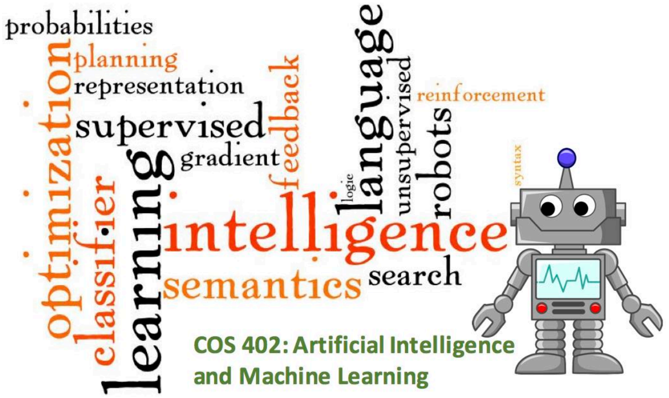


COS 402 – Machine Learning and Artificial Intelligence
Fall 2016

Lecture 20: Reinforcement Learning – part III (function approximation)

Sanjeev Arora Elad Hazan

PRINCETON UNIVERSITY

Admin

- (programming) exercise MCMC – due today
- exercise on RL - announced hereby – due in 1 week
- Last lecture of the course: course summary + “ask us anything”, Prof. Arora + myself. Exercise: submit a question the lecture before (graded)
- Asking questions in class – **everything is allowed, including “can you explain again”** (especially for RL material)
- Next class: Prof. Seung on deep learning
- Class after the next: Dr. Li (please submit questions)

Markov Decision Process

Markov Reward Process, definition:

- Tuple (S, P, R, A, γ) where
  - S = states, including start state
  - A = set of possible actions
  - P = transition matrix $P_{ss'}^{a} = \Pr[S_{t+1} = s' | S_t = s, A_t = a]$
  - R = reward function, $R_s^a = E[R_{t+1} | S_t = s, A_t = a]$
  - $\gamma \in [0,1]$ = discount factor

- Return

$$G_t = \sum_{i=1\ to\ \infty} R_{t+i} \gamma^{i-1}$$

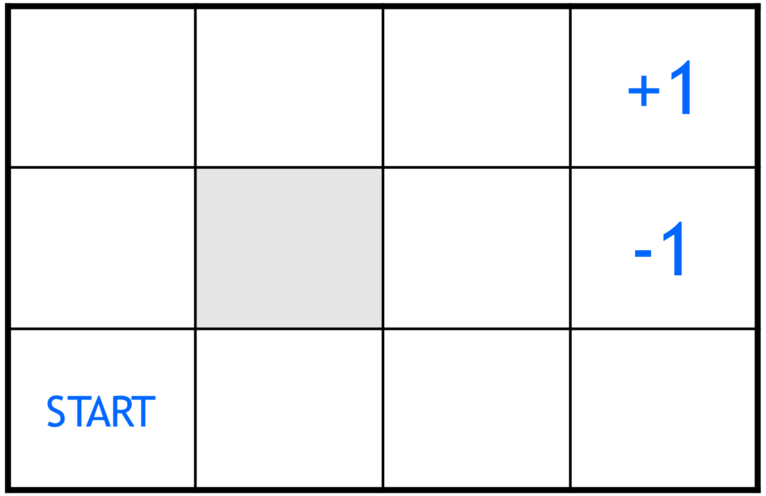
- Goal: take actions to maximize expected return

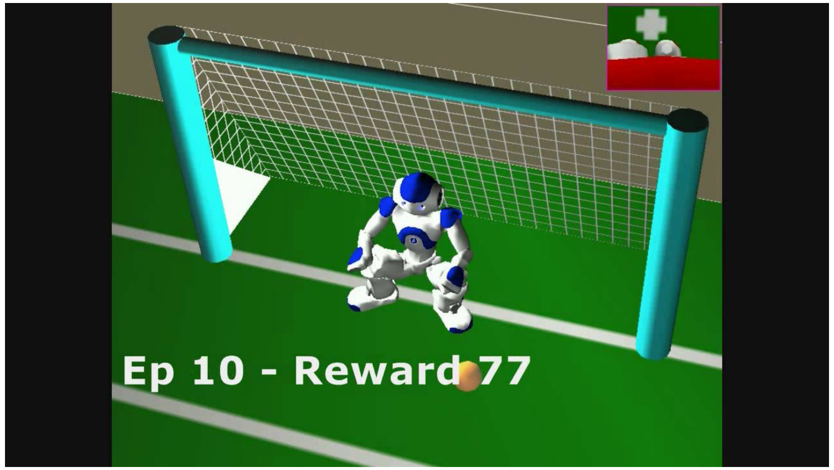
Policies

The Markovian structure ➔ best action depends only on current state!

- Policy = mapping from state to distribution over actions
  $\pi: S \mapsto \Delta(A),\ \pi(a|s) = \Pr[A_t = a|S_t = s]$
- Given a policy, the MDP reduces to a Markov Reward Process

Reminders





Bellman optimality equations

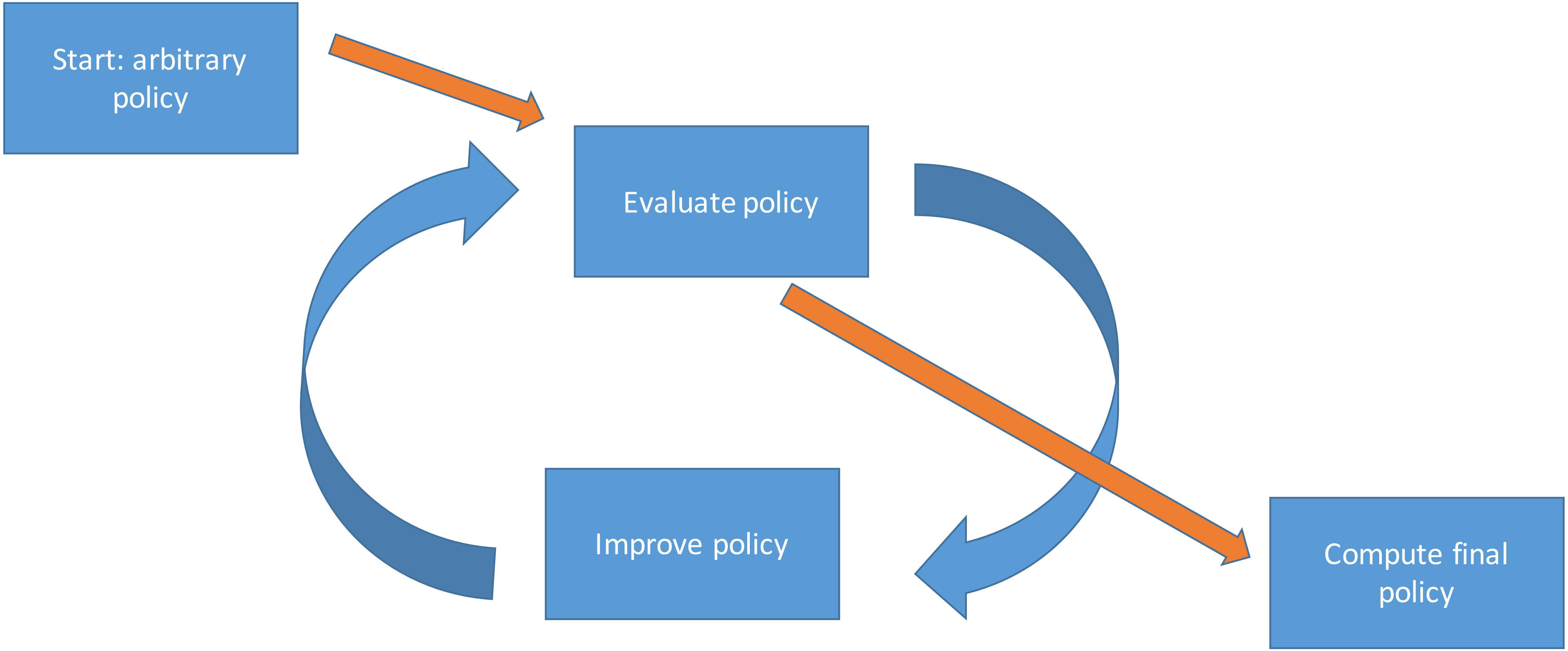
- Bellman equation: $v_*(s) = \max_a \{q_*(s,a)\}$ implies **Bellman optimality equations**:

$$q_*(s,a) = R_s^a + \gamma \sum_{s'} P_{ss'}^a \max_{a'} \{q_*(s',a')\}$$

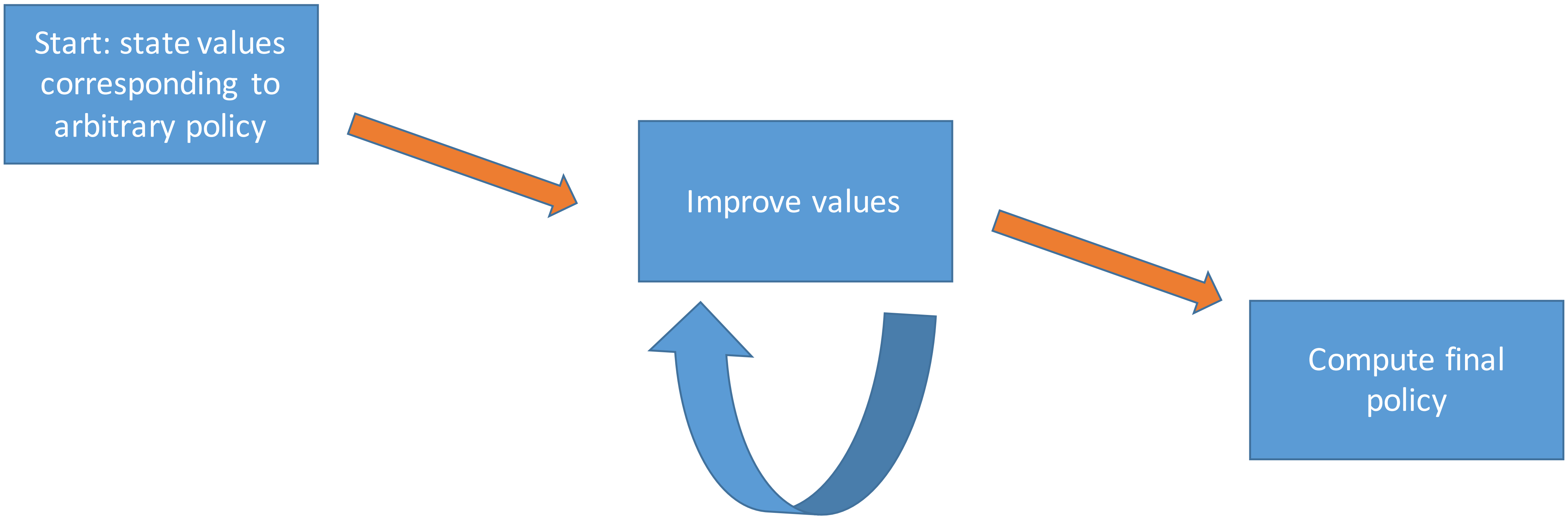
$$v_*(s) = \max_a \left\{ R_s^a + \gamma \sum_{s'} P_{ss'}^a v_*(s') \right\}$$

- Iterative methods based on the Bellman equations: dynamic programming
  - Policy iteration
  - Value iteration

Policy iteration



Value iteration



Model-free RL

Thus far: assumed we know transition matrices, rewards, states, and they are not too large.

What if transitions/rewards are:

1. unknown
2. too many to keep in memory / compute over

“model free” = we do not have the ”model” = transition matrix P and reward vector R

- can estimate P and R from history, and use any of the methods we saw
  (solving for estimate may not be optimal!)

Monte Carlo policy iteration/evaluation

Instead of computing, estimate $v_\pi(s) = E_\pi[G_t | S_t = s]$ by random walk:

- The first time state s is visited, update counter N(s) (increment every time it's visited again)
- Keep track of all rewards from this point onwards
- Estimate of G_t is sum of rewards / N(s).
- Claim: this estimator has expectation $\mathrm{G_t}(s)$, and converges to it by law of large numbers
- Similarly can estimate value-action function $q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$

- What do we do with estimated values?
  - policy iteration requires rewards+transitions
  - Model-free policy improvement:

$$\pi(s) = \arg\max_a \{q_\pi(s, a)\}$$

Temporal Difference learning

Similar idea, but instead of long-horizon estimation, iteratively update by

$$\begin{aligned} v^{\pi}(s) &= v^{\pi}(s) + \alpha(G_t - v^{\pi}(s)) \\ &= v^{\pi}(s) + \alpha(R_{t+1} + \gamma v^{\pi}(s') - v^{\pi}(s)) \end{aligned}$$

- More flexible than MC learning (don't need to wait for estimates to converge)
- Similar idea applies to state-action function q(s,a)

- Never estimate the "model" (transition matrix and reward vector)

LARGE state space

of states may still be prohibitively large!

- Backgammon: 10^{20} states
- Chess: 10^{40} states
- Go: 10^{70} states

Previous methods still infeasible!

Function Approximation: **approximate** the state space (and all model parameters) with a more compact one!

- Reduction in # of states (computation and space)
- More importantly: generalization to unseen states!

Types of (value / action-value) function approximation:

- Linear
- Neural network
- Decision tree
- …

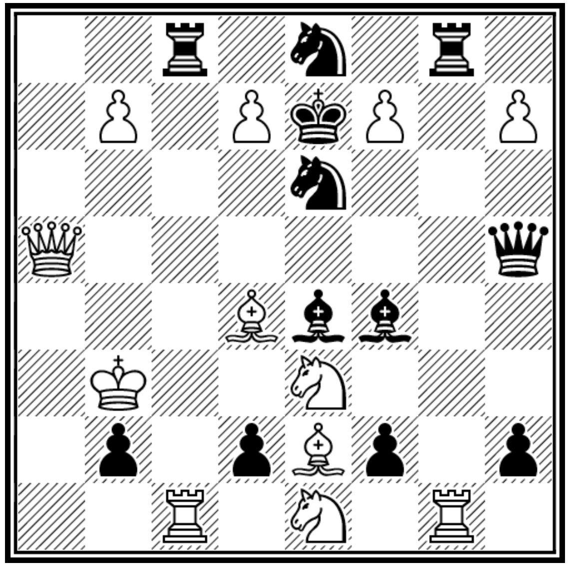
Function approximation

Finding optimal θ → knowledge of value for ALL states!

$$v_\theta(s) = \theta_1 x_1(s) + \theta_2 x(s) + \ \ldots + \theta_n x_n(s) = \theta^\top x(s)$$

10^{40} states are mapped to linear function over n “important” features, i.e.

1. Number of white pieces – black pieces
2. Distance between kings
3. Etc.



Learning a value function over n parameters: supervised learning!

Recall 1st part of coruse: sample complexity, computational complexity,…

Function approximation – computing value function

Natural objective: MSE between approximation and true value per state, i.e.

$$f(\theta) = E_\pi\big(v_\pi(s) - v_\theta(s)\big)^2$$

Minimizing $f(\theta)$?

Stochastic gradient descent!!

$$\theta_{t+1} = \theta_t - \widehat{\nabla f(\theta_t)}$$

Consider linear approximation: $v_\theta(s) = \theta^\top x(s)$, then algorithm becomes:

$$\theta_{t+1} = \theta_t - \eta\, E_\pi\big(v_\pi(s) - v_\theta(s)\big) \times x(s)$$

TD algorithm:

$$\theta_{t+1} = \theta_t - \eta\, (R_{t+1} + \gamma\, \theta^\top x(s') - \theta^\top x(s)) \times x(s)$$

How to improve the policy?

Apply same idea for state-action function, i.e. linear approximation: $q_\theta(s,a) = \theta^\top x(s,a)$ for a state-action vector x(s,a). Optimize MSE of state-action error:

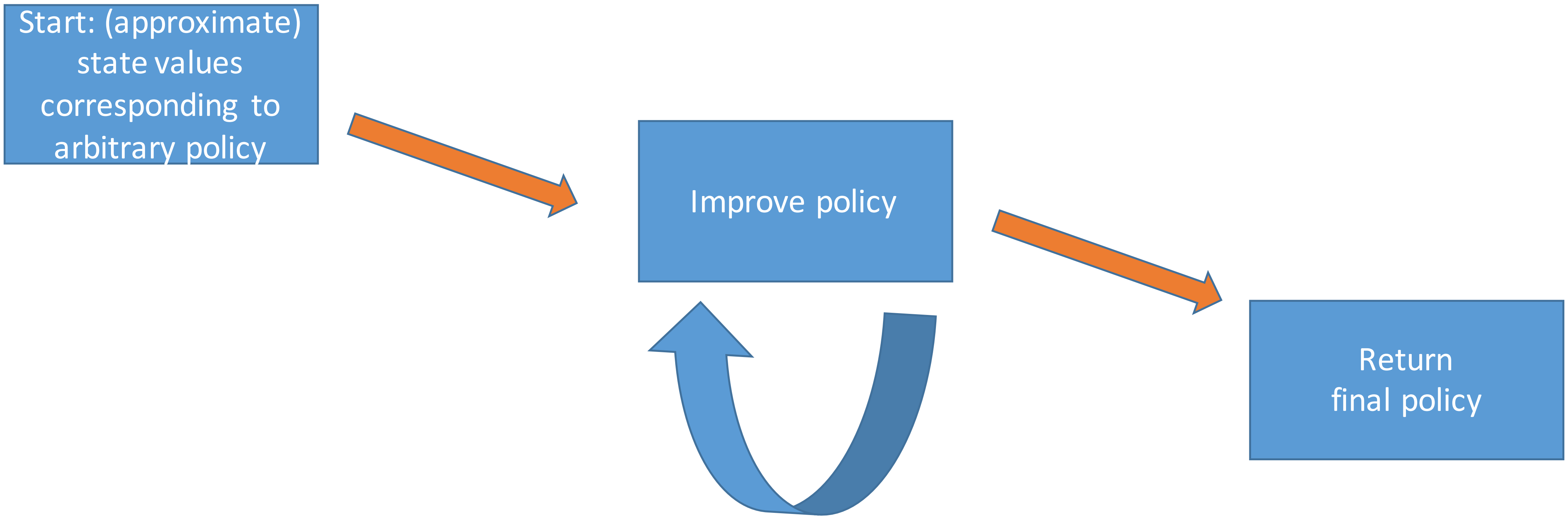
$$f(\theta) = E_\pi\big(q_\pi(s,a) - q_\theta(s,a)\big)^2$$

TD algorithm:

$$\theta_{t+1} = \theta_t - \eta\ \Big(R_{t+1} + \gamma \max_{a'}\{\theta^\top x(s',a')\} - \theta^\top x(s,a)\Big) \times x(s,a)$$

Off-policy vs. on-policy: for on need to add exploration (e.g. instead of greedy a' choice, choose with small probability an action at random).

Policy gradient + function approximation



Policy gradient algorithm for approximate MDP

Parametrized policy, $\pi_\theta(s)$, for example, could be the max action according to q functions:

$$\pi_\theta(s) = \max_a q_\theta(\, s, a)$$

(many times – soft approximation to max to ensure smoothness)

Q-functions can be linear / deep nets, etc.

Plan: gradient descent on the parameter θ to optimize policy directly.

NOT the same as Q-learning w. value approximation! (not trying to optimize q function).

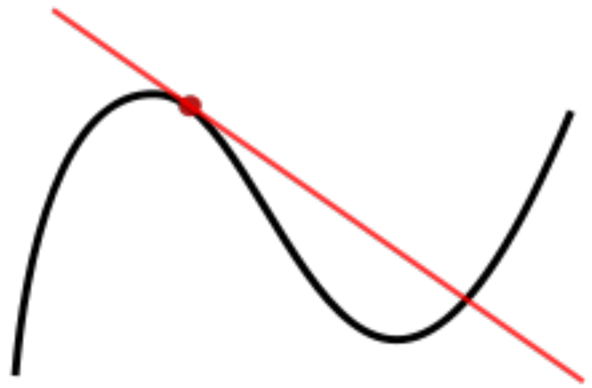
How do we compute gradient?

We can compute: $f(\theta) = E_{\pi_\theta}[v^{\pi_\theta}(s_1)]$

(by evaluating return, running policy)

gradient descent without a gradient

The derivative of a function $f(x): R \mapsto R$



$$f'(x) = \lim_{\delta \mapsto 0} \frac{f(x+\delta) - f(x-\delta)}{2\delta}$$

$$\approx E_{y \in_R \{-1,1\}} \left[\frac{f(x+\delta y) \cdot y}{2\delta} \right]$$

Idea: can sample unbiased coin, and return gradient estimator by single evaluation of the function!

Can you see how to continue?

gradient descent without a gradient

Stokes' theorem for $f(x): R^d \mapsto R$, let $\delta \ll 1$ be very small,

$$\nabla f(x) \approx \nabla E_{|v| \leq 1}[f(x+\delta v) = \frac{d}{\delta} E_{|y|=1}[f(x+\delta v) \cdot v]$$

Idea: can sample function at a **single** point $x + \delta v$, and estimate the gradient for stochastic gradient descent!

(or, almost equivalently, do the previous slide for each coordinate)

Policy gradient without a gradient

Parametrized policy, $\pi_\theta(s)$, for example, could be the max action according to q functions:

$$\pi_\theta(s) = \max_a q_\theta(s, a)$$

(many times – soft approximation to max to ensure smoothness)

Update using gradient descent:

$$\theta_{t+1} = \theta_t - \eta \widetilde{\nabla f(\theta_t)}$$

Where the gradient estimator is obtained by:

$$\frac{d}{\delta} E_{|y|=1}[f(\theta_t + \delta v) \cdot v]$$

for $f(\theta) = E_{\pi_\theta}[v^{\pi_\theta}(s_1)]$

(by evaluating return, running policy)

Summary

- Model free algorithms for solving MDPs
  - Q-function (state-action) and value function estimation via MCMC
  - Same via temporal difference
  - Q-function optimization via temporal difference (or MCMC)

- Function approximation idea – generalization and efficiency
  - Gradient descent approximation to estimate value/Q functions
  - gradient descent to optimize the optimal Q-function directly

- Policy gradient method
  - Gradient descent without a gradient idea