Reinforcement Learning

Function approximation

Mario Martin

CS-UPC

May 18, 2018

Recap

- Algorithms:
  - MonteCarlo methods for Policy Evaluation and Policy Learning
  - Temporal Differences methods for Policy Evaluation and Policy Learning
  - Extension of TD to n-steps back-ups and TD(λ)
  - Sarsa and Expected Sarsa algorithms
- Differences between on-policy and off-policy learning
- Exploration vs. Exploitation
- Alpha parameter for learning

Index

- Problem
- Incremental algorithms
  - Prediction methods
    - ★ Prediction Objective: Mean Squared Value Error, denoted VE:
    - ★ Easy case: Linear methods and TD
    - ★ Representation: RBFs, tile Coding, Coarse represent
    - ★ Stochastic gradient
    - ★ MC
    - ★ TD
  - Control methods: Sarsa Q-learning
- batch methods
  - Least Squares LSTD (LSPI)
  - Kernel methods
  - NNs and Deep Learning
- Average reward

Goal of this lecture

- Methods we have seen so far work well when we have a *tabular* representation for each state, that is, when we represent value function with a lookup table.
- This is not reasonable on most cases:
  - In Large state spaces: There are too many states and/or actions to store in memory (f.i. Backgammon: 10^{20} states, Go 10^{170} states)
  - and in continuous state spaces (f.i. robotic examples)

Goal of this lecture

- Methods we have seen so far work well when we have a *tabular* representation for each state, that is, when we represent value function with a lookup table.
- This is not reasonable on most cases:
  - In Large state spaces: There are too many states and/or actions to store in memory (f.i. Backgammon: 10^{20} states, Go 10^{170} states)
  - and in continuous state spaces (f.i. robotic examples)
- In addition, we want to generalize from/to similar states to speed up learning. It is too slow to learn the value of each state individually.

Goal of this lecture

- We'll see now methods to learn policies for large state spaces by using *function approximation to estimate value functions*:

$$V_\theta(s) \approx V^\pi(s) \quad (1)$$

$$Q_\theta(s,a) \approx Q^\pi(s,a) \quad (2)$$

- θ is the set of parameters of the function approximation method (with size much lower than $|S|$)
- Function approximation allow to generalize from seen states to unseen states and to save space.
- Now, instead of storing V values, we will update θ parameters using MC or TD learning so they fulfill (1) or (2).

Which Function Approximation?

- There are many function approximators, e.g.
  - Artificial neural network
  - Decision tree
  - Nearest neighbor
  - Fourier/wavelet bases
  - Coarse coding
- In principle, any function approximator can be used. However, the choice may be affected by some properties of RL:
  - Experience is not i.i.d. – Agent's action affect the subsequent data it receives
  - During control, value function V(s) changes with the policy (non-stationary)

Incremental methods

Which Function Approximation?

- Incremental methods allow to directly apply the control methods of MC, Q-learning and Sarsa, that is, back up is done using *"on-line"* sequence of data of the trial reported by the agent following the policy.
- Most popular method in this setting is **gradient descent**, because it adapts to changes in the data (non-stationary condition)

Gradient Descent

- Let $L(\theta)$ be a differentiable function of parameter vector θ, we want to minimize
- Define the gradient of $L(\theta)$ to be:

$$\nabla_\theta L(\theta) = \begin{bmatrix} \frac{\partial L(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial L(\theta)}{\partial \theta_n} \end{bmatrix}$$

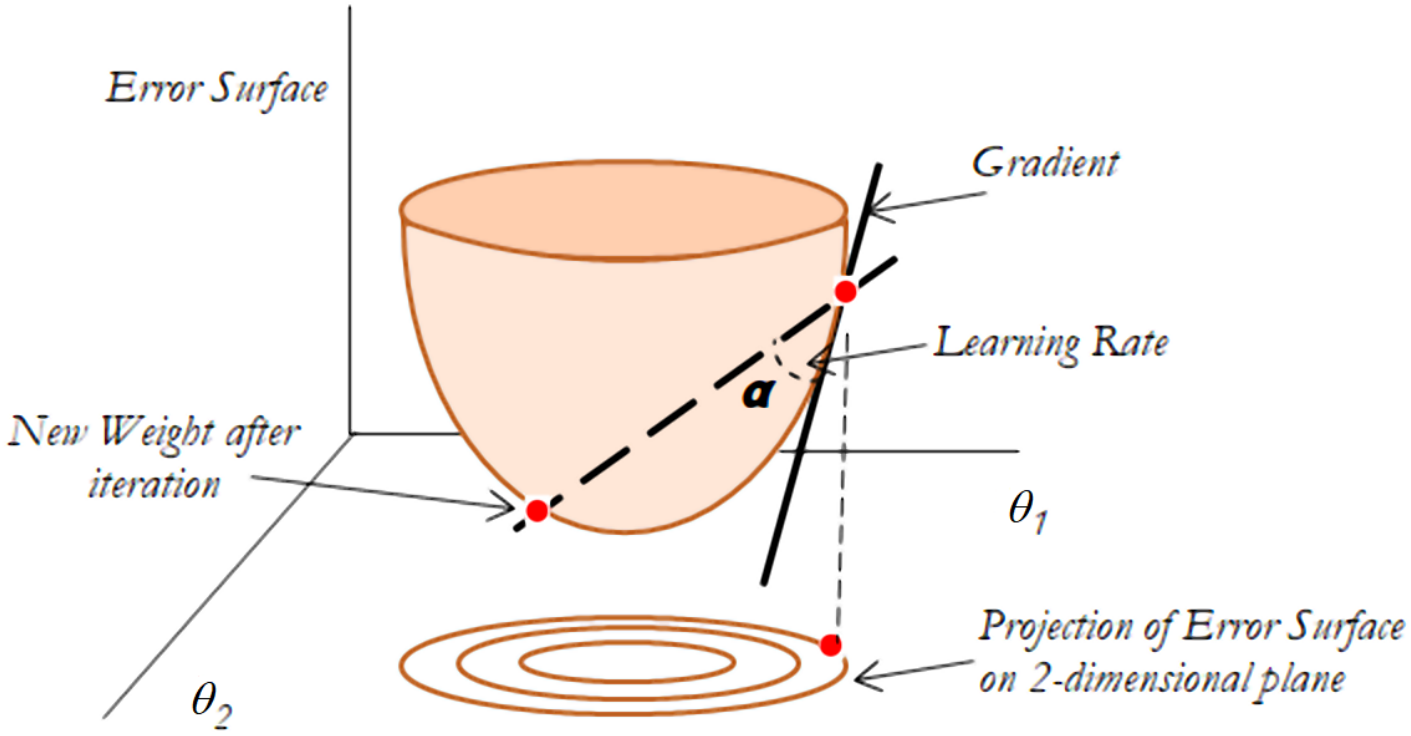
- To find a local minimum of $L(\theta)$, gradient descent method adjust the parameter in the direction of negative gradient:

$$\Delta\theta = -\frac{1}{2}\alpha\nabla_\theta L(\theta)$$

where is a stepsize parameter

Gradient Descent

$$\Delta\theta = -\frac{1}{2}\alpha\nabla_\theta L(\theta)$$



Value Function Approx. by SGD

Minimizing Loss function of the approximation

Goal: Find parameter vector θ minimizing mean-squared error between approximate value function $V_\theta(s)$ and true value function $V^\pi(s)$

$$L(\theta) = \mathbb{E}_\pi \left[(V^\pi(s) - V_\theta(s))^2 \right] = \sum_{s \in \mathcal{S}} \mu^\pi(s) \left[V^\pi(s) - V_\theta(s) \right]^2$$

where $\mu^\pi(s)$ is the time spent in state s while following π

- Gradient descent finds a *local* minimum:

$$\begin{aligned} \Delta\theta &= -\frac{1}{2}\alpha \nabla_\theta L(\theta) \\ &= \mathbb{E}_\pi \left[(V^\pi(s) - V_\theta(s)) \, \nabla_\theta V_\theta(s) \right] \end{aligned}$$

- **Stochastic gradient descent** (SGD) *samples* the gradient

$$\Delta\theta = \alpha (V^\pi(s) - V_\theta(s)) \, \nabla_\theta V_\theta(s)$$

Subsection 1

Linear approximation

Linear representation of the state

- Represent state by a feature vector:

$$\phi(s) = \begin{bmatrix} \phi_1(s) \\ \vdots \\ \phi_n(s) \end{bmatrix}$$

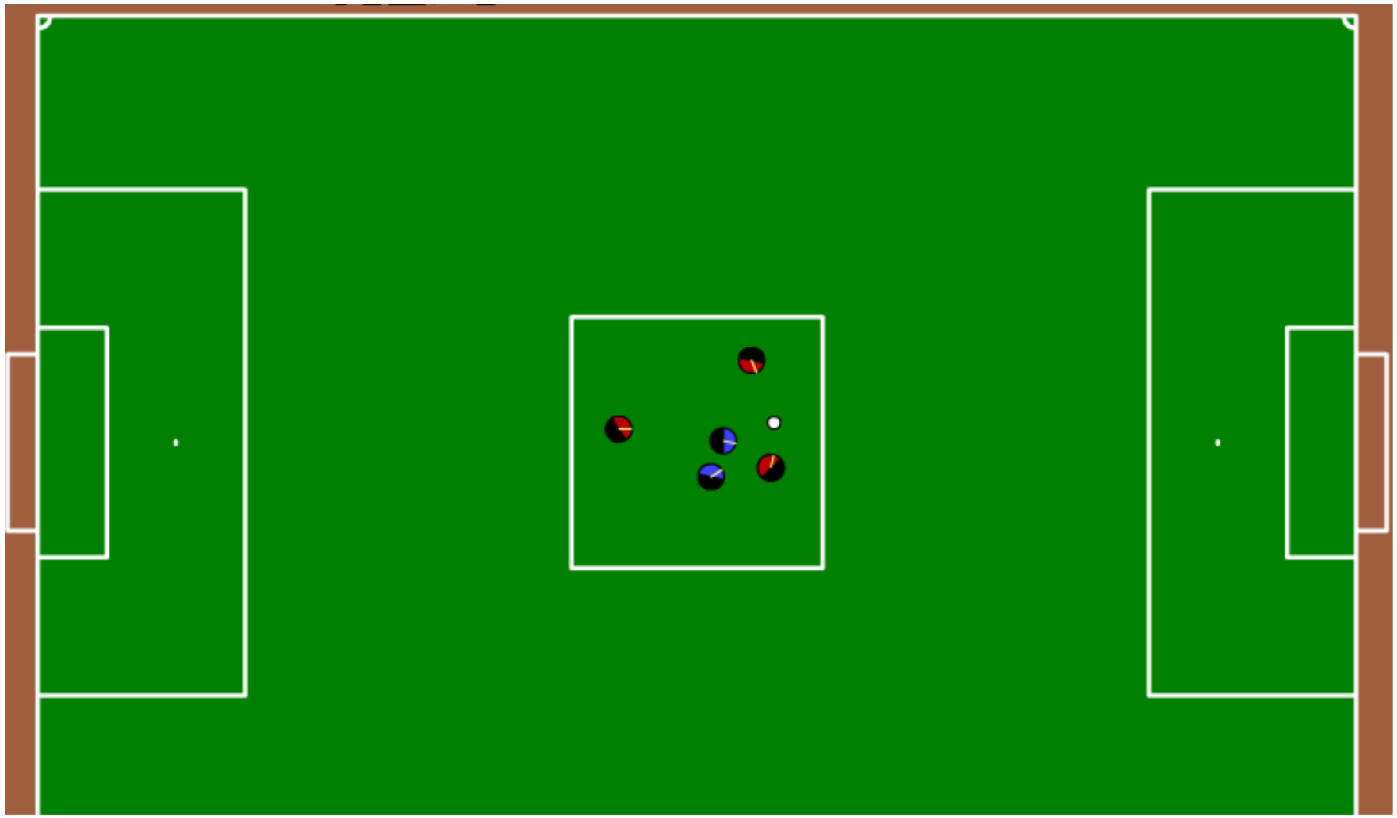
- Represent value function by a linear combination of features:

$$V_\theta(s) = \phi(s)^T \theta = \sum_{j=1}^{n} \phi_j(s)\theta_j$$

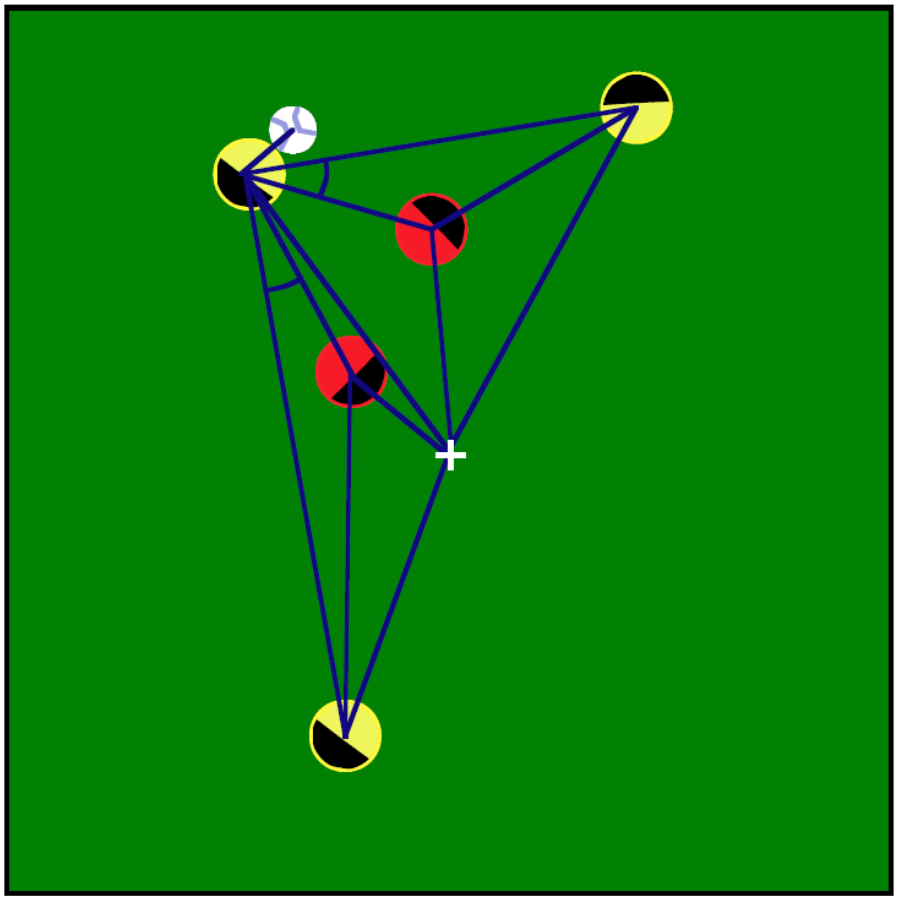
Linear representation of the state

- For example:
  - Distance of robot from landmarks
  - Trends in the stock market
  - Piece and pawn configurations in chess

Example: RoboCup soccer keepaway (Stone, Sutton & Kuhlmann, 2005)



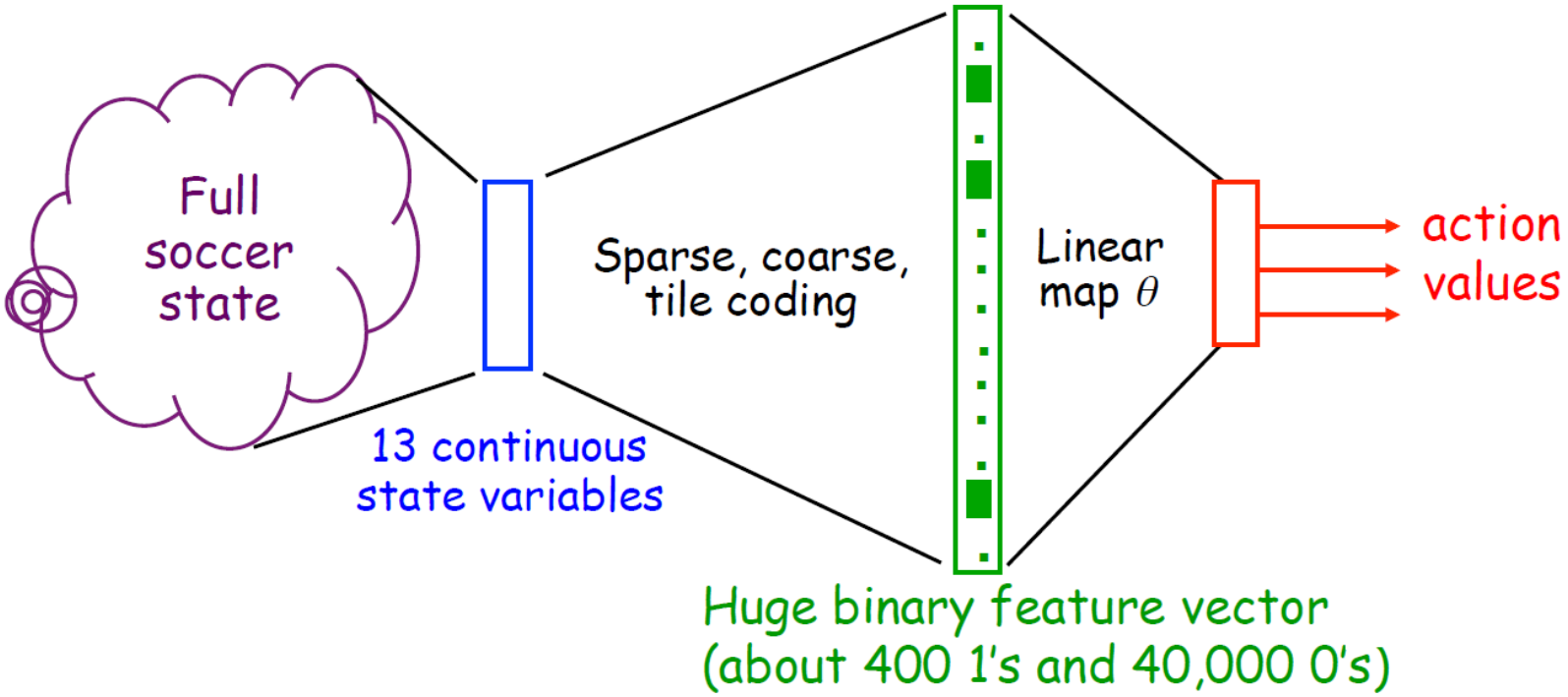
Example: RoboCup soccer keepaway (Stone, Sutton & Kuhlmann, 2005)



State is encoded in 13 continuous variables:

- 11 distances among the players, ball, and the center of the field
- 2 angles to takers along passing lanes

Example: RoboCup soccer keepaway (Stone, Sutton & Kuhlmann, 2005)



Linear representation of the state

- Table lookup is a special case of linear value function approximation. Using table lookup features:

$$\phi^{table}(S) = \begin{bmatrix} 1(S = s_1) \\ \vdots \\ 1(S = s_n) \end{bmatrix}$$

- Parameter vector is exactly value of each individual state

$$V_\theta(S) = \begin{bmatrix} 1(S = s_1) \\ \vdots \\ 1(S = s_n) \end{bmatrix}^T \cdot \begin{bmatrix} \theta_1 \\ \vdots \\ \theta_n \end{bmatrix}$$

Linear representation of the state

- Another obvious way of reducing the number of states is by grouping some of them using a grid.
- Drawback is that all states in the cell are equal and you don't learn "softly" from neighbor cells.
- Better approach is *Coarse Coding*.
- Coarse coding provides large feature vector $\phi(s)$ that "overlap"

Linear representation of the state

- Another obvious way of reducing the number of states is by grouping some of them using a grid.
- Drawback is that all states in the cell are equal and you don't learn "softly" from neighbor cells.
- Better approach is *Coarse Coding*.
- Coarse coding provides large feature vector $\phi(s)$ that "overlap"

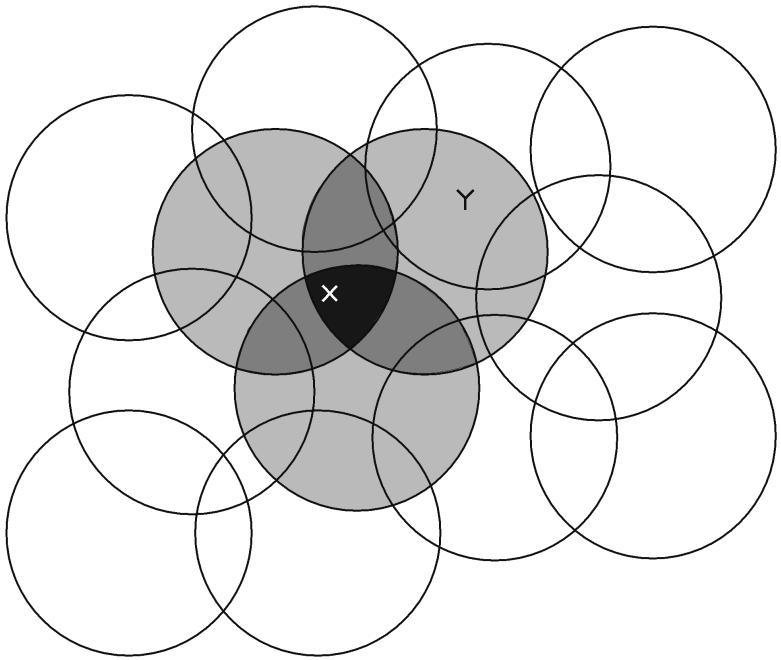
Caution

When using linear FA, we should ask ourselves if V can be approximated by a linear function and what's the error of this approximation.

Usually value functions are smooth (compared with reinforcement function). However, linear FA approximation error could be large, depending on the features selected.

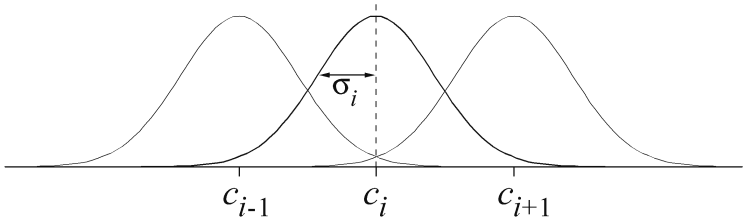
Coarse coding using RBFs

- Each circle is a *Radial Basis Function* (center c and a width σ) that represents a feature.



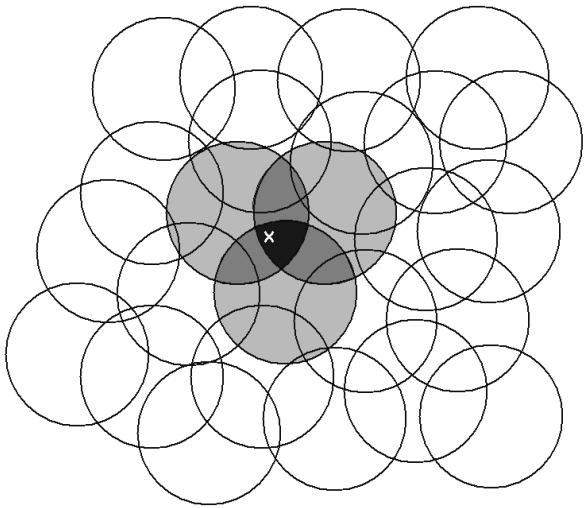
Value for each feature is:

$$\phi_i(s) = e^{-\frac{\|x-c_i\|^2}{(2\sigma^2)}}$$

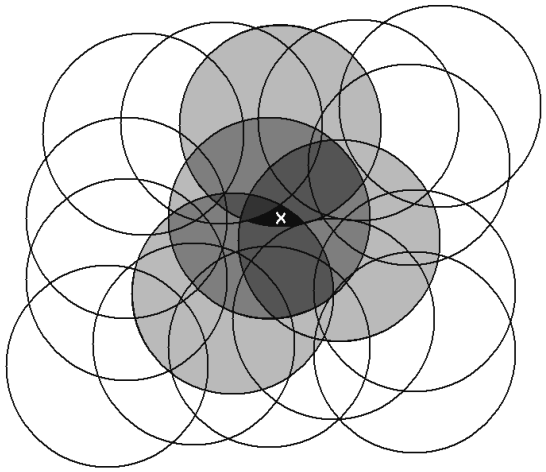


Coarse coding using RBFs

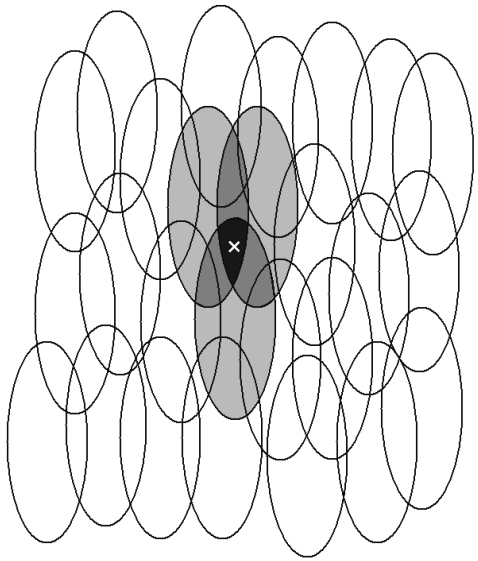
- Several possibilities:
  - Number of centers (RBF's)
  - Width
  - Different width for each variable of the state



a) Narrow generalization



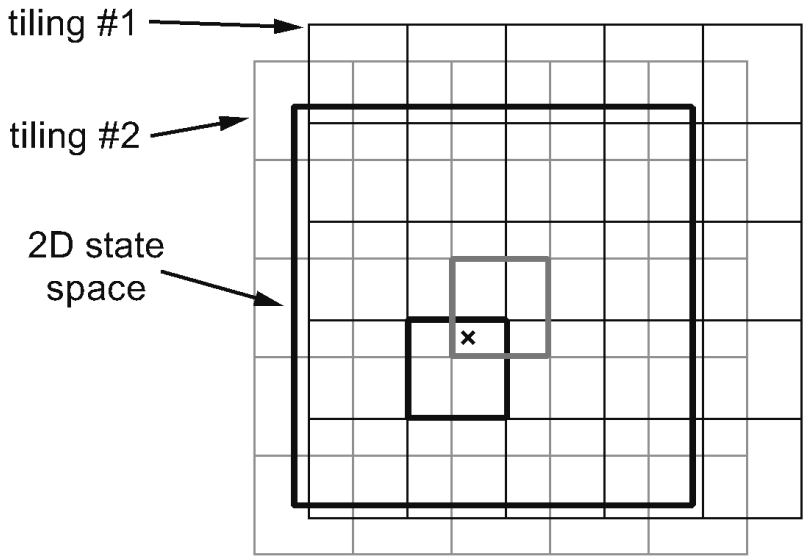
b) Broad generalization



c) Asymmetric generalization

Coarse coding using Tiles

- *RBFs* return a real value for each feature. *Tiles* define a **binary feature** for each tile.
- Binary features means weighted sum easy to compute
- Number of features present at any one time is constant
- Easy to compute indexes of the features present



Shape of tiles $\Rightarrow$ Generalization

#Tilings $\Rightarrow$ Resolution of final approximation

- You can use irregular tilings or superposition of different tilings

Going back to SGD

First nice property of SGD in linear F.A.

In the case of linear function approximation, objective function is *quadratic*:

$$L(\theta) = \mathbb{E}_\pi \left[(V^\pi(s) - \phi(s)^T \theta)^2 \right]$$

so SGD converges to global optimum:

Going back to SGD

Second nice property of SGD in linear F.A.

Gradient vector of value function is vector of feature values:

$$\begin{aligned} \frac{\partial V_\theta(s)}{\partial \theta_i} &= \sum_{j=1}^{n} \phi_j(s)\theta_j \\ &= \phi_i(s) \end{aligned}$$

So, update rule is particularly simple:

$$\Delta\theta_i = \alpha(V^\pi(s) - V_\theta(s))\ \phi_i(s)$$

Subsection 2

Prediction algorithms for linear case

Prediciton algorithms for linear case

- Have assumed true value function $V^{\pi}(s)$ is given in a supervised learning way
- But in RL there is only rewards of trial, not examples
- Solution: substitute $V^{\pi}(s)$ by a target that is an estimation of it. In practice,
  - For MC, the target is the return R_t
  - For TD(0), the target is the Bellman equation

MC prediction algorithm for the linear case

- The long-term-return of a trial R_t is an unbiased, noisy sample of true value $V^\pi(s)$.
- Using R_t as a target, we have linear Monte–Carlo policy evaluation

$$\begin{aligned} \Delta\theta &= \alpha(R_t - V_\theta(s))\nabla_\theta V_\theta(s) \\ &= \alpha(R_t - V_\theta(s))\phi(s) \end{aligned}$$

- Monte–Carlo evaluation converges to *optimum* (θ with global minimum error)
- Moreover, MC, even when using non–linear value function approximation converges, but in this case to a *local optimum*

MC prediction algorithm for the linear case

Monte Carlo policy evaluation

Given π, the policy to be evaluated, initialize parameters θ as appropriate (e.g., $\theta = 0$)

repeat

  Generate trial using π

  for each s_t in trial **do**

    $R_t \leftarrow$ return following the first occurrence of s_t

    $\theta \leftarrow \theta + \alpha(R_t - V_\theta(s_t))\phi(s_t)$ // Notice θ and ϕ are vectors

  end for

until true

TD prediction algorithm for the linear case

Changes applying TD to the linear case:

1. Function approximation is now for the Q value function:

$$Q^{\pi}(s,a) \approx Q_{\theta}(s,a) = \phi(s,a)^{T}\theta = \sum_{j=1}^{n} \phi_j(s,a)\theta_j$$

2. Loss function is also now for Q value function:

$$L(\theta) = \mathbb{E}_{\pi}\left[(Q^{\pi}(s,a) - \phi(s,a)^{T}\theta)^2 \right]$$

TD prediction algorithm for the linear case

- In TD(0) we use Q of *next state* to estimate Q on the *current state* using Bellman equations. So, in general,

$$\begin{aligned}\Delta\theta_i =& \alpha(\quad Q^\pi(s,a) \quad - Q_\theta(s,a))\nabla_\theta Q_\theta(s,a) \\ =& \alpha(r + \gamma Q_\theta(s', \pi(s')) - Q_\theta(s,a))\nabla_\theta Q_\theta(s,a)\end{aligned}$$

- And, in particular, for the linear case:

$$\begin{aligned}\frac{\partial Q_\theta(s,a)}{\partial \theta_i} &= \frac{\partial\left(\sum_{j=1}^{n} \phi_j(s,a)\theta_j\right)}{\partial \theta_i} \\ &= \phi_i(s,a)\end{aligned}$$

- and so,

$$\Delta\theta_i = \alpha(r + \gamma Q_\theta(s', \pi(s')) - Q_\theta(s,a))\ \phi_i(s,a)$$

TD prediction algorithm for the linear case

Caution!

- No same guarantees that MC had when *bootstrapping* estimate of $Q(S_t, a)$ is used as the target
- Notice that TD targets are not independent of parameters. In TD(0):

$$r + \gamma \mathbf{Q}_\theta(s', \pi(s')$$

  depends of θ
- Bootstrapping methods are not true gradient descent: they take into account the effect of changing θ on the estimate, **but ignore its effect on the target**. They include only a part of the gradient and, accordingly, we call them *semi-gradient methods.*

- However, it can be proved that **linear** TD(0) policy evaluation converges (close) to global optimum.

TD(0) prediction algorithm for the linear case

TD(0) policy evaluation

Given π, initialize initialize parameters θ arbitrarily (e.g., $\theta = 0$)

repeat

  $s \leftarrow$ initial state of episode

  repeat

    $a \leftarrow \pi(s)$

    Take action a and observe s' and r

    $\theta \leftarrow \theta + \alpha \left(r + \gamma Q_\theta(s', \pi(s')) - Q_\theta(s, a) \right) \phi(s_t)$

    $s \leftarrow s'$

  until s is terminal

until convergence

Subsection 3

Control algorithms for the linear case

TD(0) prediction algorithm for the linear case

- Like the Control methods we used in tabular learning algorithms, we will build algorithms that iterate the two following steps:
  1. Policy evaluation - Follow a method for approximate policy evaluation $Q_\theta \approx Q^\pi$
  2. Policy improvement - do policy improvement of the policy
- Depending on the Policy evaluation procedure used (MC, TD, etc.), we have a different method

Examples of Control using PI

Linear FA Monte Carlo

Initialize parameters θ as appropriate (e.g., $\theta = 0$)
repeat
  Generate trial using ϵ-greedy policy derived from Q_θ
  for each s_t in trial **do**
    $R_t \leftarrow$ return following the first occurrence of s_t
    $\theta \leftarrow \theta + \alpha(R_t - Q_\theta(s_t, a_t))\phi(s_t, a_t)$
  end for
until true

Examples of Control using PI

Linear FA Monte Carlo

Initialize parameters θ as appropriate (e.g., $\theta = 0$)
repeat
  Generate trial using ϵ-greedy policy derived from Q_θ
  for each s_t in trial **do**
    $R_t \leftarrow$ return following the first occurrence of s_t
    $\theta \leftarrow \theta + \alpha(R_t - Q_\theta(s_t, a_t))\phi(s_t, a_t)$
  end for
until true

Function $Q_\theta(s, a)$

Given θ, state s and action a
  return $\theta^T \phi(s, a)$

Select action following policy

Function $\pi_\theta(s)$

Given θ and state s

return $\arg\max_a \ \theta^T \phi(s,a)$

Function implementing ϵ-greedy

Given θ, $\epsilon \leq 1$ and state s

Select p number from uniform distribution in range $[0,1]$
if $p \leq \epsilon$ **then**
$a \leftarrow$ Random action from $\mathcal{A}$
else
$a \leftarrow \pi_\theta(s)$
end if
return a

Examples of Control using PI

Linear FA Q-learning

initialize parameters θ arbitrarily (e.g. $\theta = 0$)
for each episode **do**
  Choose initial state s
  repeat
    Choose a from s using policy π_θ derived from Q_θ (e.g., ϵ-greedy)
    Execute action a, observe r, s'
    $\theta \leftarrow \theta + \alpha \left(r + \gamma Q_\theta(s', \pi_\theta(s')) - Q_\theta(s, a) \right) \phi(s, a)$
    $s \leftarrow s'$
  until s is terminal
end for

Example of Control using PI

Linear FA Sarsa: on-line learning

initialize parameters θ arbitrarily (e.g. $\theta = 0$)
for each episode **do**
  Choose initial state s
  Choose a from s using policy derived from Q_θ (e.g., ϵ-greedy)
  repeat
    Execute action a, observe r, s'
    Choose a' from s' using policy derived from Q_θ (e.g., ϵ-greedy)
    $\theta \leftarrow \theta + \alpha \left(r + \gamma Q_\theta(s', a') - Q_\theta(s, a)\right) \phi(s)$
    $s \leftarrow s'; a \leftarrow a'$
  until s is terminal
end for

Convergence of methods

- Experiments show it is desirable to bootstrap (TD in practice better than MC)
- But now we should consider convergence issues. When do incremental prediction algorithms converge?
  - When using bootstrapping (i.e., TD with < 1)?
  - When using linear function approximation?
  - When using off–policy learning?
  - When using non-linear approximation?
- Ideally, we would like algorithms that converge in all cases

Convergence of Gradient methods

- We have examples of TD divergence even when exact solution is representable with linear function
- Fortunately, in practice, TD(0) works well... but we don't have guarantees
- Problem can be solved if we update parameters following an on-policy distribution (we have a proof of that). Good for Sarsa.
- Unfortunately convergence guarantees on TD incremental methods only work for linear approximation
- Main cause is that TD does not follow true gradient.

Deadly triad

- The risk of divergence arises whenever we combine three things:

  Function approximation: Significantly generalizing from large numbers of examples.

  Bootstrapping: Learning value estimates from other value estimates, as in dynamic programming and temporal-difference learning.

  Off-policy learning: Learning about a policy from data not due to that policy, as in Q-learning, where we learn about the greedy policy from data with a necessarily more exploratory policy.

- Any two without the third is ok.

Convergence of incremental Gradient methods for prediction

On/Off–Policy	Algorithm	Table Lookup	Linear	Non–Linear
On–Policy	MC	OK	OK	OK
	TD(0)	OK	OK	KO
	TD(λ)	OK	OK	KO
Off–Policy	MC	OK	OK	OK
	TD(0)	OK	KO	KO
	TD(λ)	OK	KO	KO

Convergence of incremental Gradient methods for control

Algorithm	Table Lookup	Linear
Monte–Carlo Control	OK	OK
SARSA	OK	(OK)
Q–learning	OK	KO

(OK) = **chatters** around near–optimal value function

Conclusions and final notes about convergence

- Value-function approximation by stochastic gradient descent enables RL to be applied to arbitrarily large state spaces
- Most algorithms just carry over the Targets from the tabular case
- With bootstrapping (TD), we don't get true gradient descent methods
  - this complicates the analysis
  - but the linear, on-policy case is still guaranteed convergent
  - and learning is still much faster
- For continuous state spaces, coarse/tile coding is a good strategy

Conclusions and final notes about convergence

- Value-function approximation by stochastic gradient descent enables RL to be applied to arbitrarily large state spaces
- Most algorithms just carry over the Targets from the tabular case
- With bootstrapping (TD), we don't get true gradient descent methods
  - this complicates the analysis
  - but the linear, on-policy case is still guaranteed convergent
  - and learning is still much faster
- For continuous state spaces, coarse/tile coding is a good strategy
- Still some possible approaches: Gradient-TD (convergence in off-line linear FA) and Batch methods

Mountain Car demonstration

- Q-learning with linear FA and Semi Gradient Methods.
  - Aggregating states
  - Tiling

Batch methods

Batch Reinforcement Learning

- Gradient descent is simple and appealing
  - It is computationally efficient (one update per sample)
  - ... But it is not sample efficient (does not take all profit from samples)
- We can do better at the cost of more computational time

Batch Reinforcement Learning

- Gradient descent is simple and appealing
  - It is computationally efficient (one update per sample)
  - ... But it is not sample efficient (does not take all profit from samples)
- We can do better at the cost of more computational time

- **Batch methods** seek to find the *best fitting value function* of given agent's experience ("training data") in a supervised way.

Subsection 1

Least Square (LS) Prediction and Least Square Policy Iteration (LSPI)

Least Squares Prediction

- Given value function approximation method with parameters θ
- And experience $\mathcal{D}$ consisting of (state,value) pairs:

$$\mathcal{D} = \{\langle s_1, V_1^\pi \rangle, \langle s_2, V_2^\pi \rangle, \ldots \langle s_T, V_T^\pi \rangle\}$$

- Find parameters θ that give the best fitting of value function $V_\theta(s)$
- *Least squares* algorithm find parameter vector θ minimizing sum-squared error between V_θ and target values V^π:

$$\begin{aligned} LS(\theta) &= \sum_{t=1}^{T} (V_t^\pi - V_\theta(s_t))^2 \\ &= \mathbb{E}_{\mathcal{D}}[(V_t^\pi - V_\theta(s_t))^2] \end{aligned}$$

SGD with Experience Replay

- Given experience consisting of (state, value) pairs,

$$\mathcal{D} = \{\langle s_1, V_1^\pi \rangle, \langle s_2, V_2^\pi \rangle, \ldots \langle s_T, V_T^\pi \rangle\}$$

- Repeat
  1. Sample state, value from experience

$$\langle s_i, V_i^\pi \rangle$$

  2. Apply stochastic gradient descent update

$$\Delta\theta = \alpha(V^\pi(s_i) - V_\theta(s_i))\, \nabla_\theta V_\theta(s)$$

- Converges to least squares solution:

$$\theta^\pi = \arg\min_\theta LS(\theta)$$

Linear Least Squares Prediction

- Experience replay finds least squares solution
- But it may take many iterations
- Using linear value function approximation $V_\theta(s) = \phi(s)^T\theta$
- We can solve the least squares solution directly

Linear Least Squares Prediction

- At minimum of LS(w), the expected update must be zero

$$
\begin{aligned}
\mathbb{E}_{\mathcal{D}}[\Delta\theta] &= 0 \\
\alpha \sum_{t=1}^{T} \phi(s_t)(V_t^{\pi} - \phi(s_t)^T\theta) &= 0 \\
\sum_{t=1}^{T} \phi(s_t) V_t^{\pi} &= \sum_{t=1}^{T} \phi(s_t)\phi(s_t)^T\theta \\
\left(\sum_{t=1}^{T} \phi(s_t)\phi(s_t)^T\right)^{-1} \sum_{t=1}^{T} \phi(s_t) V_t^{\pi} &= \theta
\end{aligned}
$$

- For N features, direct solution time is $O(N^3)$
- Extensible to Q value function and pairs (s, a)

Linear Least Squares Prediction

- We do not know true values V_t^π
- As always, substitute in equation V_t^π for estimation from samples:

  LSMC Least Squares Monte-Carlo uses return:
  $V_t^\pi \approx R_t$

  LSTD Least Squares Temporal-Difference uses TD target
  $V_t^\pi \approx r_t + \gamma V_\theta(s_{t+1})$
- In each case solve directly for fixed point of MC / TD

Convergence of LS for prediction

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✓	✗
	LSTD	✓	✓	-
Off-Policy	MC	✓	✓	✓
	LSMC	✓	✓	-
	TD	✓	✗	✗
	LSTD	✓	✓	-

Linear Least Squares Policy Iteration (LSPI)

- How to turn the Least Square prediction algorithm into a Control algorithm?
- LSPI Algorithm - two iterated steps

  Policy evaluation Policy evaluation by least squares Q-learning
  Policy improvement Greedy policy improvement
- As in Q-learning, we now use Q value function to get rid of transition probabilities

Linear Least Squares Policy Iteration (LSPI)

- For policy evaluation, we want to efficiently use all experience
- For control, we also want to improve the policy
- This experience is generated from many policies
- So to evaluate Q^π we must learn **off-policy**
- We use the same idea as Q-learning:
  - Use experience generated by old policy $S_t, A_t, R_{t+1}, S_{t+1} \sim \pi_{old}$
  - Consider alternative successor action $A' = \pi_{new}(S_{t+1})$
  - Update $Q_\theta(S_t, A_t)$ towards value of alternative action $R_{t+1} + \gamma Q_\theta(S_{t+1}, A')$

Linear Least Squares Policy Iteration (LSPI)

- LSTDQ algorithm: solve for total update = zero

$$\alpha \sum_{t=1}^{T} \phi(s_t, a_t)(V_t^{\pi} - \phi(s_t, a_t)^T \theta) = 0$$

$$\alpha \sum_{t=1}^{T} \phi(s_t, a_t)(r_{t+1} + \gamma \phi(s_{t+1}, \pi(s_{t+1}))^T \theta - \phi(s_t, a_t)^T \theta) = 0$$

$$\left(\sum_{t=1}^{T} \phi(s_t, a_t)(\phi(s_t, a_t) - \gamma \phi(s_{t+1}, \pi(s_{t+1}))^T \right)^{-1} \sum_{t=1}^{T} \phi(s_t, a_t) r_{t+1} = \theta$$

Linear Least Squares Policy Iteration (LSPI)

- LSPI-TD uses LSTDQ for policy evaluation
- It *repeatedly re-evaluates experience* $\mathcal{D}$ *with different policies*
- Obtain $\mathcal{D}$ with any probabilistic policy (e.g. random policy)

LSPI-TD algorithm

Given $\mathcal{D}$, initialize π'
repeat
  $\pi \leftarrow \pi'$
  $\theta \leftarrow LSTDQ(\mathcal{D}, \pi)$
  for each $s \in S$ **do**
    $\pi'(s) = \arg\max_{a \in \mathcal{A}} \phi(s,a)^T \theta$ // i.e. $\arg\max_{a \in \mathcal{A}} Q_\theta(s,a)$
  end for
until $\pi \approx \pi'$
return π

Convergence of LSPI

Algorithm	Table Lookup	Linear	Non-Linear
Monte-Carlo Control	✓	(✓)	✗
Sarsa	✓	(✓)	✗
Q-learning	✓	✗	✗
LSPI	✓	(✓)	-

Fitted Q-learning: Non-linear approximation

Incremental Q-learning with FA

Q-learning with FA

initialize parameters θ arbitrarily (e.g. $\theta = 0$)
for each episode **do**
  Choose initial state s
  repeat
    Choose a from s using policy π_θ derived from Q_θ (e.g., ϵ-greedy)
    Execute action a, observe r, s'
    $Q_\theta(s,a) \leftarrow Q_\theta(s,a) + \alpha\,(r + \gamma Q_\theta(s', \pi_\theta(s')) - Q_\theta(s,a))\nabla_\theta Q_\theta(s,a))$
    $s \leftarrow s'$
  until s is terminal
end for

Problems with incremental Q-learning with FA

Essence of off-policy learning.

repeat

Choose a, execute it and observe r and s' (s, a, r, s') using any probabilistic policy

$Q_\theta(s,a) \leftarrow Q_\theta(s,a) + \alpha\left(r + \gamma Q_\theta(s', \pi_\theta(s')) - Q_\theta(s,a)\right)\nabla_\theta Q_\theta(s,a))$

$s \leftarrow s'$

until s is terminal

- Several problems with incremental off-policy TD learning
  - SGD does converge because gradient does not follow true gradient
  - Target value is always changing and SGD does not converge
  - Data is not even close to iid (it is strongly correlated) so another problem for SGD convergence
- How to solve all these problems?

Generalizarion of off-policy learning

Let's generalize the method:

Generalizarion of off-policy learning.

repeat

Choose from $\mathcal{D} = \{\langle s, a, r, s' \rangle\}$ N samples randomly. $\mathcal{D}$ has been obtained using any probabilistic policy

for K times **do**

for each sample i in $\mathcal{D}$ **do**

$y_i \leftarrow r + \gamma Q_\theta(s'_i, \max_a Q_\theta(s'_i a))$

end for

$\theta \leftarrow \arg\min_\theta \sum (Q_\theta(s_i, a_i) - y_i)^2$ *// Any ML regression method*

end for

until convergence

Generalizarion of off-policy learning

- Notice several differences:
  1. Sample a set of N examples instead of only 1
  2. Don't use 1-step of gradient descent but compute exact solution (regression problem)
  3. Repeat K times the Policy iteration method with the selected examples

Generalizarion of off-policy learning

- Notice several differences:
  1. Sample a set of N examples instead of only 1
  2. Don't use 1-step of gradient descent but compute exact solution (regression problem)
  3. Repeat K times the Policy iteration method with the selected examples
- Each difference improves convergence
  1. Samples obtained randomly reduce correlation between them and stabilize Q value function for the regressor learner
  2. Computation of exact solution avoid the true gradient problem
  3. Repeat K times the Policy iteration method with the selected examples

Fitted Q-learning

- Implements fitted value iteration
- Given a dataset of experience tuple D, solve a sequence of **regression problems**
  - At iteration i, build an approximation Q_i over a dataset obtained by (TQ_{i-1})
- Allows to use a large class of regression methods, e.g.
  - Kernel averaging
  - Regression trees
  - Fuzzy regression
- With other regression methods it may diverge
- In practice, good results also with **neural networks**

Fitted Q-learning

Fitted Q-learning

Given $\mathcal{D}$ of size T with examples $(s_t, a_t, r_{t+1}, s_{t+1})$, and regression algorithm, set N to zero and $Q_N(s, a) = 0$ for all a and s

repeat

$N \leftarrow N + 1$

Build training set $TS = \{\langle (s_t, a_t), r_{t+1} + \gamma \max_a Q_N(s_{t+1}, a)\rangle\}_{t=1}^{T}$

$Q_{N+1} \leftarrow$ regression algorithm on TS

until $Q_N \approx Q_{N+1}$ or $N >$ limit

return π based on greedy evaluation of Q_N

- Works specially well for forward Neural Networks as regressors (Neural Fitted Q-learning)