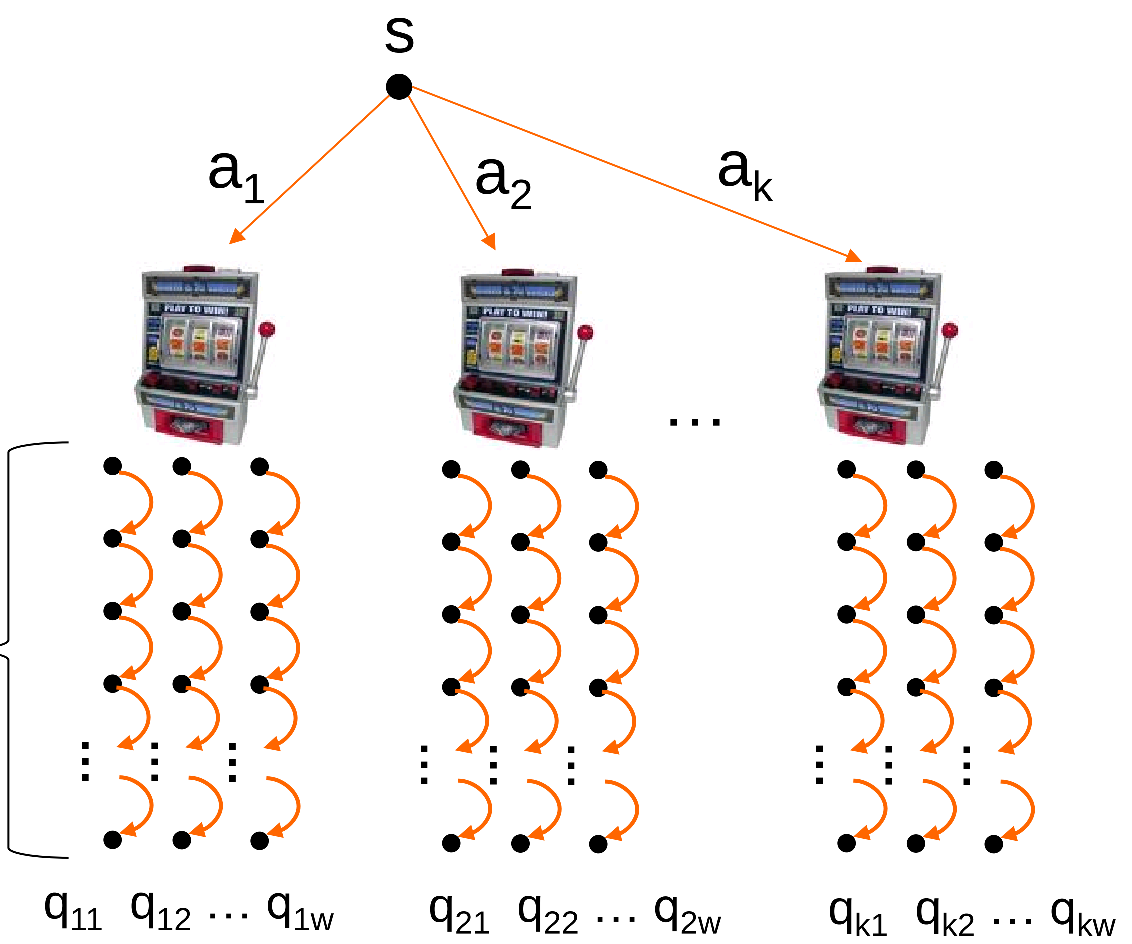
Classifier-Based Approximate Policy Iteration

Alan Fern

Uniform Policy Rollout Algorithm

Rollout[π,h,w](s)

1. For each a_i run SimQ(s,a_i,π,h) **w** times
2. Return action with best average of SimQ results



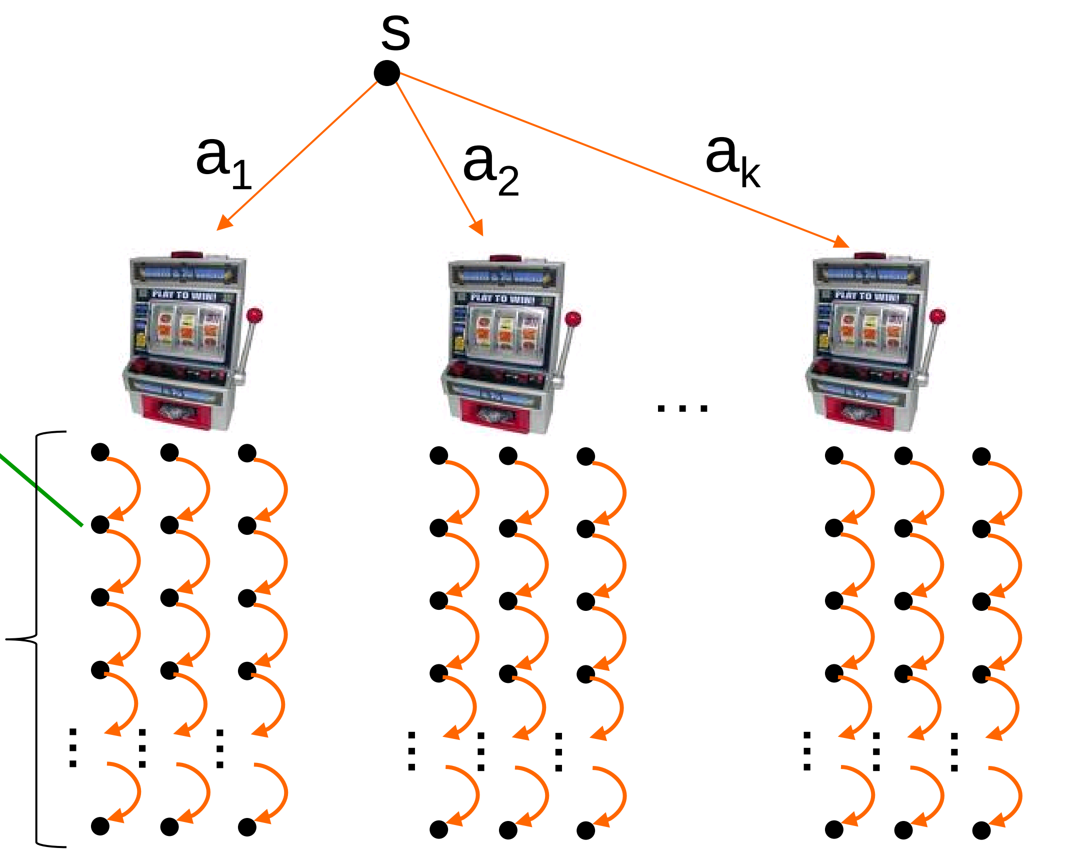
s
a_1
a_2
a_k
…
q_{11} q_{12} … q_{1w}
q_{21} q_{22} … q_{2w}
q_{k1} q_{k2} … q_{kw}

SimQ(s,a_i,π,h) trajectories

Each simulates taking action a_i then following π for h-1 steps.

Samples of SimQ(s,a_i,π,h)

Multi-Stage Rollout


s
a_1
a_2
a_k
…
Each step requires khw simulator calls for Rollout policy
Trajectories of SimQ(s,a_i,Rollout[π,h,w],h)

- Two stage: compute rollout policy of “rollout policy of π”
- Requires (khw)^2 calls to the simulator for 2 stages
- In general exponential in the number of stages

Example: Rollout for Solitaire [Yan et al. NIPS'04]

Player	Success Rate	Time/Game
Human Expert	36.6%	20 min
(naïve) Base Policy	13.05%	0.021 sec
1 rollout	31.20%	0.67 sec
2 rollout	47.6%	7.13 sec
3 rollout	56.83%	1.5 min
4 rollout	60.51%	18 min
5 rollout	70.20%	1 hour 45 min

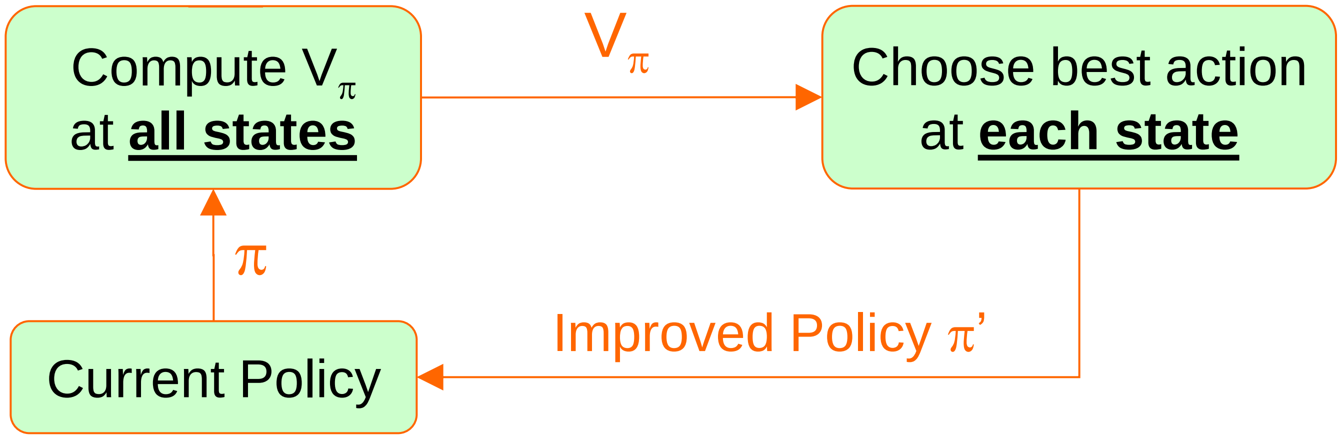
- Multiple levels of rollout can payoff but is expensive

Can we somehow get the benefit of multiple levels without the complexity?

Approximate Policy Iteration: Main Idea

- Nested rollout is expensive because the "base policies" (i.e. nested rollouts themselves) are expensive
- Suppose that we could approximate a level-one rollout policy with a very fast function (e.g. O(1) time)
- Then we could approximate a level-two rollout policy while paying only the cost of level-one rollout
- Repeatedly applying this idea leads to approximate policy iteration

Return to Policy Iteration

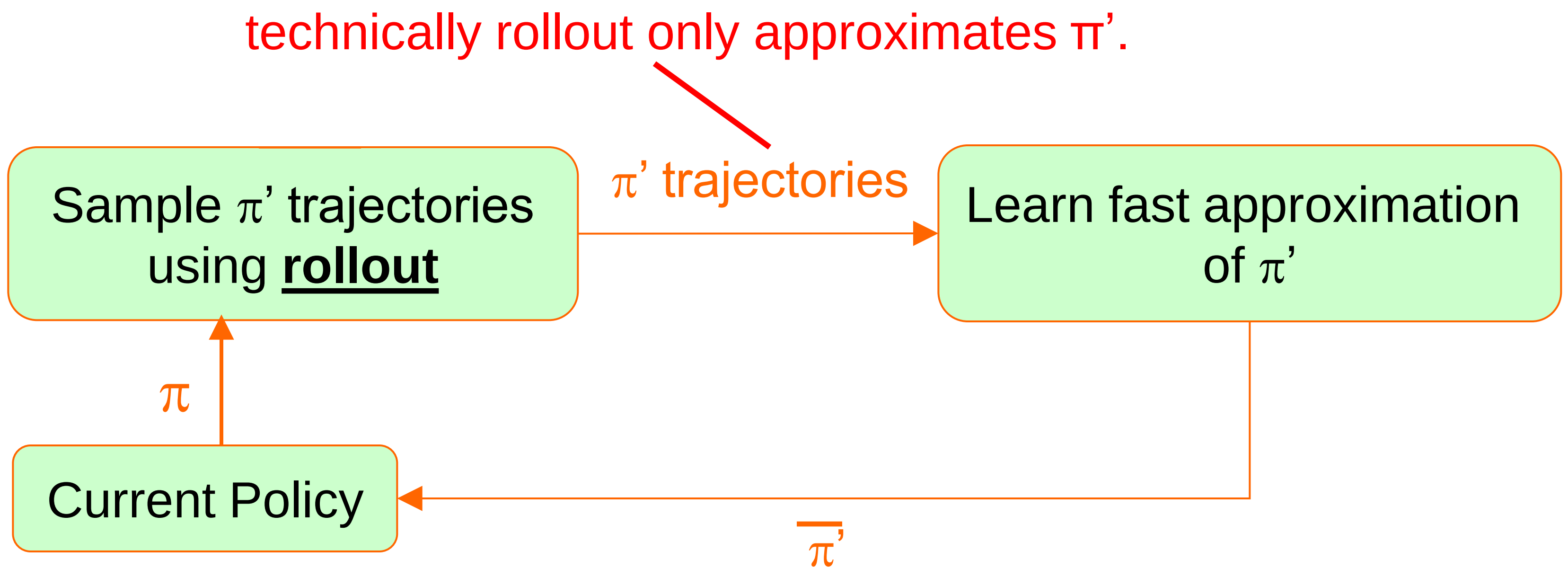


Compute V_π at **all states**
V_π
Choose best action at **each state**
π
Improved Policy π'
Current Policy

Approximate policy iteration:

- Only computes values and improved action at some states.
- Uses those to infer a fast, compact policy over all states.

Approximate Policy Iteration

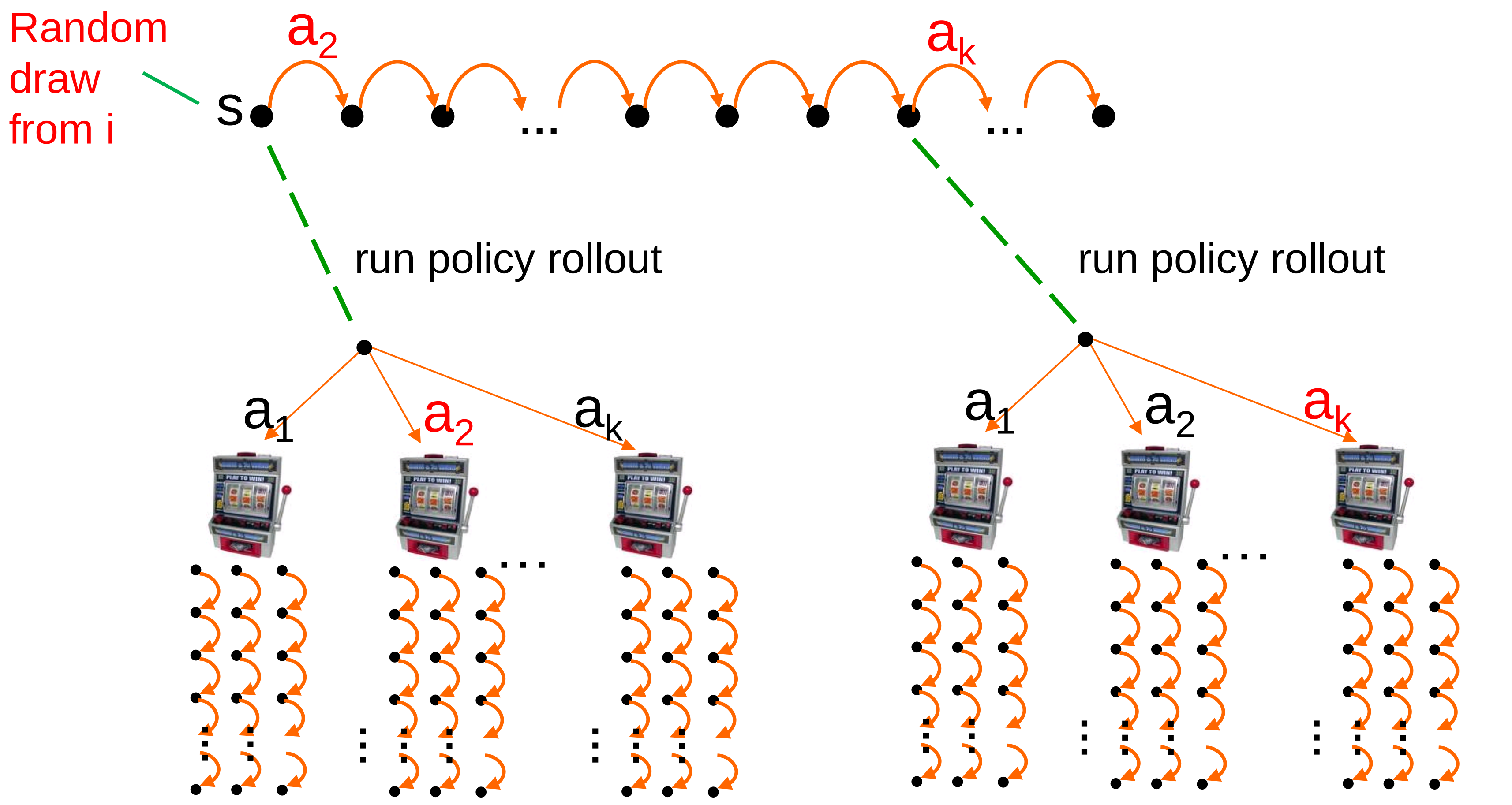

technically rollout only approximates π'.
Sample π' trajectories using **rollout**
π' trajectories
Learn fast approximation of π'
π
Current Policy
$\overline{\pi'}$

1. Generate trajectories of rollout policy (starting state of each trajectory is drawn from initial state distribution I)
2. “Learn a fast approximation” of rollout policy
3. Loop to step 1 using the learned policy as the base policy

What do we mean by generate trajectories?

Generating Rollout Trajectories

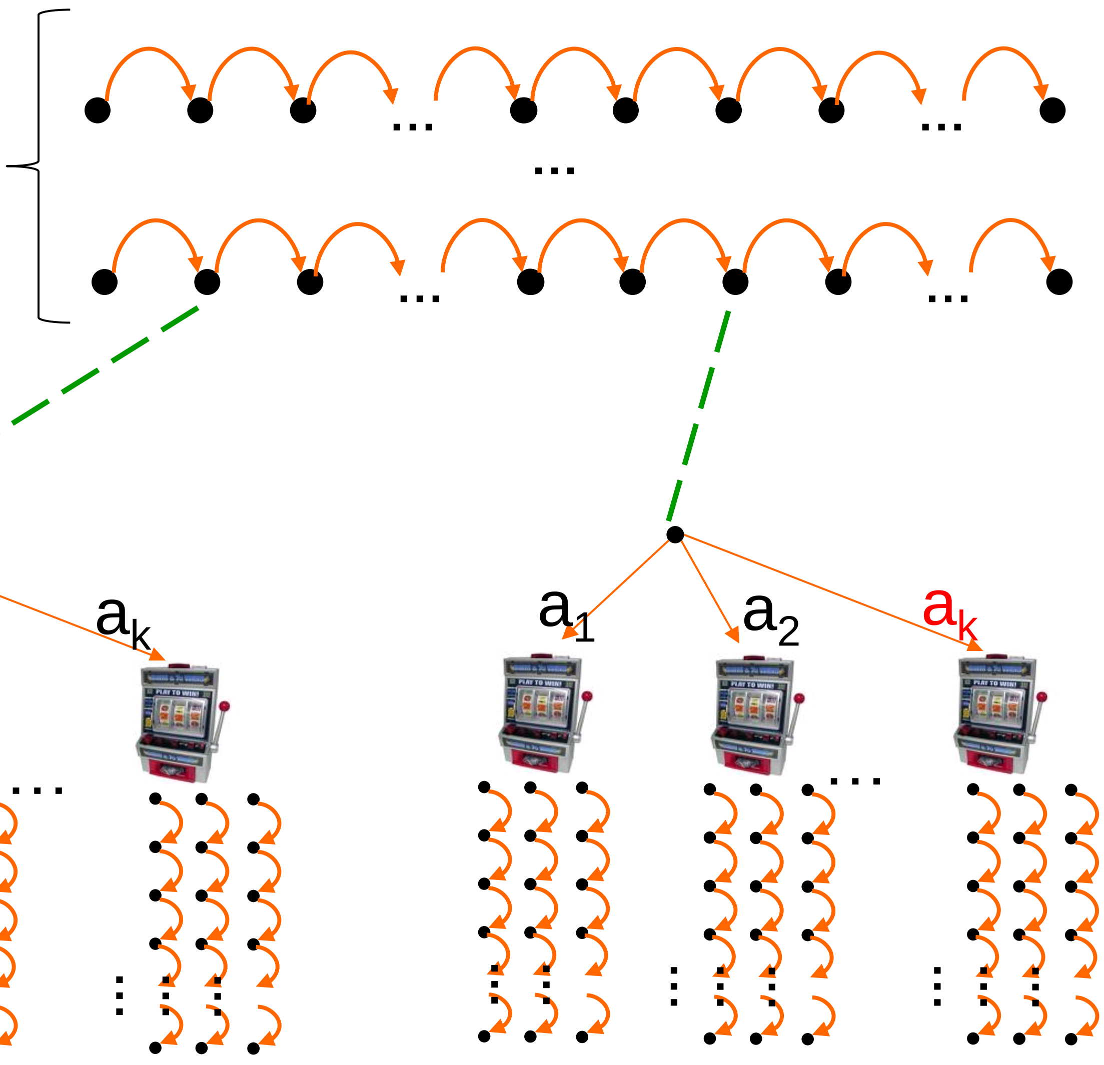
Get trajectories of current rollout policy from an initial state



Generating Rollout Trajectories

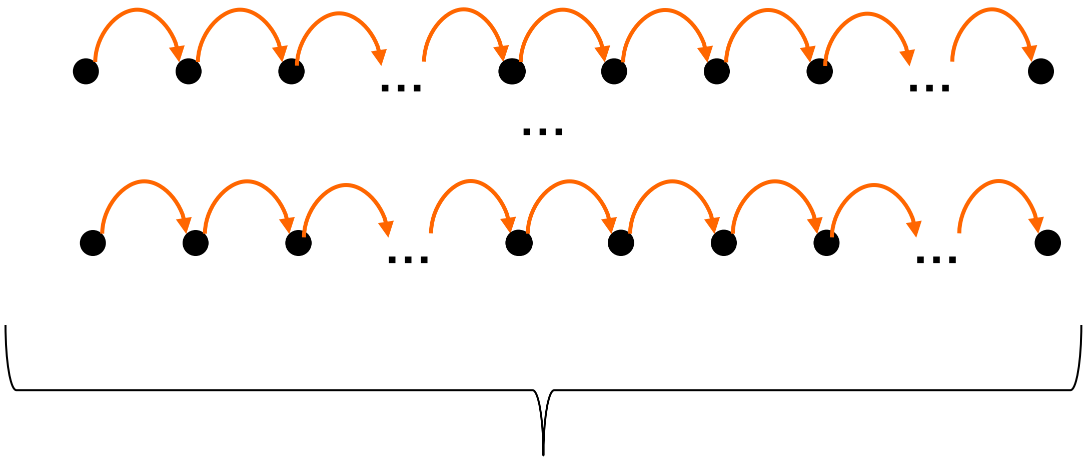
Get trajectories of current rollout policy from an initial state

Multiple trajectories differ since initial state and transitions are stochastic



Generating Rollout Trajectories

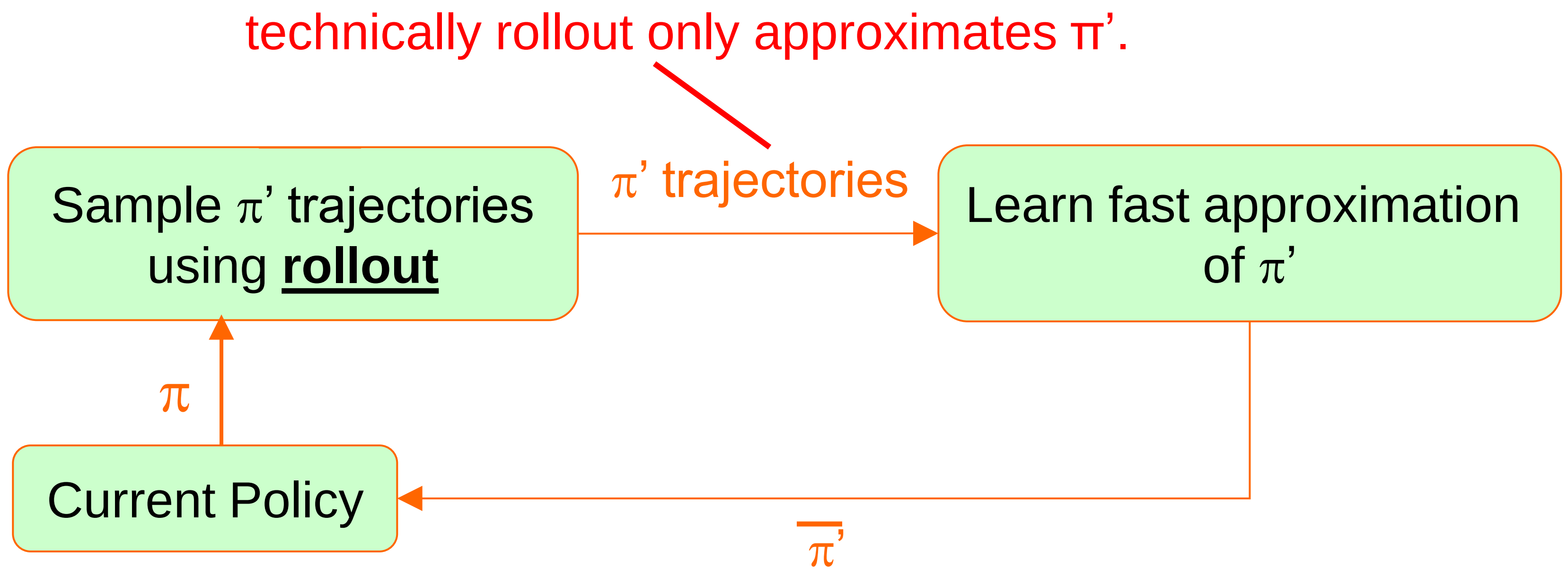
Get trajectories of current rollout policy from an initial state



Results in a set of state-action pairs giving the action selected by “improved policy” in states that it visits.

$$\{(s_1,a_1),\ (s_2,a_2),\ldots,(s_n,a_n)\}$$

Approximate Policy Iteration


technically rollout only approximates π'.
Sample π' trajectories using **rollout**
π' trajectories
Learn fast approximation of π'
π
Current Policy
$\overline{\pi}$'

1. Generate trajectories of rollout policy (starting state of each trajectory is drawn from initial state distribution I)
2. "Learn a fast approximation" of rollout policy
3. Loop to step 1 using the learned policy as the base policy

What do we mean by "learn an approximation"?

Aside: Classifier Learning

- A **classifier** is a function that labels inputs with class labels.
- "Learning" classifiers from training data is a well studied problem (decision trees, support vector machines, neural networks, etc).

$$\begin{tikzcd}
\text{Training Data} \\
\{(x_1,c_1),(x_2,c_2),\ldots,(x_n,c_n)\} \arrow[d] \\
\boxed{\begin{array}{l}\text{Learning}\\ \text{Algorithm}\end{array}} \arrow[d] \\
\text{Classifier} \\
H : X \rightarrow C
\end{tikzcd}$$

Example problem:

x_i - image of a face

$c_i \in$ {male, female}

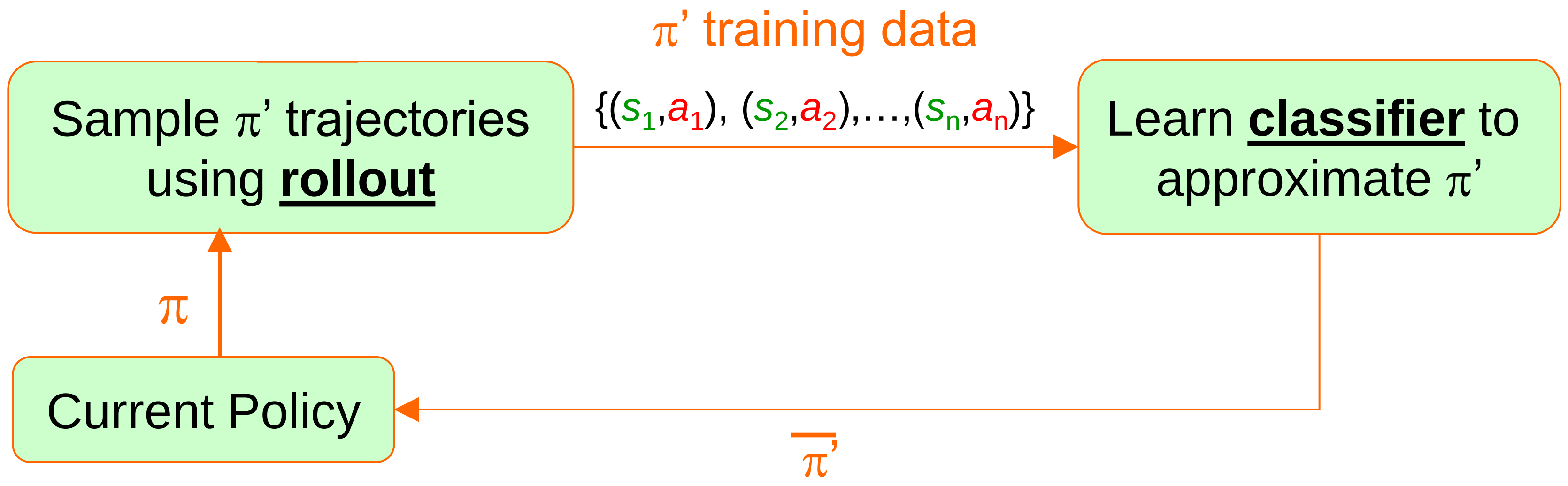
Aside: Control Policies are Classifiers

A **control policy** maps states and goals to actions.

π : states $\rightarrow$ actions

$$\begin{tikzcd} \begin{array}{c}\text{Training Data}\\ \{(s_1,a_1),\ (s_2,a_2),\ldots,(s_n,a_n)\}\end{array} \arrow[d] \\ \boxed{\begin{array}{l}\text{Learning}\\ \text{Algorithm}\end{array}} \arrow[d] \\ \begin{array}{c}\text{Classifier/Policy}\\ \pi : \text{states} \rightarrow \text{actions}\end{array} \end{tikzcd}$$

Approximate Policy Iteration

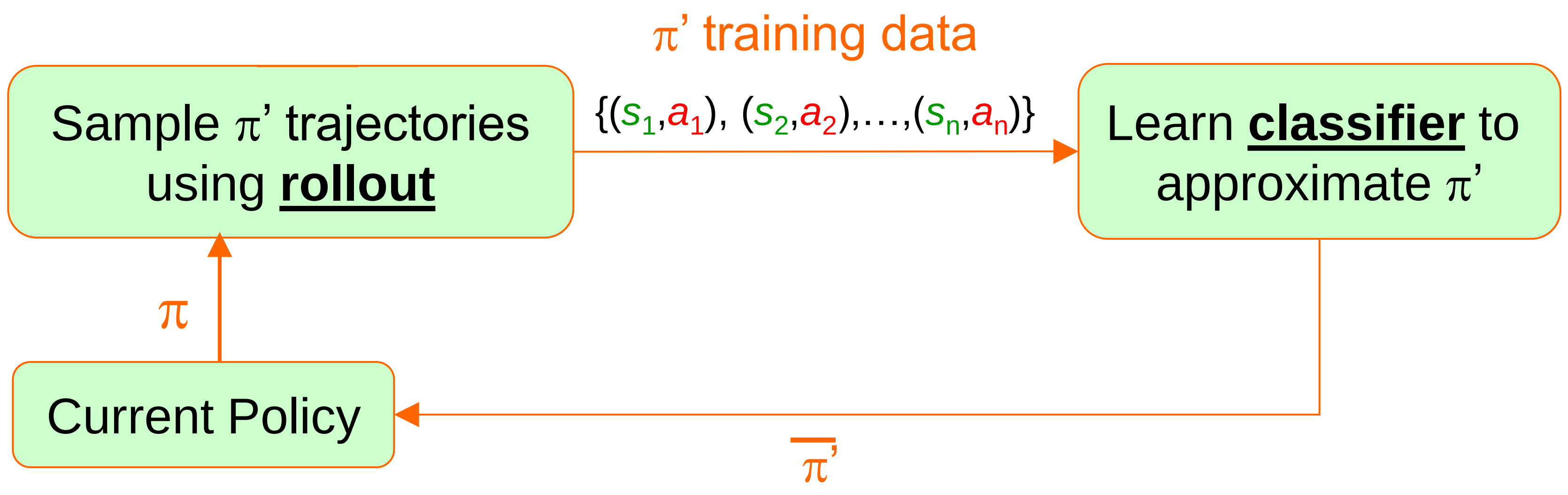

π' training data
Sample π' trajectories using rollout
{(s1,a1), (s2,a2),…,(sn,an)}
Learn classifier to approximate π'
π
Current Policy
π̄'

1. Generate trajectories of rollout policy
   Results in training set of state-action pairs along trajectories

$$T = \{(s_1,a_1), (s_2,a_2),\ldots,(s_n,a_n)\}$$

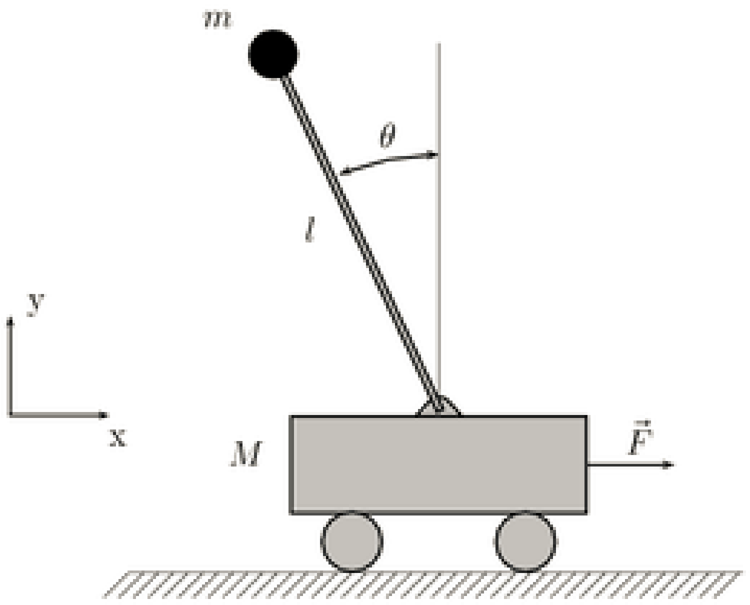
2. Learn a classifier based on *T* to approximate rollout policy
3. Loop to step 1 using the learned policy as the base policy

Approximate Policy Iteration


π' training data
Sample π' trajectories using **rollout**
{(s_1,a_1), (s_2,a_2),…,(s_n,a_n)}
Learn **classifier** to approximate π'
π
Current Policy
$\overline{\pi'}$

- The hope is that the learned classifier will capture the general structure of improved policy from examples
- Want classifier to quickly select correct actions in states outside of training data (classifier should generalize)
- Approach allows us to leverage large amounts of work in machine learning

API for Inverted Pendulum

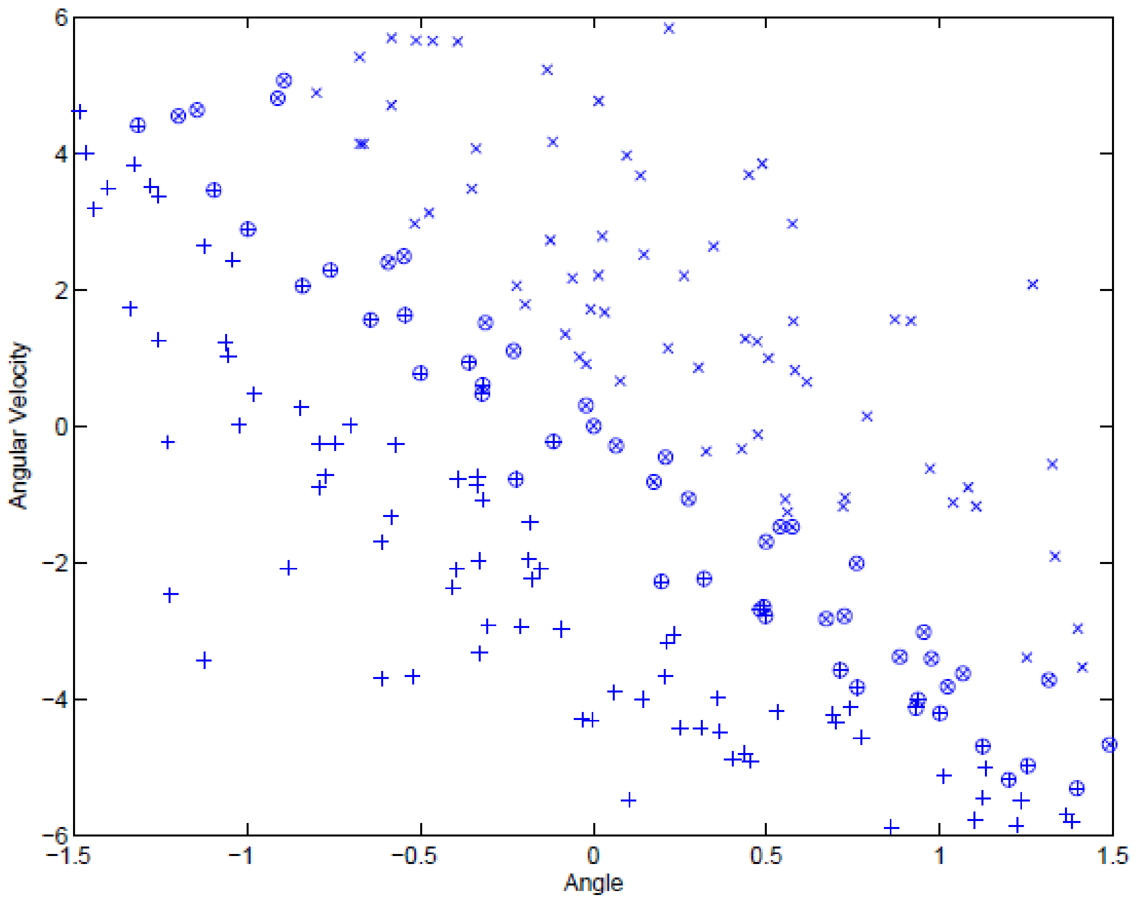


Consider the problem of balancing a pole by applying either a positive or negative force to the cart.

The state space is described by the velocity of the cart and angle of the pendulum.

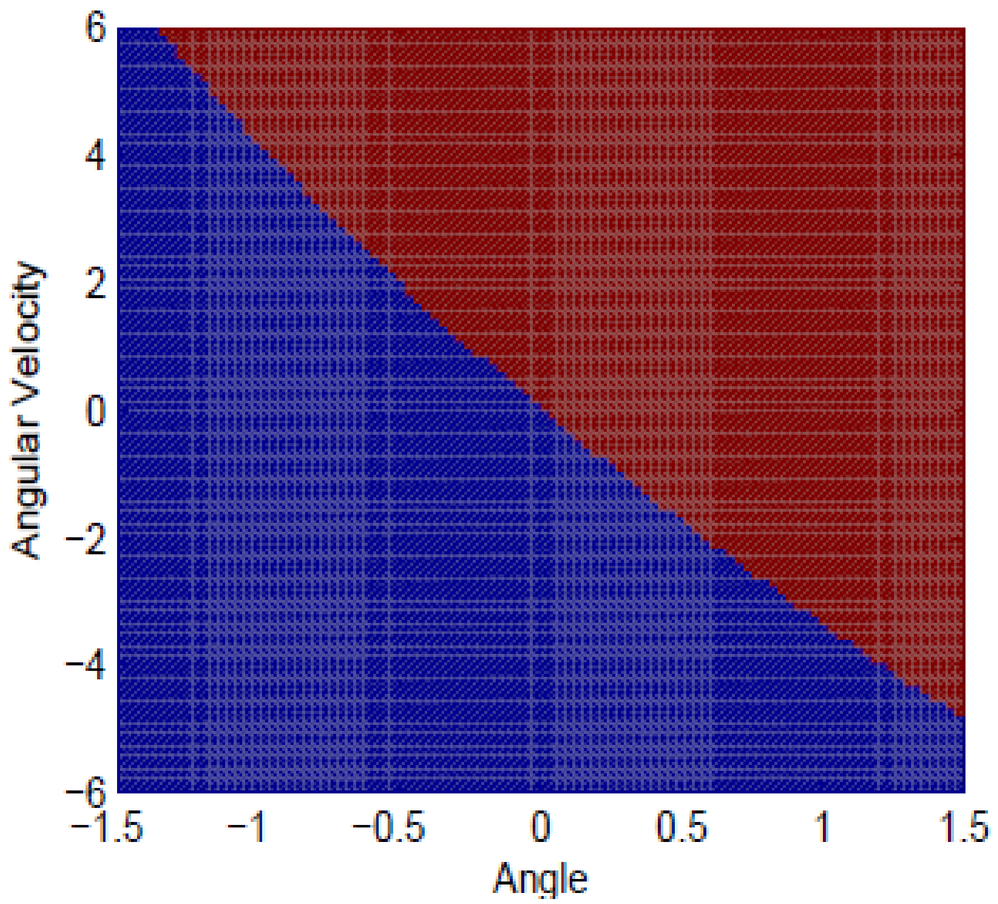
There is noise in the force that is applied, so problem is stochastic.

Experimental Results



A data set from an API iteration. + is positive action, x is negative (ignore the circles in the figure)

Experimental Results

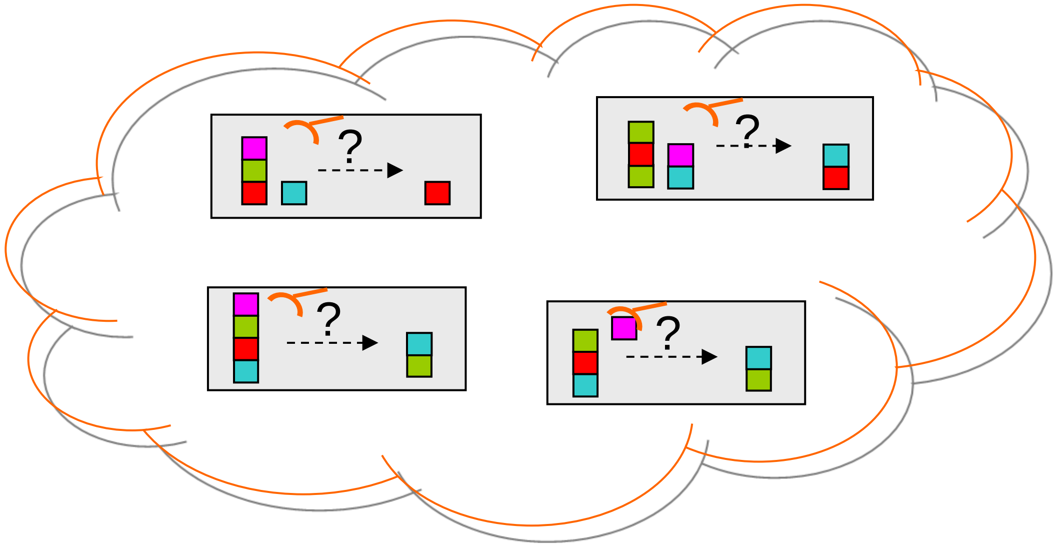


Support vector machine used as classifier.
(take CS534 for details)

Maps any state to + or –

Learned classifier/policy after 2 iterations: (near optimal)
blue = positive, red = negative

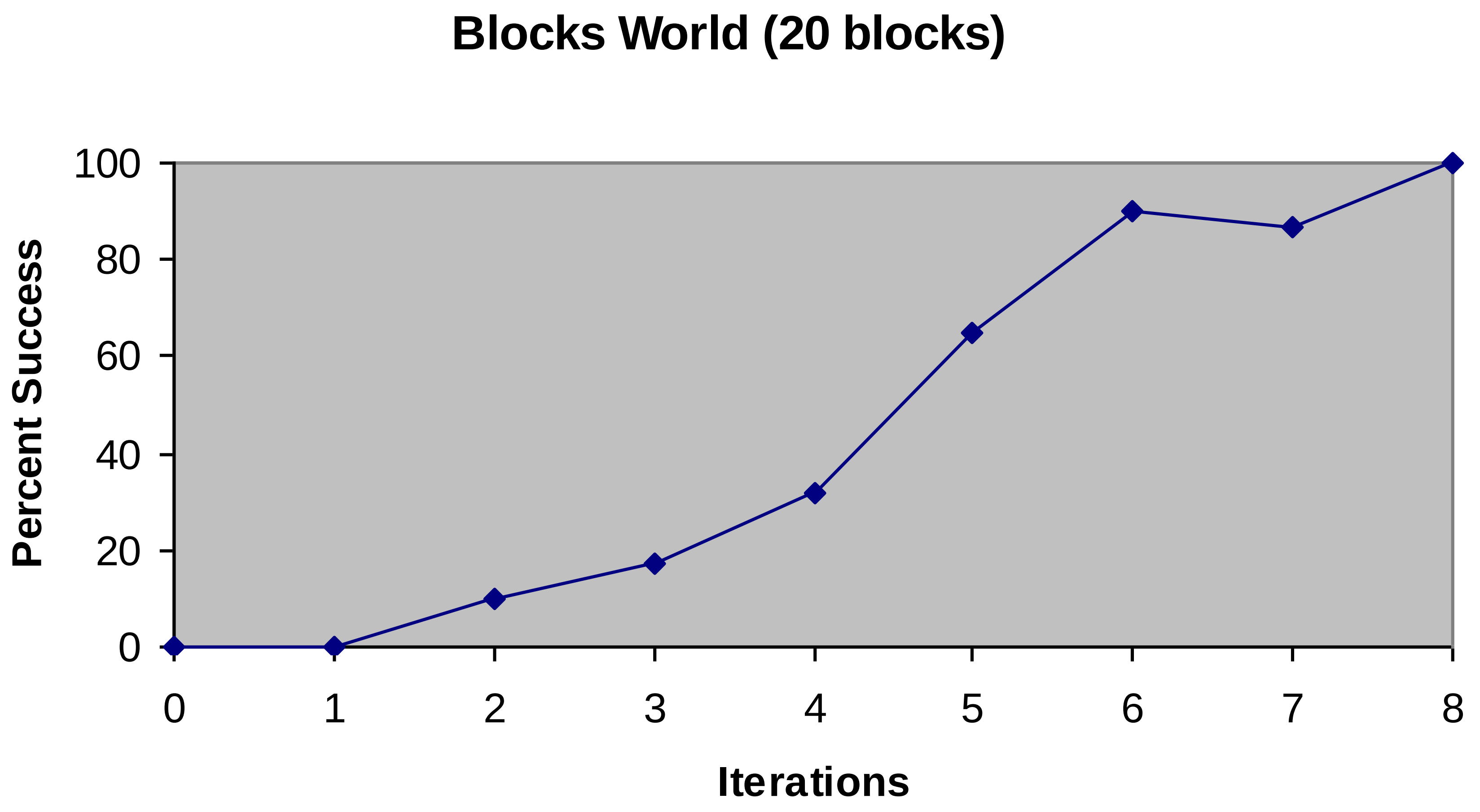
API for Stacking Blocks



Consider the problem of form a goal configuration of blocks/crates/etc. from a starting configuration using basic movements such as pickup, putdown, etc.

Also handle situations where actions fail and blocks fall.

Experimental Results



The resulting policy is fast near optimal. These problems are very hard for more traditional planners.

Summary of API

- Approximate policy iteration is a practical way to select policies in large state spaces
- Relies on ability to learn good, compact approximations of improved policies (must be efficient to execute)
- Relies on the effectiveness of rollout for the problem

- There are only a few positive theoretical results
  - convergence in the limit under strict assumptions
  - PAC results for single iteration
- But often works well in practice