Reinforcement Learning (3)

Machine Learning 64-360

Norman Hendrich

University of Hamburg
MIN Faculty, Dept. of Informatics
Vogt-Kölln-Str. 30, D-22527 Hamburg
hendrich@informatik.uni-hamburg.de

SS 2015

Contents

TD(λ) and Eligibility Traces

- Q-learning and SARSA look one step into the future
- updating $Q(s, a)$ online
- while Monte-Carlo waits until episode ends

$\Rightarrow$ the TD(λ) algorithms combine both ideas

- a family of methods to improve learning (e.g. speed)
- better handle *delayed rewards* (far in the future)
- update multiple Q values, not just current $Q(s, a)$
- allows MC techniques to be used on non-episodic tasks

Watkins 1989, Jaakkola, Jordan and Singh 1994, Sutton 1998, Singh and Sutton 1996

TD(λ) and Eligibility Traces

theoretical viewpoint, or *forward view*:

- a bridge from TD to Monte Carlo methods
- TD methods augmented with eligibility traces produce a spectrum of algorithms, with Monte Carlo methods at one end, and one-step TD methods at the other
- intermediate methods maybe better than either "pure" method

pragmatical viewpoint, the *backward view*:

- gain intuition about the algorithms
- the trace marks the memory parameters associated with the event as (eligible) candidates for learning changes
- TD steps update multiple (visited) states or actions

n-step TD prediction

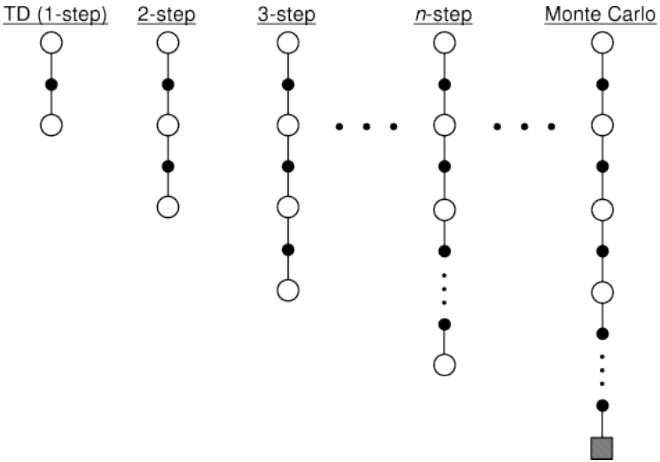
consider estimating $V^{\pi}(s)$ from sample episodes generated following policy π

- MC methods perform a backup based on the entire episode
- simple TD methods just consider the next reward, plus the discounted value of the state one step later, which encodes the estimates of the remaining rewards

$\Rightarrow$ why not use n-step methods that perform a backup based on an intermediate number of rewards: more than one, but less than all?

- those methods are still TD methods, because they update an earlier estimate based on how it differs from a later estimate; in this case up to n steps later.

Spectrum of *n*-step TD methods



spectrum of *n*-step methods, ranging from simple one-step TD methods to the full-episode backups of Monte Carlo

The *n*-step return

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T \qquad \text{MonteCarlo}$$

$$R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1}) \qquad 1 - \text{step}$$

$$R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2}) \qquad 2 - \text{step}$$

$$\ldots$$

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n})$$

also called the *corrected n-step truncated return*: the Return truncated after *n*-steps, and then approximately corrected by adding the estimated value of the *n*-th next state.

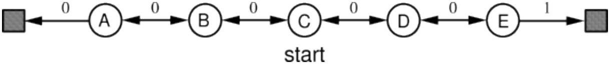
The *n*-step backup

- one backup operation towards the *n*-step return
- in the tabular case:

$$\Delta V_t(s_t) = \alpha \left[R_t^{(n)} - V_t(s_t) \right],$$

  with α a positive step-size parameter
- all other states $s \neq s_t$ are not updated
- *on-line update*: during an episode, $V_{t+1}(s) = V_t(s) + \Delta V_t(s)$
- *off-line update*: increments are accumulated in a separate array, but not used to change the value estimates until the end of this episode.

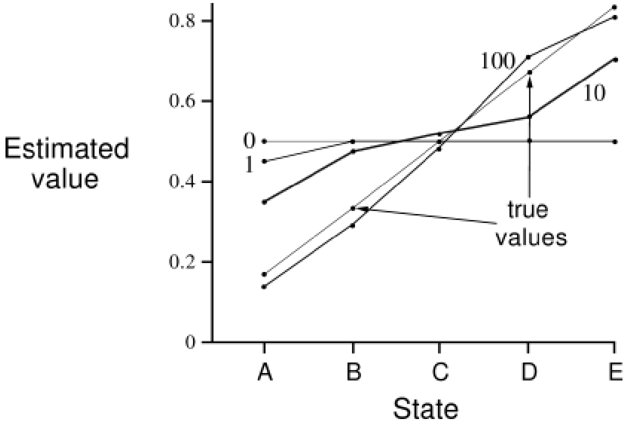
Example: n-step TD on Random Walk



- random-walk starting at state (C)
- one step to the left or right at each step, equal probabilities
- reward 0 on every step, 1 for reaching the right goal state
- episode terminates on either the left or right goal state

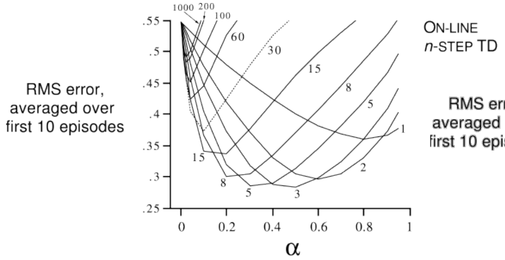
- in this case, $V(s)$ is just the probability of terminating on the right when starting in S: $\{0, \frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}, \frac{5}{6}, 1\}$

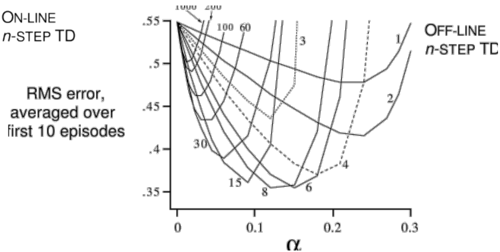
Example: TD(0) on Random Walk



Value function learned by TD(0) after 0,1,10,100 episodes for a 5-state random walk.

Example: *n*-step TD on Random Walk



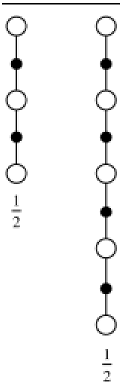


- *n*-step TD methods on the 19-state random walk
- performance measured as RMS error: $\sum_s [V_t(s) - V(s)]^2$
- as a function of step-size α for different values of *n*
- online (during episode) and off-line updates

The Forward view of TD(λ)

- we can also combine different n-step methods
- e.g., backup using half a two-step return and half a four-step return, $R_t^{ave} = \frac{1}{2} R_t^{(2)} + \frac{1}{2} R_t^{(4)}$.
- well-defined, if weights sum to 1

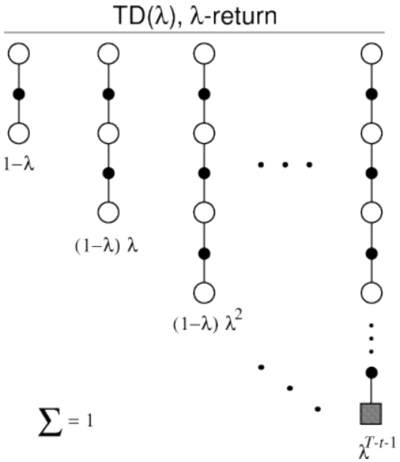
- a completely new class of algorithms
- combining properties of the different individual methods



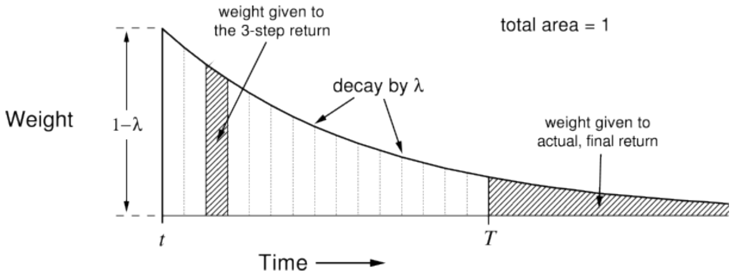
Backup diagram for TD(λ)

- one particular way
  to average n-step backups
  weighted proportional to λ^{n-1}

- λ-return:
  $R_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} R_t^{(n)}$



Weighting of each n-step return

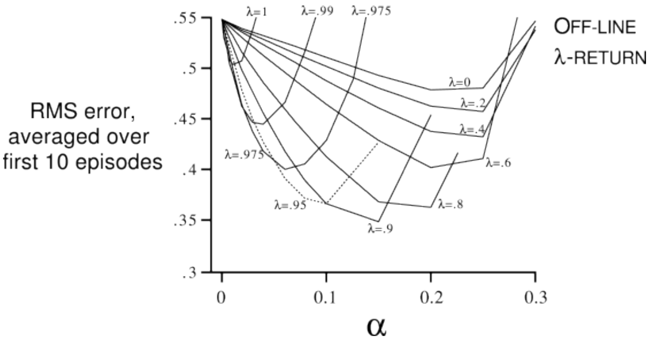


$$R_t^{\lambda} = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + \lambda^{T-t-1} R_t$$

$\lambda = 1$: main sum is zero, remaining term is R_t: Monte Carlo

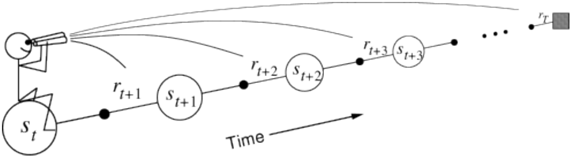
$\lambda = 0$: reduces to $R_t^{(1)}$, so TD(0)

Example: TD(λ) on the Random Walk



- performance of TD(λ) on the 19-state random walk
- step-size α, different values of λ
- smallest RMS error with intermediate values of λ

The Forward view

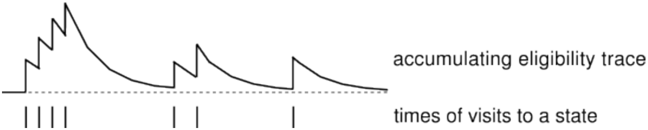


- from each state s visited, look forward in time to all future rewards, and decide how best to combine them.
- problem: this is hard to implement, using at each step knowledge of what will happen many steps later ...

The Backward view of TD(λ)

- reserve an additional memory variable for each state, the *eligibility trace* $e_t(s) \in \mathbb{R}^+$
- On each step t, the elibibility traces for all states decay by γ^λ, but the trace for the one state visited on the step is incremented by 1:

$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t;, \\ \gamma\lambda e_{t-1}(s) + 1, & \text{if } s = s_t; \end{cases}$$



The Backward view of TD(λ)

- the traces record which states have recently been visited,
- where recently is defined in terms of γ^λ

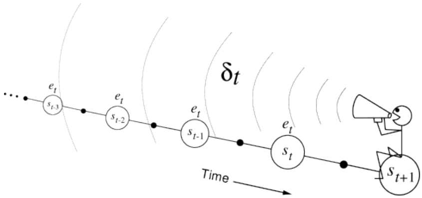
- the traces indicate the degree to which each state is *eligible* to change during learning
- for example, the TD "error" for state-value prediction is
  $\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$

- and the TD(λ) update becomes:
  $\Delta V_t(s) = \alpha \delta_t e_t(s)$, for all $s \in S$

On-line tabular TD(λ)

initialize $V(s)$ arbitrarily and $e(s) = 0$, for all $s \in S$
repeat (for each episode):
  initialize s
  repeat (for each step of episode):
    $a \leftarrow$ action given by policy π for s
    take action a, observe reward r and next state s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $e(s) \leftarrow e(s) + 1$
    for all s:
      $V(s) \leftarrow V(s) + \alpha\delta e(s)$
      $e(s) \leftarrow \gamma\lambda e(s)$
    $s \leftarrow s'$
  until s is terminal

The Backward view



- "shouting" updates back to previously visited states
- $\lambda = 0$: all traces are zero, except for those at s_t, Q-learning and SARSA are TD(0) methods
- $0 < \lambda < 1$: more of the preceding states are changed, but each more temporally distant state is changed less
- $\lambda = 1$: credit given to earlier states falls by γ at each step, giving Monte Carlo for $\gamma = 1$

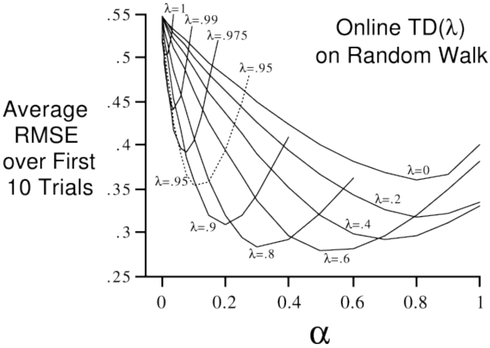
Equivalence of forward and backward view

- trying to build an online-algorithm (backward) that achieves the same weight updates as the off-line λ-return algorithm
- align the forward (theoretical) and backward (implementation) views of TD(λ)
- want to show that the value-function updates are the same at the end of an episode, so

$$\sum_{t=0}^{T-1} \Delta V_t^{TD}(s) = \sum_{t=0}^{T-1} \Delta V_t^{\lambda}(s_t) I_{ss_t}, \quad \text{for all } s \in S,$$

- see Sutton and Barto section 7.4 for the math and proof ideas

Example: Online TD(λ) on the Random Walk



- performance of online TD(λ) on the 19-state random walk
- step-size α, different values of λ
- note: a bit better performance than the off-line algorithm

Sarsa(λ)

How to generalize TD(λ) for control? That is, learning $Q(s, a)$ instead of learning $V(s)$?

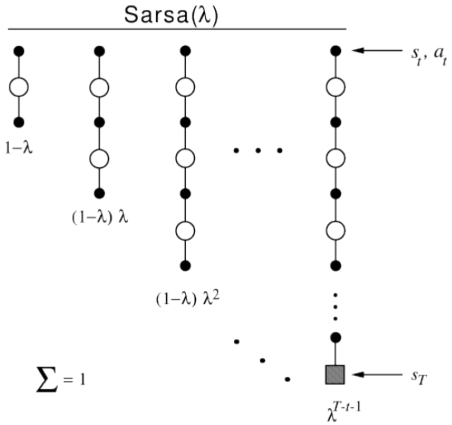
- $Q_{t+1}(s, a) = Q_t(s, a) + \alpha \delta_t e_t(s, a)$, for all (s, a)

$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1}) - Q_t(s_t, a_t)$$

$$e_t(s, a) = \begin{cases} \gamma \lambda e_{t-1}(s, a) + 1, & \text{if } s = s_t \text{ and } a = a_t; \\ \gamma \lambda e_{t-1}(s, a) & \text{otherwise} \end{cases}$$

- from time to time, improve policy π using greedification

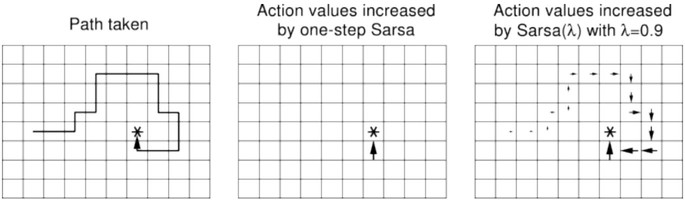
Sarsa(λ) backup diagram



Sarsa(λ) algorithm

initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s, a
repeat (for each episode):
  initialize s, a
  repeat (for each step of episode):
    $a \leftarrow$ action given by policy π for s
    take action a, observe reward r and next state s'
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
    $e(s,a) \leftarrow e(s,a) + 1$
    for all s, a:
      $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
      $e(s,a) \leftarrow \gamma\lambda e(s,a)$
    $s \leftarrow s'$, $a \leftarrow a'$
  until s is terminal

Speedup of learning using Sarsa(λ)



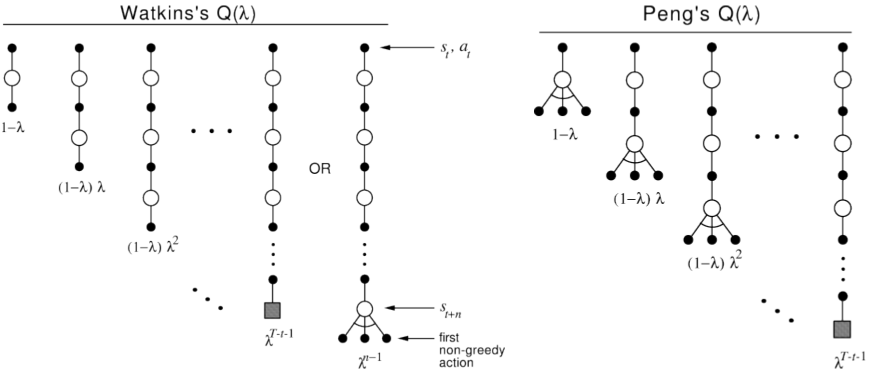
- example path of the learner, ending in goal state '*'
- TD(0) methods will only update the single $Q(s, a)$ for the immediately preceding state
- eligibility-trace-methods update many $Q(s, a)$ values weighted by relevance

$Q(\lambda)$ algorithm?!

- $Q(\lambda)$: Watkins's and Peng's algorithms
  - Q-learning learns greedy policy while following another policy
  - n-step update only possible while using greedy policy
- eligibility traces for actor-critic methods
- replacing traces vs. accumulating traces
  - clip $e_t(s) \leq 1$, can improve learning speed
- methods that use variable λ
- implementation issues
- can TD(λ) also works in non-Markovian environments?

- see Sutton and Barto, chapter 7 for details

$Q(\lambda)$



- learning $Q(s, a)$ for greedy policy while following current π
- two different ways to handle non-greedy actions

Summary: TD(λ)

A family of improved temporal-difference learning algorithms to speed-up learning

- interpolate between 1-step TD(0) and n-step TD(n) Monte-Carlo methods
- update more than one element of V or Q at each time step
- simple implementation using *eligibility traces*

- parameter λ sets the time-scale of the learning

Generalization and function approximation

Application of RL ideas to "real world" problems?!

Generalization and function approximation

so far, we considered the so-called *tabular case*:

- discrete state and action spaces,
  $S = \{s_0, s_1, \dots, s_n\}$, $A = \{a_0, a_1, \dots, a_m\}$
- state-space small enough for in-memory representation
- many theoretical results assume that all (s, a) pairs are visited infinitely often
- corresponding time requirements in addition to memory

for continuous state-spaces we need *generalization*:

- most states visited never experienced exactly before
- need to generalize from previously experienced similar states
- combine RL algorithms with *function approximation*

The key issue: Generalization

"How can experience with a limited subset of the state space be usefully generalized to produce a good approximation over a much larger subset?"

- most states visited never experienced exactly before
- most actions never performed exactly before
- "complex sensations": e.g., visual images, high-DOF problems

Generalization via function approximation

Basic idea: represent continuous state-space or state-action-space using *feature functions* with parameters $\vec{\theta} \in \{\Theta\}$, with $|\Theta| \ll |S|$ or $|\Theta| \ll |S \times A|$. Then, use RL-algorithms to adjust the parameters θ;

All common function approximation methods can be used:

- polynomial and spline interpolation functions (low degrees)
- statistical curve-fitting, decision trees
- artificial neural-networks (multi-layer perceptron)
- kernel-SVMs
- ...

where the context is often high-DOF problems.

Example: Pacman

- let's say we discover through experience that this state is bad:

- in naive Q-learning, we know nothing about this state or its neighbor states

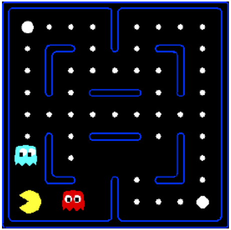
- or even this one:

(idea and images: Abbeel & Peters, RL tutorial, ICRA-2012)

Pacman features

Solution: describe the environment state using a vector of *features*

- features are functions from states to real numbers (often 0/1) that capture important properties of the state
- examples:
  - distance to closest ghost
  - distance to closest dot
  - number of ghosts
  - $1/(\text{dist to dot})^2$
  - is Pacman in a tunnel? (0/1)
  - . . . . etc.



- of course, can also describe a Q-state (s, a) with features
  - e.g. action (s, a) moves closer to food

Pacman features

- using a feature representation, we can write a Q-function or the value-function V for any state using a few weights:

$$V(s) = w_1 f_1(s) + w_2 f_2(s) + \ldots + w_n f_n(s)$$

$$Q(s,a) = w_1 f_1(s,a) + w_2 f_2(s,a) + \ldots + w_n f_n(s,a)$$

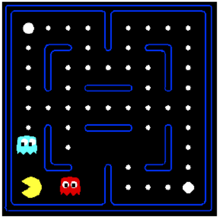
- advantage: experience is summed up in a few powerful numbers
- disadvantage: states may share features but be very different in value

Pacman features

$$Q(s,a) = 4.0 f_{DOT}(s,a) - 1.0 f_{GST}(s,a)$$

$$f_{DOT}(s, \text{NORTH}) = 0.5$$

$$f_{GST}(s, \text{NORTH}) = 1.0$$



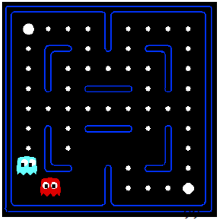
$$Q(s,a) = +1$$

$$R(s,a,s') = -500$$

$$error = -501$$

$$w_{DOT} \leftarrow 4.0 + \alpha\,[-501]\,0.5$$

$$w_{GST} \leftarrow -1.0 + \alpha\,[-501]\,1.0$$



$$Q(s,a) = 3.0 f_{DOT}(s,a) - 3.0 f_{GST}(s,a)$$

Q-function: tabular vs. linear

Tabular Q-function	Linear Q-function
Q table	$Q(x,u) = \sum_{i=1}^{n} w_i f_i(x,u)$

Sample: $r + \gamma \max_{u'} Q(x', u')$

Difference: $\left[r + \gamma \max_{u'} Q(x', u') \right] - Q(x,u)$

Update:

Tabular	Linear
$Q(x,u) \leftarrow$	$\forall i, \quad w_i \leftarrow$
$Q(x,u) + \alpha$ [difference]	$w_i + \alpha$ [difference] $f_i(x,u)$

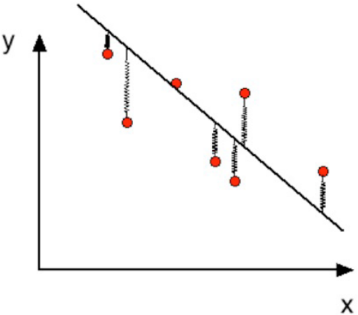
(Abbeel & Peters, ICRA 2012)

Value prediction with function approximation

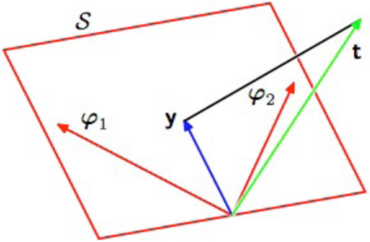
- try to estimate $V^{\pi}(s)$ from experience generated using policy π
- but $V^{\pi}(s)$ no longer repesented as a table,
- instead approximated as $V^{\pi}_{t,\theta}\big((s)\big)$ at time step t

- measure approximation error using suitable loss-functions,
  e.g. weighted mean-squared error:
  $MSE(\vec{\theta}) = \sum_{s \in S} P(s) \big[V^{\pi}(s) - V^{\pi}_{t,\theta}(s) \big]^2$
- where P is a distribution weighting the errors of different states
- usually impossible to reduce the error to zero at all states
- remember: many more states s than parameters θ

Interpretation of the least-squares cost function

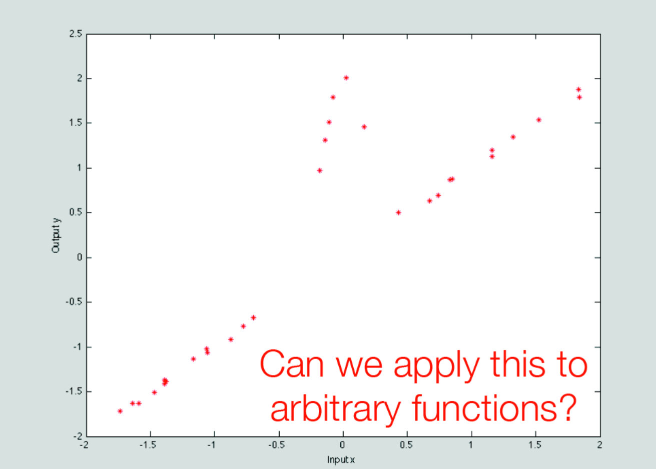


physical interpretation

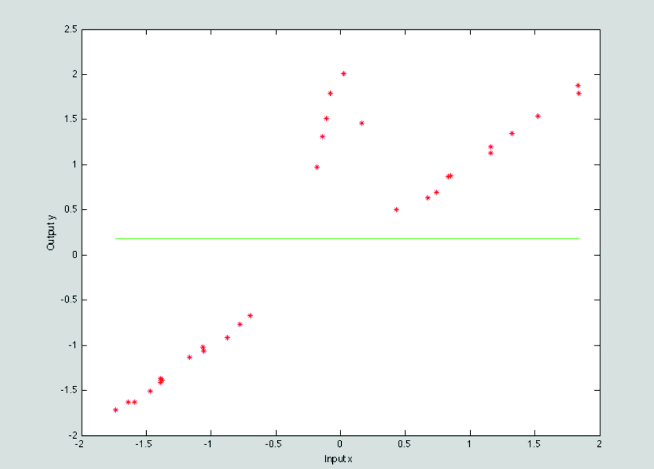


geometrical interpretation

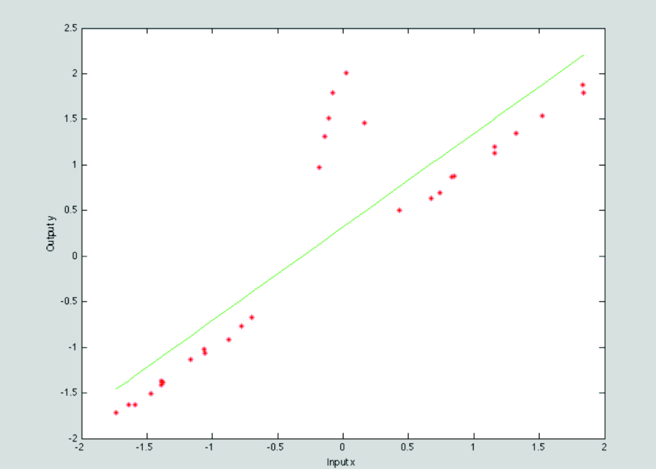
Function approximation: example data



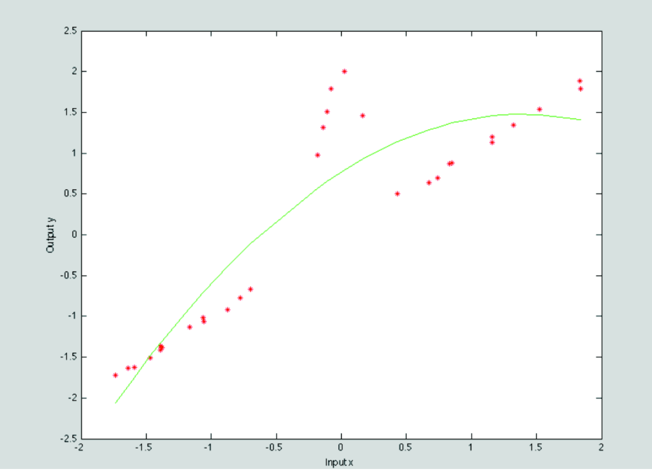
Fitting an easy model: polynomial with $n = 0$



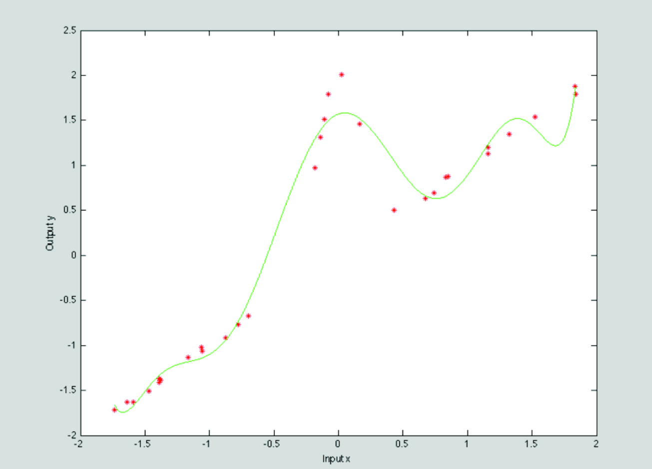
More features: polynomial with $n = 1$



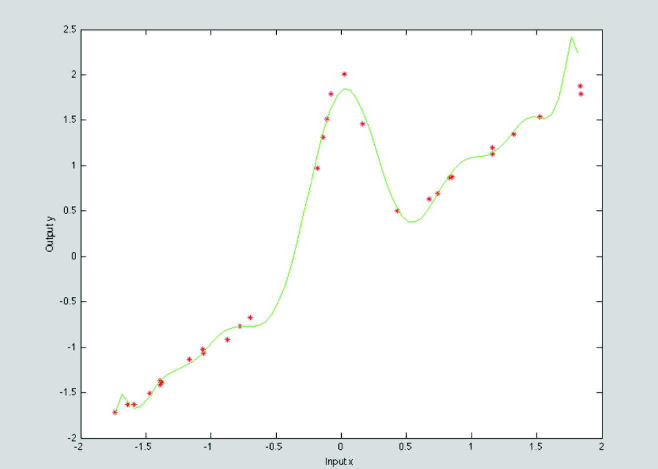
More features: $n = 2$



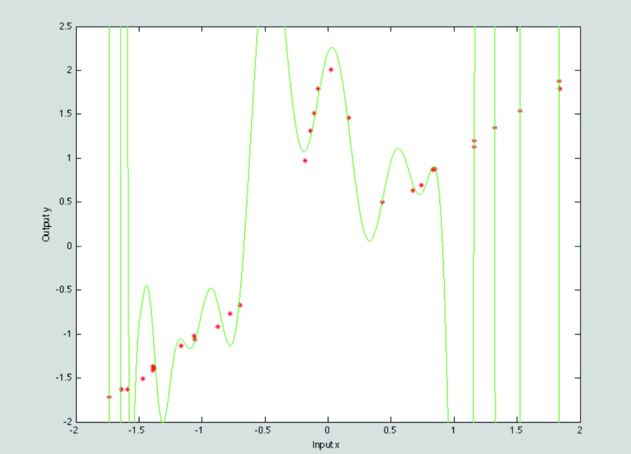
More features: $n = 8$



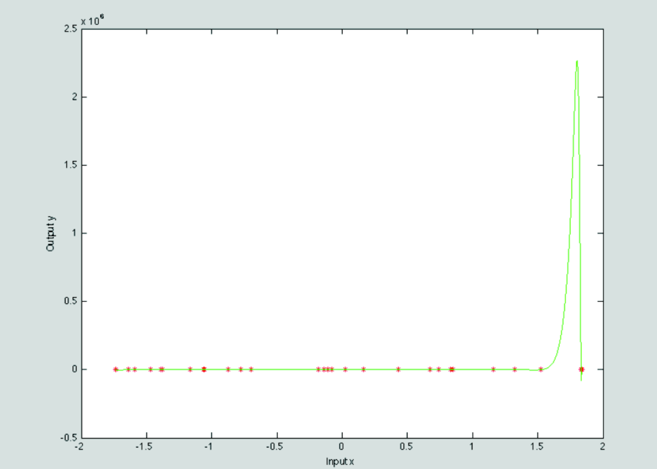
More features: $n = 15$



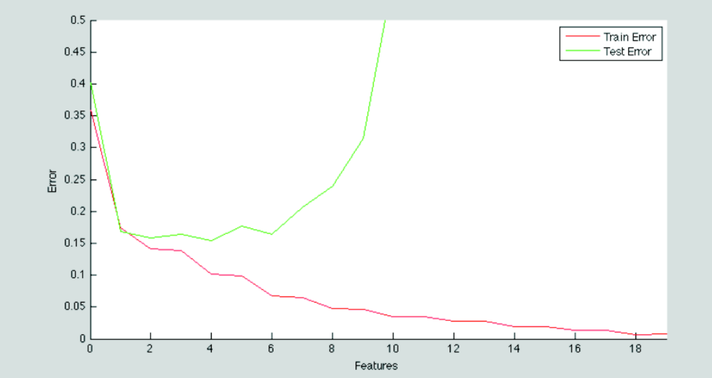
More features: $n = 200$



More features: $n = 200$



Training error vs. test error



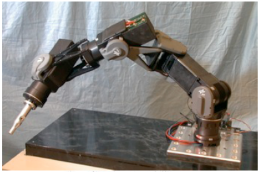
(remember the magic tool: leave-one-out cross-validation)

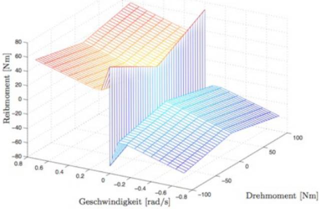
Selecting the approximating functions

What to do when you don't know the features?

- useful features are known in many real applications
- however, we almost certainly don't know all features needed
- example: rigid body dynamics
  - friction has no good features, and may be self-referential
  - unknown dynamics causes huge problems (requires more state variables)

- there may also be too many features...





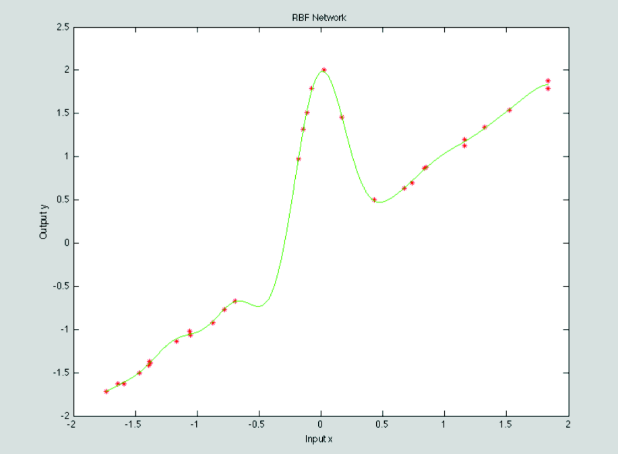
Can we proceed when we don't know the features?

Yes!

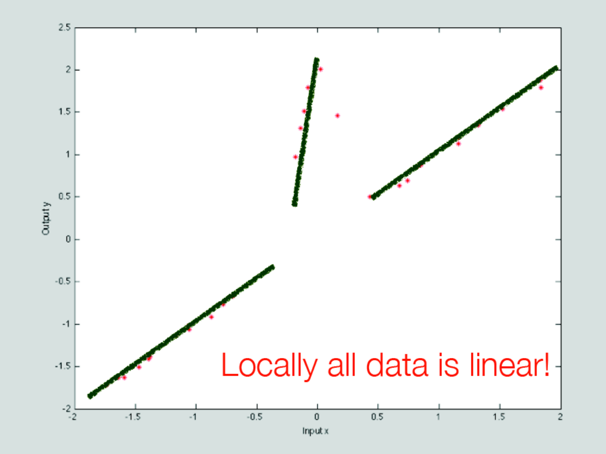
we need to find machine learning approaches that generate the features directly based on the data.

1. *radial basis functions*: create an optimal smooth approximation
2. *locally-weighted regression*: localize relevant parts of the data and try to interpolate
3. *kernel regression*: find useful features by going into *function space* using a kernel

Better features: radial-basis functions

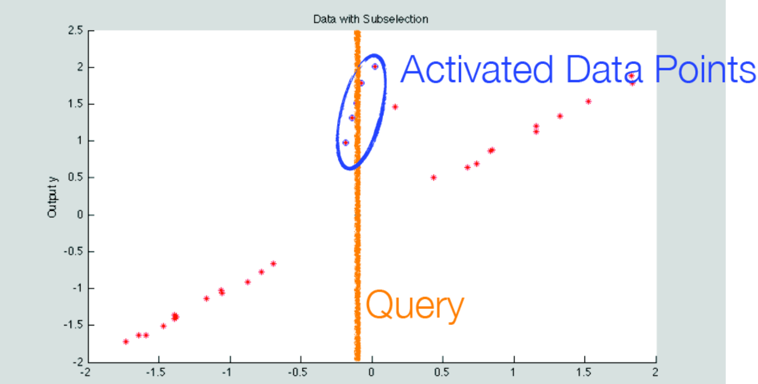


Locally, all data is linear

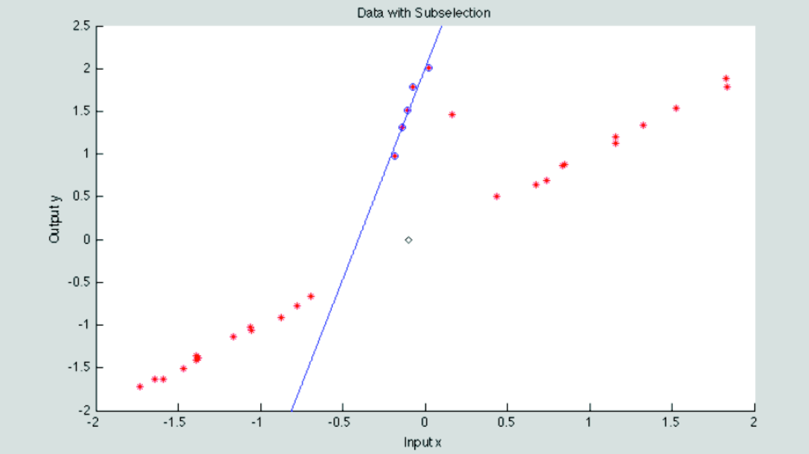


Locally linear solutions

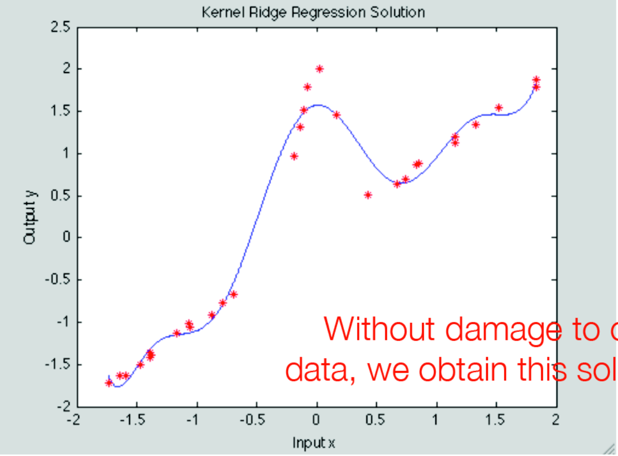
$$w(\mathbf{x}) = \begin{cases} 1 & \text{if } \|\mathbf{x} - \mathbf{x}_q\| \leq \epsilon, \\ 0 & \text{otherwise.} \end{cases}$$



Locally linear solutions for a query



Exponential kernel



Without damage to our data, we obtain this solution!

Value prediction with function approximation

- try to estimate $V^{\pi}(s)$ from experience generated using policy π
- but $V^{\pi}(s)$ no longer repesented as a table,
- instead approximated as $V^{\pi}_{t,\theta}((s))$ at time step t

- measure approximation error using suitable loss-functions,

  e.g. weighted mean-squared error:

  $MSE(\vec{\theta}) = \sum_{s \in S} P(s) \left[V^{\pi}(s) - V^{\pi}_{t,\theta}(s) \right]^2$
- where P is a distribution weighting the errors of different states
- usually impossible to reduce the error to zero at all states
- remember: many more states s than parameters θ

Gradient-Descent methods

- parameter vector $\vec{\theta}$ is a column vector with a fixed number of real-valued components
- assume that $V_\theta(s)$ is a smooth differentiable function of $\vec{\theta}$ for all $s \in S$
- on each time step t we observe a new example $s_t \to V^\pi(s_t)$

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \frac{1}{2}\alpha\nabla_{\vec{\theta}_t}\left[V^\pi(s_t) - V_t(s_t)\right]^2$$

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\left[V^\pi(s_t) - V_t(s_t)\right]\nabla_{\vec{\theta}_t}V_t(s_t)$$

where ∇ denotes the vector of partial derivatives, the *gradient*

Gradient-Descent methods for TD(λ)

- optimize the approximation error on the observed examples
- GD-methods adjust the paramter vector by a small amount in the direction that would most reduce the error on that example:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\left[R_t^\lambda - V_t(s_t)\right]\nabla_{\vec{\theta}_t} V_t(s_t)$$

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t$$

$$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$$

$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} V_t(s_t)$$

On-line Gradient-Descent TD(λ)

initialize parameters $\vec{\theta}$ arbitrarily
repeat (for each epiode):
  $\vec{e} = 0$
  $s \leftarrow$ initial state of episode
    repeat (for each step of episode):
    $a \leftarrow$ action given by π for s
    eake action a, observe reward r and next state s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $\vec{e} \leftarrow \gamma\lambda\vec{e} + \nabla_{\vec{\theta}} V(s)$
    $\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
    $s \leftarrow s'$
  until s is terminal

Linear Methods

- assume that V_t is a linear function of the parameter vector
- column vector of *features* $\vec{\Phi}_s$ for every state s
- same number of components as $\vec{\theta}_t$

- $V_t(s) = \vec{\theta}_t^{\,T} \vec{\Phi}_s = \sum_{i=1}^{n} \theta_t(i) \Phi_s(i)$

  $\nabla_{\vec{\theta}_t} V_t(s) = \vec{\Phi}_s$

- only one optimum $\vec{\theta}$, any method guaranteed to converge will converge to the (global) optimum
- note: the feature functions $\Phi(s)$ can be highly non-linear in s

Value prediction with function approximation

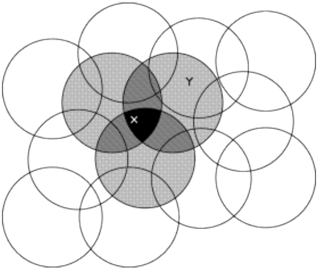
$$MSE(\vec{\theta}_t) = \sum_{s \in S} P(s) \left[V^{\pi}(s) - V_t(s) \right]^2,$$

P is a distribution weighting the errors of different states

- P usually also gives the distribution of states used for training,
- therefore, also the states used for backups
- if we want to minimize error for some states: train the function approximator on this distribution

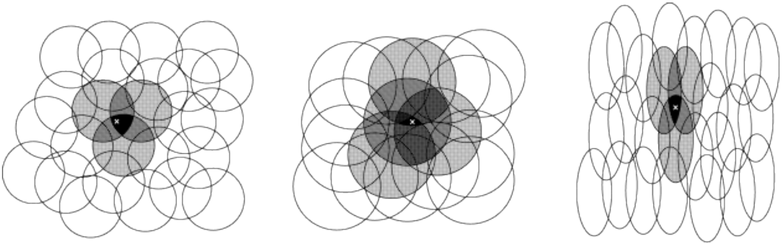
- P may depend on the current policy π: the *on-policy distribution*
- minimizing *MSE* related to a good policy at all?

Example: Coarse coding



- 2D continuous state-space
- circular *binary-features*: $\Phi(x) = 1$ if x inside circle, 0 otherwise
- *receptive field* of a feature
- *coarse coding*: representing a state with a number of overlapping binary features

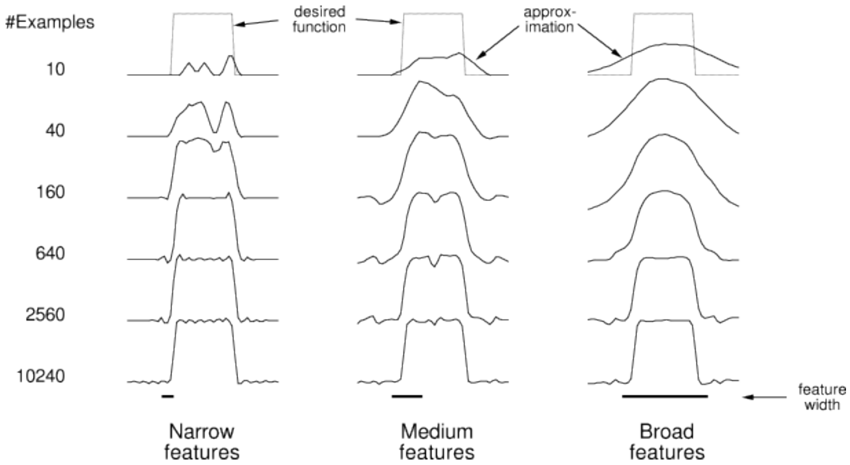
Generalization in linear function approximation



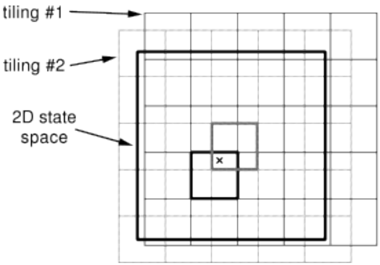
a) Narrow generalization   b) Broad generalization   c) Asymmetric generalization

- one weight parameter θ_j for each feature (circle) Φ_j
- training at a state s affects the weights of all features that cover s
- generalization occurs on the union of the affected features
- the size (and shape) of the functions determine the detail that can be represented and learned

Effect of feature-width on generalization



Tile Coding

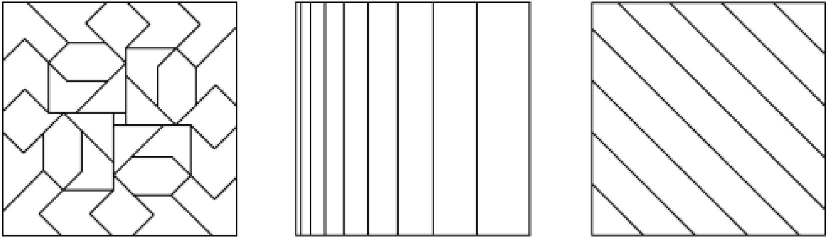


Shape of tiles ⇒ Generalization

#Tilings ⇒ Resolution of final approximation

- receptive fields of the features selected to cover the input space
- exhaustive partitions of the input space, called a *tiling*
- each *tile* is the receptive field for one binary feature
- examples: a regular grid, overlapping (shifted) grids, etc.
- efficient: only sum over "active" tiles, most gradients are 0

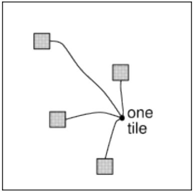
None-uniform grids



a) Irregular b) Log stripes c) Diagonal stripes

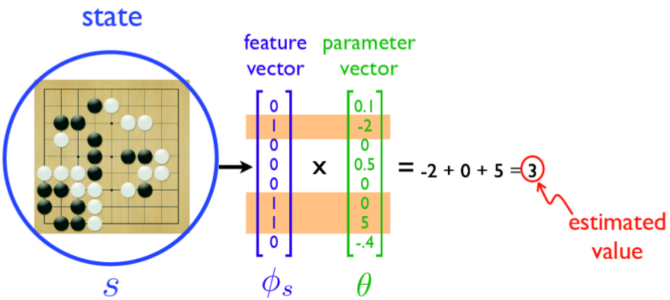
- tilings don't need to be regular grids
- use tile shapes and sizes adapted to the problem at hand
- e.g., use finer tiles where the state-space requires better precision
- e.g., (c) above will promote generalization along one diagonal

Tile coding with hashing



- reduce memory requirements using *hashing*
- only allocate/use memory-cells encountered so far
- represent large (unimportant) parts of the state-space with few large tiles, but add more tiles for the important parts (or dimensions) of the state space

Example: Go



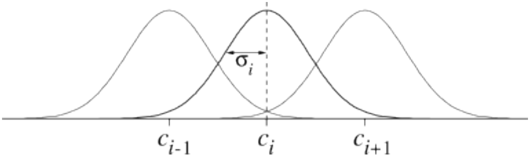
10^{35} states, 10^5 binary features and parameters

(Sutton, presentation at ICML 2009)

Radial basis functions

- RBFs are the natural generalization of coarse-coding to continuous-value features, representing various degrees 0..1 to which a feature is present
- Gaussian $\Phi_s(i)$ functions measure the distance between state s and the feature center c_i:

  $\Phi_s(i) = \exp\left(-\frac{||s-c_i||^2}{2\sigma_i^2}\right)$



Control with Function Approximation

How to improve the policy π? Again, one idea is to follow the GPI pattern: approximate $Q(s,a)$ instead of $V(s)$, then change the policy by greedification.

- build $Q(s,a)$ as a function with parameter vector $\vec{\theta}$.
- general gradient-descent update for action-value prediction is:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\left[v_t - Q_t(s_t,a_t)\right]\nabla_{\vec{\theta}_t} Q_t(s_t,a_t).$$

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\delta_t\vec{e}_t,$$
$$\delta_t = r_{t+1} + \gamma Q_t(s_{t+1},a_{t+1}) - Q_t(s_t,a_)),$$
$$\vec{e}_t = \gamma\lambda\vec{e}_{t-1} + \nabla_{\vec{\theta}_t} Q_t(s_t,a_t).$$

Control with Function Approximation

Two examples:

- Sarsa(λ) (on-policy)
- Q(λ) (off-policy)
- linear, gradient-descent function approxiation (binary features)
- ϵ-greedy action selection

- compute sets of features $\mathcal{F}_a$ corresponding to the current state s and all possible actions a
- use of eligibility traces more complex than in the tabular case
- each time a state encountered that has feature i, the trace for feature i is set to 1 (instead of being incremented by 1)

Linear Gradient-Descent Sarsa(λ) (1)

with binary features and ϵ-greedy policy

initialize parameters $\vec{\theta}$ arbitrarily
repeat (for each epiode):
  $\vec{e} = 0$
  $s, a \leftarrow$ initial state and action of episode
  $\mathcal{F}_a \leftarrow$ set of features present in s, a
  repeat (for each step of episode):
    for all $i \in \mathcal{F}_a$:
      $e(i) \leftarrow e(i) + 1$ (accumulating traces)
      or $e(i) \leftarrow 1$ (replacing traces)
    take action a, observe reward r and next state s'
    $\delta \leftarrow r - \sum_{i \in \mathcal{F}_a} \theta(i)$
    ...

Linear Gradient-Descent Sarsa(λ) (2)

with probability $1 - \epsilon$:
  for all $a \in \mathcal{A}(s)$: // *greedy actions*
    $\mathcal{F}_a \leftarrow$ set of features present in s, a
    $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
  $a \leftarrow \arg\max_a Q_a$
else // *exploration action with probability* ϵ
  $a \leftarrow$ a random action $\in \mathcal{A}(s)$
  $\mathcal{F}_a \leftarrow$ set of features present in s, a
  $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
$\delta \leftarrow \delta + \gamma Q_a$
$\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$
$\vec{e} \leftarrow \gamma \lambda \vec{e}$
until s is terminal

Off-Policy Gradient-Descent Watkins's Q(λ) (1)

binary features, ϵ-greedy policy, accumulating traces

initialize parameters $\vec{\theta}$ arbitrarily

repeat (for each epiode):

- $\vec{e} = 0$
- $s, a \leftarrow$ initial state and action of episode
- $\mathcal{F}_a \leftarrow$ set of features present in s, a
- repeat (for each step of episode):
  - for all $i \in \mathcal{F}_a : e(i) \leftarrow e(i) + 1$
  - take action a, observe reward r and next state s'
  - $\delta \leftarrow r - \sum_{i \in \mathcal{F}_a} \theta(i)$
  - for all $a \in \mathcal{A}(s)$:
    - $\mathcal{F}_a \leftarrow$ set of features present in s, a
    - $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
  - ...

Off-Policy Gradient-Descent Watkins's Q(λ) (2)

...

$\delta \leftarrow \delta + \gamma \max_a Q_a$

$\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$

$\vec{e} \leftarrow \gamma\lambda\vec{e}$

with probability $1 - \epsilon$:

for all $a \in \mathcal{A}(s)$:

$Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$

$a \leftarrow \arg\max_a Q_a$

$\vec{e} \leftarrow \gamma\lambda\vec{e}$

else

$a \leftarrow$ a random action $\in \mathcal{A}(s)$

$\vec{e} \leftarrow 0$

until s is terminal

Example: Mountain-car (repeated)

- underpowered car should climb a mountain-slope
- simplified physics model
- actions are full-throttle $a \in \{-1, 0, +1\}$
- but constant $a = +1$ is not sufficient to reach the summit
- car must go backwards first a bit or even oscillate to build sufficient momentum to climb the mountain

- simple example of problems where the agent cannot reach the goal directly, but must explore intermediate solutions that seem counterintuitive
- remember: typical example of *delayed reward*

Mountain-car: setup and reward function

- $+100$ reward for reaching the mountain-summit
- -1 reward for every timestep without reaching the summit

- simplified physics model:

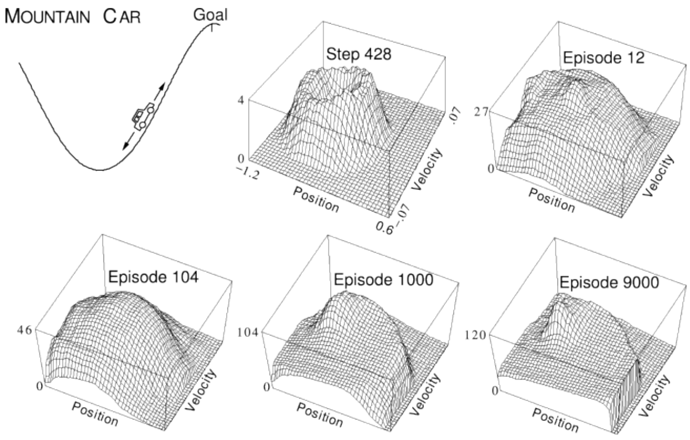
  $x_{t+1} = x_t + \dot{x}_t$

  $\dot{x}_{t+1} = \dot{x}_t + 0.001 a_t + -0.0025 \cos(3 x_t)$

  and $x, \dot{x}$ are clipped to a certain range

- using regular grid-tiling
- every episode is terminated after 1000 timesteps

Mountain-car: cost-to-go function $-\max_a Q_t(s, a)$



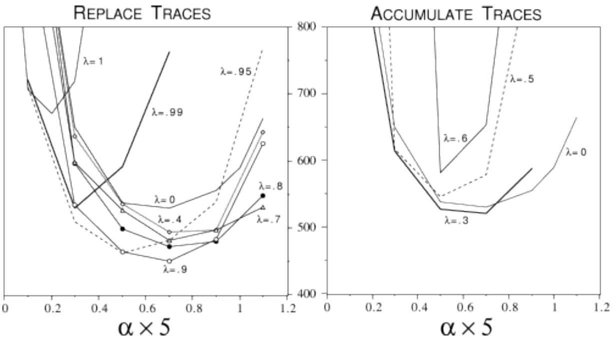
Details: Sutton and Barto, chapter 8.10

Mountain-car: analysis

- use optimistic initial estimates to encourage exploration
- no success during the first episodes:
  $Q(s, a)$ all negative initially
- visited states valued worse than unexplored states

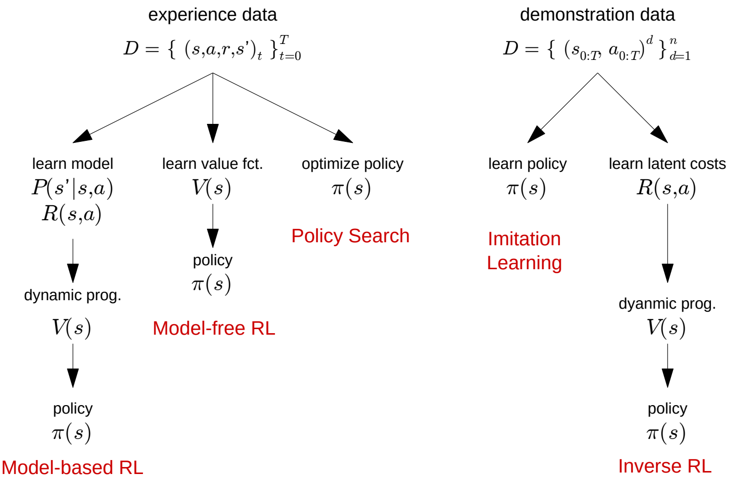
Mountain-car:

Steps per episode
averaged over
first 20 trials
and 30 runs



- effect of α, λ, and the kind of traces on the early performance of the mountain-car task.

RL taxonomy


experience data
$D = \{ \ (s,a,r,s')_t \ \}_{t=0}^{T}$
learn model
$P(s'|s,a)$
$R(s,a)$
dynamic prog.
$V(s)$
policy
$\pi(s)$
Model-based RL
learn value fct.
$V(s)$
policy
$\pi(s)$
Model-free RL
optimize policy
$\pi(s)$
Policy Search
demonstration data
$D = \{ \ (s_{0:T}, \ a_{0:T})^d \ \}_{d=1}^{n}$
learn policy
$\pi(s)$
Imitation Learning
learn latent costs
$R(s,a)$
dyanmic prog.
$V(s)$
policy
$\pi(s)$
Inverse RL

Learning in Policy Space

- convergence proofs are nice, but ...
- ... many tasks don't require the optimal policy
- ... survival of the learner also is important

- many applications cannot afford to explore the full state-space, because there exist "deadly" parts
- more interested in a good policy than the optimal one π^*
- concentrate on those parts of the state-space that are safe
- experience shows that value-function gradients are often unstable

Policy Gradient Methods

- Policy Gradient Methods
  - Finite Difference Methods (FD)
  - Likelihood Ratio Methods
    - 'Vanilla' Policy Gradients (VPG)
    - 'Natural' Policy Gradients(NPG)

Policy Gradient Methods

- consider randomized policy $\mu(s,a) = \Pr(a|s)$
  (deterministic policy is a special case)
- performance measure is

$$J(\mu) = E_\mu\left[r_0 + \gamma r_1 + \gamma^2 r_2 + \dots\right]$$

- policy $\mu_\theta(s,a)$ is parameterized by a parameter space $\theta \in IR^d$
- parametric performance measure becomes

$$J(\theta) = E_\theta\left[r_0 + \gamma r_1 + \gamma^2 r_2 + \dots\right]$$

- iterative solution using gradient-descent algoriths

Policy Gradient Update

- policy gradient update:

$$\theta_{k+1} = \theta_k + \alpha \nabla_\theta J(\theta_k)$$

- guarantee for performance improvements?!

$$J(\theta_{\mu'}) \geq J(\theta_\mu) \Rightarrow \mu' \text{ at least better or equal to } \mu$$

- approximate the gradient using supervised learning
- collect data $\mathcal{D} = \{\delta\theta_i, \delta J_i\}$ (that is, sample gradients).
  - perturb the parameters: $\theta + \delta\theta$
  - apply resulting new policy $\mu(\theta + \delta\theta)$ to get $\delta J_i = J)(\theta + \delta\theta) - J(\theta)$
- finite difference gradient estimation (FD):

$$g_{\mathrm{FD}}(\theta) = (\Delta\Theta^T \Delta\Theta)^{-1} \Delta\Theta \Delta J$$

LSPI: Least Squares Policy Iteration

- gradient-descent methods are sensitive to the choice of learning rates and initial parameter values.
- calculating a policy from the value function is difficult

- least-square temporal difference (LSTD) method: LSPI.
- Bellman residual minimization
- least squares fixed-point approximation

LSPI: Least Squares Policy Iteration

the Q-function for a given policy π fulfils for any s, a:
$Q^\pi(s,a) = R(s,a) + \gamma \sum_{s'} P(s'|a,s) Q^\pi(s', \pi(s'))$
if we have n data points $D = \{(s_i, a_i, r_i, s_i')\}_{i=1}^n$, we require that this equation holds (approximately) for these n data points:
$\forall i : Q^\pi(s_i, a_i) = r_i + \gamma Q^\pi(s_i', \pi(s_i'))$
written in vector notation: $Q = R + g\overline{Q}$ with N-dim data vectors Q, R, $\overline{Q}$

- written as optimization: minimize the *Bellman residual error*
  $L(Q^\pi) = ||R + \gamma P \Pi Q^\pi - Q^\pi||$ (true residual)
  $= \sum_{i=1}^n \left[Q^\pi(s_i, a_i) - r_i - \gamma Q^\pi(s_i', \pi(s_i')) \right]^2 = ||R - Q + \gamma \overline{Q}||^2$

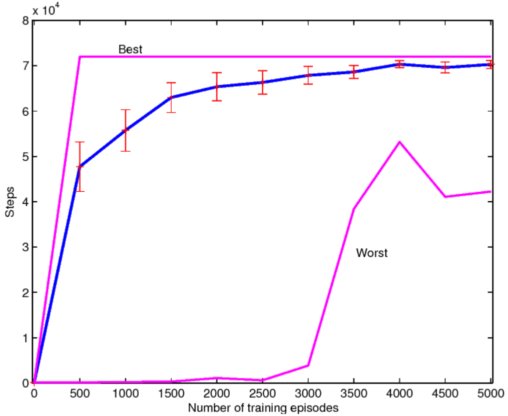
Example: Learning how to ride a bicycle

- states $= \{\theta, \dot{\theta}, \omega, \dot{\omega}, \ddot{\omega}, \psi\}$
  θ is the angle of the handlebar,
  ω the vertical angle of the bicycle,
  ψ is the angle of the bicycle to the goal.
- actions: $\{\tau, \nu\}$
  $\tau \in \{-2, 0, 2\}$ the torque applied to the handlebar,
  $\nu \in \{-0.02, 0, 0.02\}$ the displacement of the rider.

- again, simplified physics model for simulation
- choose function approximation, run RL...
  $(1, \omega, \dot{\omega}, \omega^2, \omega\dot{\omega}, \theta, \dot{\theta}, \theta^2, \theta\dot{\theta}, \omega\theta, \omega\theta^2, \omega^2\theta, \psi, \psi^2, \psi\theta, \overline{\psi}, \overline{\psi}^2, \overline{\psi}\theta)$

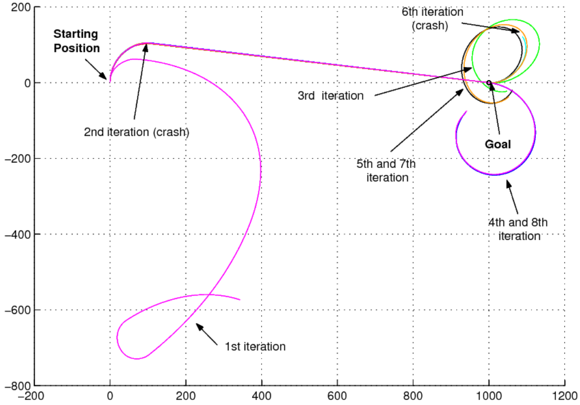
ipvs.informatik.uni-stuttgart.de/mlr/wp-content/uploads/2013/11/04-FunctionApproximation.pdf

Learning how to ride a bicycle



Lagoudakis & Parr, JMLR 2003

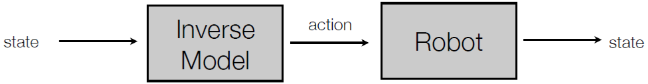
Learning how to ride a bicycle



Lagoudakis & Parr, JMLR 2003

Application: robot learning in joint-space

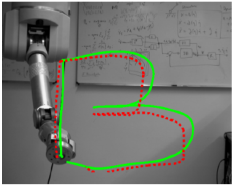
- learn a model for accurate control in joint-space
- if we could map states to the required actions, this could be executed on the robot immediately:



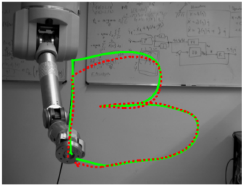
Application: model-based robot motion

- learn a model for accurate control in joint-space
- compare with traditionally modeled solution
- compliant, low-gain control of fast and accurate motions

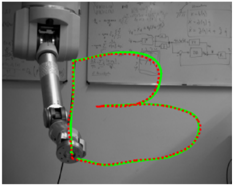
Analytical Rigid-Body Model with CAD data



Offline Trained

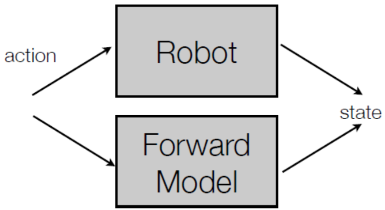


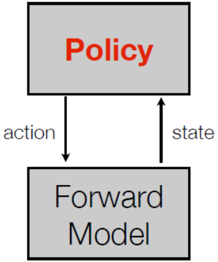
Online Trained



Nguyen-Tuong Peters, IROS 2008

Learning a forward model





1 learn an forward model of the system dynamics
2 use an optimal-control model to derive the policy

Apprenticeship Learning

- learning from a teachers' demonstration
  - demonstration on the target system
  - demonstration on another system
  - with or without model of the target system
- one of the hot topics in RL today
- several recent examples: robot table-tennis playing, autonomous car-driving, helicopter aerobatics
- aka *inverse RL*: given a demonstration (= policy), derive the teachers' reward function, then reproduce on the target system

Apprenticeship Learning: Motivation

- see original paper

Current Research Areas

- *hierarchical reinforcement learning*
- inverse RL: learning from demonstrations
  - introduction to inverse RL
  - inverse RL vs. behavioral cloning
  - IRL algorithms

- learning high-DOF problems (humanoids $\approx$ 70-DOF)
- combining learning and planning

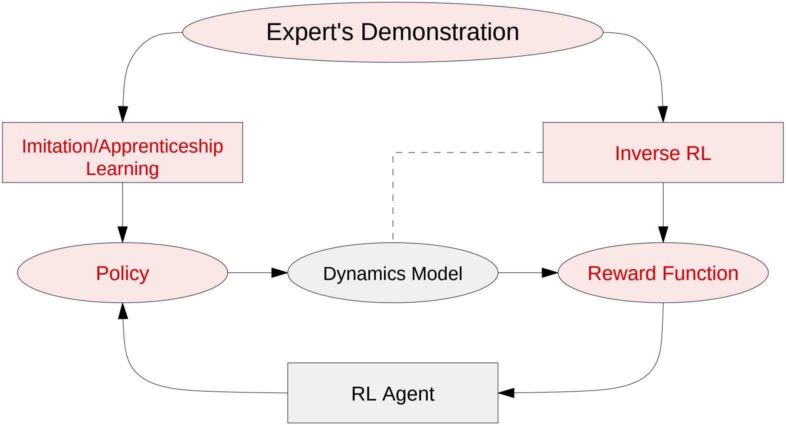
Inverse RL: informal definition

Given:

- measurements of an agent's behaviour π over time (s_t, a_t, s'_t),
- possibly, the transition model $T(s, a, s')$.
- not given: the reward model.

Goal: find the reward funtion $R^{\pi}(s, a, s')$

Inverse RL: big picture



Inverse RL: problem formulation

Given:

- state space S, action space A
- transition model $T(s, a, s') = P(s'|s, a)$
- not given reward function $R(s, a, s')$
- teacher's demonstration (from teacher's policy π^*): $s_0, a_0, s_1, a_1, \ldots,$

Inverse Reinforcement Learning (IRL):

- recover R

Apprenticeship learning via IRL:

- use R to compute a good policy π

Behaviour cloning:

- using supervised learning to learn the teacher's policy

IRL vs. Behavioral cloning

- behavioral cloning: formulated as a supervised learning problem (e.g. using SVM, NN, deep learning, ...)
  - given $(s_0, a_0), (s_1, a_1), \ldots$ generate from a policy π^*
  - estimate a policy mapping from s to a
- this can only mimic the demonstrated trajectories of the teacher
  - can not change goal/destination
  - can not handle non-Markovian enviroment
- IRL vs. behaviour cloning is R^* vs π^*.

Inverse RL: mathematical formulation

Given:

- state space S, action space A
- transition model $T(s, a, s') = P(s'|s, a)$
- not given reward function $R(s, a, s')$
- teacher's demonstration (from teacher's policy π^*): $s_0, a_0, s_1, a_1, \ldots,$

Find reward function R such that

$$E\left[\sum_{t=0}^{\infty} \gamma^t R^*(s_t)|\pi^*\right] \geq E\left[\sum_{t=0}^{\infty} \gamma^t R^*(s_t)|\pi\right], \forall \pi$$

Inverse RL: problems

Find reward function R such that

$$E\left[\sum_{t=0}^{\infty} \gamma^t R^*(s_t) | \pi^*\right] \geq E\left[\sum_{t=0}^{\infty} \gamma^t R^*(s_t) | \pi\right], \forall \pi$$

- $R = 0$ is a solution for this equation ...
- solution is not unique, multiple R^* can satisfy the equation
- teacher's π^* in only given partially, so unclear how to evaluate the expectation terms.

Applications

References

Andrew Y. Ng, Stuart J. Russell: Algorithms for Inverse Reinforcement Learning. ICML 2000: 663-670

Pieter Abbeel, Andrew Y. Ng: Apprenticeship learning via inverse reinforcement learning. ICML 2004

Pieter Abbeel, Adam Coates, Morgan Quigley, Andrew Y. Ng: An Application of Reinforcement Learning to Aerobatic Helicopter Flight. NIPS 2006: 1-8

Adam Coates, Pieter Abbeel, Andrew Y. Ng: Apprenticeship learning for helicopter control. Commun. ACM 52(7): 97-105 (2009)

Summary: Reinforcement Learning

- agent in a (known or unknown) environment
- agent takes actions, receives a scalar reward
- learn a policy that maximizes accumulated reward
- learn how to avoid bad parts of the state-space

- in-between unsupervised and supervised learning
- learn how to reach delayed rewards
- exploration vs. exploitation dilemma

- very general setup, many application areas

Markov Decision Process

- MDP:
  - states $s \in S$
  - actions $a \in A(s)$
  - immediate reward r after taking action a in state s
  - transition probabilities $P^a_{ss'}$
  - reward probabilities $R^a_{ss'}$
  - accumulated return $R_t = \sum_{i=0}^{t} \gamma^i r_i$
- Markov property/assumption
- goal: maximize return R
- sub-goal: learn policy π that leads to good actions

Value functions

- assigning values to states: estimation of future rewards
  - $V(s)$ state value function
  - $Q(s, a)$ state-action value function
- Bellman equation: relating $V(s)$ to $V(s')$
  - backup-operations based on the Bellman equation
- optimal value-functions $V^*(s)$ and $Q^*(s, a)$
- greedy policy π^* derived from V^* is optimal

Algorithms

- Dynamic Programming
- Policy evaluation and policy iteration
- Monte-Carlo methods
- Temporal-Difference idea, SARSA and Q-learning
- TD(λ) methods

- combining value-functions with function approximation

- direct policy search methods