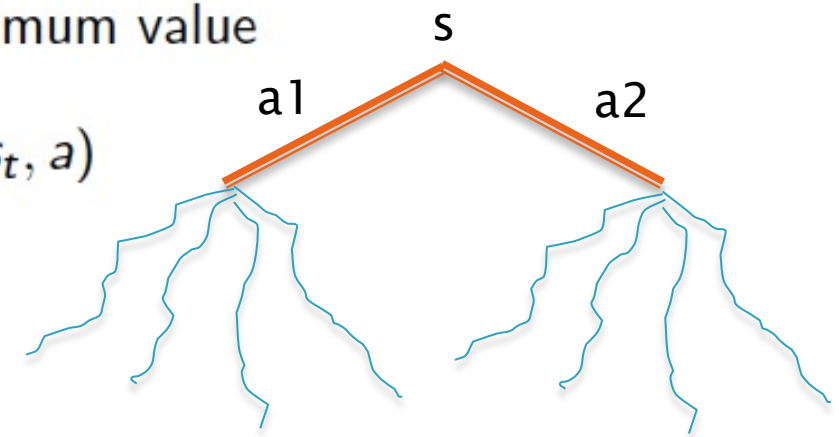
Also called Rollout Policy

Only the root Is evaluated

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

- Select current (real) action with maximum value

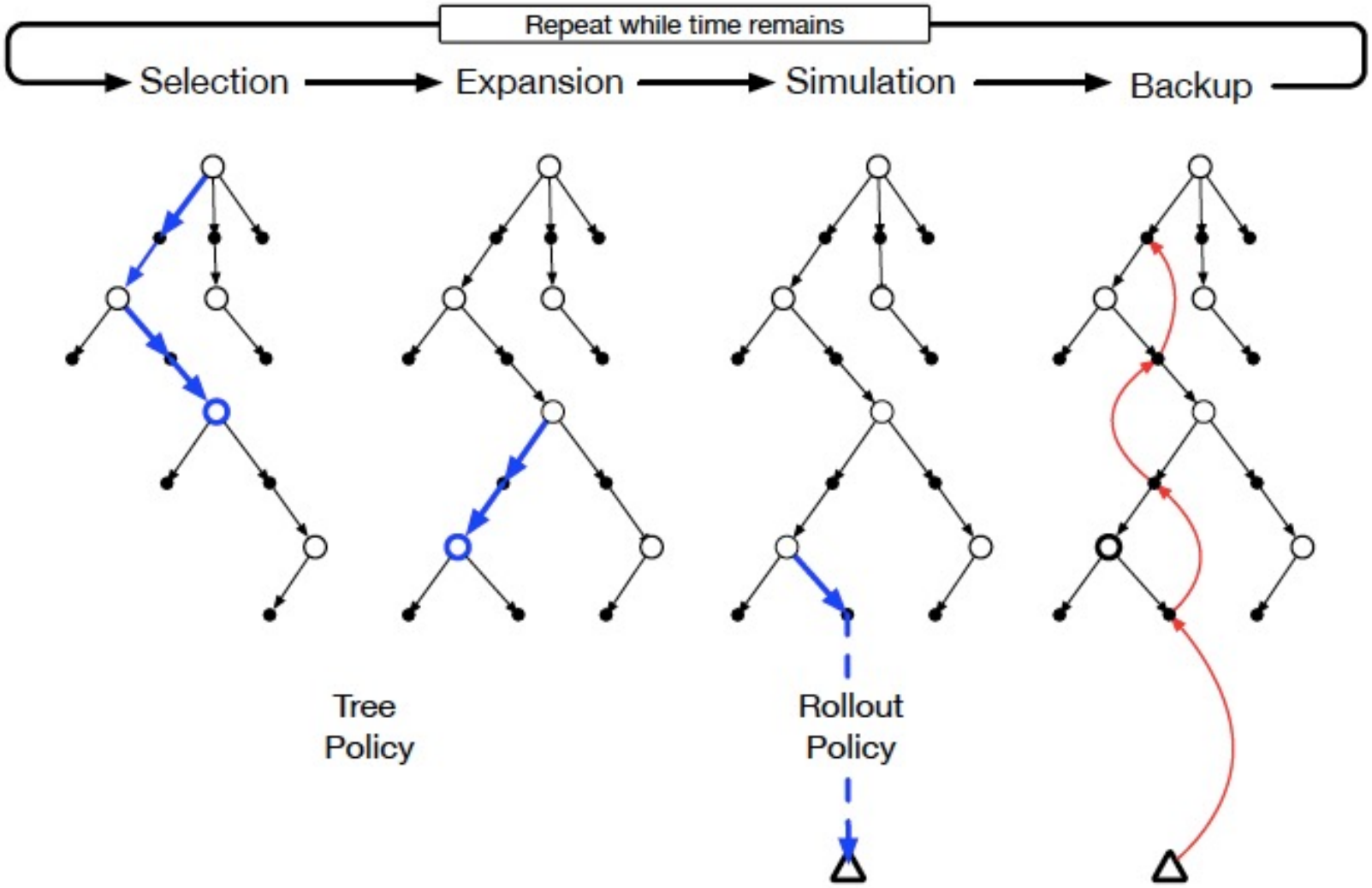
$$a_t = \underset{a \in \mathcal{A}}{\operatorname{argmax}}\ Q(s_t, a)$$



Monte Carlo Tree Search

- Can we do better?
- Main Idea: Evaluate more than just the Root State (but still not all the states)

Monte Carlo Tree Search (MCTS)

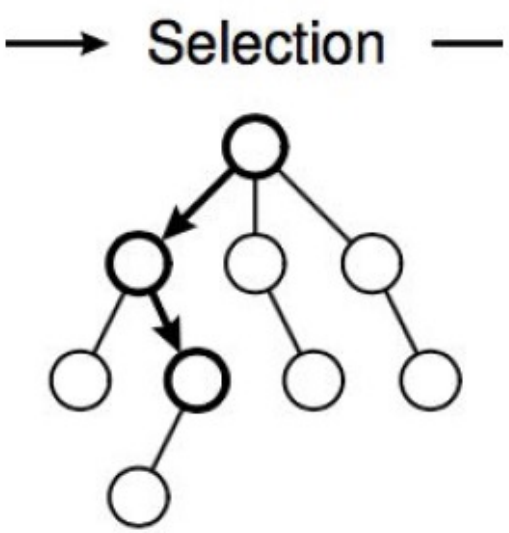


MCTS Policies

- Policies are crucial for how MCTS operates
- Tree policy
  - Used to determine how children are selected
- Default policy
  - Used to determine how simulations are run (ex. randomized)
  - Result of simulation used to update values

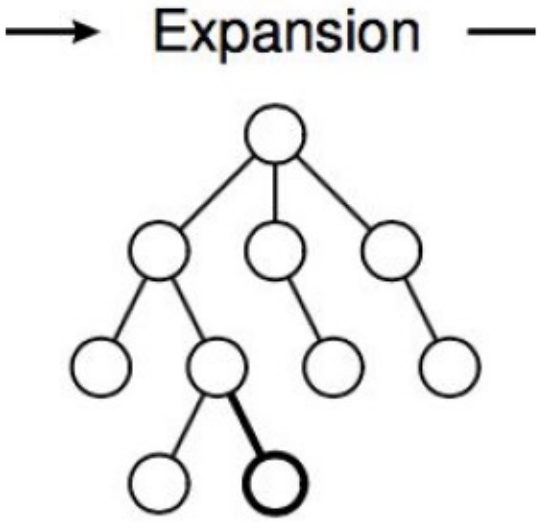
MCTS: Selection

- Start at root node
- Based on Tree Policy select child
- Apply recursively - descend through tree
  - Stop when expandable node is reached
  - *Expandable* -
    - Node that is non-terminal and has unexplored children



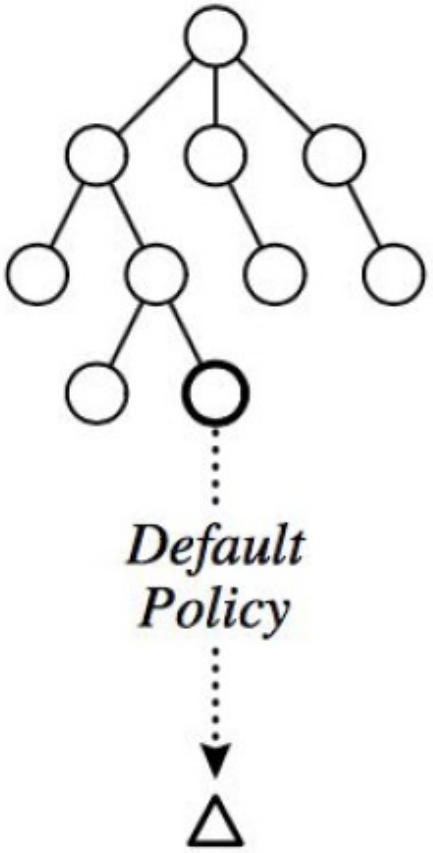
MCTS: Expansion

- Add one or more child nodes to tree
  - Depends on what actions are available for the current position
  - Method in which this is done depends on Tree Policy



MCTS: Simulation

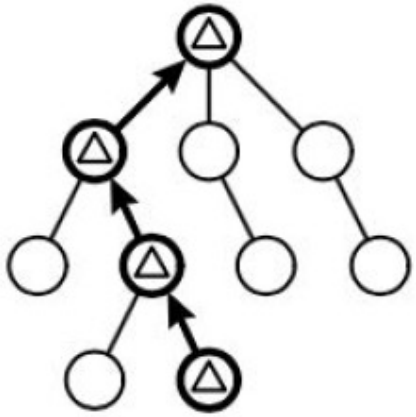
- Runs simulation of path that was selected
- Get position at end of simulation
- Default Policy determines how simulation is run
- Board outcome determines value



MCTS: Backup Computation

- Moves backward through saved path
- Value of Node
  - representative of benefit of going down that path from parent
- Values are updated dependent on board outcome
  - Based on how the simulated game ends, values are updated

→ Backpropagation -



Tree Policy: UCT

Works better than Epsilon greedy!

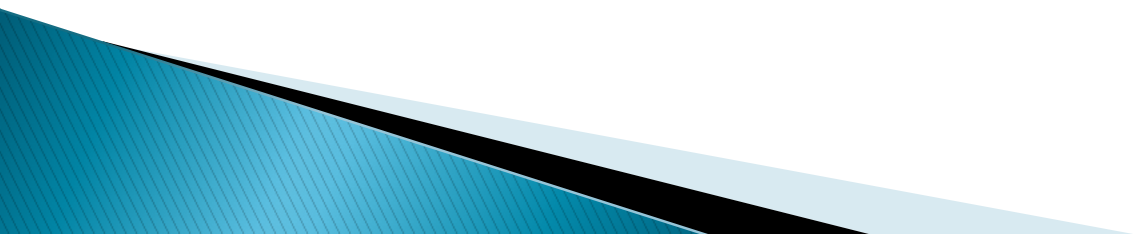
- Selecting Child Node - Multi-Arm Bandit Problem
- UCB1 for each child selection
- UCT - $$UCT = \overline{X}_j + 2C_p\sqrt{\frac{2\ln n}{n_j}}$$

- n - number of times current(parent) node has been visited
- n_j - number of times child j has been visited
- C_p - some constant > 0
- X_j - mean reward of selecting this position
  - [0, 1]

Advantages of MC Tree Search

- Highly selective best-first search
- Evaluates states *dynamically* (unlike e.g. DP)
- Uses sampling to break curse of dimensionality
- Works for "black-box" models (only requires samples)
- Computationally efficient, anytime, parallelisable

Playing GO with MCTS and Deep Reinforcement Learning



ARTICLE

doi:10.1038/nature16961

Mastering the game of Go with deep neural networks and tree search

David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]

The game of Go has long been viewed as the most challenging of classic games for artificial intelligence owing to its enormous search space and the difficulty of evaluating board positions and moves. Here we introduce a new approach to computer Go that uses 'value networks' to evaluate board positions and 'policy networks' to select moves. These deep neural networks are trained by a novel combination of supervised learning from human expert games, and reinforcement learning from games of self-play. Without any lookahead search, the neural networks play Go at the level of state-of-the-art Monte Carlo tree search programs that simulate thousands of random games of self-play. We also introduce a new search algorithm that combines Monte Carlo simulation with value and policy networks. Using this search algorithm, our program AlphaGo achieved a 99.8% winning rate against other Go programs, and defeated the human European Go champion by 5 games to 0. This is the first time that a computer program has defeated a human professional player in the full-sized game of Go, a feat previously thought to be at least a decade away.

All games of perfect information have an optimal value function, $v^*(s)$, which determines the outcome of the game, from every board position or state s, under perfect play by all players. These games may be solved by recursively computing the optimal value function in a search tree containing approximately b^d possible sequences of moves, where b is the game's breadth (number of legal moves per position) and d is its depth (game length). In large games, such as chess ($b \approx 35$, $d \approx 80$)[1] and especially Go ($b \approx 250$, $d \approx 150$)[1], exhaustive search is infeasible[2,3], but the effective search space can be reduced by two general principles. First, 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 $v(s) \approx v^*(s)$ that predicts the outcome from state s. This approach has led to superhuman performance in chess[4], checkers[5] and othello[6], but it was believed to be intractable in Go due to the complexity of the game[7]. Second, the breadth of the search may be reduced by sampling actions from a policy $p(a|s)$ that is a probability distribution over possible moves a in position s. For example, Monte Carlo rollouts[8] search to maximum depth without branching at all, by sampling long sequences of actions for both players from a policy p. Averaging over such rollouts can provide an effective position evaluation, achieving superhuman performance in backgammon[8] and Scrabble[9], and weak amateur level play in Go[10].

Monte Carlo tree search (MCTS)[11,12] uses Monte Carlo rollouts to estimate the value of each state in a search tree. As more simulations are executed, the search tree grows larger and the relevant

policies[13–15] or value functions[16] based on a linear combination of input features.

Recently, deep convolutional neural networks have achieved unprecedented performance in visual domains: for example, image classification[17], face recognition[18], and playing Atari games[19]. They use many layers of neurons, each arranged in overlapping tiles, to construct increasingly abstract, localized representations of an image[20]. We employ a similar architecture for the game of Go. We pass in the board position as a 19×19 image and use convolutional layers to construct a representation of the position. We use these neural networks to reduce the effective depth and breadth of the search tree: evaluating positions using a value network, and sampling actions using a policy network.

We train the neural networks using a pipeline consisting of several stages of machine learning (Fig. 1). We begin by training a supervised learning (SL) policy network p_σ directly from expert human moves. This provides fast, efficient learning updates with immediate feedback and high-quality gradients. Similar to prior work[13,15], we also train a fast policy p_π that can rapidly sample actions during rollouts. Next, we train a reinforcement learning (RL) policy network p_ρ that improves the SL policy network by optimizing the final outcome of games of self-play. This adjusts the policy towards the correct goal of winning games, rather than maximizing predictive accuracy. Finally, we train a value network v_θ that predicts the winner of games played by the RL policy network against itself. Our program AlphaGo efficiently combines the policy and value networks with MCTS.

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

Position Evaluation in Go

- How good is a position s?
- Reward function (undiscounted):

$$R_t = 0 \text{ for all non-terminal steps } t < T$$

$$R_T = \begin{cases} 1 & \text{if Black wins} \\ 0 & \text{if White wins} \end{cases}$$

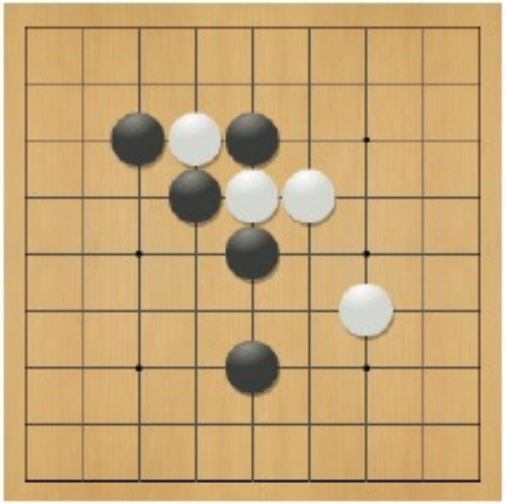
- Policy $\pi = \langle \pi_B, \pi_W \rangle$ selects moves for both players
- Value function (how good is position s):

$$v_\pi(s) = \mathbb{E}_\pi \left[R_T \mid S = s \right] = \mathbb{P} \left[\text{Black wins} \mid S = s \right]$$

$$v_*(s) = \max_{\pi_B} \min_{\pi_W} v_\pi(s)$$

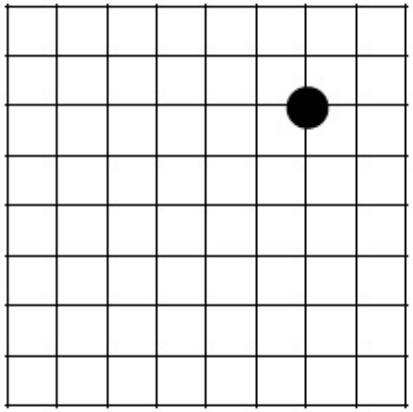
Rules of Go

- 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



The Game of GO

d = 1



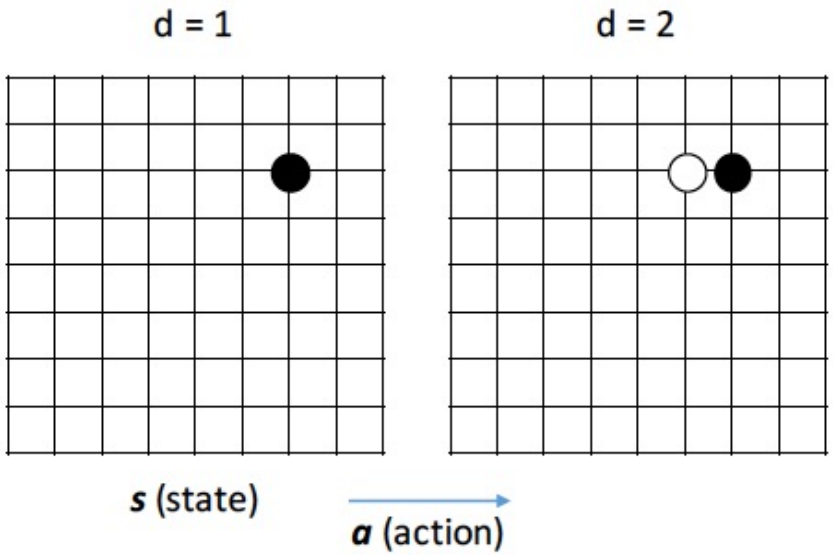
s (state)

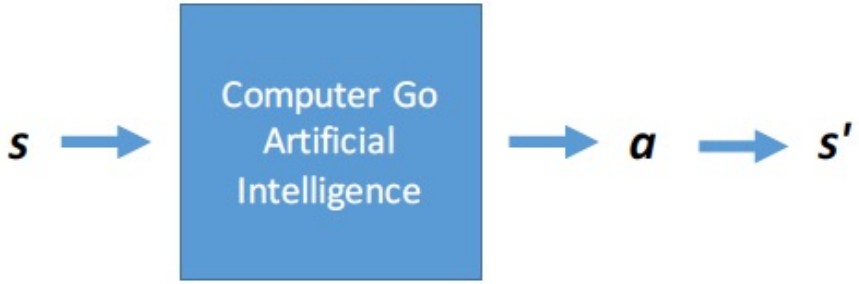
=

$$\begin{pmatrix} 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0 \end{pmatrix}$$

(e.g. we can represent the board into a matrix-like form)

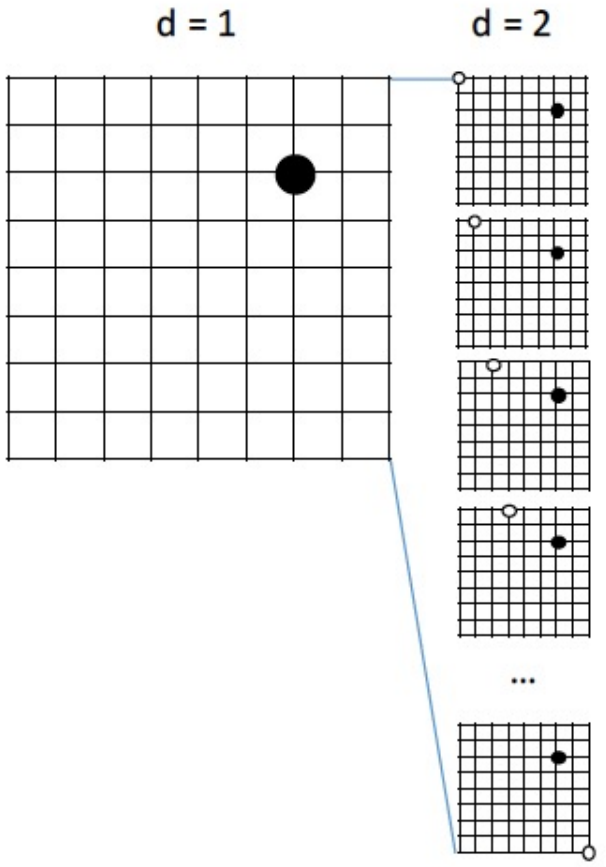
Computer Aided GO





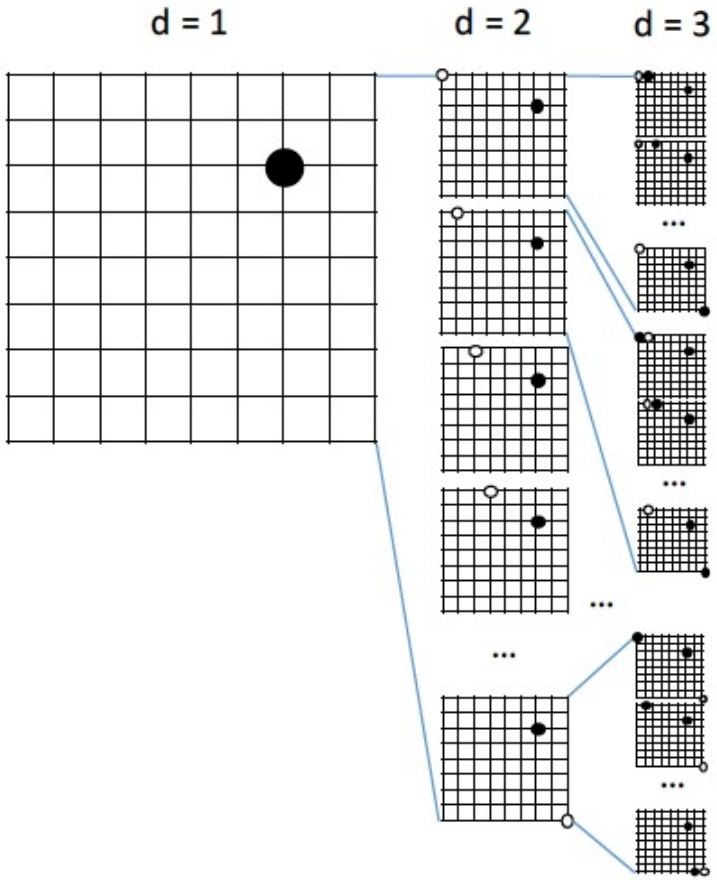
Given ***s***, pick the best ***a***

Computer GO: An Algorithm



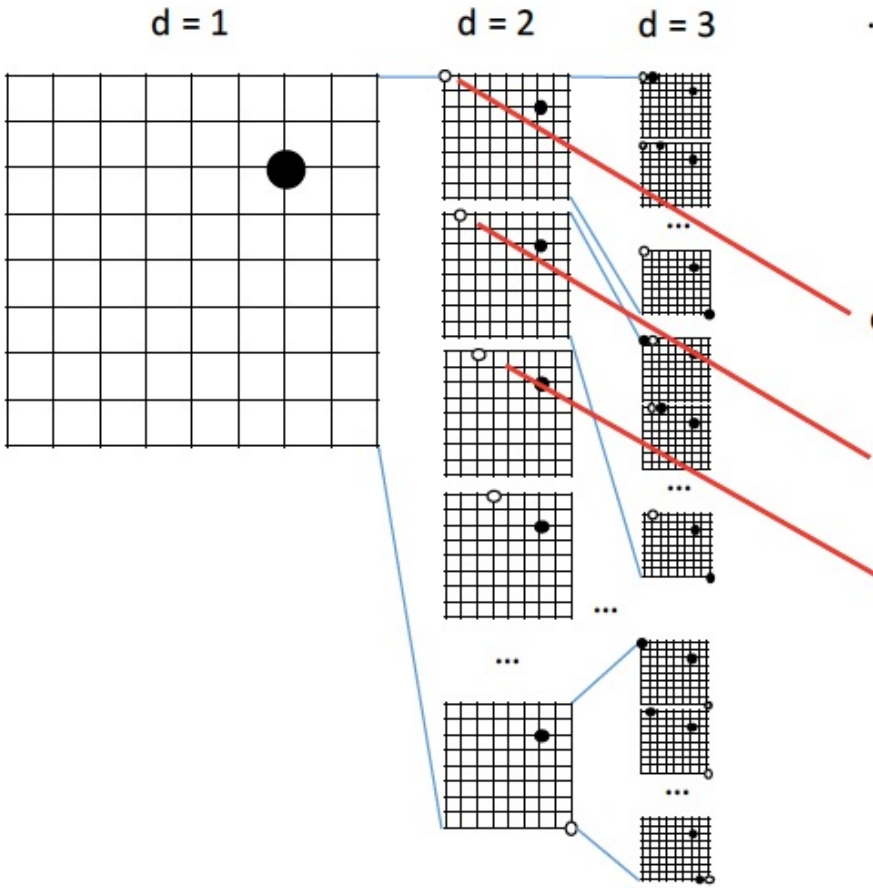
How about simulating all possible board positions?

Computer GO: An Algorithm



Process the simulation until the game ends, then report win / lose results

Computer GO: An Algorithm



Process the simulation until the game ends, then report win / lose results

e.g. it wins 13 times if the next stone gets placed here

37,839 times

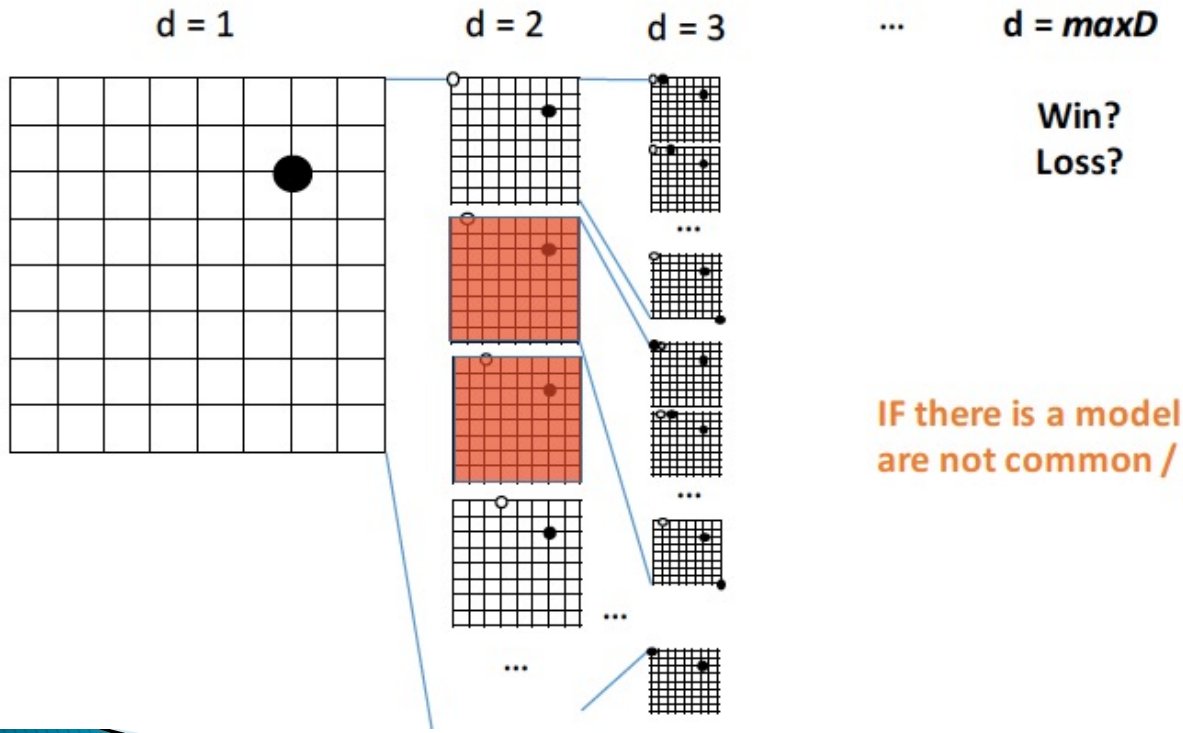
431,320 times

Choose the "next action / stone" that has the most win-counts in the full-scale simulation

This is NOT possible; it is said the possible configurations of the board exceeds the number of atoms in the universe

Reducing Search Space: Breadth

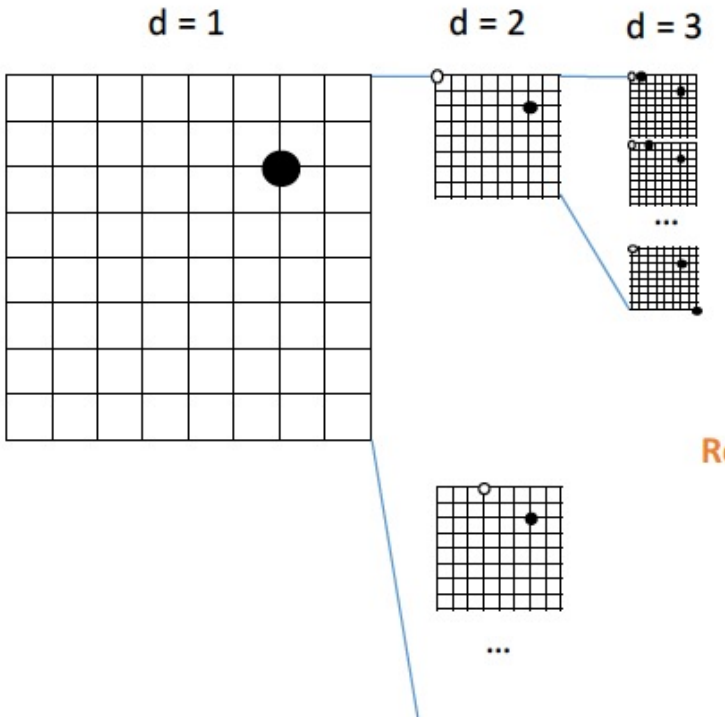
1. Reducing "action candidates" (Breadth Reduction)



IF there is a model that can tell you that these moves are not common / probable (e.g. by experts, etc.) ...

Reducing Search Space: Breadth

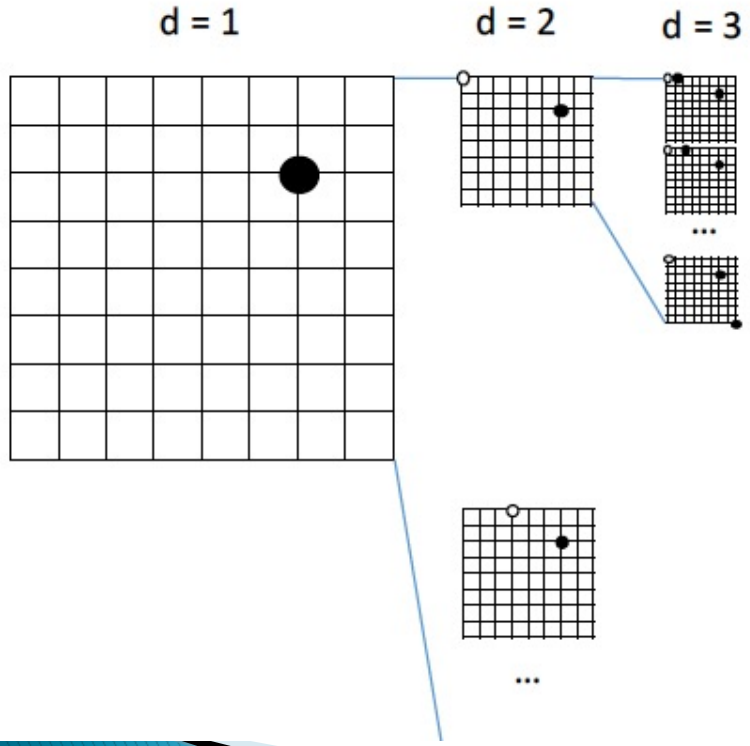
1. Reducing "action candidates" (Breadth Reduction)



Remove these from search candidates in advance (breadth reduction)

Reducing Search Space: Depth

2. Position evaluation ahead of time (Depth Reduction)



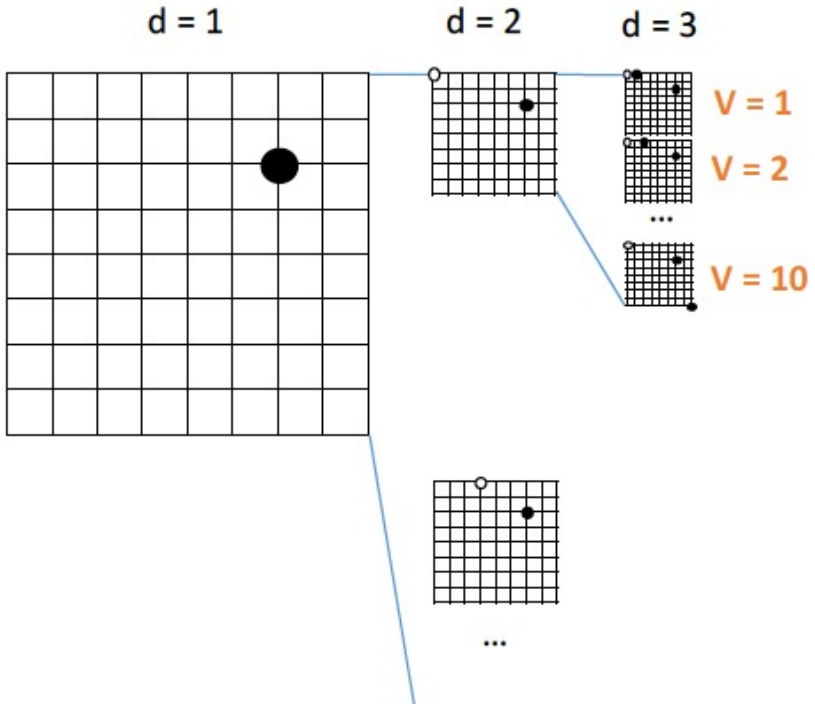
... d = *maxD*

Win?
Loss?

Instead of simulating until the maximum depth ..

Reducing Search Space: Depth

2. Position evaluation ahead of time (Depth Reduction)



IF there is a function that can measure:
V(*s*): "board evaluation of state *s*"

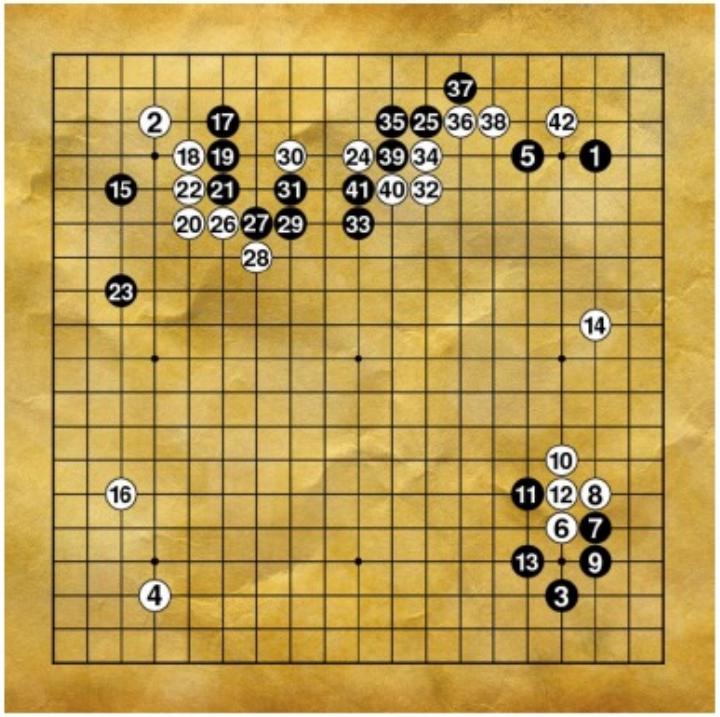
Reducing Action Candidates

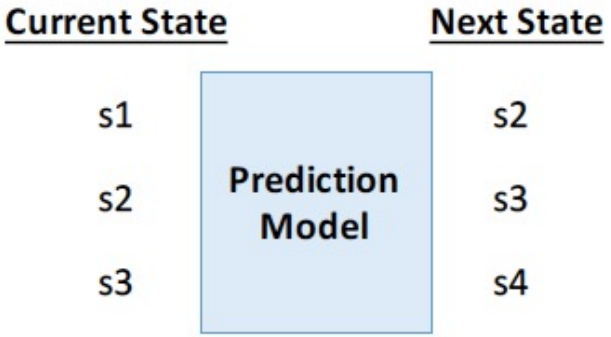
Learning: **P (next action | current state)**

= P (a | s)

Reducing Action Candidates using Supervised Learning

(1) Imitating expert moves (supervised learning)



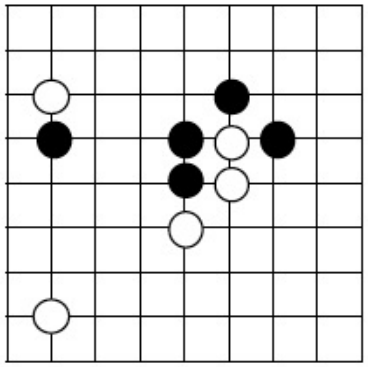


Data: Online Go experts (5~9 dan)
160K games, 30M board positions

Reducing Action Candidates using Supervised Learning

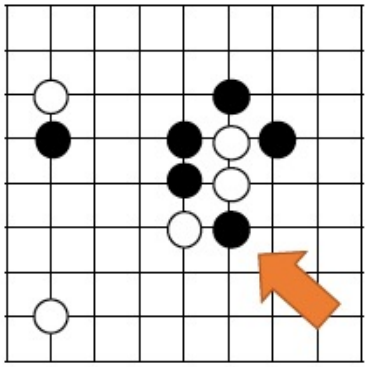
(1) Imitating expert moves (supervised learning)

Current Board



Prediction Model

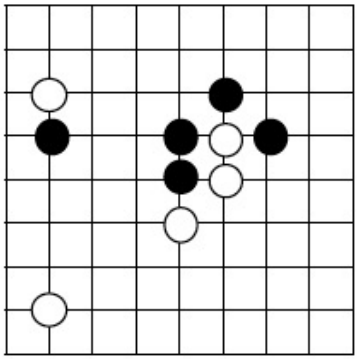
Next Board



Reducing Action Candidates using Supervised Learning

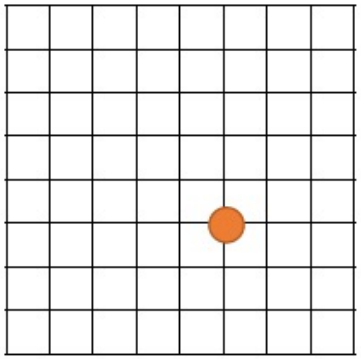
(1) Imitating expert moves (supervised learning)

Current Board



Prediction Model

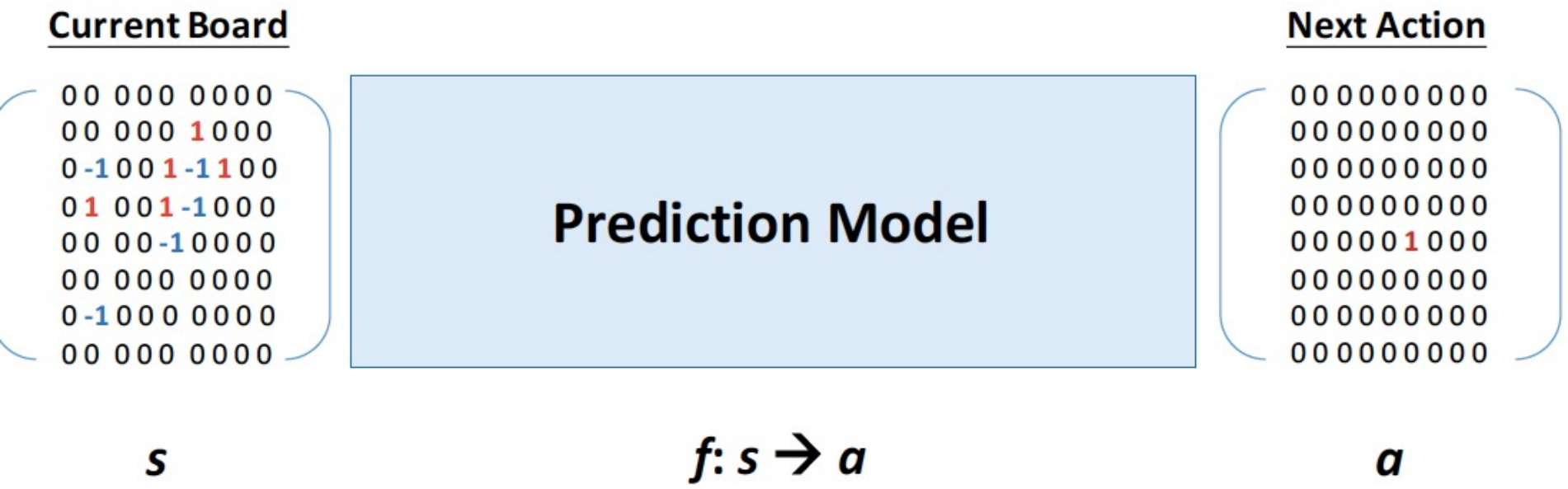
Next Action



There are 19 X 19 = 361
possible actions
(with different probabilities)

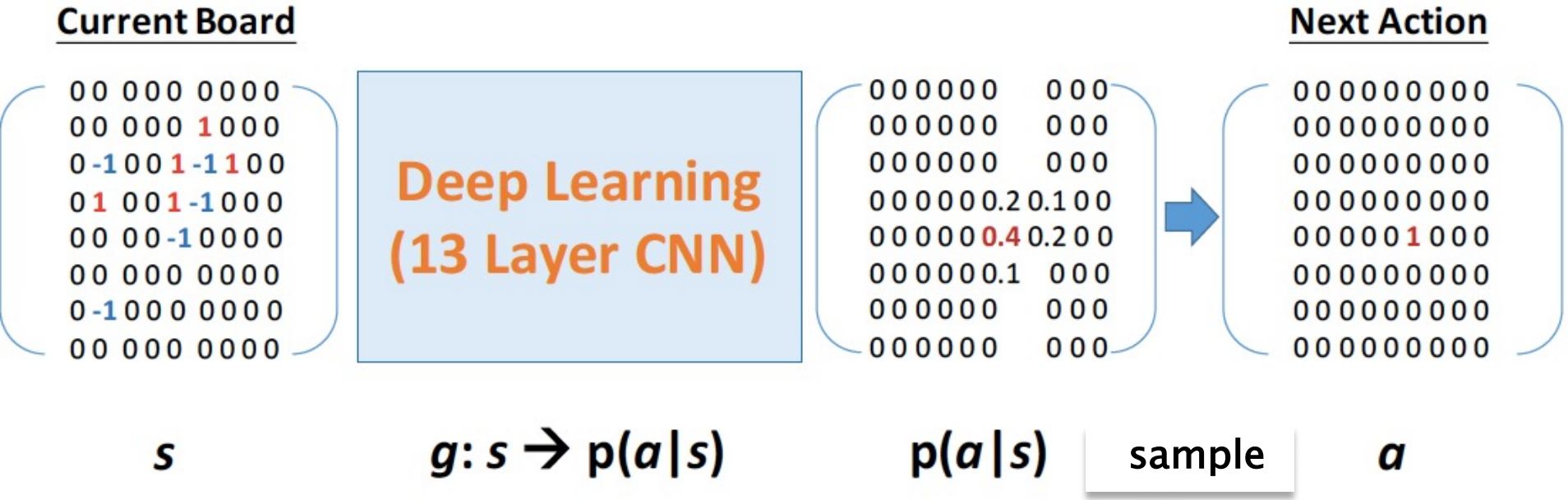
Reducing Action Candidates using Supervised Learning

(1) Imitating expert moves (supervised learning)



Reducing Action Candidates using Supervised Learning

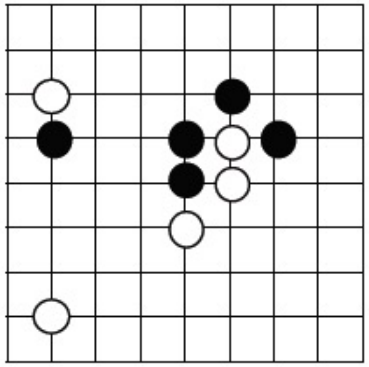
(1) Imitating expert moves (supervised learning)



Reducing Action Candidates using Policy Gradients

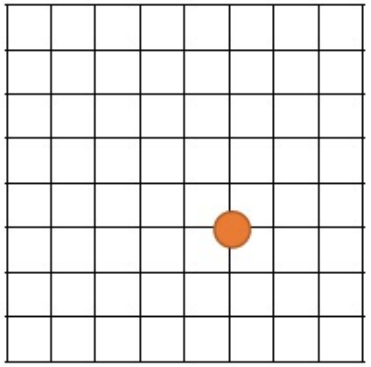
(1) Imitating expert moves (supervised learning)

Current Board



Expert Moves Imitator Model (w/ CNN)

Next Action



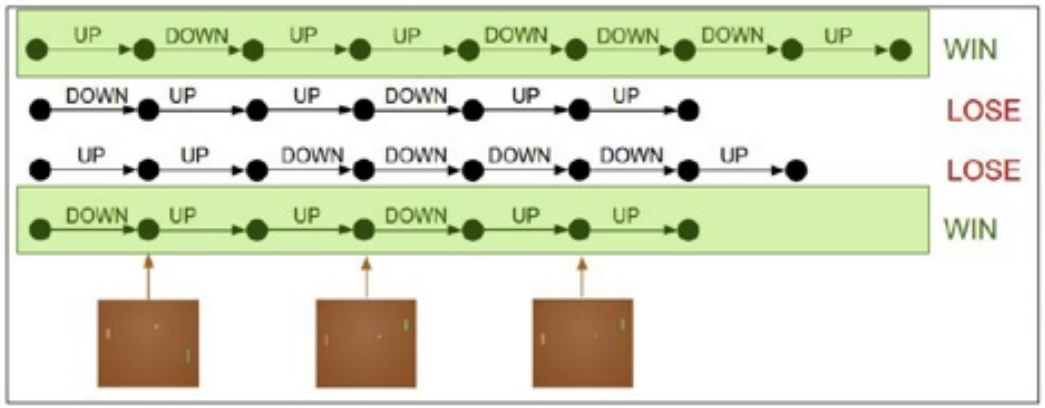
Training: $\Delta\sigma \propto \frac{\partial \log p_\sigma(a|s)}{\partial \sigma}$

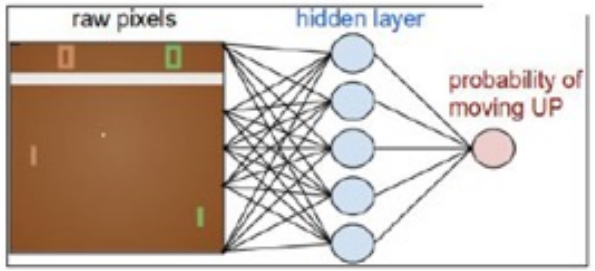
Reducing Action Candidates using Policy Gradients

(2) Improving through self-plays (reinforcement learning)

Improving by playing against itself

Expert Moves Imitator Model (w/ CNN) **vs** **Expert Moves Imitator Model (w/ CNN)**

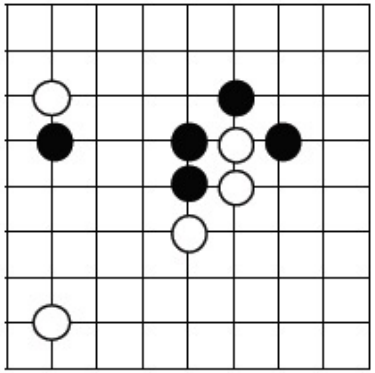




Reducing Action Candidates using Policy Gradients

(2) Improving through self-plays (reinforcement learning)

Board position



Expert Moves Imitator Model (w/ CNN)

win/loss

Loss

z = -1

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

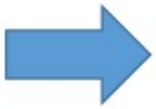
Reducing Action Candidates using Policy Gradients

(2) Improving through self-plays (reinforcement learning)

Older models vs. newer models

Updated Model ver 1.1 **VS** **Updated Model ver 1.3**

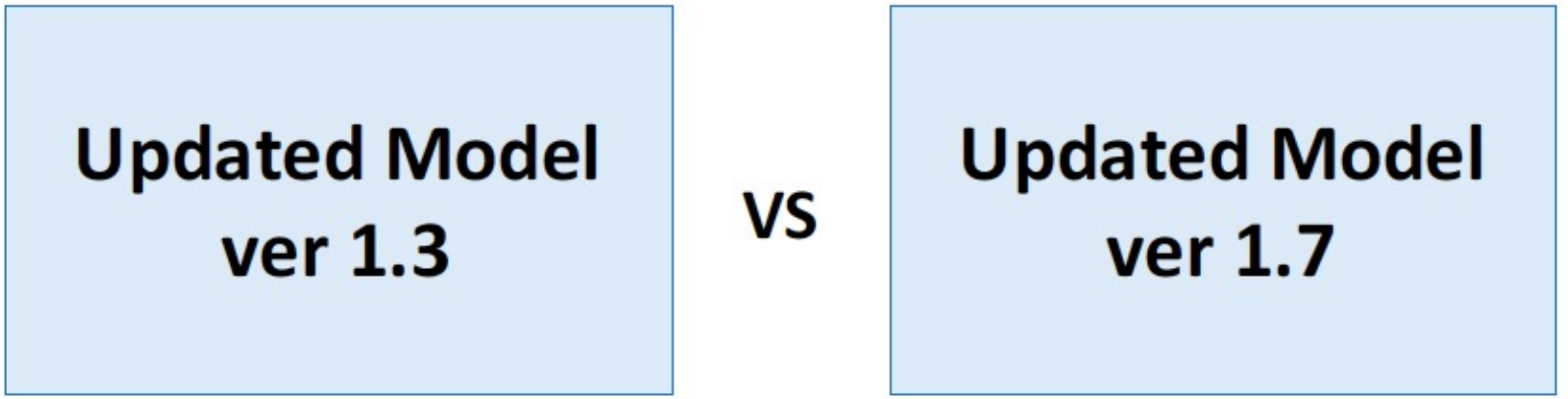
It uses the same topology as the expert moves imitator model, and just uses the updated parameters



Return: board positions, win/lose info

Reducing Action Candidates using Policy Gradients

(2) Improving through self-plays (reinforcement learning)



Return: board positions, win/lose info

Reducing Action Candidates using Policy Gradients

(2) Improving through self-plays (reinforcement learning)

Updated Model ver 3204.1

VS

Updated Model ver 46235.2

Return: board positions, win/lose info

Reducing Action Candidates using Policy Gradients

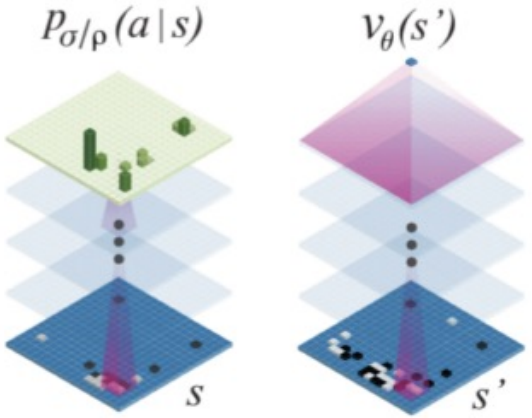
(2) Improving through self-plays (reinforcement learning)

Expert Moves Imitator Model **VS** **Updated Model ver 1,000,000**

The final model wins 80% of the time when playing against the first model

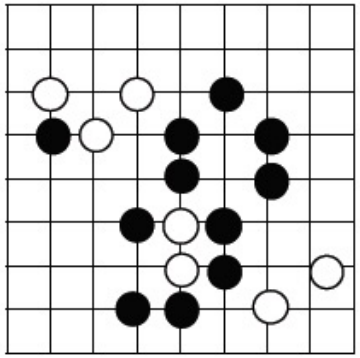
At this point we have a Policy Network that supplies Good Moves most of the time!
Solves Breadth Reduction Problem

Board Evaluation Function



Adds a regression layer to the model
Predicts values between 0~1
Close to 1: a good board position
Close to 0: a bad board position

Board Position



Updated Model ver 1,000,000

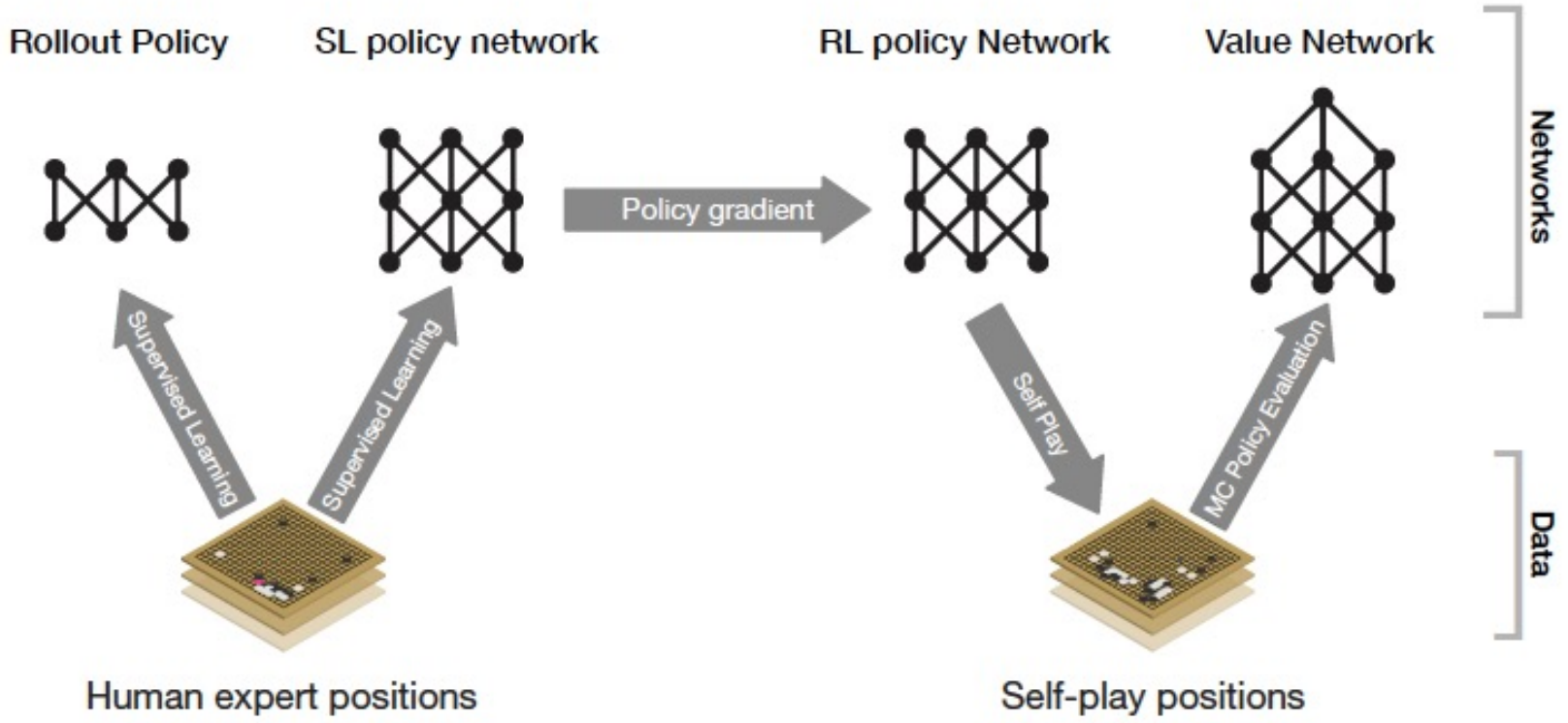
Value Prediction Model (Regression)

Win / Loss

Win (0~1)

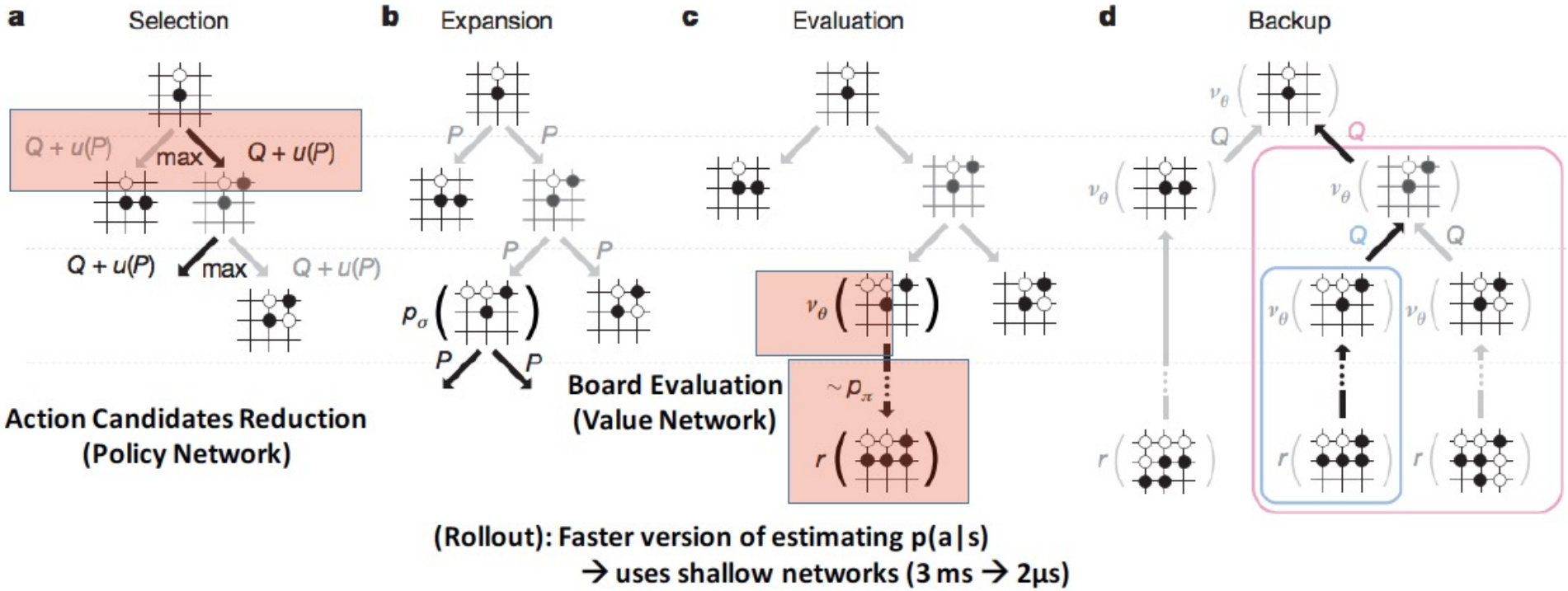
Training: $\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}(z - v_\theta(s))$

Policy and Value Networks Used



All these Networks are developed before the actual game starts!

Applying Monte Carlo Tree Search

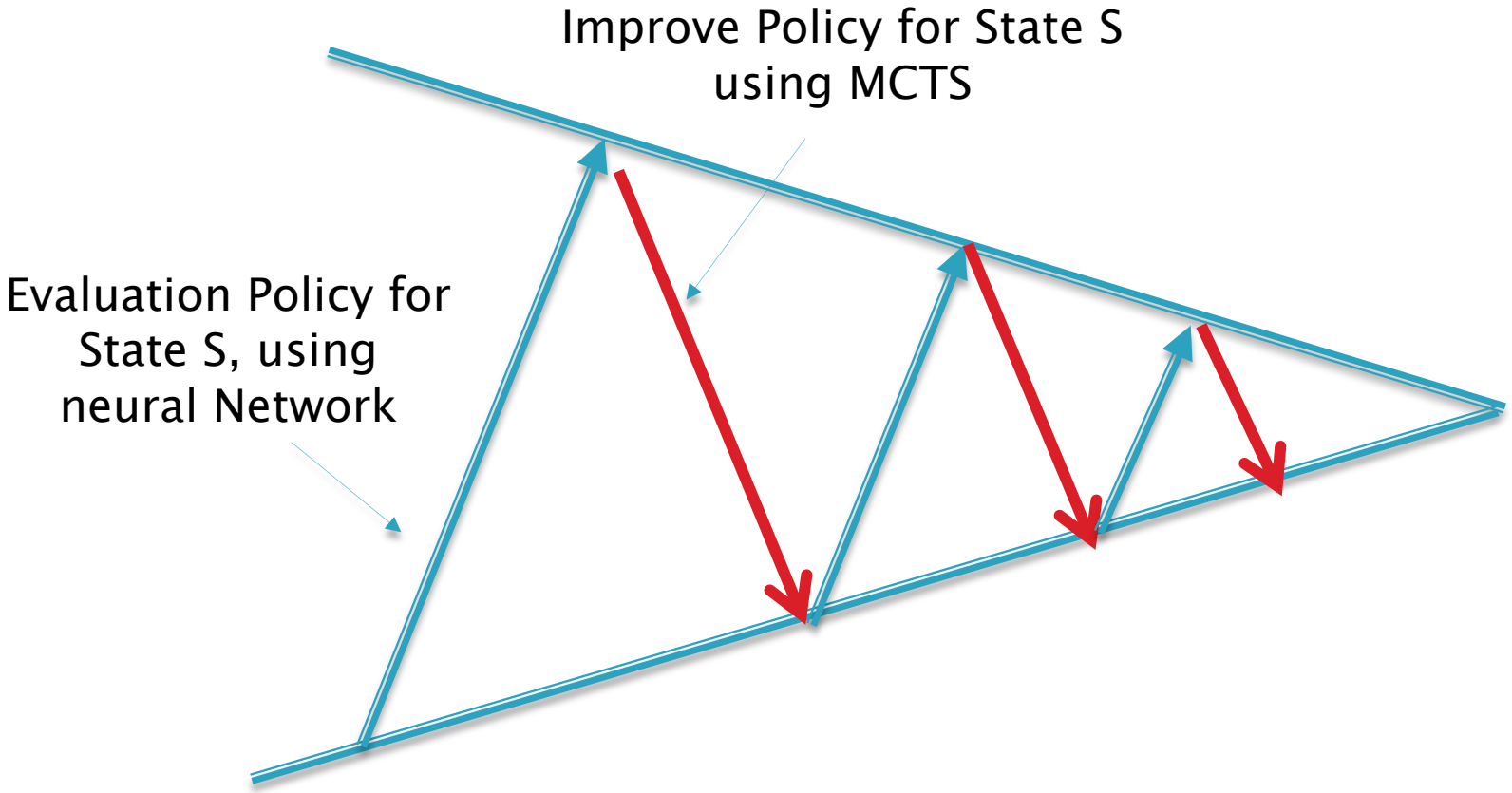


$$v(s) = (1 - \eta)v_\theta(s) + \eta G,$$

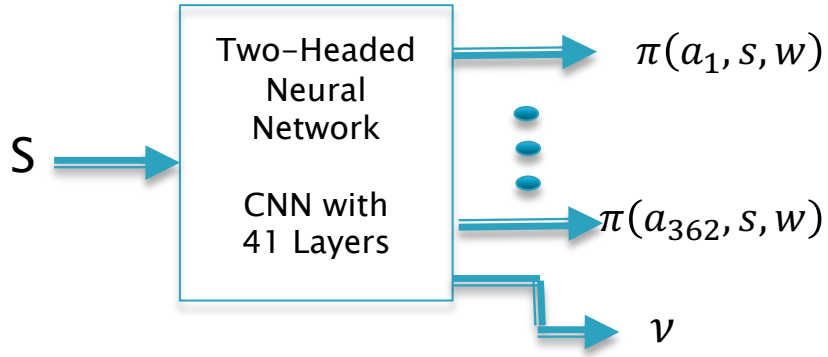
Alpha Go Zero

- Does even better than Alpha Go
- Has a simpler design
- Main ideas:
  - Use Policy Network to choose the best move during game play
  - Use MCTS to develop the best Policy Network
  - Can be considered to be a type of Policy Iteration:
    1. Use the Neural Network to compute a Policy and Value for the Input State S
    2. Use the output of the Neural Network to do MCTS. Use the results of MCTS to find a better action for state S
    3. Use the MCTS supplied Action as a training label for the Policy Network, and use gradient descent to improve the Network
    4. Go back to Step 1

Alpha Go Zero

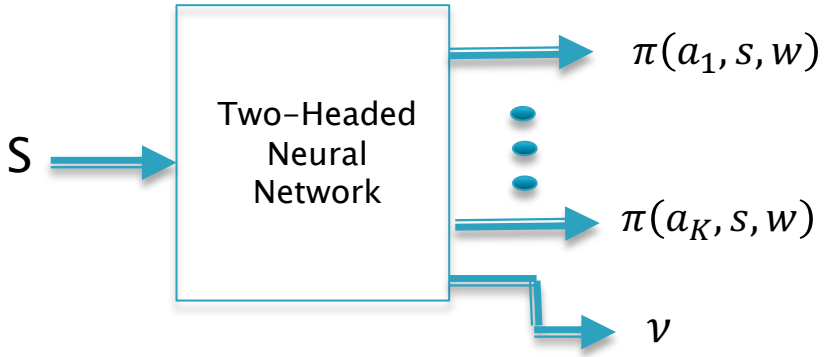


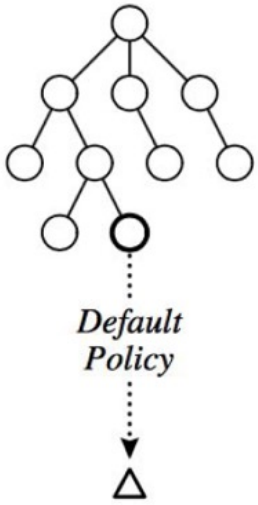
AlphaGo Zero: Functional Policy Iteration + MCTS



- 'Two-Headed' Neural Network: Output 1- The Action Probabilities for State S, Output 2 – The Value Function for State S, which is the probability of winning the game starting from State S
- Input: 19x19x17 Image Stack consisting of 17 binary feature planes.
  - First 8 Feature Planes are raw representations of the board position for Player 1 + 7 previous board positions
  - Next 8 Feature planes similarly code the positions for Player 2
  - The final Feature Plane has a constant value indicating the color of the current play

AlphaGo Zero: Functional Policy Iteration + MCTS

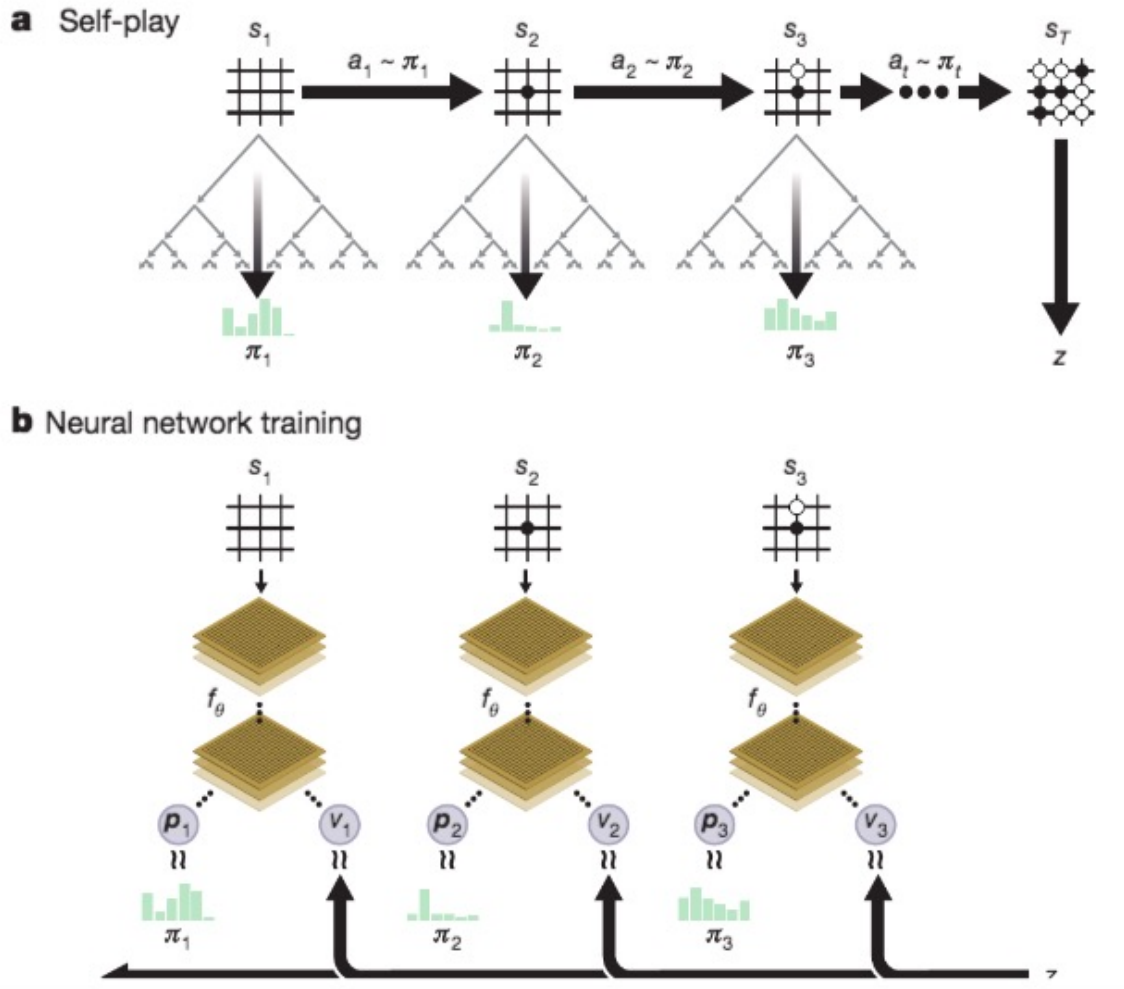




1. Traverse tree using Actions being generated by Neural Network (from a chosen Root State)
2. Get "better" Actions for the Root State using MCTS
3. Use this to improve the Neural Network

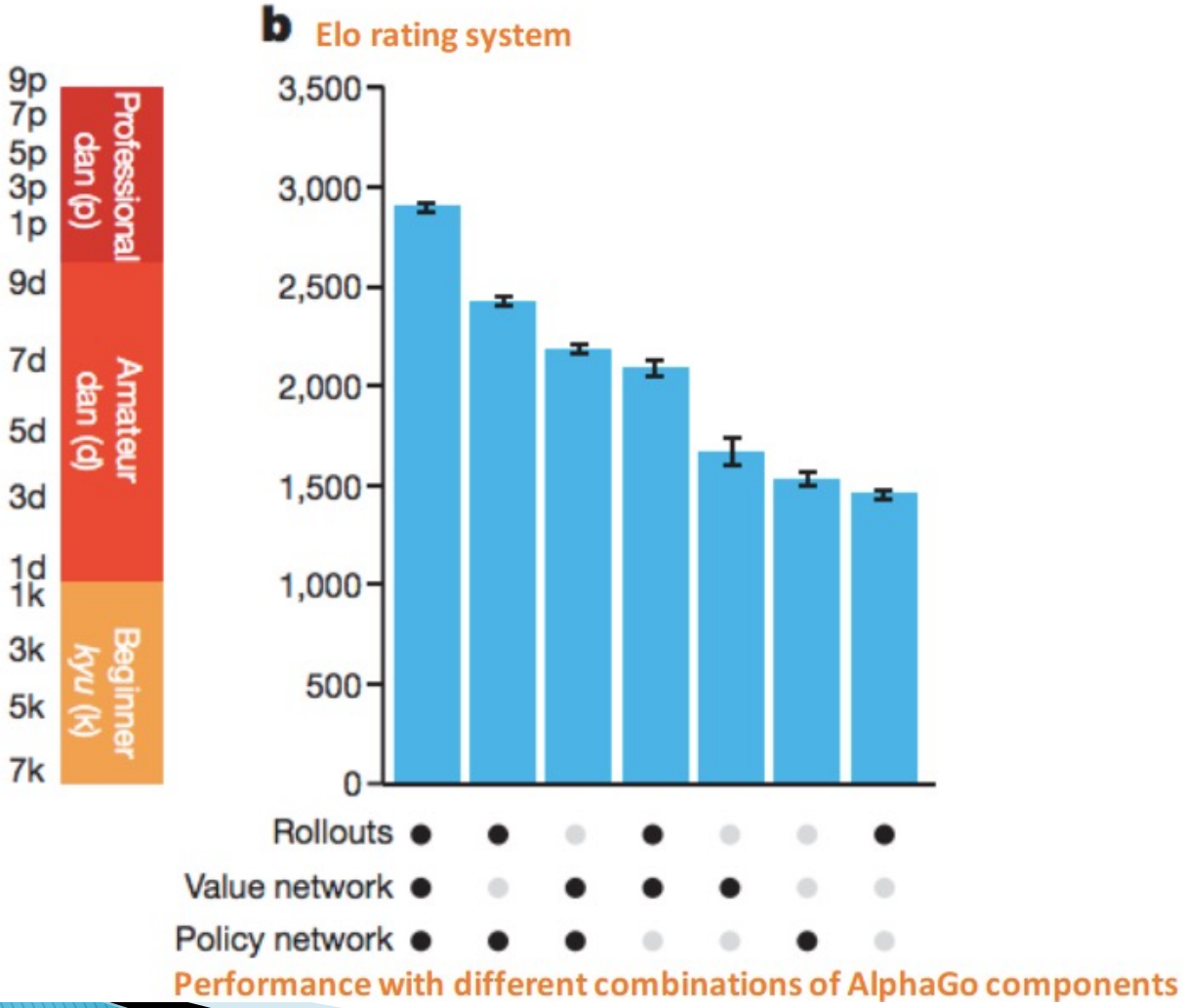
$$(\boldsymbol{p}, v) = f_\theta(s) \text{ and } l = (z - v)^2 - \boldsymbol{\pi}^{\mathrm{T}} \log \boldsymbol{p} + c\|\theta\|^2$$

AlphaGo Zero



$$(\boldsymbol{p}, v) = f_\theta(s) \text{ and } l = (z - v)^2 - \boldsymbol{\pi}^{\mathrm{T}} \log \boldsymbol{p} + c\|\theta\|^2$$

Results



Performance with different combinations of AlphaGo components

Further Reading

- Model Learning and Background Planning: Chapter 8, Sections 8.1-8.2
- Decision Time Planning and Monte Carlo Tree Search: Chapter 8, Sections 8.8-8.11
- Playing Go: Chapter 16, Section 16.6