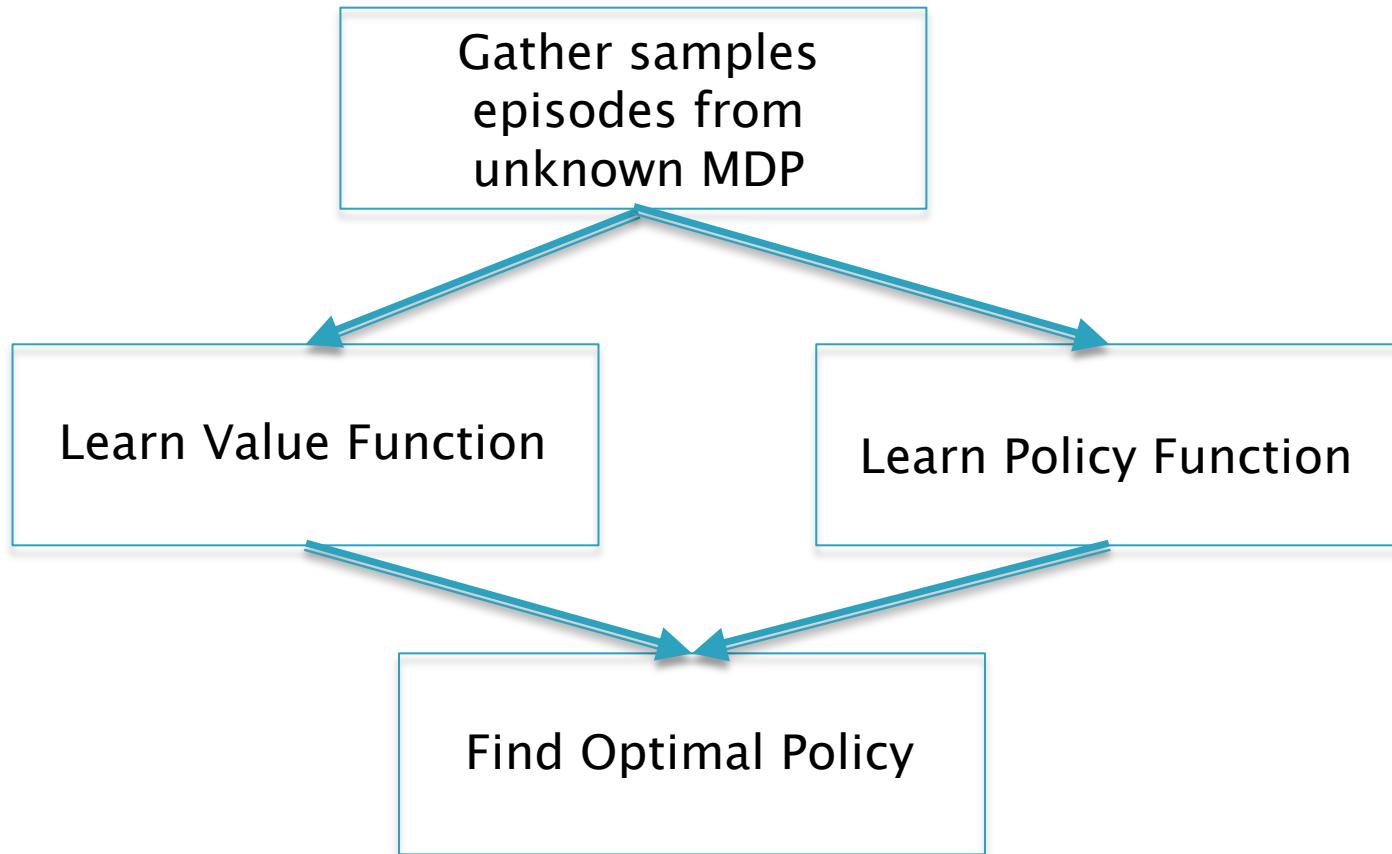


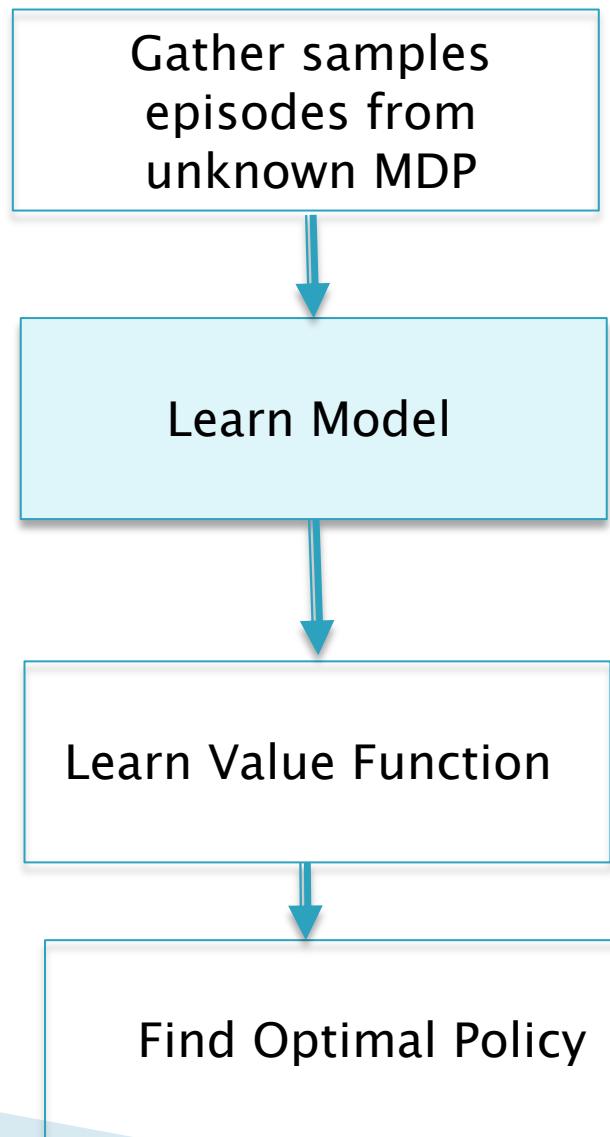
# Model based Planning

Lecture 9  
Subir Varma

# Last Few Lectures: Model Free



# This Lecture: Model Based



Policy Iteration  
Value Iteration

Monte Carlo  
TD  
Q Learning

# Model Based and Model Free RL

## ■ Model-Free RL

- No model
- **Learn** value function (and/or policy) from experience

Learning



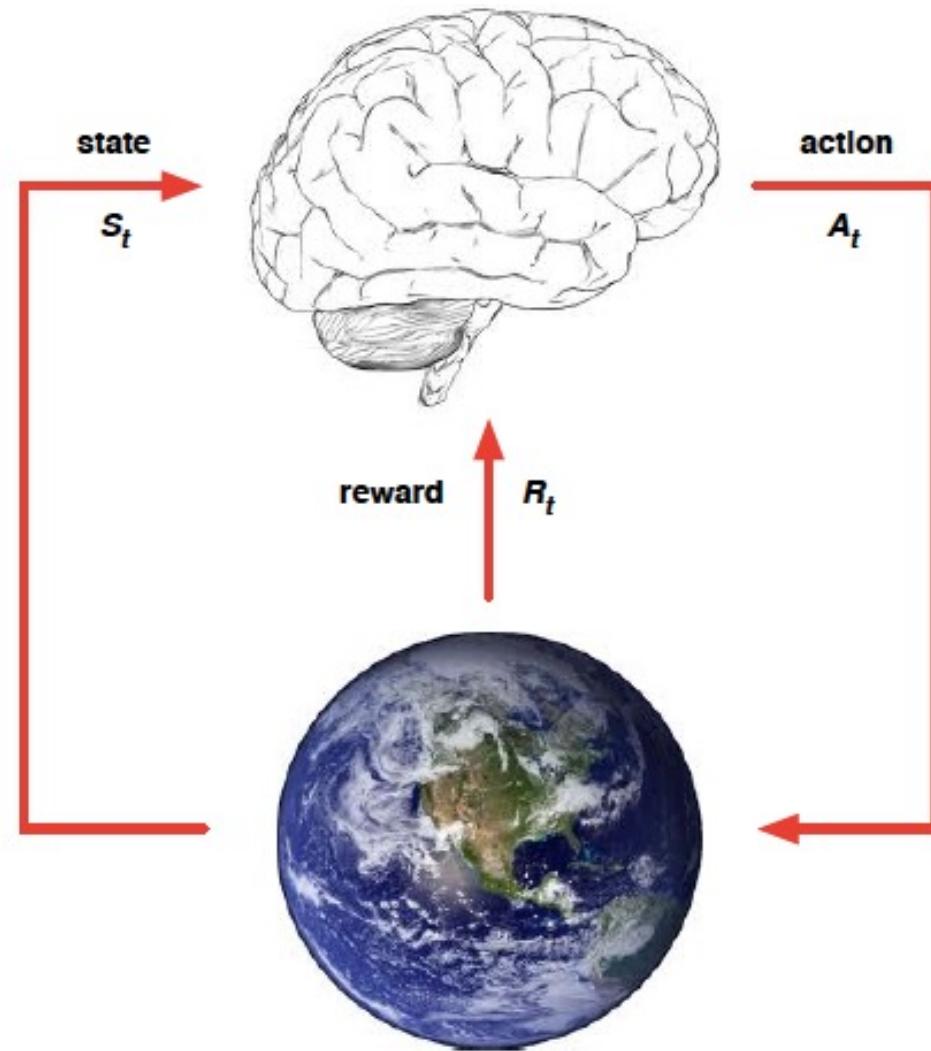
## ■ Model-Based RL

- Learn a model from experience
- **Plan** value function (and/or policy) from model

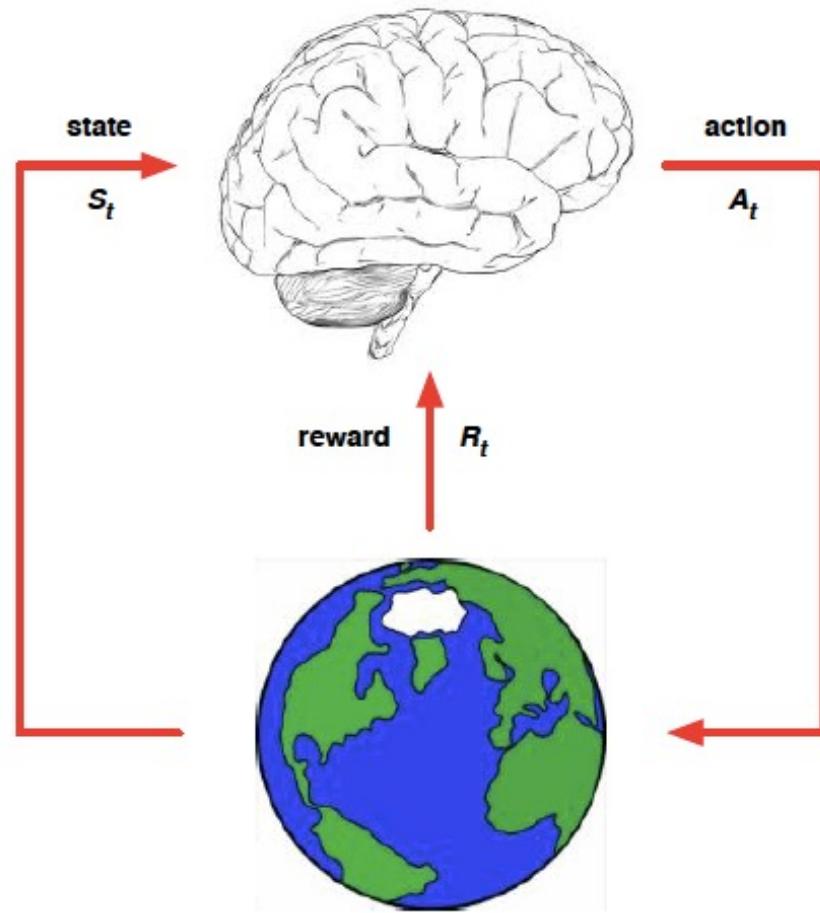
Planning



# Model Free RL - Learning



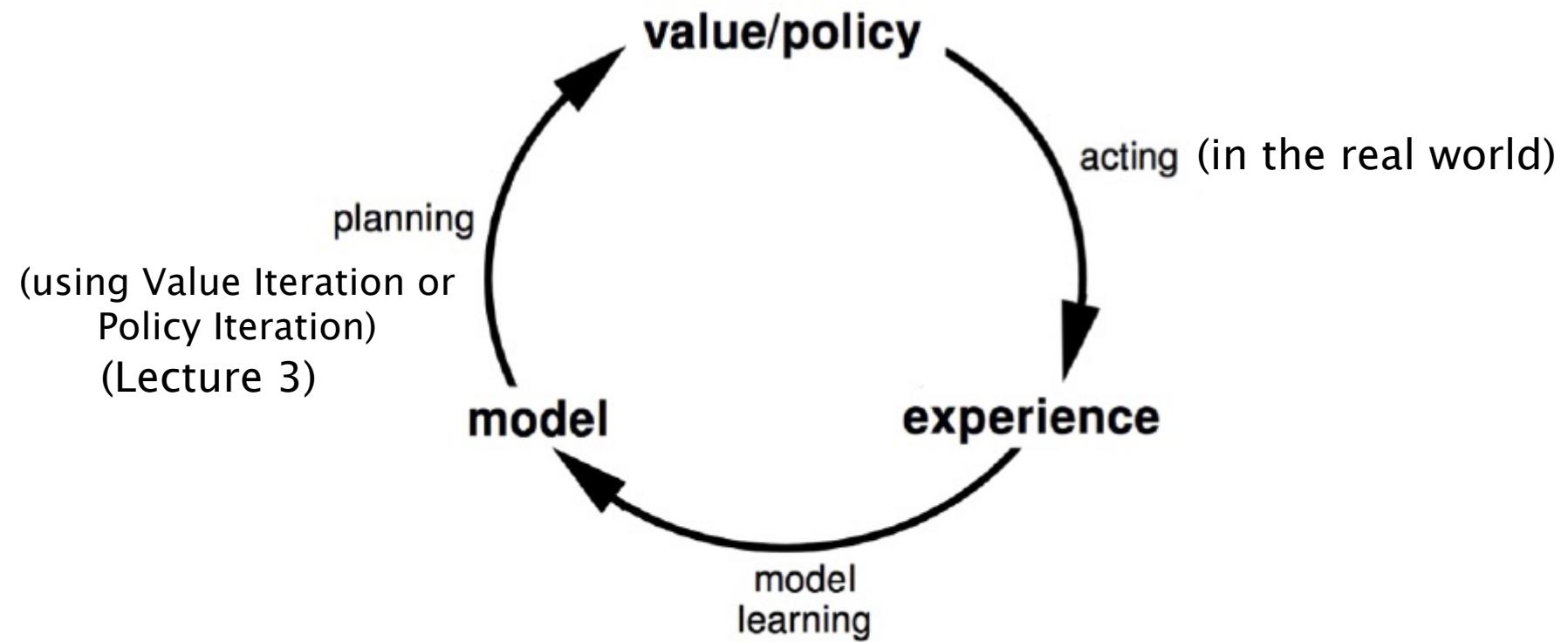
# Model based RL – Planning



# 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  
    ⇒ 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  $\eta$
- 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  $\{S_1, A_1, R_2, \dots, S_T\}$
- This is a supervised learning problem

$$S_1, A_1 \rightarrow R_2, S_2$$

$$S_2, A_2 \rightarrow R_3, S_3$$

⋮

$$S_{T-1}, A_{T-1} \rightarrow R_T, S_T$$

- 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  $\eta$  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

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

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

- 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

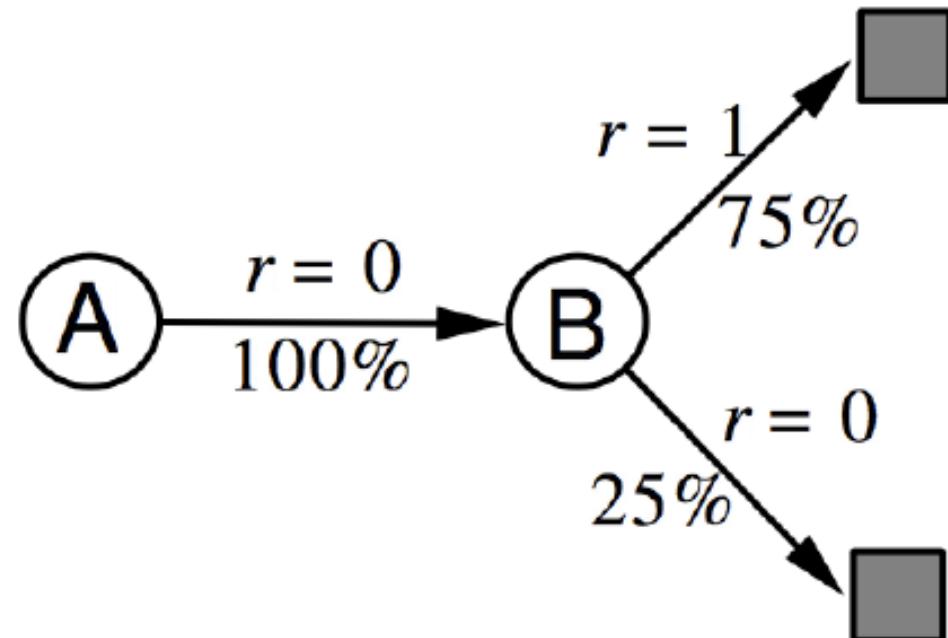
B, 1

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

$$S_{t+1} \sim \mathcal{P}_\eta(S_{t+1} | S_t, A_t)$$

$$R_{t+1} = \mathcal{R}_\eta(R_{t+1} | S_t, A_t)$$

- 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:

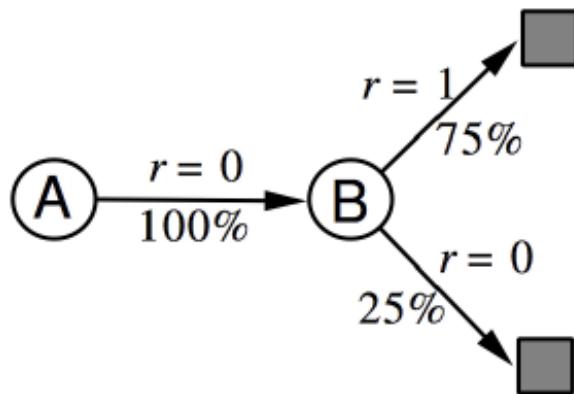
1. Select a state,  $S \in \mathcal{S}$ , and an action,  $A \in \mathcal{A}(S)$ , at random
2. Send  $S, A$  to a sample model, and obtain  
a sample next reward,  $R$ , and a sample next state,  $S'$
3. Apply one-step tabular Q-learning to  $S, A, R, S'$ :  
$$Q(S, A) \leftarrow Q(S, A) + \alpha [R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

# 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, 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:

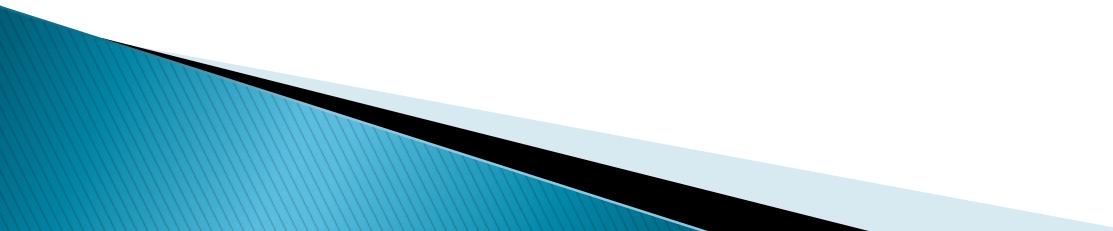
- ▶ **Background Planning**

- 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)

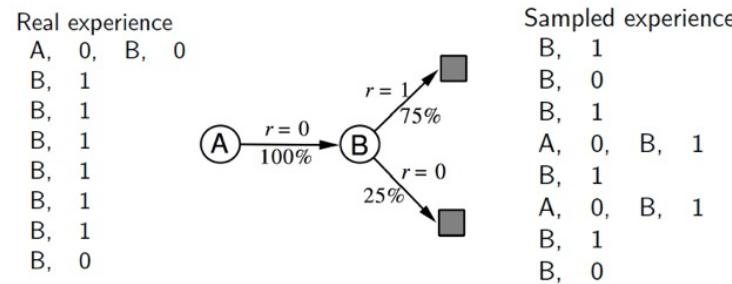
$$S' \sim \mathcal{P}_{s,s'}^a$$

$$R = \mathcal{R}_s^a$$

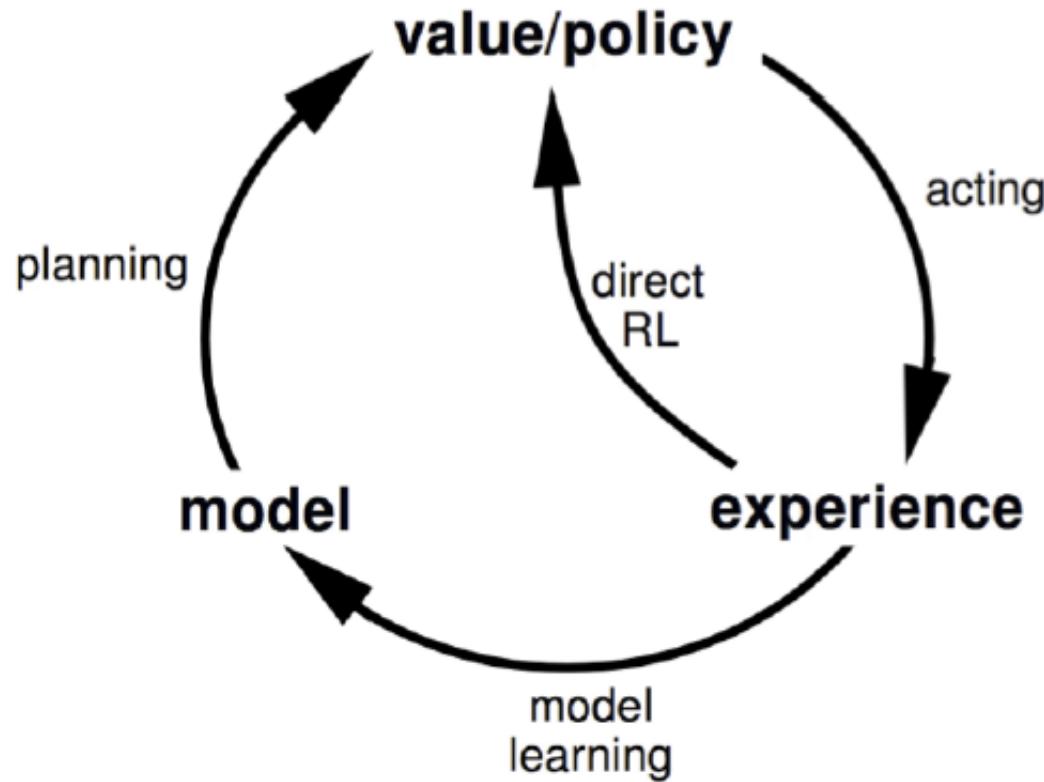
Simulated experience Sampled from model (approximate MDP)

$$S' \sim \mathcal{P}_\eta(S' | S, A)$$

$$R = \mathcal{R}_\eta(R | S, A)$$



# Dyna Architecture



# Dyna Assumptions

1. The Environment is deterministic
2. If the model is queried with the State–Action pair that has been experienced before, it simply returns the last observed next State and next Reward as its prediction
3. During planning the Q–planning algorithm randomly samples only from the State–Action pairs that have previously been experienced

# 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)  $A \leftarrow \varepsilon\text{-greedy}(S, Q)$
- (c) Execute action  $A$ ; observe resultant reward,  $R$ , and state,  $S'$
- (d)  $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
- (e)  $Model(S, A) \leftarrow R, S'$  (assuming deterministic environment)

- (f) Repeat  $n$  times:

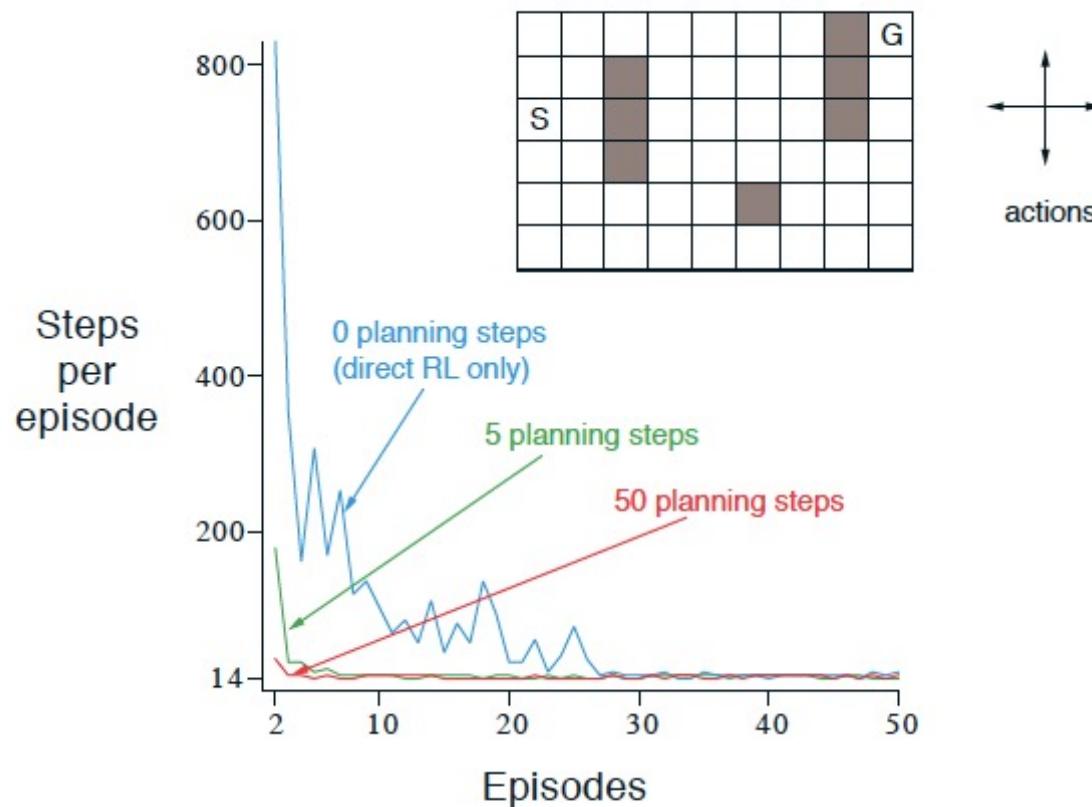
$S \leftarrow$  random previously observed state

$A \leftarrow$  random action previously taken in  $S$

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

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

# 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

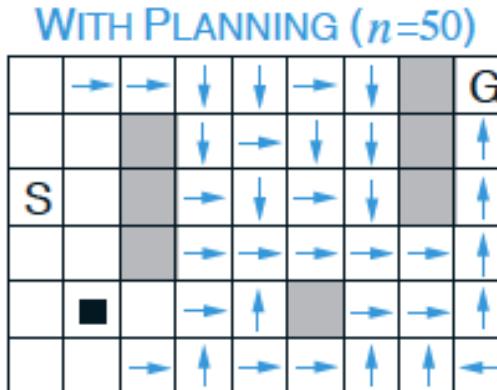
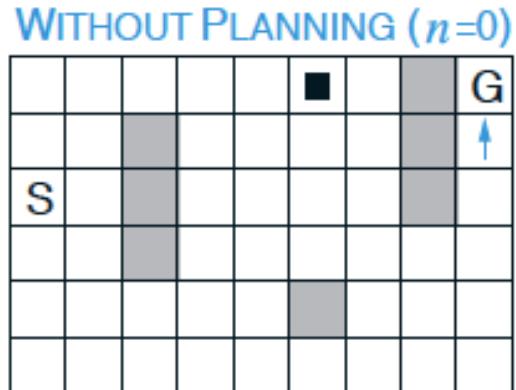


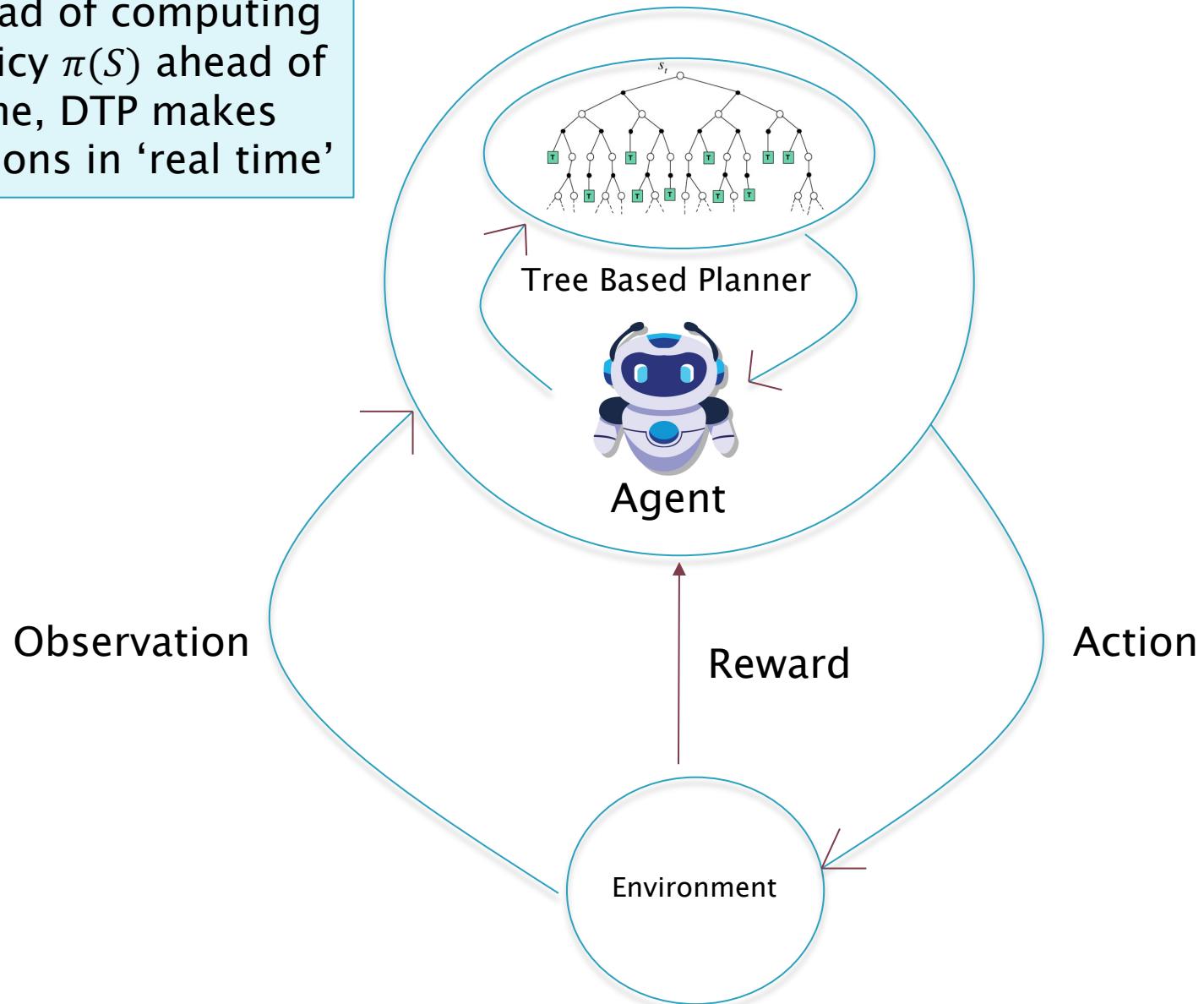
Figure 8.3: Policies found by planning and nonplanning Dyna-Q agents halfway through the second episode. The arrows indicate the greedy action in each state; if no arrow is shown for a state, then all of its action values were equal. The black square indicates the location of the agent. ■

# Decision Time Planning Monte Carlo Tree Search

# Decision Time Planning

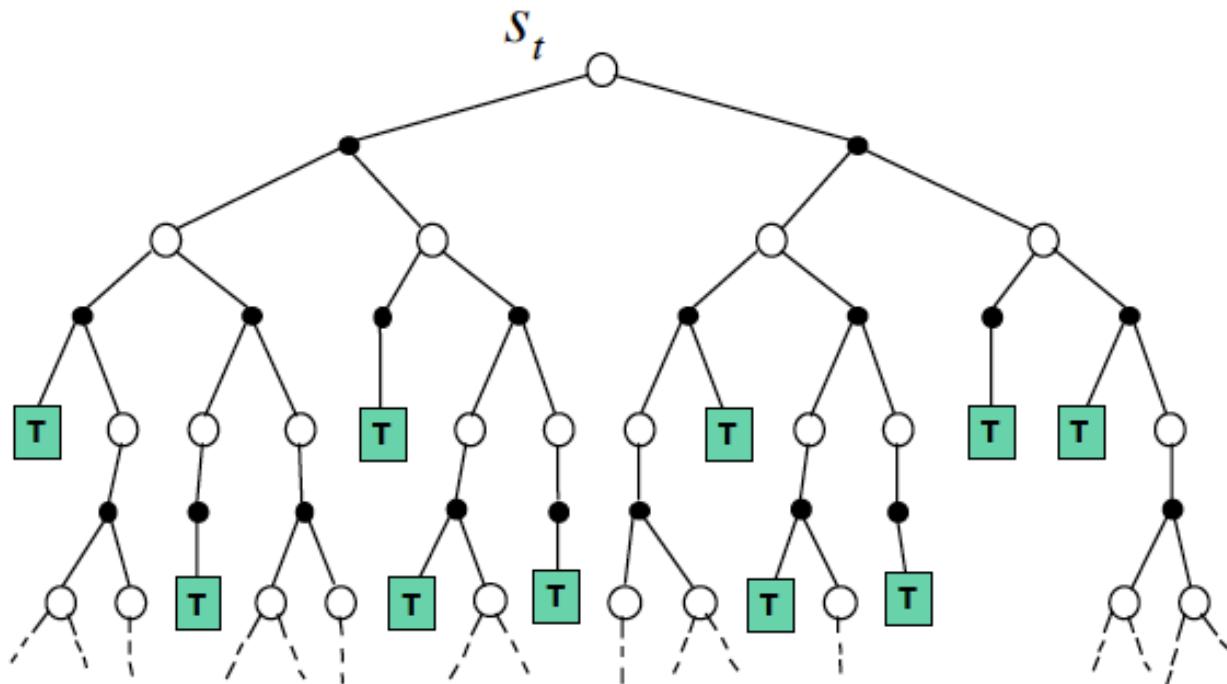
- ▶ Decision Time Planning focuses on a particular state
- ▶ Agent finds itself in state S
  - Agent begins planning, with the objective of choosing a single best action to take
  - The Agent takes the chosen Action, the planning process stops
- ▶ Planning process re-started for each new state

Instead of computing a policy  $\pi(S)$  ahead of time, DTP makes decisions in 'real time'



# Forward Search

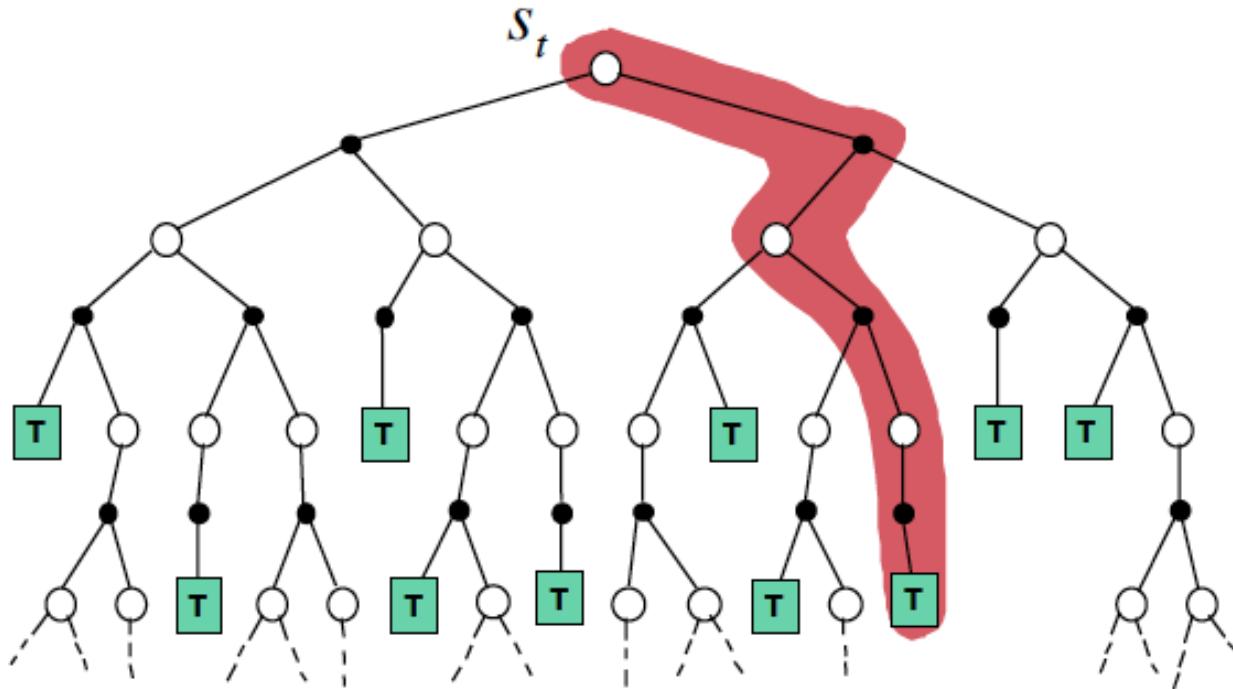
- Forward search algorithms select the best action by lookahead
- They build a search tree with the current state  $s_t$  at the root
- Using a model of the MDP to look ahead



- 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

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

- 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**  $\pi$
- For each action  $a \in \mathcal{A}$ 
  - Simulate  $K$  episodes from current (real) state  $s_t$

Also called  
Rollout Policy

$$\{\mathbf{s}_t, \mathbf{a}, R_{t+1}^k, S_{t+1}^k, A_{t+1}^k, \dots, S_T^k\}_{k=1}^K \sim \mathcal{M}_\nu, \pi$$

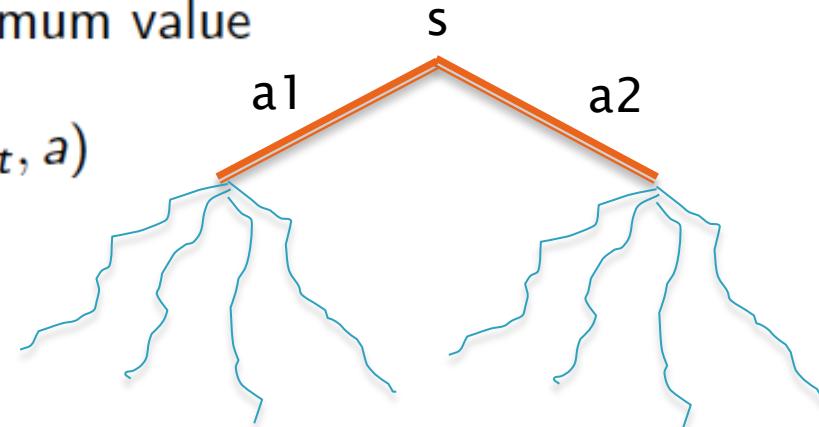
- Evaluate actions by mean return (**Monte-Carlo evaluation**)

Only the root  
Is evaluated

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

- Select current (real) action with maximum value

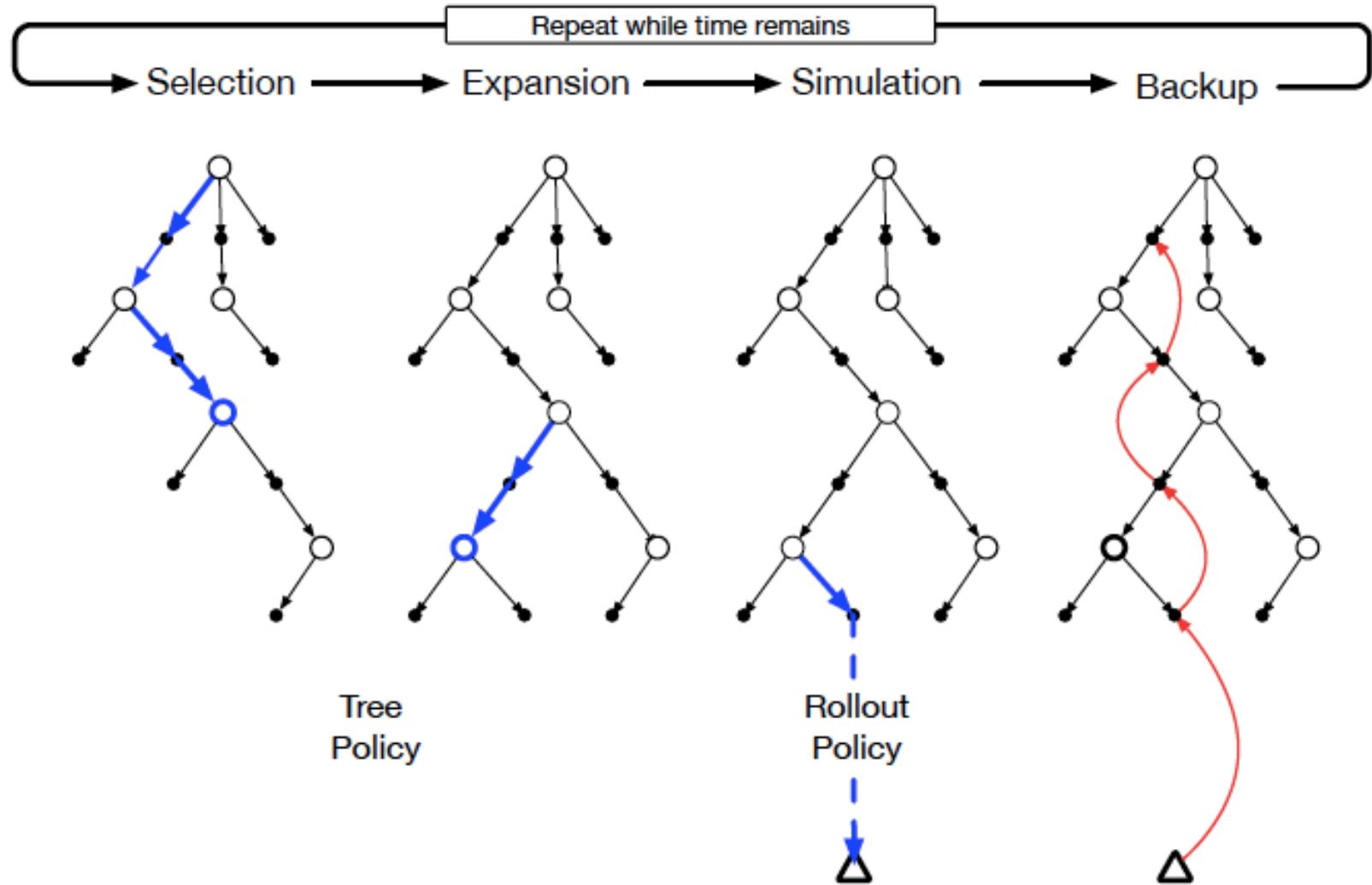
$$a_t = \operatorname{argmax}_{a \in \mathcal{A}} Q(s_t, a)$$



# 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

## ▶ Selection

- Two types of nodes:
  1. Those with child nodes whose Q values have been evaluated: For these nodes selection is done using algorithms such as epsilon greedy, UCB etc
  2. Those which have at least one un-evaluated child node  
For these nodes, choose one of the un-evaluated Actions and expand using the Rollout policy.

## ▶ Expansion

- Proceed down the tree using the Selection rule, until you come across a node that has at least one un-evaluated Action. Choose one of the un-evaluated Actions and expand using the Rollout Policy.

## ▶ Simulation

- This is the Rollout process. Proceed down the tree using the Rollout Policy until you hit the terminal state.

## ▶ Backup

- Use the Backup Formula to compute the Q values for all the (S,A) pairs along the path

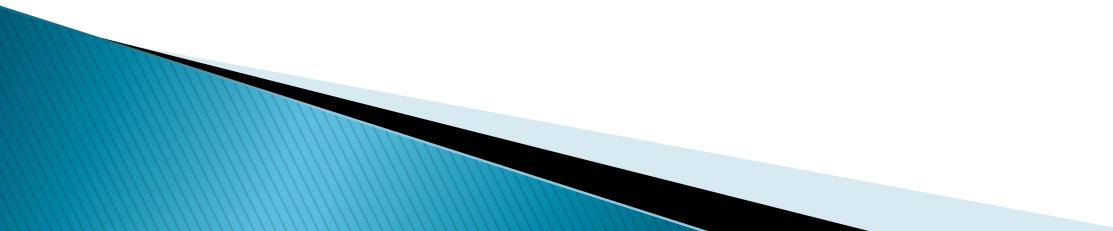
$$Q(S, A) \leftarrow Q(S, A) + \alpha(G - Q(S, A))$$

# 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

Can potentially re-use the Q values for the next run of the tree search

# Playing GO with MCTS and Deep Reinforcement Learning



# Alpha Go Zero Algorithm



## Mastering the Game of Go without Human Knowledge

David Silver\*, Julian Schrittwieser\*, Karen Simonyan\*, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, Demis Hassabis.

DeepMind, 5 New Street Square, London EC4A 3TW.

\*These authors contributed equally to this work.

A long-standing goal of artificial intelligence is an algorithm that learns, *tabula rasa*, superhuman proficiency in challenging domains. Recently, *AlphaGo* became the first program to defeat a world champion in the game of Go. The tree search in *AlphaGo* evaluated positions and selected moves using deep neural networks. These neural networks were trained by supervised learning from human expert moves, and by reinforcement learning from self-play. Here, we introduce an algorithm based solely on reinforcement learning, without human data, guidance, or domain knowledge beyond game rules. *AlphaGo* becomes its own teacher: a neural network is trained to predict *AlphaGo*'s own move selections and also the winner of *AlphaGo*'s games. This neural network improves the strength of tree search, resulting in higher quality move selection and stronger self-play in the next iteration. Starting *tabula rasa*, our new program *AlphaGo Zero* achieved superhuman performance, winning 100–0 against the previously published, champion-defeating *AlphaGo*.

Much progress towards artificial intelligence has been made using supervised learning systems that are trained to replicate the decisions of human experts<sup>1–4</sup>. However, expert data is often expensive, unreliable, or simply unavailable. Even when reliable data is available it may impose a ceiling on the performance of systems trained in this manner<sup>5</sup>. In contrast, reinforcement learning systems are trained from their own experience, in principle allowing them to exceed human capabilities, and to operate in domains where human expertise is lacking. Recently, there has been rapid progress towards this goal, using deep neural networks trained by reinforcement learning. These systems have outperformed humans in computer games such as Atari<sup>6,7</sup> and 3D virtual environments<sup>8–10</sup>. However, the most challenging domains in terms of human intellect – such as the

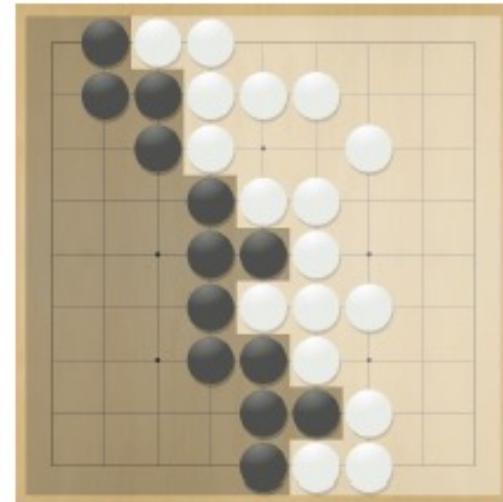
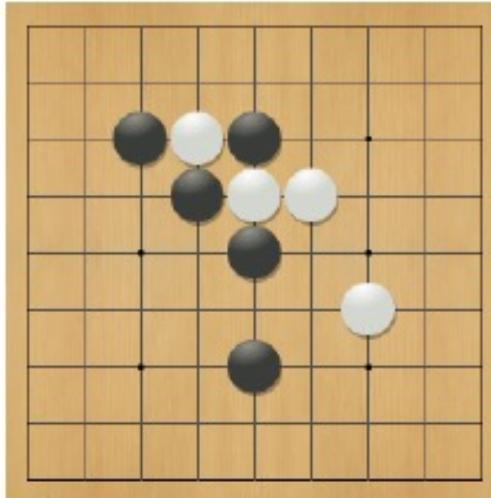
# Game of Go

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



# Rules of Go

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



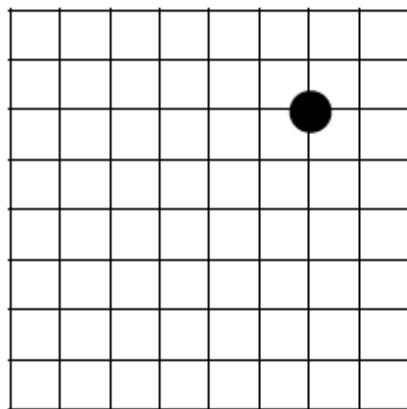
# Challenges in Playing Go

- ▶ Older methods that worked well for games such as Chess did not work well for Go
- ▶ Search Space for Go is significantly larger than that for Chess
  - Go Legal Moves per Position approx. 250 vs 35 for Chess
  - Number of moves in a Go game approx. 150 vs 80 for Chess
- ▶ Exhaustive Search is infeasible
- ▶ Difficulty in defining an adequate Position Evaluation Function

# The Game of GO

19 X 19 Board

$d = 1$

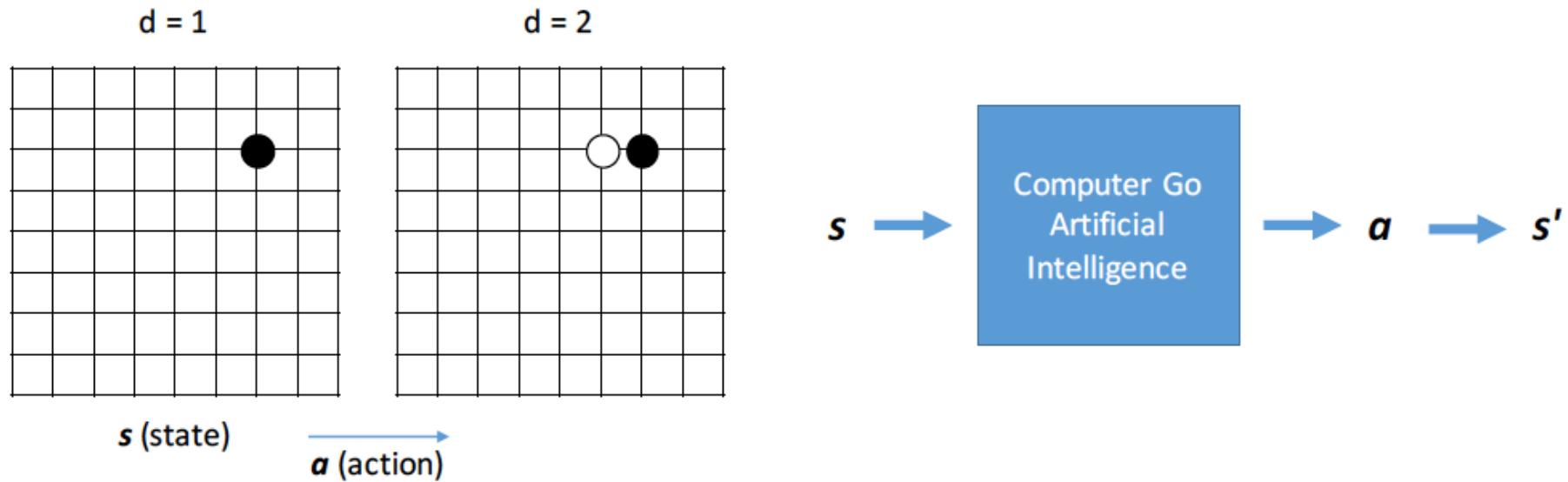


$s$  (state)

$$= \left[ \begin{array}{c} 000000000 \\ 000000000 \\ 000000\textcolor{red}{1}00 \\ 000000000 \\ 000000000 \\ 000000000 \\ 000000000 \\ 000000000 \end{array} \right]$$

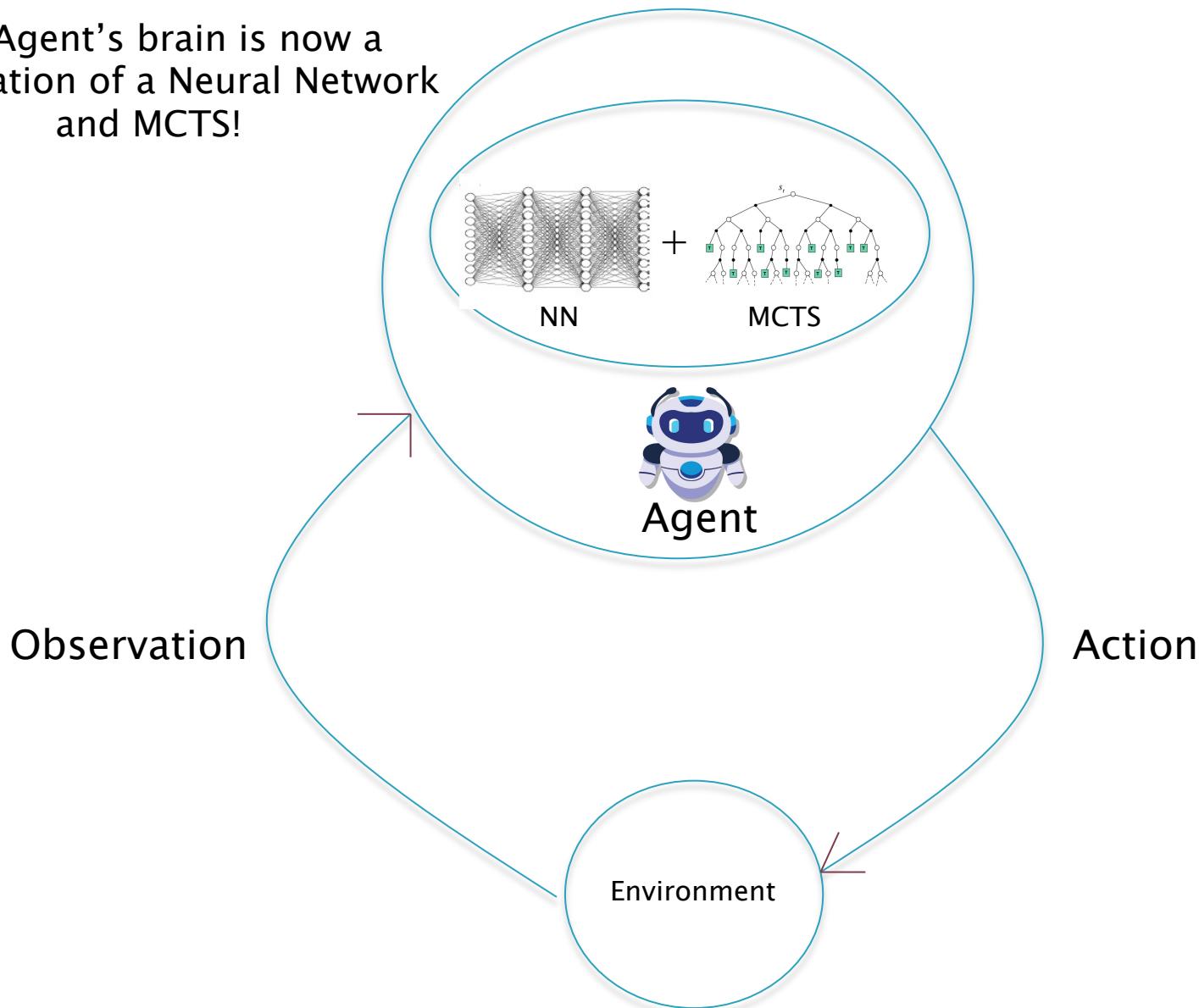
(e.g. we can represent the board into a matrix-like form)

# Computer Aided GO

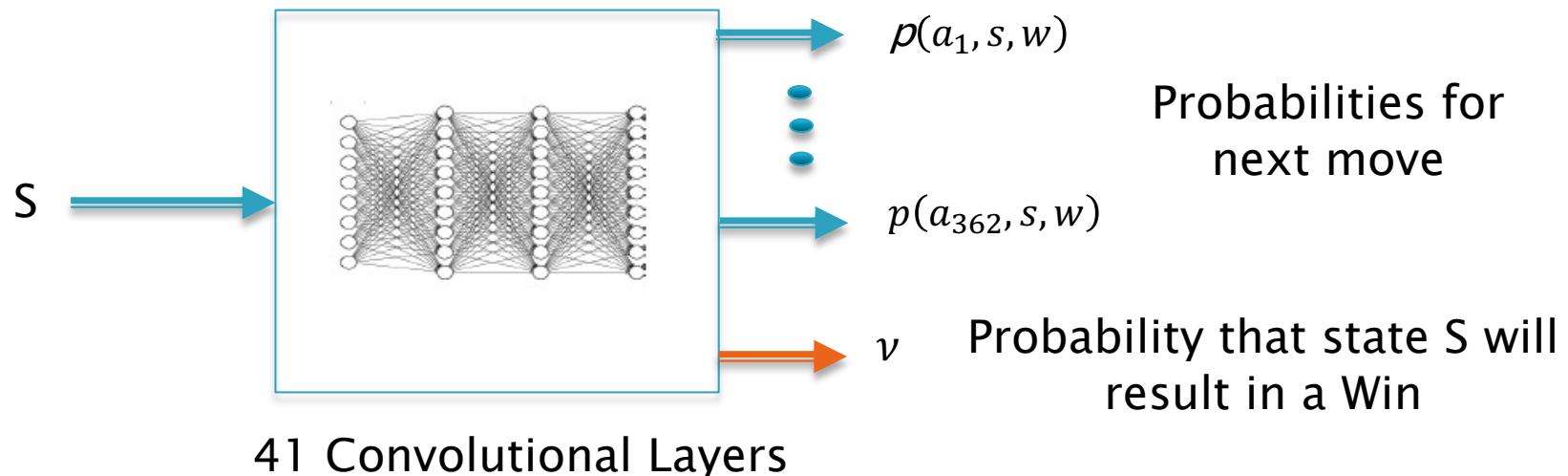


Given  $s$ , pick the best  $a$

The Agent's brain is now a combination of a Neural Network and MCTS!

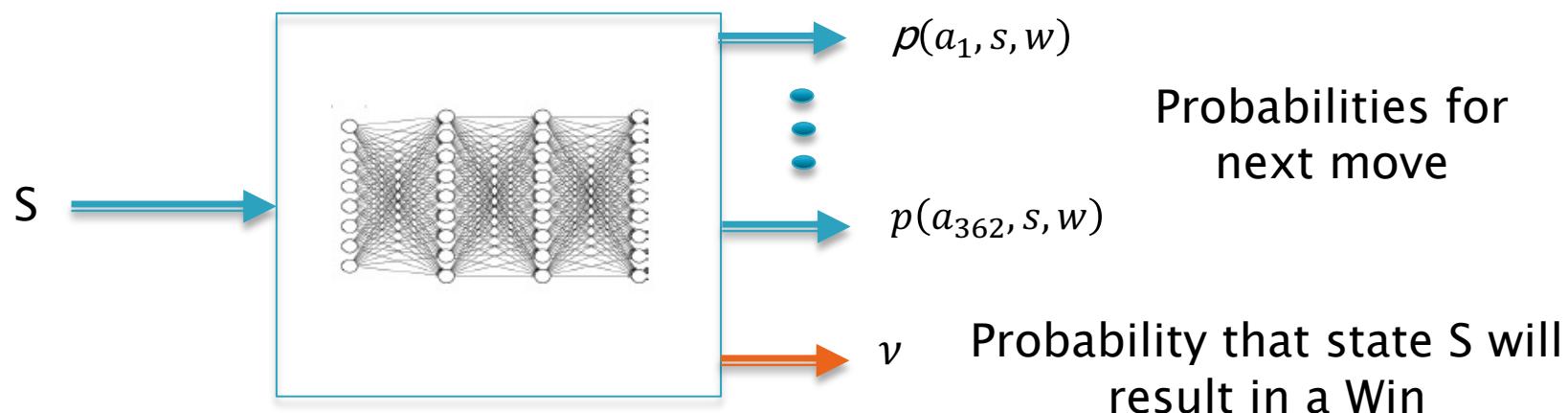


# AlphaGo Zero Neural Network



- ‘Two-Headed’ Neural Network with two outputs:
  - Output 1: The Move (Action) Probabilities for State  $S$ ,
  - Output 2: The Value Function for State  $S$ , which is the probability of winning the game starting from State  $S$

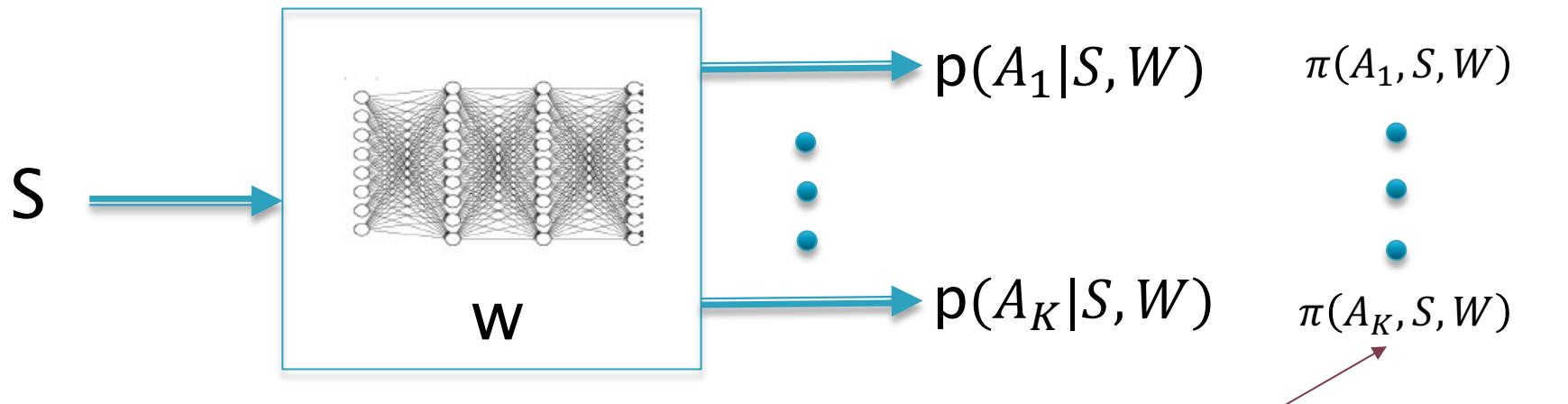
# AlphaGo Zero State



- Input  $S$ :  $19 \times 19 \times 17$  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

# How Does AlphaGo Zero fit within the RL Framework?

- ▶ Recall the Policy Gradient Algorithm
- ▶ We were not able to train the Neural Network using plain Logistic Regression since we didn't know the optimal Action for state  $S$



What if we knew the Optimal Action?

Then we can simply use the Logistic Regression Reward Function  $L$  to optimize the Neural Network!

$$L(W) = \sum_{k=1}^K \pi_k \log p_k$$

# AlphaGo Zero Algorithm

1. For a given state  $S$ , use the Neural Network to do MCTS using the Move probabilities  $p(S)$  and Value  $v(S)$ .
2. Use the results of MCTS to find better Move Probabilities  $\pi(S)$  for state  $S$
3. Proceed in this manner until the game ends, resulting in a win ( $z = +1$ ) or a loss ( $z = -1$ )
4. Use  $(\pi(S), z)$  values as a training label for the Neural Network, and run Gradient Descent on the network using the Loss Function I given by

5. 
$$l = (z - v)^2 - \pi^T \log p + c \|\theta\|^2$$

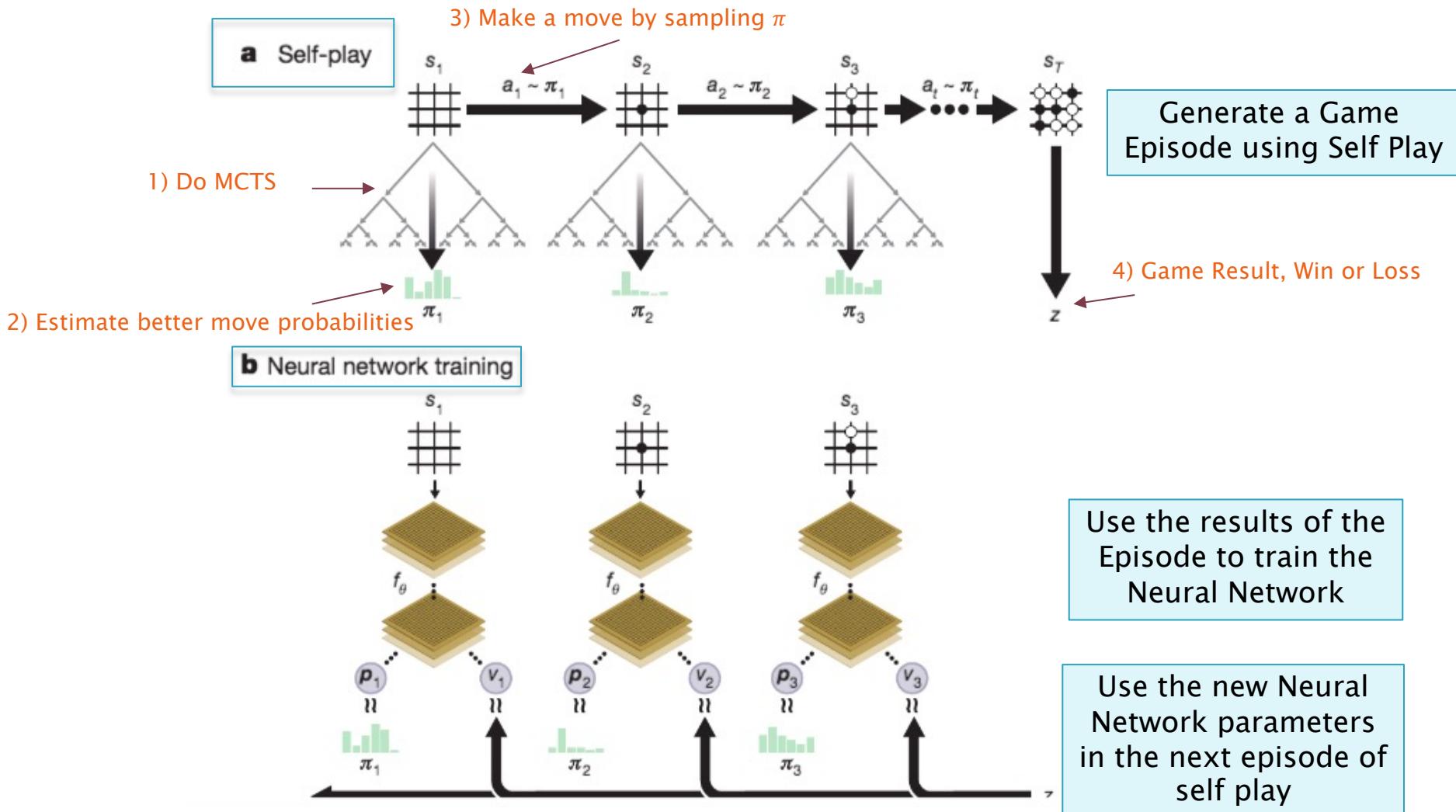
Mean Square Error

Cross Entropy Loss between Label  $\pi$  and Network Output  $p$

Regularization Term to prevent Overfitting

(See Lecture 8 p. 53)

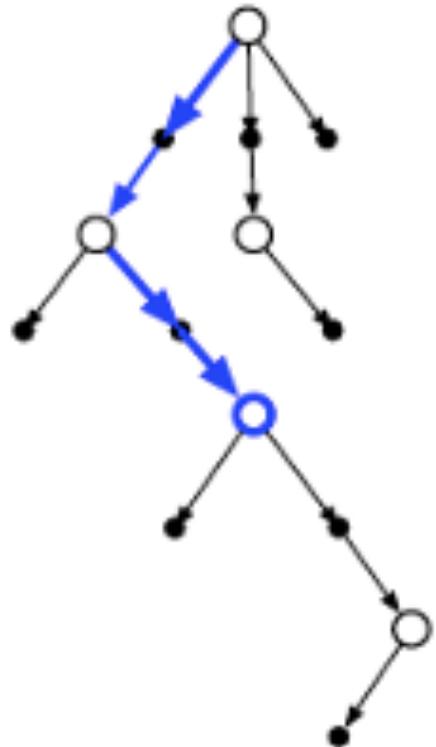
# AlphaGo Zero Training Algorithm



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

4.9 million games of self-play used!  
Took about 3 days

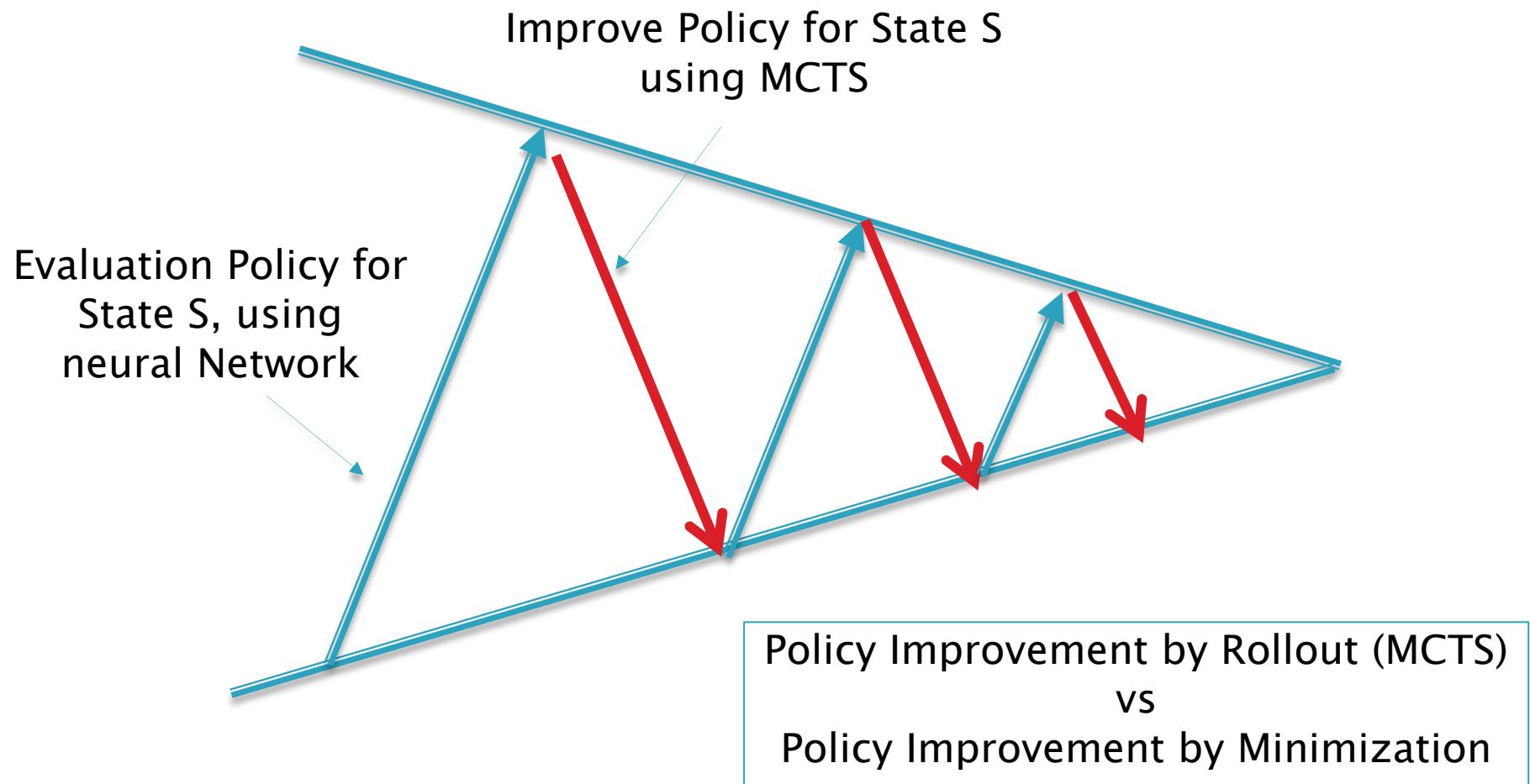
# AlphaGo Zero MCTS



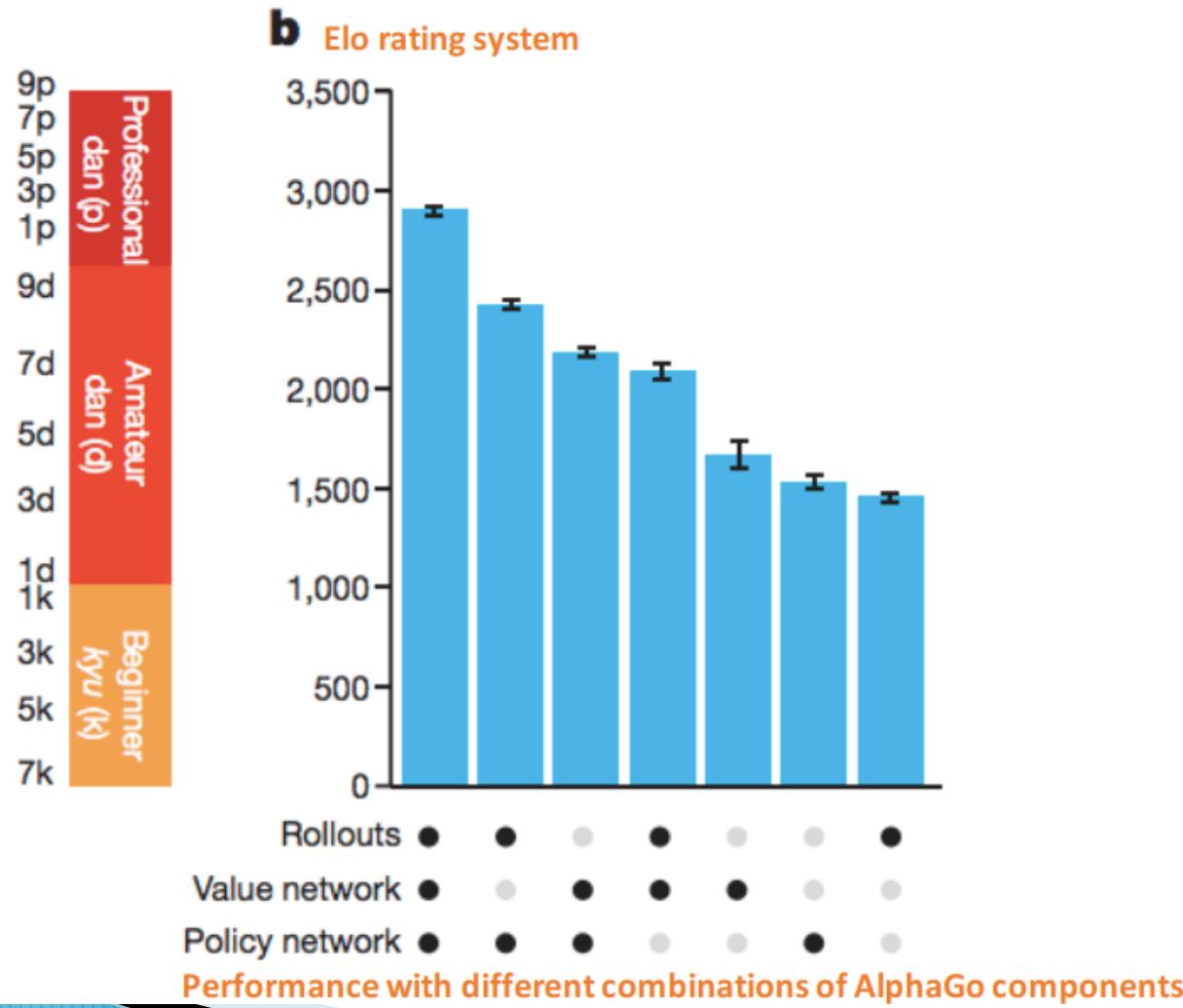
MCTS run for 1600 iterations

1. Each edge  $(s,a)$  in the tree stores a Prior Probability  $P(s,a)$ , a Visit Count  $N(s,a)$  and an Action Value  $Q(s,a)$
2. Traverse the tree by selecting the edge with maximum value of  $Q(s,a) + U(s,a)$ , where
$$U(s,a) \propto \frac{P(s,a)}{1 + N(s,a)}$$
Until a Leaf Node  $s'$  is encountered
3. This leaf node is evaluated using the Neural Network to compute both  $(P(s',a), V(s'))$
4. Each edge traversed in the tree is updated to increment its visit count  $N(s,a)$  and Action Value  $Q(s,a)$  using
$$Q(s,a) = \frac{1}{N(s,a)} \sum_{s'|(s,a) \rightarrow s'} V(s')$$
5. Once the search is complete, move probabilities  $\pi$  are returned, proportional to  $N^{1/\tau}$  where  $N$  is the Visit Count of each move from the Root State and  $\tau$  is a temperature parameter

# Alpha Go Zero



# Results



# 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