Model-Based RL

How do we decide what to do?

- Emotions/Intuition $V_t(s) \qquad Q_t(s, a)$

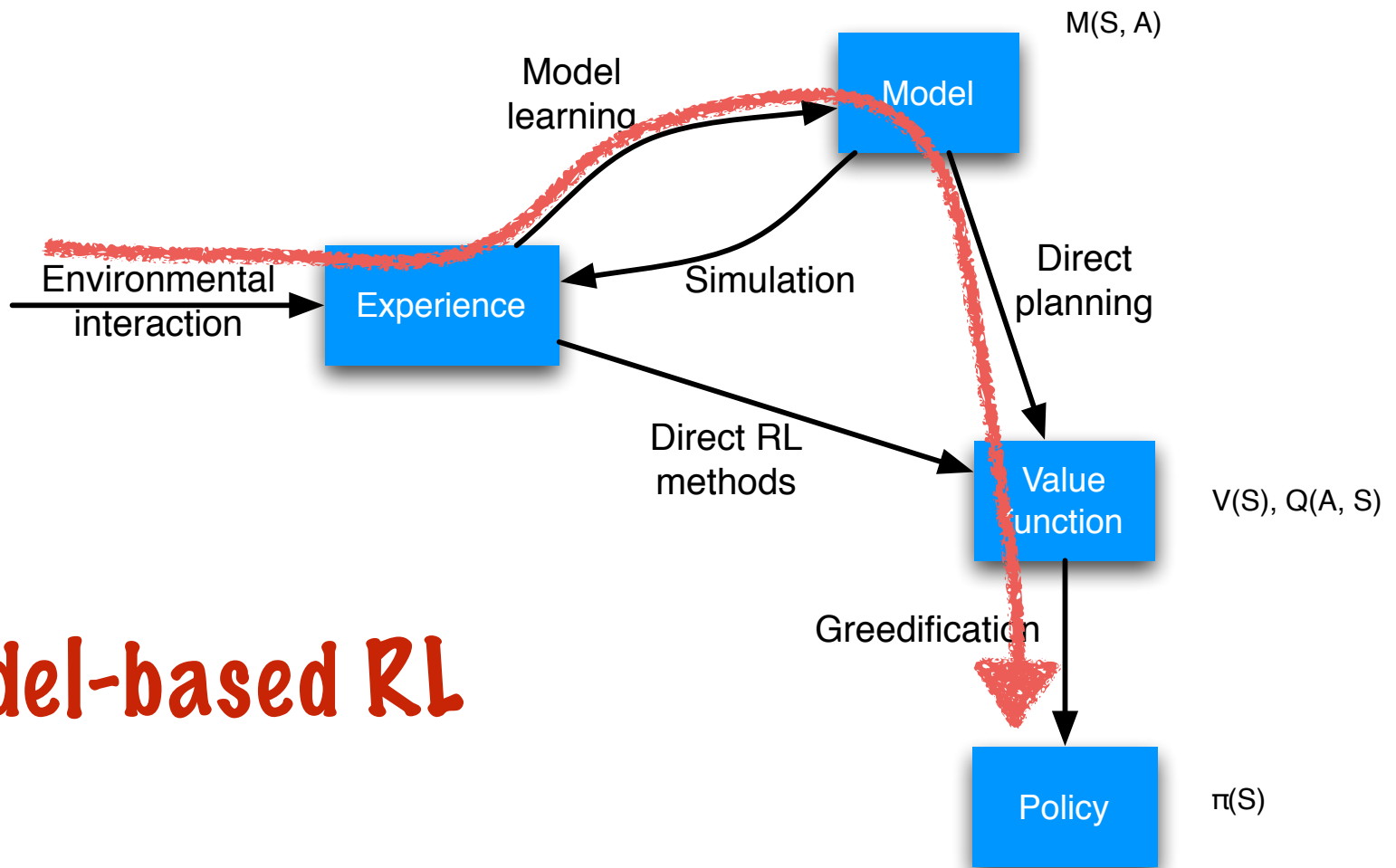- Thinking $S_{t+1} = M(S_t, A_t, \theta)$

- Reflexes/Habits

  THE KNEE-JERK REACTION

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

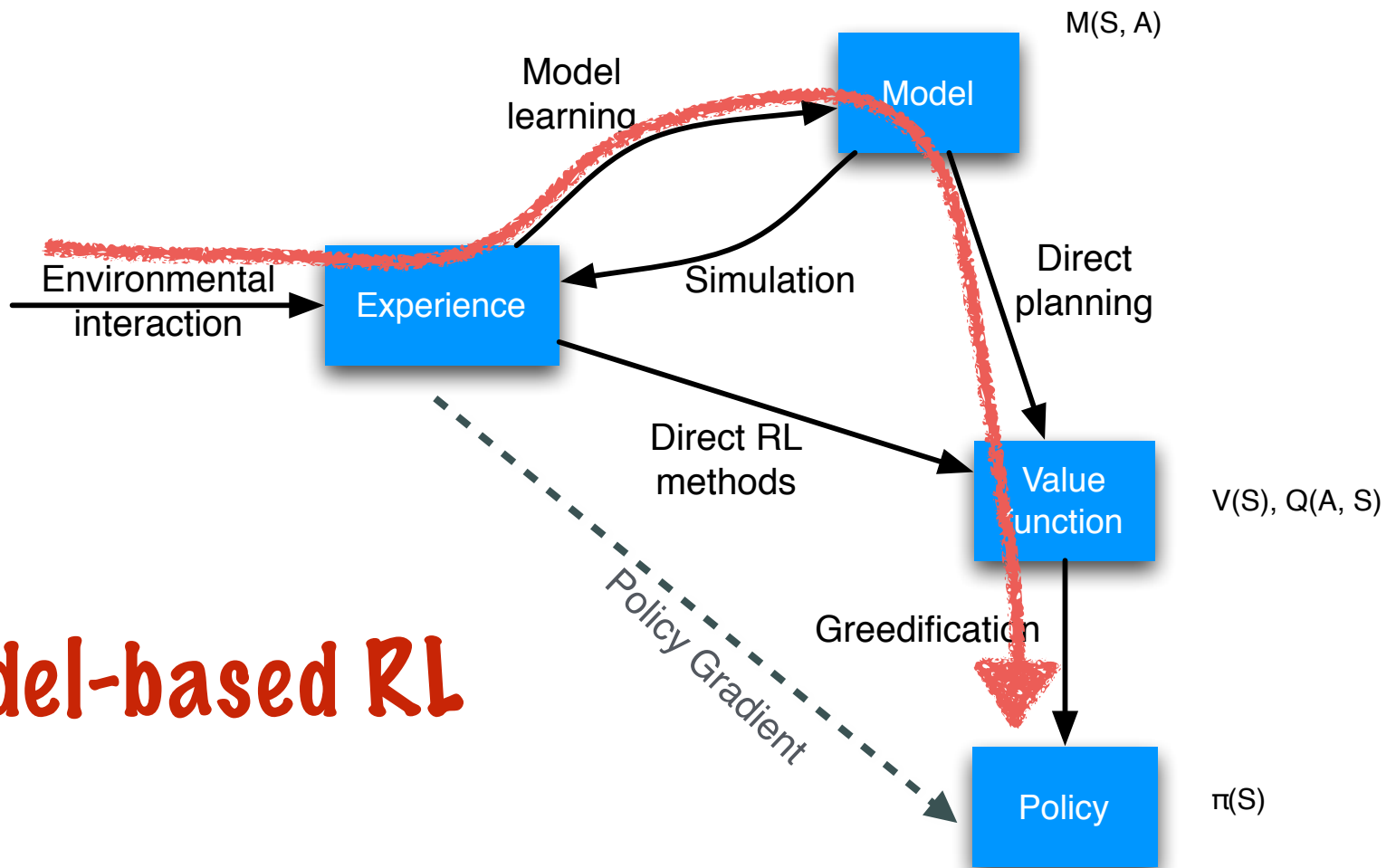# Chapter 8: Planning and Learning

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

M(S, A)

Model

Model learning

Environmental interaction

Experience

Simulation

Direct planning

Direct RL methods

Value function

V(S), Q(A, S)

Greedification

Model-based RL

Policy

π(S)

Paths to a policy

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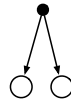
  2. *Sample model*: have the ability to generate sampled next states / features

  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., $p(s', r \mid s, a)$ for all s, a, s', r
- In Dynamic Programing 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 mimic or simulate experience: to produce hypothetical experience

Planning

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

$$\text{model} \xrightarrow{\quad \text{planning} \quad} \text{policy}$$
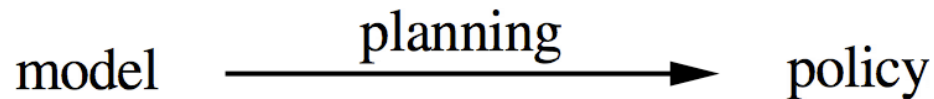
- We take the following (unusual) view:
  - update value functions using both real and simulated experience
  - all state-space planning methods involve computing value functions, either explicitly or implicitly
  - they all apply updates from simulated experience

$$\text{model} \longrightarrow \substack{\text{simulated} \\ \text{experience}} \xrightarrow{\quad \text{updates} \quad} \text{values} \longrightarrow \text{policy}$$

Planning Cont.

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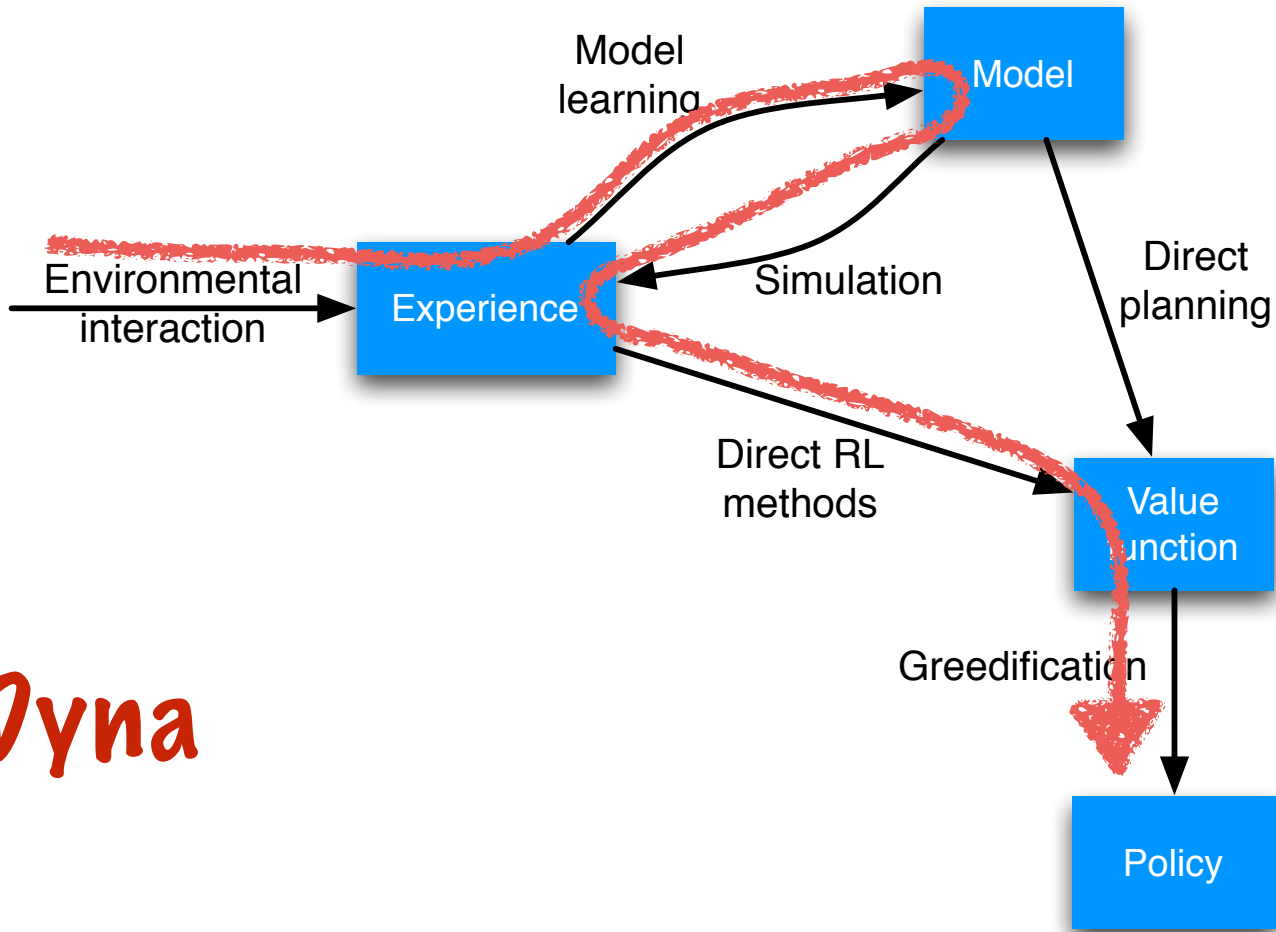- A planning method based on Q-learning:

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

Environment program
Experiment program
Agent program

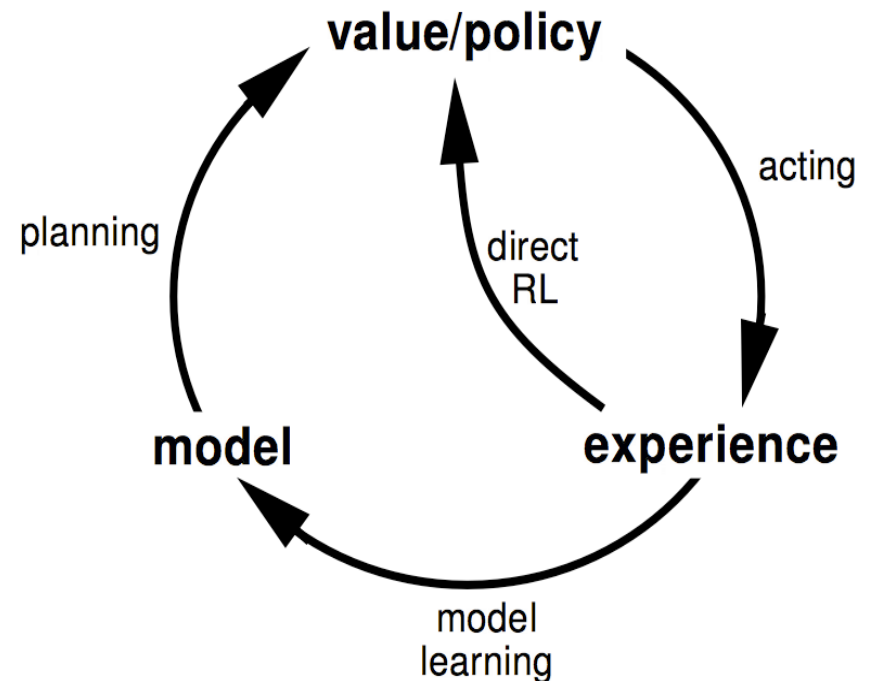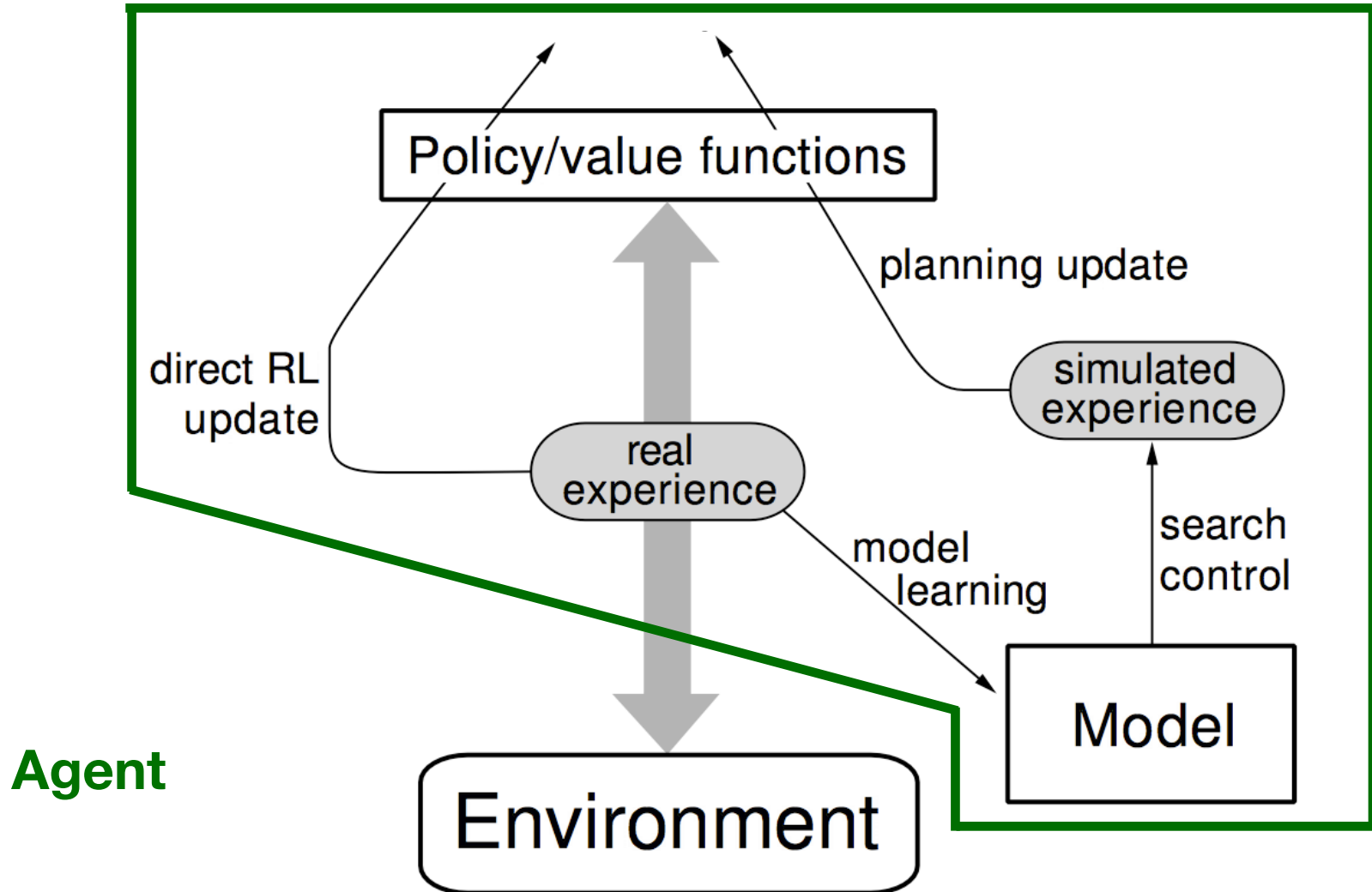# Paths to a policy

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

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

rewards = 0 until goal, when =1
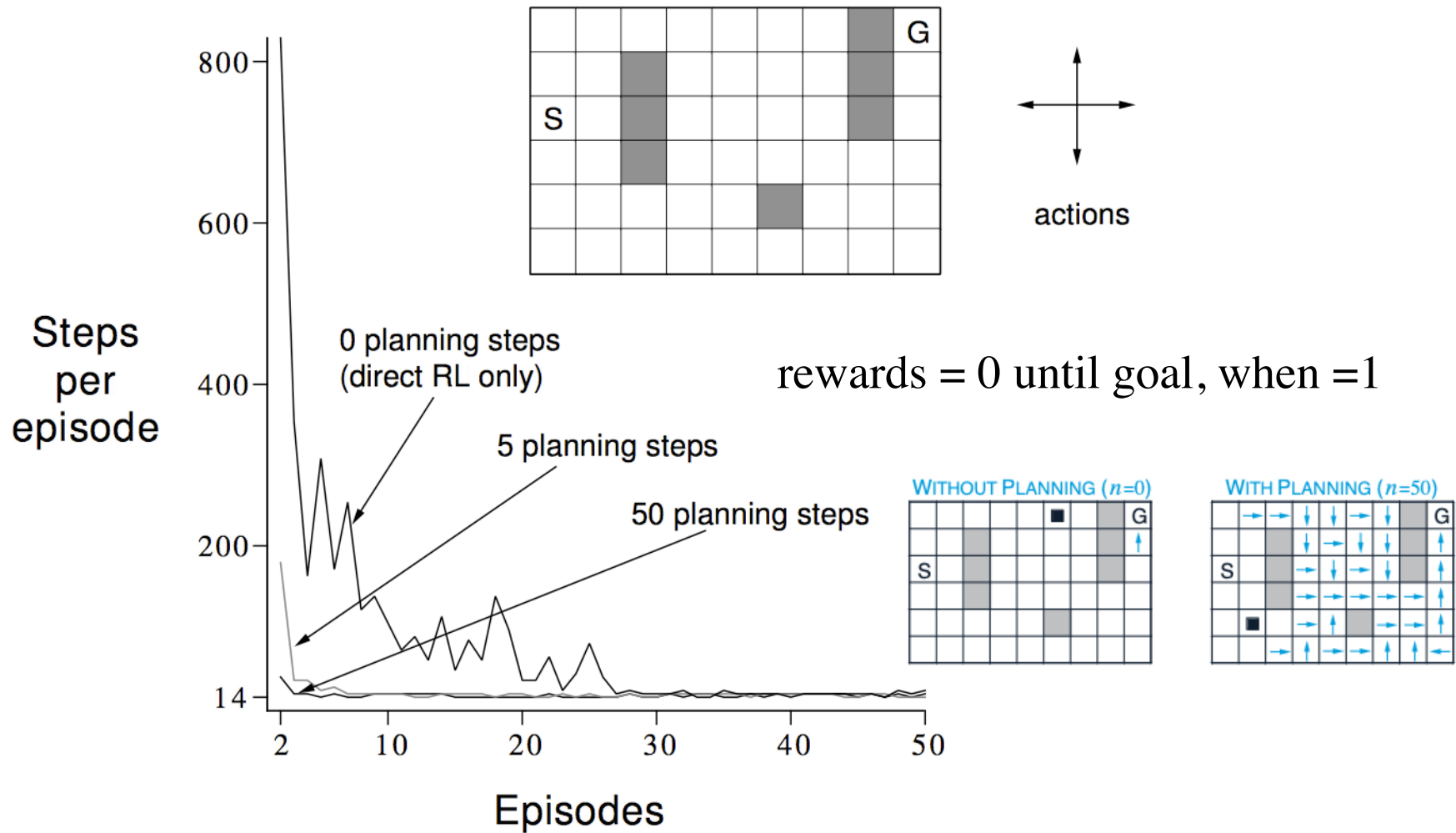
Prioritized Sweeping

- 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 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:
        $S,A \leftarrow first(PQueue)$
        $R,S' \leftarrow Model(S,A)$
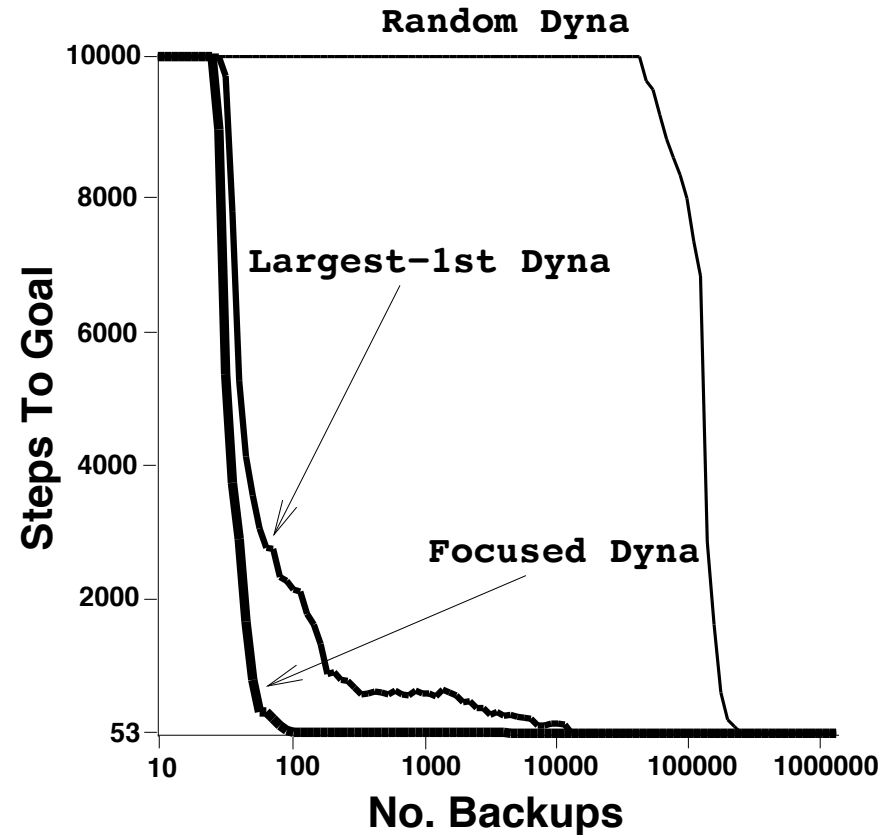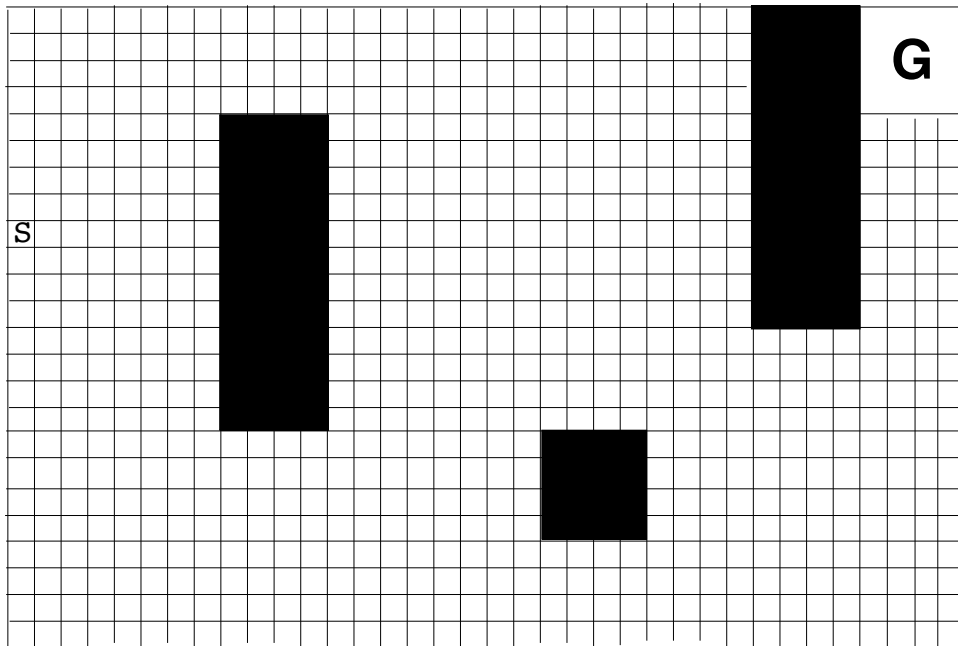 $Q(S,A) \leftarrow Q(S,A) + \alpha[R + \gamma \max_a Q(S',a) - Q(S,A)]$
        Loop for all $\bar{S}, \bar{A}$ predicted to lead to S:
            $\bar{R} \leftarrow$ predicted reward for $\bar{S}, \bar{A}, S$
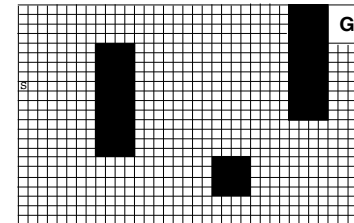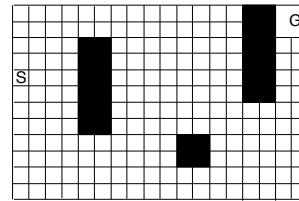 $P \leftarrow |\bar{R} + \gamma \max_a Q(S,a) - Q(\bar{S}, \bar{A})|$.
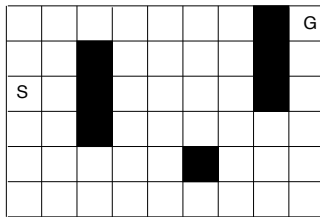   if $P > \theta$ then insert $\bar{S}, \bar{A}$ into $PQueue$ with priority P

Large maze and random search control

more finely partitioned version of the maze of Figure 1.

(Peng and Williams, 1993)

Figure 8: Performance on the maze of Figur

Prioritized Sweeping vs. Dyna-Q

Figure 1: A maze navigation task.   Figure 5: A more finely

2   An Illustrative Task

Suppose a learning agent is placed in the discretized 2-dimensional maze shown
1. Shaded cells in this maze represent barriers, and the agent can occupy any
within this maze and can move about by choosing one of four actions at each disc
tick. Each of these actions has the effect of moving the agent to an adjacent c
of the four compass directions, north, east, south, or west, except that any ac
would ostensibly move the agent into a barrier cell or outside the maze has t
effect of keeping the agent at its current location. The agent initially has no k
of the effect of its actions on what state (i.e., cell) it will occupy next, although
knows its current state.

We also assume that this environment provides rewards to the agent and that th
structure is also initially unknown to the agent. For example, the agent may g
reward when it enters into the state marked G in the maze. Loosely st
objective is for the learning agent to discover a policy, or assignment of choice

Backups until optimal solution

Backups until optimal solution

Both use *n*=5 backups per environmental interaction

Figure 6: An even more fi

Gridw

R. S. Sutton and A. G. Barto: Reinforcement Learning: An Introduction   28
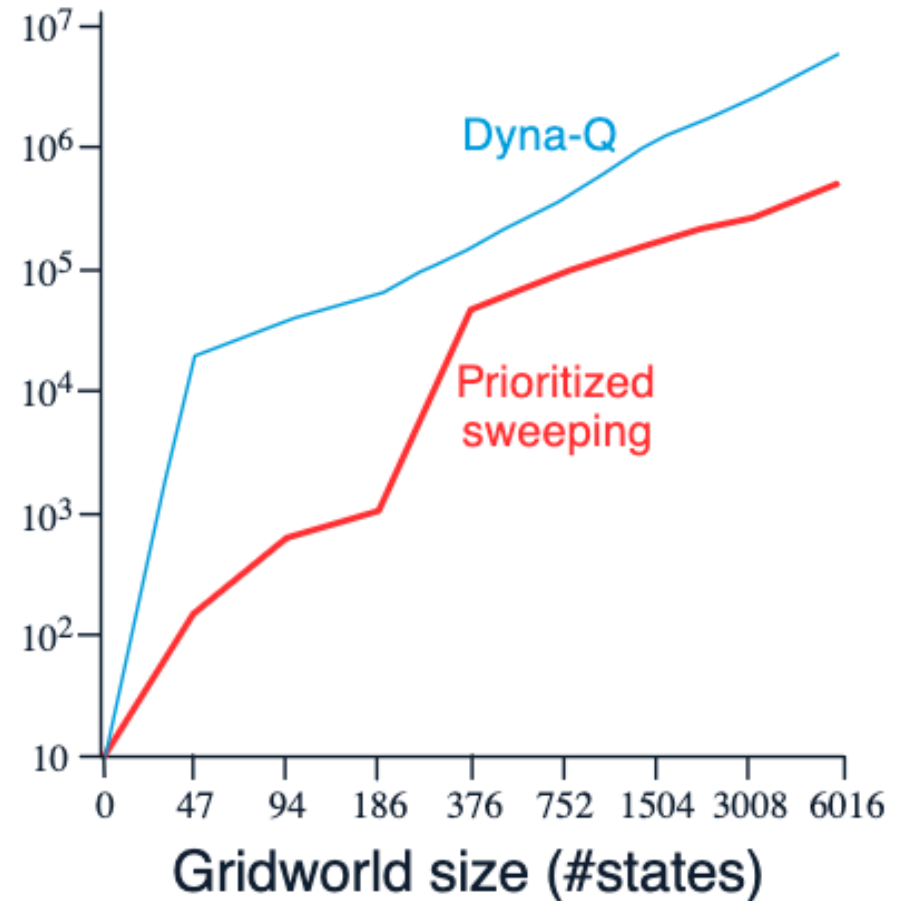
Special case: Linear Expectation Models

$$F_a \phi = \phi'$$
$$b_a^\top \phi = r$$

start state vector — ϕ

transition model — F_a

$=$ ϕ' — predicted next-state vector

reward model — b_a — r — predicted reward
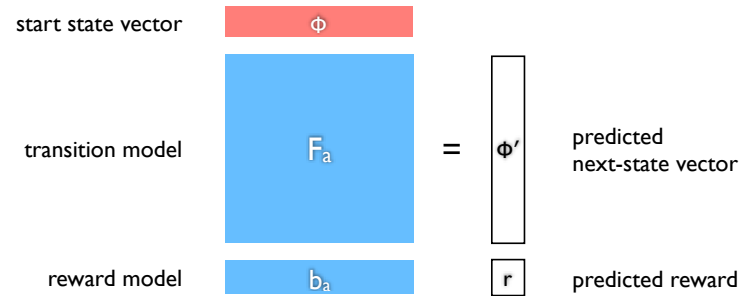
- states are represented by feature vectors

$$s \longrightarrow \phi_s \qquad s_t \longrightarrow \phi_t \qquad \in \Re^n$$

- 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
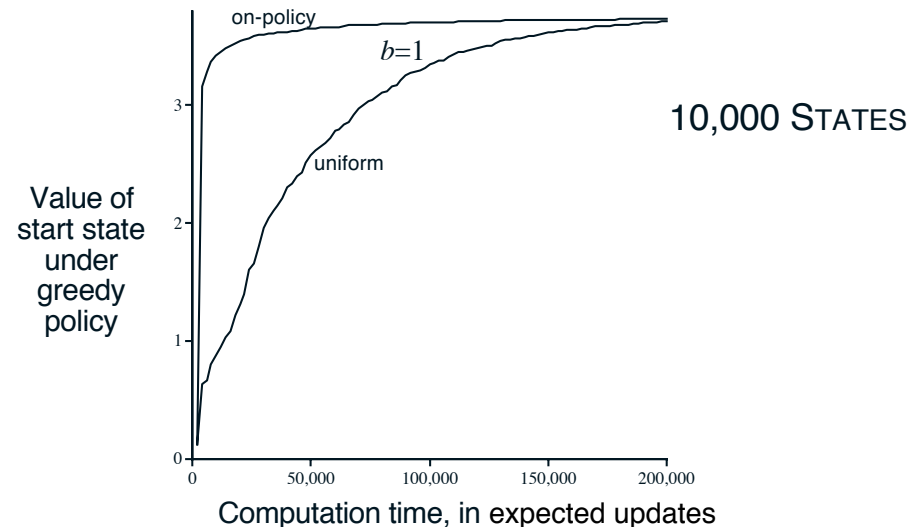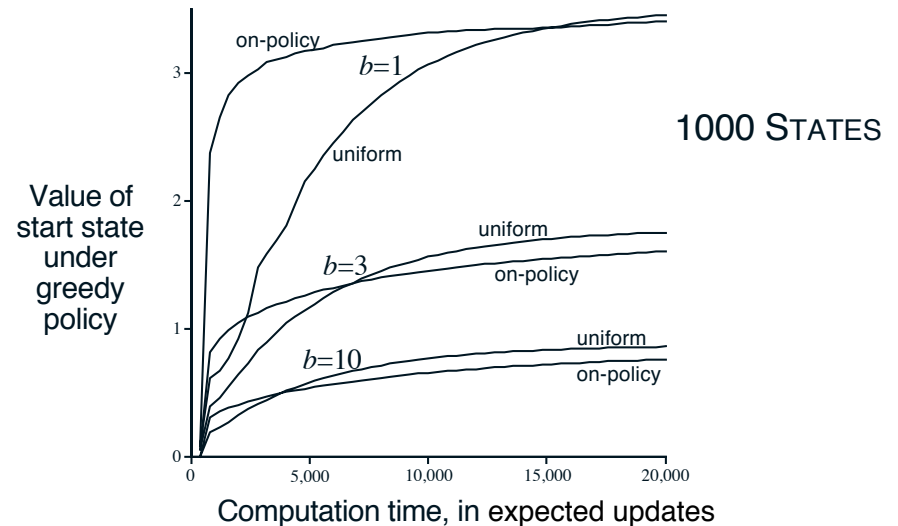
Trajectory Sampling

- 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:
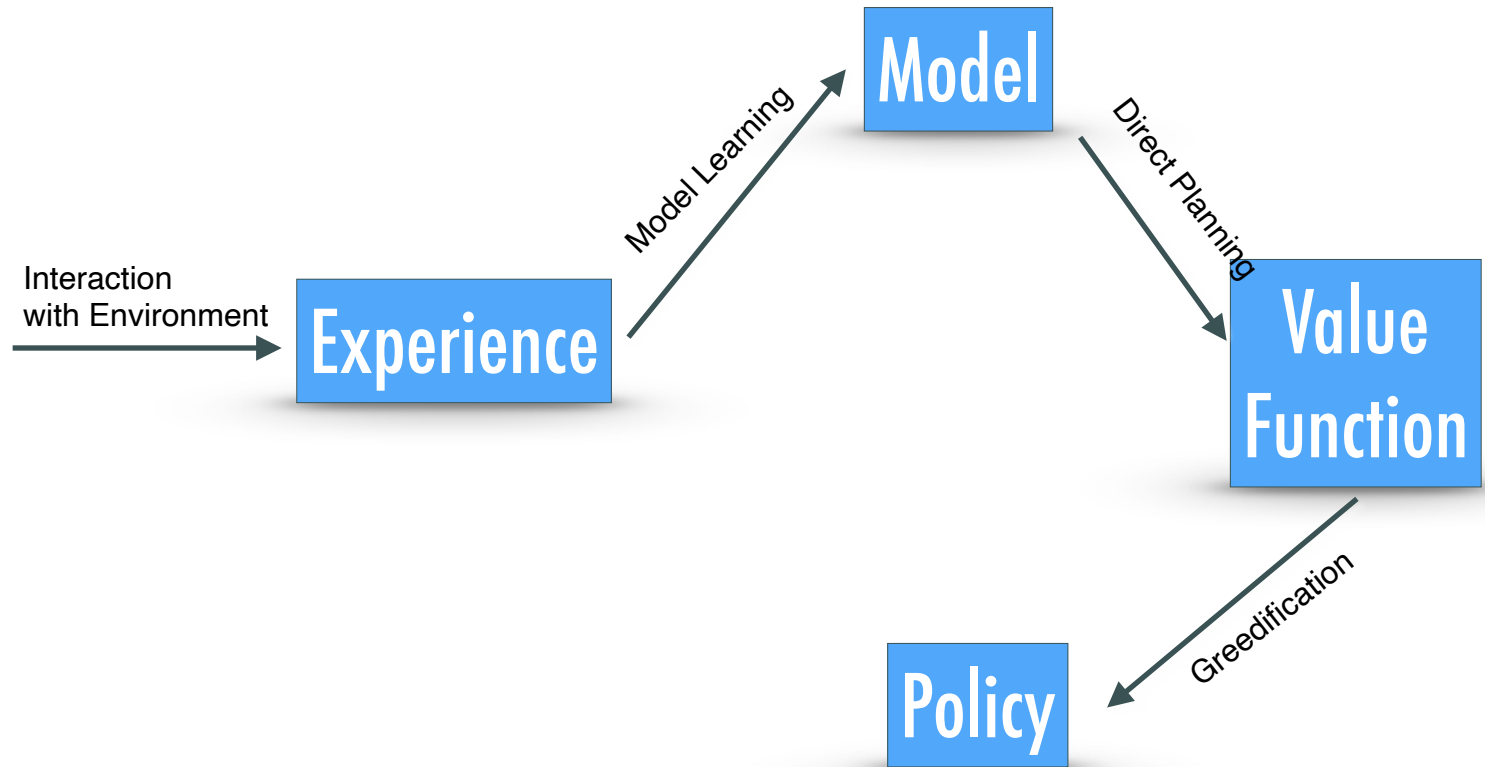 can cause vast uninteresting parts of the state space to be ignored:

Initial states

Reachable under optimal control

Irrelevant states
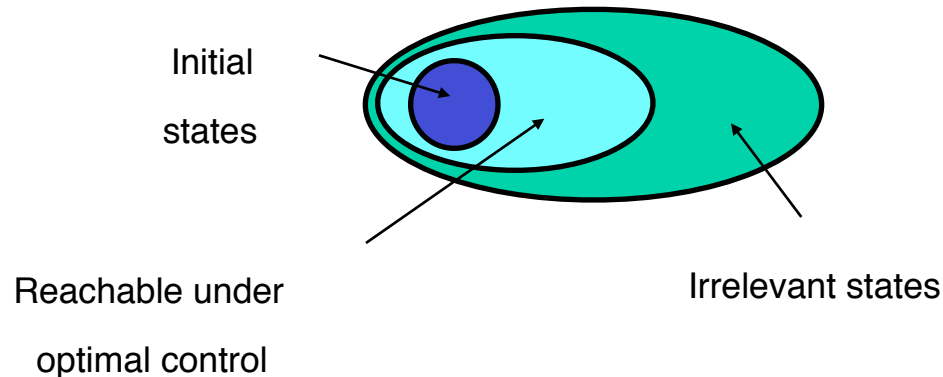
Trajectory Sampling Experiment

- 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

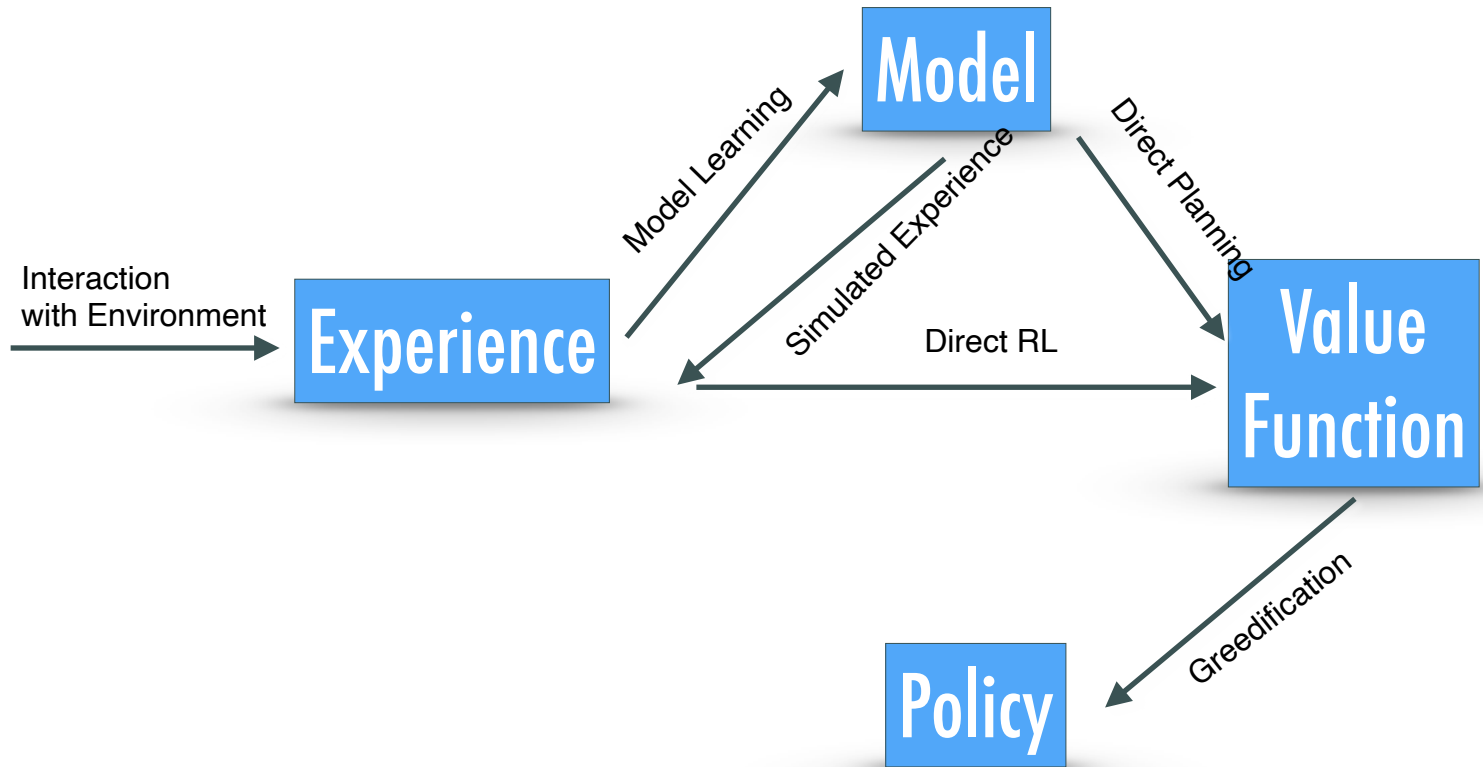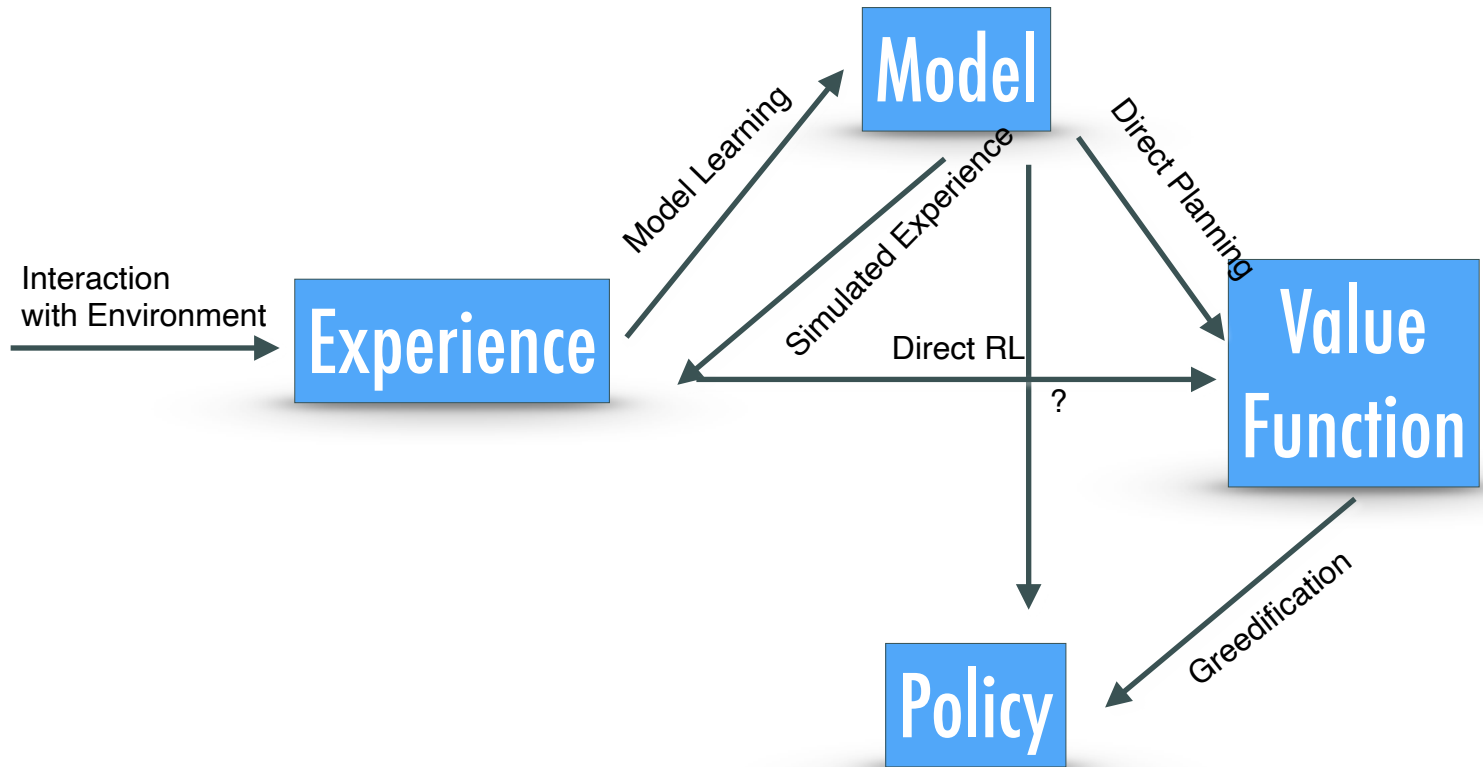- expected reward on each transition selected from mean 0 variance 1 Gaussian

1000 STATES

Value of start state under greedy policy

on-policy
b=1
uniform
uniform
b=3
on-policy
uniform
b=10
on-policy

Computation time, in expected updates

10,000 STATES

Value of start state under greedy policy

on-policy
b=1
uniform

Computation time, in expected updates

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:

Conclusion

Interaction with Environment → **Experience**

Experience — Model Learning → **Model**

Value Function

Policy

Conclusion

Conclusion

Conclusion

Conclusion

- Model-Based RL is most Useful:

  - When Trajectories are more « expensive » than « thinking/compute ».

  - When environment or reward can change

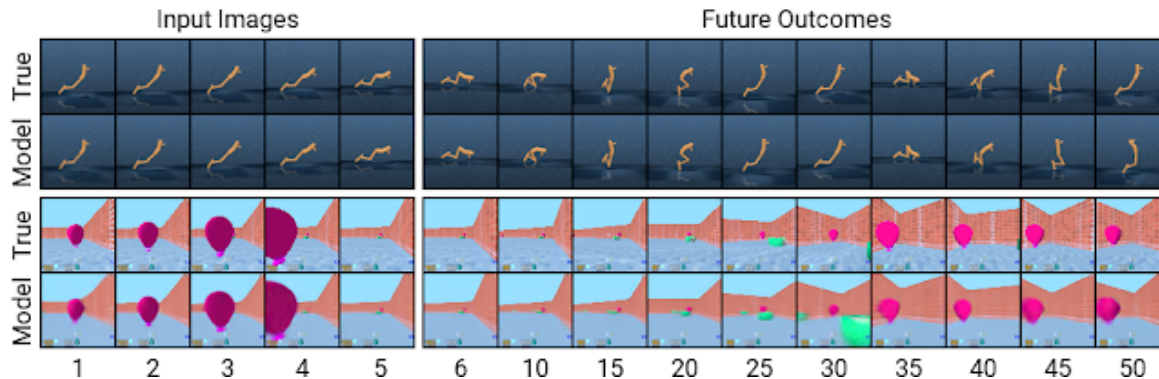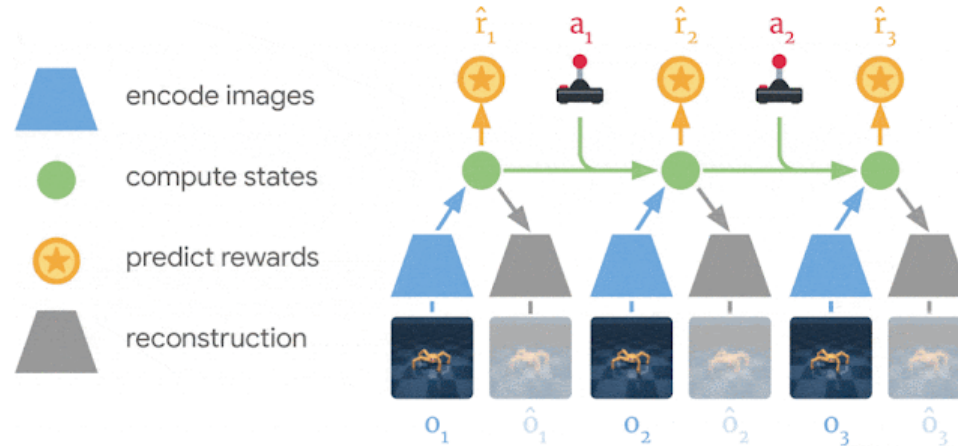# Using Approximate Models: PlaNet (Hafner et al, 2019)
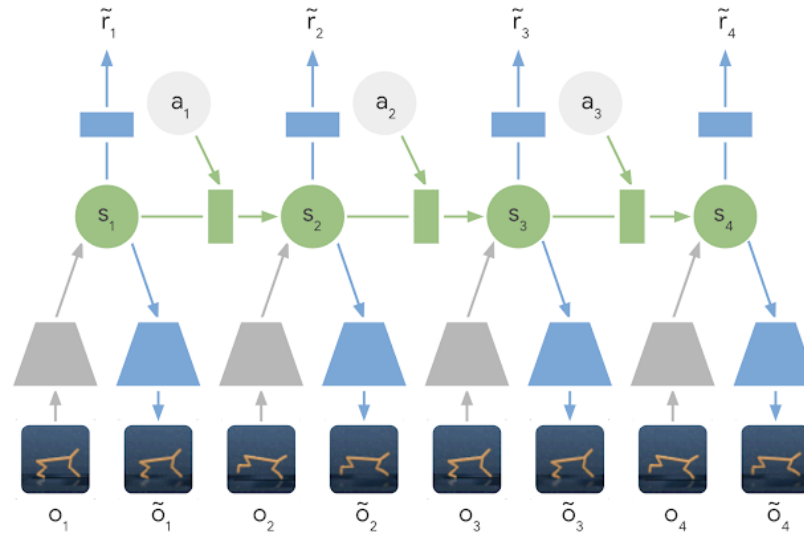
- Building on world models work by Ha and Schmidhuber (2017)
- Learn a model that tries to fit the observations (using a loss function)
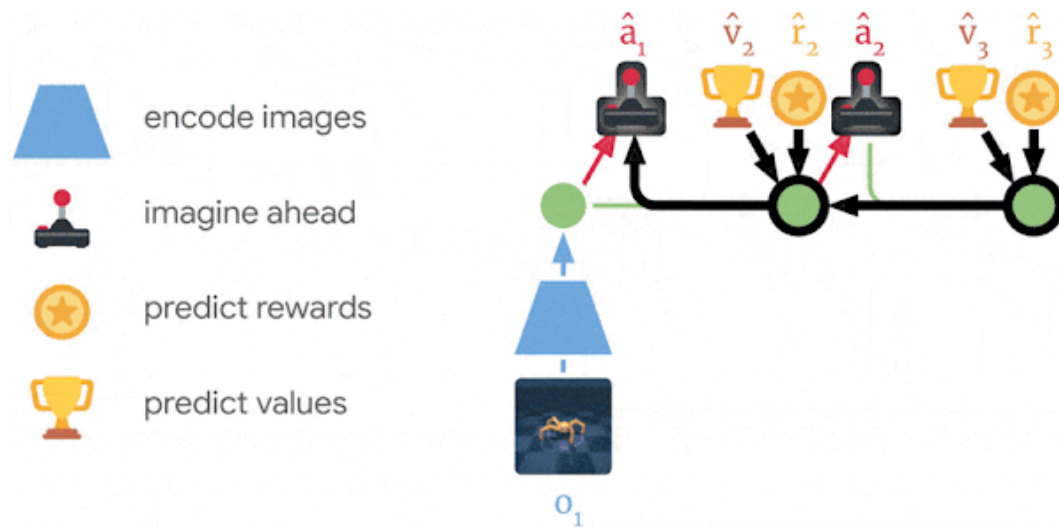
PlaNet Planning Process
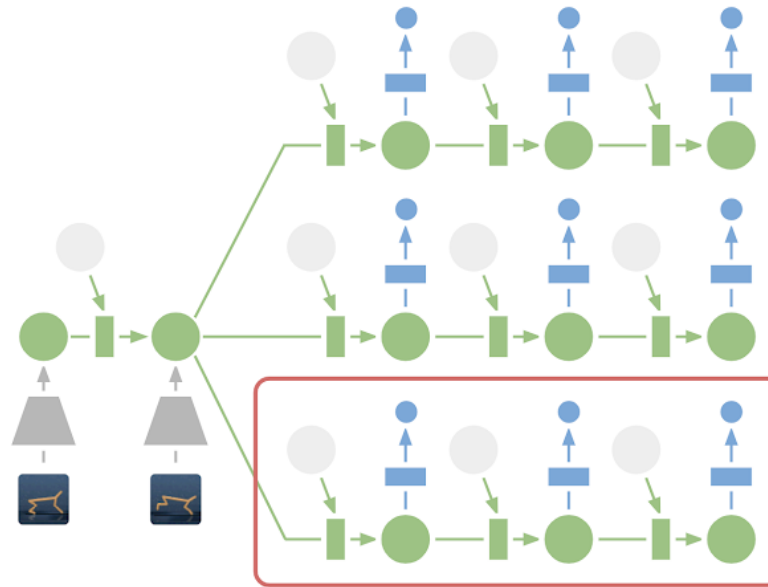
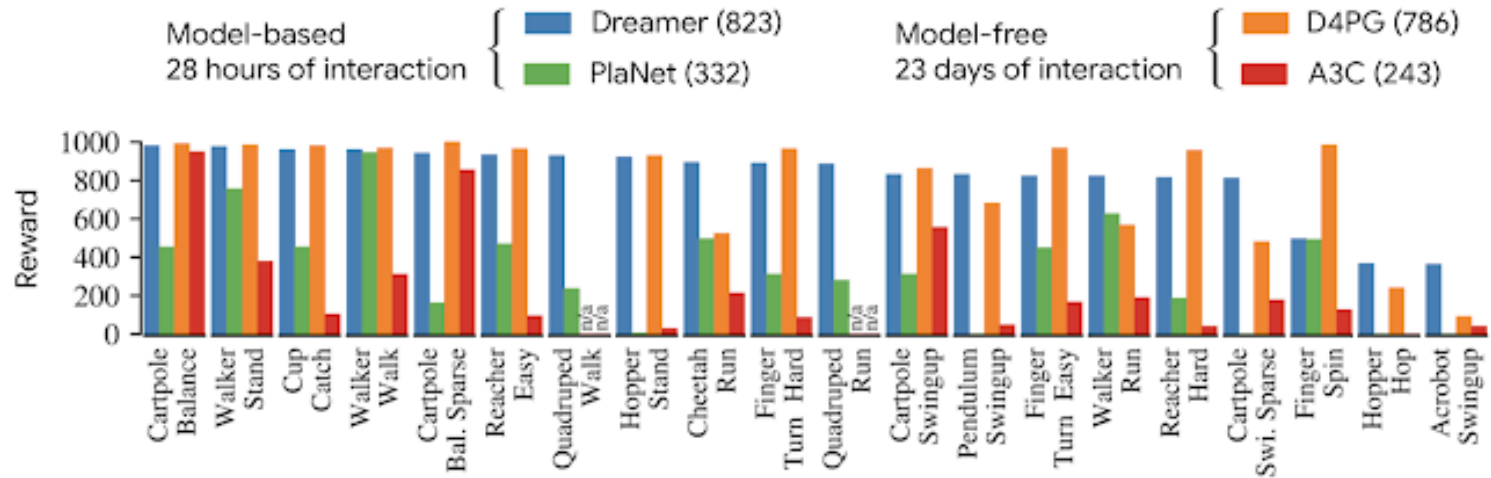- At planning time, only abstract states are generated

Dreamer (Hafner et al, 2020)

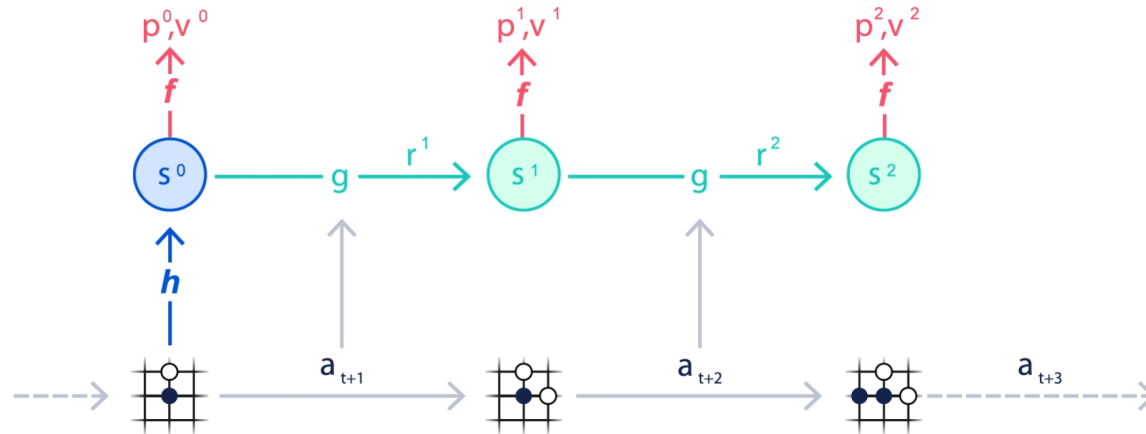# Value Propagation in Dreamer

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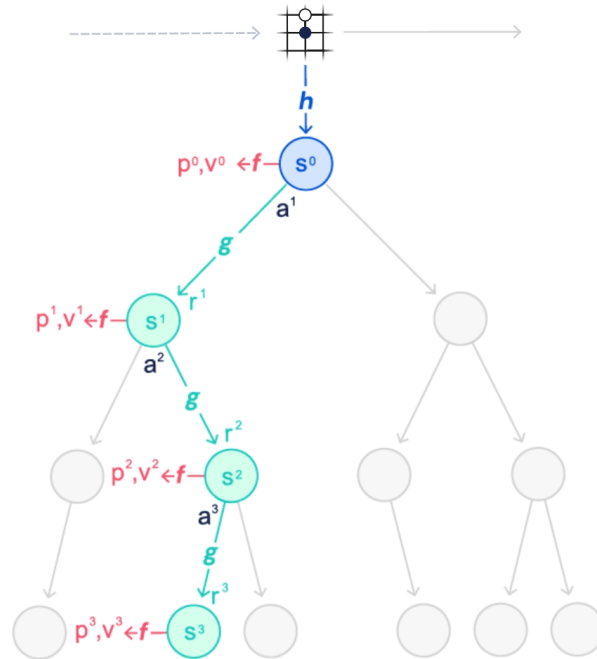
- Model is rolled forward in Monte Carlo Tree Search-style

MuZero Results

Chess · Shogi · Go · Atari
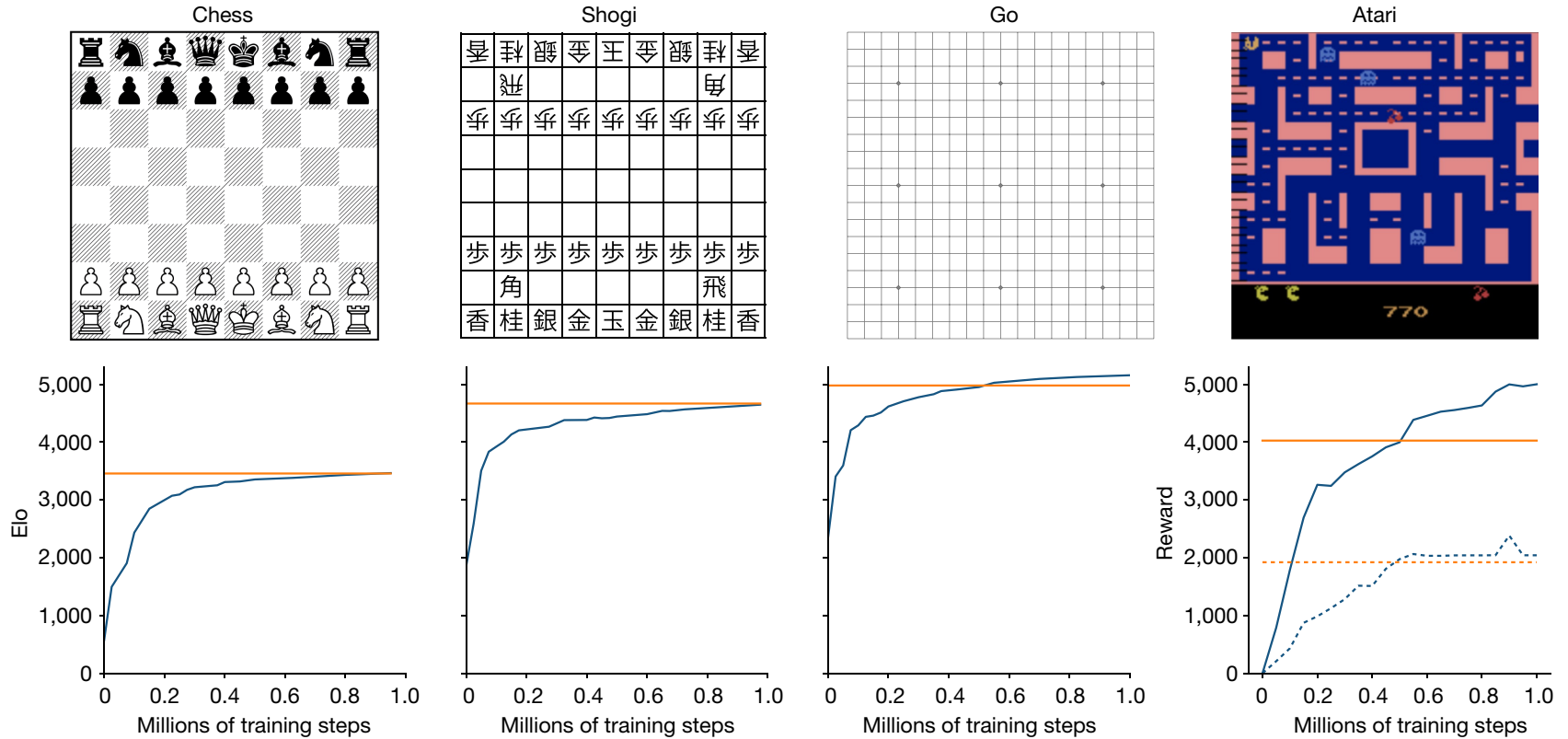
MuZero outperforms R2D2 (best model-free agent at the time)

$$v_t^k \approx \mathbb{E}[u_{t+k+1} + \gamma u_{t+k+2} + ... | o_1, \, ..., o_t, \, a_{t+1}, ..., \, a_{t+k}]$$
$$r_t^k \approx u_{t+k}$$