Prediction with approximation

Herke van Hoof

Recap value approximation

With large or continuous state space, impractical to represent value for each state separately

Instead, use parametrised value function approximator $\hat{v}(s, \mathbf{w})$

Possible objective: value error

$$\overline{\mathrm{VE}}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2$$

Minimized by gradient MC algorithm

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[G_t - \hat{v}\left(S_t, \mathbf{w}_t\right) \right] \nabla \hat{v}\left(S_t, \mathbf{w}_t\right)$$

Types of function approximator

Types of function approximator

Linear function approximation

- Specify some transformation $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \ldots, x_d(s))^\top$
- Now define

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s)$$

- Note: value is linear in **w**, but not generally in *s*
- We will see some useful transformation in a minute

Types of function approximator

Linear function approximation

- Specify some transformation $\mathbf{x}(s) \doteq (x_1(s), x_2(s), \ldots, x_d(s))^\top$
- Now define

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s)$$

- Note: value is linear in **w**, but not generally in *s*
- We will see some useful transformation in a minute

Non-linear function approximation

- Any function approximation that **cannot** be brought in above form
- E.g.: (deep) neural networks
- Will be discussed later in today's lecture

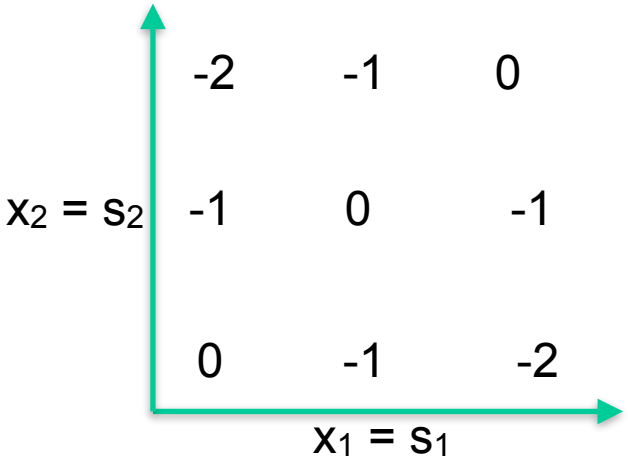
Linear function approximation

Think of **x** as specifying important properties of the state

What is important depends on the task!

Each element of **x** influences approximate value *linearly,* thus, no interaction between dimensions of **x**

True value, unity transformation



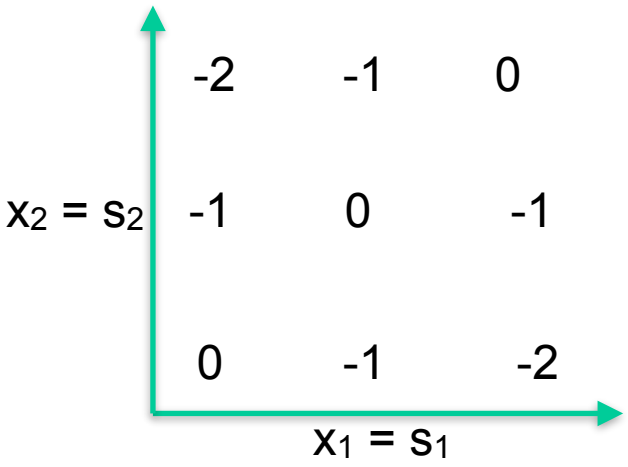
Linear function approximation

Think of **x** as specifying important properties of the state

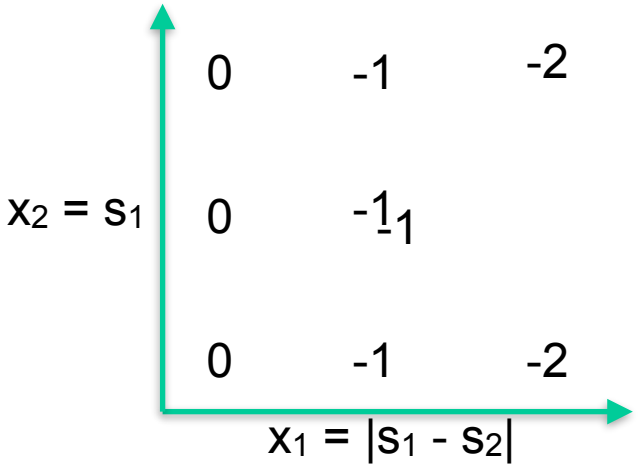
What is important depends on the task!

Each element of **x** influences approximate value *linearly,* thus, no interaction between dimensions of **x**

True value, unity transformation



True value, non-linear transformation



Polynomials

If we don't have task-specific transformation (*features*), use generic features that can approximate any function

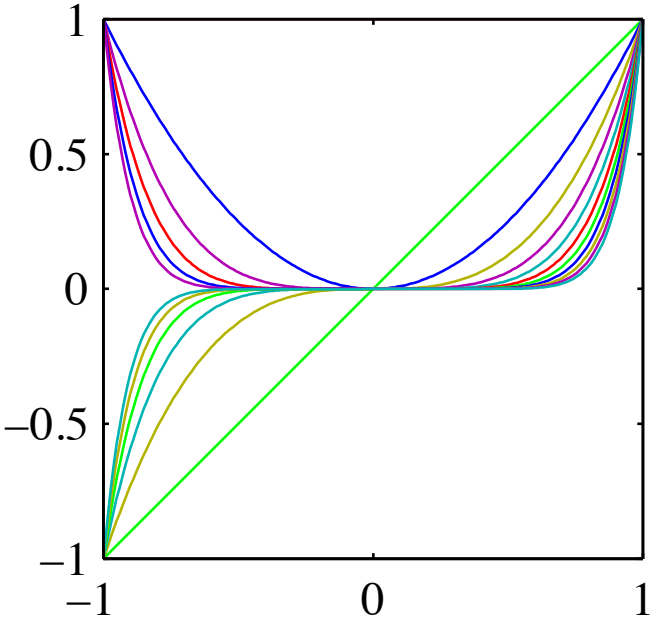
Example: Polynomials

If we take "enough" polynomials, can approximate any function

With many features, less generalisation

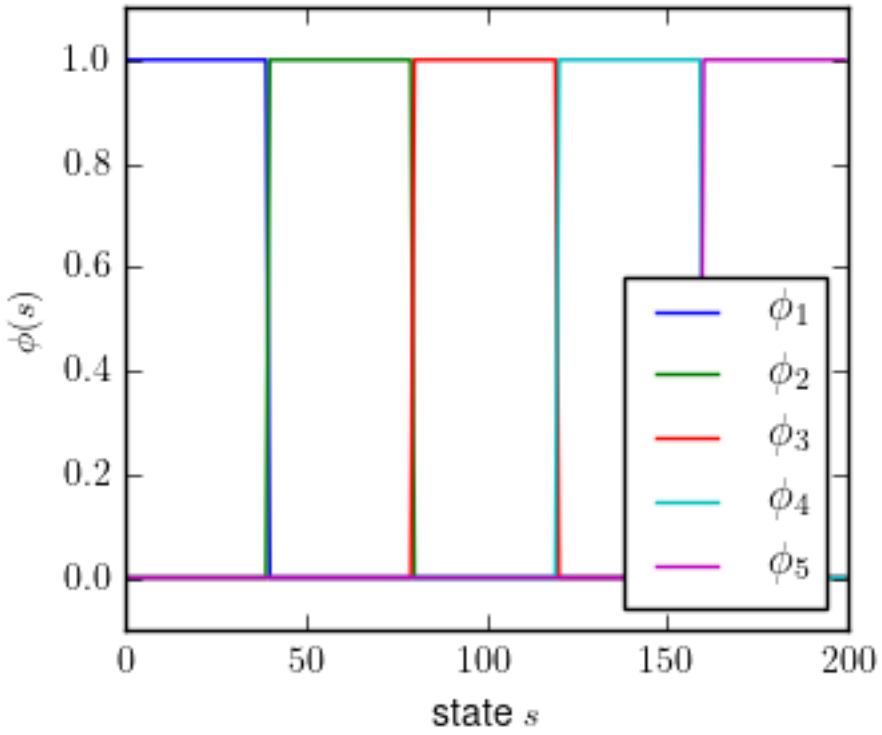
e.g. in two-d (including constant feature):

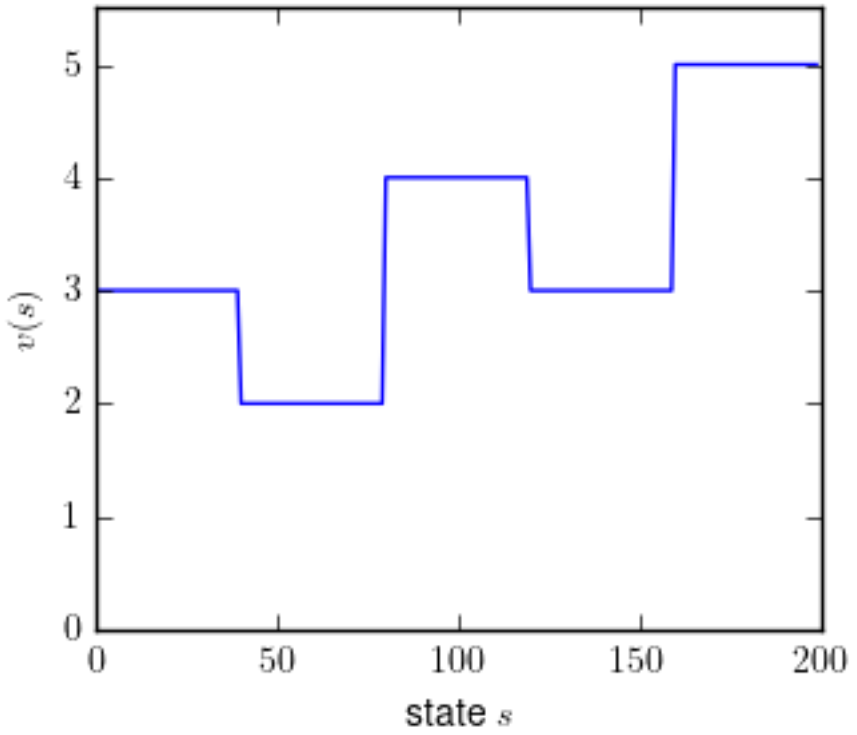
$$\mathbf{x} = [1, s_1, s_2, s_1^2, s_2^2, s_1 s_2, s_1^2 s_2, s_2^2 s_1, s_1^2 s_2^2]^T$$



C.M. Bishop: PRML

Aggregation and tiling



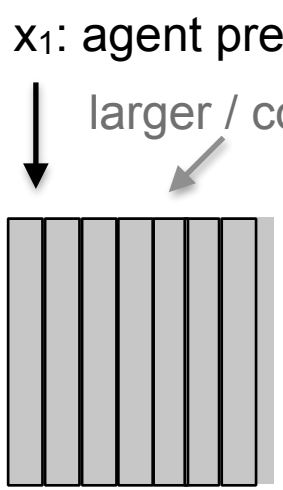


Aggregation and tiling

Different types of aggregation

x_1: agent present in box 1 (1 or 0)

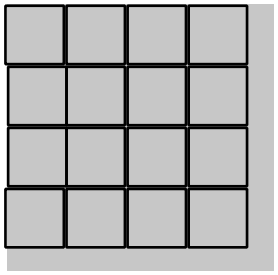
larger / continuous 2-d state-space



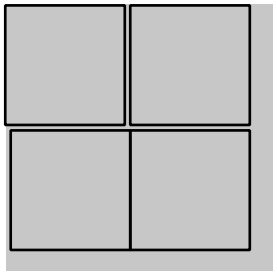
Vertical generalization: suitable if y-coordinate does not matter

Aggregation and tiling

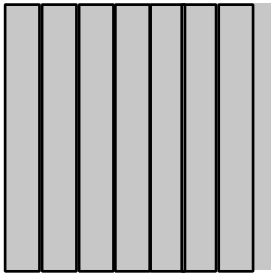
Different types of aggregation



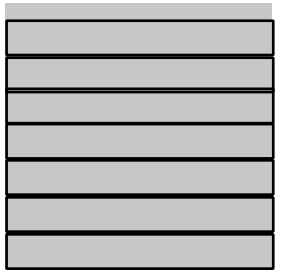
Both coordinates matter, relatively fine discretisation (little generalization)



Both coordinates matter, relatively coarse discretisation (high error possible)



Vertical generalization: suitable if y-coordinate does not matter

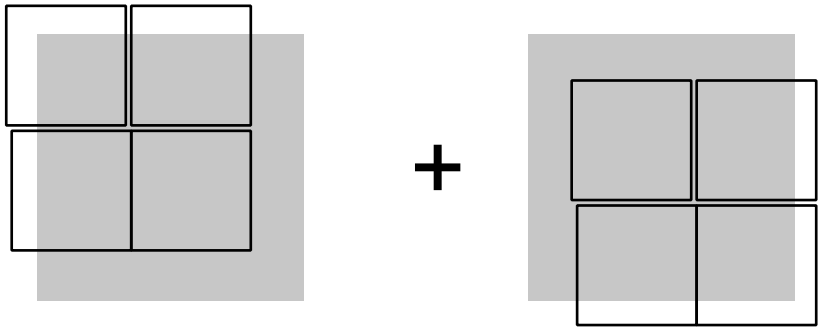


Horizontal generalization: suitable if x-coordinate does not matter

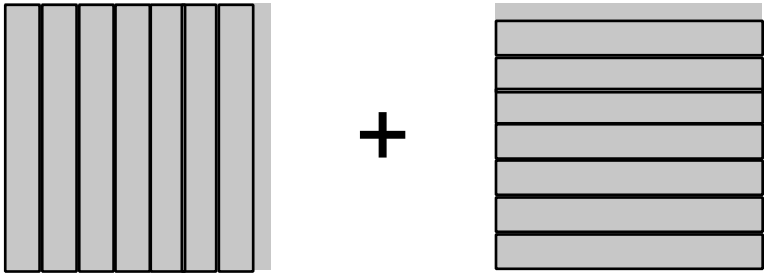
Aggregation and tiling

Pure aggregation: exactly one feature is one, all others zero

Instead: can combine multiple tiling. Exactly n features =1



Relatively much generalization, but still able to learn functions with more detail



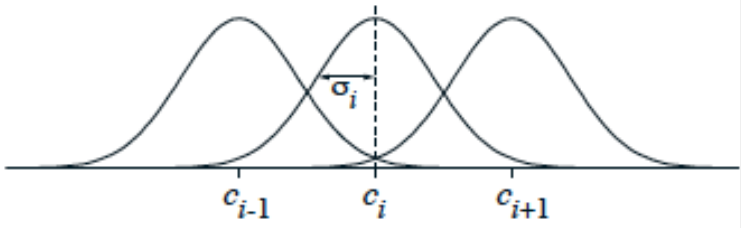
Generalize in x and y directions, but avoid modelling interaction terms

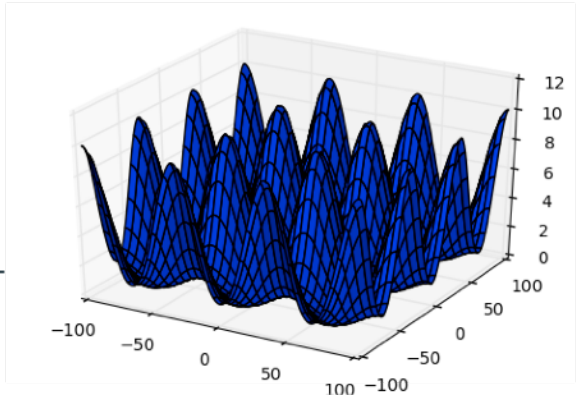
Radial basis functions

With aggregation or tiling: jump in features and value function when border in state space is crossed. Not smooth.

But: we can have features where we go from 0 to 1 smoothly

Radial basis features:





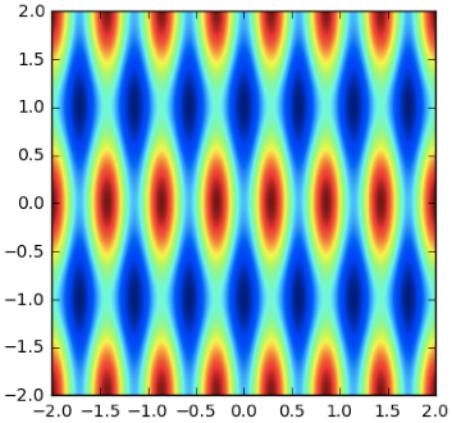


Figure: Sutton & Barto. RL:AI

Linear function approximation continued

Any of these choices leads to linear approximations to the value function, i.e.:

$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^\top \mathbf{x}(s) \doteq \sum_{i=1}^{d} w_i x_i(s)$$

for different choices of the *features* **x**

Convince yourself:

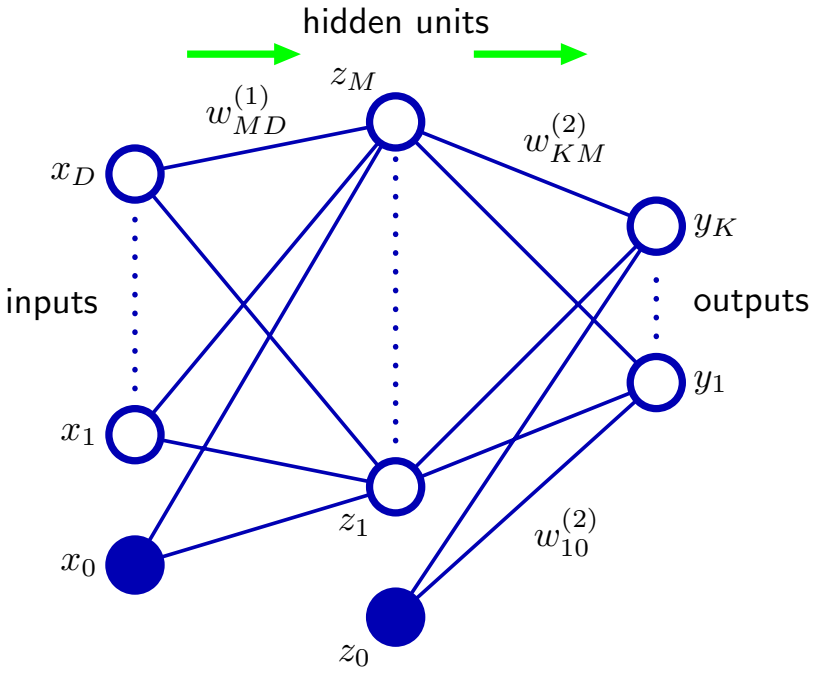
- What does **x** look like for the tilings example?
- Is tabular RL a specific case of a linear function? What would **x** be?

Non-linear function approximation

Any other type of function approximation in non-linear

A popular non-linear approximator is a neural network

E.g. feedforward network:



$$\hat{v}\left(\mathbf{x}, \mathbf{W}^{(1)}, \mathbf{W}^{(2)}\right) = \sum_{m=0}^{M} w_m^{(2)} h^{(1)} \left(\underbrace{\sum_{d=0}^{D} w_{md}^{(1)} x_d}_{a_m} \right)$$

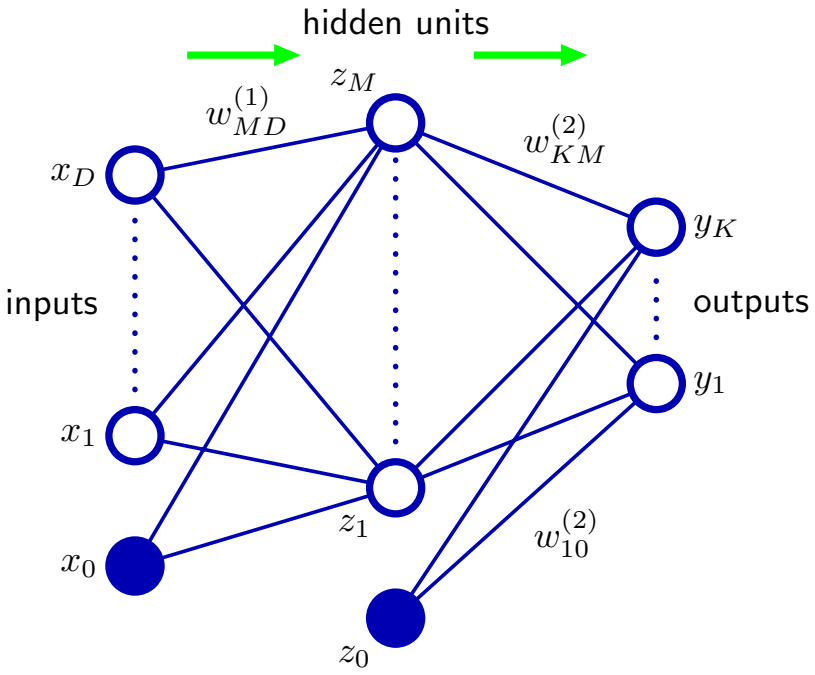
Non-linear activation function

Non-linear function approximation

Any other type of function approximation in non-linear

A popular non-linear approximator is a neural network

E.g. feedforward network:



‘weights’ ‘features’

$$\hat{v}\left(\mathbf{x}, \mathbf{W}^{(1)}, \mathbf{W}^{(2)}\right) = \sum_{m=0}^{M} w_m^{(2)} h^{(1)} \left(\underbrace{\sum_{d=0}^{D} w_{md}^{(1)} x_d}_{a_m} \right)$$

Non-linear activation function

Using the defined features

Recall the definition of gradient Monte Carlo

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha \left[G_t - \hat{v} \left(S_t, \mathbf{w}_t \right) \right] \nabla \hat{v} \left(S_t, \mathbf{w}_t \right)$$

We can simply ‘plug in’ known quantities and estimated values, but we still need $\nabla \hat{v} \left(S_t, \mathbf{w} \right)$

Using the defined features

For linear function approximation extremely easy!

$$\nabla_{\mathbf{w}} \hat{v}\left(S_t, \mathbf{w}\right) = \nabla_{\mathbf{w}} \mathbf{w}^T \mathbf{x}(s) = \mathbf{x}(s)$$

For non-linear approximation a bit more involved

- E.g. neural network

$$\hat{v}\left(\mathbf{x}, \mathbf{W}^{(1)}, \mathbf{W}^{(2)}\right) = \sum_{m=0}^{M} w_m^{(2)} h^{(1)} \left(\underbrace{\sum_{d=0}^{D} w_{md}^{(1)} x_d)}_{a_m} \right)$$

- Can figure this out: repeated application of chain rule, etc (backpropagation)
- Easy with auto-diff frameworks (PyTorch, TensorFlow, …)

Using the defined features

Linear function approximation very nice to work with

- Easy to calculate required gradient
- Update convenient form $\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[G_t - \hat{v}\left(S_t, \mathbf{w}_t\right) \right] \mathbf{x}\left(S_t\right)$

- Gradient MC

  With linear features, can prove all local optima are global optima

  So: always converges to global minimum of VE

  With non-linear functions, there might be local minima

Aside: Objective

In the end, we want to learn the best **policy**

- Not clear that lowest value error leads to best policy
- But no obvious better candidate for now…

TD with function approximation

Gradient MC:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha \left[G_t - \hat{v} \left(S_t, \mathbf{w}_t \right) \right] \nabla \hat{v} \left(S_t, \mathbf{w}_t \right)$$

Can we also do this with bootstrapping?

Could use the bootstrapping estimate as target instead of G

$$R + \gamma \hat{v} \left(S', \mathbf{w} \right)$$

TD with function approximation

Gradient MC:

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha \left[G_t - \hat{v}\left(S_t, \mathbf{w}_t\right) \right] \nabla \hat{v}\left(S_t, \mathbf{w}_t\right)$$

Can we also do this with bootstrapping?

Could use the bootstrapping estimate as target instead of G

$$R + \gamma \hat{v}\left(S', \mathbf{w}\right)$$

Semi-gradient TD(0)

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha \left[R + \gamma \hat{v}(S', \mathbf{w}_t) - \hat{v}\left(S_t, \mathbf{w}_t\right) \right] \nabla \hat{v}\left(S_t, \mathbf{w}_t\right)$$

- Gradient of TD error? No! ignores that the target depends on w, too!
- Called a *semi-gradient* method
- (using the true gradient of MSTD doesn't work well - next lecture…)

Gradient MC vs semi-gradient TD(0)

Remember: gradient MC, always converges to minimum of VE (global minimum if linear features)

How about TD(0)?

Fixed point of semi-gradient TD(0)

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left(R_{t+1} + \gamma \mathbf{x}_{t+1}^\top \mathbf{w}_t - \mathbf{x}_t^\top \mathbf{w}_t \right) \mathbf{x}_t$$

Fixed point of semi-gradient TD(0)

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t + \alpha \left(R_{t+1} + \gamma \mathbf{x}_{t+1}^\top \mathbf{w}_t - \mathbf{x}_t^\top \mathbf{w}_t\right) \mathbf{x}_t \\ &= \mathbf{w}_t + \alpha \left(R_{t+1}\mathbf{x}_t - \mathbf{x}_t \left(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}\right)^\top \mathbf{w}_t\right)\end{aligned}$$

$$\mathbb{E}[\mathbf{w}_{t+1} | \mathbf{w}_t] = \mathbf{w}_t + \alpha \left(\underbrace{\mathbb{E}[R_{t+1}\mathbf{x}_t]}_{\mathbf{b}} - \underbrace{\mathbb{E}[\mathbf{x}_t \left(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}\right)^\top]}_{\mathbf{A}} \mathbf{w}_t\right)$$

At convergence $\mathbb{E}[\mathbf{w}_{t+1} | \mathbf{w}_t] = \mathbf{w}_t$, so

$$\begin{aligned}\mathbf{b} - \mathbf{A}\mathbf{w}_{\mathrm{TD}} &= \mathbf{0} \\ \mathbf{b} &= \mathbf{A}\mathbf{w}_{\mathrm{TD}} \\ \mathbf{w}_{\mathrm{TD}} &\doteq \mathbf{A}^{-1}\mathbf{b},\end{aligned}$$

the TD fixed point

Fixed point of semi-gradient TD(0)

Semi-gradient TD with linear features can be proven to **converge** to this fixed point
(when on-policy, independent features)

With non-linear features, semi-gradient TD might **diverge**

Fixed point of semi-gradient TD(0)

Semi-gradient TD with linear features can be proven to **converge** to this fixed point
(when on-policy, independent features)

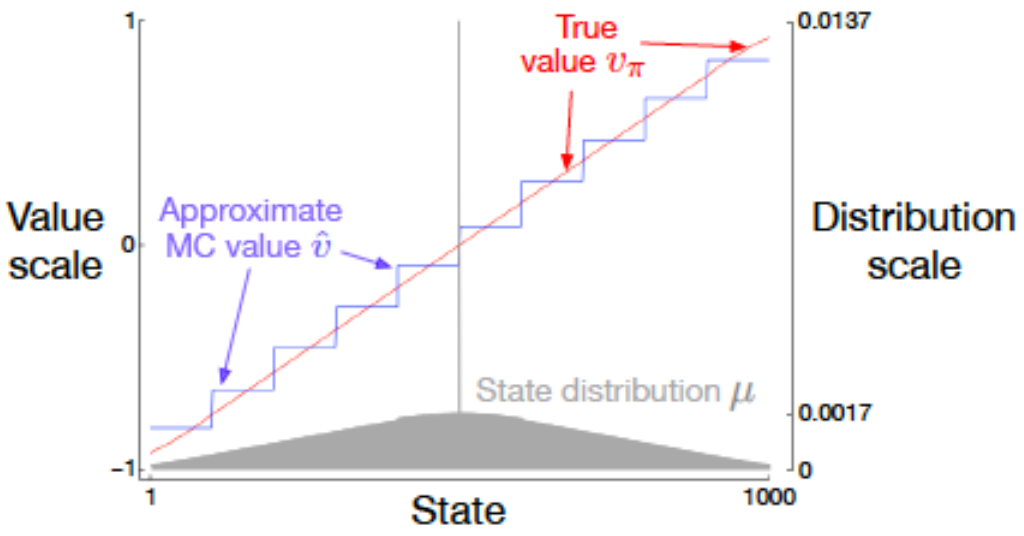
With non-linear features, semi-gradient TD might **diverge**

Unfortunately, w_{TD} is generally not the minimiser of VE

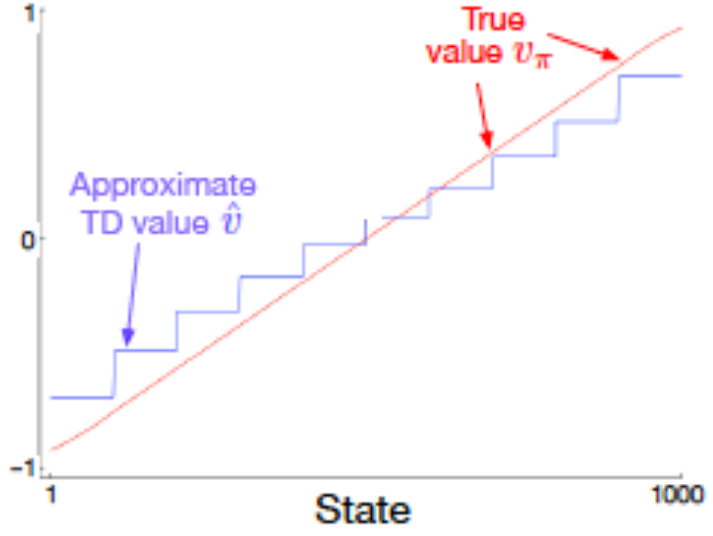
Still, semi-gradient TD converges much faster. Within a fixed amount of samples, TD can still beat MC…

Gradient MC vs semi-gradient TD(0)

Gradient MC



Semi-gradient TD(0)



Figures: Sutton & Barto. RL:AI

Like before, TD has much lower variance and tends to learn faster. However, if we can train however long we want the final results can be worse.

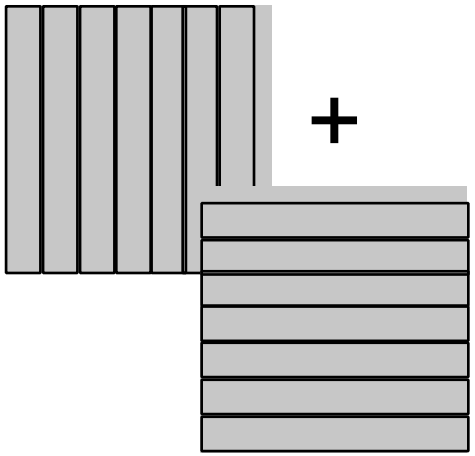
Selecting step size

Selecting step size is hard!

Tabular intuition: α=1 jumps to best solution for this experience

To average over, say, 10 experiences use α≈0.1

Function approximation bit more tricky. With e.g. two tilings and α=1, where do we end up?



(reminder:)

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[\text{target} - \hat{v}\left(S_t, \mathbf{w}_t\right)\right] \mathbf{x}\left(S_t\right)$$

Selecting step size

Take the typical norm of gradient into account?

Rule of thumb:

$$\alpha \doteq \left(\tau \mathbb{E}\left[\mathbf{x}^\top \mathbf{x}\right]\right)^{-1}$$

where τ is the number of experiences to average over.

Works best if the norm of features is close to constant

- for which discussed features is that the case?
- what could we do for the other types of features?

Least-squares temporal-difference (LSTD)

Can we get around defining stepsizes?

Try to directly find the point gradient descent will converge to…

Least-squares temporal-difference (LSTD)

Can we get around defining stepsizes?

Try to directly find the point gradient descent will converge to…

Recall the TD fixed point

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha \left(\underbrace{\mathbb{E}[R_{t+1}\mathbf{x}_t]}_{\mathbf{b}} - \underbrace{\mathbb{E}[\mathbf{x}_t \left(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}\right)^\top]}_{\mathbf{A}} \mathbf{w}_t \right)$$

$$\mathbf{w}_{\mathrm{TD}} \doteq \mathbf{A}^{-1}\mathbf{b}, \quad \text{doesn't depend on step size!}$$

Try to estimate **b** and **A** *directly from data?*

Least-squares temporal-difference (LSTD)

Use sample averages to estimate

$$\mathbf{b} = \mathbb{E}[R_{t+1}\mathbf{x}_t], \quad \mathbf{A} = \mathbb{E}[\mathbf{x}_t \left(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}\right)^\top]$$

then (ignoring the equal constant $1/t$ in the average):

$$\widehat{\mathbf{A}}_t \doteq \sum_{k=0}^{t-1} \mathbf{x}_k \left(\mathbf{x}_k - \gamma \mathbf{x}_{k+1}\right)^\top + \varepsilon \mathbf{I} \quad \text{and} \quad \widehat{\mathbf{b}}_t \doteq \sum_{k=0}^{t-1} R_{k+1}\mathbf{x}_k$$

and

$$\mathbf{w}_t \doteq \widehat{\mathbf{A}}_t^{-1}\widehat{\mathbf{b}}_t$$

Regularisation

Least-squares temporal-difference (LSTD)

More sample efficient than semi-gradient TD

More computationally intensive

- Naive inversion is cubic in d, but we can do it incrementally (quadratic)
- Semi-gradient was O(d) in memory and computation

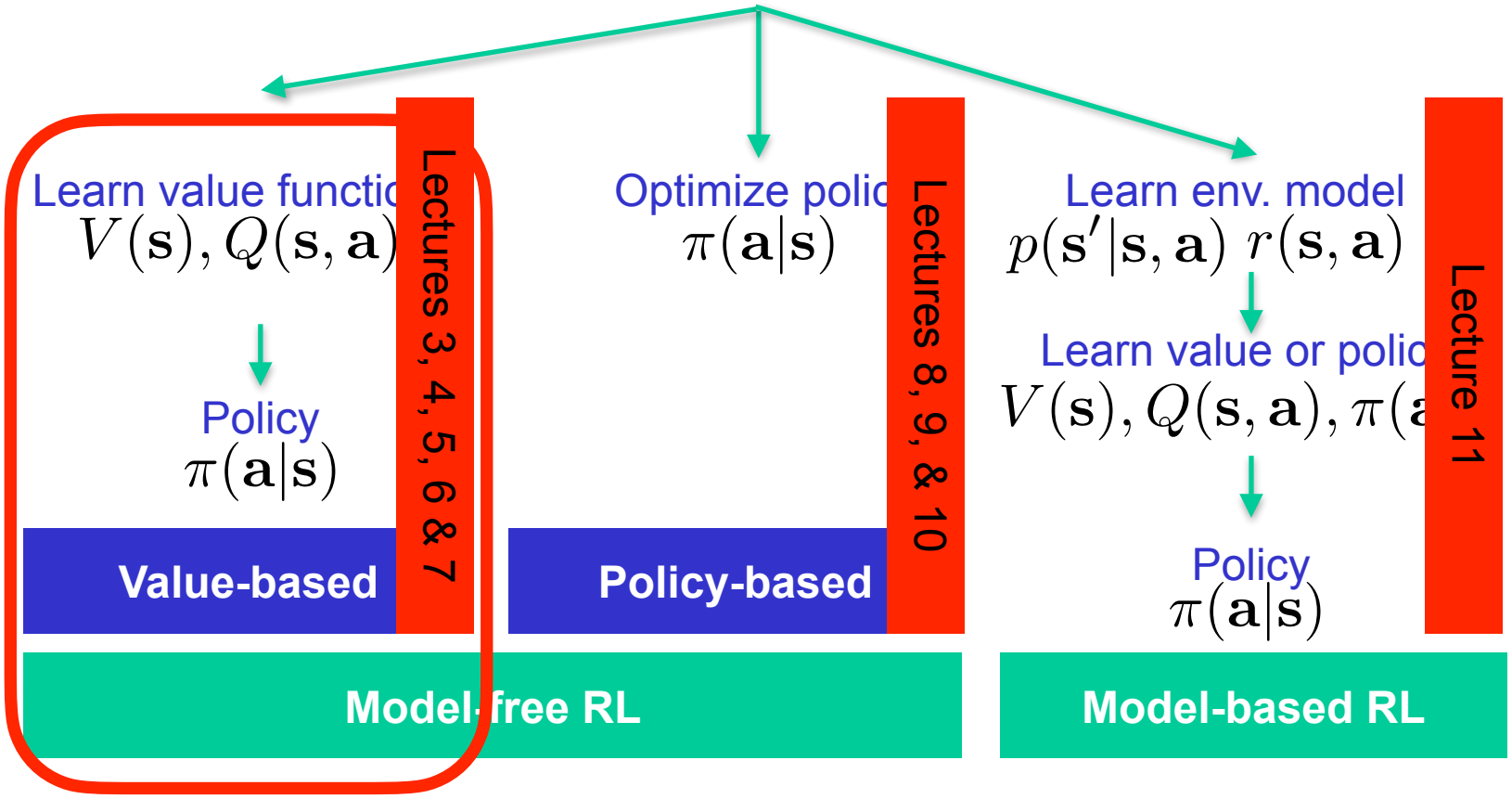
Doesn’t require step size parameter

- But it does require setting ε

LSTD never forgets

- Can be good, but means system cannot change!

Big picture: How to learn policies



Thanks to Jan Peters

Back to the big picture

We have talked so far about *prediction* methods

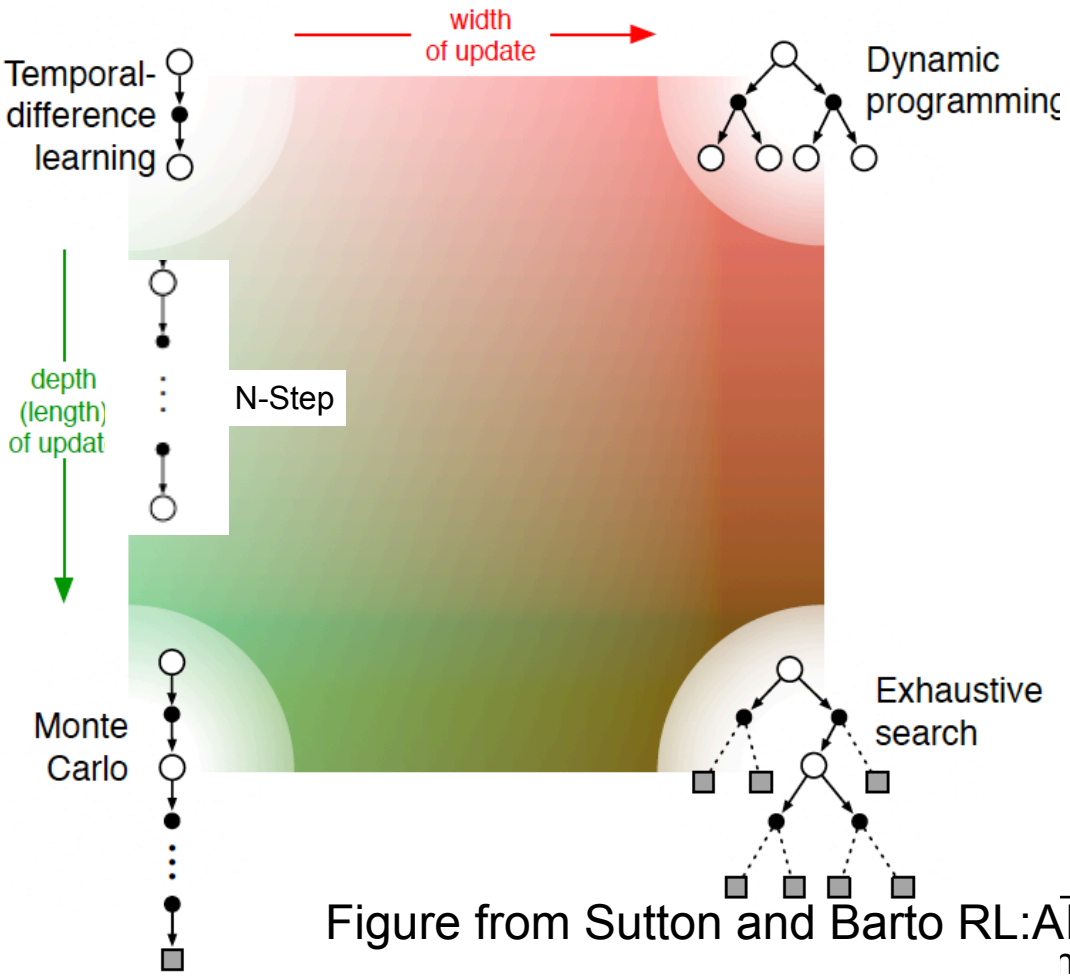


Figure from Sutton and Barto RL:AI

Back to the big picture

We have talked so far about *prediction* methods

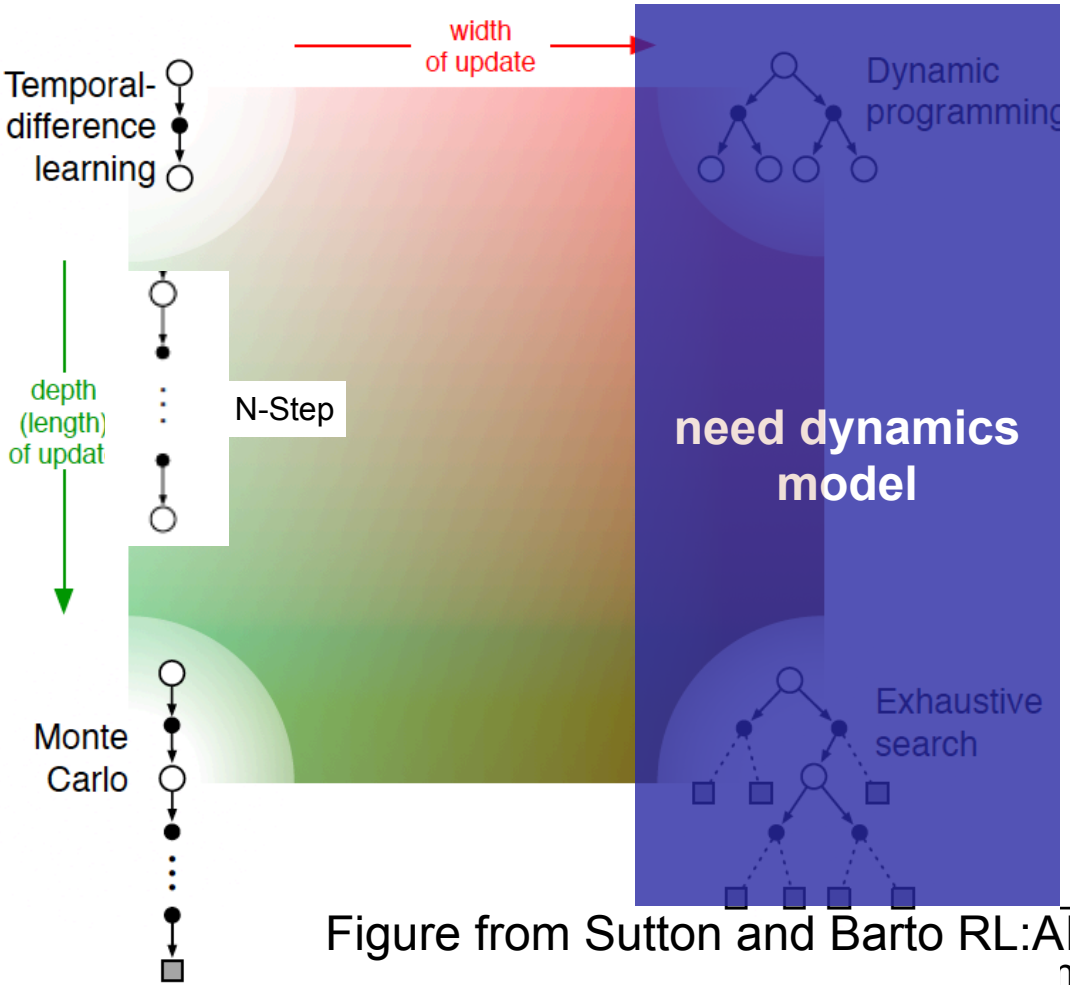


Figure from Sutton and Barto RL:AI

Back to the big picture

We have talked so far about *prediction* methods

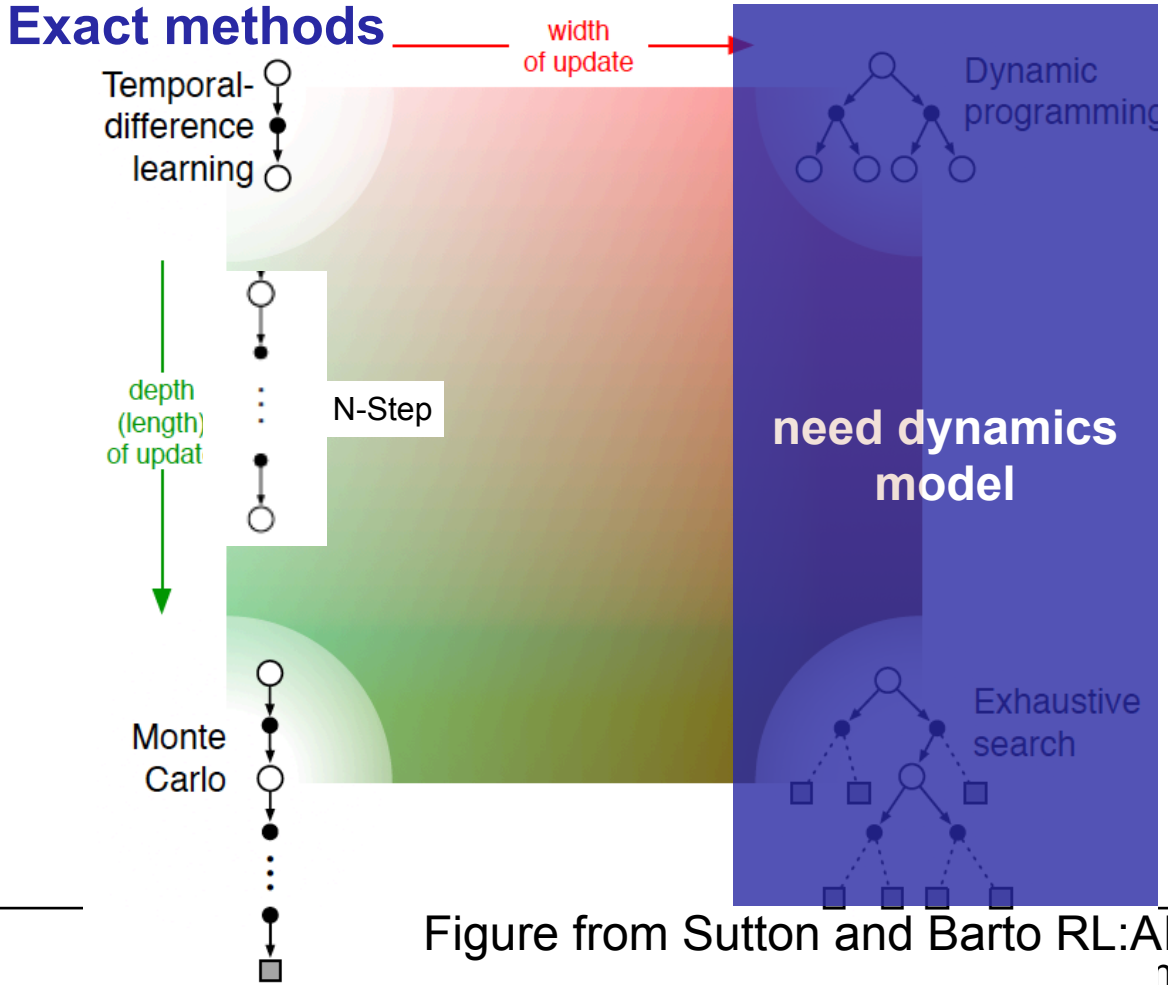


Figure from Sutton and Barto RL:AI

Back to the big picture

We have talked so far about *prediction* methods

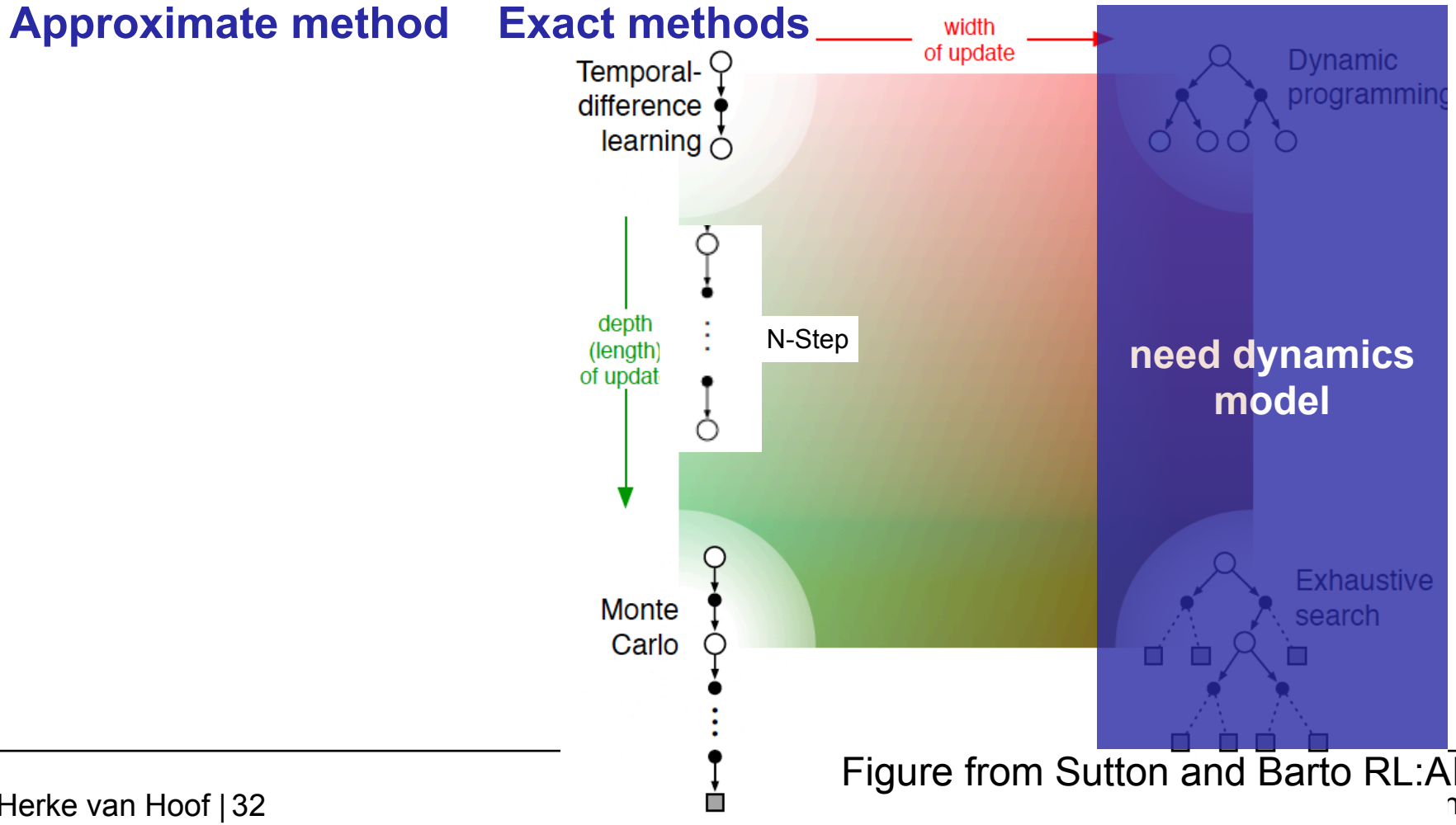


Figure from Sutton and Barto RL:AI

Back to the big picture

We have talked so far about *prediction* methods

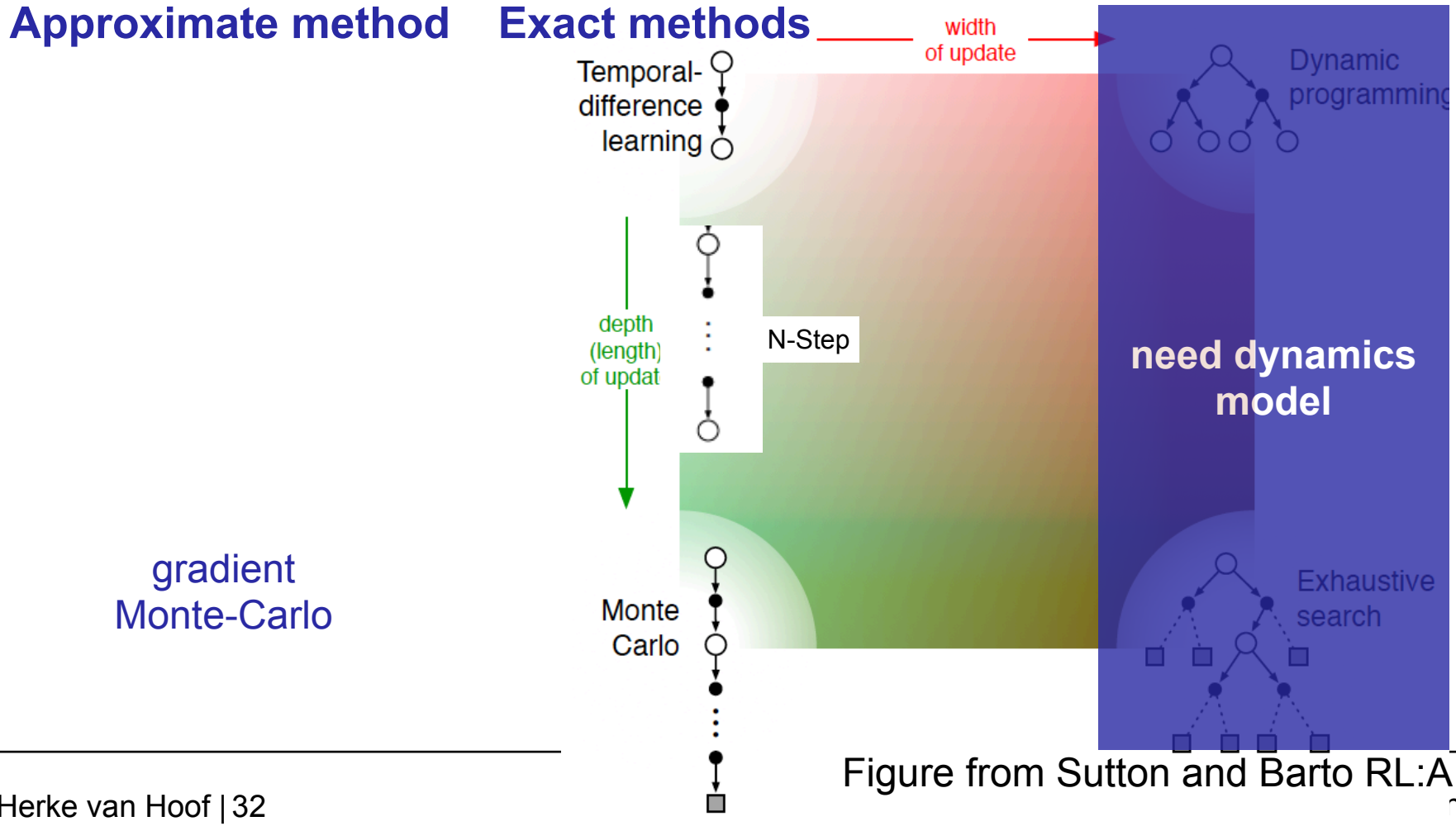


Figure from Sutton and Barto RL:AI

Back to the big picture

We have talked so far about *prediction* methods

Approximate method

semi-gradient TD(0)
& LSTD

gradient
Monte-Carlo

Exact methods

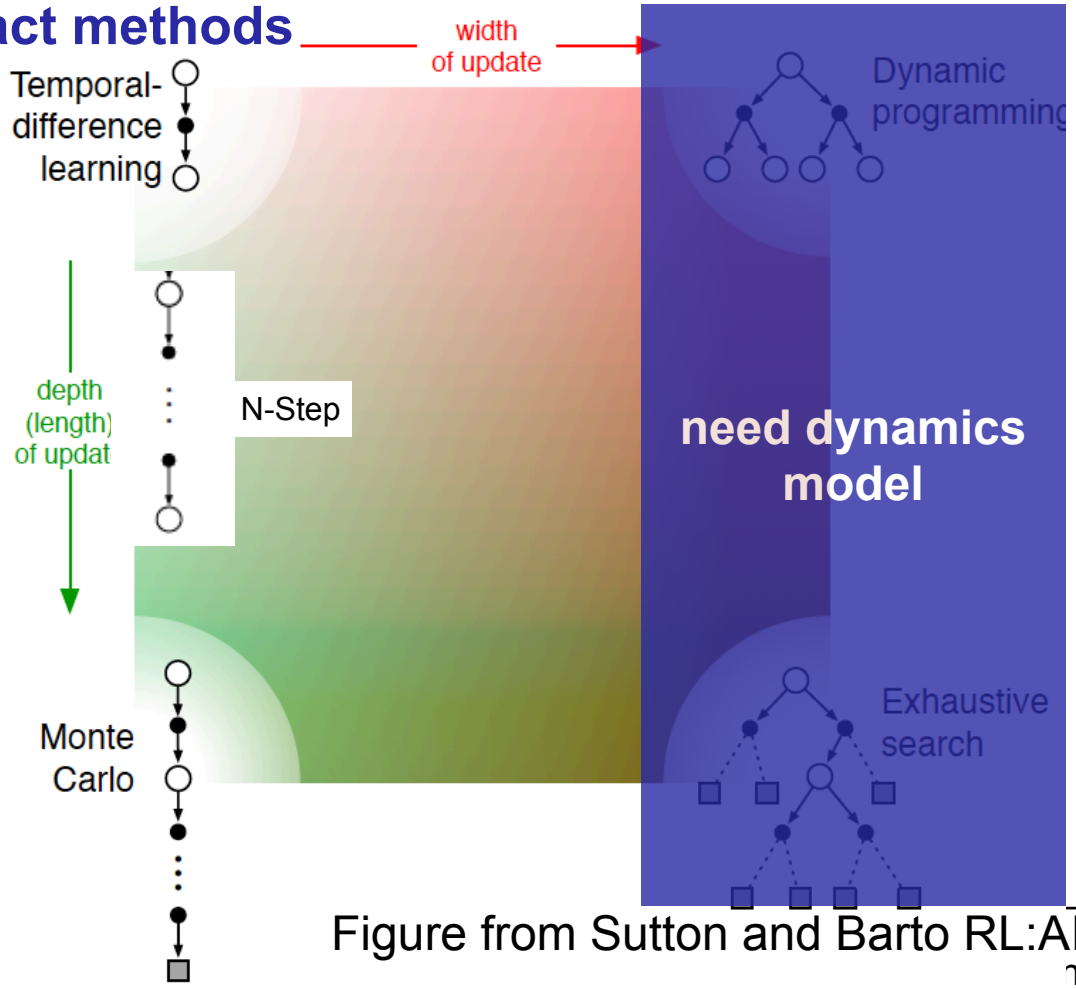


Figure from Sutton and Barto RL:AI

Back to the big picture

We have talked so far about *prediction* methods

Approximate method

semi-gradient TD(0)
& LSTD

↓

Straightforward
generalisation
to n-step
(not discussed)

gradient
Monte-Carlo

Exact methods

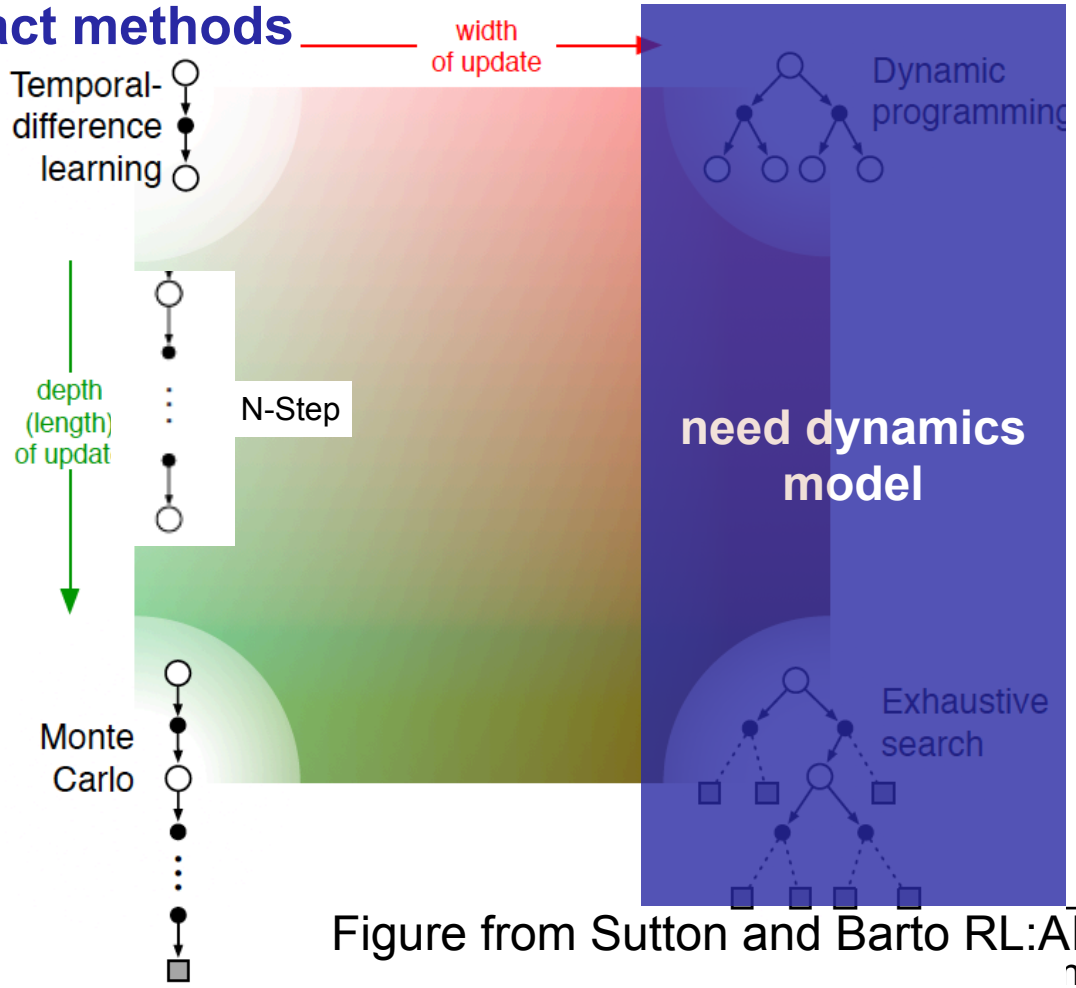


Figure from Sutton and Barto RL:AI

Back to the big picture

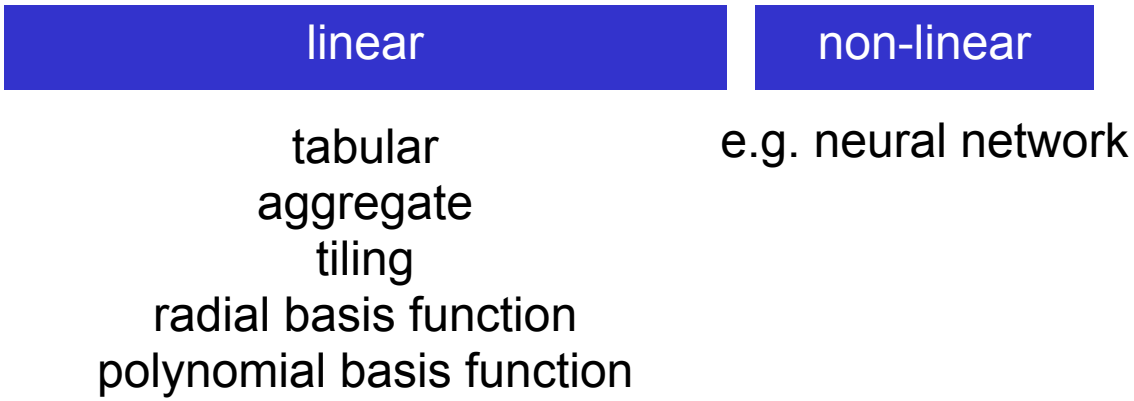
Guarantees depend on combination of approximator & method

	Tabular & Linear function approximator **	Nonlinear function approximator
Gradient MC	Convergence* to a ***global* minimum of VE**	Convergence* to a ***local* minimum of VE**
Semi-gradient TD	Convergence* to **TD fixed point**	No guarantee of convergence
LSTD	Convergence to **TD fixed point**	-

* with on-policy data and appropriate step-size schedule
** if features independent, single solution

Back to the big picture

For any of the methods (gradient MC / semi-gradient TD), choice of function approximation



Back to the big picture

Main problems:

- prediction
- on-policy control
- off-policy control

Back to the big picture

Main problems:	Tabular (Exact)	Approximate
• prediction	lecture 2 - 5	lectures 5, 6 (today)
• on-policy control	lecture 2 - 5	lecture 6 (today)
• off-policy control	lecture 2 - 5	lecture 7 (next)

Control with approximation

So far, we've looked at learning v functions with approximation

Let's try the same strategy for learning q functions!

We'll focus on **episodic** tasks
(Continuing tasks requires a couple more modifications)

Episodic semi-gradient Sarsa

As before we can define a semi-gradient method

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha \left(R_t + \hat{q}\left(S_{t+1}, A_{t+1}, \mathbf{w}_t\right) - \hat{q}\left(S_t, A_t, \mathbf{w}_t\right) \right) \nabla \hat{q}\left(S_t, A_t, \mathbf{w}_t\right)$$

Here with SARSA target (n-step also possible)

We are still on-policy, so we iterate

- Take action from soft approximation of greedy policy (e.g. ε-greedy)
- Update $\mathbf{w}$ to improve Q^{π} estimate

Episodic semi-gradient Sarsa

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

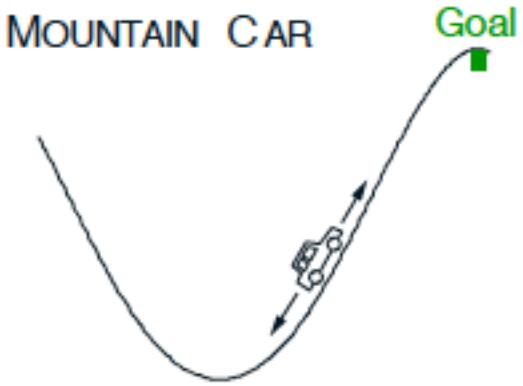
Input: a differentiable action-value function parameterization $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameters: step size $\alpha > 0$, small $\varepsilon > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

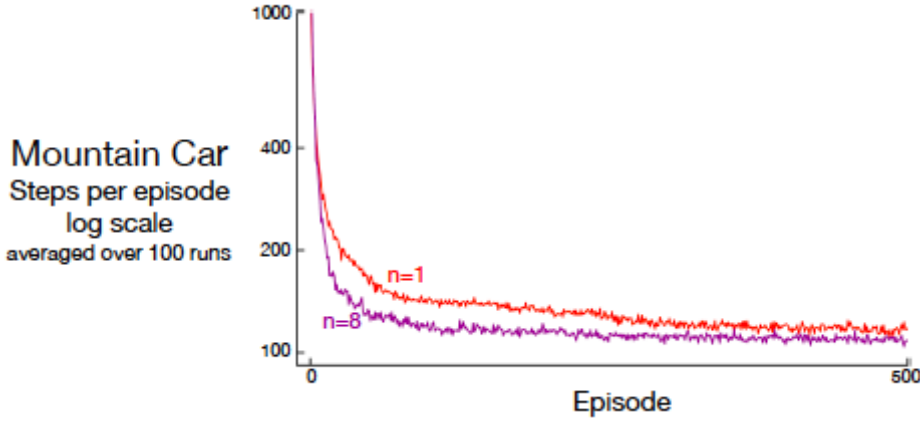
Loop for each episode:
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Loop for each step of episode:
    Take action A, observe R, S'
    If S' is terminal:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \mathbf{w})$ (e.g., ε-greedy)
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{q}(S', A', \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\big]\nabla\hat{q}(S, A, \mathbf{w})$
    $S \leftarrow S'$
    $A \leftarrow A'$

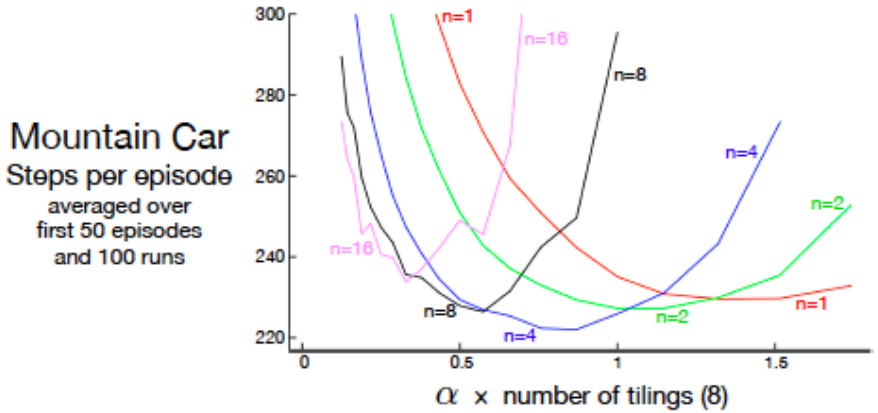
Figures: Sutton & Barto. RL:AI

Episodic semi-gradient Sarsa

As before, we can use n-step Sarsa as well







Figures: Sutton & Barto. RL:AI

What you should know

What are linear and non-linear function approximators?

What are semi-gradient TD and LSTD, and how do they relate to gradient MC? What is the TD fix point?

How can we do on-policy control using semi-gradient SARSA?

Thanks for your attention

Feedback?

h.c.vanhoof@uva.nl

Extra: Aggregation and tiling

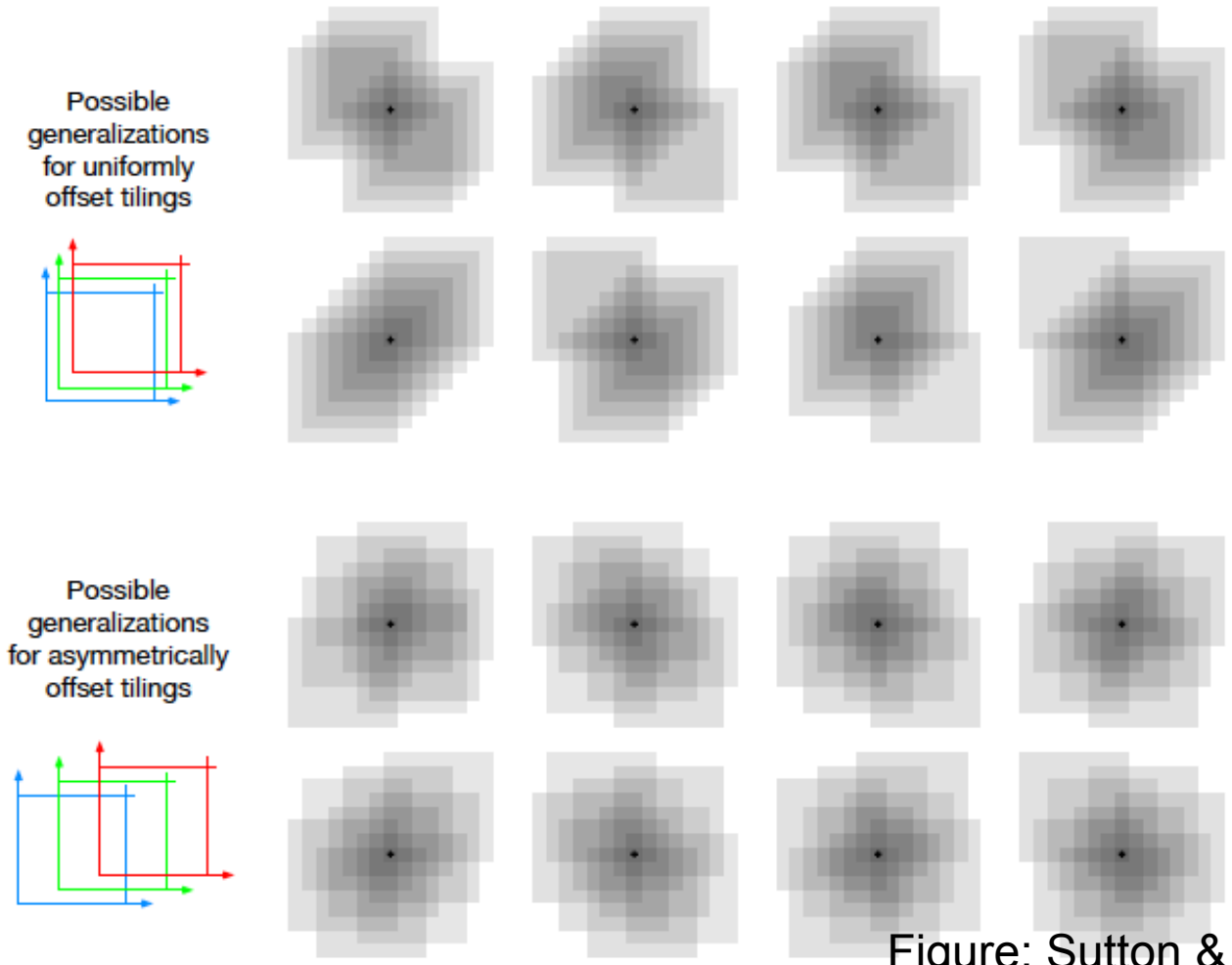


Figure: Sutton & Barto. RL:AI

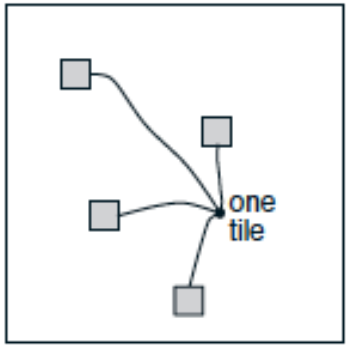
Extra: Aggregation and tiling

Special aggregations

Irregular

Log stripes

Diagonal stripes



Hashing

Figure: Sutton & Barto. RL:AI

Generic approaches: Fourier basis

In multi-d, consider interaction between original dimensions

n: max frequency in any direction; *k*: # original dimensions

$$x_i(s) = \cos\left(\pi \mathbf{s}^\top \mathbf{c}^i\right)$$

$$\mathbf{c}^0 = [0, 0, \ldots]^T$$

$$\mathbf{c}^1 = [1, 0, \ldots]^T$$

$$\mathbf{c}^2 = [0, 1, \ldots]^T$$

$$\vdots$$

$$\mathbf{c}^{(n+1)^k} = [n, n, \ldots]^T$$

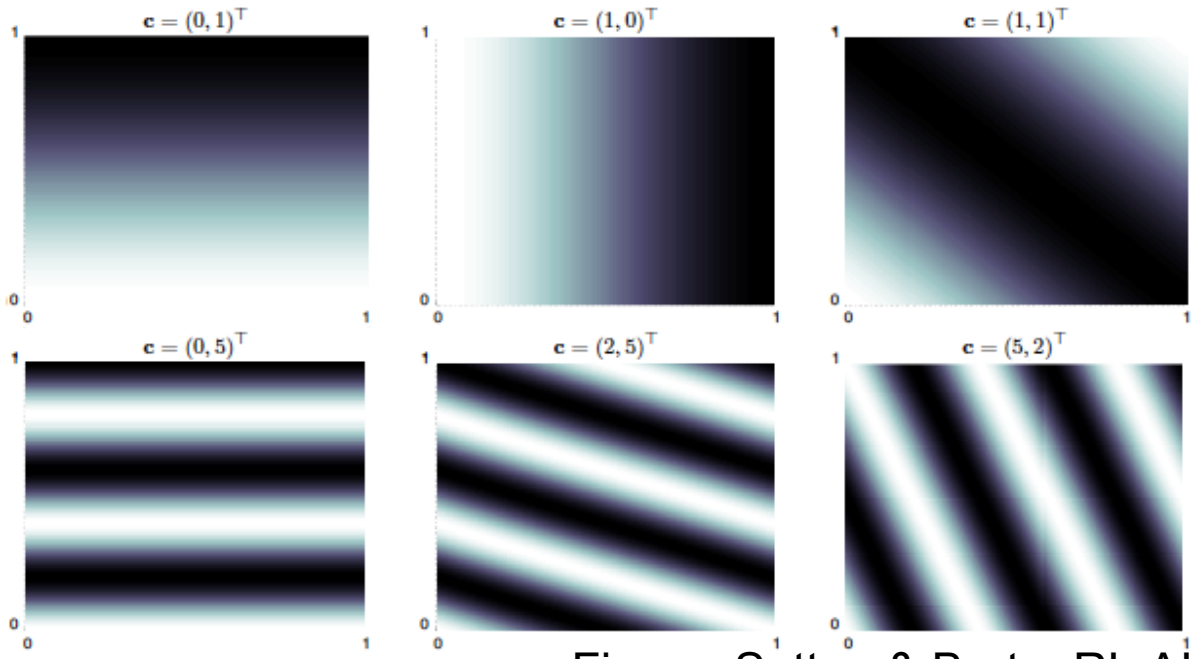


Figure: Sutton & Barto. RL:AI

Extra: Non-parametric approximation

Another way to approximate value functions is to directly use the training data rather than a parametric function

- Nearest neighbour
- Weighted average of n-nearest neighbour
- Locally weighted regression

More flexible, more precise where there is a lot of data

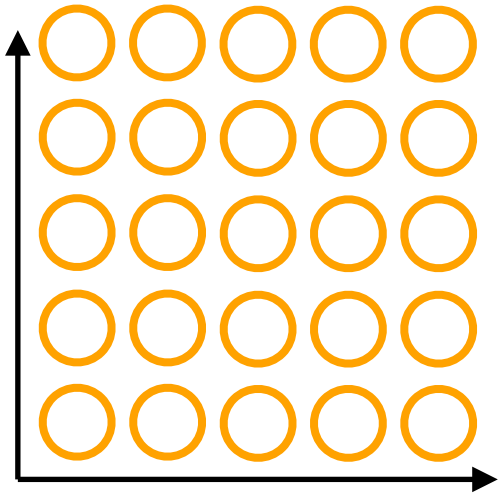
However, finding the nearest neighbours at every step is usually more expensive than evaluating a parametric function

Extra: Non-parametric approximation

Radial basis functions

$$\hat{v}(\mathbf{s}, \mathbf{w}) = \sum_i w_i \exp\left(-\frac{\|\mathbf{s} - \mathbf{c}_i\|^2}{2\sigma_i^2}\right)$$

predefined features



Stationary kernel

$$\hat{v}(\mathbf{s}, \mathbf{w}) = \sum_i w_i \exp\left(-\frac{\|\mathbf{s} - \mathbf{s}_i\|^2}{2\sigma_i^2}\right)$$

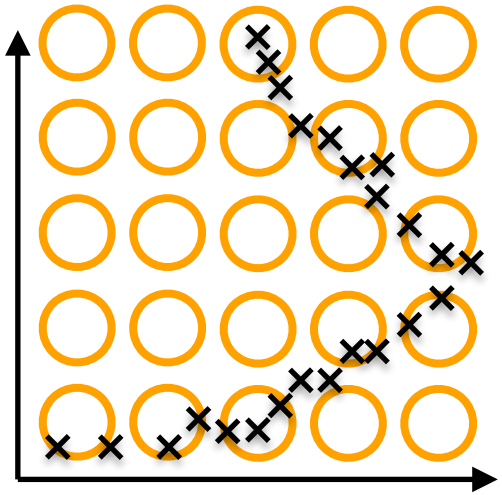
data points

Extra: Non-parametric approximation

Radial basis functions

$$\hat{v}(\mathbf{s}, \mathbf{w}) = \sum_i w_i \exp\left(-\frac{\|\mathbf{s} - \mathbf{c}_i\|^2}{2\sigma_i^2}\right)$$

predefined features



Stationary kernel

$$\hat{v}(\mathbf{s}, \mathbf{w}) = \sum_i w_i \exp\left(-\frac{\|\mathbf{s} - \mathbf{s}_i\|^2}{2\sigma_i^2}\right)$$

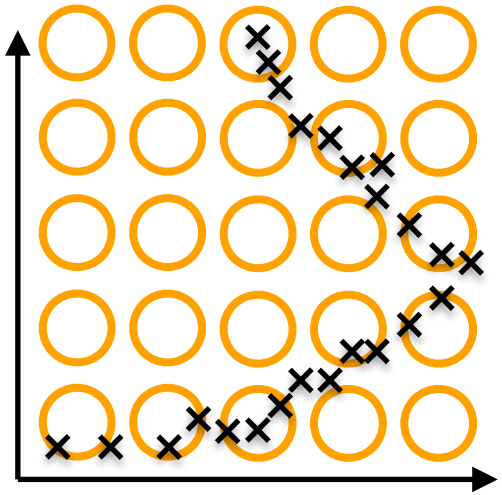
data points

Extra: Non-parametric approximation

Radial basis functions

$$\hat{v}(\mathbf{s}, \mathbf{w}) = \sum_i w_i \exp\left(-\frac{\|\mathbf{s} - \mathbf{c}_i\|^2}{2\sigma_i^2}\right)$$

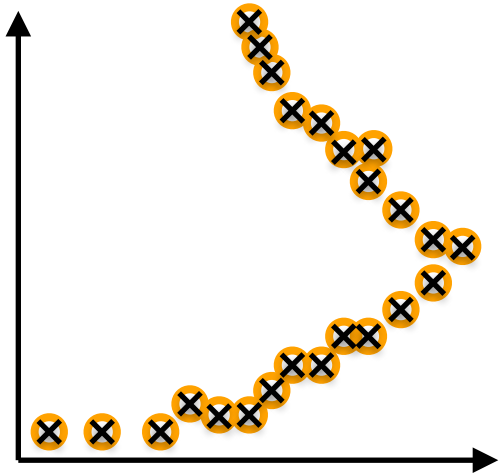
predefined features



Stationary kernel

$$\hat{v}(\mathbf{s}, \mathbf{w}) = \sum_i w_i \exp\left(-\frac{\|\mathbf{s} - \mathbf{s}_i\|^2}{2\sigma_i^2}\right)$$

data points

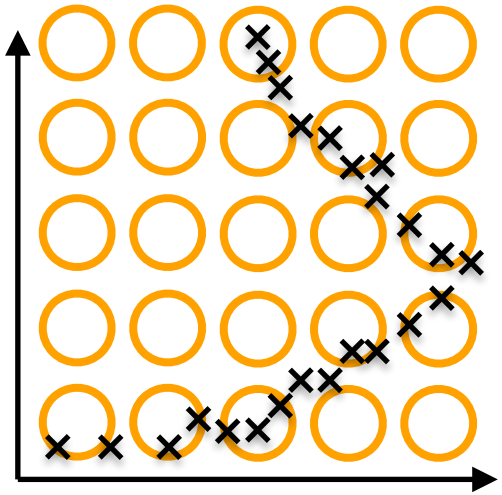


Extra: Non-parametric approximation

Radial basis functions

$$\hat{v}(\mathbf{s}, \mathbf{w}) = \sum_i w_i \exp\left(-\frac{\|\mathbf{s} - \mathbf{c}_i\|^2}{2\sigma_i^2}\right)$$

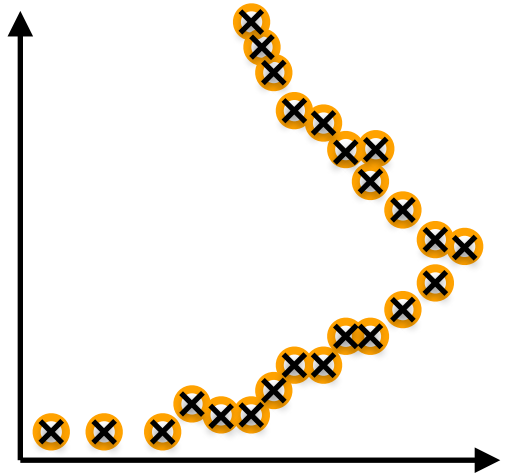
predefined features



Stationary kernel

$$\hat{v}(\mathbf{s}, \mathbf{w}) = \sum_i w_i \exp\left(-\frac{\|\mathbf{s} - \mathbf{s}_i\|^2}{2\sigma_i^2}\right)$$

data points



The kernel based approach has many of the nice properties of the linear function approximation case, while being highly flexible (more info in ML1)