Reinforcement Learning (3)

Machine Learning 64-360, Part II

Norman Hendrich

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

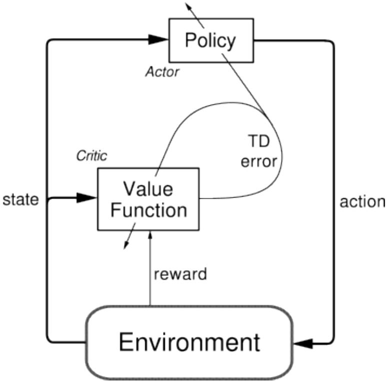
30/06/2014

Contents

Actor-Critic Methods

- separate "blocks" to represent the policy independent from the value function
- policy structure called the *actor*
- value estimation called the *critic*, usually building $V(s)$, not $Q(s, a)$

- learning is on-policy: the critic must learn about and critique the policy currently followed by the actor



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
- when a TD error occurs, only the eligible states or actions are updated

n-step TD prediction

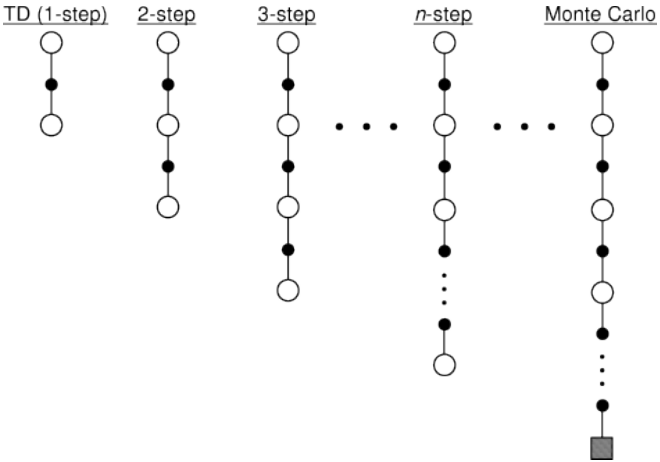
- consider estimating $V^{\pi}(s)$ from sample episodes generated following policy π

- 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 on 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?

- all 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.

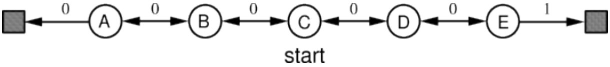
The *n*-step backup

- 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.

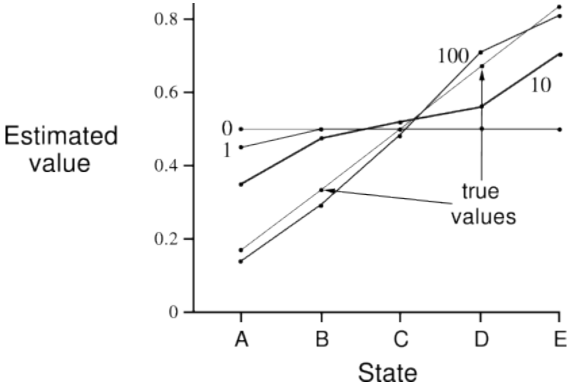
Example: *n*-step TD on Random Walk



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

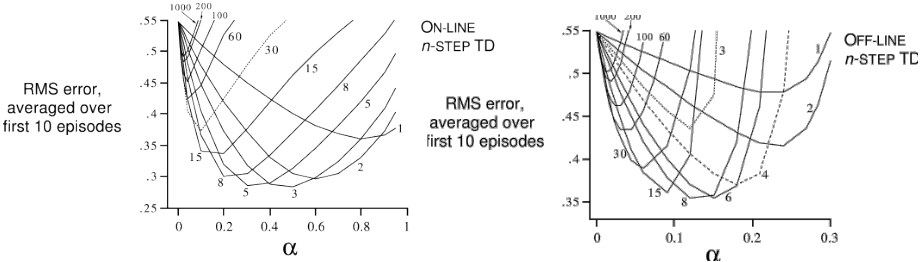
- in this case, $V(s)$ is just the probability of terminating on the right when starting in S: $\{0, \frac{1}{6}, \frac{2}{6}, \frac{3}{6}, \frac{4}{6}, \frac{5}{6}, 1\}$

Example: TD(0) on Random Walk



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

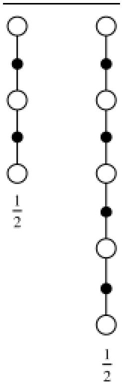
Example: *n*-step TD on Random Walk



- *n*-step TD methods on the 19-state random walk
- performance measured as RMS error
- 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

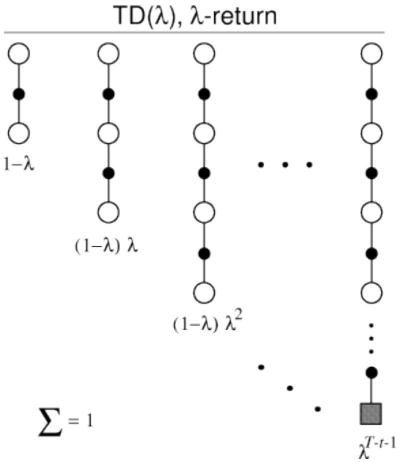


- a complete new class of algorithms
- combining properties of the different individual methods

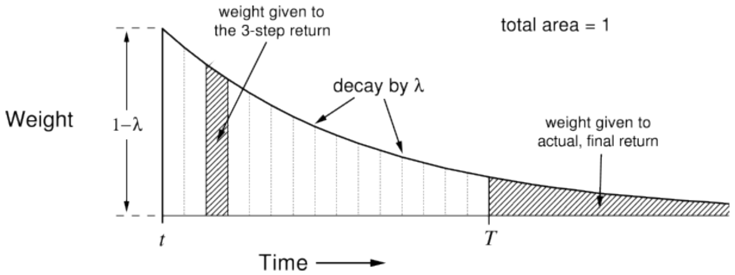
Backup diagram for TD(λ)

- 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

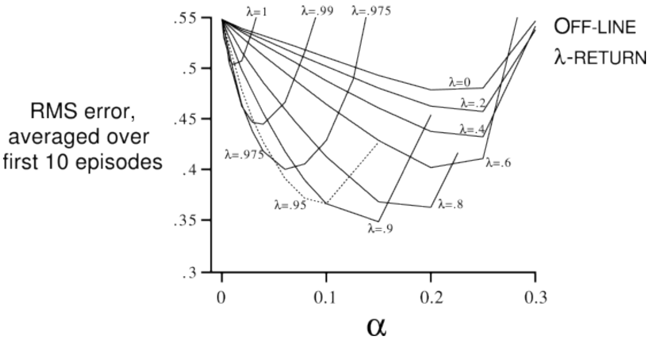


$$R_t^\lambda = (1-\lambda)\sum_{n=1}^{T-t-1} \lambda^{n-1} R_t^{(n)} + \lambda^{T-t-1} R_t$$

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

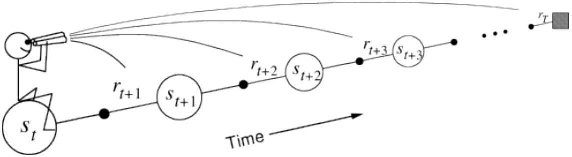
$\lambda = 0$: reduces to $R_t^{(1)}$, so TD(0)

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

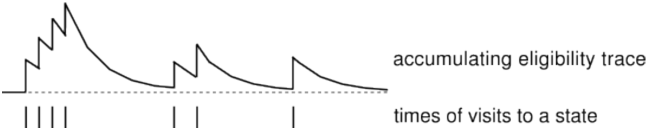


- from each state s visited, look forward in time to all future rewards, and decide how best to combine them.
- problem: this is hard to implement, using at each step knowledge of what will happen many steps later ...

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 therms of γ^{λ}

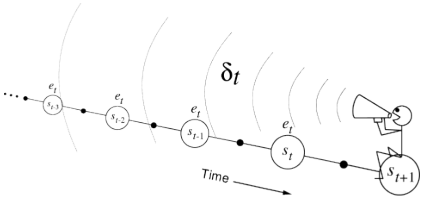
- the traces indicate the degree to which each state is *eligible* to change during learning
- for example, the TD "error" for state-value prediction is
  $\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$

- 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 byp 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 change less
- $\lambda = 1$: credit given to earlier states fall by γ at each step, giving Monte Carlo for $\gamma = 1$, also called TD(1)

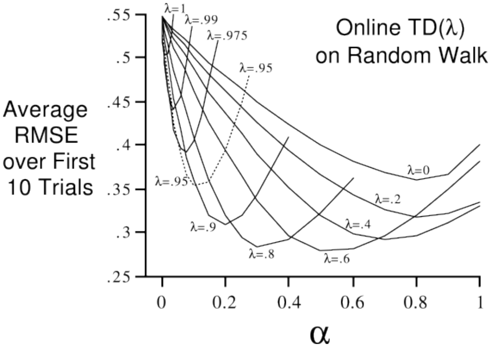
Equivalence of Forward and Backward View

- 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:
- learning $Q(s, a)$ instead of learning $V(s)$?

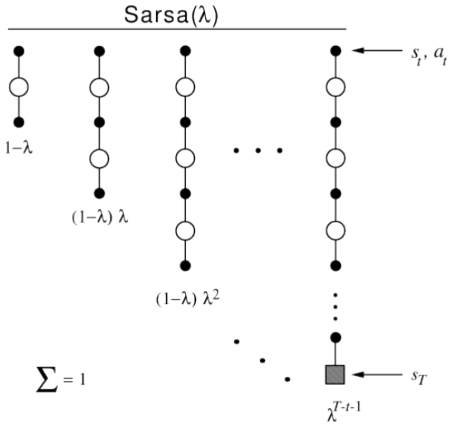
- $Q_{t+1}(s, a) = Q_t(s, a) + \alpha \delta_t e_t(s, a)$, for all (s, a)

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

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

- from time to time, improve policy π using greedification

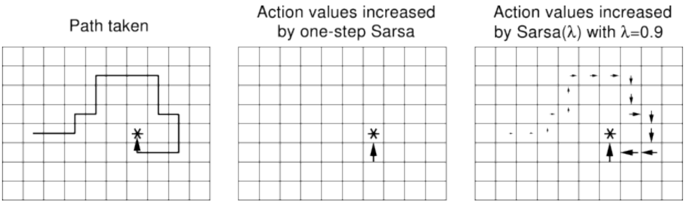
Sarsa(λ) backup diagram



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)



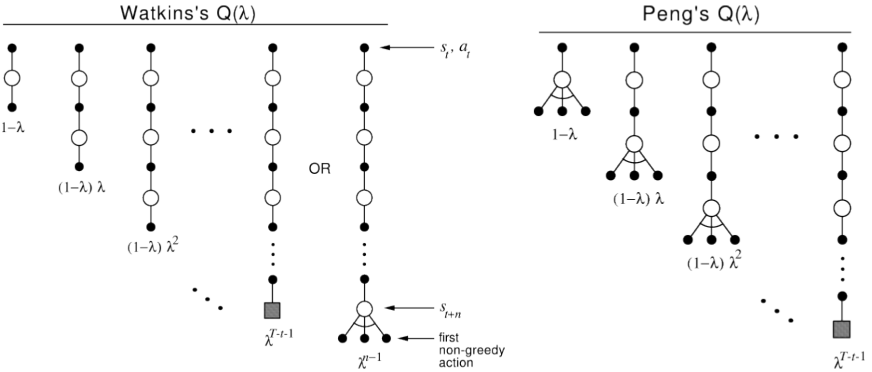
- example path of the learner, ending in goal state '*'
- TD(0) methods will only update the single $Q(s, a)$ for the immediately preceding state
- eligibility-trace-methods update many $Q(s, a)$ values weighted by relevance

Advanced topics

- $Q(\lambda)$: 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

Generalization and function approximation

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

- discrete state s and action a spaces
- state-space small enough for in-memory representation
- 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*

Generalization and function approximation

Basic idea: represent continuous state-space or state-action-space using function approximation with parameters $\vec{\theta} \in \{\Theta\}$.
Then, use RL-algorithms to adjust the parameters θ.

All common function approximation methods can be used:

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

but the context is usually high-DOF problems.

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 $V(\theta(\vec{s}))$

- measure approximation error using suitable loss-functions,

  e.g. mean-squared error:

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

Reminder: least-squares cost function

The classical cost function is the one of least-squares

$$J = \frac{1}{2}\sum_{i=1}^{N}(y_i - \mathbf{f}_\theta(\mathbf{x}_i))^2.$$

Using

$$\begin{aligned} \mathbf{\Phi} &= \left[\ \phi(\mathbf{x}_1),\ \ \phi(\mathbf{x}_2),\ \ \phi(\mathbf{x}_3),\ \ \ldots,\ \ \phi(\mathbf{x}_n)\ \right]^T, \\ \mathbf{Y} &= \left[\ y_1,\ \ y_2,\ \ y_3,\ \ \ldots,\ \ y_n\ \right]^T. \end{aligned}$$

we can rewrite it as

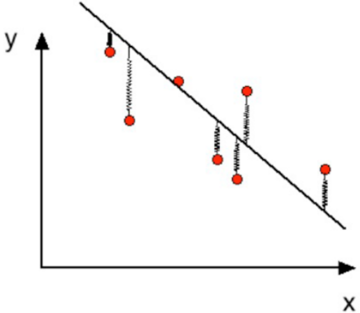
$$J = \frac{1}{2}(Y - \Phi\theta)^T(Y - \Phi\theta).$$

and solve it

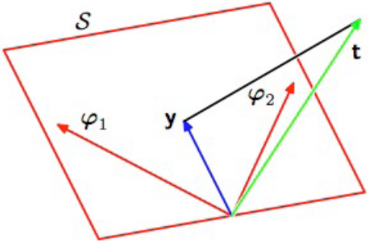
$$\theta = (\Phi^T\Phi)^{-1}\Phi^T Y$$

this and many of the next slides: (Abbeel & Peters, ICRA-RL tutorial 2012)

Interpretation of the least-squares cost function

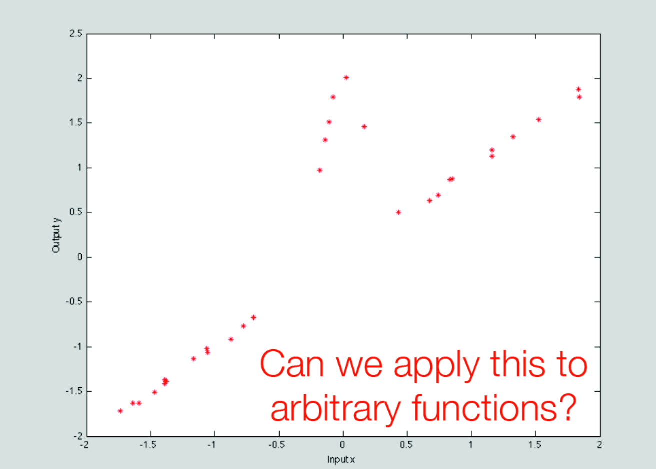


physical interpretation

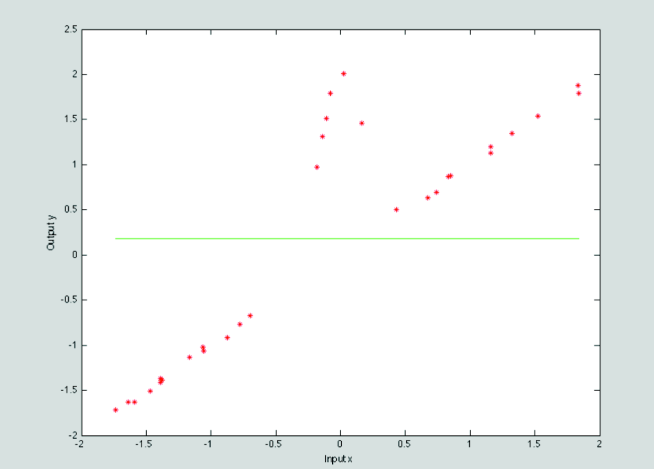


geometrical interpretation

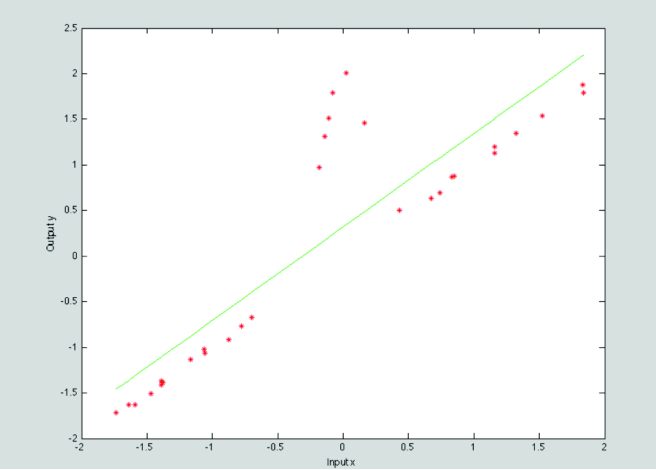
Function approximation: example data



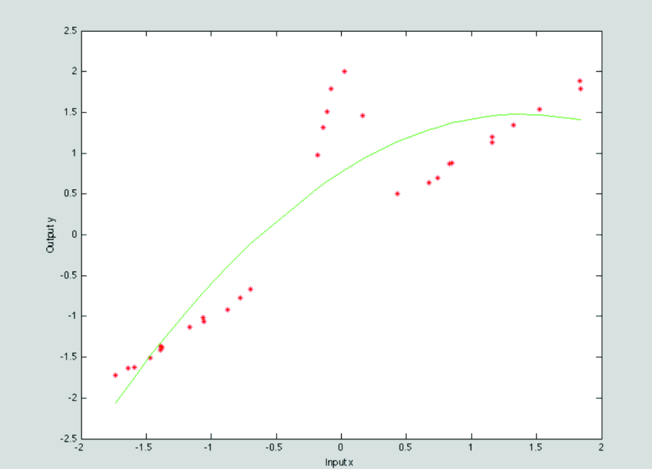
Fitting an easy model: $n = 0$



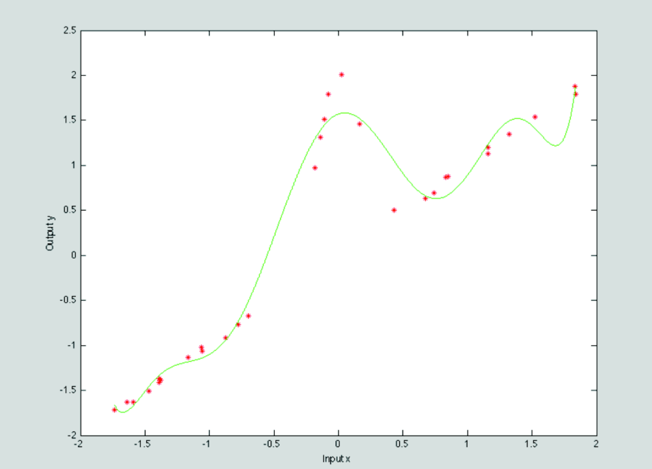
More features: $n = 1$



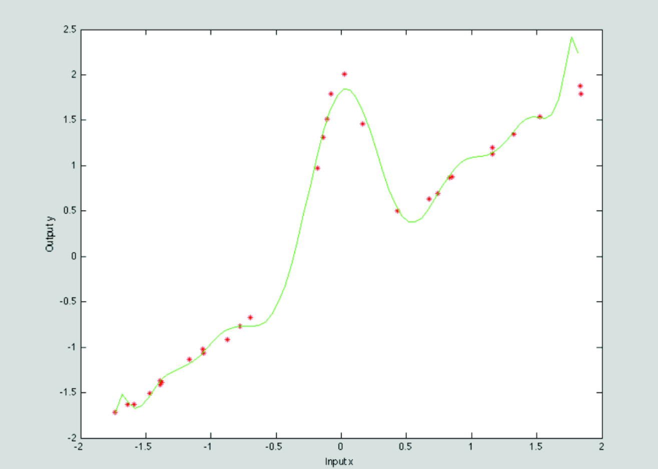
More features: $n = 2$



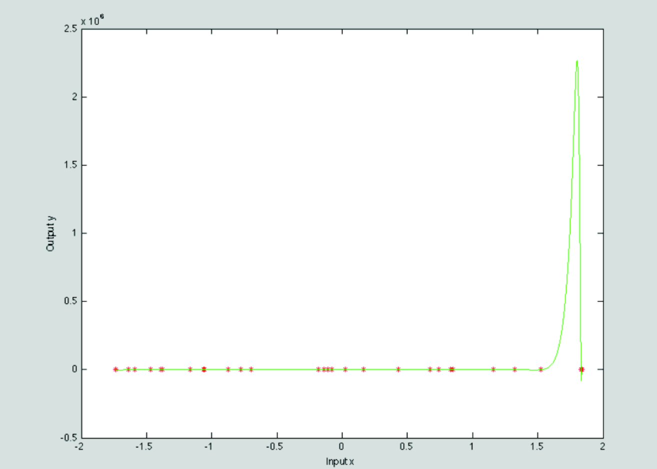
More features: $n = 8$



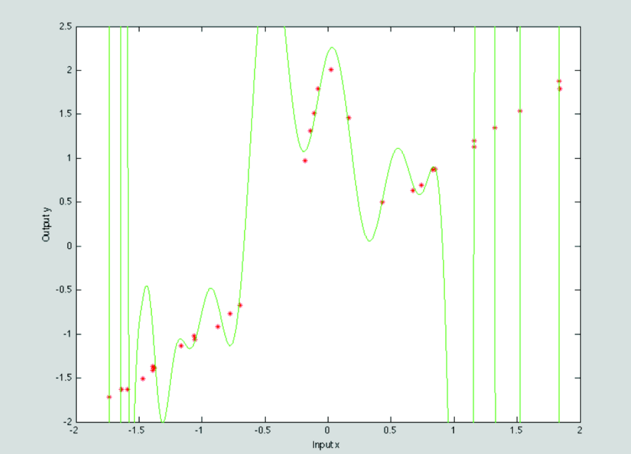
More features: $n = 15$



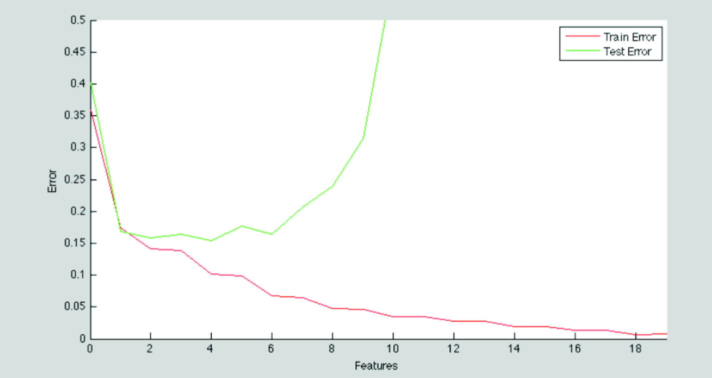
More features: $n = 200$



More features: $n = 200$



Training vs. test error



- remember the *magic tool*: leave-one-out cross-validation

What was the problem with $n = 200$?

- polynom with degree $n = 200$ has too many parameters
- polynoms not well-behaved for $x \to \infty$
- overfitting the data completely
  - too many parameters
  - too large parameters

How to avoid this?

We could punish the size of the parameters (Complexity Control):

$$J = \frac{1}{2}(Y - \Phi\theta)^T(Y - \Phi\theta) + \theta^T \mathbf{W}\theta$$

This yields Ridge Regression

$$\theta = (\Phi^T\Phi + \mathbf{W})^{-1}\Phi^T Y$$

with

$$\mathbf{W} = \lambda\mathbf{I} \qquad \lambda < 10^{-6}$$

The probabilistic interpretation is called Maximum-A-Priori:

$$\text{argmax}_\theta p(\mathcal{D}|\theta)p(\theta) \qquad p(\theta) = \mathcal{N}(0, \mathbf{W})$$

Full Bayesian Regression?

- Full Bayesian Regression wants to

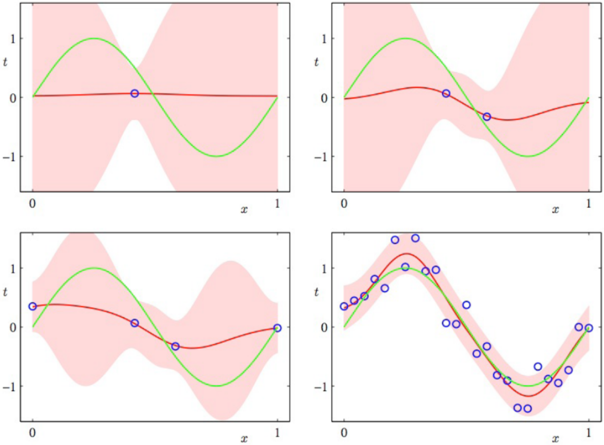
$$p(y|\mathcal{D},\mathbf{x}) = \int p(y|\mathbf{x},\theta)p(\theta|\mathcal{D})d\theta$$

- *Intuition*: If you assign each estimator a “probability of being right”, the average of these estimators will be better than the single one.

- Yields:

$$p(y|\mathcal{D},\mathbf{x}) = \mathcal{N}\left(\phi(\mathbf{x})^T\left(\frac{\lambda}{\beta}\mathbf{I}+\Phi^T\Phi\right)^{-1}\Phi^T\mathbf{Y}, \frac{1}{\beta}\left(1+\phi(\mathbf{x})^T\left(\frac{\lambda}{\beta}\mathbf{I}+\Phi^T\Phi\right)^{-1}\phi(\mathbf{x})\right)\right)$$

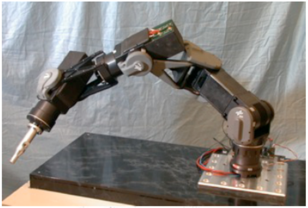
Example

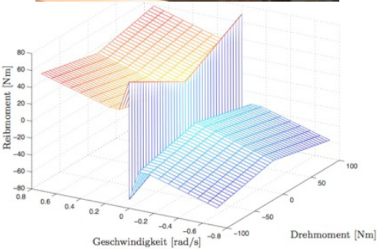


(Bishop 2006)

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

- In most real applications, we know good features.
- However, we almost certainly don't know all features we need.
- ***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 way too many features!





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

Yes, we can!

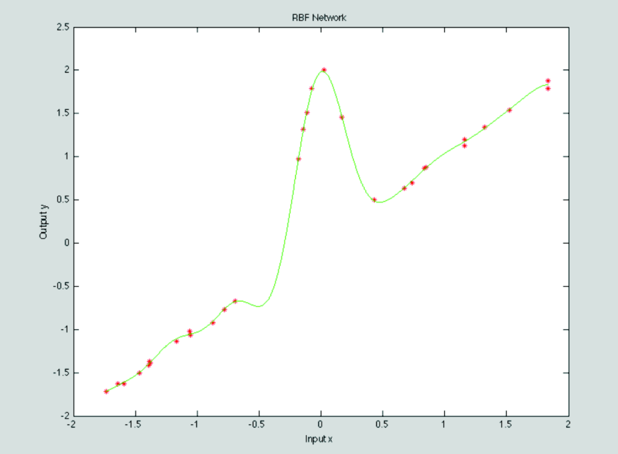
We need to find machine learning approaches that generate the features directly based on data.

Example 1: *Radial basis functions* create an optimal smooth

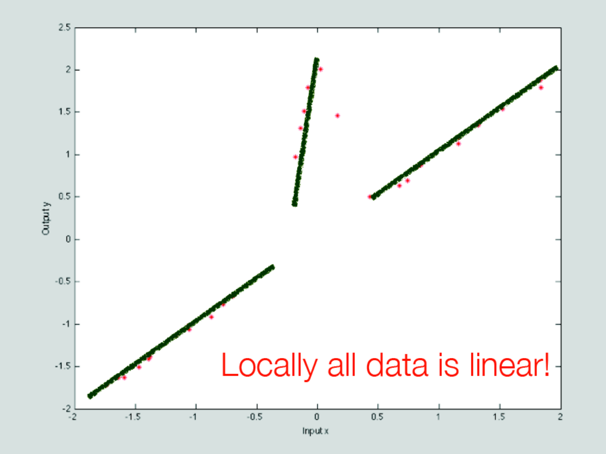
Example 2: *Locally-Weighted Regression* localize in your data and try to interpolate with similar data.

Example 3: *Kernel Regression* find the features by going into *function space* using a *kernel*?

More features: radial-basis functions

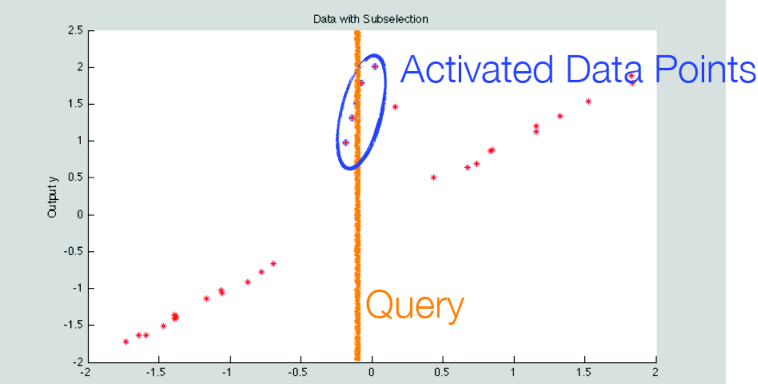


Locally, all data is linear

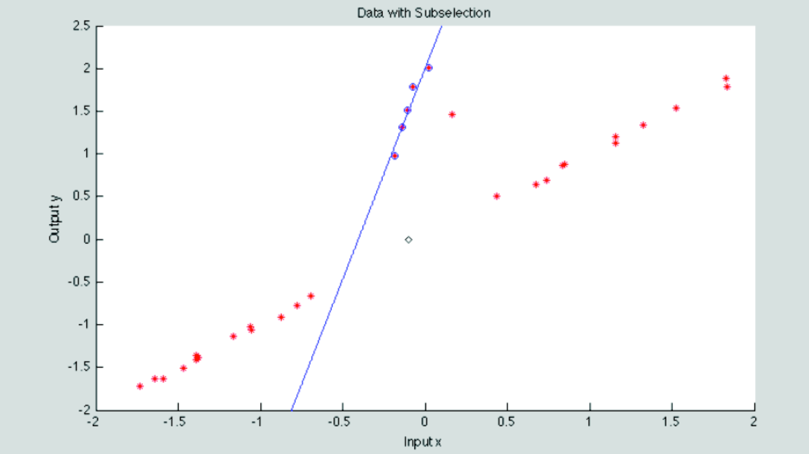


Locally linear solutions

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



Locally linear solutions for a query



Locally linear solutions: cost function

We can formalize this in a cost function. Let us use our on-off function in the cost function and we obtain:

$$J = \frac{1}{2} \sum_{i=1}^{N} w_i(\mathbf{x})(y_i - \mathbf{f}_\theta(\mathbf{x}_i))^2,$$

In matrix form with $W = \mathrm{diag}(w_1, w_2, w_3, \ldots, w_n)$:

$$J = \frac{1}{2}(\mathbf{Y} - \Phi\theta)^T \mathbf{W} (\mathbf{Y} - \Phi\theta),$$

The solution to this problem

$$\theta = (\Phi^T \mathbf{W} \Phi)^{-1} \Phi^T \mathbf{W} \mathbf{Y}.$$

W can be large - don't implement it in MATLAB like this...

Kernel methods

- Let us define the kernels:

$$k(\mathbf{x}, \mathbf{y}) = \phi(\mathbf{x})^T \phi(\mathbf{y}),$$
$$\mathbf{K}_{ij} = k(\mathbf{x}_i, \mathbf{x}_j),$$
$$\mathbf{k}_i = k(\mathbf{x}, \mathbf{x}_i),$$

- Now we can rewrite the equation by

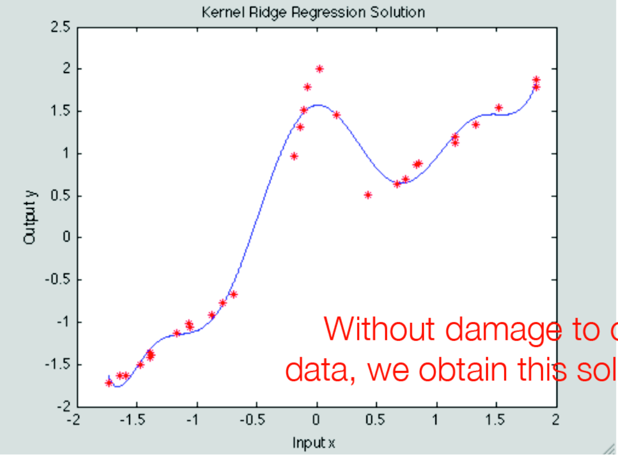
$$y(\mathbf{x}) = \phi(\mathbf{x})^T \Phi(\Phi\Phi^T + \lambda\mathbf{I})^{-1}\mathbf{Y} = \mathbf{k}(\mathbf{K} + \lambda\mathbf{I})^{-1}\mathbf{Y}.$$

- This is called **kernel ridge regression**. Why would this be cool?

- Because we can use another kernel if we are unhappy with our features!

$$k(\mathbf{x}, \mathbf{y}) = \exp\left(\frac{1}{\sigma^2}\|\mathbf{x} - \mathbf{y}\|^2\right).$$

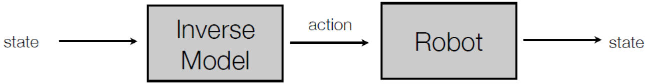
Exponential kernel



Without damage to our data, we obtain this solution!

Application: robot learning in joint-space

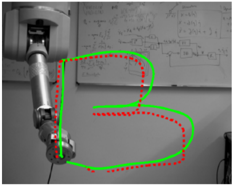
- learn a model for accurate control in joint-space
- if we could map states to the required actions, this could be executed on the robot immediately:



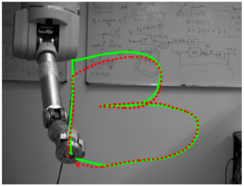
Application: model-based robot motion

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

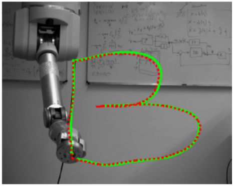
Analytical Rigid-Body Model with CAD data



Offline Trained

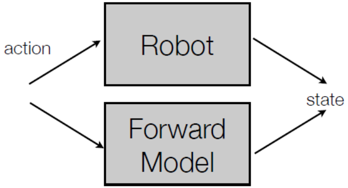


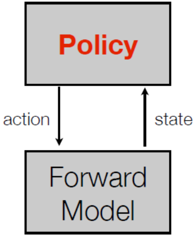
Online Trained



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

Example: Pacman

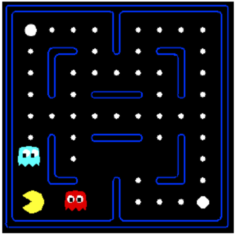
- Let’s say we discover through experience that this state is bad:

- In naïve q learning, we know nothing about this state or its q states:

- Or even this one!

Pacman: Features

- Solution: describe a state using a vector of features
  - Features are functions from states to real numbers (often 0/1) that capture important properties of the state
  - Example features:
    - Distance to closest ghost
    - Distance to closest dot
    - Number of ghosts
    - 1 / (dist to dot)^2
    - Is Pacman in a tunnel? (0/1)
    - …… etc.
  - Can also describe a q-state (s, a) with features (e.g. action moves closer to food)



Pacman: Features

- Using a feature representation, we can write a q function (or value function) for any state using a few weights:

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

$$Q(x,u) = w_1 f_1(x,u) + w_2 f_2(x,u) + \ldots + w_n f_n(x,u)$$

- Advantage: our experience is summed up in a few powerful numbers
- Disadvantage: states may share features but be very different in value!

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:

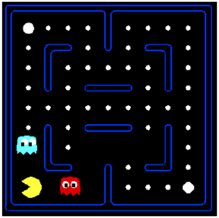
Tabular	Linear
$Q(x,u) \leftarrow$	$\forall i, \quad w_i \leftarrow$
$Q(x,u) + \alpha$ [difference]	$w_i + \alpha$ [difference] $f_i(x,u)$

Pacman: Features

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

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

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



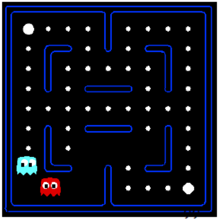
$$Q(s,a) = +1$$

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

$$error = -501$$

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

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



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

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?

Gradient-Descent methods

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

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

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

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

Gradient-Descent methods

- optmize 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
  - Take 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

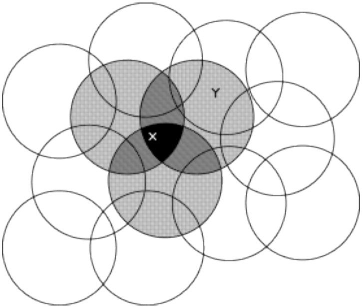
- assume that V_t is a linear function of the parameter vector
- column vector of *features* $\vec{\Phi}_s$ for every state s
- same number of components as $\vec{\theta}_t$

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

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

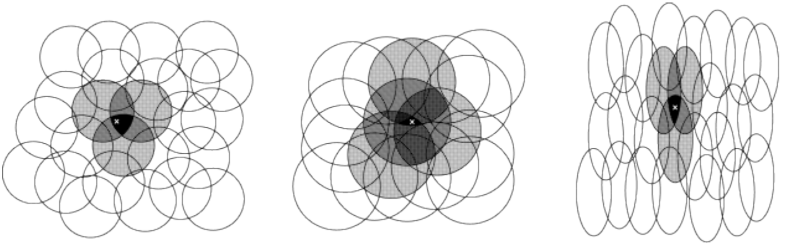
- only one optimum $\vec{\theta}$, any method guaranteed to converge will converge to the (global) optimum

Coarse coding



- example for 2D continuous state-space
- consider circular binary-features: state x inside a circle or not
- *receptive field* of a feature
- *coarse coding*: representing a state with overlapping features

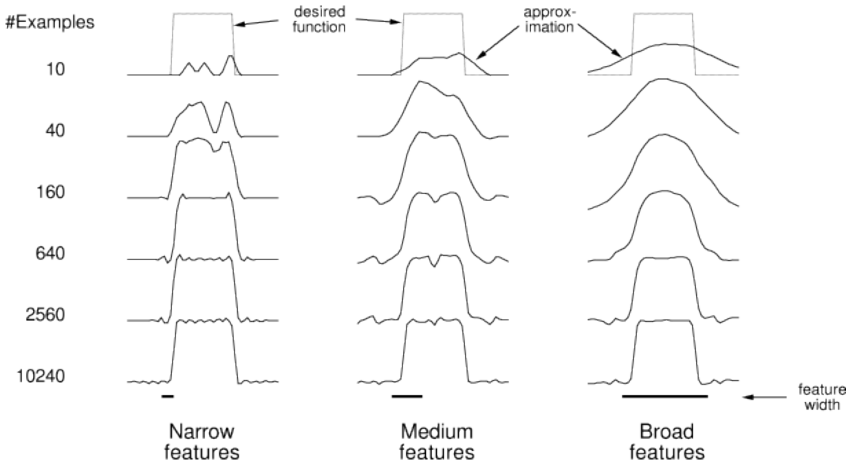
Generalization in linear function approximation



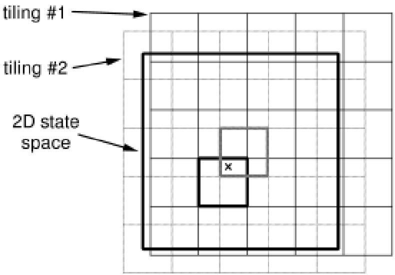
a) Narrow generalization b) Broad generalization c) Asymmetric generalization

- each circle represents one parameter (component of $\vec{\theta}$), which will be updated during learning
- training at a state s will affect all circles (features) that cover the state s
- size (and shape) of the functions determine the detail that can be represented and learned

Effect of feature-width on generalization



Tile Coding

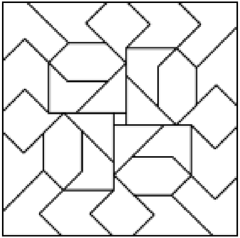


Shape of tiles ⇒ Generalization

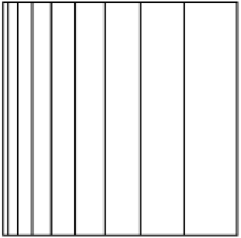
#Tilings ⇒ Resolution of final approximation

- receptive fields of the features selected to cover the input space
- exhaustive partitions of the input space, called a *tiling*
- each *tile* if the receptive field for one binary feature
- examples: a regular grid, overlapping (shifted) grids, etc.

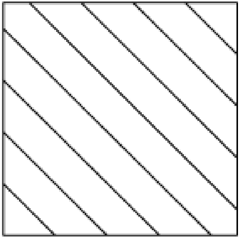
None-uniform grids



a) Irregular



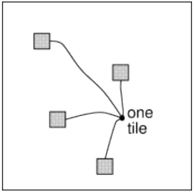
b) Log stripes



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

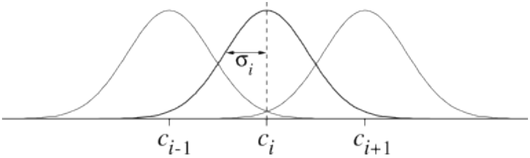


- reduce memory requirements using *hashing*
- only allocate/use memory-cells encountered so far
- represent large (unimportant) parts of the state-space with few large tiles, but add more tiles for the important parts (or dimensions) of the state space

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:

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

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

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 epode):
- $\vec{e} = 0$
- $s, a \leftarrow$ 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 epode):
    $\vec{e} = 0$
    $s, a \leftarrow$ 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: one 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

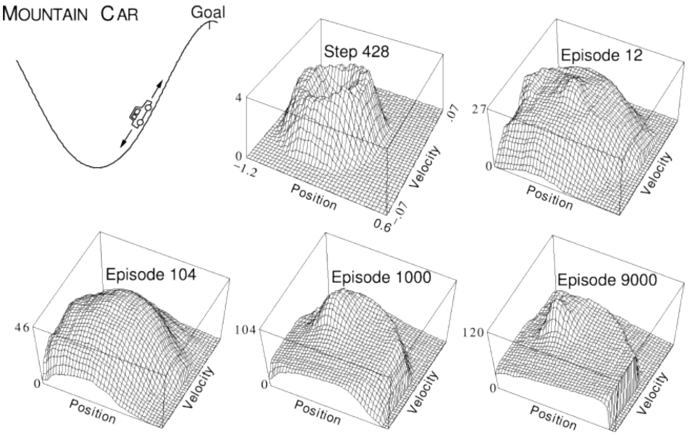
- simplified physics model:

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

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

  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 $-\max_a Q_t(s, a)$

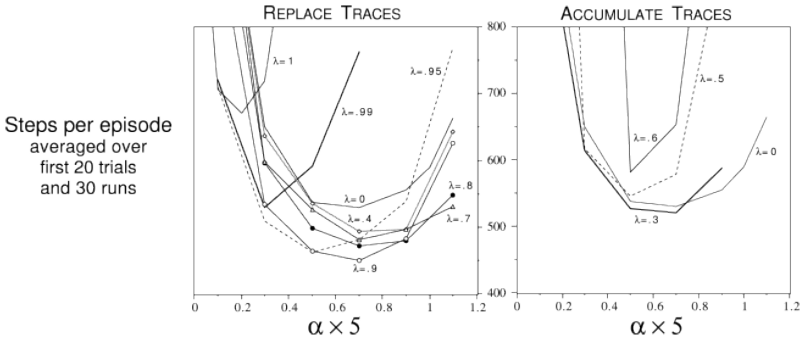


Details: Sutton and Barto, chapter 8.10

Mountain-car: analysis

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

Mountain-car:



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

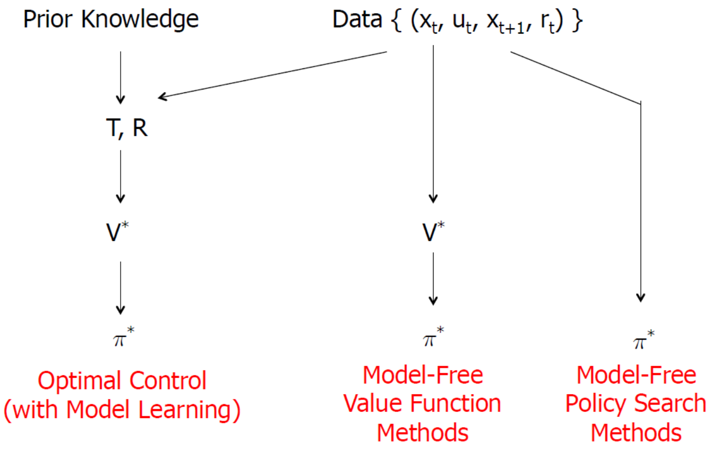
Contents

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
- avoid unsafe states and actions

Learning in Policy Space



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

- slides not ready

Current Research Areas

- *hierarchical reinforcement learning*
- inverse-RL: learning from demonstrations
- learning high-DOF problems (humanoids $\approx$ 70-DOF)
- combining learning and planning

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 Problem: setup

- 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 condition/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 idea

- 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