Lecture 20: Approximation Methods in MDPs

- General principle
- Gradient descent methods
- Using linear function approximation
- Control methods with linear function approximation

Why function approximation?

- In general, state spaces are continuous or too large to represent as a table
- If every state has a separate entry in the table, and if we use learning, then every state has to be visited at least a few times before having a good approximation; in the limit every state should be visited infinitely often, which is not feasible

Main idea: Use a function approximator to generalize from the seen states to unseen ones

Representation

- Each state (or state-action pair) is represented as a *feature vector* $\langle \phi_1, \ldots \phi_n \rangle$
- Features are usually chosen a priori, and their range is typically known
- Today we assume no model regarding how features evolve individually over time, but we do assume the Markov property *at the state level*

Value-based methods

We will use a function approximator to represent the *value function*

- The input is the feature vector of the state (or state-action pair)
- The output is the predicted value of the state (or state-action pair)
- The target (desired) output comes from the MDP/RL algorithm
  E.g. for TD(0), the target would be $r_{t+1} + \gamma V(s_{t+1})$

What kind of function approximator can we use?

In principle, there are lots of options:

- A table where several states are mapped to the same location - *state aggregation*
- Gradient-based methods:
  - Linear approximators
  - Artificial neural networks
  - Radial Basis Functions
  - ...
- Memory-based methods:
  - Nearest-neighbor
  - Locally weighted regression

Special requirements

- Allow incremental updates
- Ability to handle non-stationary target functions
- Fast adaptation

State aggregation

- Map the state space S into a finite number of partitions $p_1, \ldots p_n$.
- Compute a value function pretending that the partitions *are* states in an MDP
- Note that if the policy is fixed, we have indeed a Markov process over partitions, so all algorithms for policy evaluation work
- But if we change the policy, the partition MDP changes! So control is not so easy... but still works
- The partition function determines how good a value function we can get over partitions

Memory-based methods

Key idea: just store all examples $\langle s, V(s) \rangle$

Nearest neighbor: Given state s, first locate "nearest" state seen, $\hat{s}$, then estimate $\hat{V}(s) \leftarrow V(\hat{s})$

k*-Nearest neighbor:* Take mean of V values of k nearest neighbors:

$$\hat{V}(s) \leftarrow \frac{\sum_{i=1}^{k} V(s_i)}{k}$$

Locally weighted regression: form an explicit approximation $\hat{V}(s)$ for region surrounding s

- Fit linear function to k nearest neighbors
- Fit quadratic, ...
- Produces "piecewise approximation" to V

Pros and ons of memory-based methods

Advantages:

- Training is very fast
- Learn complex target functions
- Do not lose any information
- Local! Hence have good convergence properties

Disadvantages:

- Slow at query time
- Easily fooled by irrelevant attributes
- Need lots of data (but this is true of RL in general)

Gradient Descent Methods

Consider the policy evaluation problem: learning V^{π} for a given policy π

The approximate value function $V(s_t) = f(\theta, \phi_t)$, where ϕ_t are the attributes (features) describing s_t, and θ is a **parameter vector**
E.g. θ could be the connection weights in a neural network

We will update θ based on the errors computed by the reinforcement learning algorithm

Performance measure

- We want to find a parameter vector θ that minimizes the mean squared error:

$$MSE(\theta) = \frac{1}{2} \sum_{s \in S} P(s) \left(V^{\pi}(s) - V(s)\right)^2$$

  What should P be?

- In our case P is the **on-policy distribution**: distribution of states created when the agent acts according to π

Gradient descent update

Works like in the supervised learning case:

$$\begin{aligned}\theta &\leftarrow \theta - \alpha \nabla_\theta MSE(\theta) \\ &= \theta - \alpha \nabla_\theta \frac{1}{2} \sum_{s \in S} P(s) \left(V^\pi(s) - V(s)\right)^2 \\ &= \theta + \alpha \sum_{s \in S} P(s) \left(V^\pi(s) - V(s)\right) \nabla_\theta V(s)\end{aligned}$$

To do this incrementally, we use the **sample gradient**:

$$\theta \leftarrow \theta + \alpha \left(V^\pi(s) - V(s)\right) \nabla_\theta V(s)$$

The sample gradient is an unbiased estimate of the true gradient.

The rule would converge to a local minimum of the error function, if α is decreased appropriately over time

But where do we get V^π?

Using TD targets

Instead of V^π, we will use the targets that come from the $TD(\lambda)$ algorithm:

$$\theta \leftarrow \theta\alpha\left(\nu_t(s) - V(s)\right)\nabla_\theta V(s)$$

If we use Monte Carlo, then $\nu_t = R_t$ is an unbiased estimate of the true value function, and the algorithm still converges to a local minimum, provided α is decreased appropriately

If $\nu_t = R_t^\lambda$ with $\lambda < 1$, ν_t is **not** an unbiased estimate, and we cannot say anything about the convergence in general

But the algorithm is well defined, and used in practice

On-line gradient descent $TD(\lambda)$

In addition to the weight vector θ, we will have an eligibility trace vector $\mathbf{e}$, with one eligibility for every weight

1. Initialize the weight vector θ arbitrarily, and $\mathbf{e}$ = 0.
2. Pick a start state s
3. Repeat for every time step t:
   (a) Choose action a based on policy π and the current state s
   (b) Take action a, observe immediate reward r and new state s'
   (c) Compute the TD error: $\delta \leftarrow r + \gamma V(s') - V(s)$
   (d) Compute the eligibility of every weight vector to be updated:

$$\mathbf{e} \leftarrow \gamma\lambda\mathbf{e} + \nabla_\theta V(s)$$

   (e) Update the weight vector: $\theta \leftarrow \theta + \alpha\delta\mathbf{e}$
   (f) $s \leftarrow s'$

Linear methods

Each state represented by feature vector $\phi(s) = (\phi_1(s) \ldots \phi_n(s))'$

The value function is a linear combination of the features:

$$V(s) = \theta \cdot \phi(s) = \sum_{i=1}^{n} \theta_i \phi_i(s)$$

So the gradient is very simple: $\nabla_\theta V(s) = \phi(s)$

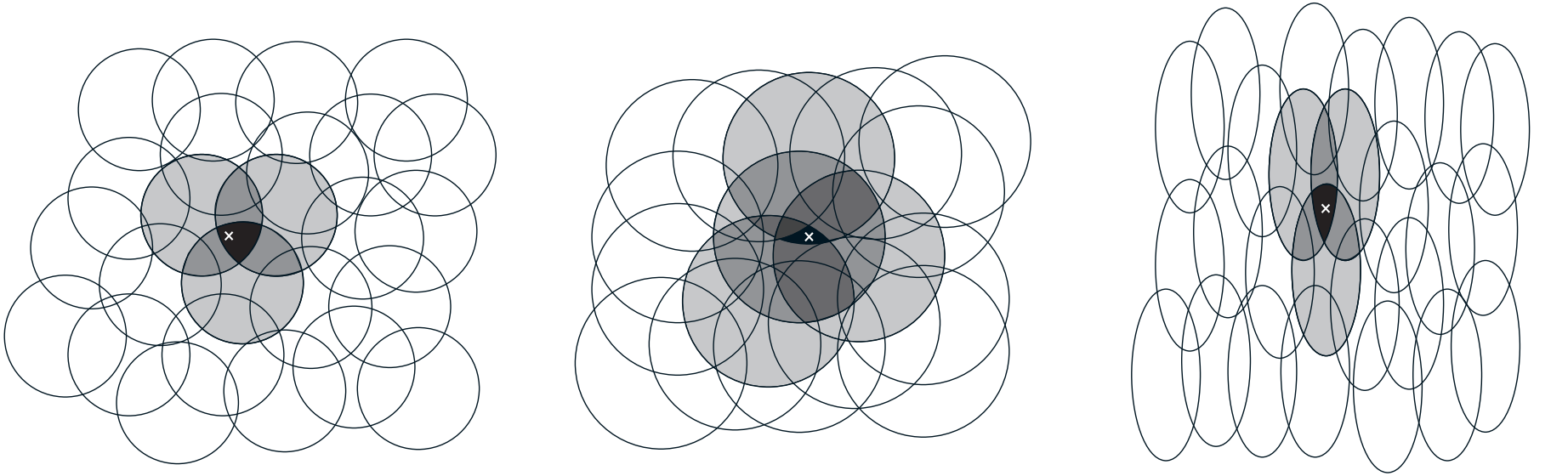
The error surface is quadratic with a single global minimum

Tsitsiklis and Van Roy: Linear gradient-descent $TD(\lambda)$ converges w.p.1 to a parameter vector θ_∞ in the "vicinity" of the best parameter vector θ^*:

$$MSE(\theta_\infty) \leq \frac{1-\gamma\lambda}{1-\gamma} MSE(\theta^*)$$

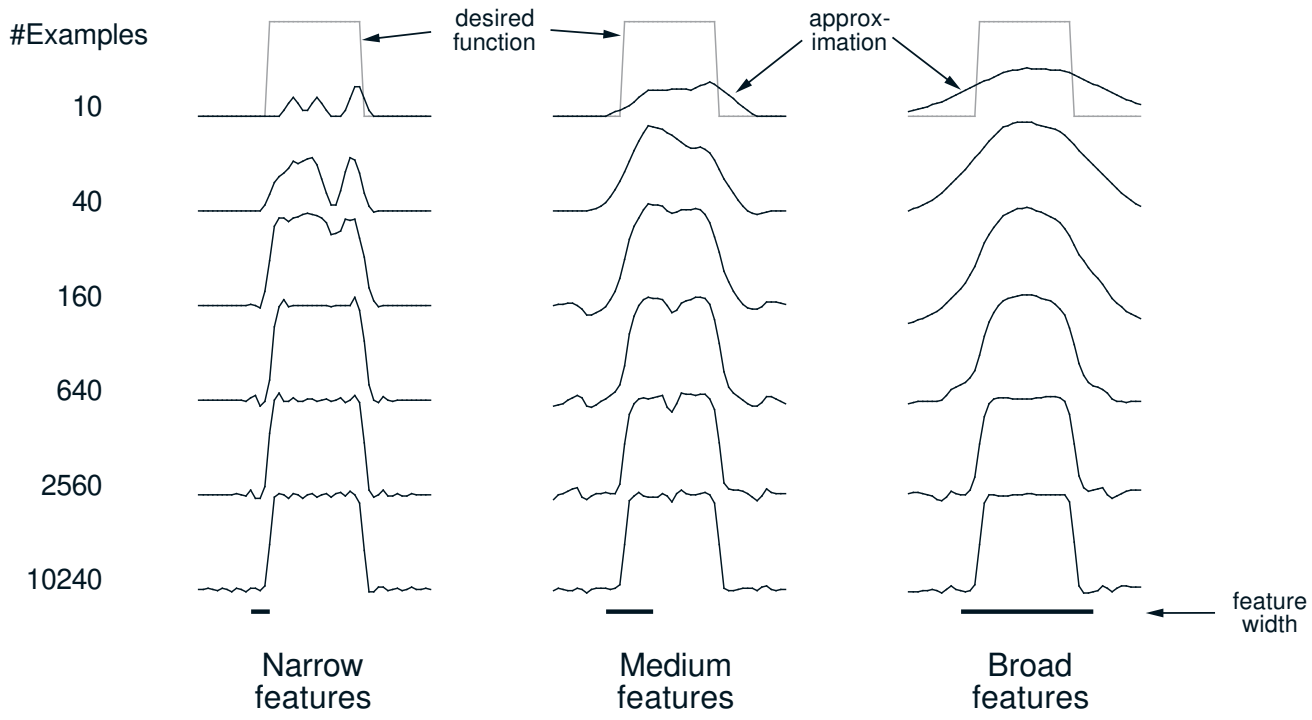
Coarse coding

Main idea: we want linear function approximators, but with **lots of features**, so they can represent complex functions



a) Narrow generalization b) Broad generalization c) Asymmetric generalization

Speed of learning with coarse coding



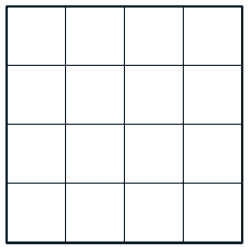
The width of the cells affects the speed, not the precision of the learner

Discretizing the state space

Suppose we have a continuous state space with two continuous variable (e.g. like in the Mountain-Car task)

The simplest tile coding approximator would be just a grid discretizing the state space:

- The features are all 0 except for the cell holding the current state, which is 1 (like a 1-of-n encoding)
- All states in the same cell have the same value (given by the weight of the cell)



Pros and cons of discretizations

Pros:

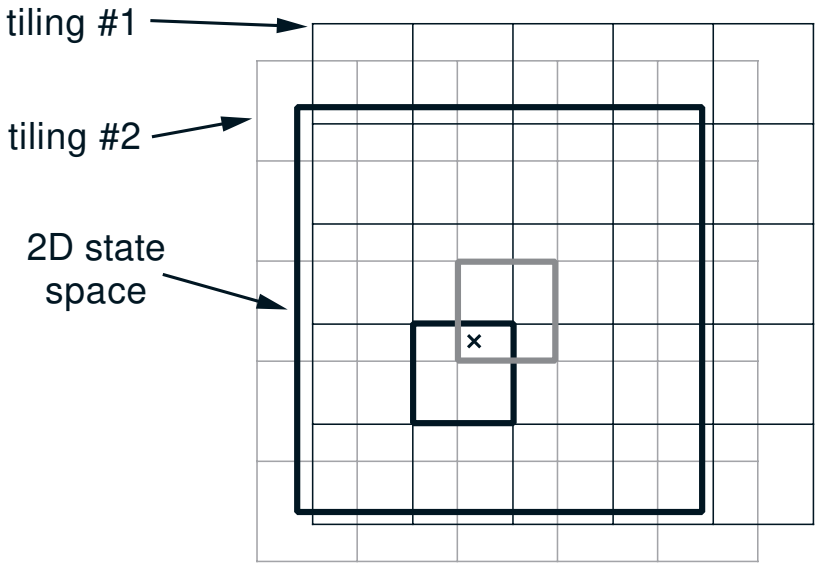
- Easy to compute the value function of a state
- Easy to update as well (more like the table lookup case).

Cons:

- To get good precision, we need a very fine grid - going back to the table lookup case?
- States in the vicinity of a separation line could have radically different values (approximation is discontinuous)

Tile coding (continued)

Main idea: Overlap several tilings!



Shape of tiles ⇒ Generalization

#Tilings ⇒ Resolution of final approximation

Characteristics of tile coding

- Each tile is a binary feature
- The number of features that are activated at any time is constant, equal to the number of tilings
- It is easy to compute the indices of the features activated, and easy to compute the weighted sum
- The overall discretization is very fine, and at the same time the discontinuities are smoothed out
- The shape of the tiles reflects prior domain knowledge

Cf. CMAC (Albus, 1971)

Control with function approximation

- Input: a description of the state-action pair (s_t, a_t)
- Output: an action-value function $Q(s_t, a_t)$
- The general gradient descent rule:

$$\theta \leftarrow \theta + \alpha\left(\nu_t - Q(s_t, a_t)\right)\nabla_\theta Q(s_t, a_t)$$

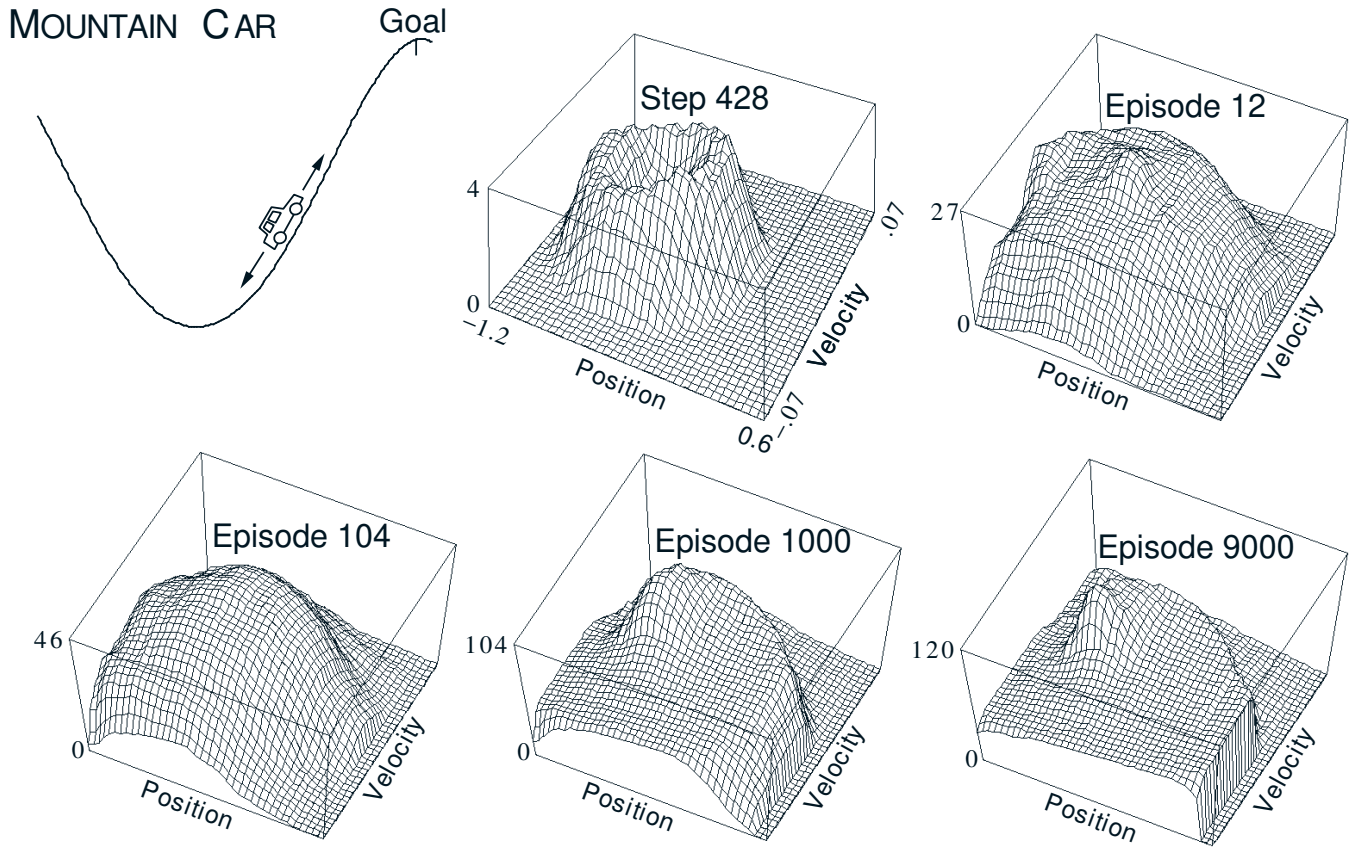
- Example: Sarsa(λ)

$$\theta \leftarrow \theta + \alpha\delta_t \mathbf{e}_t$$

where

$\delta_t = r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)$ and $\mathbf{e}_t = \gamma\lambda\mathbf{e}_t + \nabla_\theta Q(s_t, a_t)$

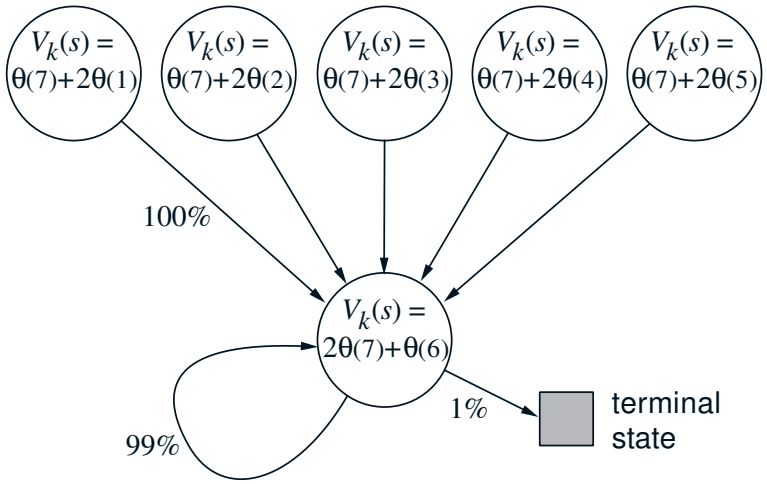
Illustration: Mountain-Car task

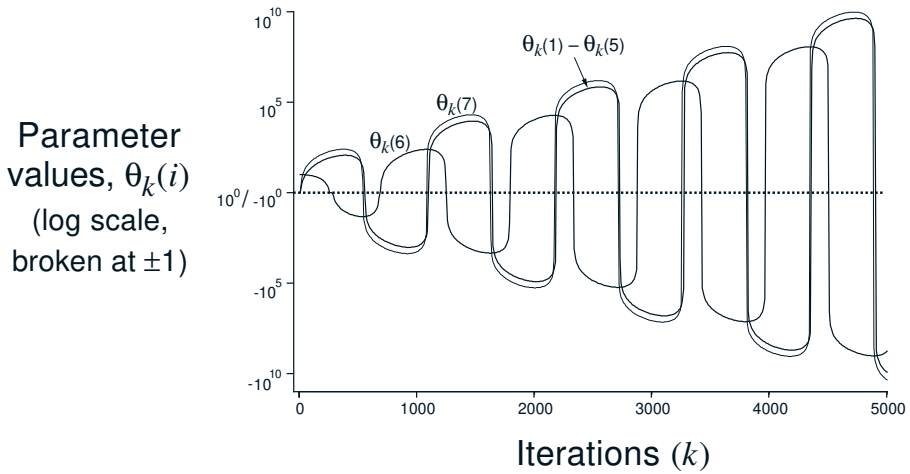


Theory of control algorithms

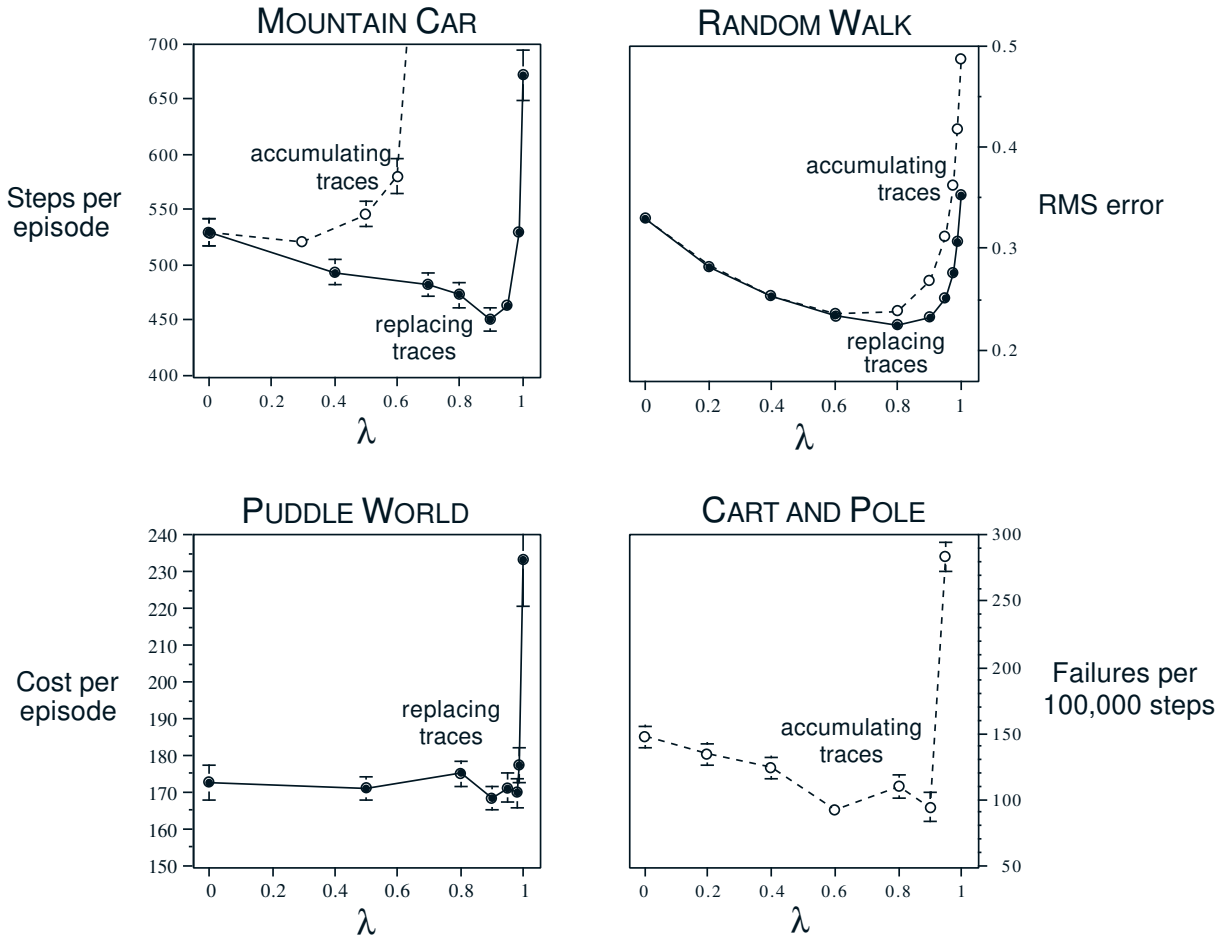
- Sarsa proven to converge to a region of policy space (Gordon, 2001)
- Q-learning shown to diverge in extremely simple examples
- A few off-policy evaluation algorithms that might shed light into Q-learning behavior (Precup et al, 2000, 2001)
- One of the convergence problems is bootstrapping

Baird's counterexample





Should we bootstrap?



Policy-based methods

Main idea: Instead of approximating the value function, approximate the policy directly

- A function approximator which outputs the probability of taking an action
- Parameters are updated in the direction of the gradient of the return
- We can compute this if the policy has special forms (e.g. softmax)
- **Much better theoretical guarantees!**
  The policy changes smoothly
- But initial empirical evidence suggests slow in practice