Model-Based RL

The Dyna-Q Algorithm

Initialize $Q(s, a)$ and $Model(s, a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Do forever:
(a) $S \leftarrow$ current (nonterminal) state
(b) $A \leftarrow \varepsilon$-greedy(S, Q)
(c) Execute action A; observe resultant reward, R, and state, S'
(d) $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$ direct RL
(e) $Model(S, A) \leftarrow R, S'$ (assuming deterministic environment) model learning
(f) Repeat n times: planning
$S \leftarrow$ random previously observed state
$A \leftarrow$ random action previously taken in S
$R, S' \leftarrow Model(S, A)$
$Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

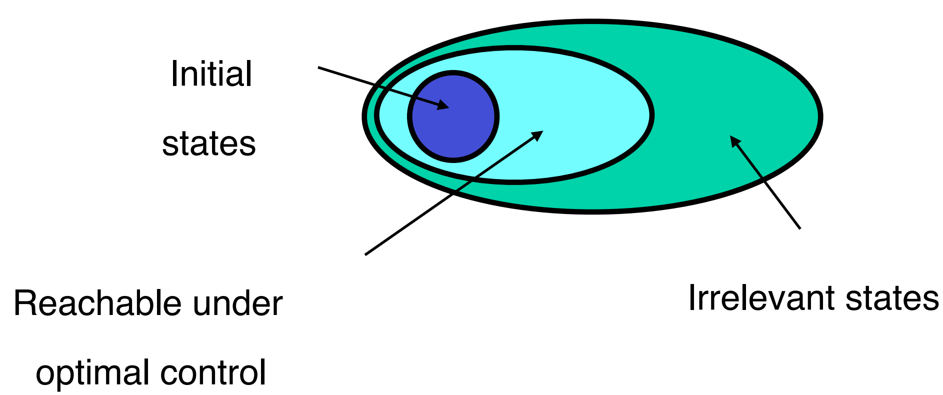
Prioritized Dyna-Q

Prioritized sweeping for a deterministic environment

Initialize $Q(s,a)$, $Model(s,a)$, for all s, a, and $PQueue$ to empty
Loop forever:
  (a) $S \leftarrow$ current (nonterminal) state
  (b) $A \leftarrow policy(S, Q)$
  (c) Take action A; observe resultant reward, R, and state, S'
  (d) $Model(S, A) \leftarrow R, S'$
  (e) $P \leftarrow |R + \gamma \max_a Q(S', a) - Q(S, A)|$.
  (f) if $P > \theta$, then insert S, A into $PQueue$ with priority P
  (g) Loop repeat n times, while $PQueue$ is not empty:
    $S, A \leftarrow first(PQueue)$
    $R, S' \leftarrow Model(S, A)$
    $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma \max_a Q(S', a) - Q(S, A)\big]$
    Loop for all $\bar{S}, \bar{A}$ predicted to lead to S:
      $\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
      $P \leftarrow |\bar{R} + \gamma \max_a Q(S, a) - Q(\bar{S}, \bar{A})|$.
      if $P > \theta$ then insert $\bar{S}, \bar{A}$ into $PQueue$ with priority P

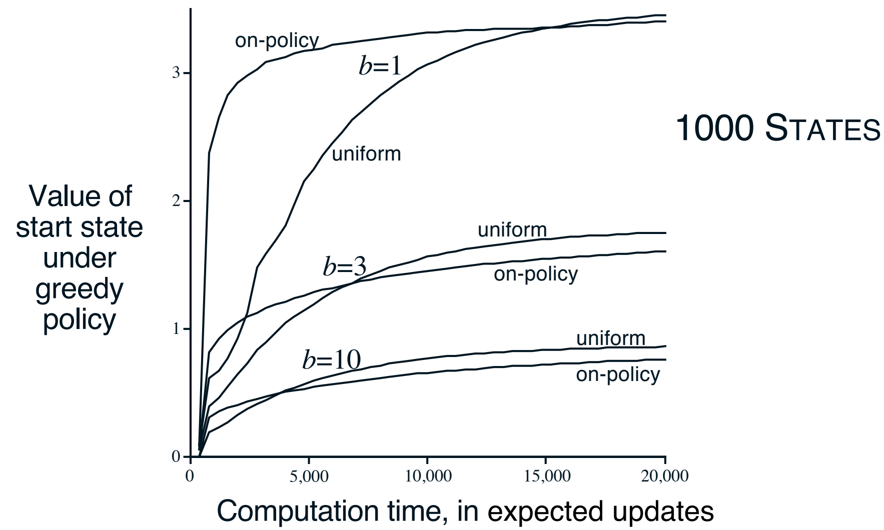
Trajectory Sampling

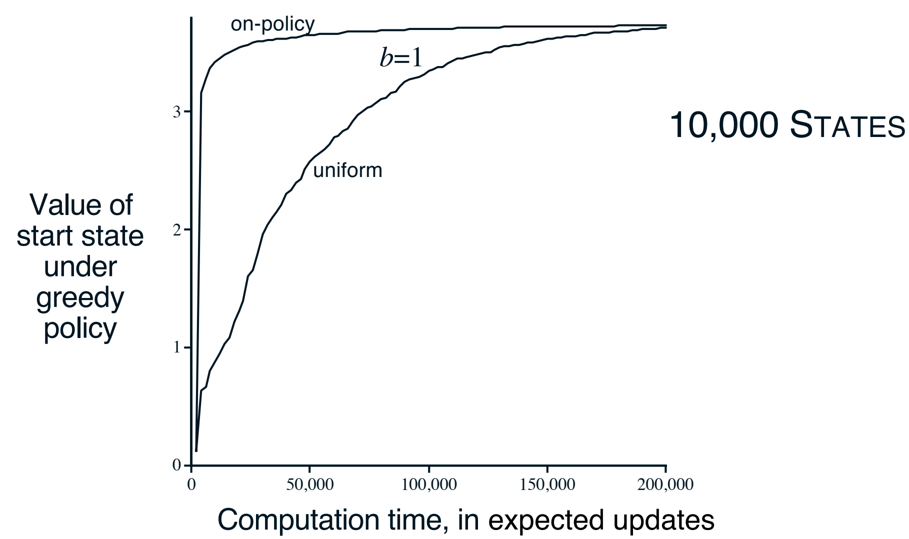
- Trajectory sampling: perform updates along simulated trajectories
- This samples from the on-policy distribution
- Advantages when function approximation is used (Part II)
- Focusing of computation:
  can cause vast uninteresting parts of the state space to be ignored:



Trajectory Sampling Experiment

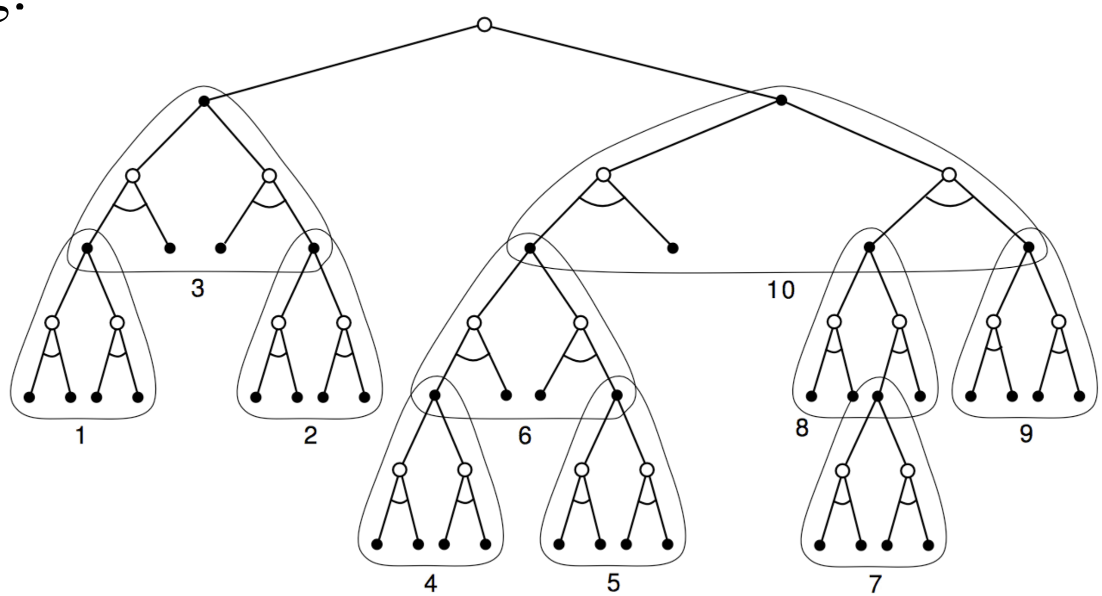
- one-step full tabular updates
- uniform: cycled through all state-action pairs
- on-policy: backed up along simulated trajectories
- 200 randomly generated undiscounted episodic tasks
- 2 actions for each state, each with b equally likely next states
- 0.1 prob of transition to terminal state
- expected reward on each transition selected from mean 0 variance 1 Gaussian



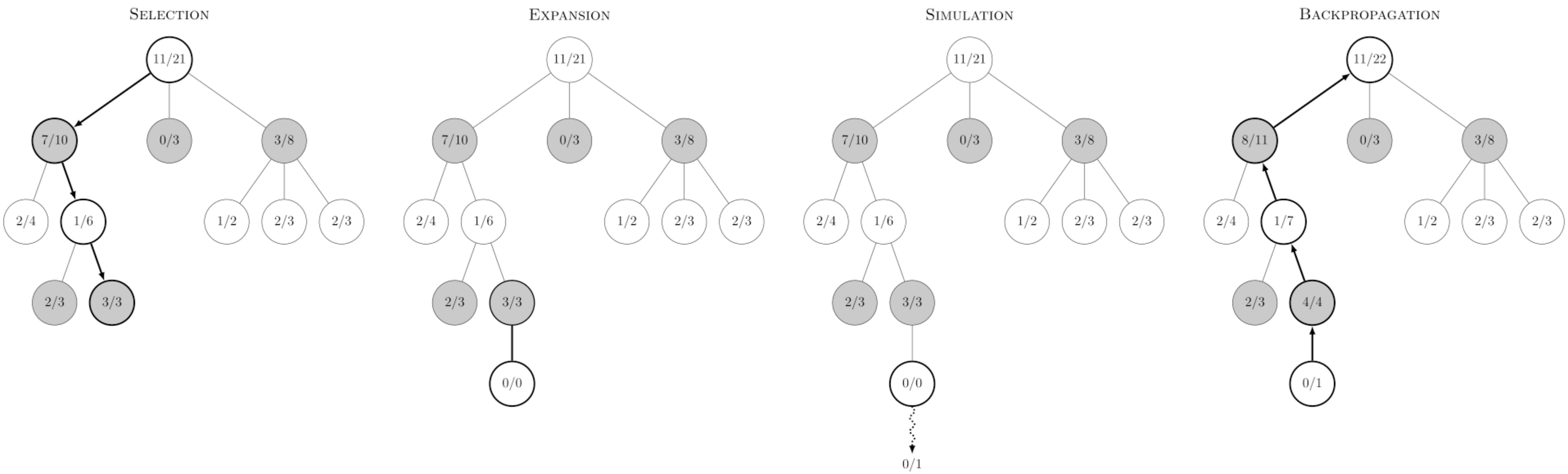


Heuristic Search

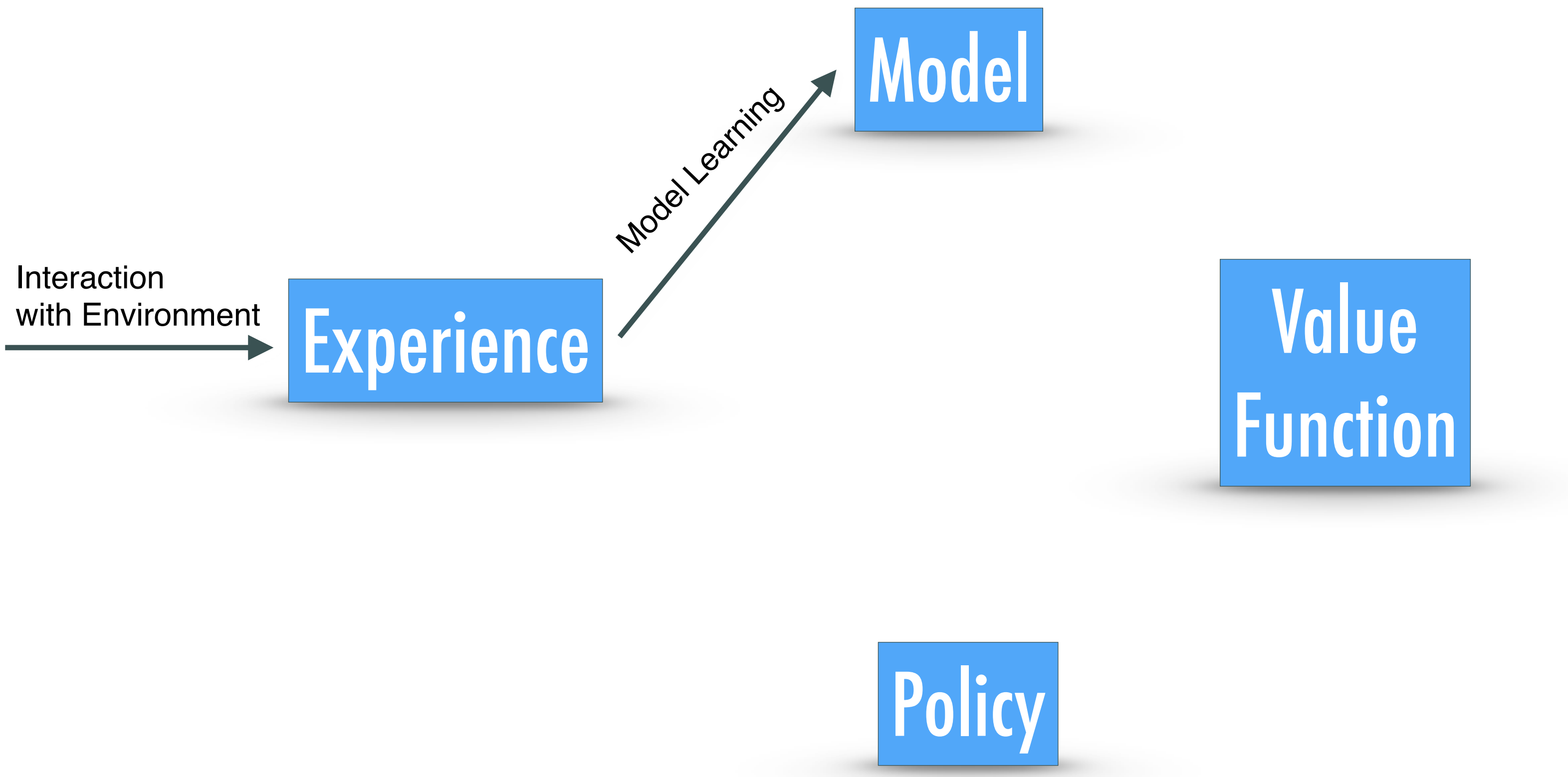
- Used for action selection, not for changing a value function (=heuristic evaluation function)
- Backed-up values are computed, but typically discarded
- Extension of the idea of a greedy policy — only deeper
- Also suggests ways to select states to backup: smart focusing:



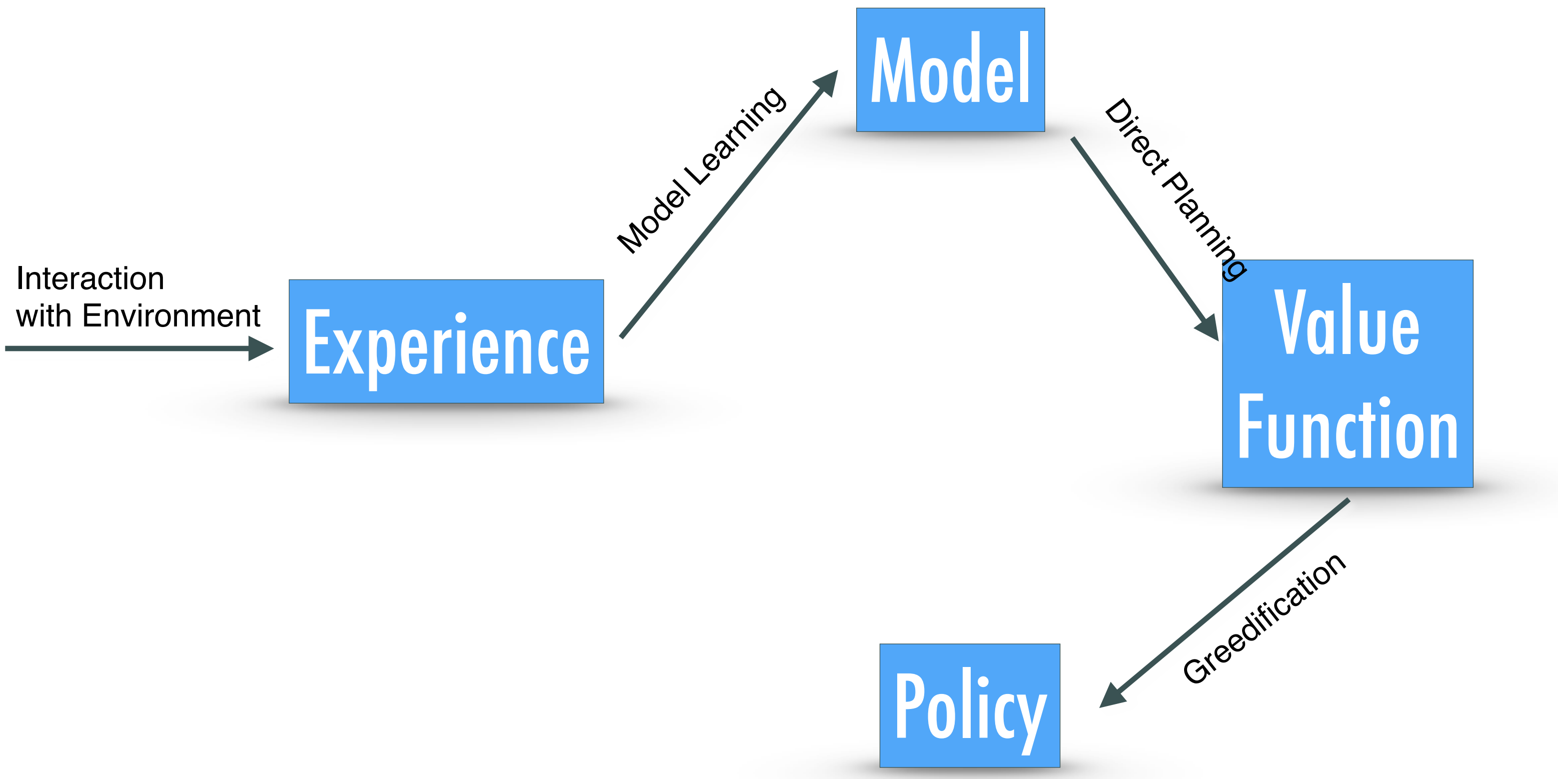
Monte-Carlo Tree Search (e.g. AlphaZero):



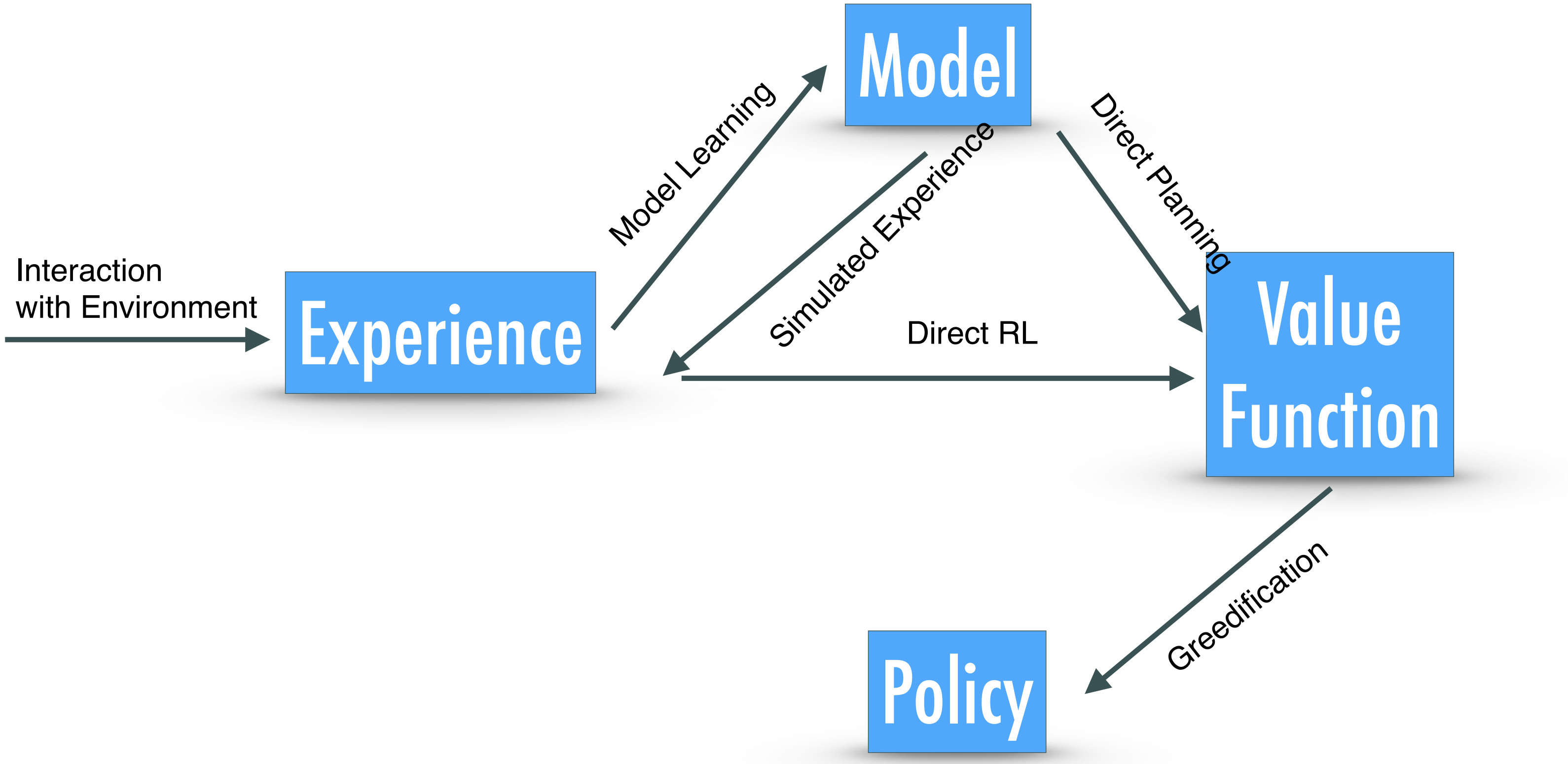
Conclusion


Model
Model Learning
Interaction
with Environment
Experience
Value
Function
Policy

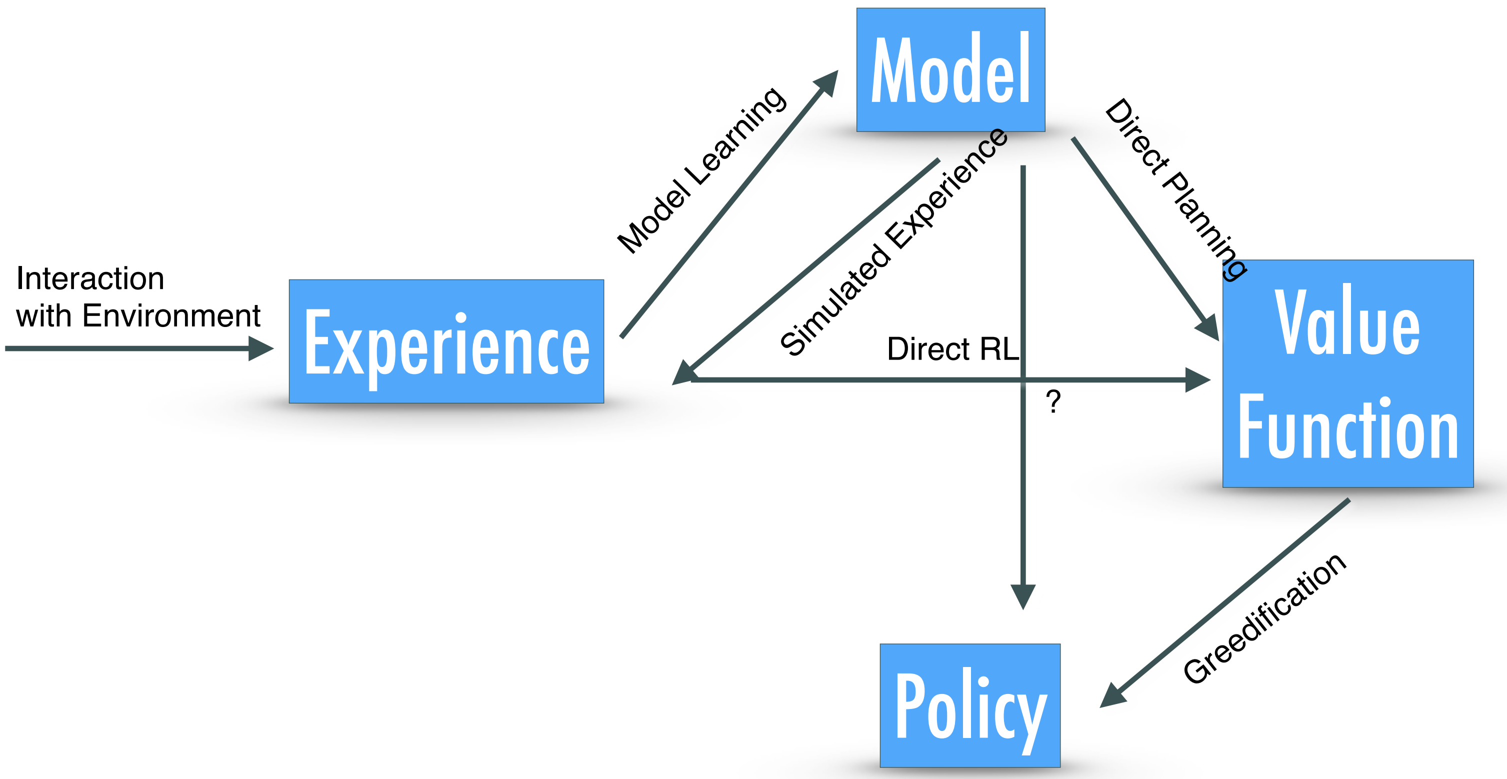
Conclusion


Interaction
with Environment
Experience
Model Learning
Model
Direct Planning
Value
Function
Greedification
Policy

Conclusion


Interaction
with Environment
Experience
Model Learning
Model
Simulated Experience
Direct Planning
Direct RL
Value
Function
Greedification
Policy

Conclusion



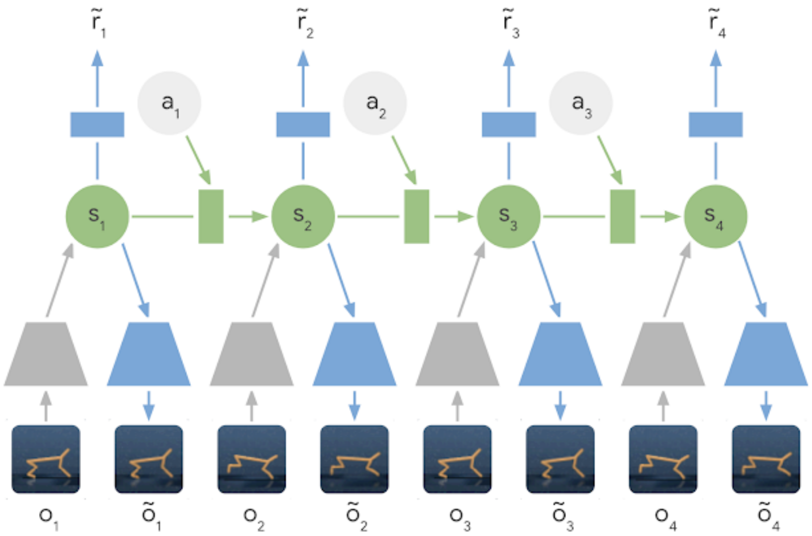
Interaction
with Environment
Experience
Model Learning
Model
Simulated Experience
Direct Planning
Value
Function
Direct RL
?
Policy
Greedification

Conclusion

- Model-Based RL is most Useful:
  - When Trajectories are more « expensive » than « thinking/compute ».
  - When environment or reward can change

Using Approximate Models: PlaNet (Hafner et al, 2019)

Only has a model M, no V,Q, or π



- Building on world models work by Ha and Schmidhuber (2017)
- Learn a model that tries to fit the observations (using a loss function)

https://arxiv.org/pdf/1811.04551.pdf

Using Approximate Models: PlaNet (Hafner et al, 2019)

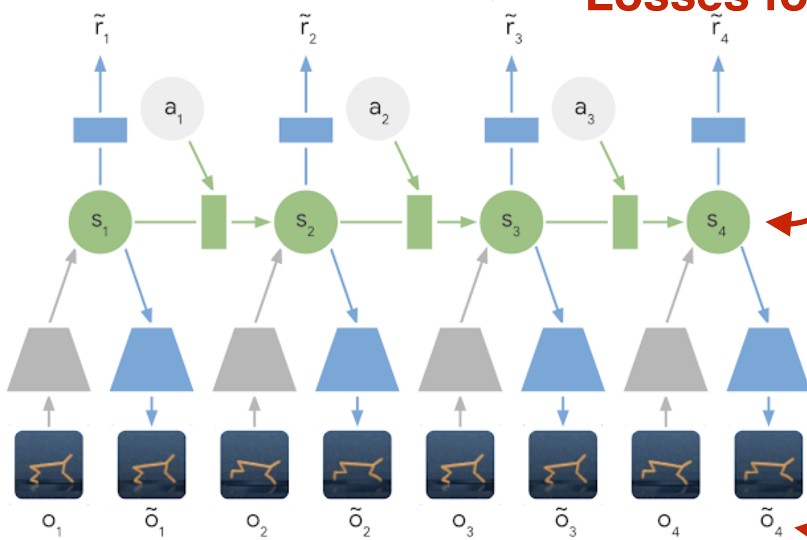
Losses for NeuralNet Training

Algorithm 1: Deep Planning Network (PlaNet)

Input :

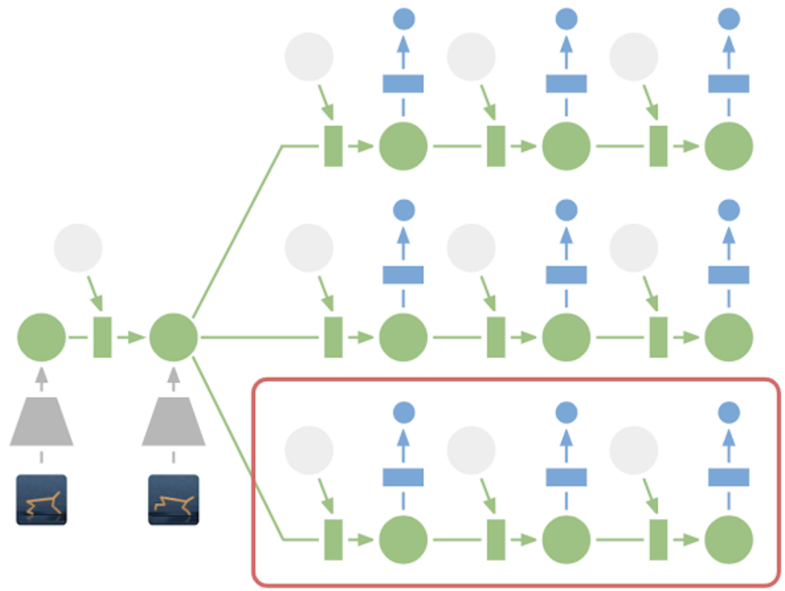
R	Action repeat	$p(s_t \mid s_{t-1}, a_{t-1})$	Transition model
S	Seed episodes	$p(o_t \mid s_t)$	Observation model
C	Collect interval	$p(r_t \mid s_t)$	Reward model
B	Batch size	$q(s_t \mid o_{\leq t}, a_{<t})$	Encoder
L	Chunk length	$p(\epsilon)$	Exploration noise
α	Learning rate		

Initialize dataset $\mathcal{D}$ with S random seed episodes.
Initialize model parameters θ randomly.
while *not converged* **do**
  `// Model fitting`
  for *update step* $s = 1..C$ **do**
    Draw sequence chunks $\{(o_t, a_t, r_t)_{t=k}^{L+k}\}_{i=1}^{B} \sim \mathcal{D}$ uniformly at random from the dataset.
    Compute loss $\mathcal{L}(\theta)$ from Equation 3.
    Update model parameters $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}(\theta)$.
  `// Data collection`
  $o_1 \leftarrow$ `env.reset()`
  for *time step* $t = 1..\lceil \frac{T}{R} \rceil$ **do**
    Infer belief over current state $q(s_t \mid o_{\leq t}, a_{<t})$ from the history.
    $a_t \leftarrow$ `planner`$(q(s_t \mid o_{\leq t}, a_{<t}), p)$, see Algorithm 2 in the appendix for details.
    Add exploration noise $\epsilon \sim p(\epsilon)$ to the action.
    for *action repeat* $k = 1..R$ **do**
      $r_t^k, o_{t+1}^k \leftarrow$ `env.step`(a_t)
    $r_t, o_{t+1} \leftarrow \sum_{k=1}^{R} r_t^k, o_{t+1}^R$
  $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_t)_{t=1}^{T}\}$



https://arxiv.org/pdf/1811.04551.pdf

PlaNet Planning Process



B. Planning Algorithm

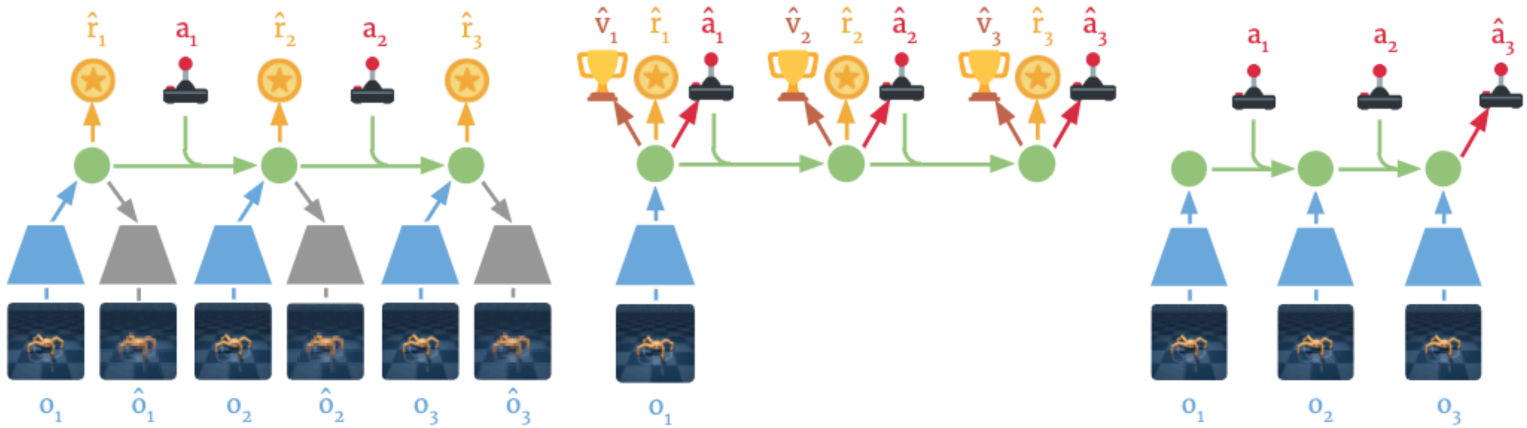
Algorithm 2: Latent planning with CEM

Input:				
	H	Planning horizon distance	$q(s_t \mid o_{\leq t}, a_{<t})$	Current state belief
	I	Optimization iterations	$p(s_t \mid s_{t-1}, a_{t-1})$	Transition model
	J	Candidates per iteration	$p(r_t \mid s_t)$	Reward model
	K	Number of top candidates to fit		

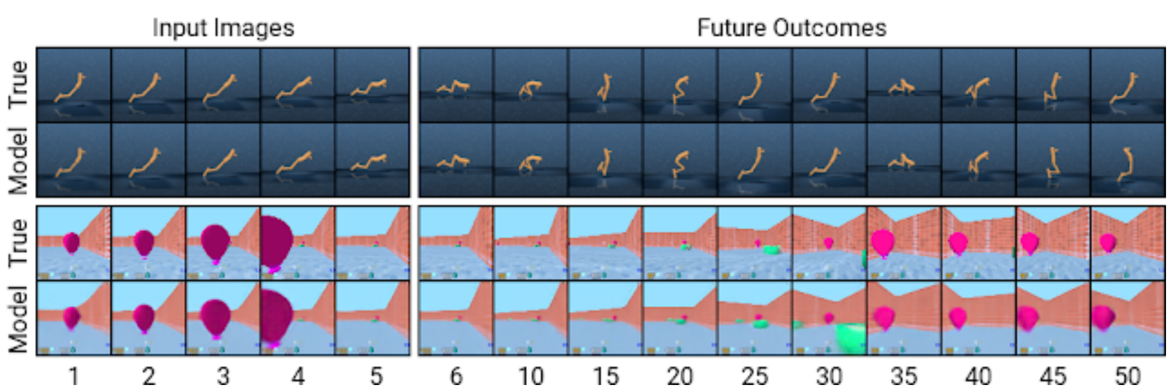
Initialize factorized belief over action sequences $q(a_{t:t+H}) \leftarrow \mathrm{Normal}(0, \mathbb{I})$.
for *optimization iteration* $i = 1..I$ **do**
    `// Evaluate` J `action sequences from the current belief.`
    for *candidate action sequence* $j = 1..J$ **do**
        $a^{(j)}_{t:t+H} \sim q(a_{t:t+H})$
        $s^{(j)}_{t:t+H+1} \sim q(s_t \mid o_{1:t}, a_{1:t-1}) \prod_{\tau=t+1}^{t+H+1} p(s_\tau \mid s_{\tau-1}, a^{(j)}_{\tau-1})$
        $R^{(j)} = \sum_{\tau=t+1}^{t+H+1} \mathrm{E}[p(r_\tau \mid s^{(j)}_\tau)]$
    `// Re-fit belief to the` K `best action sequences.`
    $\mathcal{K} \leftarrow \mathrm{argsort}(\{R^{(j)}\}_{j=1}^{J})_{1:K}$
    $\mu_{t:t+H} = \frac{1}{K} \sum_{k \in \mathcal{K}} a^{(k)}_{t:t+H}, \quad \sigma_{t:t+H} = \frac{1}{K-1} \sum_{k \in \mathcal{K}} |a^{(k)}_{t:t+H} - \mu_{t:t+H}|.$
    $q(a_{t:t+H}) \leftarrow \mathrm{Normal}(\mu_{t:t+H}, \sigma^2_{t:t+H} \mathbb{I})$
return *first action mean* μ_t.

- At planning time, only abstract states are generated
- Planning using Cross-Entropy-Method Algo

Dreamer (Hafner et al, 2020)



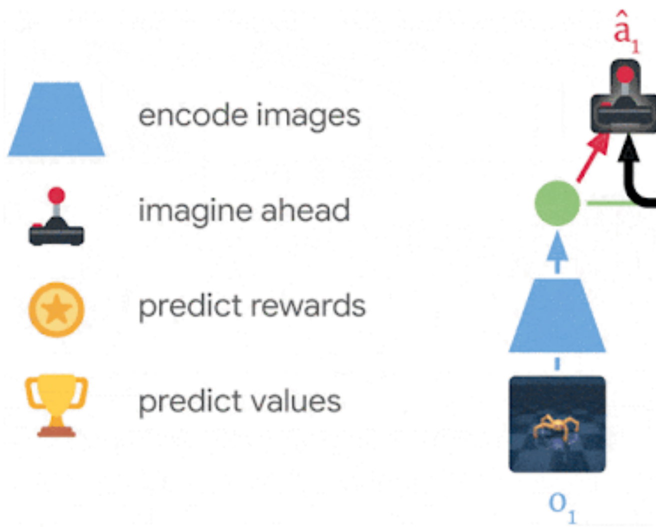
(a) Learn dynamics from experience (b) Learn behavior in imagination (c) Act in the environment



Has model M, state value fct V, and policy π, but no Q.

https://arxiv.org/abs/1912.01603

Value Propagation in Dreamer



Algorithm 1: Dreamer

Initialize dataset $\mathcal{D}$ with S random seed episodes.
Initialize neural network parameters θ, ϕ, ψ randomly.
while *not converged* **do**
  for *update step* $c = 1..C$ **do**
    `// Dynamics learning`
    Draw B data sequences $\{(a_t, o_t, r_t)\}_{t=k}^{k+L} \sim \mathcal{D}$.
    Compute model states $s_t \sim p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$.
    Update θ using representation learning.
    `// Behavior learning`
    Imagine trajectories $\{(s_\tau, a_\tau)\}_{\tau=t}^{t+H}$ from each s_t.
    Predict rewards $\mathrm{E}\big(q_\theta(r_\tau \mid s_\tau)\big)$ and values $v_\psi(s_\tau)$.
    Compute value estimates $\mathrm{V}_\lambda(s_\tau)$ via Equation 6.
    Update $\phi \leftarrow \phi + \alpha \nabla_\phi \sum_{\tau=t}^{t+H} \mathrm{V}_\lambda(s_\tau)$.
    Update $\psi \leftarrow \psi - \alpha \nabla_\psi \sum_{\tau=t}^{t+H} \frac{1}{2} \big\| v_\psi(s_\tau) - \mathrm{V}_\lambda(s_\tau) \big\|^2$.
  `// Environment interaction`
  $o_1 \leftarrow$ `env.reset()`
  for *time step* $t = 1..T$ **do**
    Compute $s_t \sim p_\theta(s_t \mid s_{t-1}, a_{t-1}, o_t)$ from history.
    Compute $a_t \sim q_\phi(a_t \mid s_t)$ with the action model.
    Add exploration noise to action.
    $r_t, o_{t+1} \leftarrow$ `env.step(`a_t`)`.
  Add experience to dataset $\mathcal{D} \leftarrow \mathcal{D} \cup \{(o_t, a_t, r_t)_{t=1}^{T}\}$.

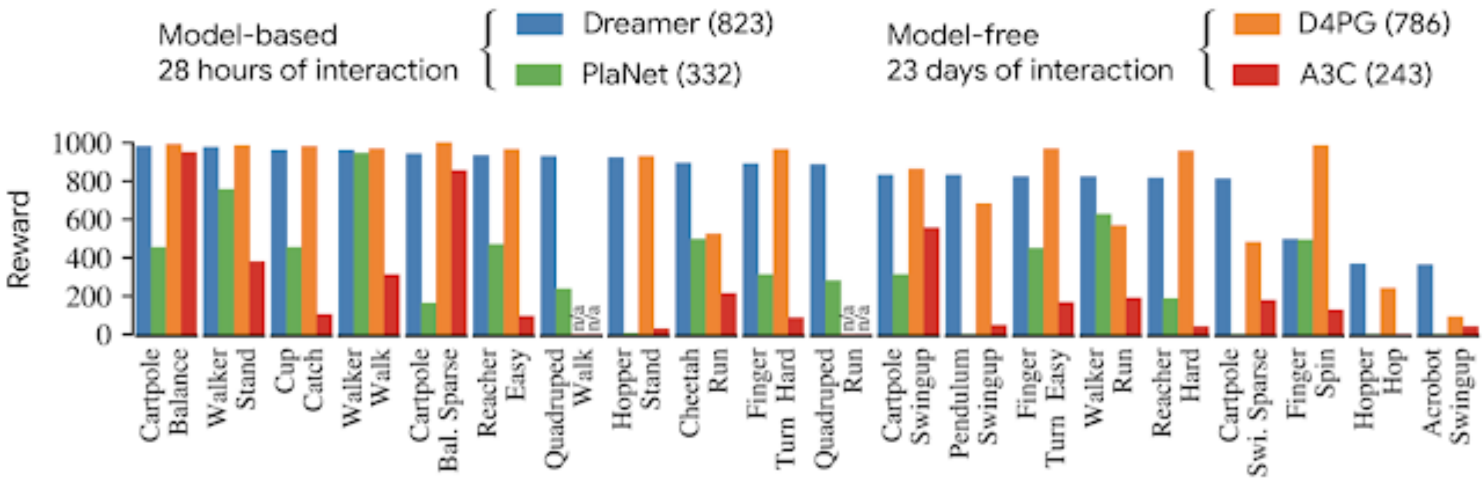
Model components

Representation	$p_\theta(s_t \mid s_{t\text{-}1}, a_{t\text{-}1}, o_t)$
Model	$q_\theta(s_t \mid s_{t\text{-}1}, a_{t\text{-}1})$
Reward	$q_\theta(r_t \mid s_t)$
Policy	$q_\phi(a_t \mid s_t)$
Value	$v_\psi(s_t)$

Hyper parameters

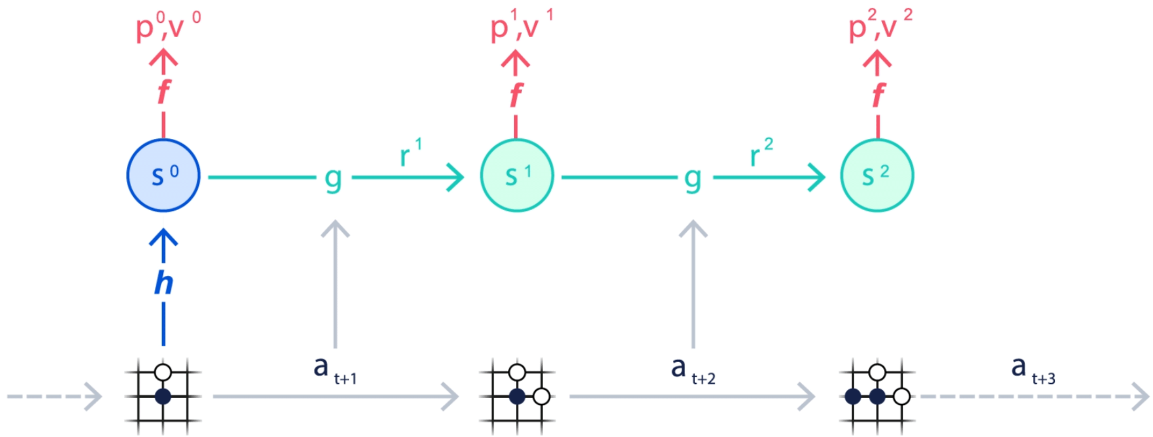
Seed episodes	S
Collect interval	C
Batch size	B
Sequence length	L
Imagination horizon	H
Learning rate	α

Dreamer and Planet Results



Model-based methods achieve comparable results to model-free with much less data

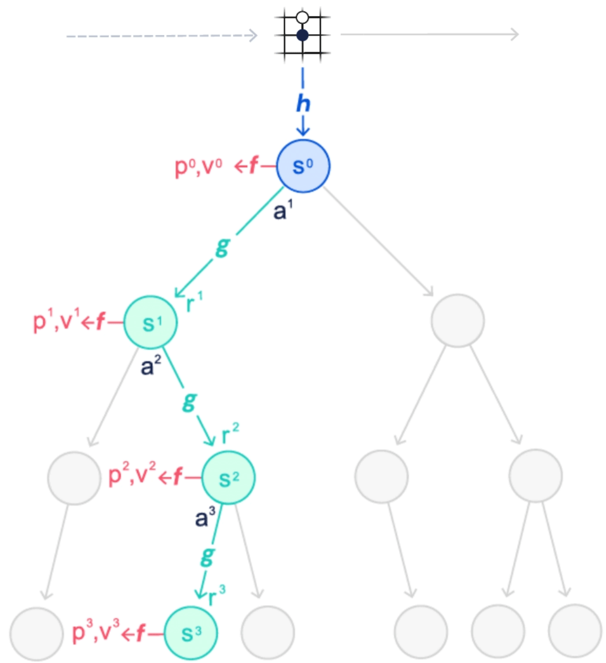
Using Approximate Models: MuZero (Schrittwieser et al, Nature, 2020)



- Rather than predict the entire environment, make sure predictions are accurate for values, rewards and actions
- Values are trained with observed returns, actions to mimic the policy obtained through search

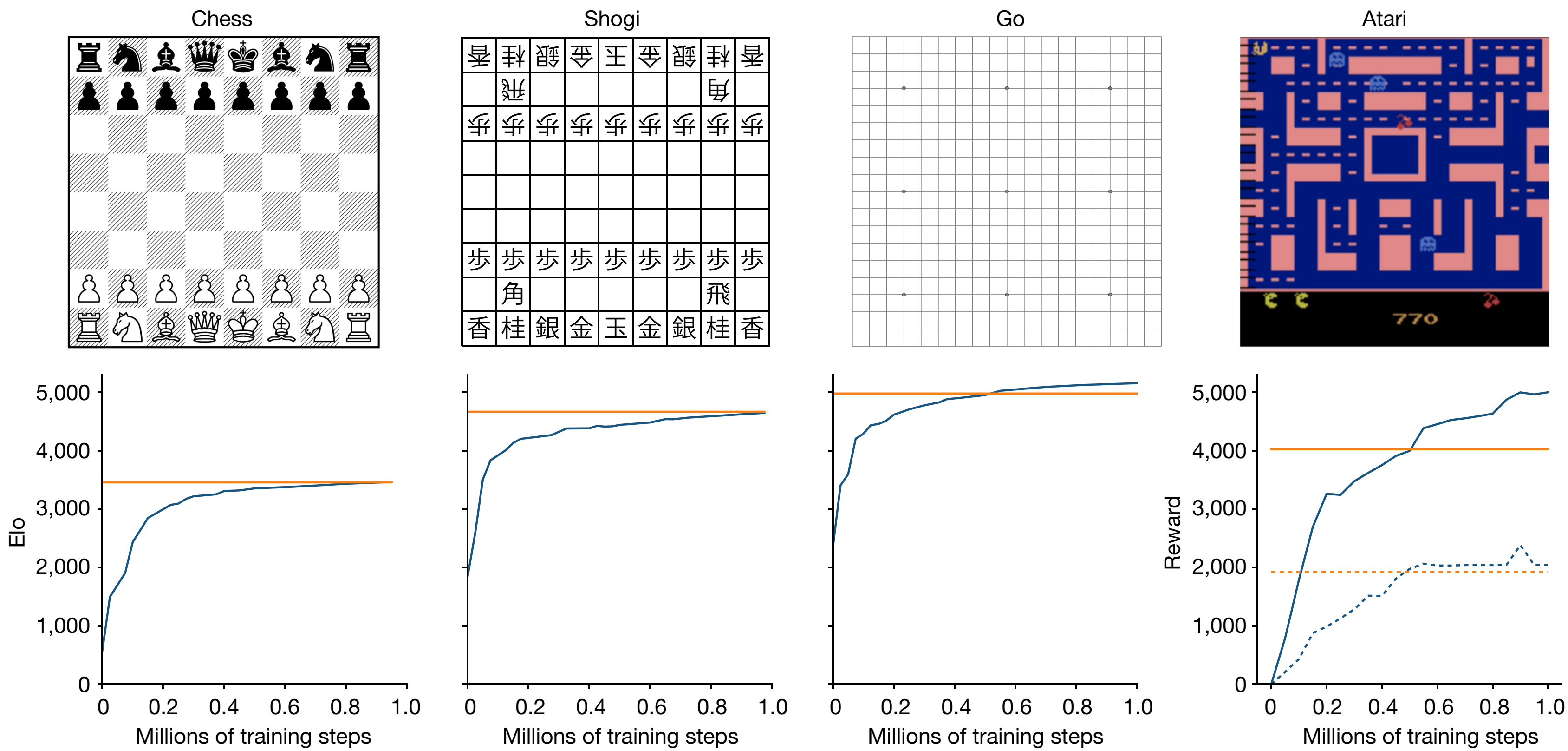
https://arxiv.org/pdf/1911.08265.pdf

Execution in MuZero



- Model is rolled forward in Monte Carlo Tree Search-style

MuZero Results



MuZero outperforms R2D2 (best model-free agent at the time)

How do we decide what to do?

- Emotions/Intuition 

$$V_t(s) \qquad Q_t(s,a)$$

- Thinking 

$$S_{t+1} = M(S_t, A_t, \theta)$$

- Reflexes/Habits 

$$A_t = \pi(S_t, \theta)$$

Why approximate policies rather than values?

- In many problems, the policy is simpler to approximate than the value function
- In many problems, the optimal policy is stochastic
  - e.g., bluffing, POMDPs
- To enable smoother change in policies
- To avoid a search on every step (the max)
- To better relate to biology

Policy Approximation

We want to learn this directly!

$$\pi(a|s, \boldsymbol{\theta})$$

- Policy = a function from state to action
  - How does the agent select actions?
  - In such a way that it can be affected by learning?
  - In such a way as to assure exploration?
- Approximation: there are too many states and/or actions to represent all policies
  - To handle large/continuous action spaces

Gradient-bandit algorithm

- Store action preferences $H_t(a)$
  rather than action-value estimates $Q_t(a)$
- Instead of ε-greedy, pick actions by an exponential soft-max:

$$\Pr\{A_t = a\} \doteq \frac{e^{H_t(a)}}{\sum_{b=1}^{k} e^{H_t(b)}} \doteq \pi_t(a)$$

- Also store the sample average of rewards as $\bar{R}_t$
- Then update:

$$\frac{\partial \pi_t(A_t)}{\partial H_t(a)} / \pi_t(A_t)$$

$$H_{t+1}(a) = H_t(a) + \alpha\big(R_t - \bar{R}_t\big)\big(\mathbf{1}_{a=A_t} - \pi_t(a)\big)$$

1 or 0, depending on whether
the predicate (subscript) is true

How can we learn $\pi(a|s, \boldsymbol{\theta})$?

How can we learn $\pi(a|s, \boldsymbol{\theta})$?

- Directly from Experience?

- From V and Q?

- From a World-Model M(S, A) = S' ?

How can we learn $\pi(a|s, \boldsymbol{\theta})$?

- Directly from Experience?
  - REINFORCE
- From V and Q?
  - Actor Critic Algorithms
  - Deterministic Policy Gradient (DPG)
- From a World-Model M(S, A) = S' ?