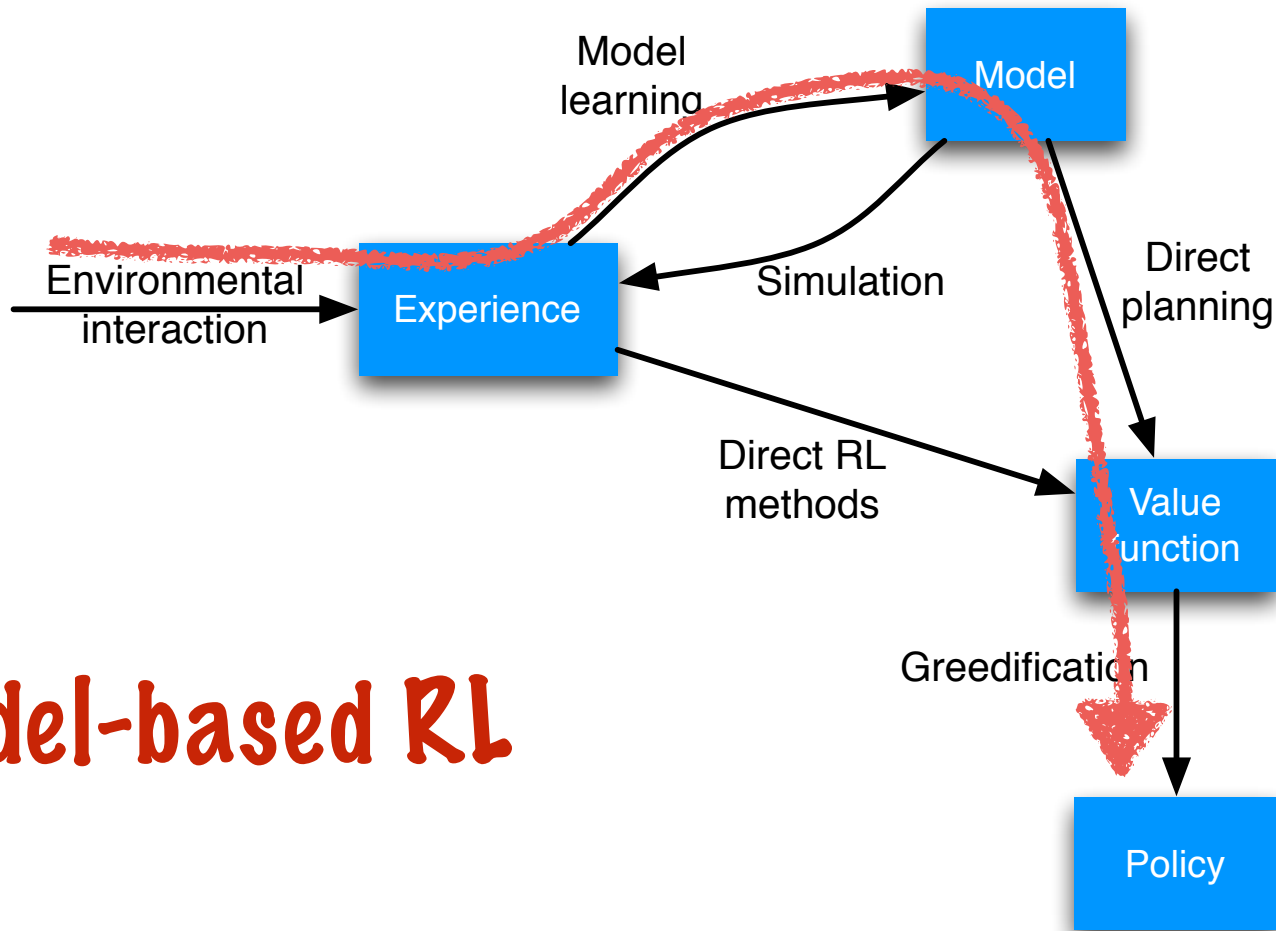


Chapter 8: Planning and Learning

Objectives of this chapter:

- To think more generally about uses of environment models
- Integration of (unifying) planning, learning, and execution
- “Model-based reinforcement learning”

Paths to a policy



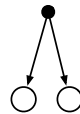
Model-based RL

Why Going Beyond Model-Free RL?

- Models provide “understanding” of the world (cf physics, causality...)
- Even if some parts of the problem change, others stay the same, which can help with faster learning
Eg. Reward may change but the layout and dynamics of the world may be the same
- Models can be used to “dream” up new experiences, and use them to update the value / policy

What should the model predict?

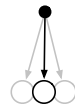
- Clearly we need the *reward*: easy problem, solved by regression
- What about the prediction of the *next state*?
 1. *Distribution model*: construct a distribution over next states / features



2. *Sample model*: have the ability to generate sampled next states / features



3. *Expectation model*: predict the expected next state / feature



DP with Distribution models

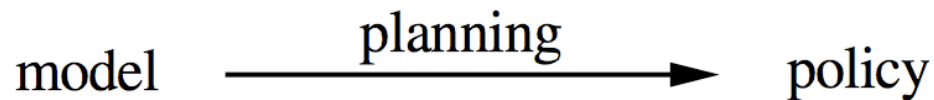
- In Chapter 4, we assumed access to a model of the world
 - These models describe all possibilities and their probabilities
 - We call them **Distribution models**
 - e.g., $p(s', r | s, a)$ for all s, a, s', r
- In Dynamic Programming we sweep the states:
 - in each state we consider all the possible rewards and next state values
 - the model describes the next states and rewards and their associated probabilities
 - using these values to update the value function
- In Policy Iteration, we then improve the policy using the computed value function

Sample Models

- **Model**: anything the agent can use to predict how the environment will respond to its actions
- **Sample model**, a.k.a. a simulation model
 - produces sample experiences for given s, a
 - sampled according to the probabilities
 - allows reset, exploring starts
 - often much easier to come by
- Both types of models can be used to mimic or simulate experience: to produce **hypothetical experience**

Planning

- **Planning**: any computational process that uses a model to create or improve a policy



- We take the following (unusual) view:
 - update value functions using both real and simulated experience
 - all state-space planning methods involve computing value functions, either explicitly or implicitly
 - they all apply updates from simulated experience



Planning Cont.

- Classical DP methods are state-space planning methods
- Heuristic search methods are state-space planning methods
- A planning method based on Q-learning:

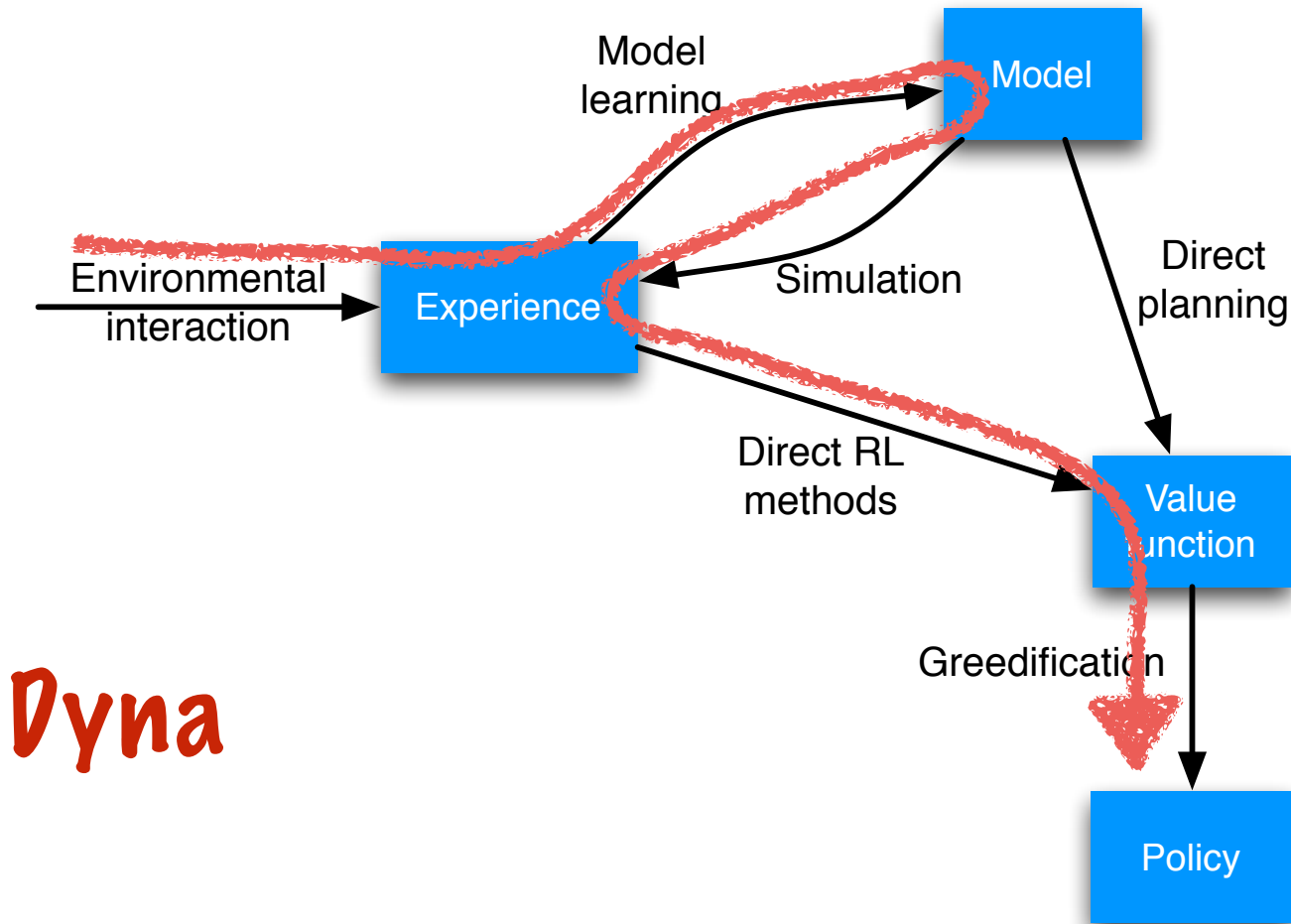
Random-sample one-step tabular Q-planning

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

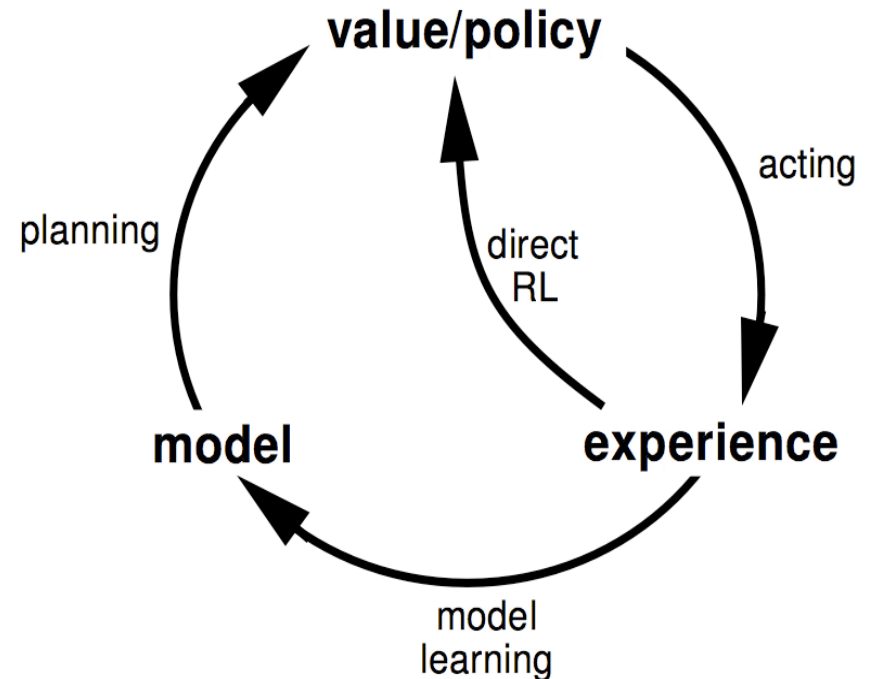
Environment program
Experiment program
Agent program

Paths to a policy

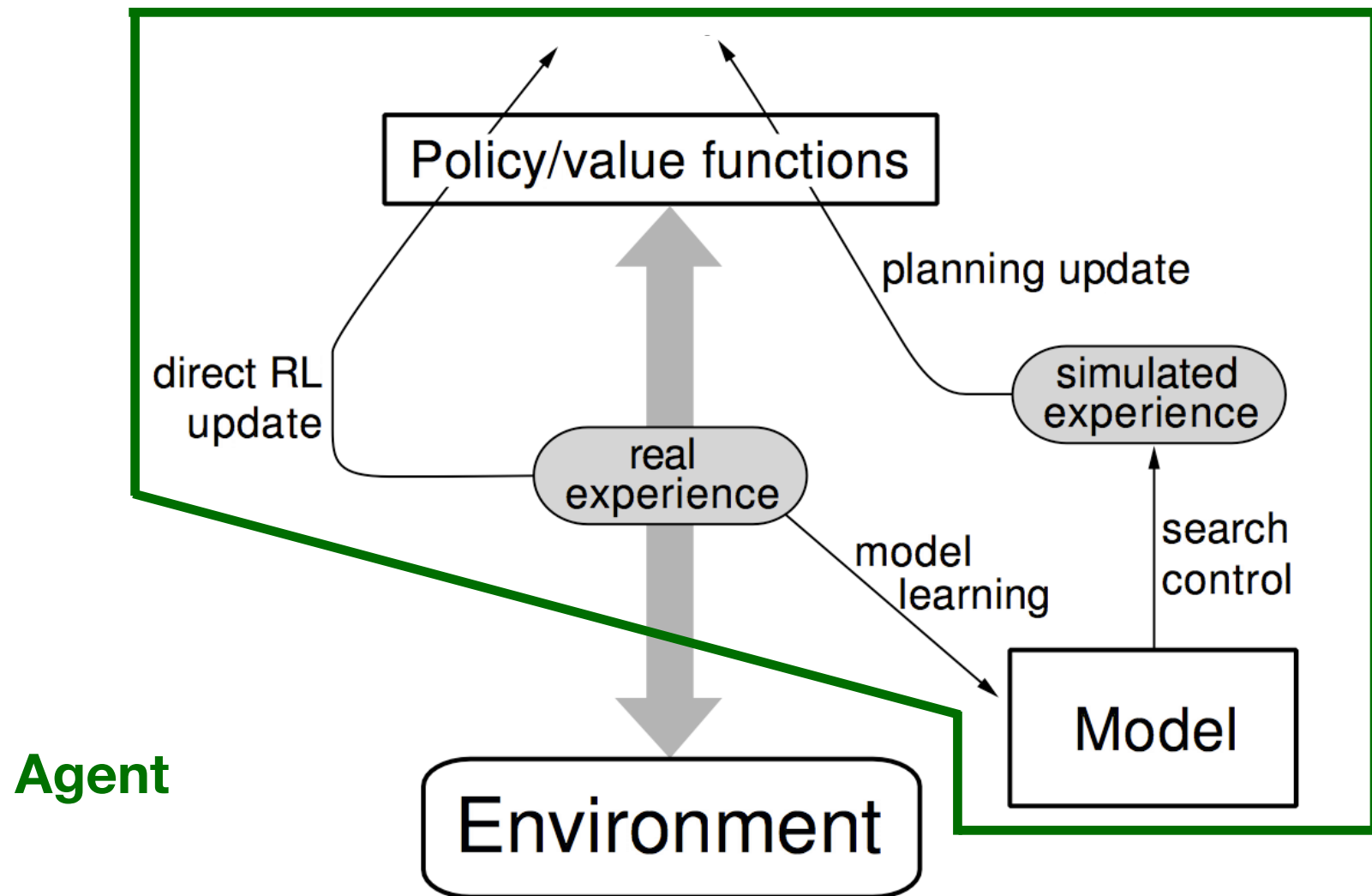


Learning, Planning, and Acting

- Two uses of real experience:
 - **model learning**: to improve the model
 - **direct RL**: to directly improve the value function and policy
- Improving value function and/or policy via a model is sometimes called **indirect RL**. Here, we call it **planning**.



The Dyna Architecture



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:

$S \leftarrow$ random previously observed state

$A \leftarrow$ random action previously taken in S

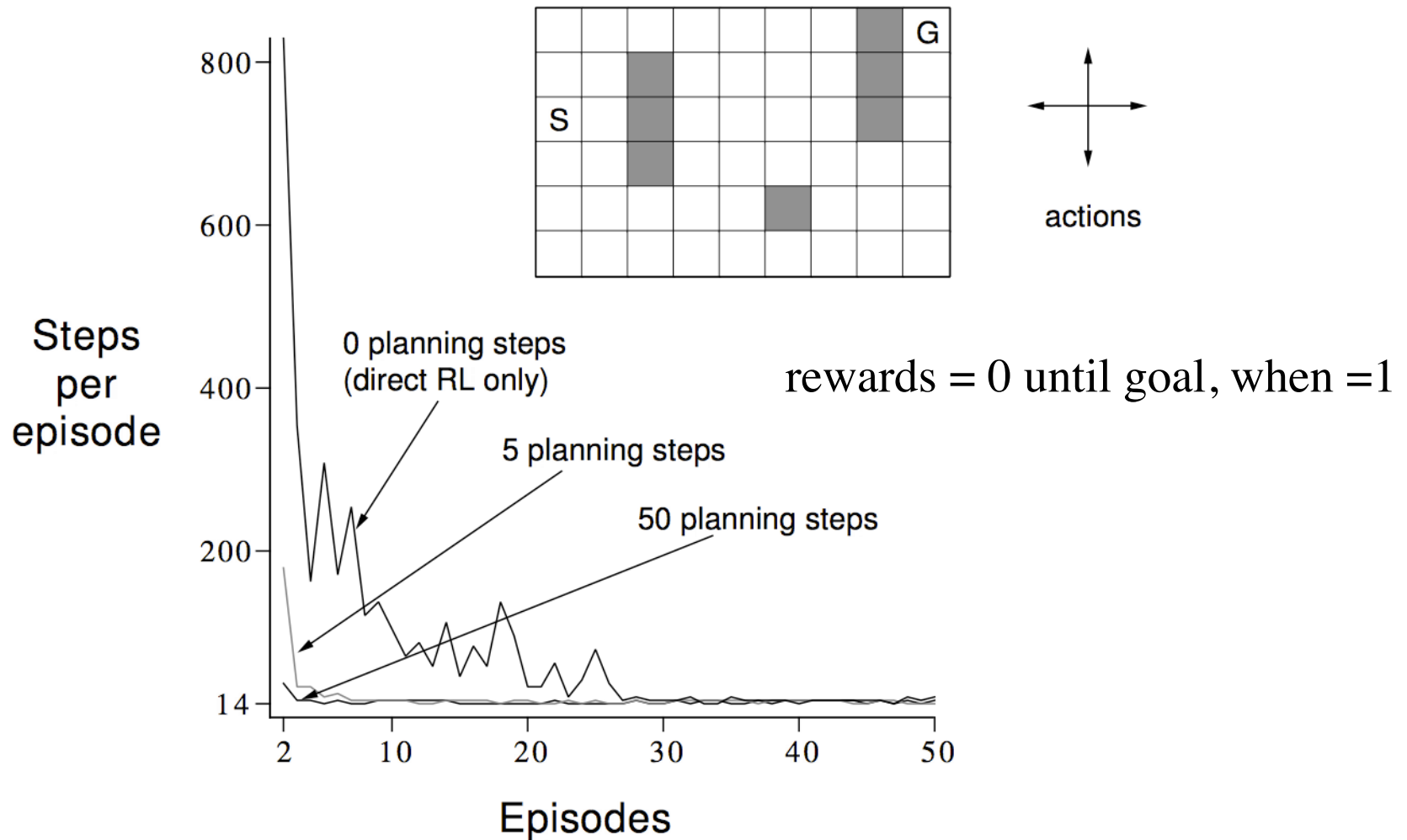
$R, S' \leftarrow Model(S, A)$

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

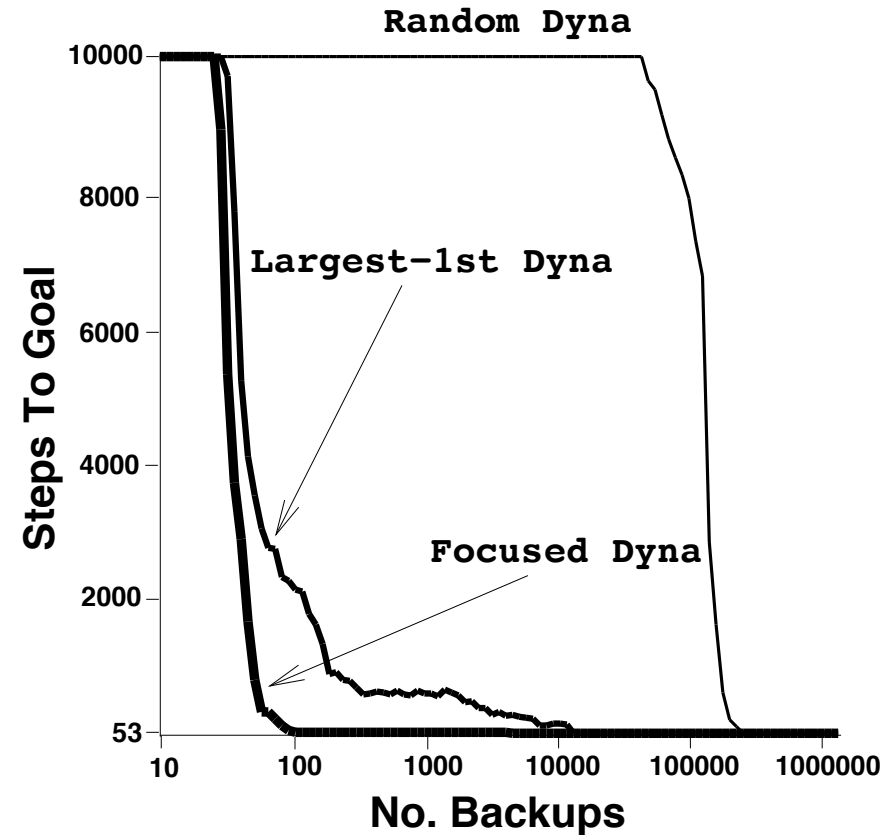
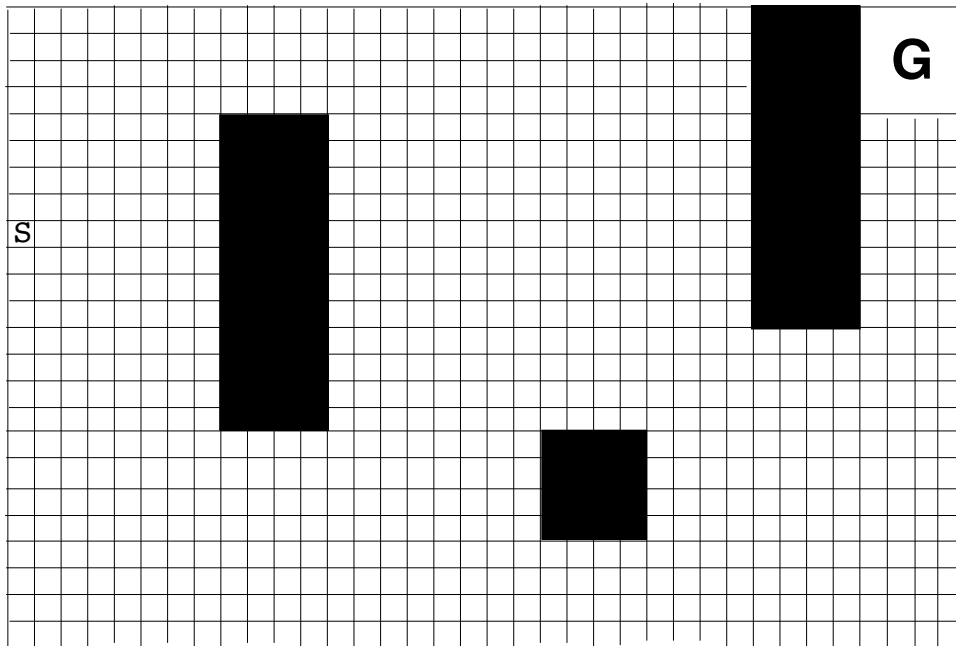
A simple maze: problem description

- 47 states, 4 actions, deterministic dynamics
- Obstacles and walls
- Rewards are 0 except +1 for transition into goal state
- $\gamma = 0.95$, discounted episodic task
- Agent parameters:
 - $\alpha = 0.1$, $\epsilon = 0.1$
 - Initial action-values were all zero
- Let's compare one-step tabular Q-learning and Dyna-Q with different values of n

Dyna-Q on a Simple Maze



Large maze and random search control

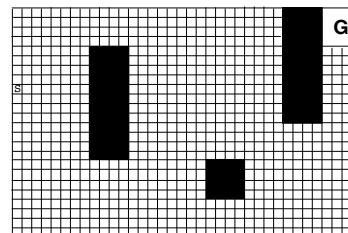
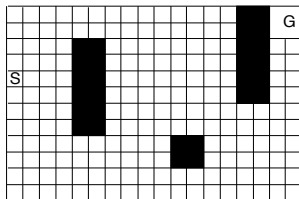
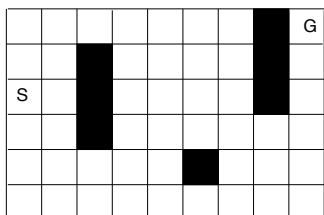


(Peng and Williams, 1993)

Prioritized Sweeping

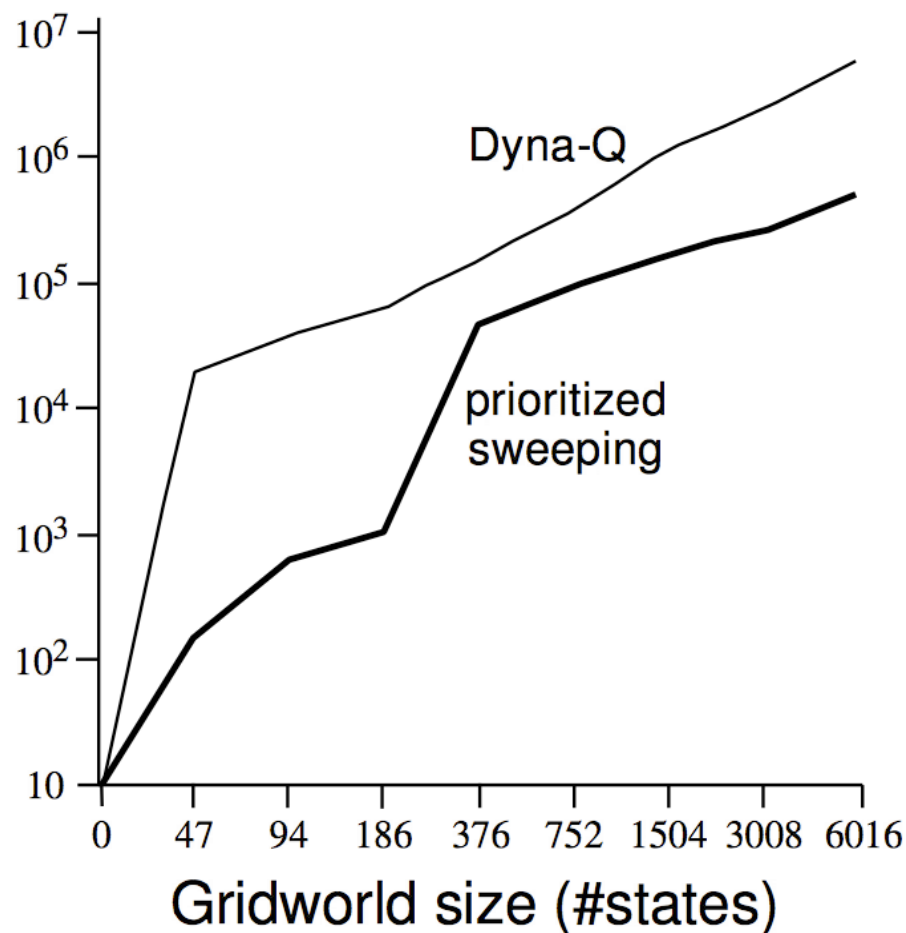
- Which states or state-action pairs should be generated during planning?
- Work backwards from states whose values have just changed:
 - Maintain a queue of state-action pairs whose values would change a lot if backed up, prioritized by the size of the change
 - When a new backup occurs, insert predecessors according to their priorities
 - Always perform backups from first in queue
- Moore & Atkeson 1993; Peng & Williams 1993
- improved by McMahan & Gordon 2005; Van Seijen 2013

Prioritized Sweeping vs. Dyna-Q

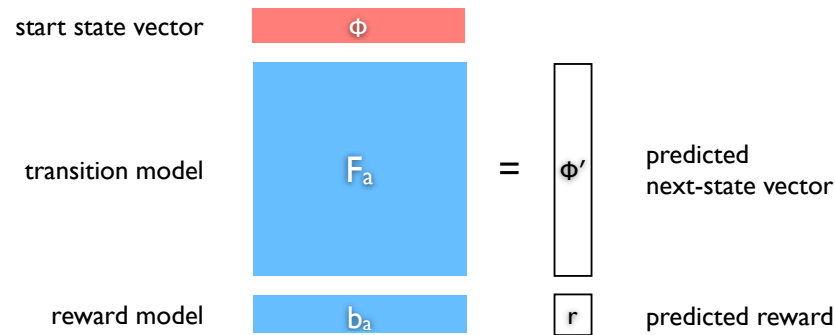


Backups
until
optimal
solution

Both use $n=5$ backups per
environmental interaction



Special case: Linear Expectation Models



- states are represented by feature vectors

$$s \longrightarrow \phi_s \quad s_t \longrightarrow \phi_t \quad \in \mathbb{R}^n$$

- the model is a set of matrix-vector pairs

$$M = \{F_a, b_a\}_{a \in Actions}$$

expected transition matrix

$$E\{\phi_{t+1} | \phi_t = \phi, a_t = a\} = F_a \phi$$

$$E\{r_{t+1} | \phi_t = \phi, a_t = a\} = b_a^\top \phi$$

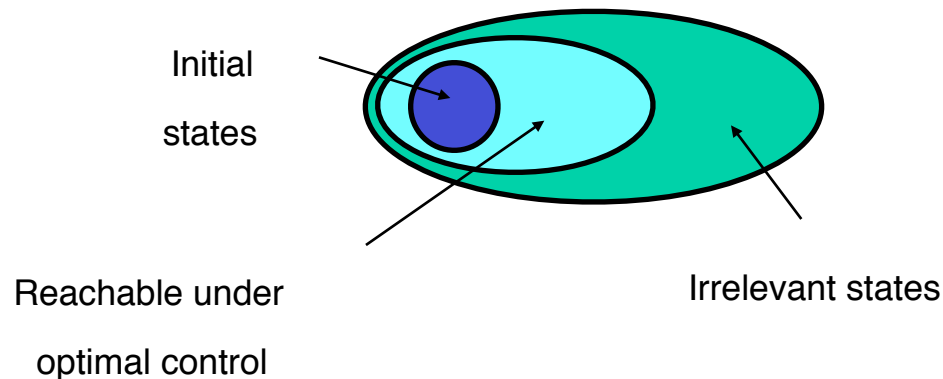
expected reward vector

Linear Dyna

- Use a linear model and a linear parametrization of the value function
- Note that the features ϕ could be non-linear (eg coming from a convnet) but they must be fixed
- In this case, value iteration using an expectation model is the same as using a full model

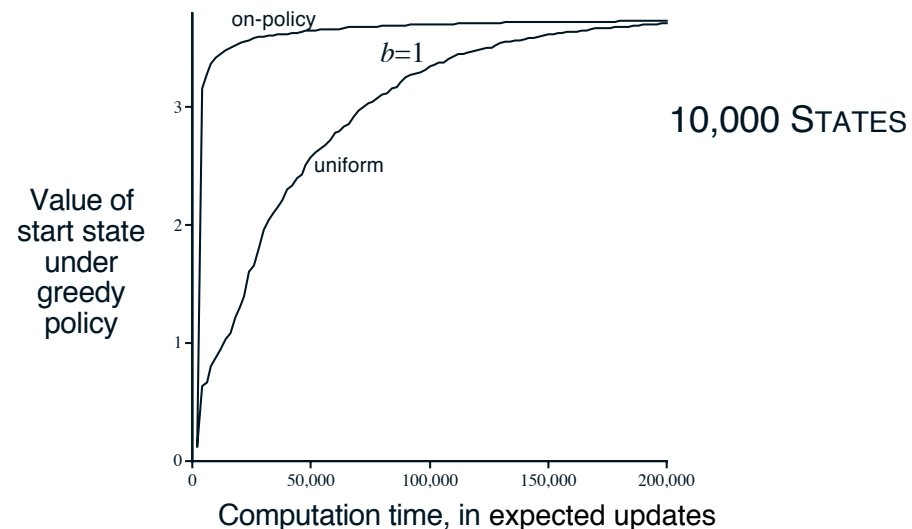
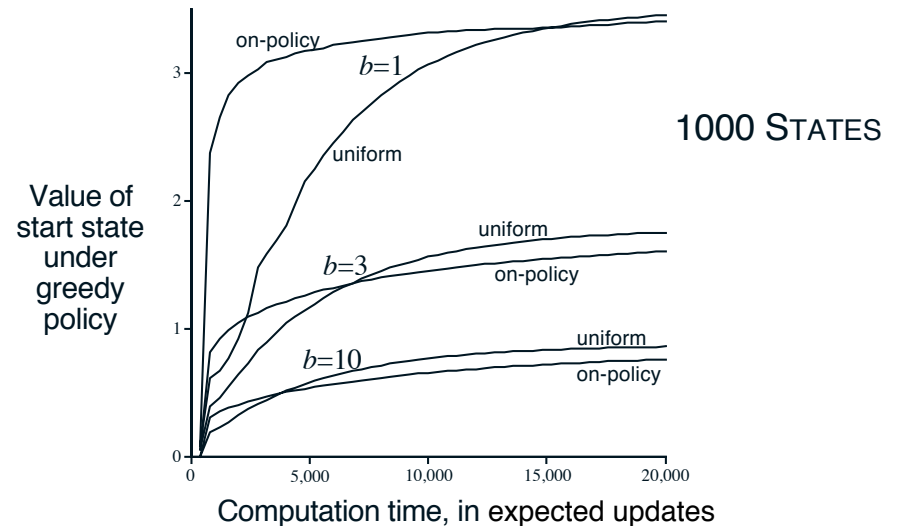
Trajectory Sampling

- **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:



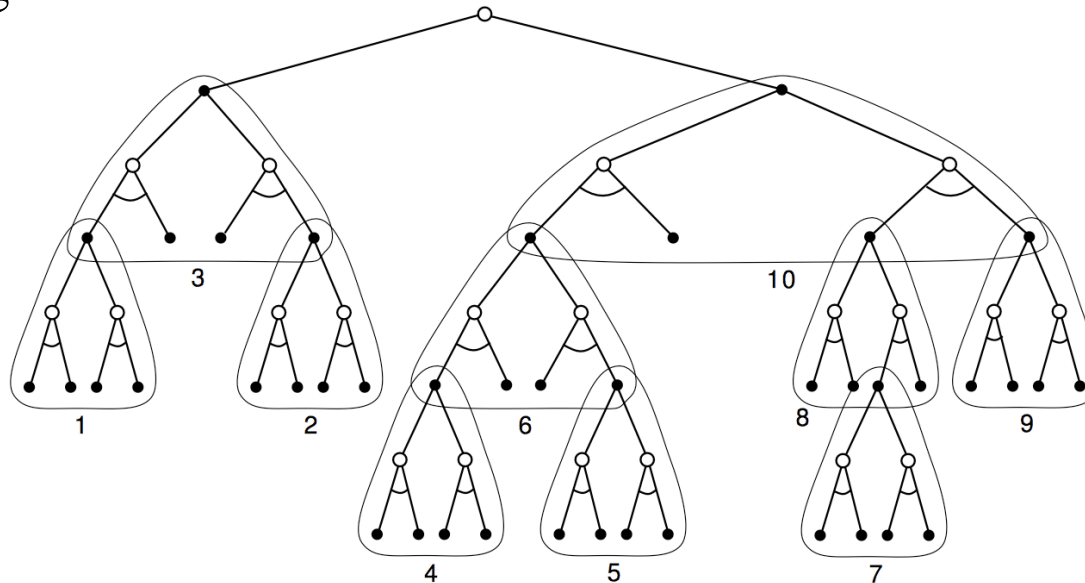
Trajectory Sampling Experiment

- 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

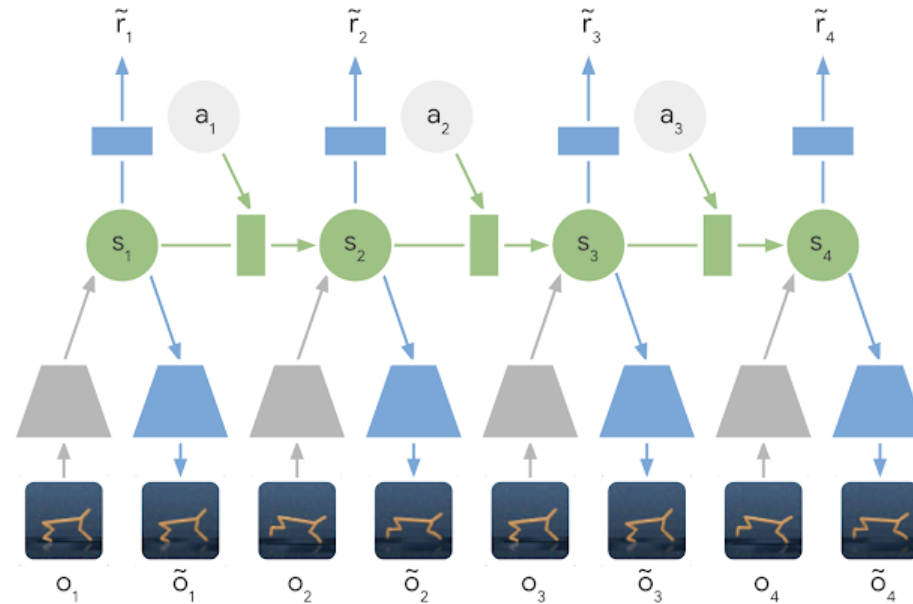


Heuristic Search

- 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:

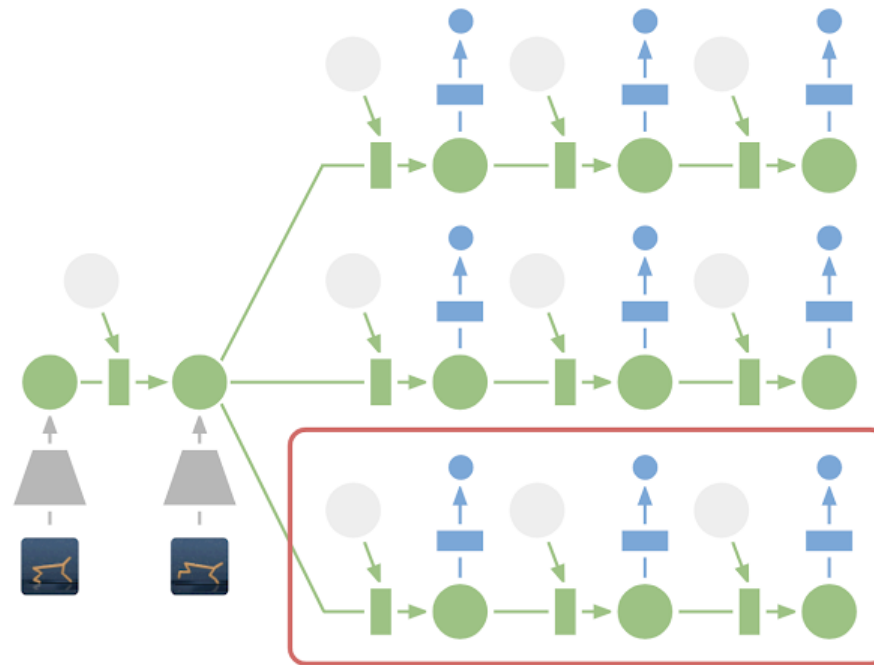


Using Approximate Models: PlaNet (Hafner et al, 2019)



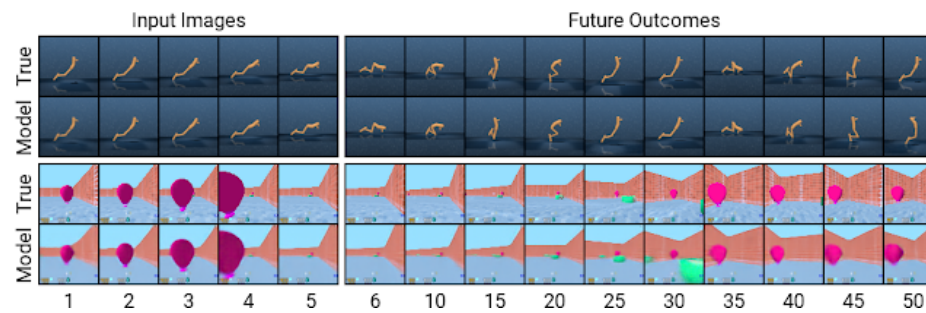
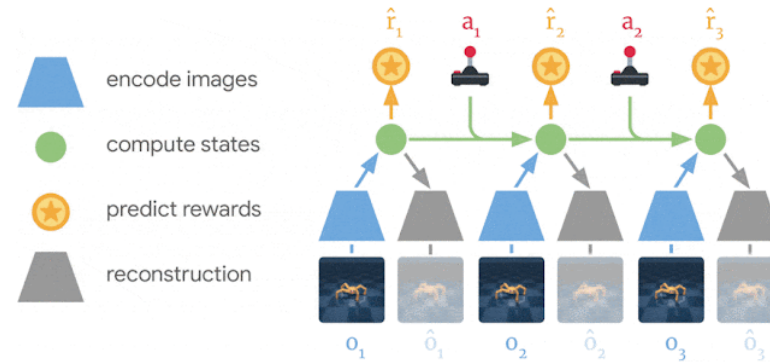
- Building on world models work by Ha and Schmidhuber (2017)
- Learn a model that tries to fit the observations (using a loss function)

PlaNet Planning Process

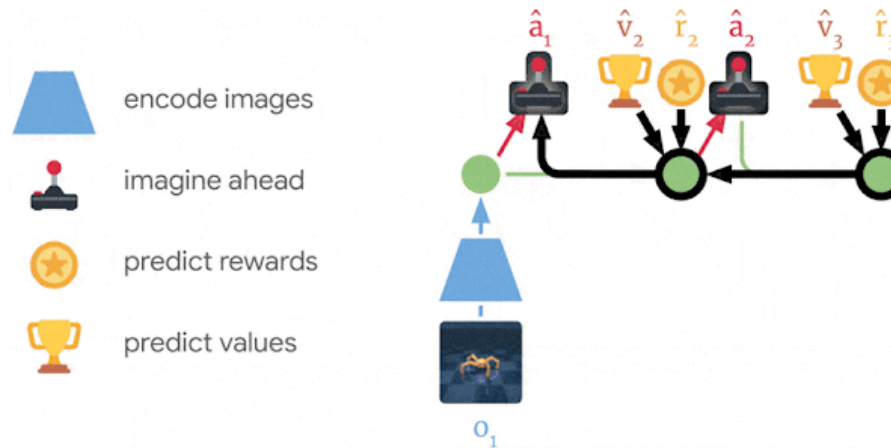


- At planning time, only abstract states are generated

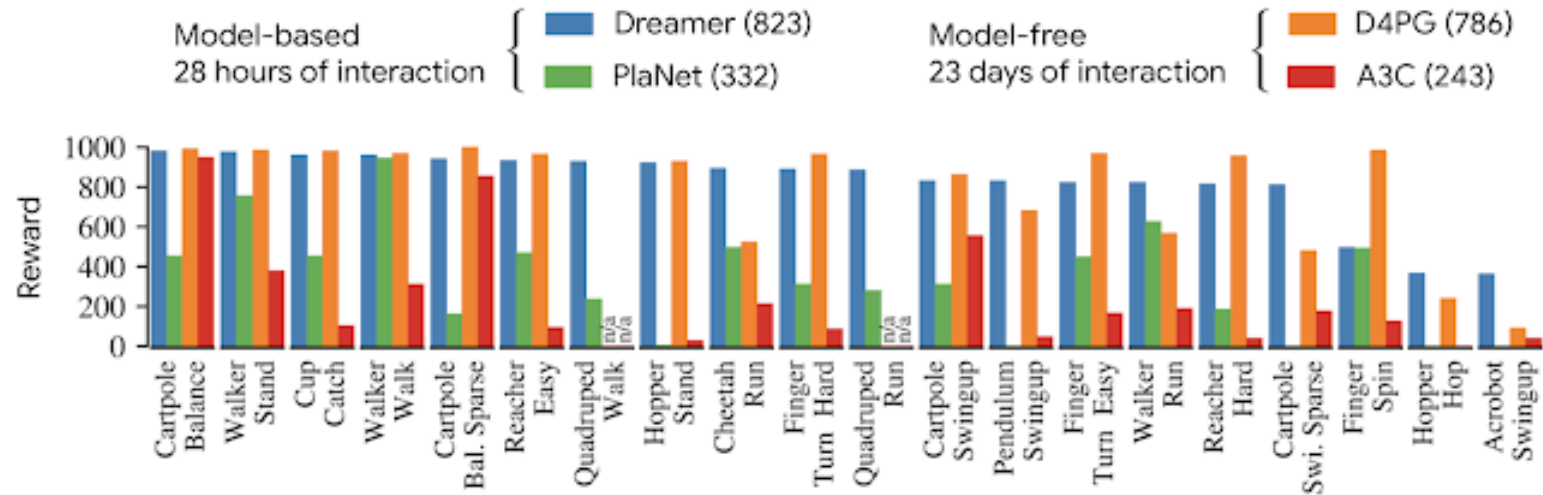
Dreamer (Hafner et al, 2020)



Value Propagation in Dreamer

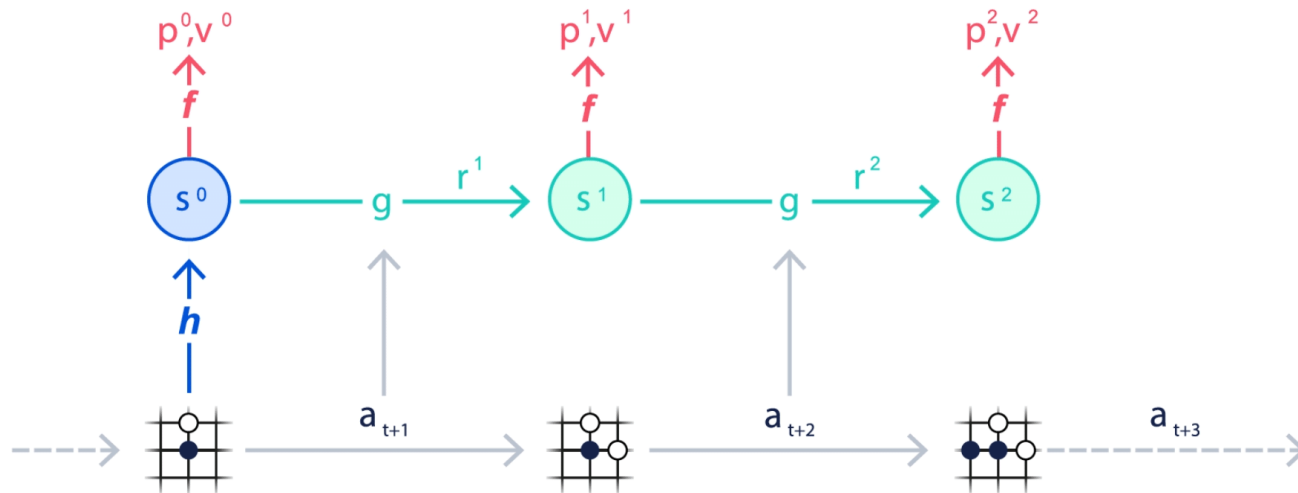


Dreamer and Planet Results



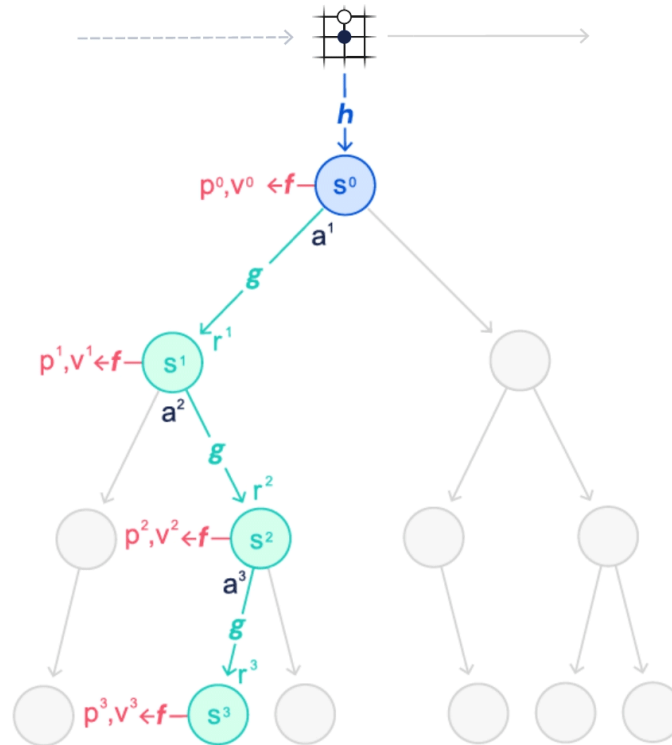
Model-based methods achieve comparable results to model-free with much less data

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



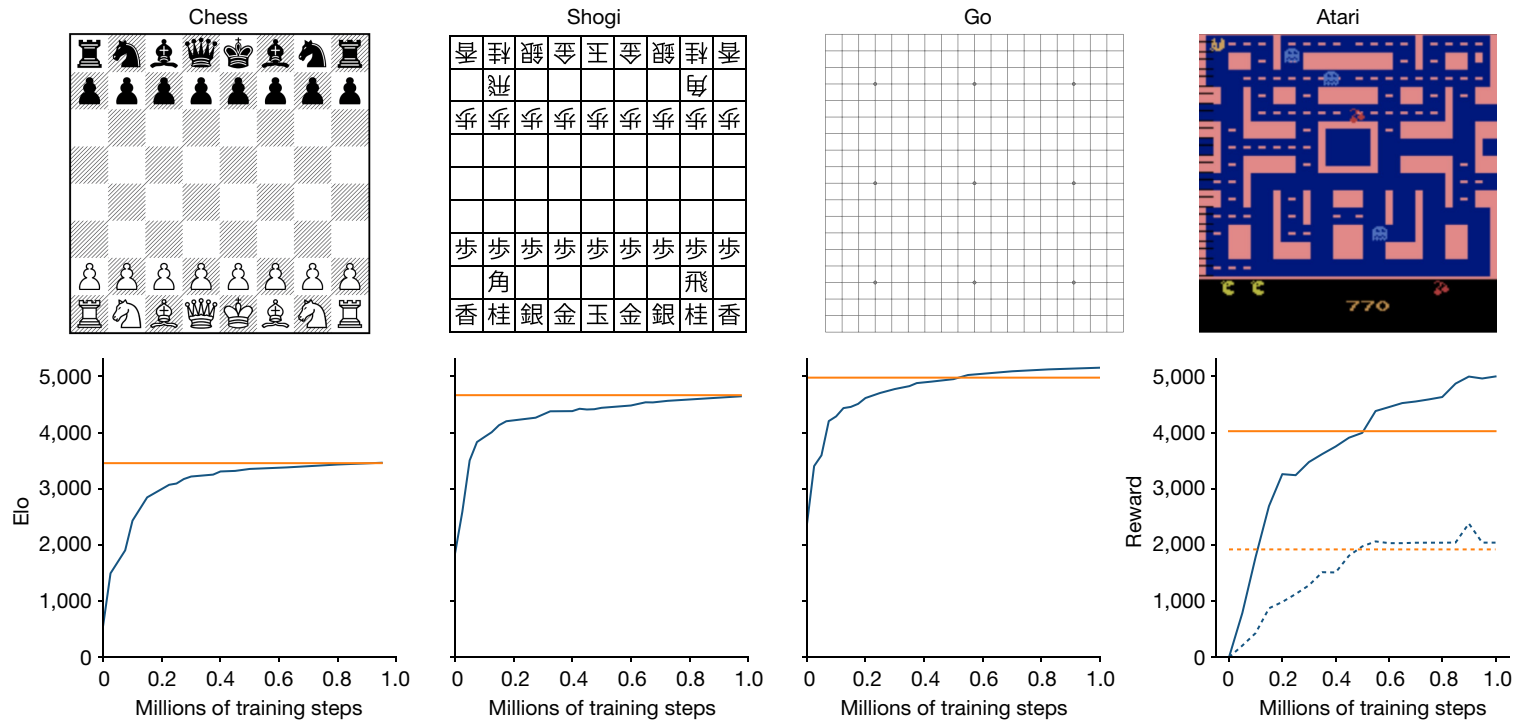
- Rather than predict the entire environment, make sure predictions are accurate for values, rewards and actions
- Values are trained with observed returns, actions to mimic the policy obtained through search

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)