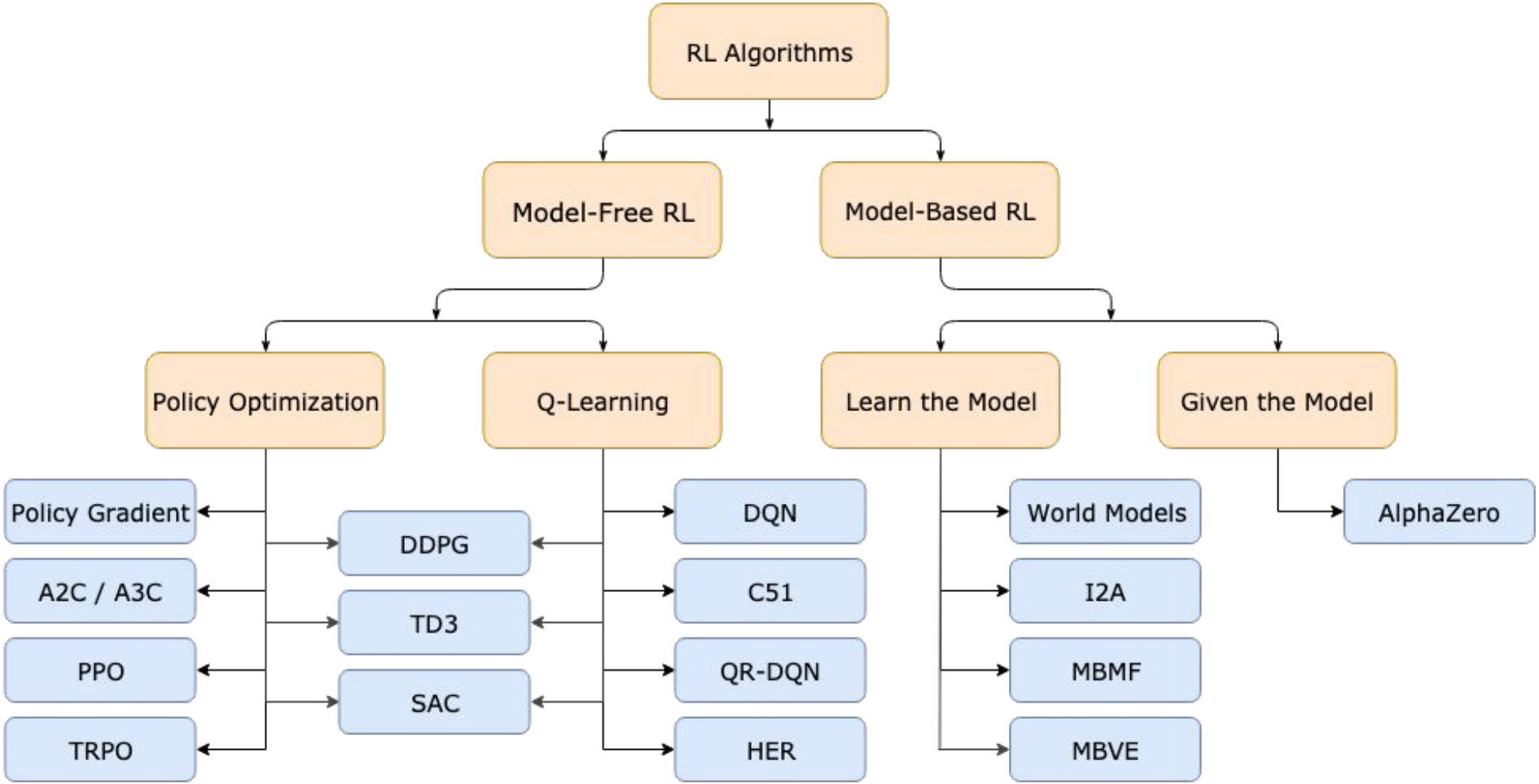
EECS 498/598: Deep Learning
Lecture 12 Reinforcement Learning 3

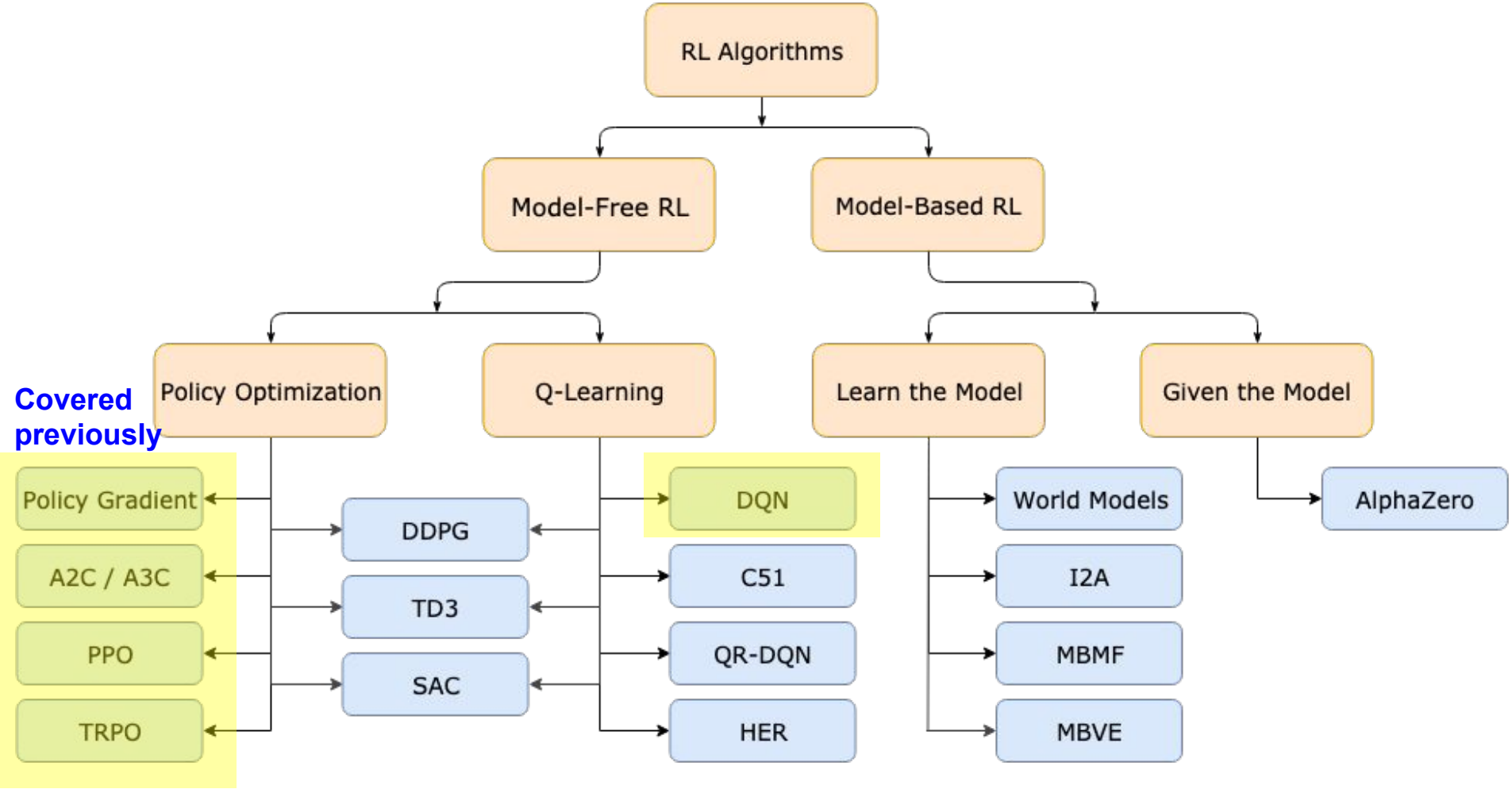
Honglak Lee

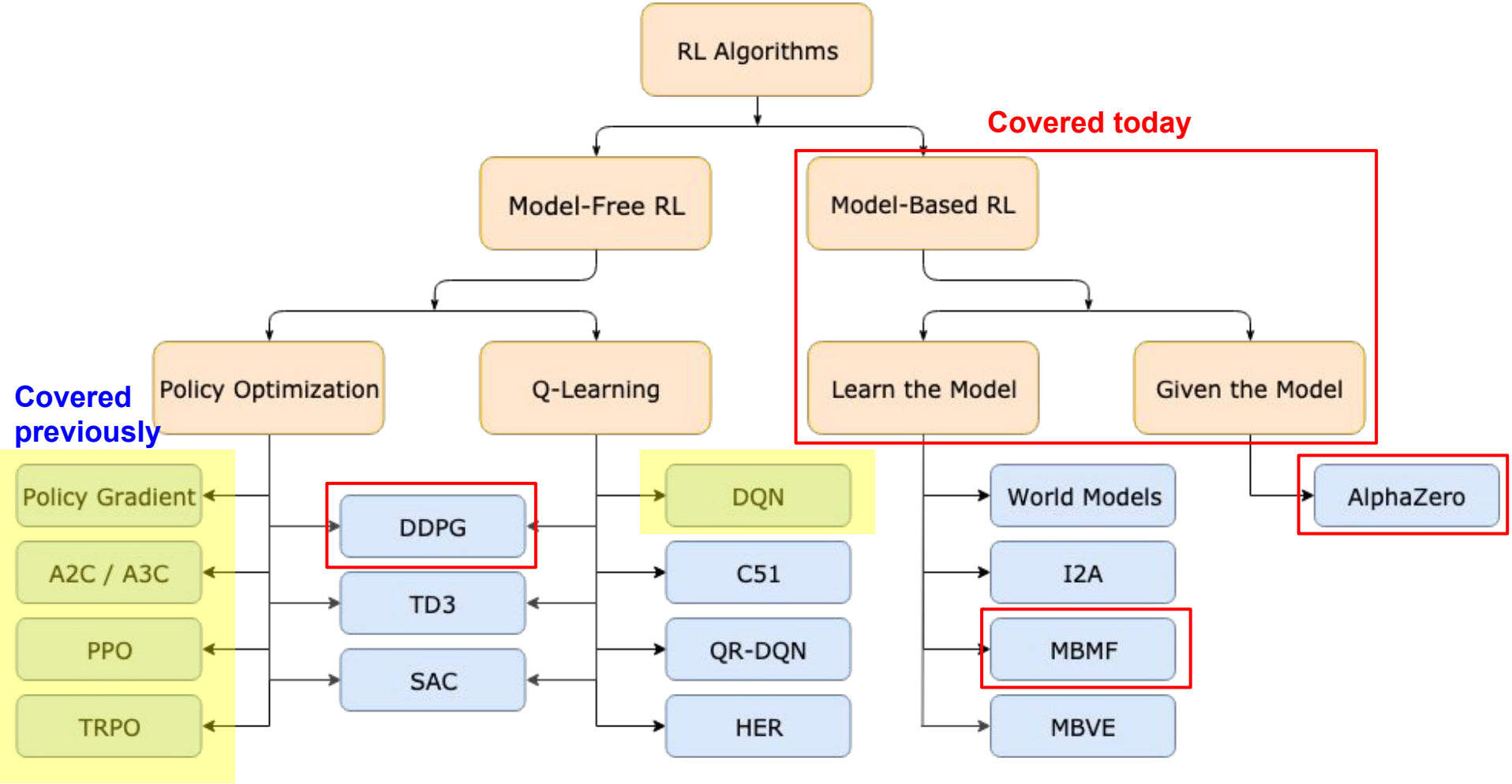
4/5/2019

Outline

- Model-Free RL (for continuous control)
  - CEM, CMA-ES, NAF, DDPG
- Model-Based RL
  - Using the true model
    - AlphaGo
    - AlphaGo Zero
  - Learning the model
    - Dyna-Q
    - MPC $\longrightarrow$ MBMF


RL Algorithms
Model-Free RL
Model-Based RL
Policy Optimization
Q-Learning
Learn the Model
Given the Model
Policy Gradient
A2C / A3C
PPO
TRPO
DDPG
TD3
SAC
DQN
C51
QR-DQN
HER
World Models
I2A
MBMF
MBVE
AlphaZero


RL Algorithms
Model-Free RL
Model-Based RL
Policy Optimization
Q-Learning
Learn the Model
Given the Model
Covered previously
Policy Gradient
A2C / A3C
PPO
TRPO
DDPG
TD3
SAC
DQN
C51
QR-DQN
HER
World Models
I2A
MBMF
MBVE
AlphaZero


RL Algorithms
Covered today
Model-Free RL
Model-Based RL
Policy Optimization
Q-Learning
Learn the Model
Given the Model
Covered previously
Policy Gradient
A2C / A3C
PPO
TRPO
DDPG
TD3
SAC
DQN
C51
QR-DQN
HER
World Models
I2A
MBMF
MBVE
AlphaZero

Outline

- Model-Free RL (for continuous control)
  - CEM, CMA-ES, NAF, DDPG
- Model-Based RL
  - Using the true model
    - AlphaGo
    - AlphaGo Zero
  - Learning the model
    - Dyna-Q
    - MPC
    - MBMF

Q-learning with Continuous Actions

What's the problem with continuous actions?

$$\pi(\mathbf{a}_t|\mathbf{s}_t) = \begin{cases} 1 \text{ if } \mathbf{a}_t = \arg\max_{\mathbf{a}_t} Q_\phi(\mathbf{s}_t, \mathbf{a}_t) \\ 0 \text{ otherwise} \end{cases}$$

this max

$$\text{target value } y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mathbf{a}'_j)$$

this max
particularly problematic (inner loop of training)

How do we perform the max?

Option 1: optimization

- gradient based optimization (e.g., SGD) a bit slow in the inner loop
- action space typically low-dimensional – what about stochastic optimization?

Slide credit: Sergey Levine

Q-learning with Stochastic Optimization

Simple solution:

$$\max_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a}) \approx \max \{Q(\mathbf{s}, \mathbf{a}_1), \ldots, Q(\mathbf{s}, \mathbf{a}_N)\}$$

$(\mathbf{a}_1, \ldots, \mathbf{a}_N)$ sampled from some distribution (e.g., uniform)

+ dead simple

+ efficiently parallelizable

- not very accurate

More accurate solution:

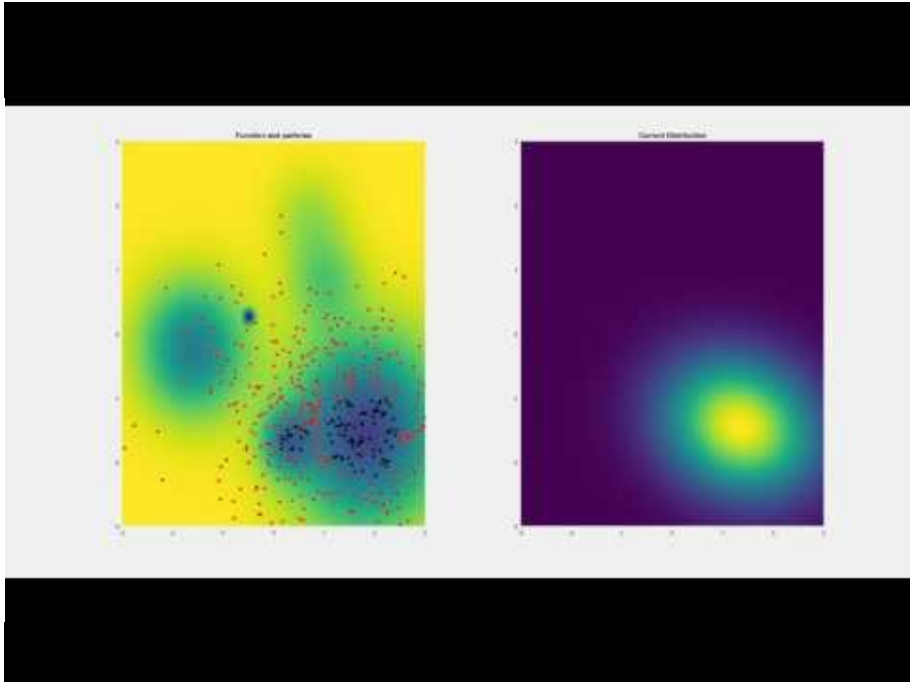
- Cross-entropy method (CEM)
  - Simple iterative optimization
  - https://en.wikipedia.org/wiki/Cross-entropy_method
- CMA-ES
  - More advanced iterative stochastic optimization
  - http://blog.otoro.net/2017/10/29/visual-evolution-strategies/

Slide credit: Sergey Levine

Cross Entropy Method (CEM)

Blue: top solutions
Red: all sampled solutions

Initialize $\mu \in \mathbb{R}^d, \sigma \in \mathbb{R}^d$
for iteration $= 1, 2, \ldots$ **do**
    Collect n samples of $\theta_i \sim N(\mu, \text{diag}(\sigma))$
    Perform a noisy evaluation $R_i \sim \theta_i$
    Select the top p% of samples (e.g. $p = 20$), which we'll call the **elite set**
    Fit a Gaussian distribution, with diagonal covariance, to the elite set, obtaining a new μ, σ.
end for
Return the final μ.

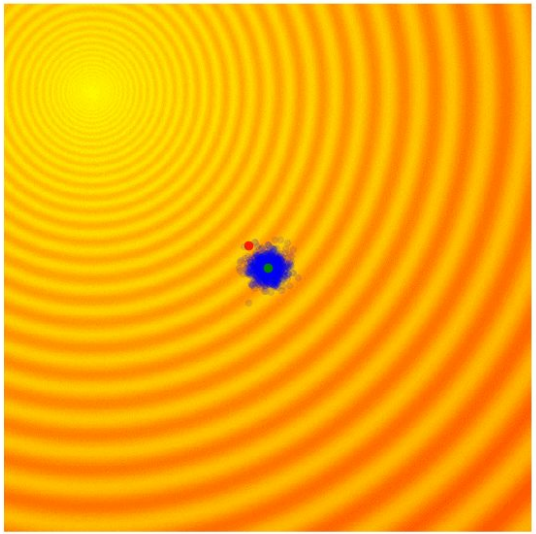


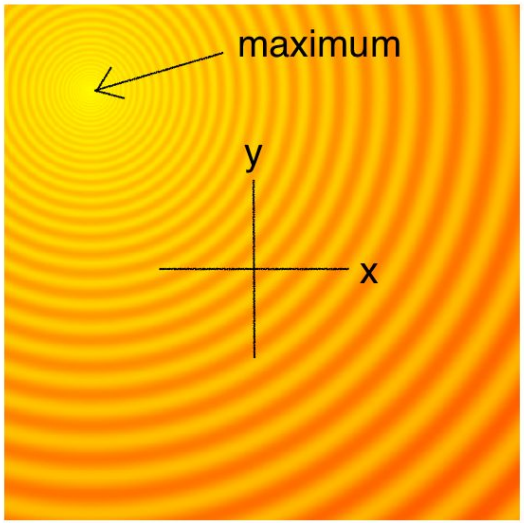
https://learning.mpi-sws.org/mlss2016/slides/2016-MLSS-RL.pdf

https://www.youtube.com/watch?v=I7fS1KDMaOo

Covariance-Matrix Adaptation Evolution Strategy (CMA-ES)

Similar to cross entropy method, but maintain covariance of the top solutions



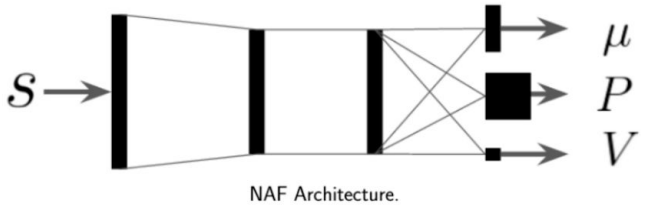


http://blog.otoro.net/2017/10/29/visual-evolution-strategies/
The CMA Evolution Strategy: A Tutorial

Easily Maximizable Q-functions

Option 2: use function class that is easy to optimize

$$Q_\phi(\mathbf{s}, \mathbf{a}) = -\frac{1}{2}(\mathbf{a} - \mu_\phi(\mathbf{s}))^T P_\phi(\mathbf{s})(\mathbf{a} - \mu_\phi(\mathbf{s})) + V_\phi(\mathbf{s})$$



NAF Architecture.

NAF: **N**ormalized **A**dvantage **F**unctions

$$\arg\max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a}) = \mu_\phi(\mathbf{s}) \qquad \max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a}) = V_\phi(\mathbf{s})$$

+ no change to algorithm

+ just as efficient as Q-learning

- loses representational power

Slide credit: Sergey Levine

Gu, Lillicrap, Sutskever, L., ICML 2016

Q-learning with Continuous Actions

Option 3: learn an approximate maximizer

DDPG (Lillicrap et al., ICLR 2016)

"deterministic" actor-critic
(really approximate Q-learning)

$\max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a}) = Q_\phi(\mathbf{s}, \arg\max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a}))$

idea: train another network $\mu_\theta(\mathbf{s})$ such that $\mu_\theta(\mathbf{s}) \approx \arg\max_{\mathbf{a}} Q_\phi(\mathbf{s}, \mathbf{a})$

how? just solve $\theta \leftarrow \arg\max_\theta Q_\phi(\mathbf{s}, \mu_\theta(\mathbf{s}))$

$$\frac{dQ_\phi}{d\theta} = \frac{d\mathbf{a}}{d\theta}\frac{dQ_\phi}{d\mathbf{a}}$$

new target $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mu_\theta(\mathbf{s}'_j))$

Slide credit: Sergey Levine

Q-learning with Continuous Actions

Option 3: learn an approximate maximizer

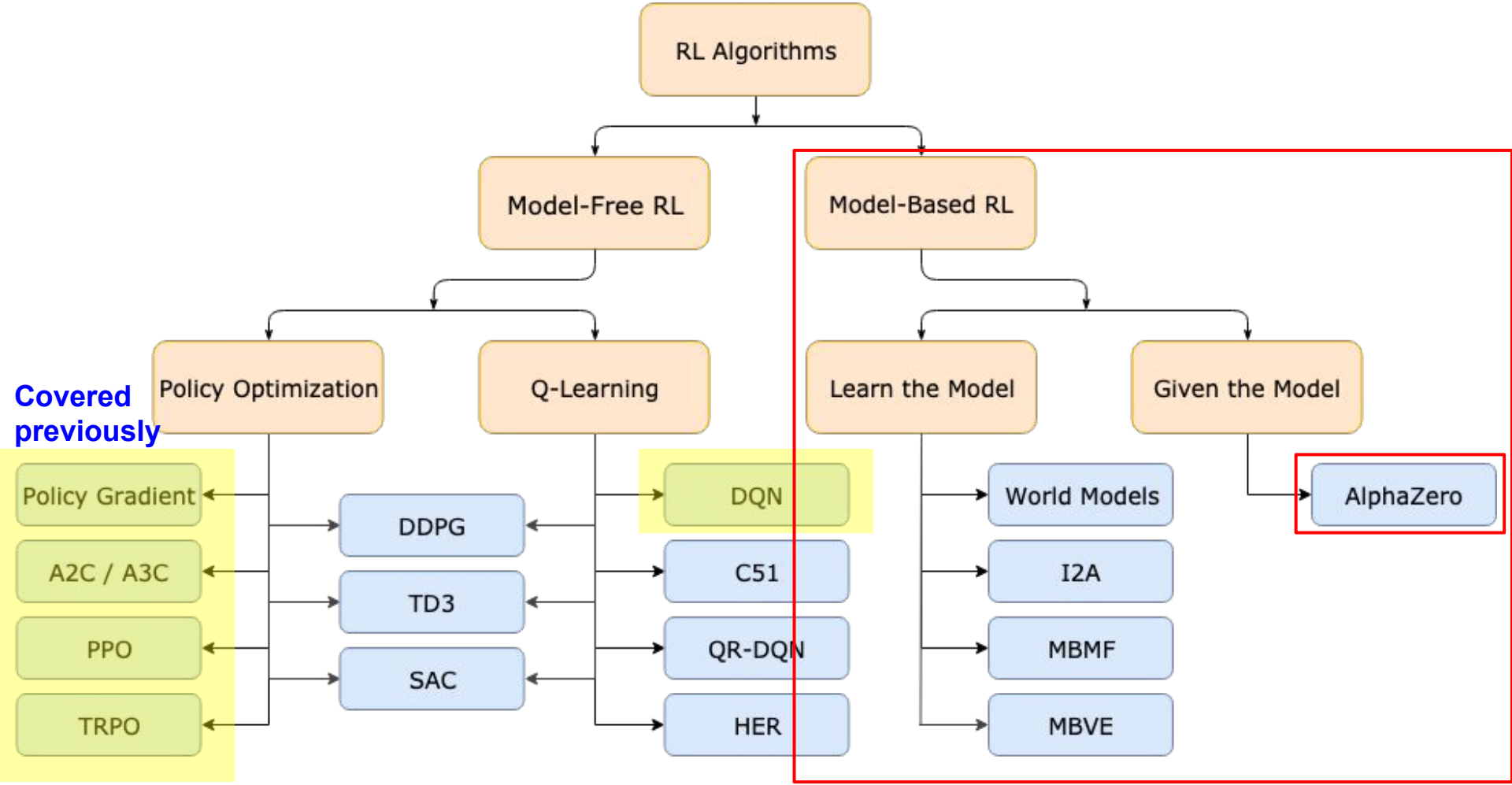
DDPG:

1. take some action $\mathbf{a}_i$ and observe $(\mathbf{s}_i, \mathbf{a}_i, \mathbf{s}'_i, r_i)$, add it to $\mathcal{B}$
2. sample mini-batch $\{\mathbf{s}_j, \mathbf{a}_j, \mathbf{s}'_j, r_j\}$ from $\mathcal{B}$ uniformly
3. compute $y_j = r_j + \gamma \max_{\mathbf{a}'_j} Q_{\phi'}(\mathbf{s}'_j, \mu_{\theta'}(\mathbf{s}'_j))$ using *target* nets $Q_{\phi'}$ and $\mu_{\theta'}$
4. $\phi \leftarrow \phi - \alpha \sum_j \frac{dQ_\phi}{d\phi}(\mathbf{s}_j, \mathbf{a}_j)(Q_\phi(\mathbf{s}_j, \mathbf{a}_j) - y_j)$
5. $\theta \leftarrow \theta + \beta \sum_j \frac{d\mu}{d\theta}(\mathbf{s}_j) \frac{dQ_\phi}{d\mathbf{a}}(\mathbf{s}_j, \mathbf{a})$
6. update ϕ' and θ' (e.g., Polyak averaging)

Slide credit: Sergey Levine

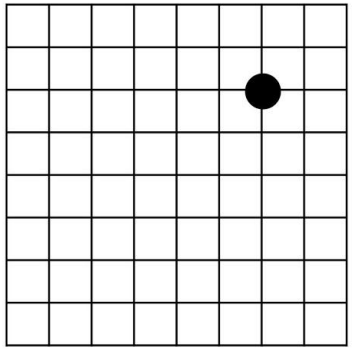
Outline

- Model-Free RL (for continuous control)
  - CEM, CMA-ES, NAF, DDPG
- **Model-Based RL**
  - **Using the true model**
    - **AlphaGo**
    - AlphaGo Zero
  - Learning the model
    - Dyna-Q
    - MPC
    - MBMF


RL Algorithms
Model-Free RL
Model-Based RL
Policy Optimization
Q-Learning
Learn the Model
Given the Model
Covered previously
Policy Gradient
A2C / A3C
PPO
TRPO
DDPG
TD3
SAC
DQN
C51
QR-DQN
HER
World Models
I2A
MBMF
MBVE
AlphaZero

Computer Go AI - Definition

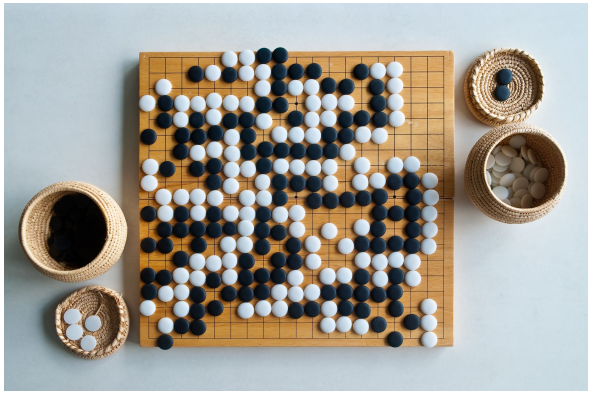
d = 1



s (state)

=

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

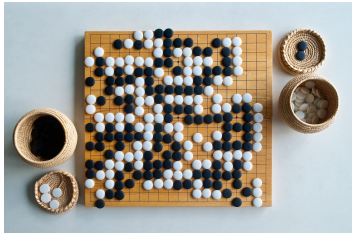


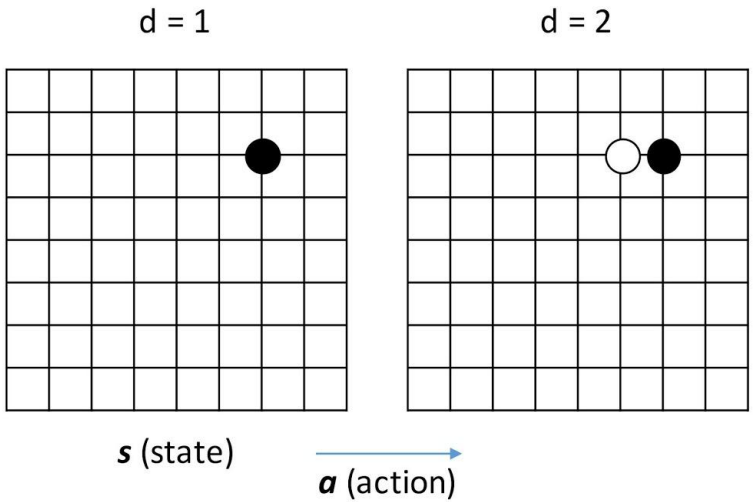
(e.g. we can represent the board into a matrix-like form)

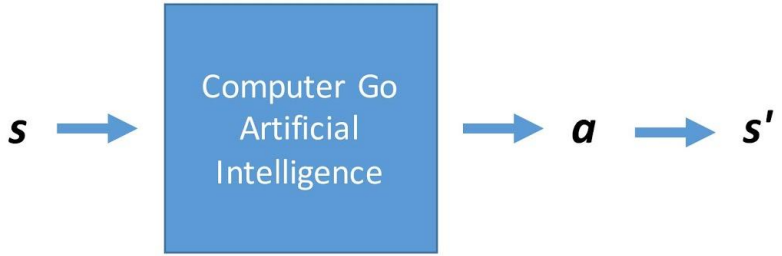
Slide credit: Shane Moon

**** The actual model uses other features than board positions as well***

Computer Go AI - Definition



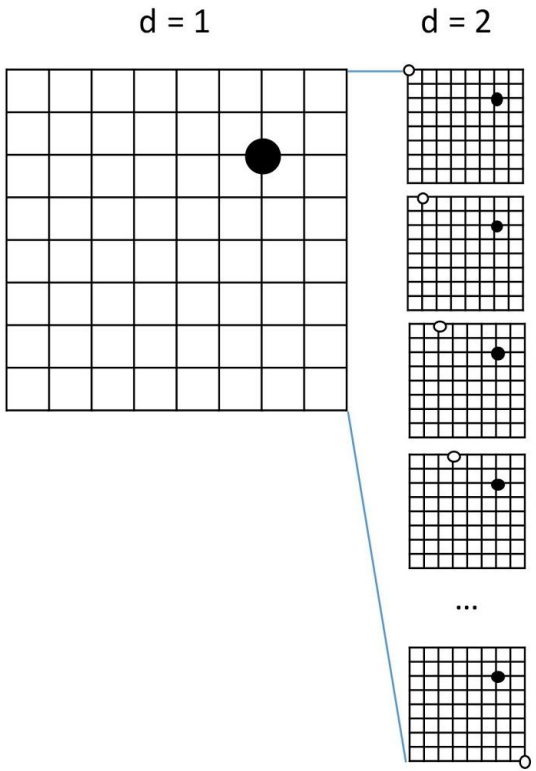




Given ***s***, pick the best ***a***

Slide credit: Shane Moon

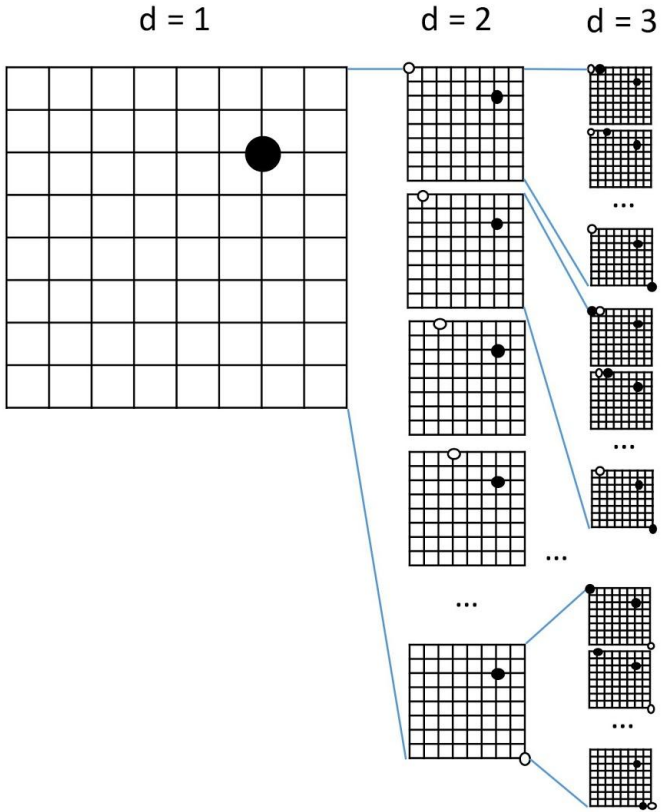
Computer Go AI - An Implementation Idea?



How about simulating all possible board positions?

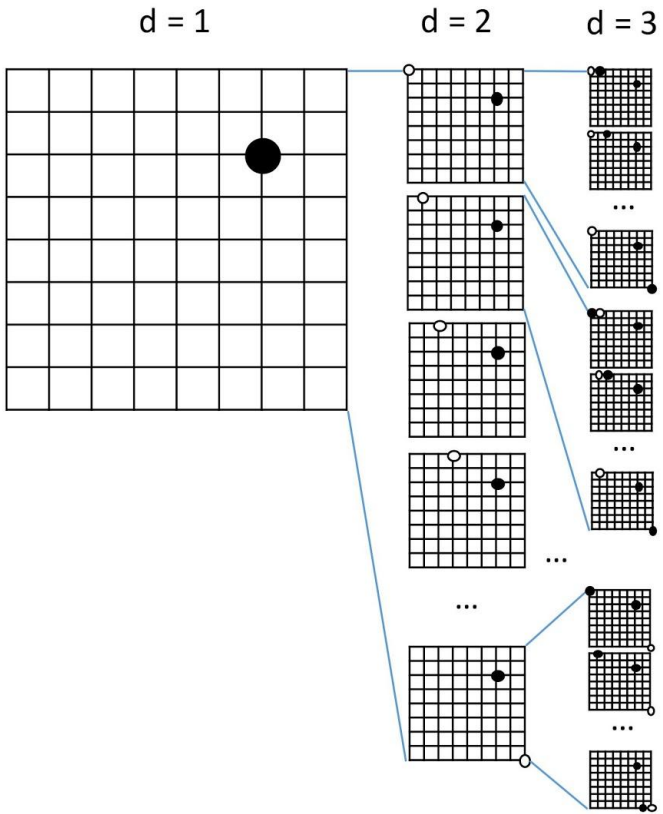
Slide credit: Shane Moon

Computer Go AI - An Implementation Idea?



Slide credit: Shane Moon

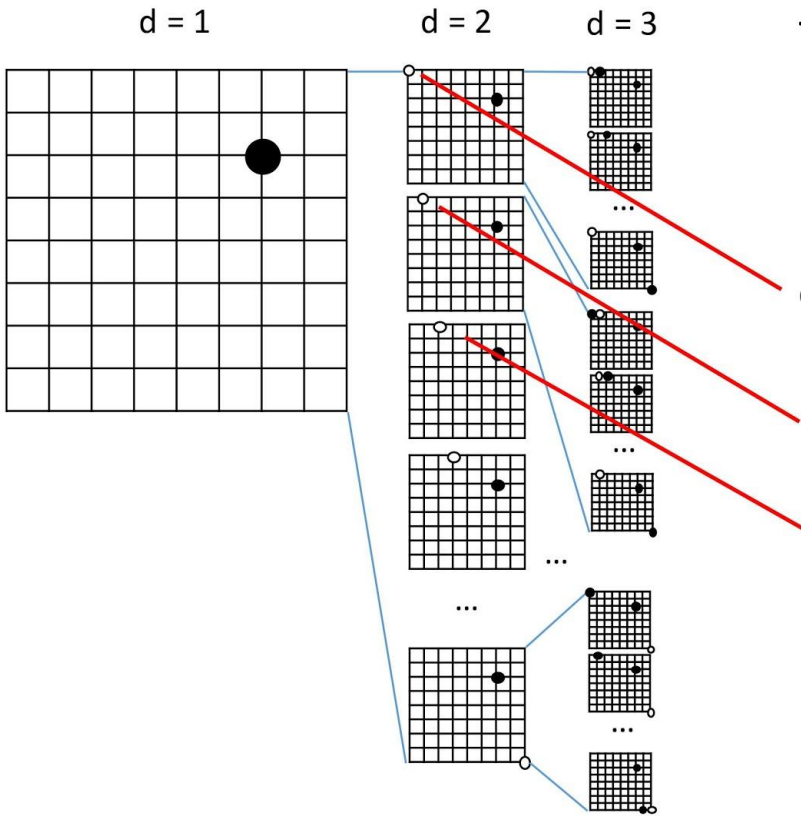
Computer Go AI - An Implementation Idea?



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

Slide credit: Shane Moon

Computer Go AI - An Implementation Idea?



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

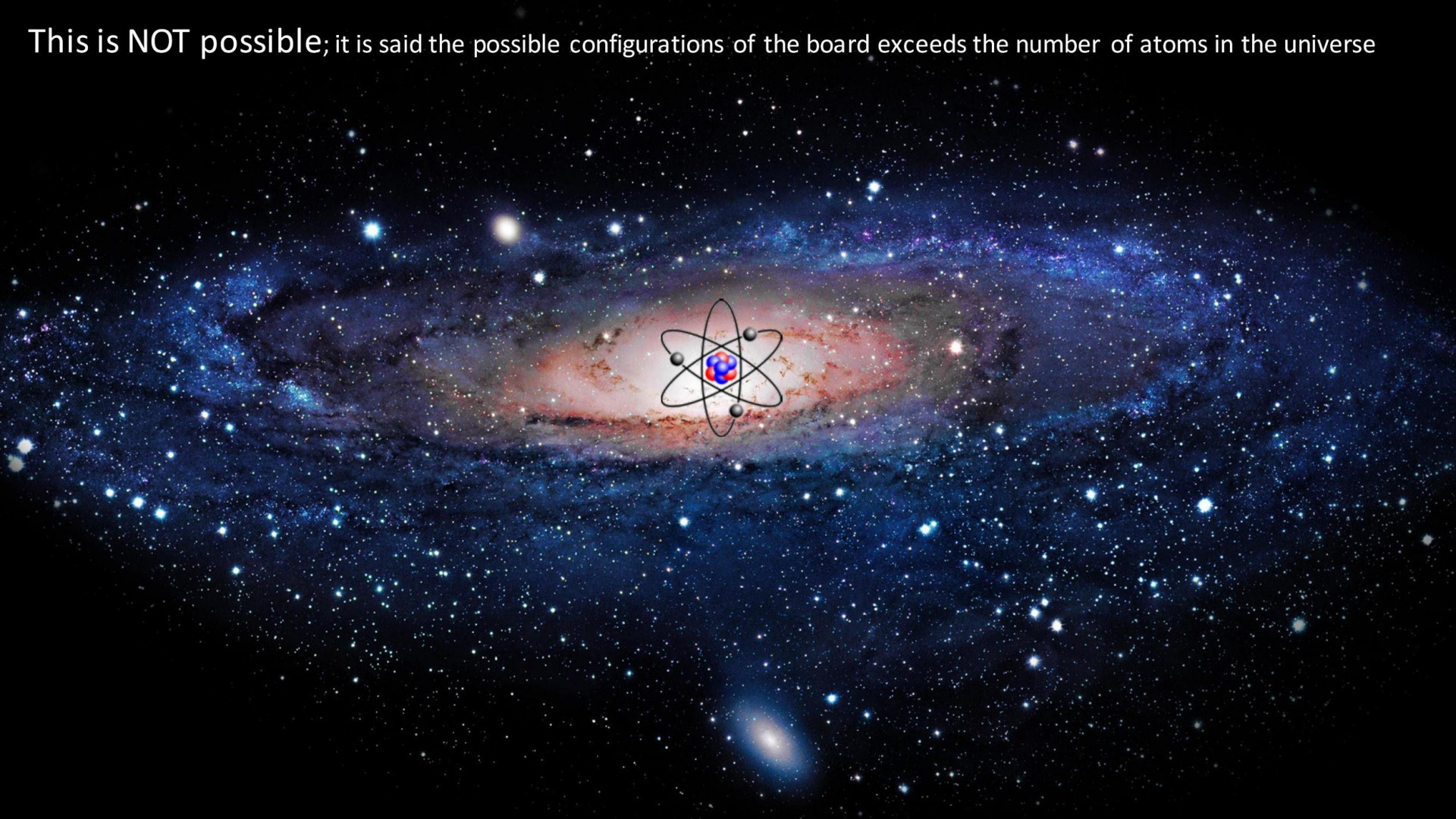
37,839 times

431,320 times

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

Slide credit: Shane Moon

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

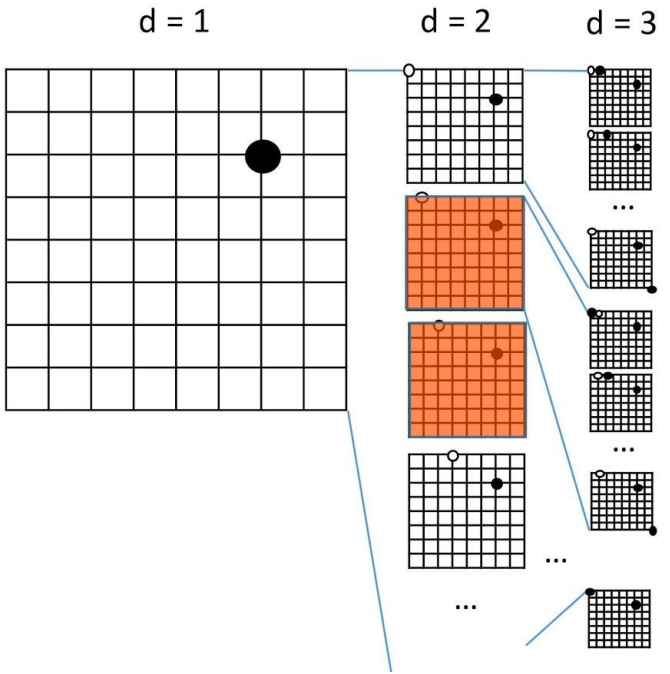


Key: Reduce Search Space

Slide credit: Shane Moon

Reducing Search Space

1. Reducing "action candidates" (Breadth Reduction)

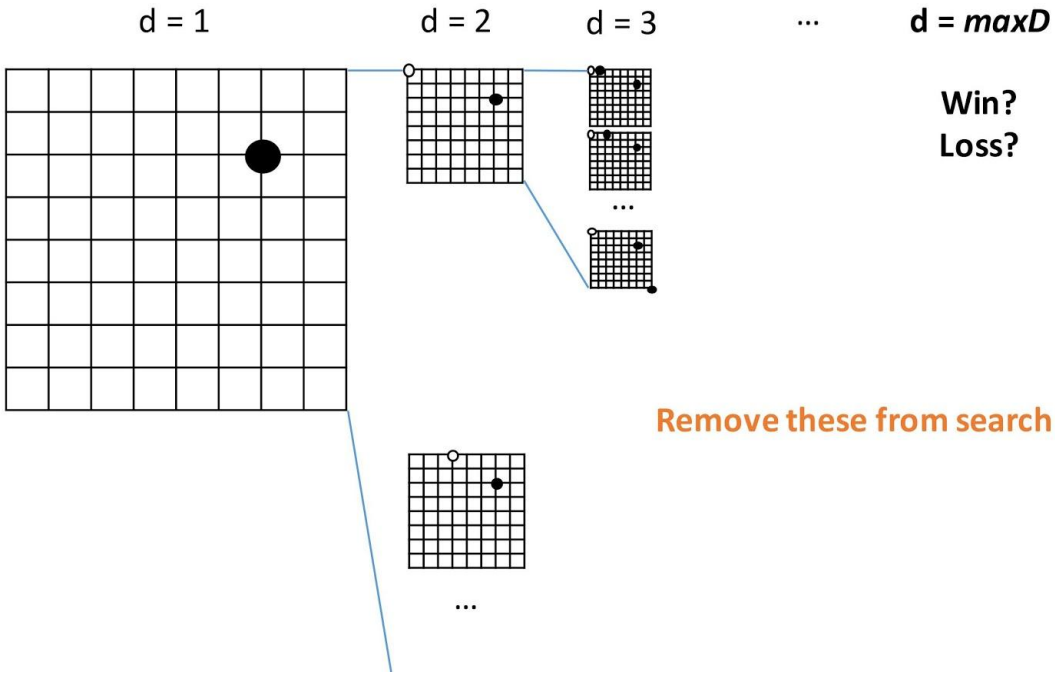


IF there is a model that can tell you that these moves are not common / probable (e.g. by experts, etc.) ...

Slide credit: Shane Moon

Reducing Search Space

1. Reducing “action candidates” (Breadth Reduction)

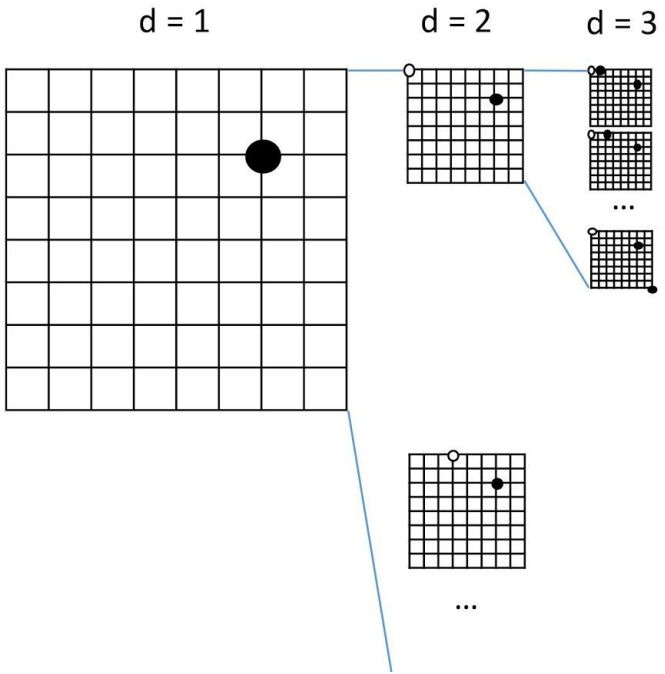


Remove these from search candidates in advance (breadth reduction)

Slide credit: Shane Moon

Reducing Search Space

2. Position evaluation ahead of time (Depth Reduction)



... d = *maxD*

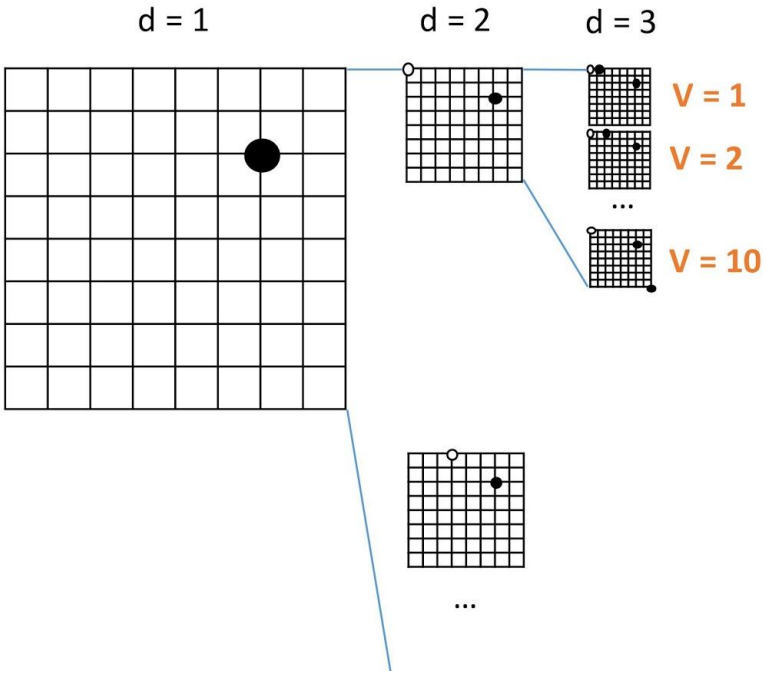
Win?
Loss?

Instead of simulating until the maximum depth ..

Slide credit: Shane Moon

Reducing Search Space

2. Position evaluation ahead of time (Depth Reduction)



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

Slide credit: Shane Moon

Reducing Search Space

1. **Reducing “action candidates” (Breadth Reduction)**

2. **Position evaluation ahead of time (Depth Reduction)**

Slide credit: Shane Moon

Reducing “Action Candidates”

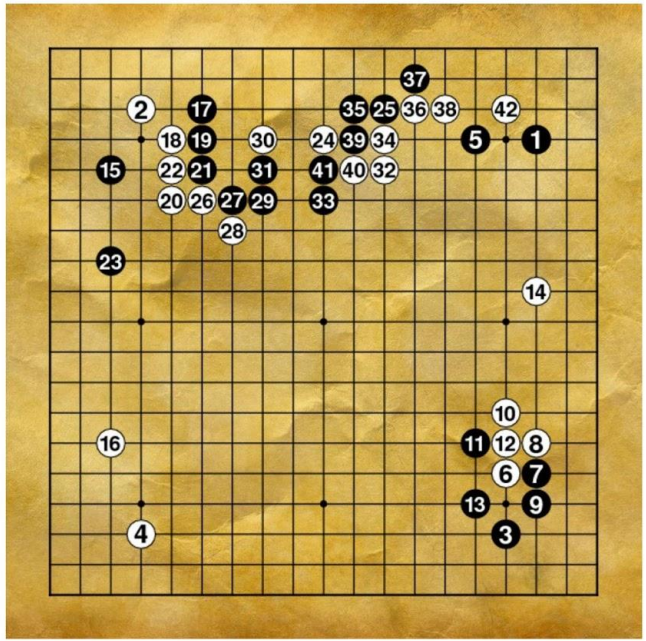
Learning: **P (next action | current state)**

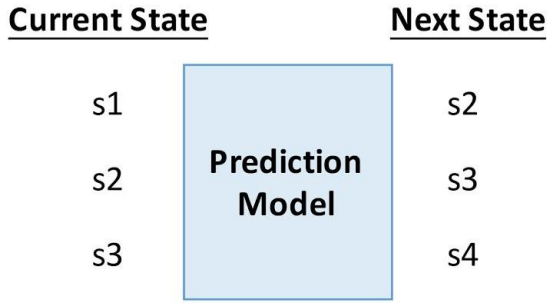
= P (a | s)

Slide credit: Shane Moon

Reducing “Action Candidates”

(1) Imitating expert moves (supervised learning)





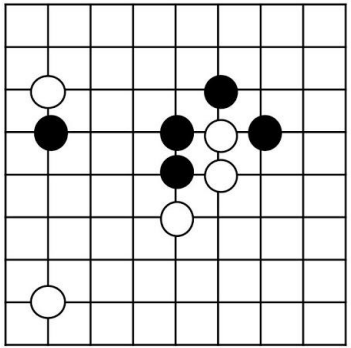
Data: Online Go experts (5~9 dan)
160K games, 30M board positions

Slide credit: Shane Moon

Reducing “Action Candidates”

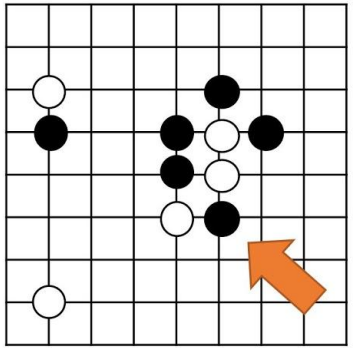
(1) Imitating expert moves (supervised learning)

Current Board



Prediction Model

Next Board

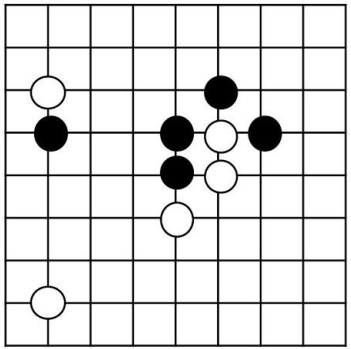


Slide credit: Shane Moon

Reducing “Action Candidates”

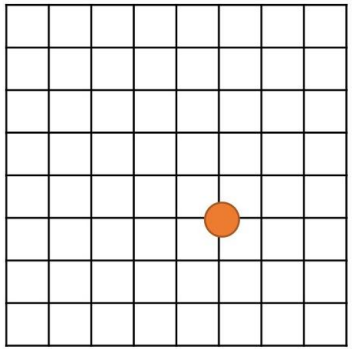
(1) Imitating expert moves (supervised learning)

Current Board



Prediction Model

Next Action

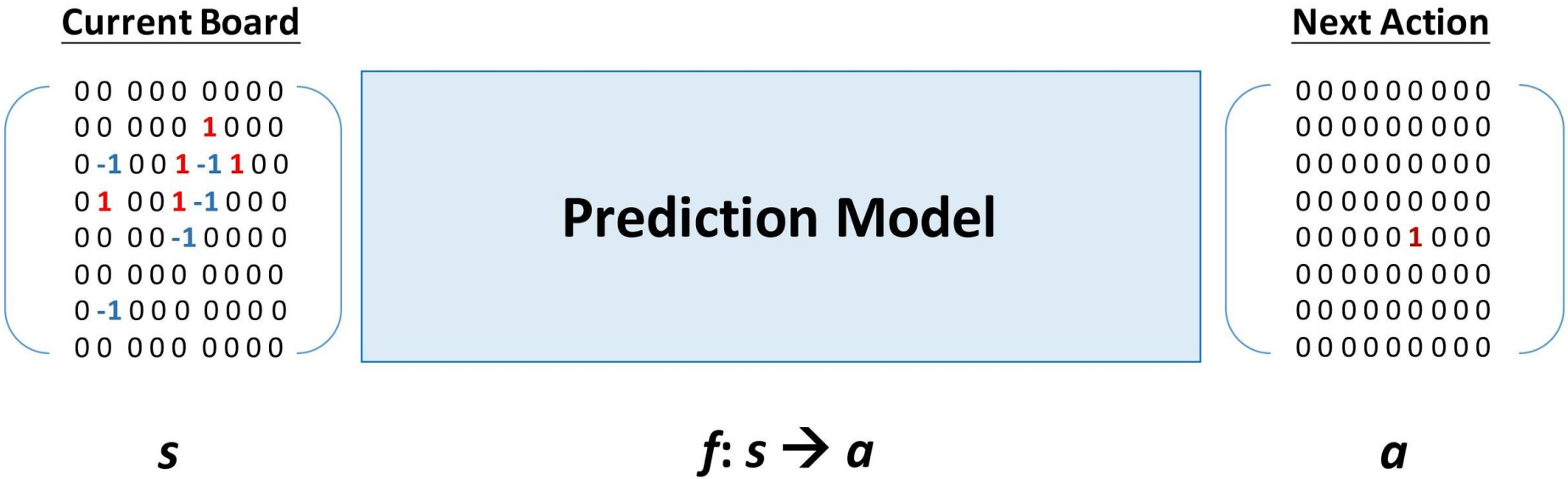


There are 19 X 19 = 361 possible actions (with different probabilities)

Slide credit: Shane Moon

Reducing “Action Candidates”

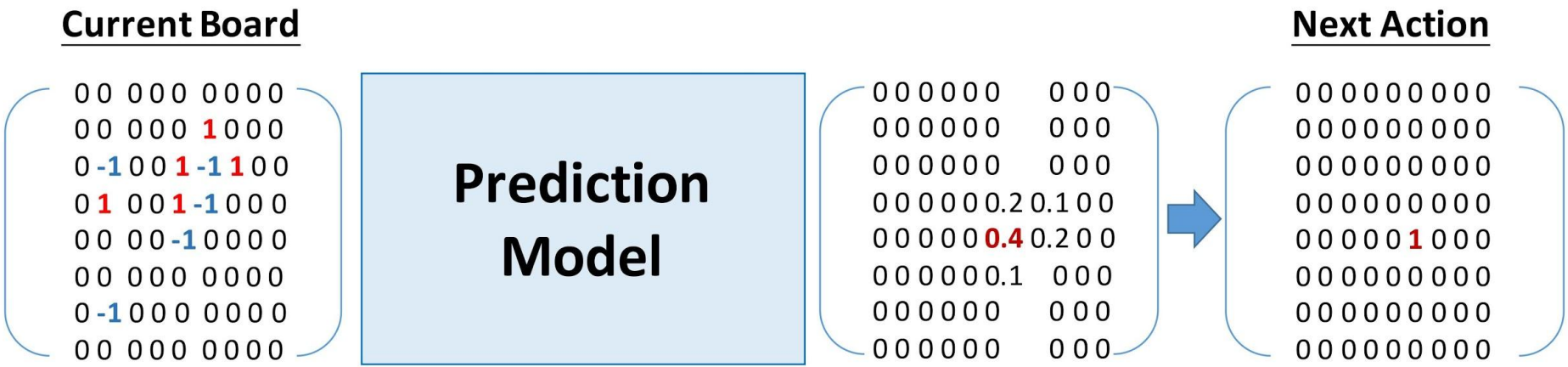
(1) Imitating expert moves (supervised learning)



Slide credit: Shane Moon

Reducing “Action Candidates”

(1) Imitating expert moves (supervised learning)



Slide credit: Shane Moon

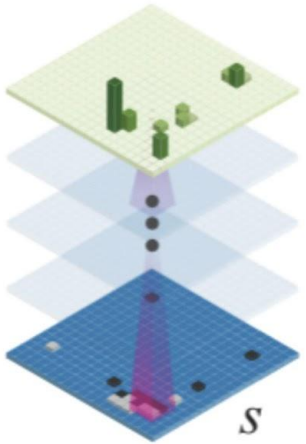
Reducing “Action Candidates”

(1) Imitating expert moves (supervised learning)

Current Board

```
00 000 0000
00 000 1000
0-100 1-110 0
01 001-1000
00 00-10000
00 000 0000
0-1000 0000
00 000 0000
```

Prediction Model



Next Action

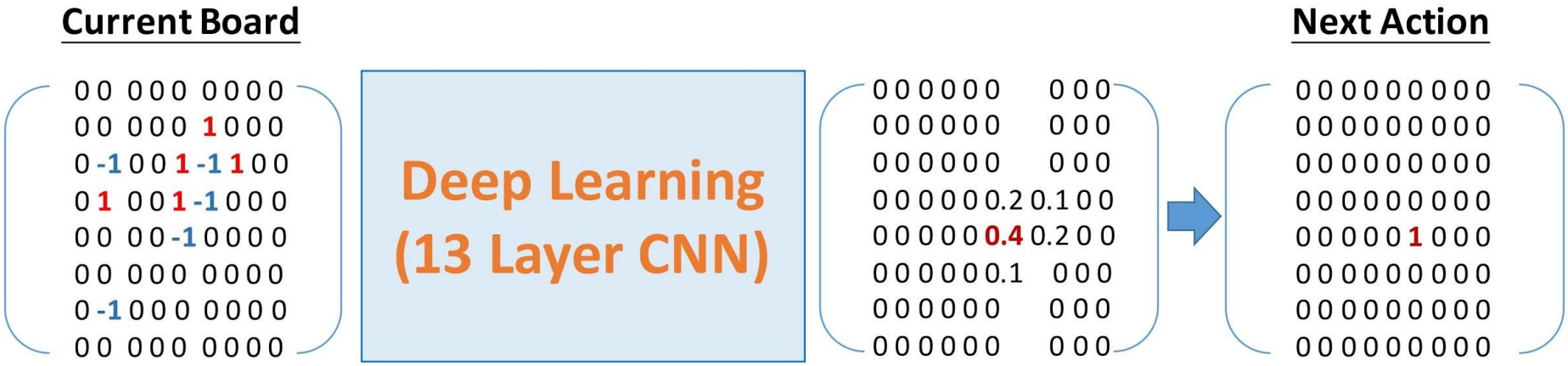
```
000000000
000000000
000000000
000000000
000010000
000000000
000000000
000000000
```

s ***g*: *s* → p(*a*|*s*)** **p(*a*|*s*)** **argmax** ***a***

Slide credit: Shane Moon

Reducing “Action Candidates”

(1) Imitating expert moves (supervised learning)



Slide credit: Shane Moon

Go: abstraction is the key to win

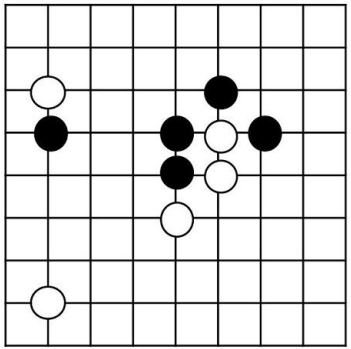
CNN: abstraction is its *forte*

Slide credit: Shane Moon

Reducing “Action Candidates”

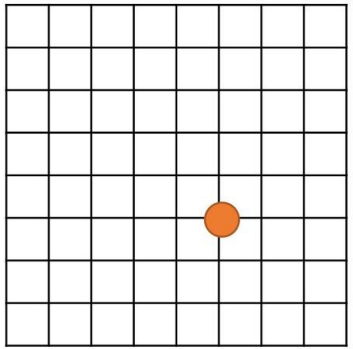
(1) Imitating expert moves (supervised learning)

Current Board



Expert Moves Imitator Model (w/ CNN)

Next Action



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

Slide credit: Shane Moon

Reducing “Action Candidates”

(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)

Slide credit: Shane Moon

Reducing “Action Candidates”

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

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

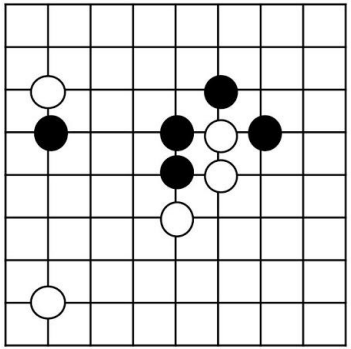
Return: board positions, win/lose info

Slide credit: Shane Moon

Reducing “Action Candidates”

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

Board position



Expert Moves Imitator Model (w/ CNN)

win/loss

Loss

z = -1

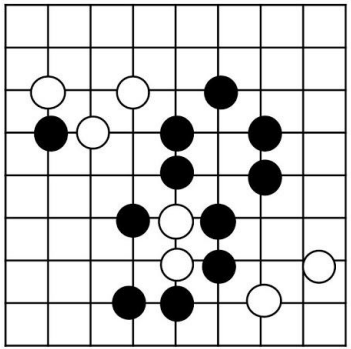
Training: $\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t$.

Slide credit: Shane Moon

Reducing “Action Candidates”

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

Board position



Expert Moves Imitator Model (w/ CNN)

win/loss

Win

z = +1

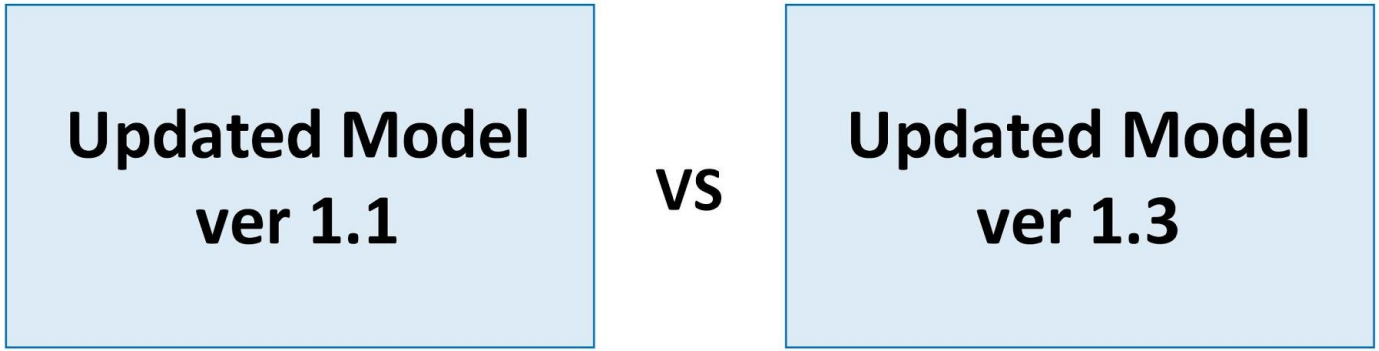
Training: $\Delta\rho \propto \frac{\partial \log p_\rho(a_t|s_t)}{\partial \rho} z_t$

Slide credit: Shane Moon

Reducing “Action Candidates”

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

Older models vs. newer models



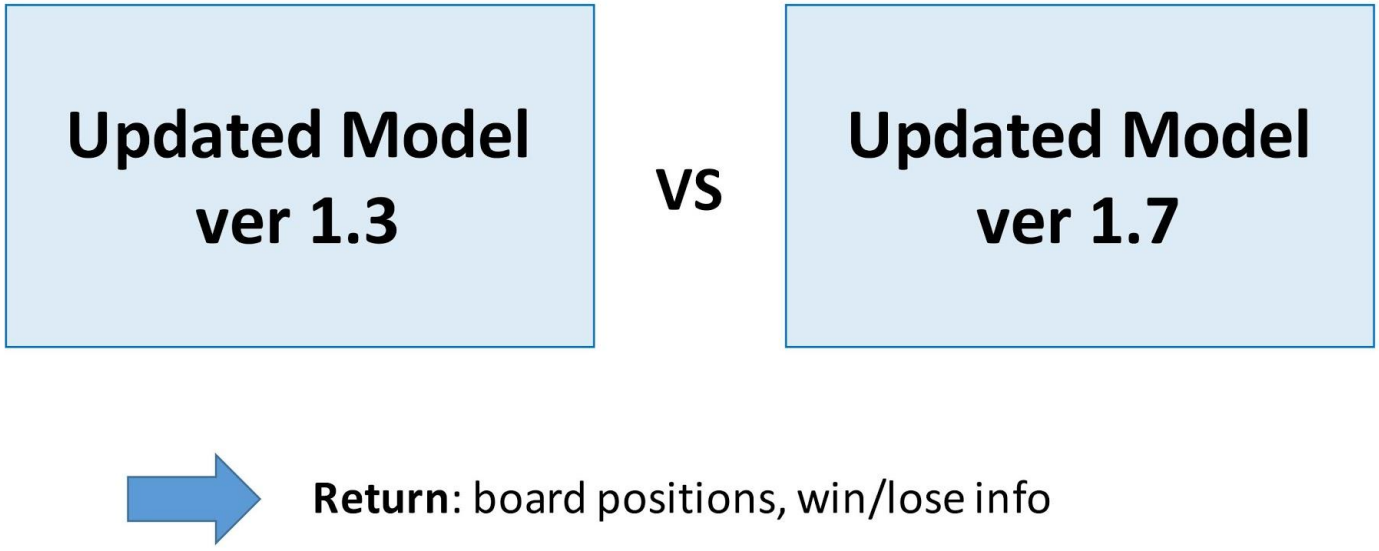
It uses the same topology as the expert moves imitator model, and just uses the updated parameters

Return: board positions, win/lose info

Slide credit: Shane Moon

Reducing “Action Candidates”

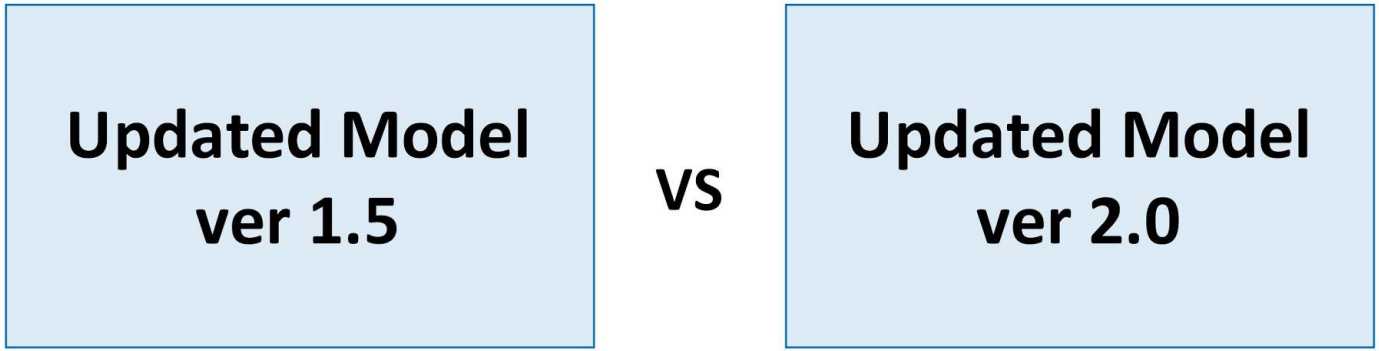
(2) Improving through self-plays (reinforcement learning)



Slide credit: Shane Moon

Reducing “Action Candidates”

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

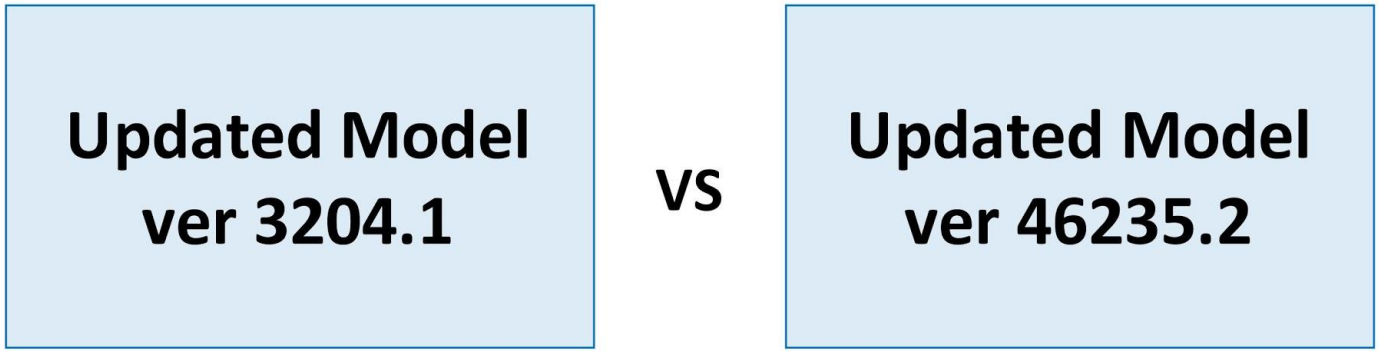


Return: board positions, win/lose info

Slide credit: Shane Moon

Reducing “Action Candidates”

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



Return: board positions, win/lose info

Slide credit: Shane Moon

Reducing “Action Candidates”

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

Expert Moves Imitator Model

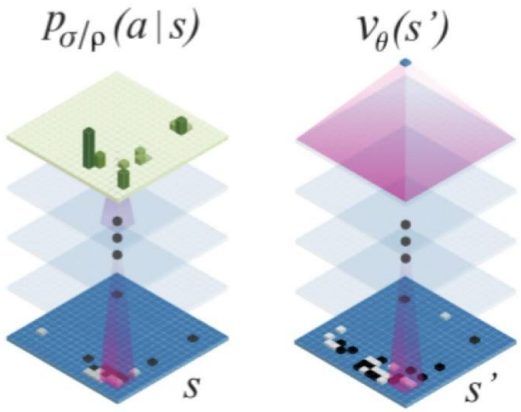
VS

Updated Model ver 1,000,000

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

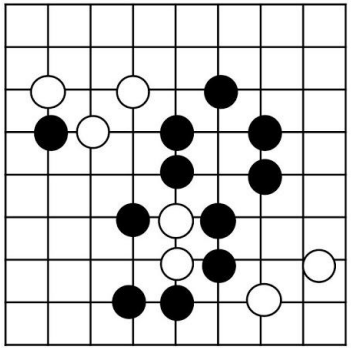
Slide credit: Shane Moon

Board Evaluation



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)

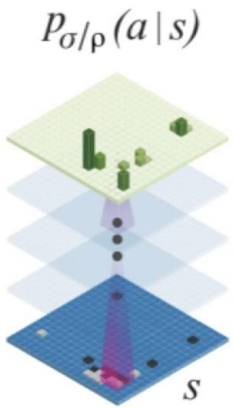
Training: $\Delta\theta \propto \frac{\partial v_\theta(s)}{\partial\theta}(z - v_\theta(s))$

Slide credit: Shane Moon

Reducing Search Space

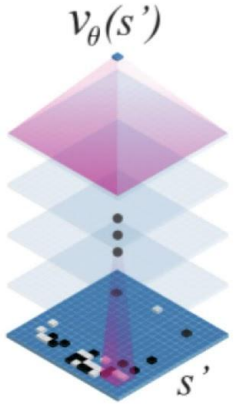
1. **Reducing "action candidates" (Breadth Reduction)**

   Policy Network



2. **Board Evaluation (Depth Reduction)**

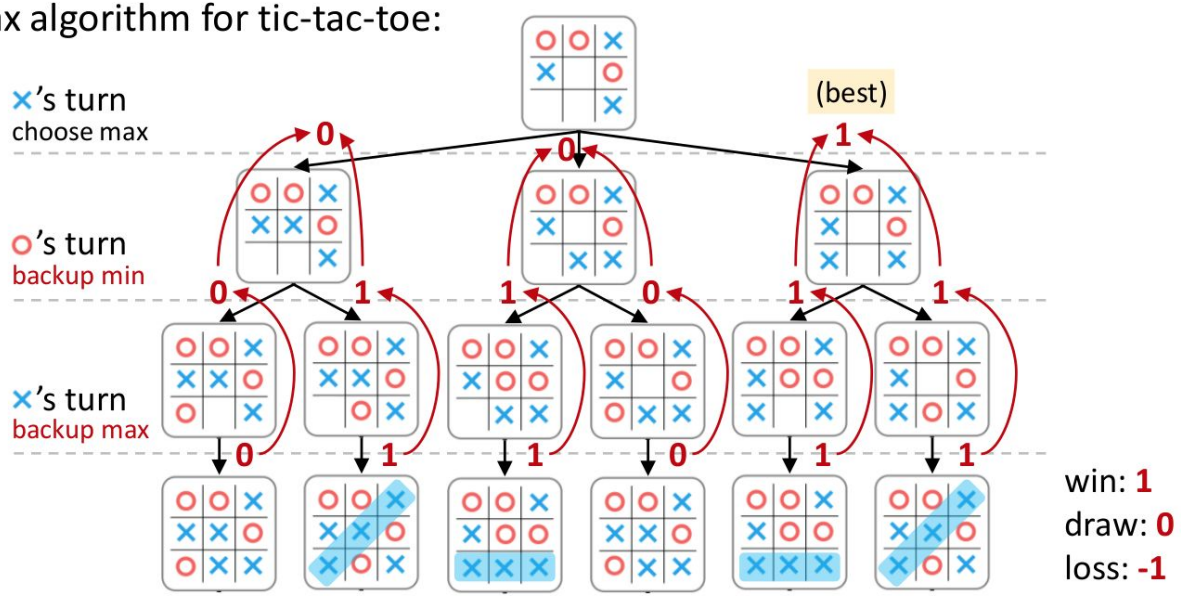
   Value Network



Slide credit: Shane Moon

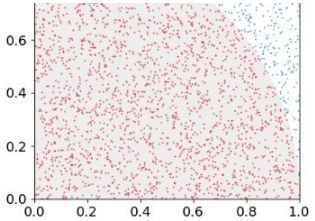
Game Tree

- **Game tree**: directed graph whose nodes are positions (states) in a game and whose edges are actions (moves)
  - Start from a position, we can **search** the tree to find the best next action
  - E.g., minimax algorithm for tic-tac-toe:



Slide credit: Chia-Ching Lin

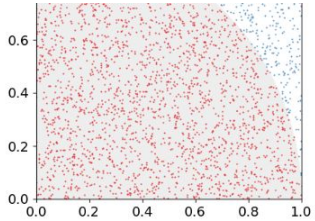
Monte Carlo Game Tree



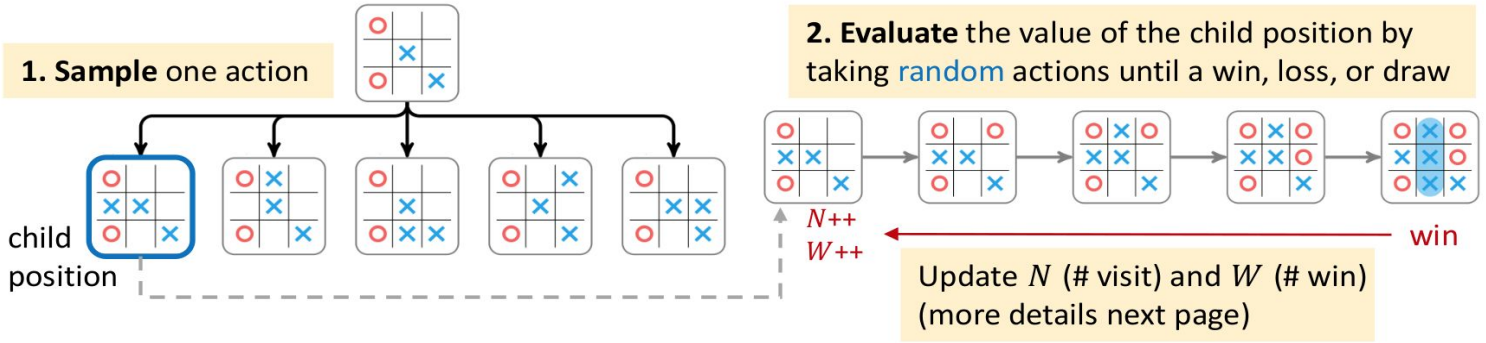
- Instead of considering all b^d sequences of actions, MCTS applies **Monte Carlo methods** that rely on repeated random sampling to obtain numerical results
- At each position s, MCTS runs many **simulations** to find the (approximately) best action
  - In each simulation, 2 general principles are applied to reduce the search space
    1. **Action sampling**: reduce b by sampling high-probability actions from a policy $p(a|s)$ that is a probability distribution over possible actions a in position s
    2. **Position evaluation**: reduce d by truncating the search tree at state s and replacing the subtree below s by an approximate value that predicts the outcome from state s

Slide credit: Chia-Ching Lin

Monte Carlo Game Tree



- Instead of considering all b^d sequences of actions, MCTS applies **Monte Carlo methods** that rely on repeated random sampling to obtain numerical results
- At each position s, MCTS runs many **simulations** to find the (approximately) best action
  - In each simulation, 2 general principles are applied to reduce the search space
    - E.g., within one simulation in tic-tac-toe:



Slide credit: Chia-Ching Lin

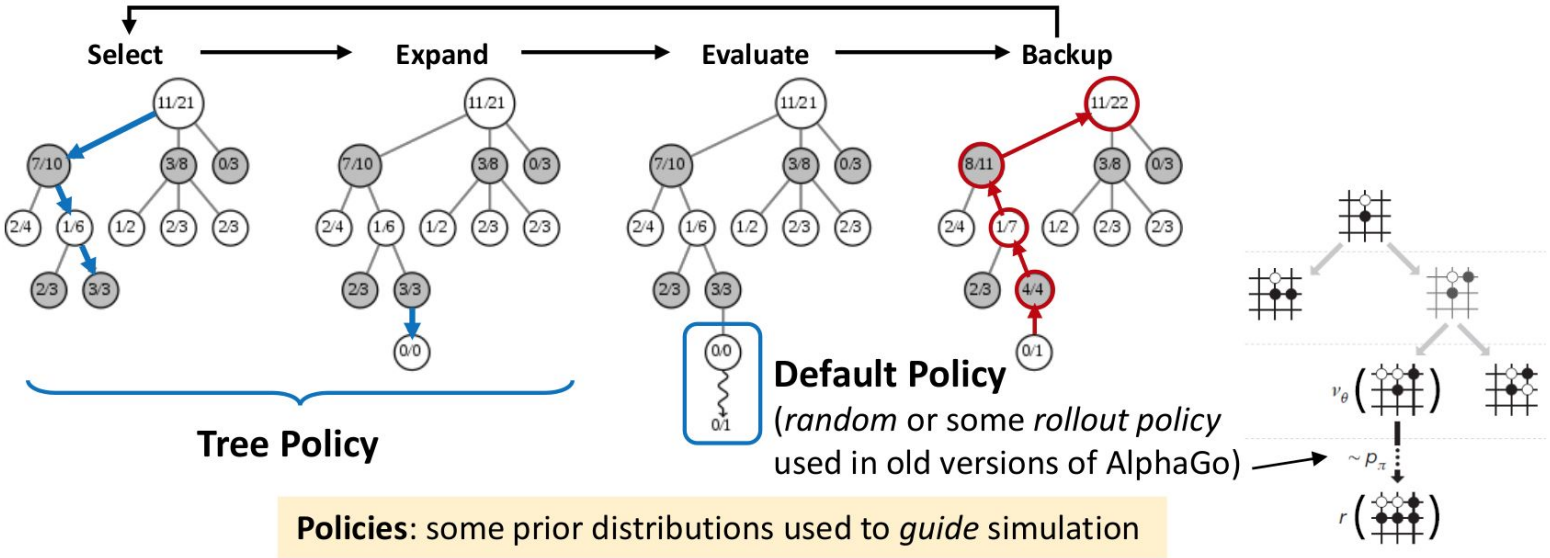
General Idea of Monte Carlo Tree Search

- MCTS iteratively builds partial search tree using 4 main steps per simulation
  - **Select**: traverse the partial search tree until leaf by sampling
  - **Expand**: sample one more action to add a new child
  - **Evaluate**: predict the outcome by a value function or rollout (playout)
    - No rollout in AlphaGo Zero (instead, it uses the value function directly)
  - **Backup**: use the **Evaluated** result to update statistics (# win / # visit) for all nodes in the path so that better nodes are more likely to be sampled in future simulations

Slide credit: Chia-Ching Lin

General Idea of Monte Carlo Tree Search

- Eventually, the most explored move (i.e., the move with the max # visit) is taken



Slide credit: Chia-Ching Lin

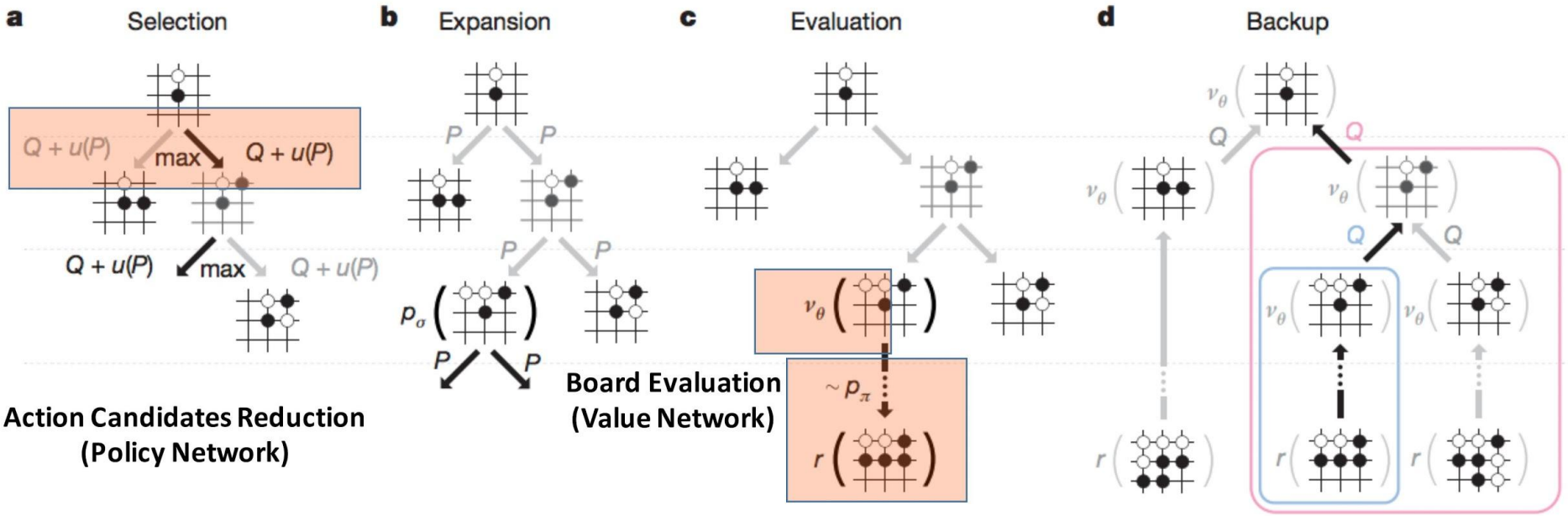
MCTS in AlphaGo

- Each node s in the search tree has edges (s, a) for all legal actions $a \in \mathcal{A}(s)$
- Statistics are stored for each edge (not for each node): visit count $N(s, a)$, total action value $W(s, a)$, mean action value $Q(s, a)$, and prior probability $P(s, a)$
  - Initially, $N(s, a) = W(s, a) = Q(s, a) = 0$, and $P(s, a) = p_a$ (from the network f_θ)

Slide credit: Chia-Ching Lin

MCTS in AlphaGo

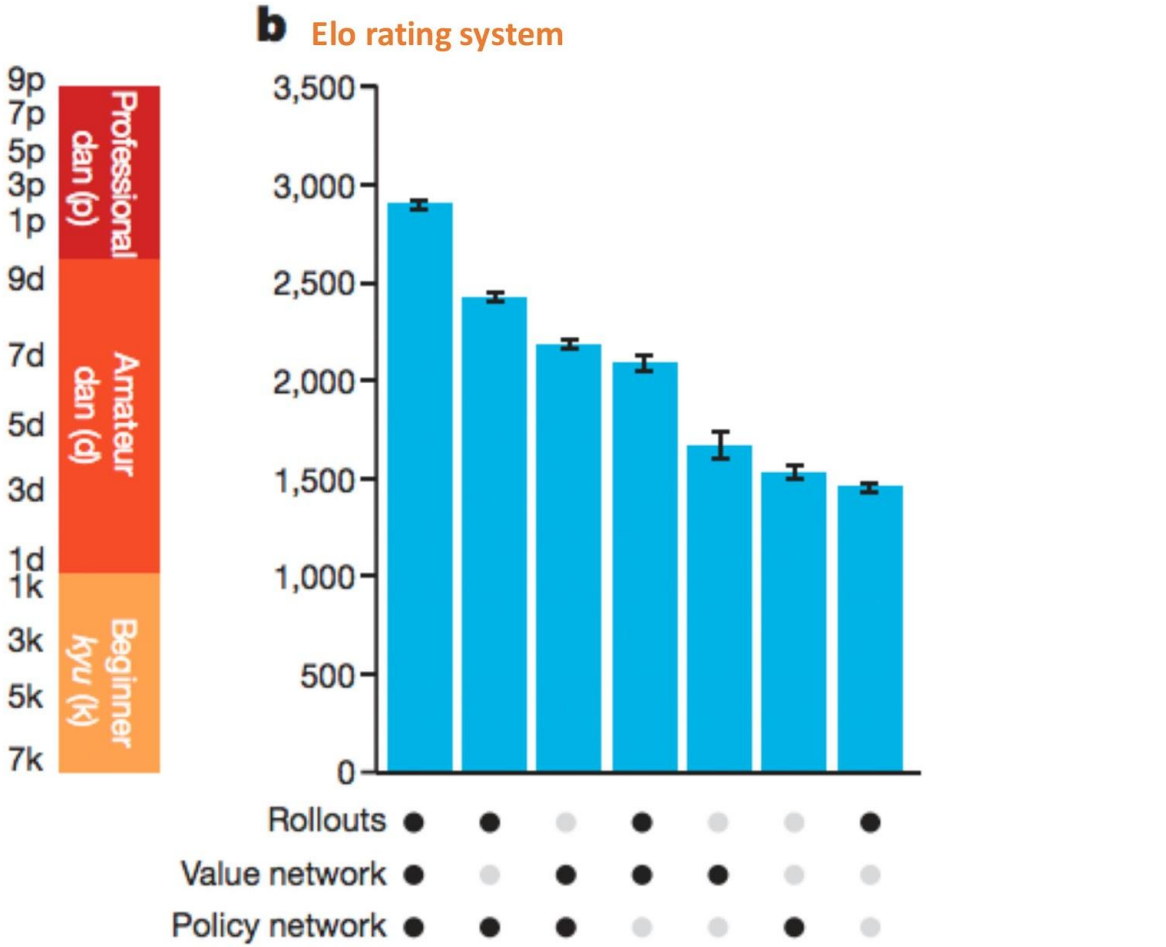
a: Choose the action with the max $Q + U$, where $U \propto P/(1 + N)$ (initially prefers actions with high prior probabilities and low visit counts, but asympotically prefers actions with high action values)



(Rollout): Faster version of estimating p(a|s)
→ uses shallow networks (3 ms → 2μs)

Slide credit: Shane Moon

Results



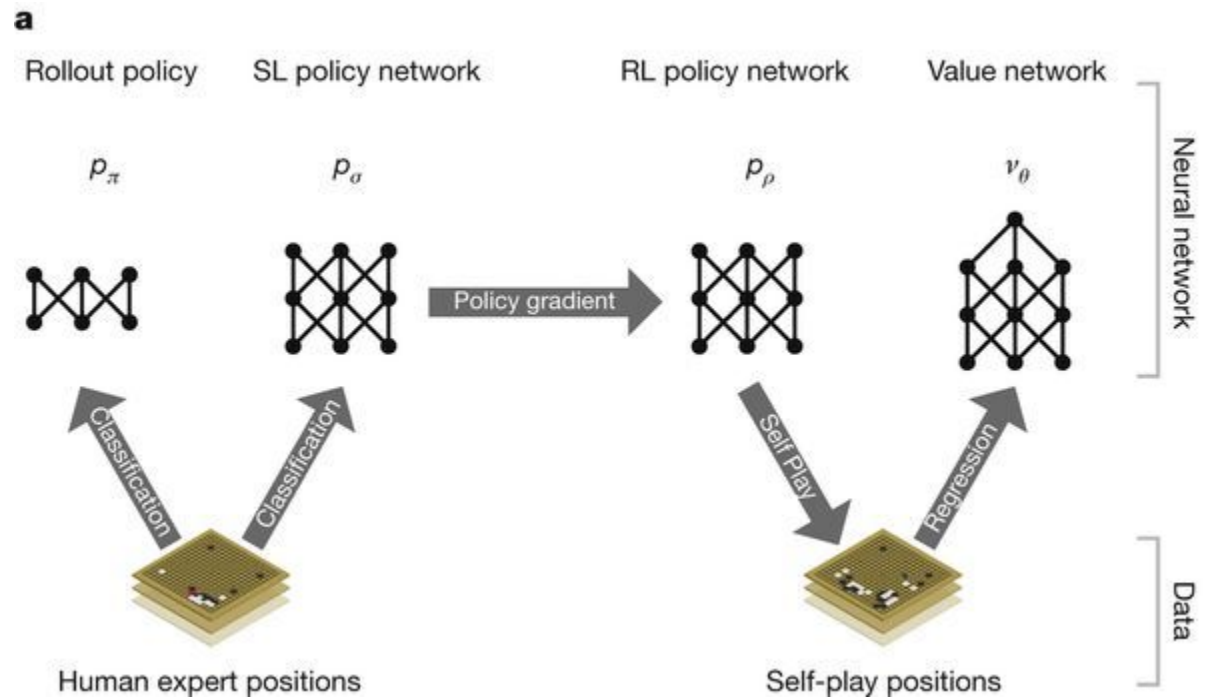
Slide credit: Shane Moon

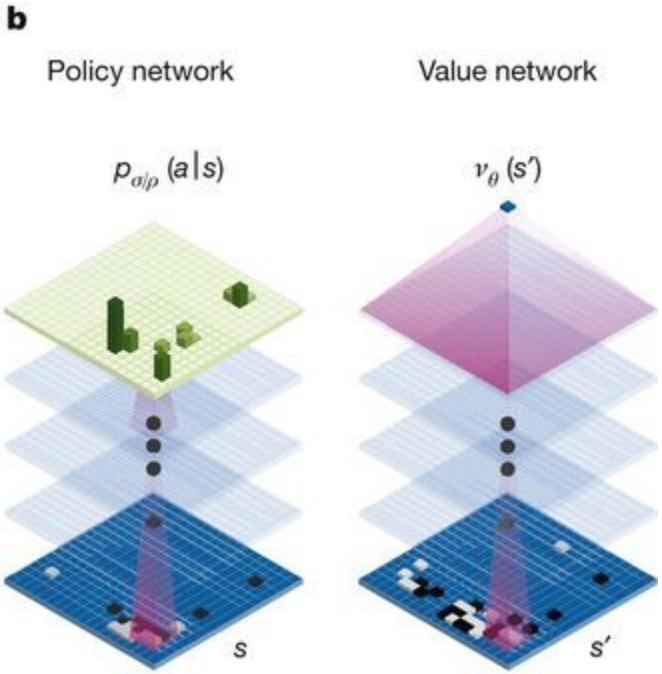
AlphaGo beating a Human Champion

AlphaGo versus Lee Sedol, also known as the **Google DeepMind Challenge Match**, was a five-game Go match between 18-time world champion Lee Sedol and AlphaGo, a computer Go program developed by Google DeepMind, played in Seoul, South Korea between 9 and 15 March 2016. AlphaGo won all but the fourth game;[1] all games were won by resignation.[2] The match has been compared with the historic chess match between Deep Blue and Garry Kasparov in 1997.

AlphaGo versus Lee Sedol - Wikipedia

Takeaways





Slide credit: Shane Moon

Quiz

- DDPG, as an extension of DQN to continuous action space, can be interpreted as a variant of actor-critic, but the input to critic network of DDPG is state-action pair instead of only state.
- Compared with DDPG, the Q network in continuous Q-learning with NAF has lower representational power.
- During MCTS in AlphaGo, consider the action selection step, the Q(s,a) is an output from the value network.
- During MCTS in AlphaGo, consider the evaluation step, the policy, which we use to take rollouts, is pre-trained with supervised learning and reinforcement learning.

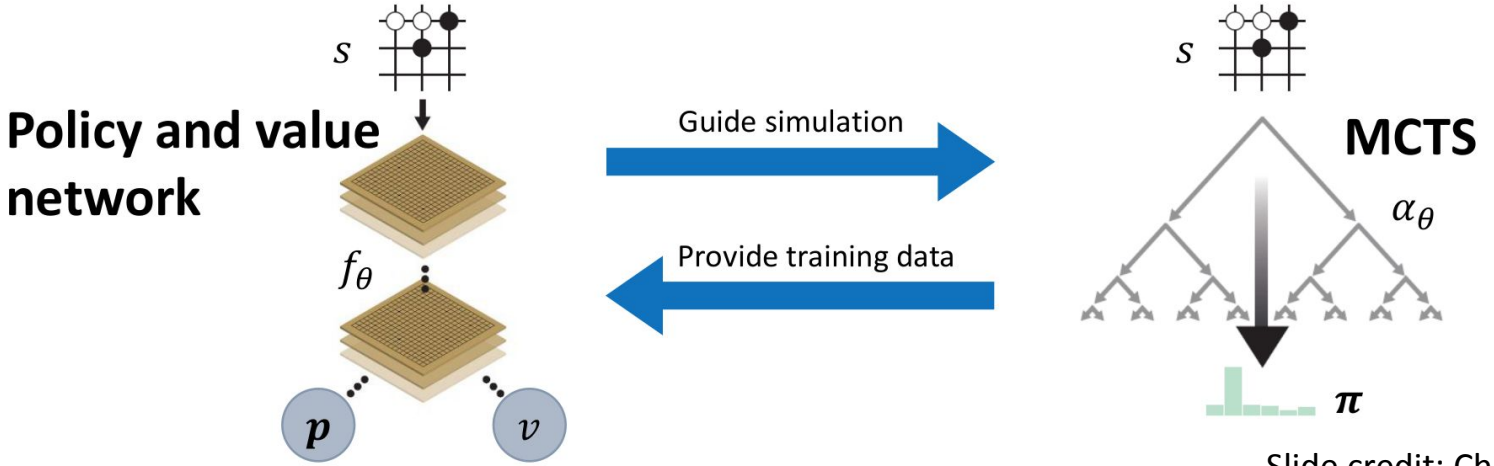
https://bit.ly/2YSduSg

Outline

- Model-Free RL (for continuous control)
  - CEM, CMA-ES, NAF, DDPG
- **Model-Based RL**
  - **Using the true model**
    - AlphaGo
    - **AlphaGo Zero**
  - Learning the model
    - Dyna-Q
    - MPC
    - MBMF

Two main components

- Two main components (work in parallel)
  1. A **policy and value network** f_θ (parameter: θ) takes as an input the raw board representation s of the position and its history, and outputs $\boldsymbol{p}$ and v
     - $\boldsymbol{p}$: probability of the next move (362-dim vector, including *pass*) from s
     - v: expected outcome of the current player from s (-1 (loss) ~ $+1$ (win))
  2. In each position s, a Monte Carlo Tree Search (**MCTS**) is executed, guided by f_θ, to output $\boldsymbol{\pi} = \alpha_\theta(s)$ (also a 362-dim probability vector) of playing each move from s



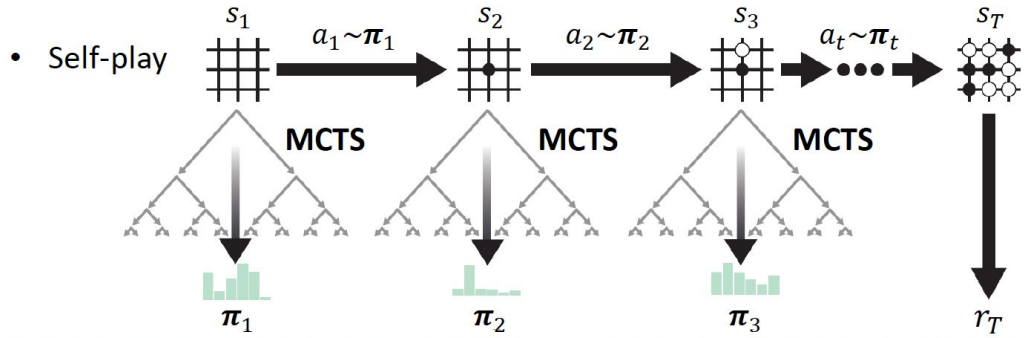
Slide credit: Chia-Ching Lin

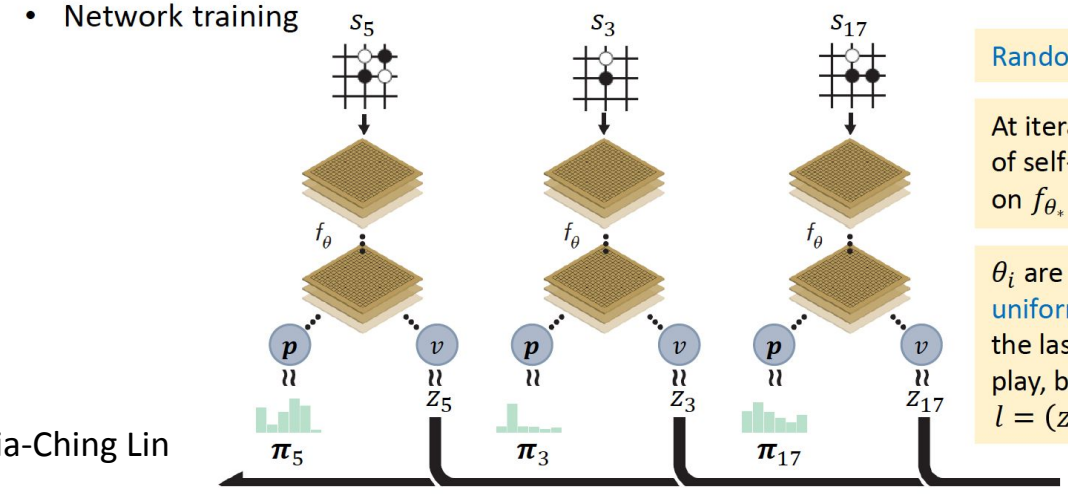
How to Train the networks

In each position s_t, an MCTS is executed to obtain $\boldsymbol{\pi}_t$, and an action a_t is sampled accordingly

From s_1 to s_T: A self-play *game* with a final reward $r_T = -1$ (loss) or $+1$ (win)

All $(s_t, \boldsymbol{\pi}_t, z_t)$ will be stored, where $z_t = \pm r_T$ with sign determined by the current player at step t





Random initialized weights θ_0

At iteration $i \geq 1$, 25,000 *games* of self-play are generated based on f_{θ_*} (➔ current best player α_{θ_*})

θ_i are trained using data sampled uniformly among all $(s, \boldsymbol{\pi}, z)$'s of the last 500,000 *games* of self-play, by minimizing loss: $l = (z - v)^2 - \boldsymbol{\pi}^T \log \boldsymbol{p} + c\|\theta\|^2$

Slide credit: Chia-Ching Lin

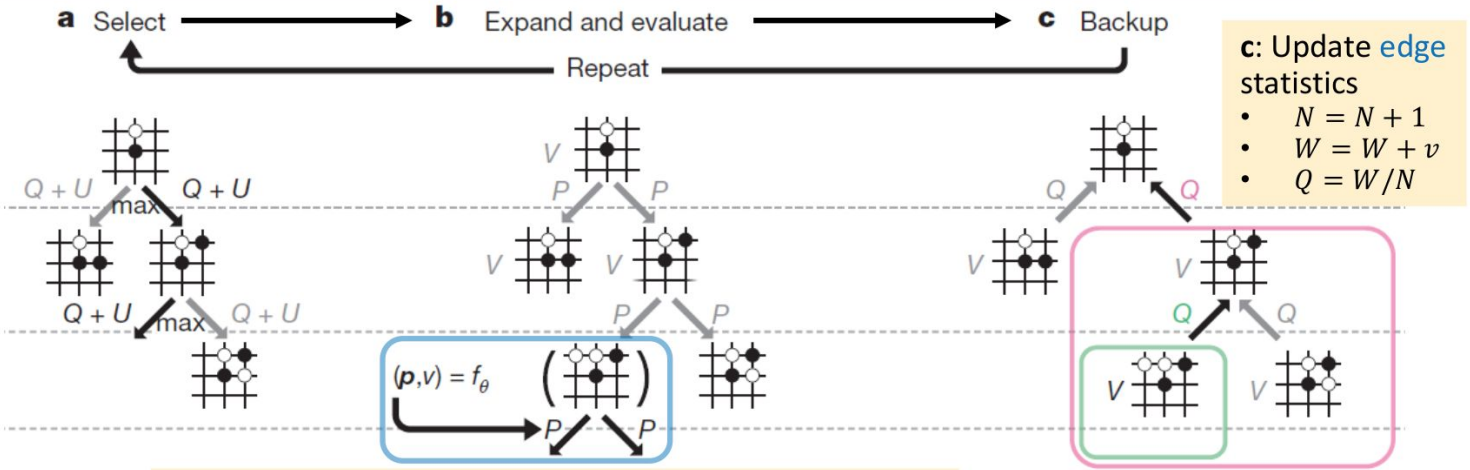
MCTS in AlphaGo Zero

Recall: Each node s in the search tree has edges (s, a) for all legal actions $a \in \mathcal{A}(s)$

Statistics are stored for each edge (not for each node): visit count $N(s, a)$, total action value $W(s, a)$, mean action value $Q(s, a)$, and prior probability $P(s, a)$

- Initially, $N(s, a) = W(s, a) = Q(s, a) = 0$, and $P(s, a) = p_a$ (from the network f_θ)

a: Choose the action with the max $Q + U$, where $U \propto P/(1 + N)$ (initially prefers actions with high prior probabilities and low visit counts, but asympotically prefers actions with high action values)



c: Update edge statistics

- $N = N + 1$
- $W = W + v$
- $Q = W/N$

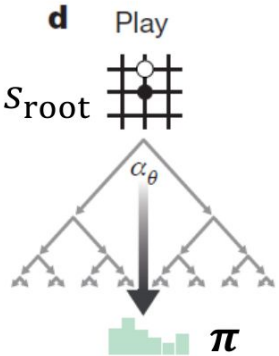
b: Whenever choosing an action that leads to a new leaf, it is added and evaluated (to decide its v and P's for all its legal actions) by the policy and value network f_θ

Eventually, good actions will have larger N than bad actions

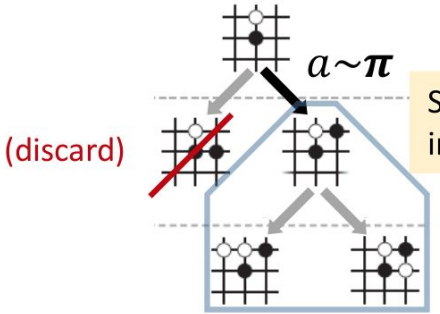
Slide credit: Chia-Ching Lin

Self-Play via MCTS

- At the end of the search (1,600 simulations), MCTS selects an action a to play in the root position s_{root}, with probability $\pi_a \propto N(s_{\text{root}}, a)^{1/\tau}$, where τ controls the level of exploration
  - The larger τ, the less differences between actions with different N ➔ exploration
  - $\tau \to 0$: choose the best move according to MCTS ➔ exploitation



d: In each round of MCTS, the 1,600 simulations only decide one move in the self-play game!



Search tree is reused in the next MCTS

all statistics retained

- The child node corresponding to the played action becomes the new root node, and another round of MCTS starts over from it with all edge statistics (N, W, Q, and P) of the subtree below this child being retained
- MCTS can thus be viewed as a self-play algorithm that, given neural network parameters θ and a root position s_{root}, computes a vector of search probabilities recommending moves to play, $\boldsymbol{\pi} = \alpha_\theta(s_{\text{root}})$

Slide credit: Chia-Ching Lin

Self-Play Details

- Some details about self-play
  - In each iteration, the best current player α_{θ_*} plays 25,000 games of self-play, using 1,600 simulations (0.4s) of MCTS to select each move
  - At step **a** (Select)
    - $U(s,a) = c_{\text{puct}} P(s,a) \frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$, where c_{puct} is a constant used to control exploration
    - Additional exploration is achieved by adding Dirichlet noise to the prior probabilities in the root node: $P(s_{\text{root}}, a) = (1-\varepsilon)p_a + \varepsilon\eta_a$, where $\boldsymbol{\eta} \sim \text{Dir}(0.03)$ and $\varepsilon = 0.25$
  - At step **b** (Expand and evaluate), a leaf node s_{leaf} will be randomly reflected or rotated before evaluated by the current network, $(d_i(\boldsymbol{p}), v) = f_\theta\big(d_i(s_{\text{leaf}})\big)$, where d_i is a dihedral reflection or rotation selected uniformly at random from $i \in [1..8]$
  - At step **d** (Play), $\tau = 1$ for the first 30 moves of each game, and $\tau \to 0$ for the remainder of the game
  - To save computation, AlphaGo Zero resigns from a self-play game if its root value and best child value are lower than a threshold value v_{resign}
    - v_{resign} is selected automatically to keep the fraction of false positives (games that could have been won if AlphaGo Zero had not resigned) below 5% (measured by disabling resignation in 10% of self-play games)

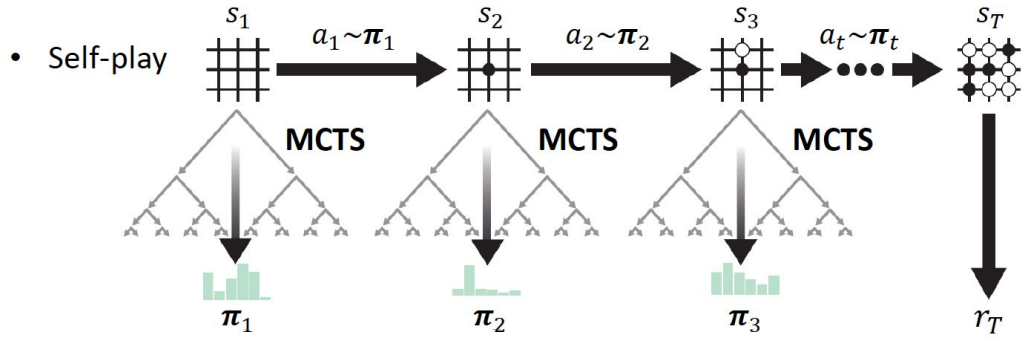
Slide credit: Chia-Ching Lin

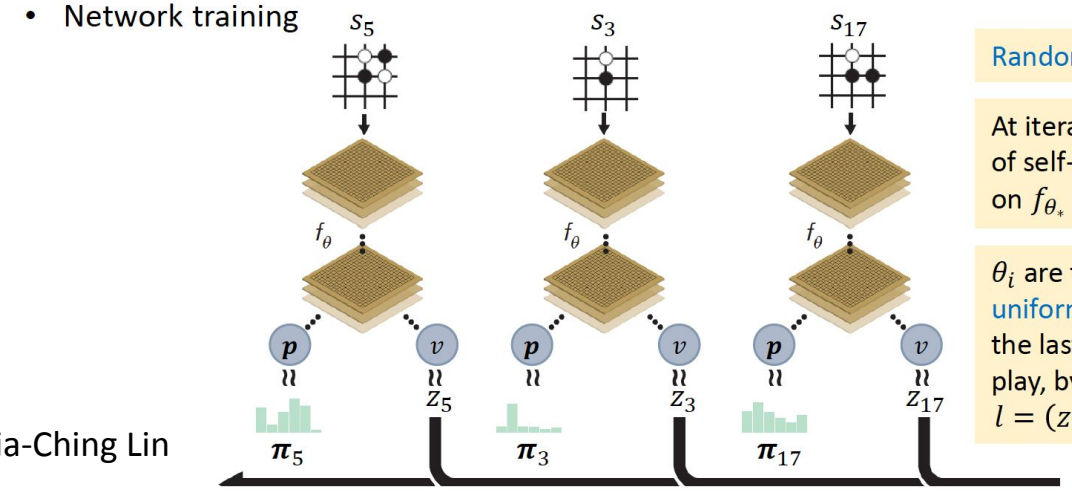
Recap: How to Train the networks

In each position s_t, an MCTS is executed to obtain $\boldsymbol{\pi}_t$, and an action a_t is sampled accordingly

From s_1 to s_T: A self-play *game* with a final reward $r_T = -1$ (loss) or $+1$ (win)

All $(s_t, \boldsymbol{\pi}_t, z_t)$ will be stored, where $z_t = \pm r_T$ with sign determined by the current player at step t





Random initialized weights θ_0

At iteration $i \geq 1$, 25,000 *games* of self-play are generated based on f_{θ_*} (➔ current best player α_{θ_*})

θ_i are trained using data sampled uniformly among all $(s, \boldsymbol{\pi}, z)$'s of the last 500,000 *games* of self-play, by minimizing loss: $l = (z - v)^2 - \boldsymbol{\pi}^T \log \boldsymbol{p} + c\|\theta\|^2$

Slide credit: Chia-Ching Lin

Network Training Details

- Some details about network training
  - Each neural network f_{θ_i} is optimized on the Google Cloud using TensorFlow, with 64 GPU workers (batch-size 32 per worker) and 19 CPU parameter servers
  - Total batch-size is 2,048, sampled uniformly at random from all positions of the most recent 500,000 games of self-play
  - Produces a new checkpoint every 1,000 training steps
  - Each checkpoint f_{θ_i} is further evaluated against the current best f_{θ_*} by 400 games using MCTS to decide actions
    - If the new player α_{θ_i} (guided by f_{θ_i}) wins by a margin of > 55% (to avoid selecting on noise alone) then it becomes the best player α_{θ_*} (guided by f_{θ_*}), and is subsequently used for self-play generation, and also becomes the baseline for subsequent comparisons

Slide credit: Chia-Ching Lin

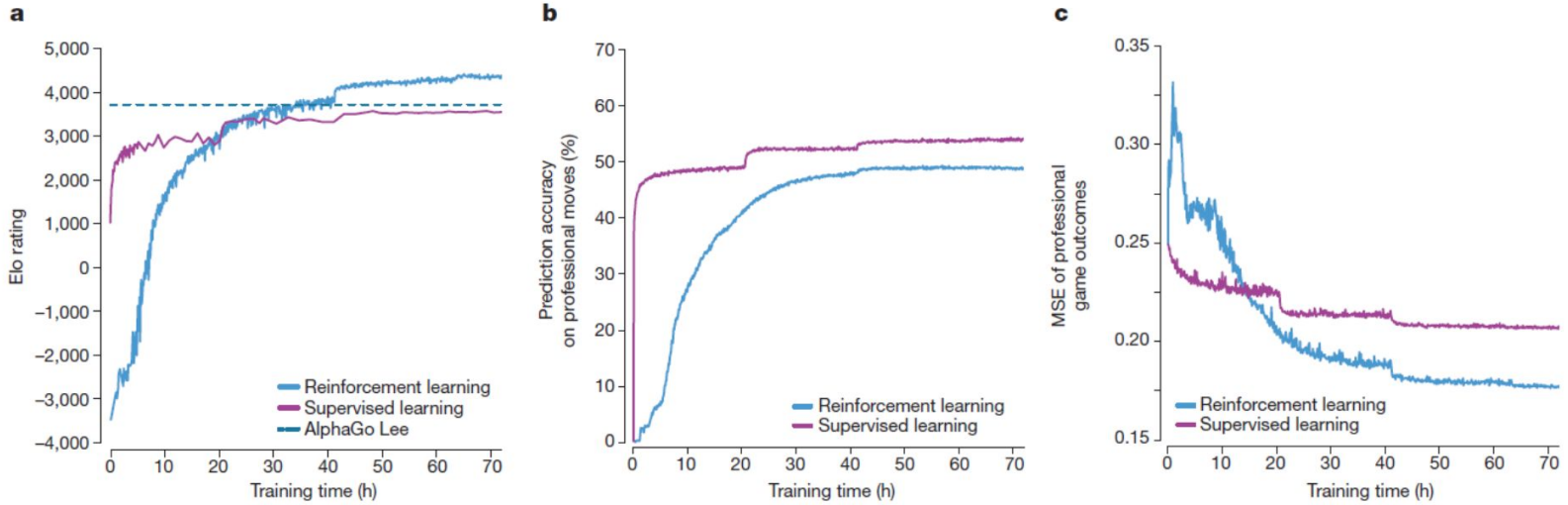
Network Architecture

- Some details about network training (cont'd)
  - The input to the neural network is a $19 \times 19 \times 17$ image stack comprising 17 binary feature planes $s_t = [X_t, Y_t, X_{t-1}, Y_{t-1}, \dots, X_{t-7}, Y_{t-7}, C]$
    - X's: indicating the presence of the current player's stones (including histories)
    - Y's: indicating the presence of the opponent's stones (including histories)
    - C: representing the color to play
  - Network architecture
    - One convolutional block (256 filters of size 3×3, BN, RELU) followed by either 19 or 39 residual blocks (256 filters of size 3×3, BN, RELU, 256 filters of size 3×3, BN, skip connection, RELU)
    - Two separate "heads" for computing the policy and value
      - **Policy head**: convolutional block (2 filters of size 1×1, BN, RELU) + fully connected linear layer that outputs a vector of size 362
      - **Value head**: convolutional block (1 filter of size 1×1, BN, RELU) + fully connected linear layer (size 256, RELU) + fully connected linear layer (size 1, tanh) to output a scalar in the range $[-1, 1]$

Slide credit: Chia-Ching Lin

Empirical Analysis

- AlphaGo Zero (19 residual blocks) outperformed AlphaGo Lee after 36 h (see **a**)
- Supervised learning (from human data using the same architecture) was better at predicting human professional moves (see **b**), but the self-trained player still performed much better overall, defeating the human-trained player within the first 24 h (see **a**)
  - This suggests that AlphaGo Zero may be learning a strategy that is qualitatively different to human play

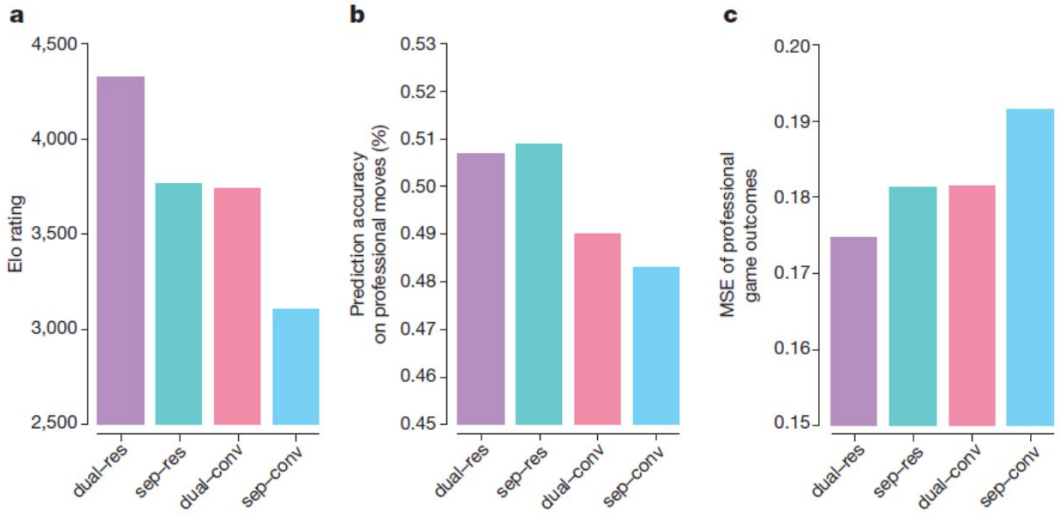


Empirical Analysis

- Comparison of network architectures for policy network and value network
  - **dual-res** (AlphaGo Zero): single network with residual blocks
  - **sep-res**: 2 separate network with residual blocks
  - **dual-conv**: single CNN
  - **sep-conv** (AlphaGo): 2 separate CNN

~600 Elo

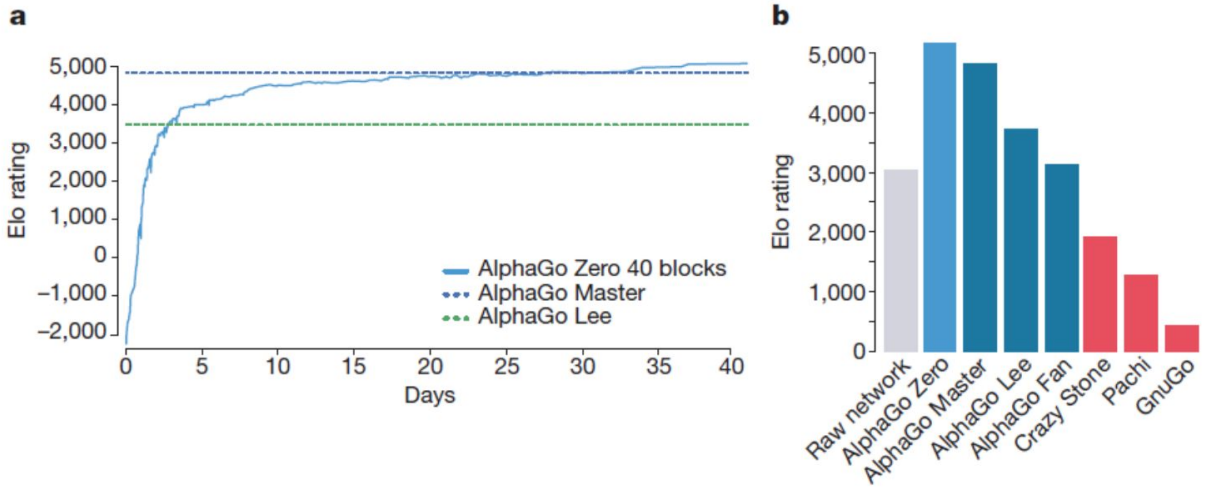
~600 Elo



Slide credit: Chia-Ching Lin

Final Performance

- AlphaGo Zero with deeper network (19 → 39 residual blocks) and longer training time (3 → 40 days)



- Raw network: directly selects the move a with maximum probability p_a output by the network, without using MCTS (i.e., π_a) to sample next move

Slide credit: Chia-Ching Lin

AlphaGo vs. AlphaGo Zero

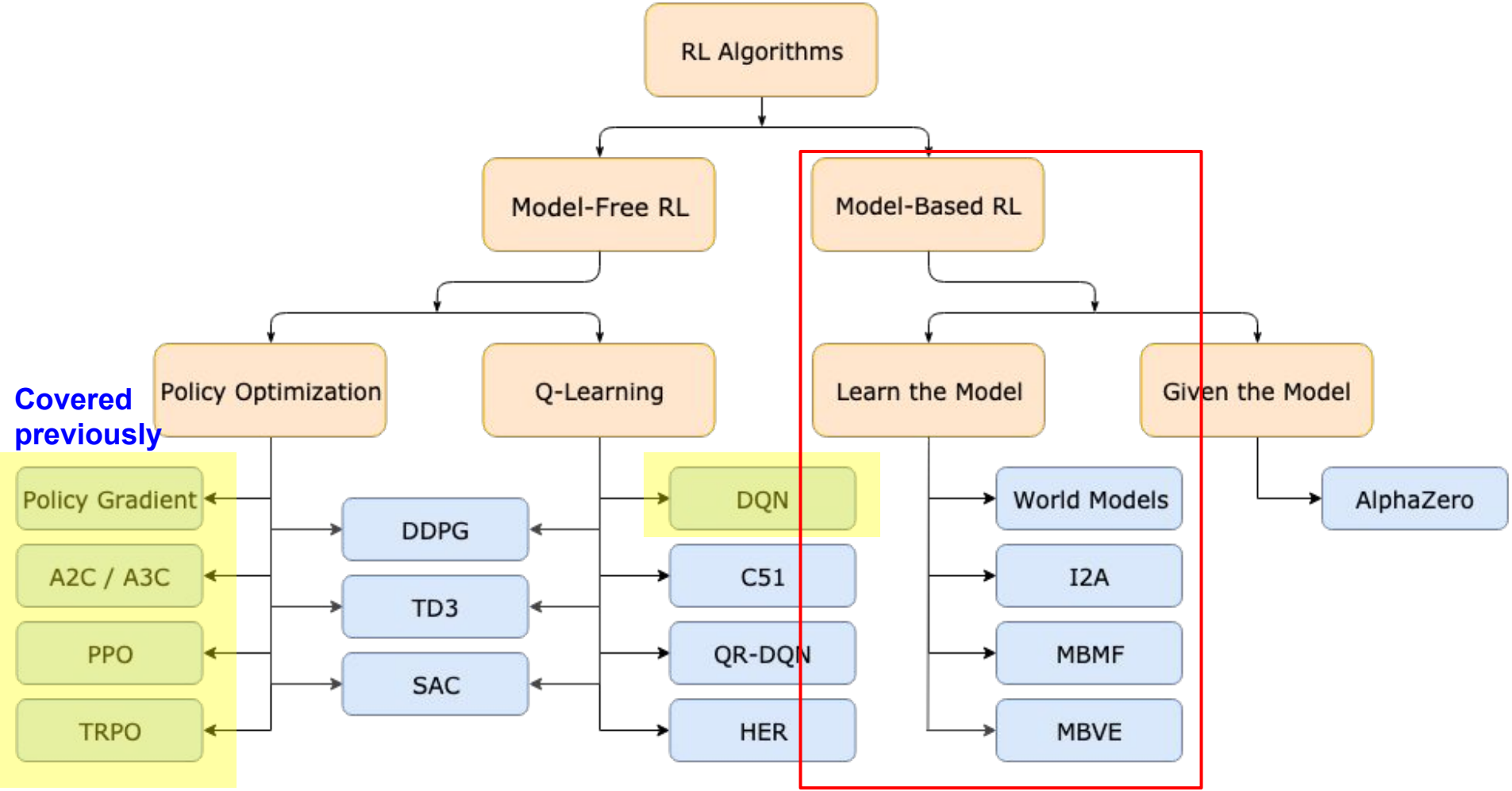
- Main modifications comparing to old versions of AlphaGo
  - Self-play reinforcement learning without any human data
  - Simpler board representations using only the black and white stones
  - Single neural network, rather than separate policy and value networks
    - Also, the residual blocks matter (mentioned in [2])
  - Simpler tree search without rollouts
- AlphaGo Zero and AlphaZero compensate for the lower number of evaluations by using its deep neural network to focus much more selectively on the most promising variations – arguably a more "human-like" approach to search

Slide credit: Chia-Ching Lin

Q. What if we don’t have a true model?

Outline

- Model-Free RL (for continuous control)
  - CEM, CMA-ES, NAF, DDPG
- **Model-Based RL**
  - Using the true model
    - AlphaGo
    - AlphaGo Zero
  - **Learning the model**
    - **Dyna-Q**
    - MPC
    - MBMF


RL Algorithms
Model-Free RL
Model-Based RL
Policy Optimization
Q-Learning
Learn the Model
Given the Model
Covered previously
Policy Gradient
A2C / A3C
PPO
TRPO
DDPG
TD3
SAC
DQN
C51
QR-DQN
HER
World Models
I2A
MBMF
MBVE
AlphaZero

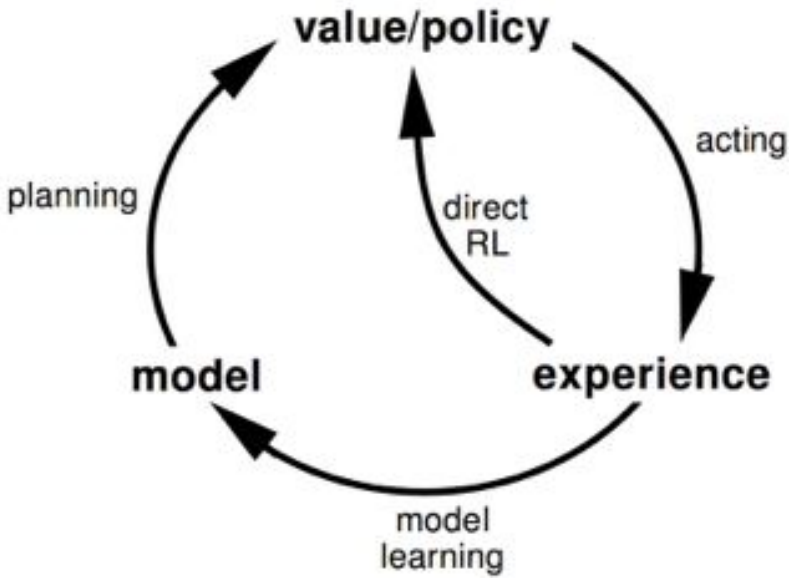
Why model-based reinforcement learning?

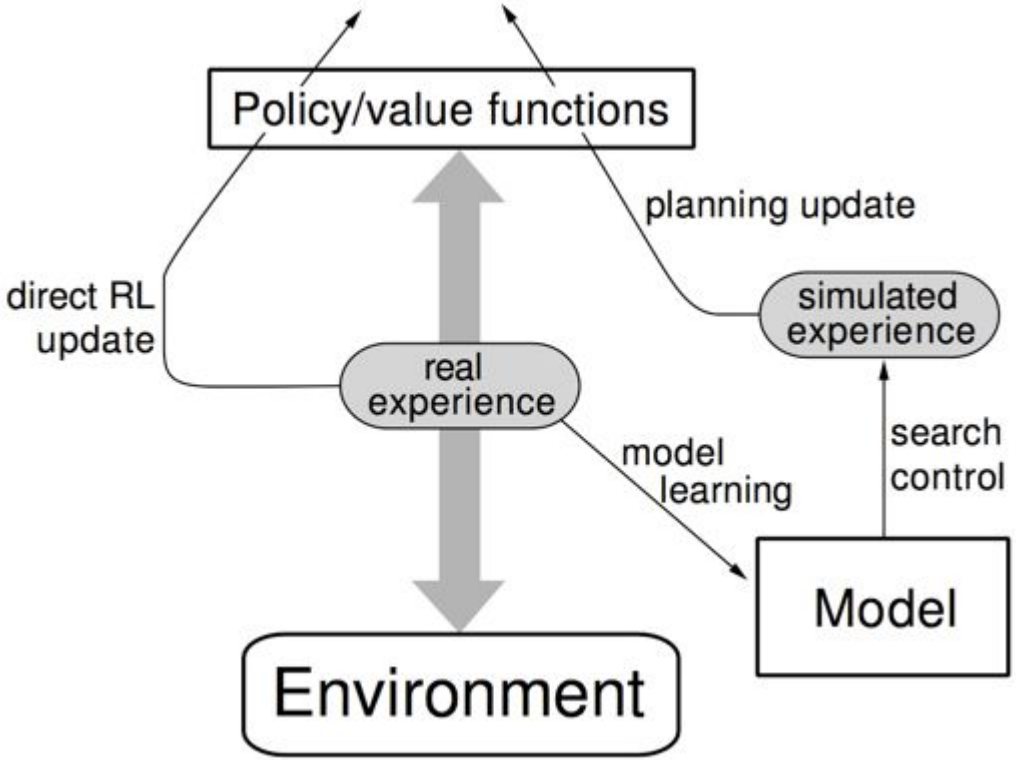
- a model enables you to plan
- sample efficiency
- transferability & generality

A model can be reused for achieving different tasks.

Slide credit: Chelsea Finn

The Dyna Architecture (pre-deep learning era)





Sutton, R. S. (1991). Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, *2*(4), 160-163.

The Dyna-Q Algorithm (pre-deep learning era)

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

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

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

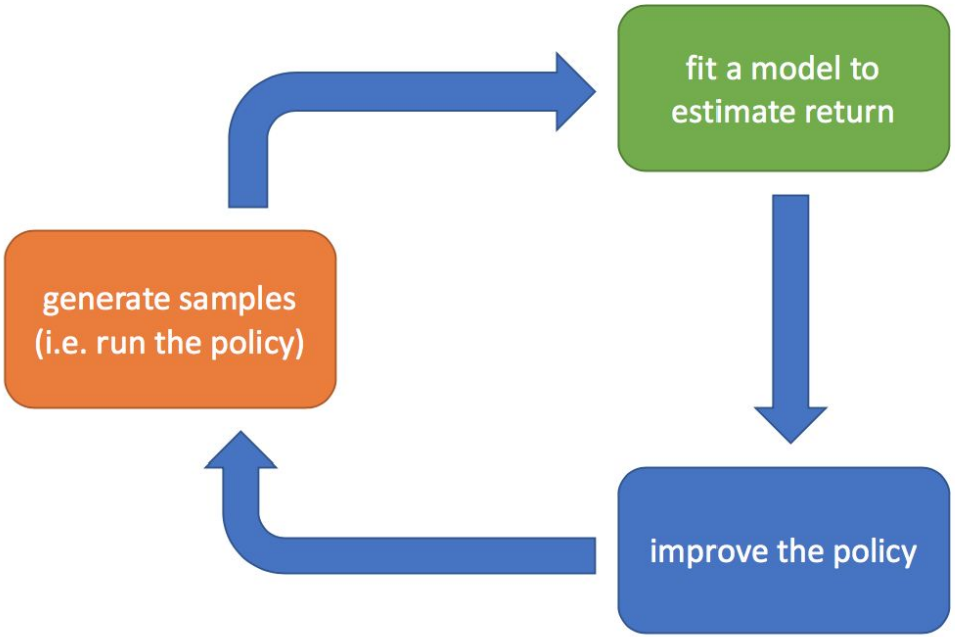
Direct RL

Model Learning

Update Q function based on simulated samples from model

Sutton, R. S. (1991). Dyna, an integrated architecture for learning, planning, and reacting. *ACM SIGART Bulletin*, *2*(4), 160-163.

The anatomy of Reinforcement Learning Problem

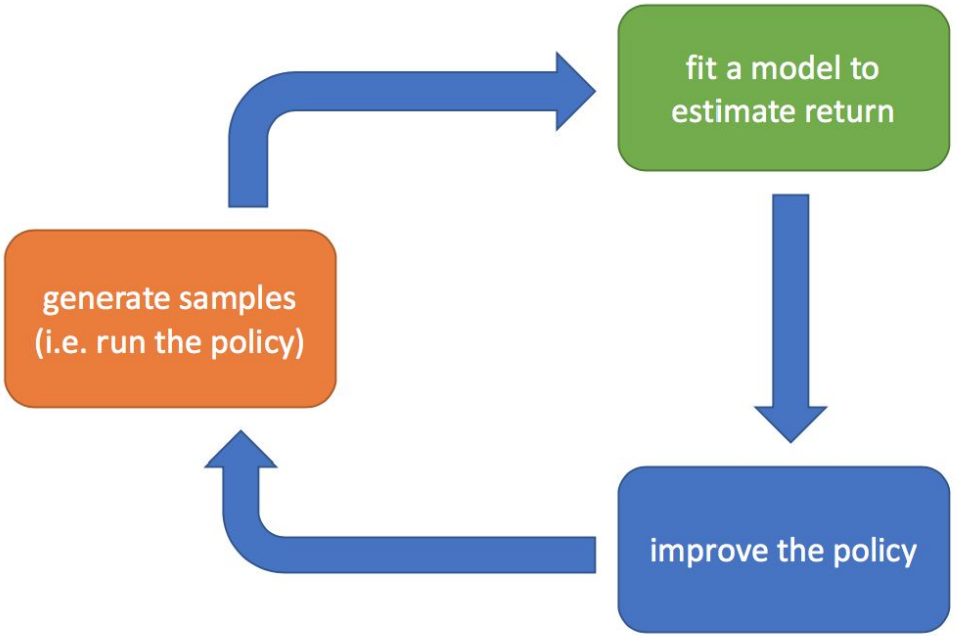


compute $\hat{Q} = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$ (MC policy gradient)
fit $Q_\phi(\mathbf{s}, \mathbf{a})$ (actor-critic, Q-learning)
estimate $p(\mathbf{s}'|\mathbf{s}, \mathbf{a})$ (model-based)

$\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$ (policy gradient)
$\pi(\mathbf{s}) = \arg\max Q_\phi(\mathbf{s}, \mathbf{a})$ (Q-learning)
optimize $\pi_\theta(\mathbf{a}|\mathbf{s})$ (model-based)

Slide credit: Sergey Levine

Model-Based Reinforcement Learning



estimate $p(\mathbf{s}'|\mathbf{s},\mathbf{a})$ (model-based)

supervised learning

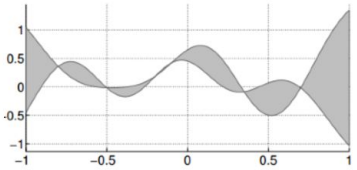
$$\min_{\phi} \sum_{i} ||f_{\phi}(\mathbf{s}_i,\mathbf{a}_i) - \mathbf{s}'_i||^2$$

optimize $\pi_{\theta}(\mathbf{a}|\mathbf{s})$ (model-based)

Slide credit: Sergey Levine

What kinds of models can we use?

Gaussian process



GP with input $(\mathbf{s}, \mathbf{a})$ and output $\mathbf{s}'$

Pro: very data-efficient

Con: not great with non-smooth dynamics

Con: very slow when dataset is big

neural network

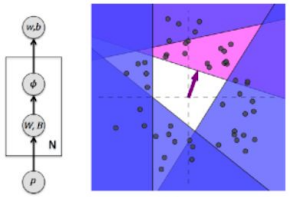


image: Punjani & Abbeel '14

Input is $(\mathbf{s}, \mathbf{a})$ and output is $\mathbf{s}'$

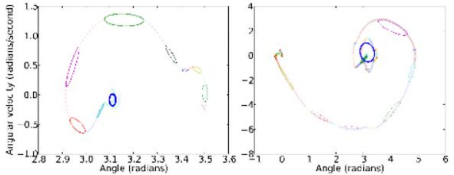
Euclidean training loss corresponds to Gaussian $p(\mathbf{s}' \mid \mathbf{s}, \mathbf{a})$

More complex losses, e.g. output parameters of Gaussian mixture

Pro: very expressive, can use lots of data

Con: not so great in low data regimes

other



GMM over $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ tuples

Train on $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$, condition to get $p(\mathbf{s}' \mid \mathbf{s}, \mathbf{a})$

For i^{th} mixture element, $p_i(\mathbf{s}, \mathbf{a})$ gives region where the mode $p_i(\mathbf{s}' \mid \mathbf{s}, \mathbf{a})$ holds

other classes: domain-specific models (e.g. physics parameters)

video prediction?
more on this later

Slide adapted from S. Levine

Model-Based Reinforcement Learning (ver 0.5)

If we knew $f(\mathbf{s}_t, \mathbf{a}_t) = \mathbf{s}_{t+1}$, (or $p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ in the stochastic case)

let's learn $f(\mathbf{s}_t, \mathbf{a}_t)$ from data, and *then* plan through it!

model-based reinforcement learning version 0.5:

1. run base policy $\pi_0(\mathbf{a}_t|\mathbf{s}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{s}, \mathbf{a}, \mathbf{s}')_i\}$
2. learn dynamics model $f(\mathbf{s}, \mathbf{a})$ to minimize $\sum_i \|f(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{s}'_i\|^2$
3. plan through $f(\mathbf{s}, \mathbf{a})$ to choose actions

Slide credit: Sergey Levine

Does it work?

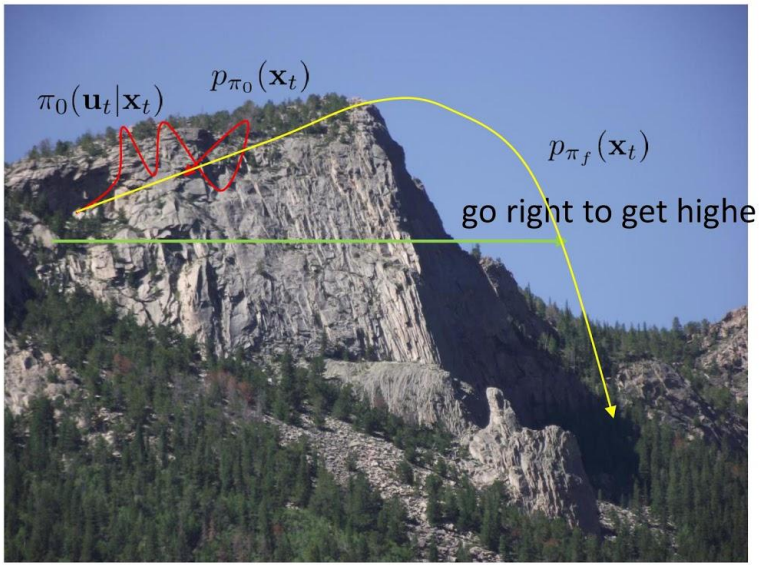
Yes!

- Essentially how system identification works in classical robotics
- Some care should be taken to design a good base policy
- Particularly effective if we can hand-engineer a dynamics representation using our knowledge of physics, and fit just a few parameters

Slide credit: Sergey Levine

Does it work?

No!



1. run base policy $\pi_0(\mathbf{a}_t|\mathbf{s}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{s}, \mathbf{a}, \mathbf{s}')_i\}$
2. learn dynamics model $f(\mathbf{s}, \mathbf{a})$ to minimize $\sum_i \|f(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{s}'_i\|^2$
3. plan through $f(\mathbf{s}, \mathbf{a})$ to choose actions

$$p_{\pi_f}(\mathbf{s}_t) \neq p_{\pi_0}(\mathbf{s}_t)$$

- Distribution mismatch problem becomes exacerbated as we use more expressive model classes

Slide credit: Sergey Levine

Outline

- Model-Free RL (for continuous control)
  - CEM, CMA-ES, NAF, DDPG
- **Model-Based RL**
  - Using the true model
    - AlphaGo
    - AlphaGo Zero
  - **Learning the model**
    - Dyna-Q
    - **MPC**
    - MBMF

Can we do better?

can we make $p_{\pi_0}(\mathbf{s}_t) = p_{\pi_f}(\mathbf{s}_t)$?

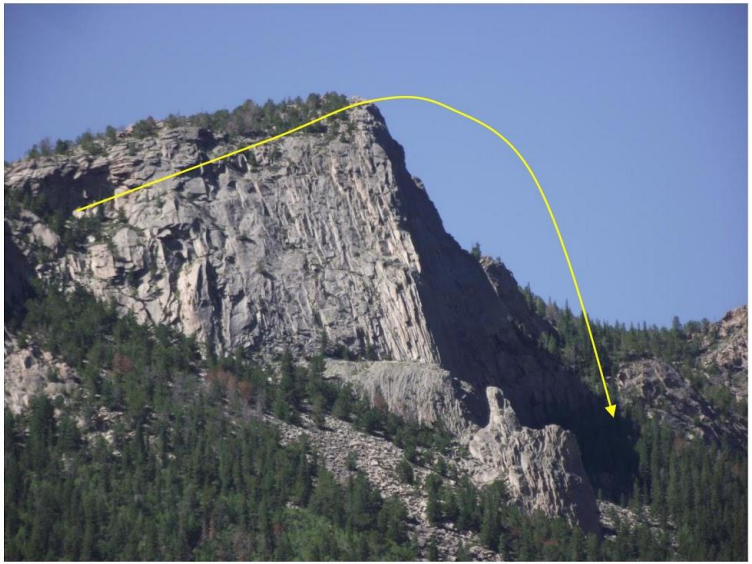
where have we seen that before? need to collect data from $p_{\pi_f}(\mathbf{s}_t)$

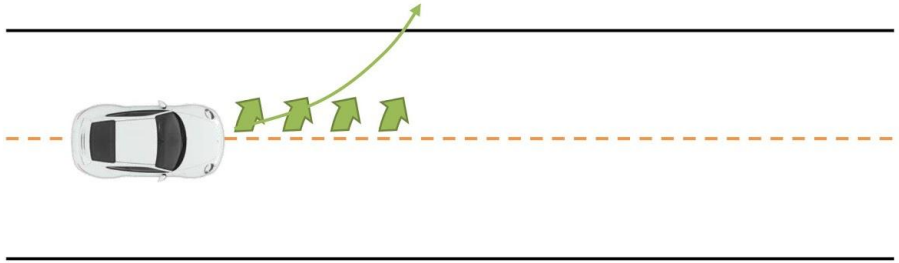
model-based reinforcement learning version 1.0:

1. run base policy $\pi_0(\mathbf{a}_t|\mathbf{s}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{s}, \mathbf{a}, \mathbf{s}')_i\}$
2. learn dynamics model $f(\mathbf{s}, \mathbf{a})$ to minimize $\sum_i \|f(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{s}'_i\|^2$
3. plan through $f(\mathbf{s}, \mathbf{a})$ to choose actions
4. execute those actions and add the resulting data $\{(\mathbf{x}, \mathbf{u}, \mathbf{x}')_j\}$ to $\mathcal{D}$

Slide credit: Sergey Levine

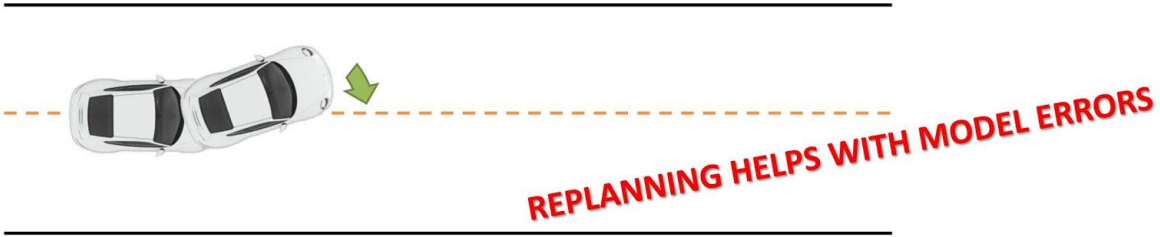
What if we make a mistake?





Slide credit: Sergey Levine

Can we do better?



model-based reinforcement learning version 1.5:

1. run base policy $\pi_0(\mathbf{a}_t|\mathbf{s}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{s}, \mathbf{a}, \mathbf{s}')_i\}$
2. learn dynamics model $f(\mathbf{s}, \mathbf{a})$ to minimize $\sum_i \|f(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{s}'_i\|^2$
3. plan through $f(\mathbf{s}, \mathbf{a})$ to choose actions
4. execute the first planned action, observe resulting state $\mathbf{s}'$ (MPC)
5. append $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ to dataset $\mathcal{D}$

every N steps

Slide credit: Sergey Levine

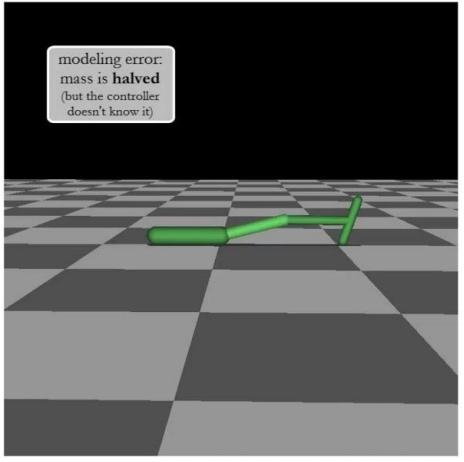
How to replan?

model-based reinforcement learning version 1.5:

1. run base policy $\pi_0(\mathbf{a}_t|\mathbf{s}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{s}, \mathbf{a}, \mathbf{s}')_i\}$
2. learn dynamics model $f(\mathbf{s}, \mathbf{a})$ to minimize $\sum_i \|f(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{s}'_i\|^2$
3. plan through $f(\mathbf{s}, \mathbf{a})$ to choose actions
4. execute the first planned action, observe resulting state $\mathbf{s}'$ (MPC)
5. append $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ to dataset $\mathcal{D}$

every N steps

- The more you replan, the less perfect each individual plan needs to be
- Can use shorter horizons
- Even random sampling can often work well here!

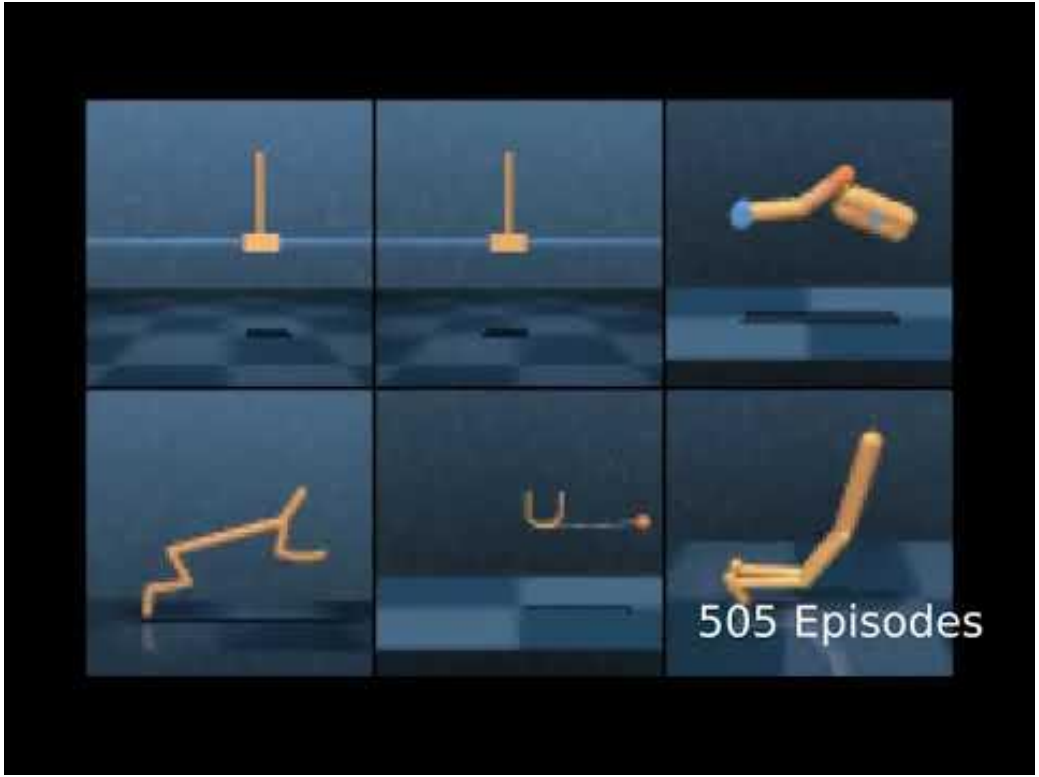


Slide credit: Sergey Levine

MPC also works for pixel observations!

Learning dynamics/reward models with latent variables from pixel-level data

Use the learned model for planning (MPC)



Hafner, D., Lillicrap, T., Fischer, I., Villegas, R., Ha, D., Lee, H., & Davidson, J. (2018). Learning latent dynamics for planning from pixels. *arXiv preprint arXiv:1811.04551*.
https://ai.googleblog.com/2019/02/introducing-planet-deep-planning.html

Outline

- Model-Free RL (for continuous control)
  - CEM, CMA-ES, NAF, DDPG
- **Model-Based RL**
  - Using the true model
    - AlphaGo
    - AlphaGo Zero
  - **Learning the model**
    - Dyna-Q
    - MPC
    - **MBMF**

Combining model-based and model-free RL

Neural Network Dynamics for Model-Based Deep Reinforcement Learning with Model-Free Fine-Tuning

Anusha Nagabandi, Gregory Kahn, Ronald S. Fearing, Sergey Levine
University of California, Berkeley

model-based reinforcement learning version 1.5:

1. run base policy $\pi_0(\mathbf{a}_t|\mathbf{s}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{s}, \mathbf{a}, \mathbf{s}')_i\}$
2. learn dynamics model $f(\mathbf{s}, \mathbf{a})$ to minimize $\sum_i \|f(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{s}'_i\|^2$
3. plan through $f(\mathbf{s}, \mathbf{a})$ to choose actions (random sampling)
4. execute the first planned action, observe resulting state $\mathbf{s}'$ (MPC)
5. append $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ to dataset $\mathcal{D}$

Slide credit: Sergey Levine

Model-Based Part

Algorithm 1 Model-based Reinforcement Learning

1: gather dataset $\mathcal{D}_{\text{RAND}}$ of random trajectories
2: initialize empty dataset $\mathcal{D}_{\text{RL}}$, and randomly initialize $\hat{f}_\theta$
3: **for** iter=1 **to** max_iter **do**
4: train $\hat{f}_\theta(\mathbf{s}, \mathbf{a})$ by performing gradient descent on Eqn. 2, using $\mathcal{D}_{\text{RAND}}$ and $\mathcal{D}_{\text{RL}}$
5: **for** $t = 1$ **to** T **do**
6: get agent's current state $\mathbf{s}_t$
7: use $\hat{f}_\theta$ to estimate optimal action sequence $\mathbf{A}_t^{(H)}$ (Eqn. 4)
8: execute first action $\mathbf{a}_t$ from selected action sequence $\mathbf{A}_t^{(H)}$
9: add $(\mathbf{s}_t, \mathbf{a}_t)$ to $\mathcal{D}_{\text{RL}}$
10: **end for**
11: **end for**

$$\mathcal{E}(\theta) = \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{s}_t, \mathbf{a}_t, \mathbf{s}_{t+1}) \in \mathcal{D}} \frac{1}{2} \| (\mathbf{s}_{t+1} - \mathbf{s}_t) - \hat{f}_\theta(\mathbf{s}_t, \mathbf{a}_t) \|^2$$

$$\mathbf{A}_t^{(H)} = \arg\max_{\mathbf{A}_t^{(H)}} \sum_{t'=t}^{t+H-1} r(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}) \quad :$$

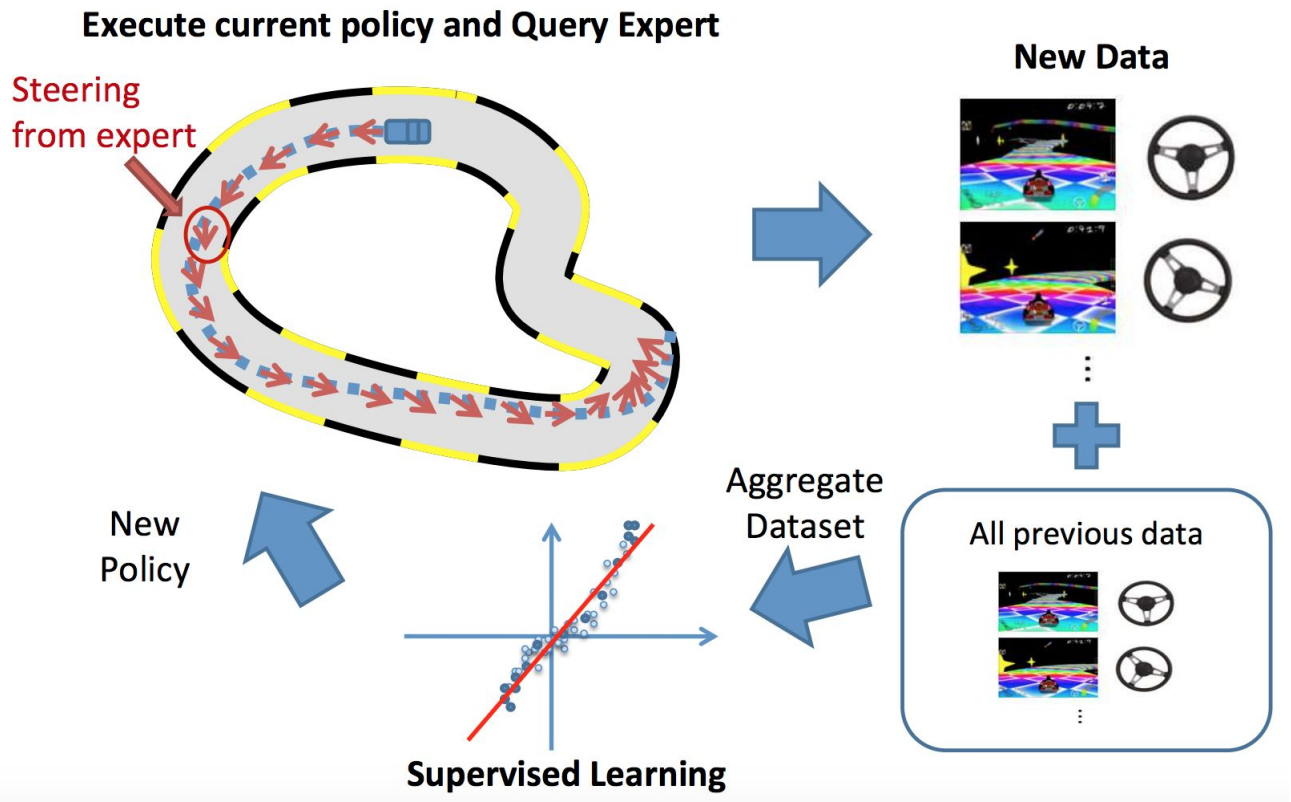
$$\hat{\mathbf{s}}_t = \mathbf{s}_t, \hat{\mathbf{s}}_{t'+1} = \hat{\mathbf{s}}_{t'} + \hat{f}_\theta(\hat{\mathbf{s}}_{t'}, \mathbf{a}_{t'}).$$

Nagabandi, A., Kahn, G., Fearing, R. S., & Levine, S. (2018, May). Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning

MB-MF: Model-Based Initialization of Model-free Reinforcement Learning Algorithm

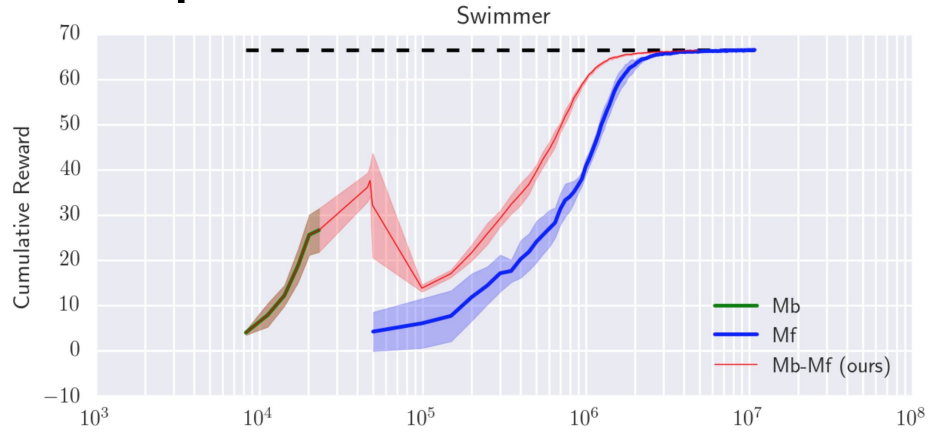
- Learn dynamic function using the model-based reinforcement learning algorithm
- Collect example “expert” trajectories through MPC controller using the learnt dynamic function
- Train a neural network policy to match the “expert” trajectories with DAGGER, where MPC controller plays the role of expert
- The initialized policy is trained further with model-free method TRPO

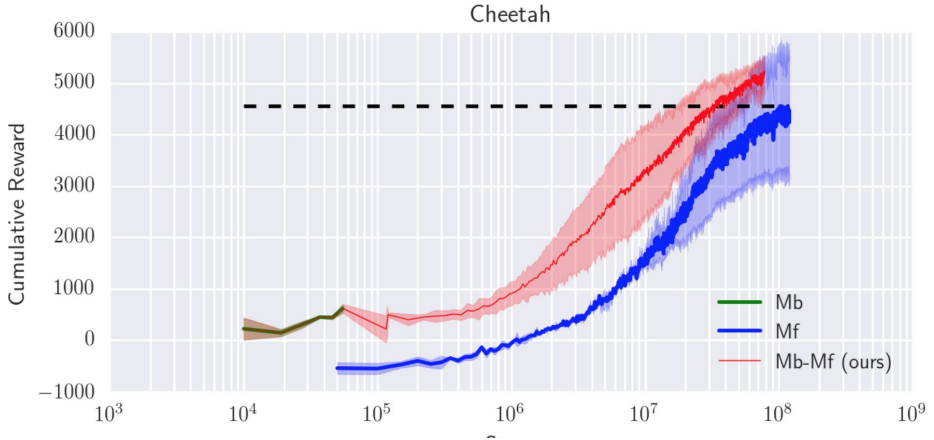
DAGGER

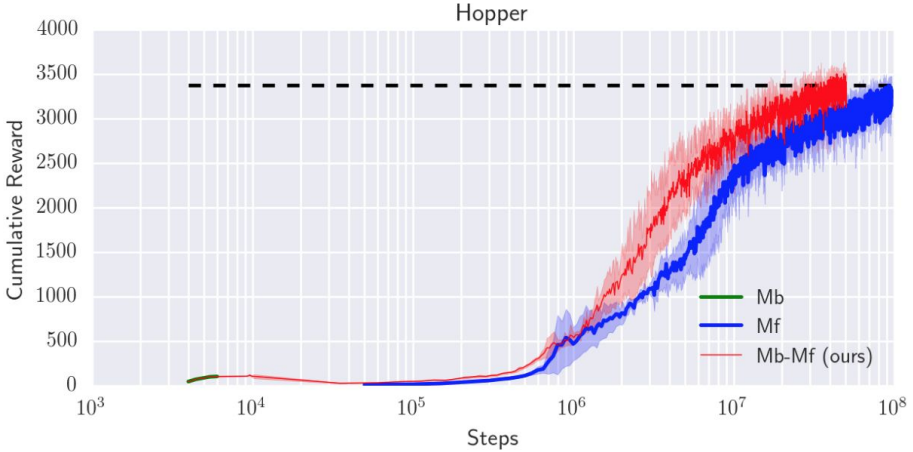


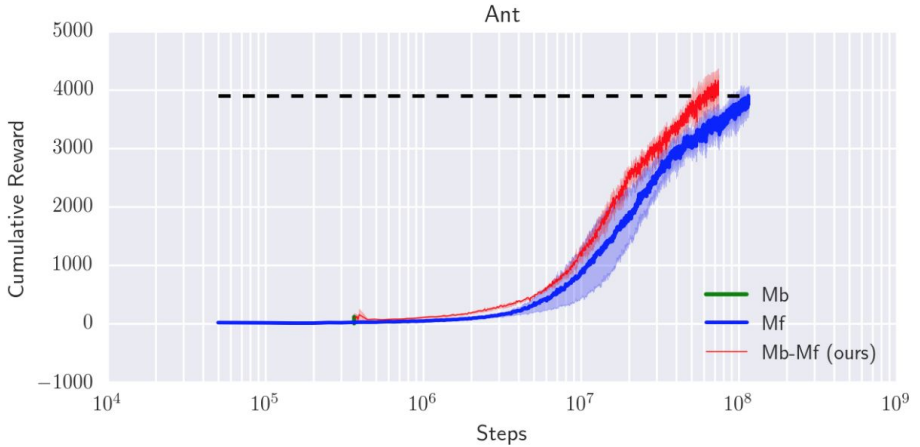
Ross, S., Gordon, G., & Bagnell, D. (2011, June). A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics* (pp. 627-635).

Experiment Results



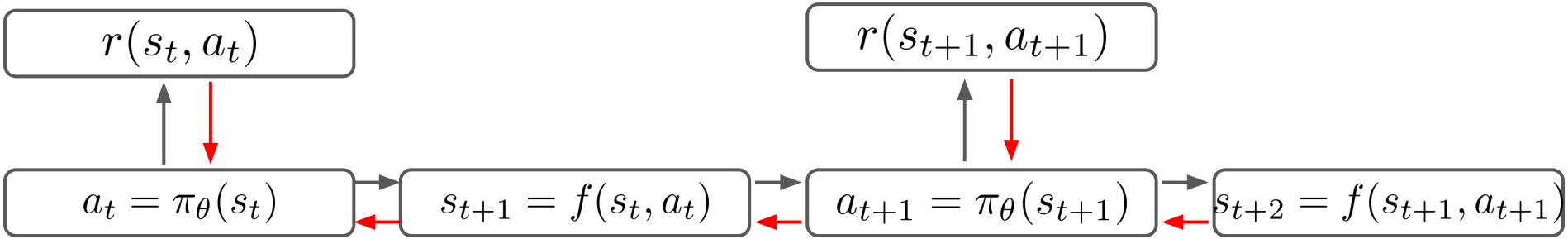






Nagabandi, A., Kahn, G., Fearing, R. S., & Levine, S. (2018, May). Neural network dynamics for model-based deep reinforcement learning with model-free fine-tuning.

Backpropagate directly into the policy? (MBRL v2.0)



model-based reinforcement learning version 2.0:

1. run base policy $\pi_0(\mathbf{a}_t|\mathbf{s}_t)$ (e.g., random policy) to collect $\mathcal{D} = \{(\mathbf{s}, \mathbf{a}, \mathbf{s}')_i\}$
2. learn dynamics model $f(\mathbf{s}, \mathbf{a})$ to minimize $\sum_i \|f(\mathbf{s}_i, \mathbf{a}_i) - \mathbf{s}'_i\|^2$
3. backpropagate through $f(\mathbf{s}, \mathbf{a})$ into the policy to optimize $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$
4. run $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$, appending the visited tuples $(\mathbf{s}, \mathbf{a}, \mathbf{s}')$ to $\mathcal{D}$

Slide credit: Sergey Levine

PILCO

Algorithm 1 PILCO

1: *Define* policy's functional form: $\pi : z_t \times \psi \rightarrow u_t$.
2: *Initialise* policy parameters ψ randomly.
3: **repeat**
4: *Execute* system, record data.
5: *Learn* dynamics model.
6: *Predict* system trajectories from $p(X_0)$ to $p(X_T)$.
7: *Evaluate* policy:
$$J(\psi) = \sum_{t=0}^{T} \gamma^t \mathbb{E}_X[\text{cost}(X_t)|\psi].$$
8: *Optimise* policy:
$$\psi \leftarrow \arg\min_{\psi} J(\psi).$$
9: **until** policy parameters ψ converge

Deisenroth, M., & Rasmussen, C. E. (2011). PILCO: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)* (pp. 465-472).

Summary

- Version 0.5: collect random samples, train dynamics, plan
  - Pro: simple, no iterative procedure
  - Con: distribution mismatch problem
- Version 1.0: iteratively collect data, replan, collect data
  - Pro: simple, solves distribution mismatch
  - Con: open loop plan might perform poorly, esp. in stochastic domains
- Version 1.5: iteratively collect data using MPC (replan at each step)
  - Pro: robust to small model errors
  - Con: computationally expensive, but have a planning algorithm available
- Version 2.0: backpropagate directly into policy
  - Pro: computationally cheap at runtime
  - Con: can be numerically unstable, especially in stochastic domains (more on this later)

Slide credit: Sergey Levine

Model-based vs Model-Free methods

Models:

+ Easy to collect data in a scalable way (self-supervised)
+ Possibility to transfer across tasks
+ Typically require a smaller quantity of supervised data
- Models don't optimize for task performance
- Sometimes harder to learn than a policy
- Often need assumptions to learn complex skills (continuity, resets)

Model-Free:

+ Makes little assumptions beyond a reward function
+ Effective for learning complex policies
- Require a lot of experience (slower)
- Not transferable across tasks

Ultimately we will want both!

Slide credit: Chelsea Finn

Further Readings in model-based RL

Use known model: Tassa et al. IROS '12, Tan et al. TOG '14, Mordatch et al. TOG '14

Guided policy search: Levine*, Finn* et al. JMLR '16, Mordatch et al. RSS '14, NIPS '15

Backprop through model: Deisenroth et al. ICML '11, Heess et al. NIPS '15, Mishra et al. ICML '17, Degrave et al. '17, Henaff et al. '17

Inverse models: Agrawal et al. NIPS '16

MBRL in latent space: Watter et al. NIPS '15, Finn et al. ICRA '16

MPC with deep models: Lenz et al. RSS '15, Finn & Levine ICRA '17, Hafner et al. 2018

Combining Model-Based & Model-Free:

- use roll-outs from model as experience: Sutton '90, Gu et al. ICML '16
- use model as baseline: Chebotar et al. ICML '17
- use model for exploration: Stadie et al. arXiv '15, Oh et al. NIPS '16
- model-free policy with planning capabilities: Tamar et al. NIPS '16, Pascanu et al. '17
- model-based look-ahead: Guo et al. NIPS '14, Silver et al. Nature '16

Slide credit: Chelsea Finn

Quiz (True / False)

- Both AlphaGo and AlphaGo Zero use self-play reinforcement learning to train the policy network.
- During MCTS in AlphaGo Zero, the policy network is used to take rollout and get value estimate of state.
- The probability π of playing each move, which is output from MCTS, usually select much stronger moves than the raw move probabilities p of the neural network $f_\theta(s)$. MCTS may therefore be viewed as a powerful policy improvement operator.
- Compared with Model-Free RL methods, Model-Based RL methods usually have better sample efficiency.

https://docs.google.com/forms/d/e/1FAIpQLScpGi4y5WYZmoSIC8IATvYecmtH2SW1XIraFK4dsp24ifxQHA/viewform