Prediction with approximation

Herke van Hoof

Why approximation

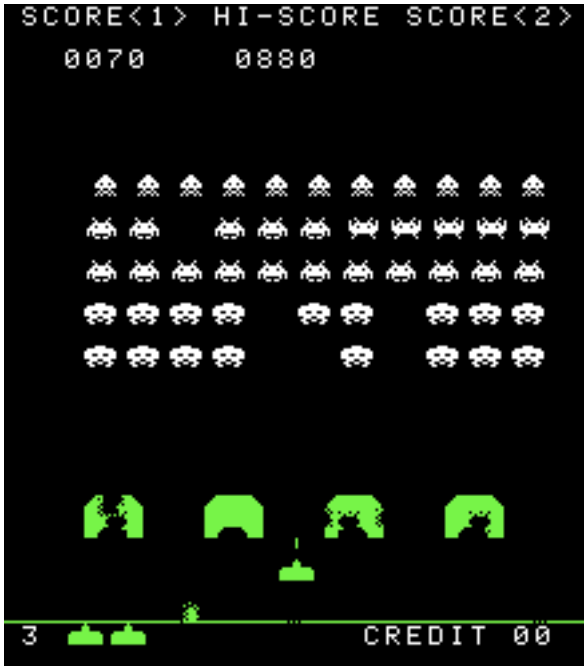
So far, we have looked at ‘small’ problems: can store the Q/V fc for every state

With a huge number of states, two problems:

- The table might be too large to fit in memory
- Before memory becomes a problem, it might take too long to collect enough experiences to fill up every row in the table
- The state space might be continuous (infinitely many states)

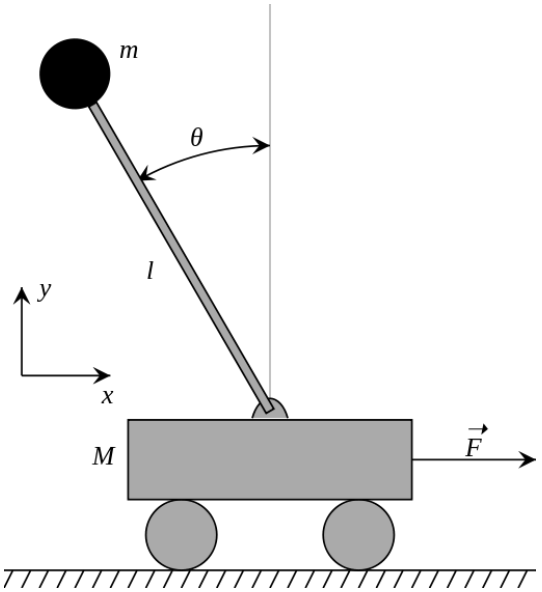
Today, we’ll look at how to overcome these problems in the *policy evaluation* (i.e., prediction) setting. *Control* next week!

Why approximation



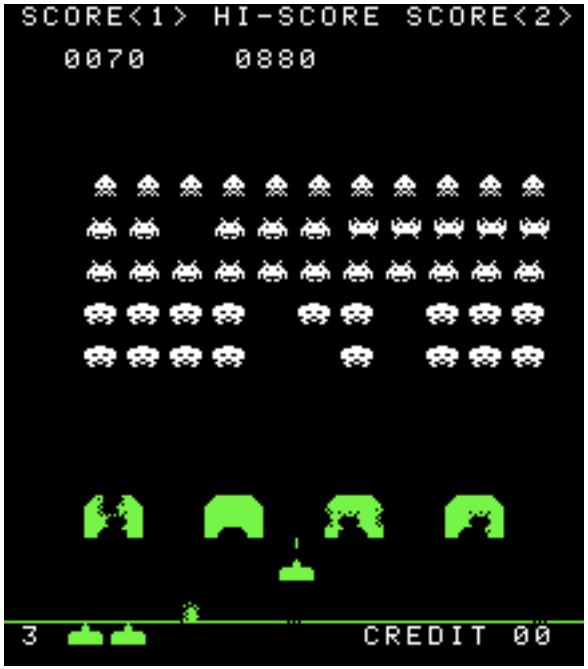
Atari games

Go



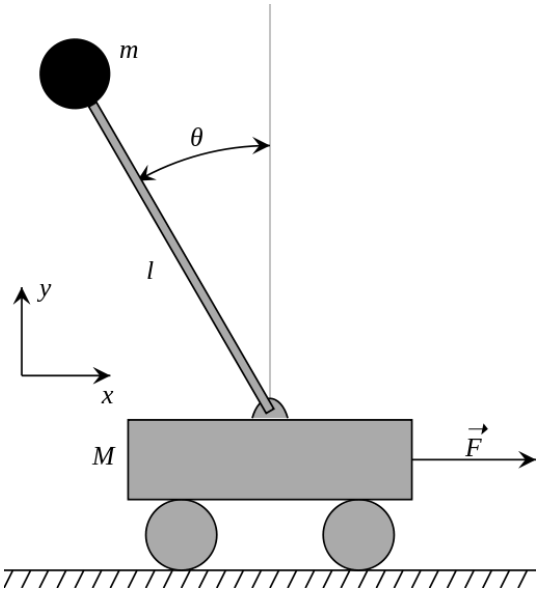
Physical systems

Why approximation



Atari games

Go



Physical systems

Luckily, certain small changes in state usually don't impact the value/action too much. **We can generalize.**

Why approximation

So we would like to represent the value function

- In a compact way
- In a way that allows generalizing an experience to nearby states

This is similar to the goal of supervised learning

- Very flexible function might not generalise well (overfitting)
- Inflexible functions are not very expressive
- Need to find a good balance, need for expressivity <> data availability

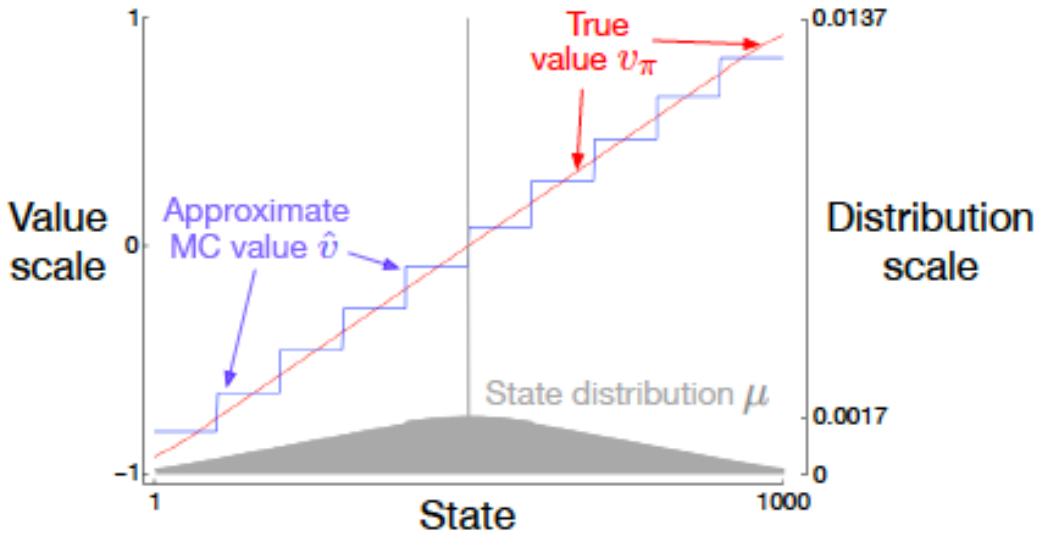
In general, less flexible function means cannot represent exactly

- Thus, we aim to approximate the true value function

An example

Let's consider just splitting the set of all states into groups

We'll only store one representative value for each group



Approximate value parametrised by this values **w**: $\hat{v}(s, \mathbf{w})$

Figure: Sutton & Barto. RL:AI

Objective

What are the ‘best’ representative values **w**?

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

Not all states are equally important!

Objective

What are the ‘best’ representative values **w**?

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

Not all states are equally important!

Today: On-policy distribution

- Continuing tasks: Stationary distribution

$$\mu_\pi(\mathbf{s}) = \sum_{\bar{\mathbf{s}}} \sum_{\mathbf{a}} p(\mathbf{s}|\bar{\mathbf{s}}, \mathbf{a}) \pi(\mathbf{a}|\bar{\mathbf{s}}) \mu_\pi(\bar{\mathbf{s}})$$

Objective

What are the 'best' representative values **w**?

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

Not all states are equally important!

Today: On-policy distribution

- Continuing tasks: Stationary distribution

$$\mu_\pi(\mathbf{s}) = \sum_{\bar{\mathbf{s}}} \sum_{\mathbf{a}} p(\mathbf{s}|\bar{\mathbf{s}}, \mathbf{a}) \pi(\mathbf{a}|\bar{\mathbf{s}}) \mu_\pi(\bar{\mathbf{s}})$$

- Episodic task: consider initial state distribution h:

$$\mu_\pi(s) = \frac{\eta_\pi(s)}{\sum_{s'} \eta_\pi(s')}, \qquad \eta_\pi(s) = h(s) + \sum_{\bar{s}} \eta_\pi(\bar{s}) \sum_{a} \pi(a|\bar{s}) p(s|\bar{s}, a)$$

fraction of steps spend in s

Average number of steps spend in s per episode

Stochastic gradient descent

Let's follow the negative gradient! But how to approximate

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

Stochastic gradient descent

Let's follow the negative gradient! But how to approximate

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

- Pick random samples from *μ(s)* (how?)
- Generate unbiased estimates from v_π (how?)

Stochastic gradient descent

Let's follow the negative gradient! But how to approximate

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

$\nabla \mathbb{E} \sum_i [G(s_i) - \hat{v}(s_i, w)]^2 \qquad s_i \sim \mu_\pi(s)$

$-\sum 2 [G(s_i) - \hat{v}(s_i, w)] \cdot \nabla \hat{v}(s_i, w)$

- Pick random samples from *μ(s)* (how?)

  As *μ* is on-policy distribution, simply pick random visited state

- Generate unbiased estimates from v_π (how?)

  Use the return *G*

Stochastic gradient descent

This leads to gradient Monte Carlo algorithm

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

- (Use samples and returns from on-policy distribution)

Objective

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

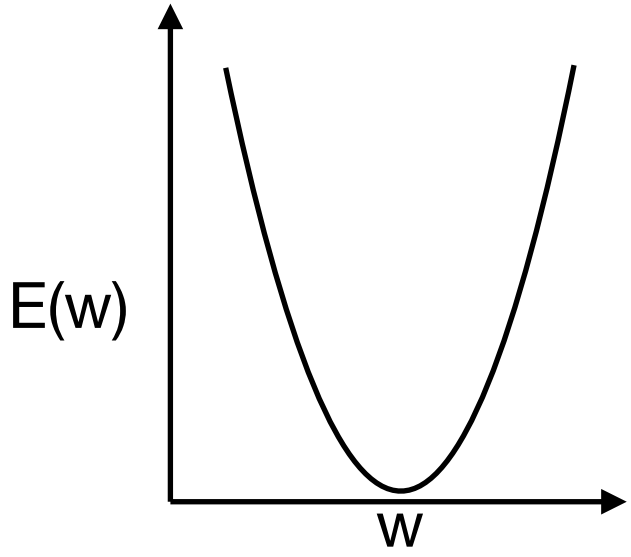
- Not clear that lowest value error leads to best policy
- But no obvious better candidate for now…

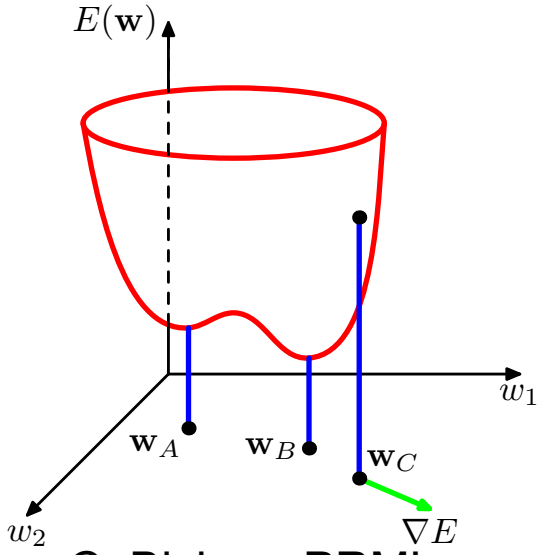
Objective

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…

Local vs. global optima





C. Bishop; PRML

TD with function approximation

Can we also do this with bootstrapping?

Could use the bootstrapping estimate as target

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

Semi-gradient TD(0)

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

TD with function approximation

Can we also do this with bootstrapping?

Could use the bootstrapping estimate as target

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

Semi-gradient TD(0)

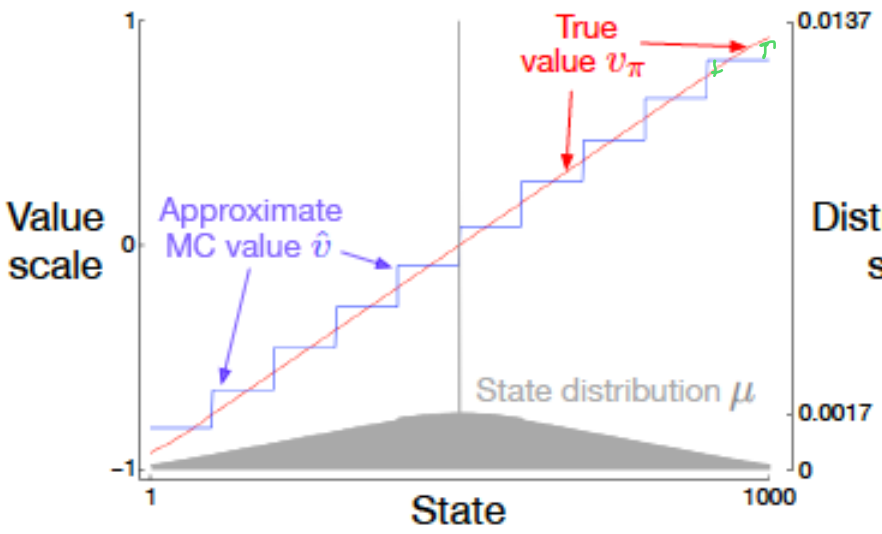
$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha \left[R + \gamma \hat{v}(S', \mathbf{w}_t) - \hat{v}\left(S_t, \mathbf{w}_t\right)\right] \nabla \hat{v}\left(S_t, \mathbf{w}_t\right)$$

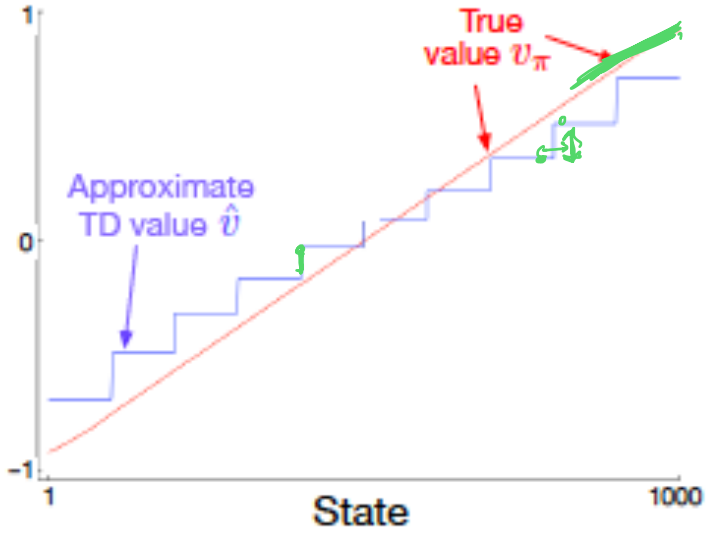
- this ignores that the target depends on w, too!
- Doesn't minimise MSTD error! Called a *semi-gradient* method
- (using the true gradient of MSTD doesn't work well - see next lecture…)

Gradient MC vs semi-gradient TD(0)

Gradient MC

Semi-gradient TD(0)

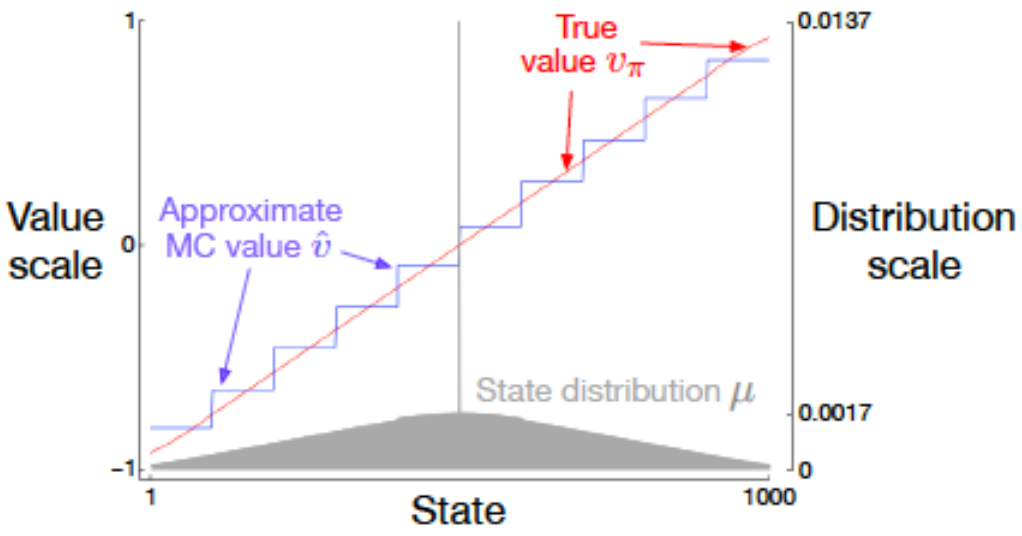




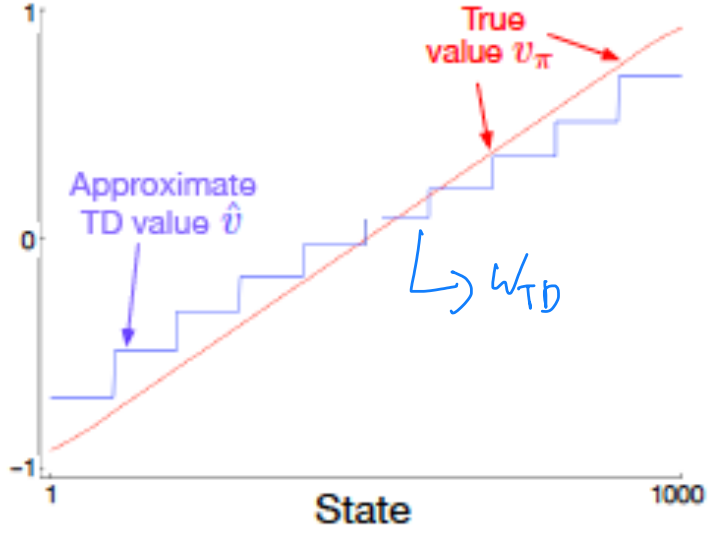
Figures: Sutton & Barto. RL:AI

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, in the long run gradient MC tends to find better solutions.

Types of function approximator

$$V_\pi = \hat{V}(s, w)$$

Types of function approximator

Linear function approximation

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

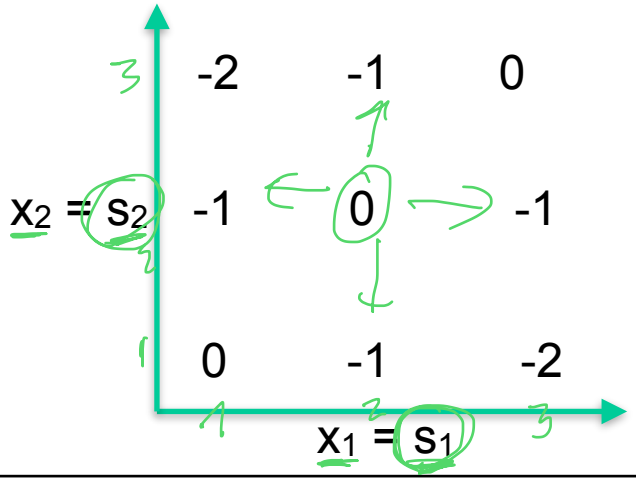
Linear function approximation

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



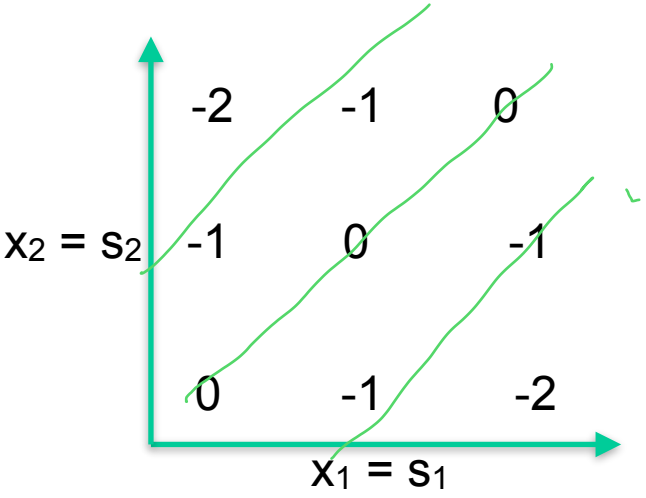
Linear function approximation

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

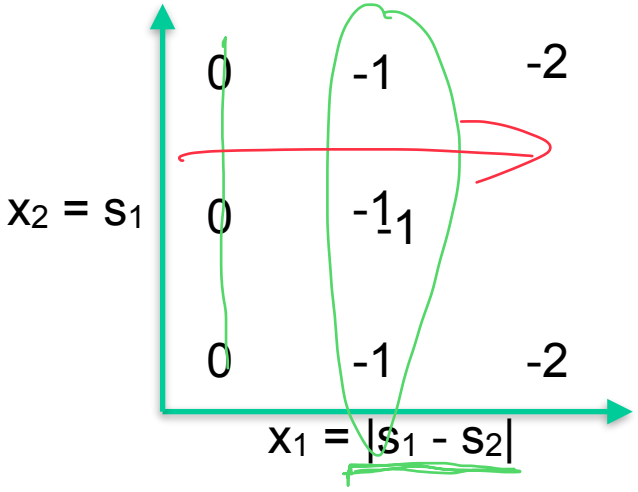
What is important depends on the task!

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

True value, unity transformation



True value, non-linear transformation



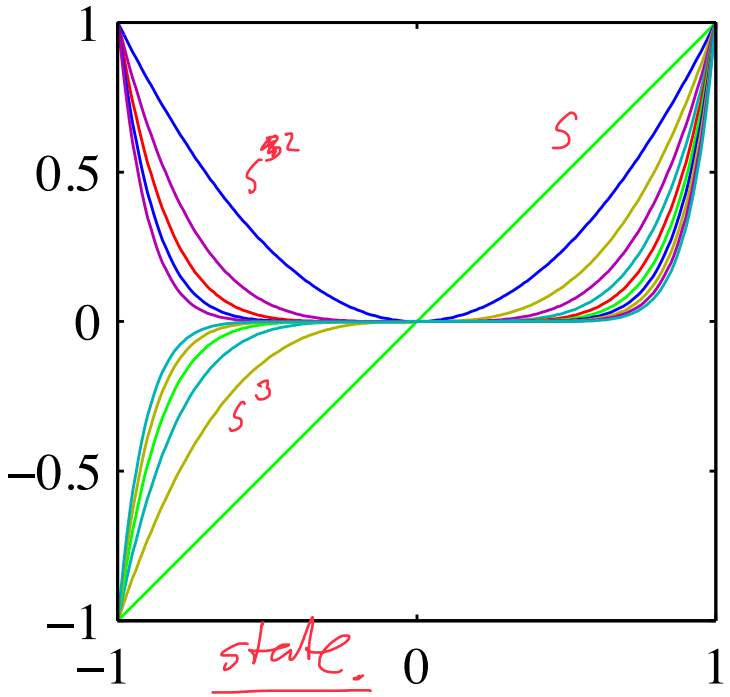
Polynomials

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

But with many features, less generalisation



C.M. Bishop: PRML

Polynomials

Have to choose “order” of polynomial, e.g. highest power of any of the ‘raw’ state dimensions that we’ll consider

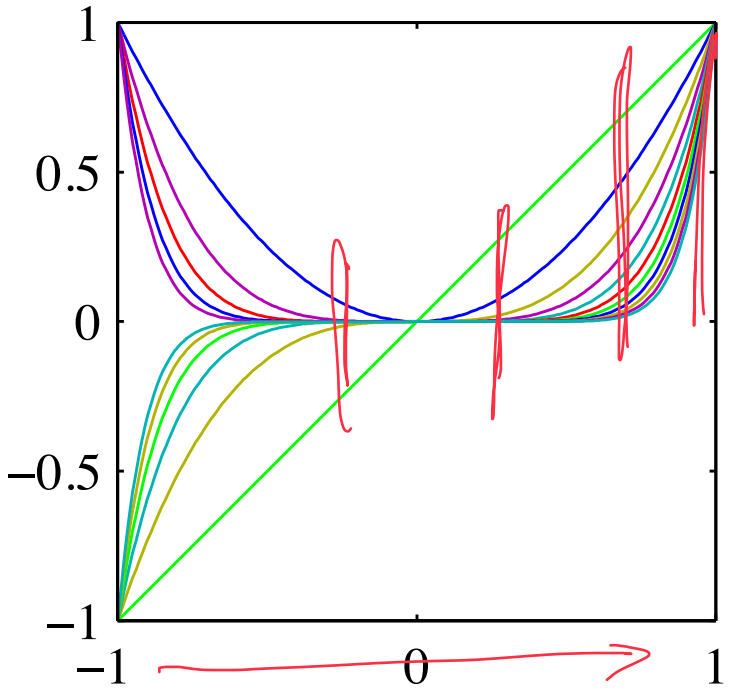
In one-d, 11th order: see image

In two-d, second order

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

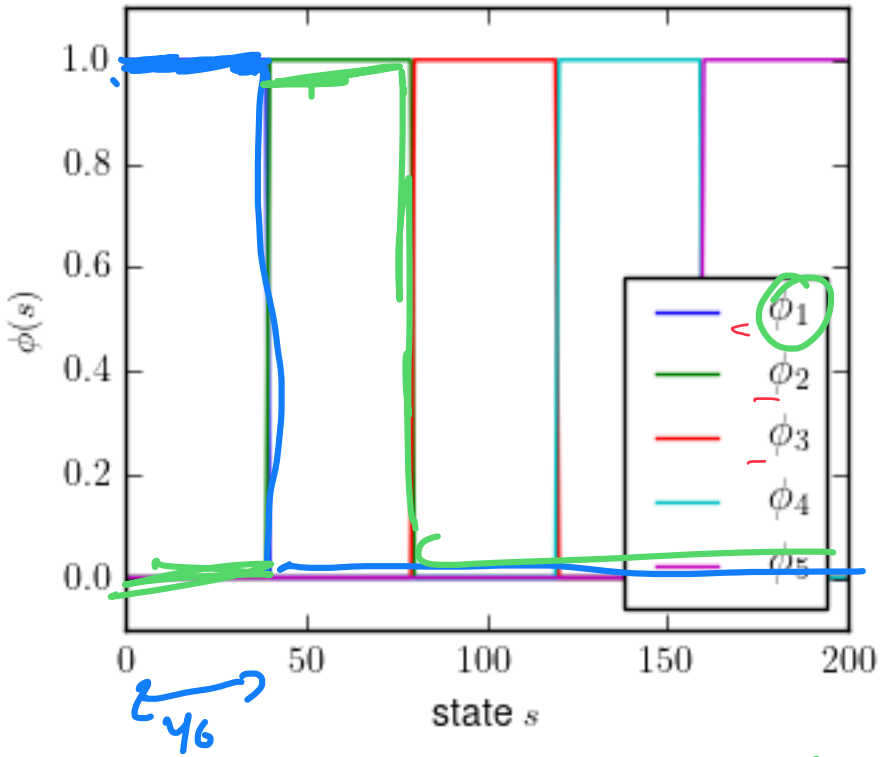
Note: include ‘constant’ feature!

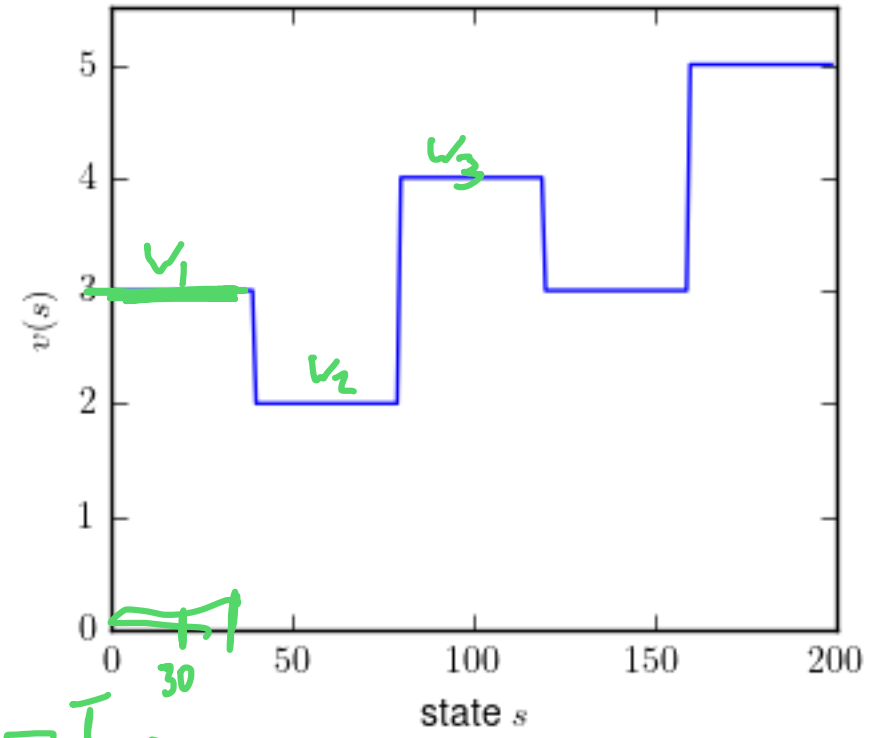
But:
0 looks very different than extremes!



C.M. Bishop: PRML

Aggregation and tiling





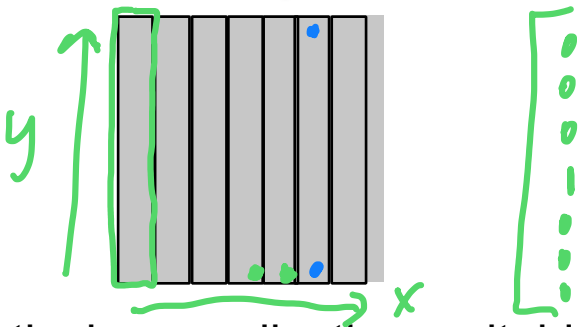
$V(s,w) = w^T x(s) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}^T \begin{bmatrix} w_1 \\ w_2 \\ w_3 \\ w_4 \\ w_5 \end{bmatrix} = w_1$

Aggregation and tiling

Different types of aggregation

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

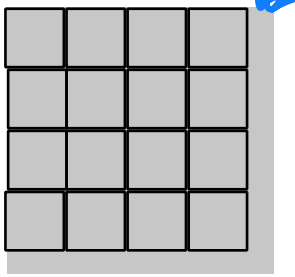
larger / continuous 2-d state-space



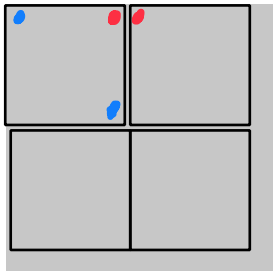
Vertical generalization: suitable if y-coordinate does not matter

Aggregation and tiling

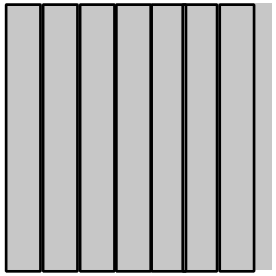
Different types of aggregation



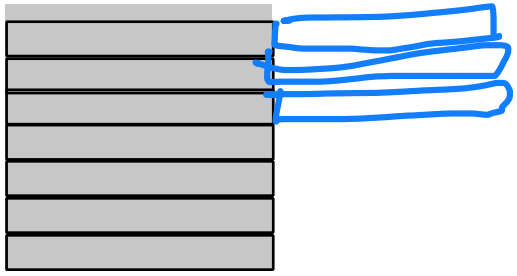
Both coordinates matter, relatively fine discretisation (little generalization)



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



Vertical generalization: suitable if y-coordinate does not matter

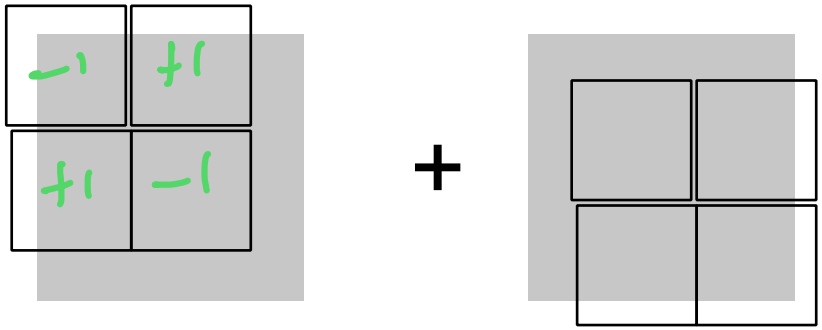


Horizontal generalization: suitable if x-coordinate does not matter

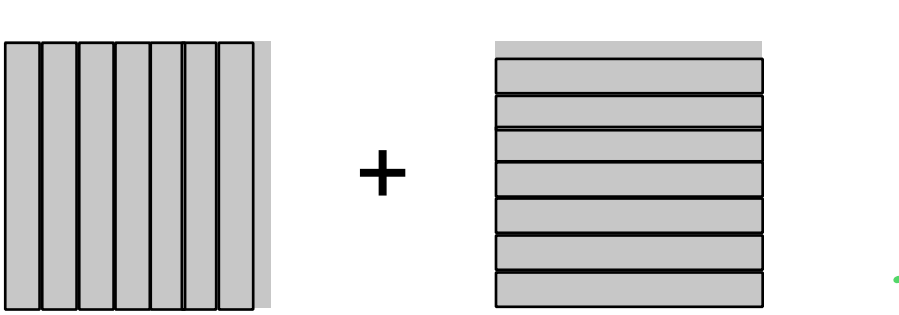
Aggregation and tiling

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

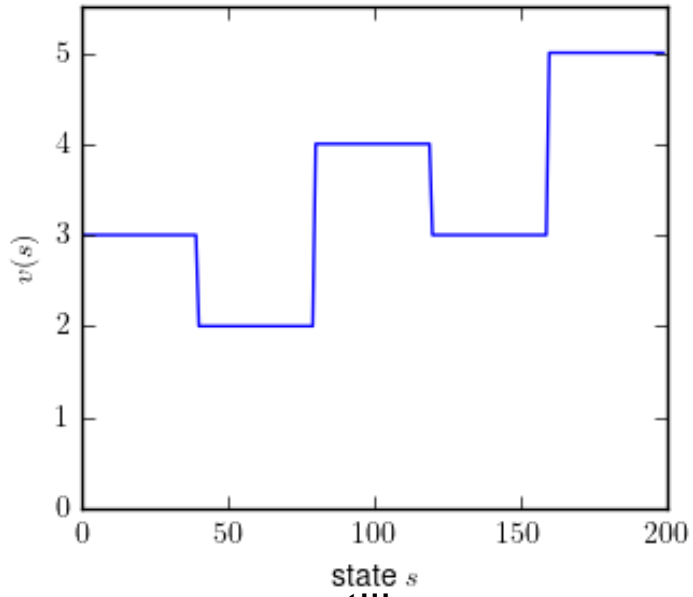


Generalize in x and y directions, but avoid modelling interaction terms

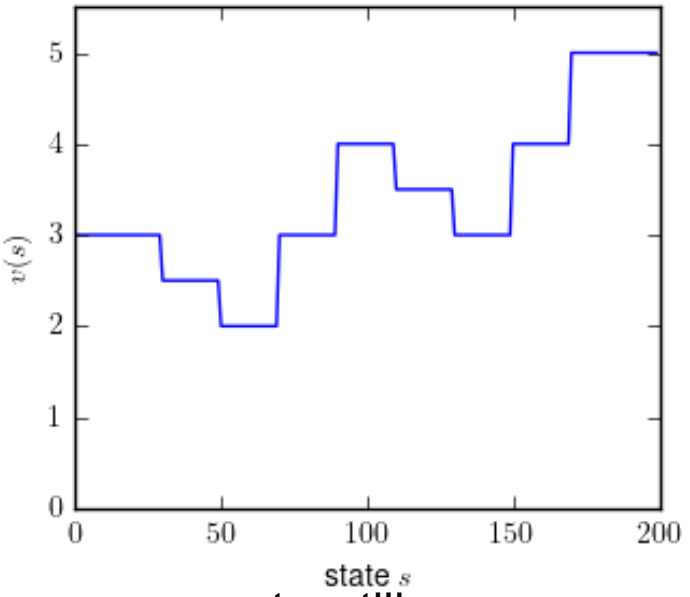
Radial basis functions

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

Tiling is a bit better. We now make small jumps for each of several aggregations.



one tiling



two tilings

Radial basis functions

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

Tiling is a bit better. We now make small jumps for each of several aggregations.

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

Radial basis features:

$x(s) = \begin{bmatrix} 0.1 \\ 0.9 \\ 0.05 \end{bmatrix}$

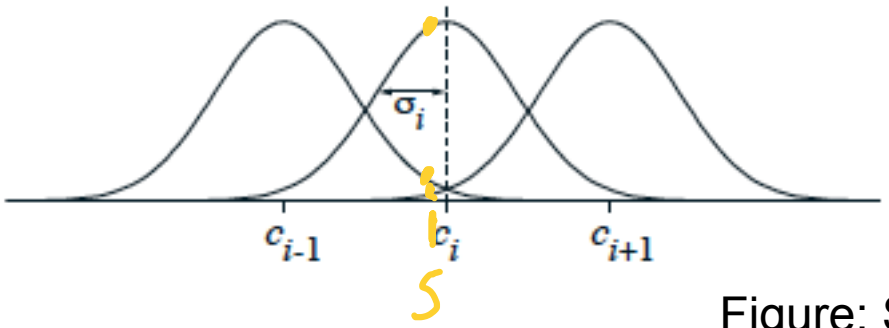


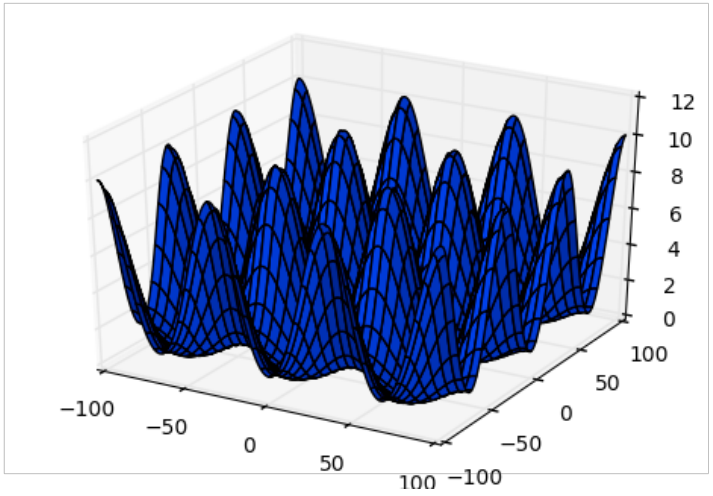
Figure: Sutton & Barto. RL:AI

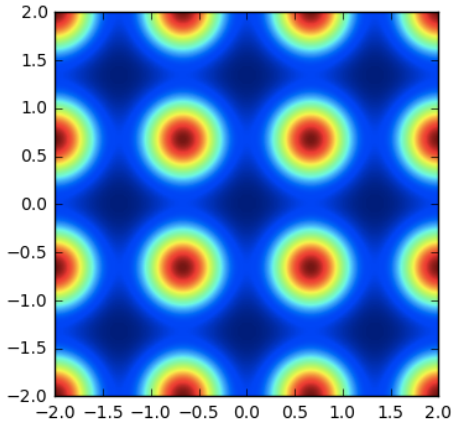
Radial basis functions

Radial basis functions only depend on distance to center, e.g.:

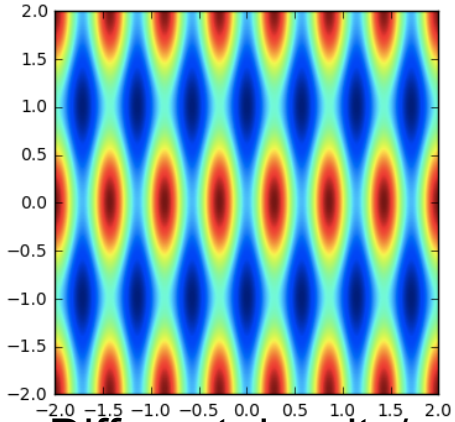
$$x_i(s) \doteq \exp\left(-\frac{\|s - c_i\|^2}{2\sigma_i^2}\right)$$

Also work in multi-d





Top-down view

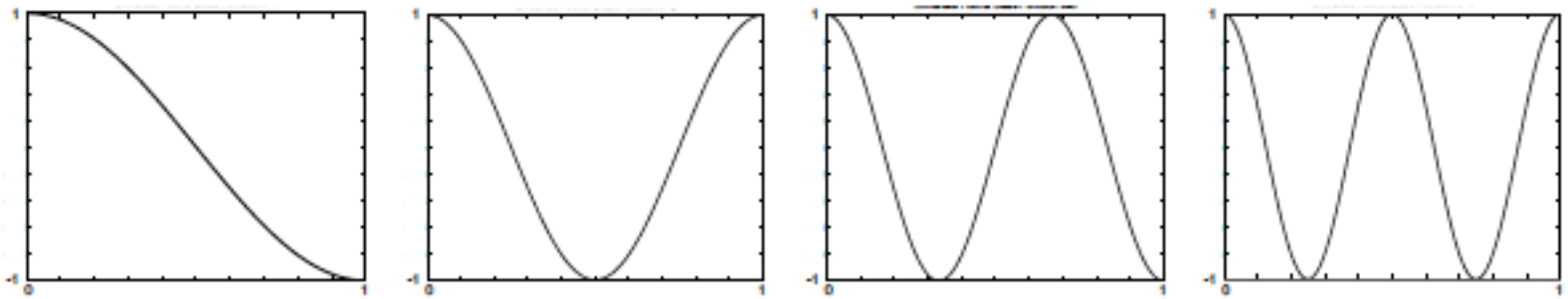


Different density/ distance function

Fourier basis

First: scale raw states between 0 and 1

Then (one-d): $x_j(s) = \cos(j\pi s), \quad s \in [0, 1]$
(so x_0 again a straight line!)



Again, pick as many features as needed…

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:

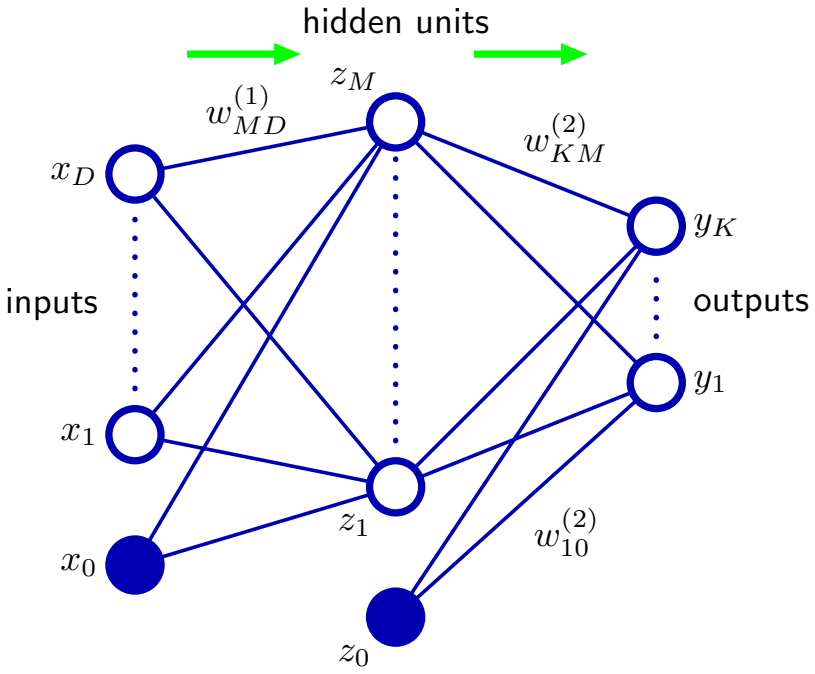
- What does **x** look like for the tilings example?
- Is tabular RL a specific case of a linear function? What would **x** be?

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:



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

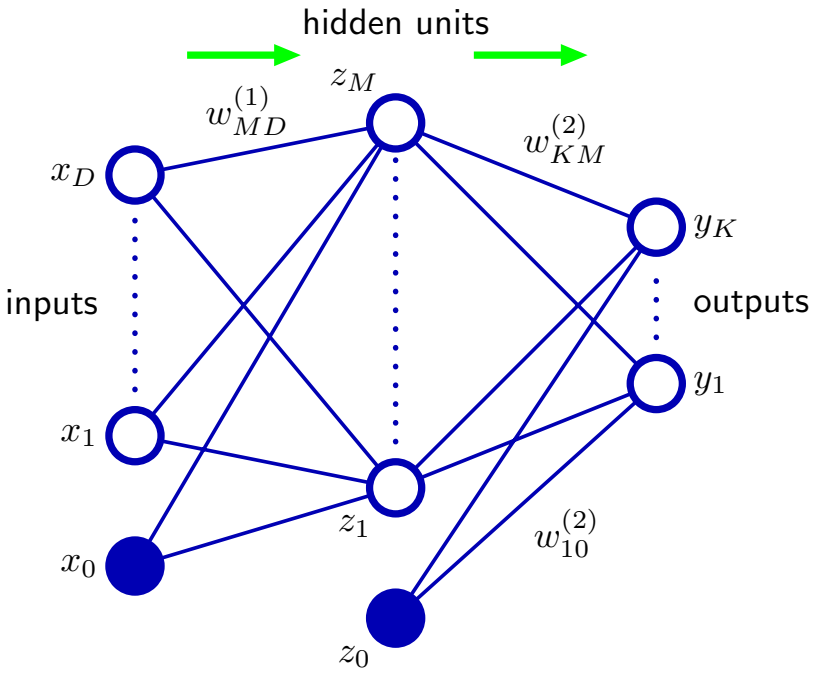
Non-linear activation function

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

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

and semi-gradient TD(0)

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

In both cases, 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 $\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[U_t - \hat{v}\left(S_t, \mathbf{w}_t\right)\right] \mathbf{x}\left(S_t\right)$
- Can prove: all local optima are global optima
- If features linearly independent: only one optimum

- So: gradient MC always converges to global minimum of VE with linear approximation
- With non-linear functions, there might be local minima
- What about semi-gradient TD(0)?

Fixed point of semi-gradient TD(0)

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

$$= w_t + \alpha \left(R_{t+1} x_t - x_t (x_t - \gamma x_{t+1})^\top w_t \right)$$

$$E[w_{t+1} | w_t] = w_t + \alpha \left(E[R_{t+1} x_t] - E[x_t (x_t - \gamma x_{t+1})^\top] w_t \right)$$

b A

$$w_{t+1} = w_t$$

$$b - A w_{TD} = 0$$

$$b = A w_{TD}$$

$$w_{TD} = A^{-1} b$$

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 $\mathbf{w}_{t+1} = \mathbf{w}_t$, so

$$\mathbf{b} - \mathbf{A}\mathbf{w}_{\mathrm{TD}} = \mathbf{0}$$

$$\mathbf{b} = \mathbf{A}\mathbf{w}_{\mathrm{TD}}$$

$$\mathbf{w}_{\mathrm{TD}} \doteq \mathbf{A}^{-1}\mathbf{b}, \qquad \text{the TD fixed point}$$

Fixed point of semi-gradient TD(0)

Semi-gradient TD with linear features can be proven to **converge** to this fixed point w_{TD}
(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

What can be proven (under some conditions) is that

$$\overline{\mathrm{VE}}\left(\mathbf{w}_{\mathrm{TD}}\right) \leq \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{\mathrm{VE}}(\mathbf{w})$$

= solution Gradient MC.

"maximally 1/(1-γ) worse than the best possible error"
- not very good for γ close to 1!

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

What can be proven (under some conditions) is that

$$\overline{\mathrm{VE}}\left(\mathbf{w}_{\mathrm{TD}}\right) \leq \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{\mathrm{VE}}(\mathbf{w})$$

“maximally 1/(1-γ) worse than the best possible error”
- not very good for γ close to 1!

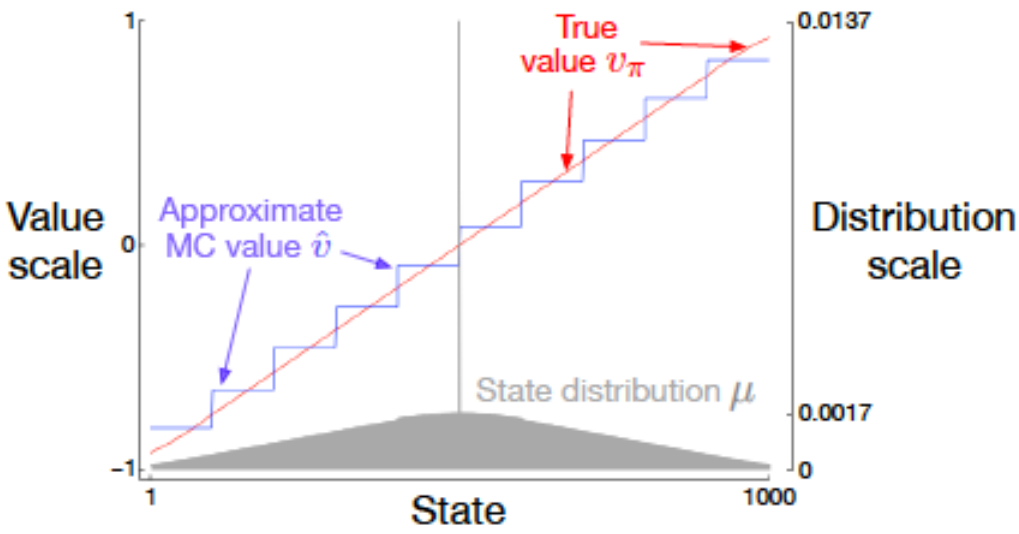
Still, semi-gradient TD converges much faster. Within a fixed amount of samples, TD can still beat MC…

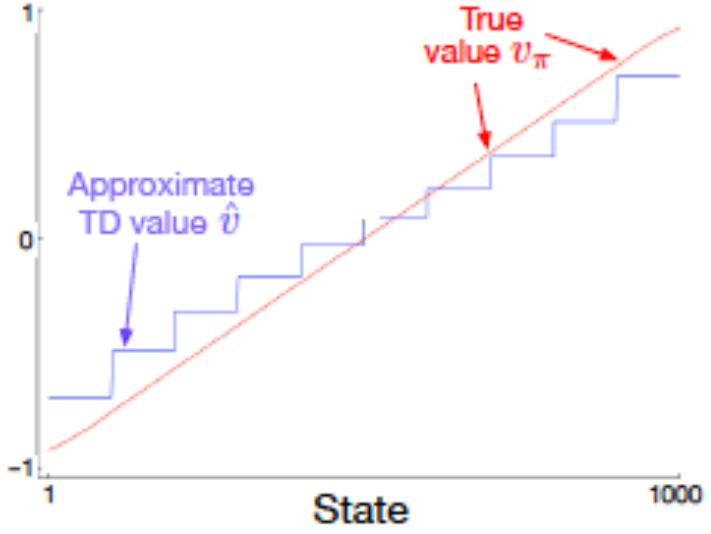
Gradient MC vs semi-gradient TD(0)

Gradient MC

Reminder!

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.

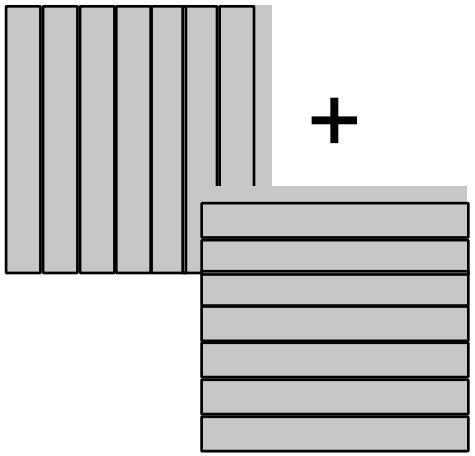
Selecting step size

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:)

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

Selecting step size

Take the typical norm of gradient into account?

Rule of thumb:

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

0.1

0.05

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

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

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

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

Least-squares temporal-difference (LSTD)

Use sample averages/sums 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 1/(t-1) term):

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

then

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

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!

Back to the big picture

We have talked today about *prediction* methods

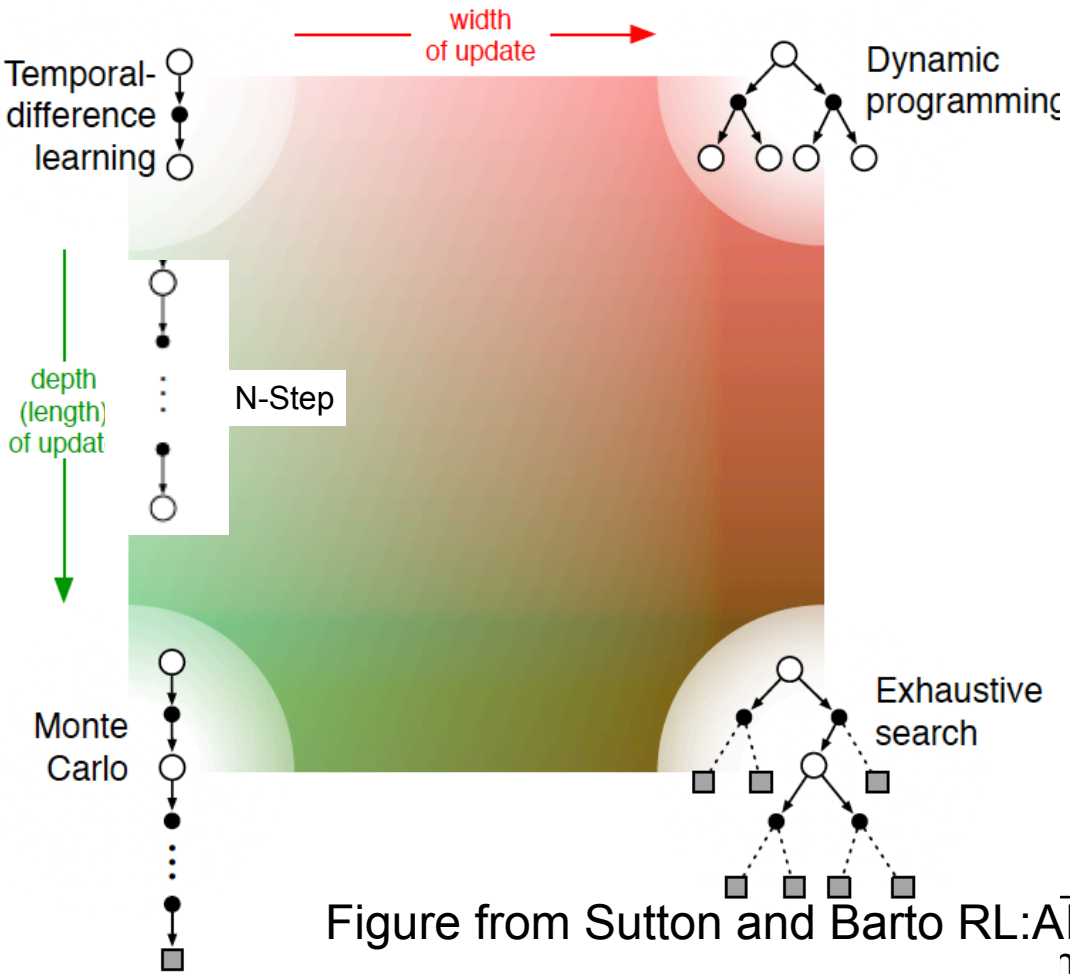


Figure from Sutton and Barto RL:AI

Back to the big picture

We have talked today about *prediction* methods

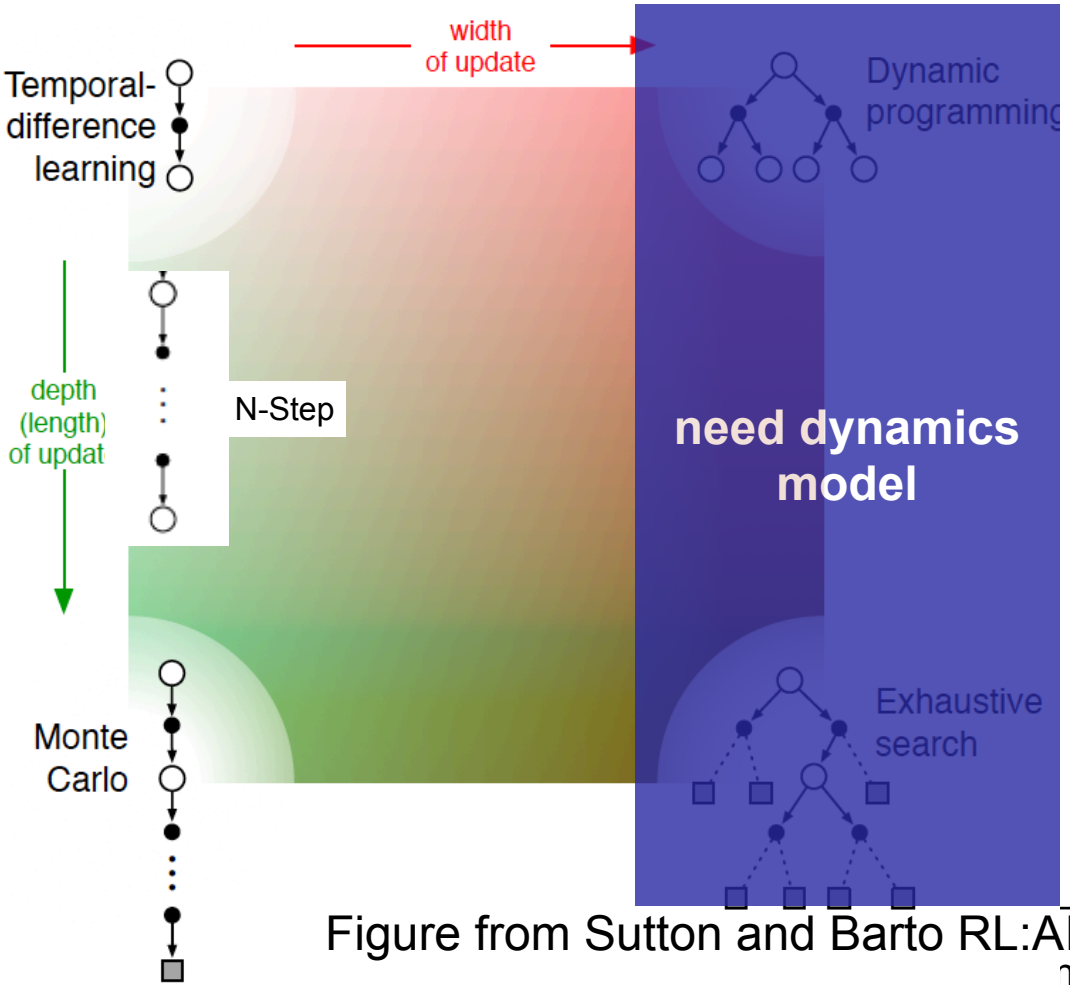


Figure from Sutton and Barto RL:AI

Back to the big picture

We have talked today about *prediction* methods

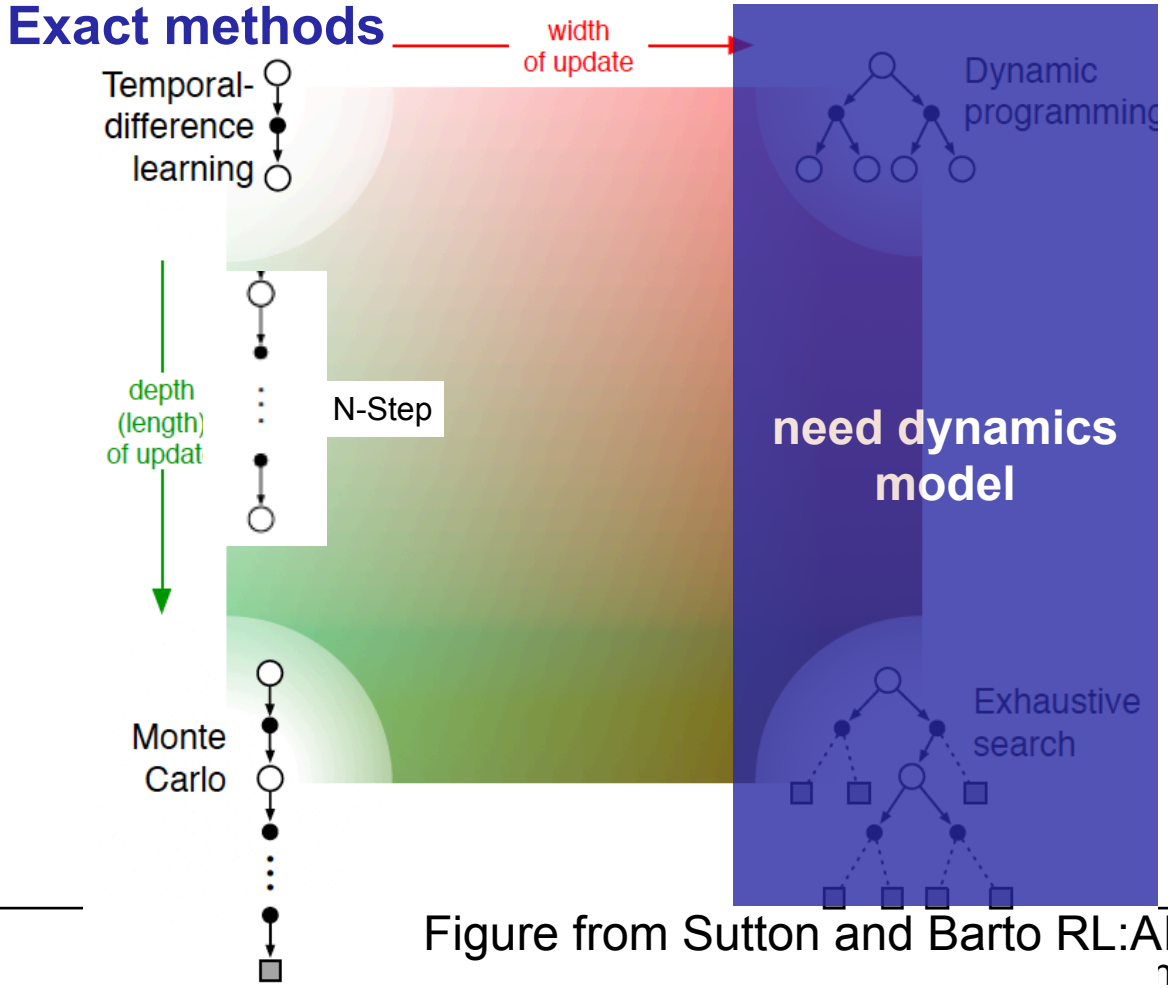


Figure from Sutton and Barto RL:AI

Back to the big picture

We have talked today about *prediction* methods

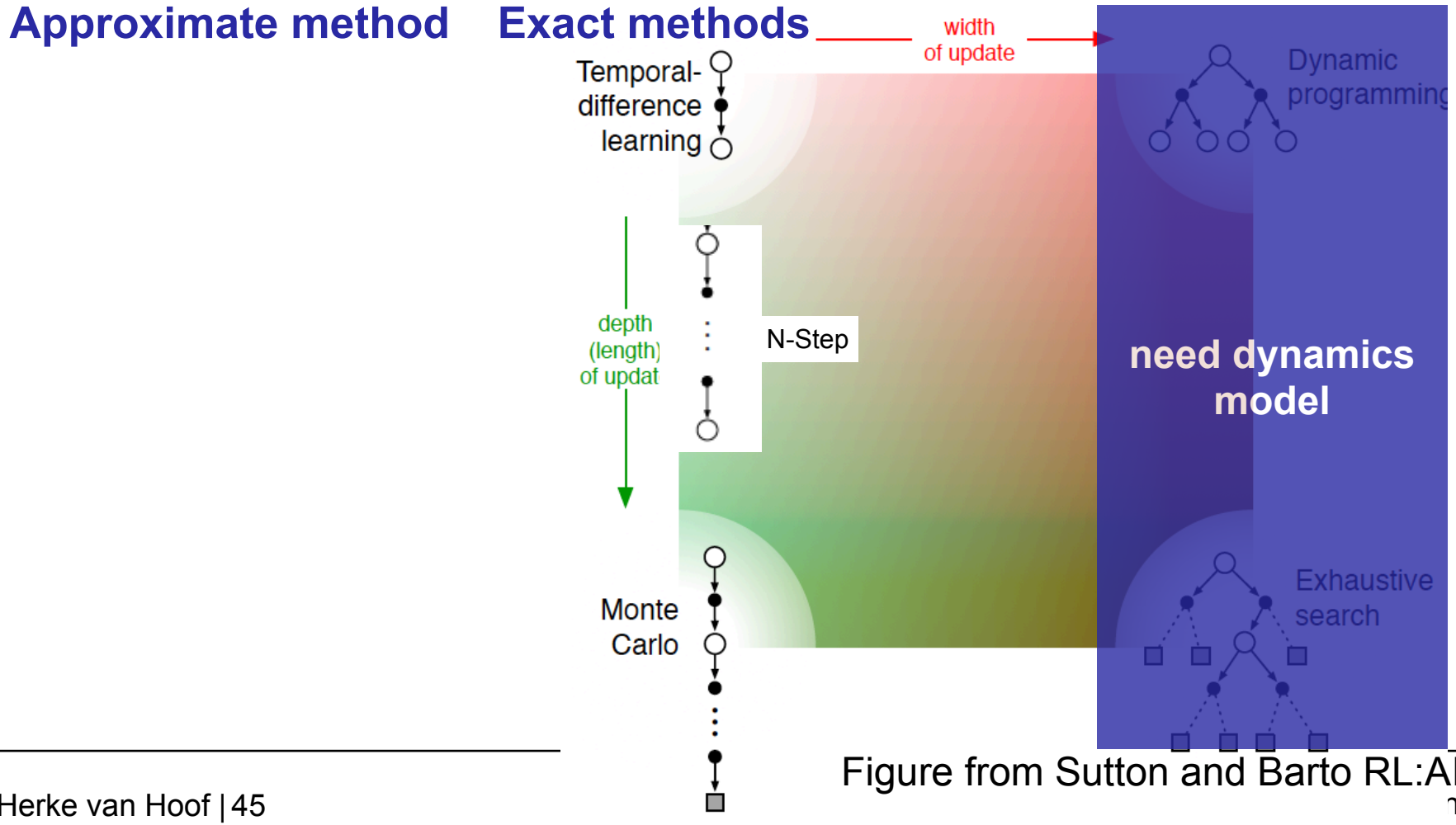


Figure from Sutton and Barto RL:AI

Back to the big picture

We have talked today about *prediction* methods

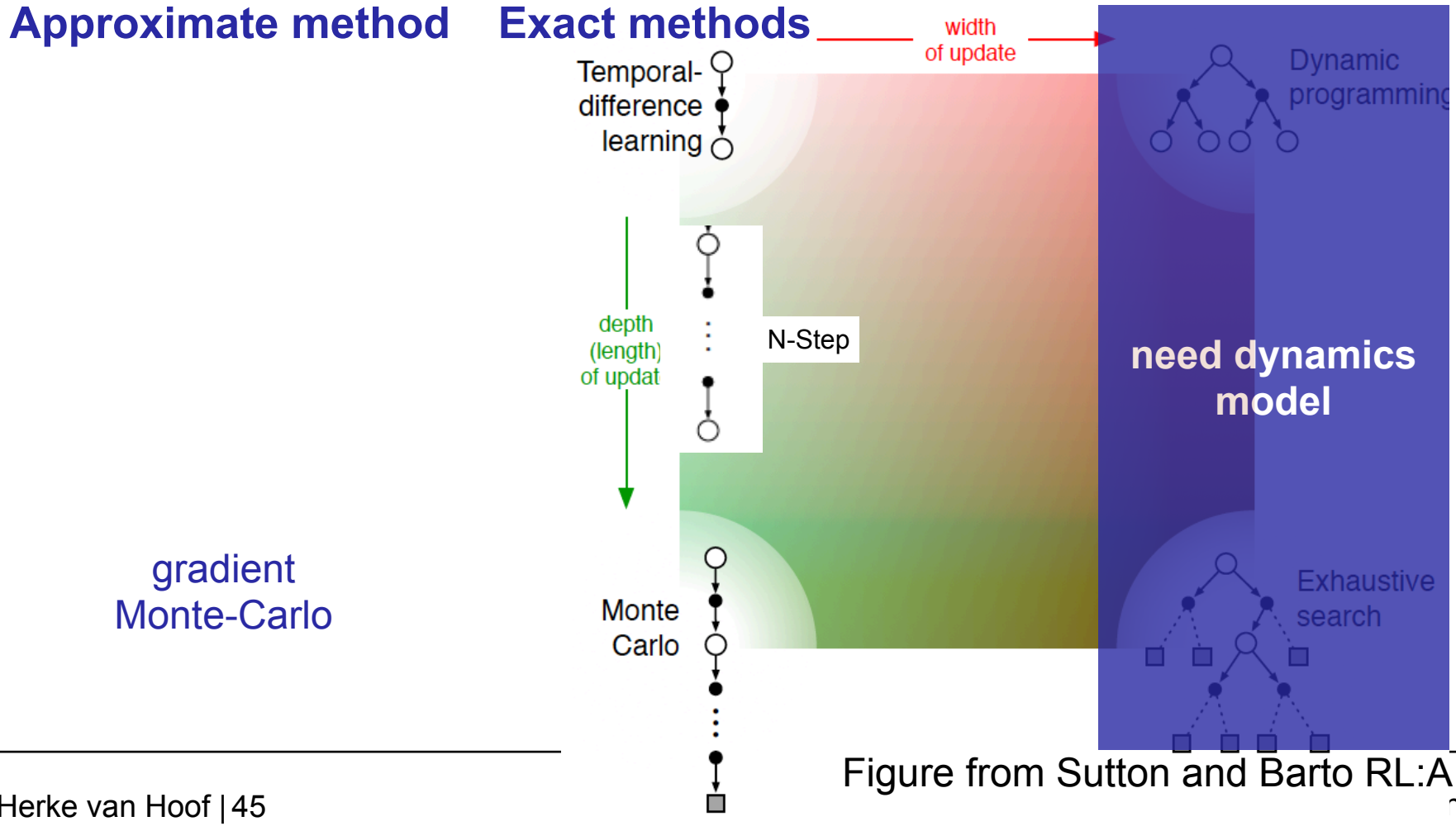


Figure from Sutton and Barto RL:AI

Back to the big picture

We have talked today about *prediction* methods

Approximate method

semi-gradient
TD(0)
& LSTD

gradient
Monte-Carlo

Exact methods

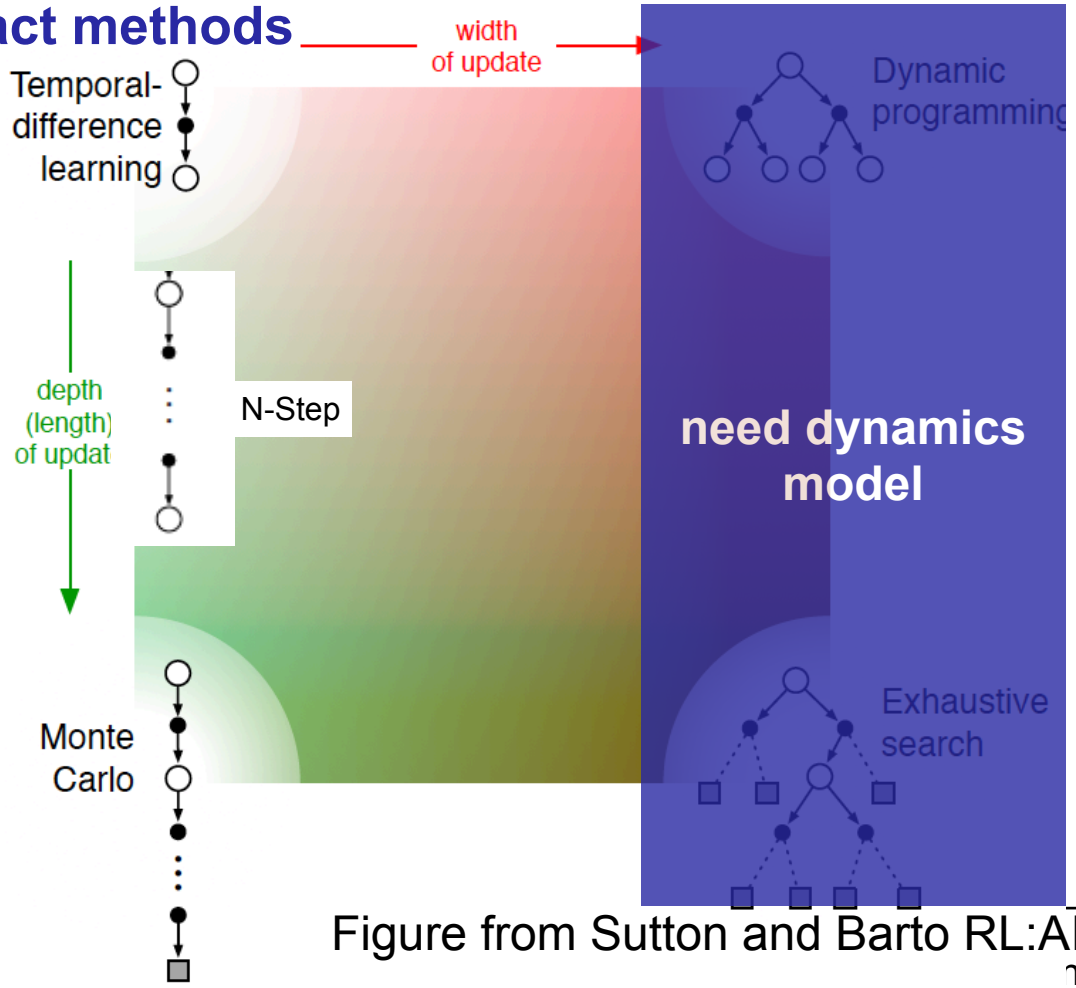


Figure from Sutton and Barto RL:AI

Back to the big picture

We have talked today about *prediction* methods

Approximate method

semi-gradient
TD(0)
& LSTD

↓

Straightforward
generalisation
to n-step
(not discussed)

gradient
Monte-Carlo

Exact methods

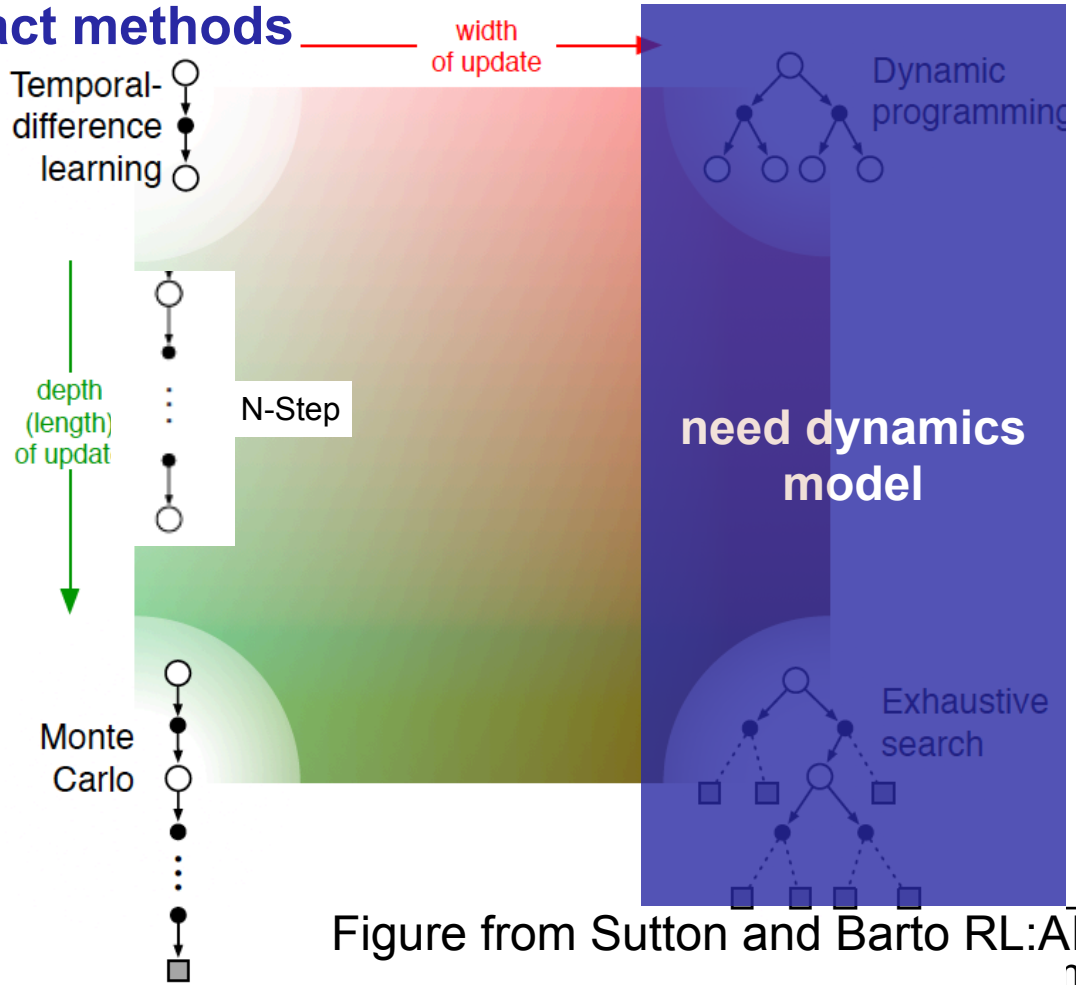


Figure from Sutton and Barto RL:AI

Back to the big picture

target computed from MC is = Gt is unbiased
target computed from TD (semi gradient TD or LSTD) is biased.

Guarantees depend on 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

tabular
aggregate
tiling
radial basis function
polynomial basis function
fourier basis function

e.g. neural network

Back to the big picture

Main problems:

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

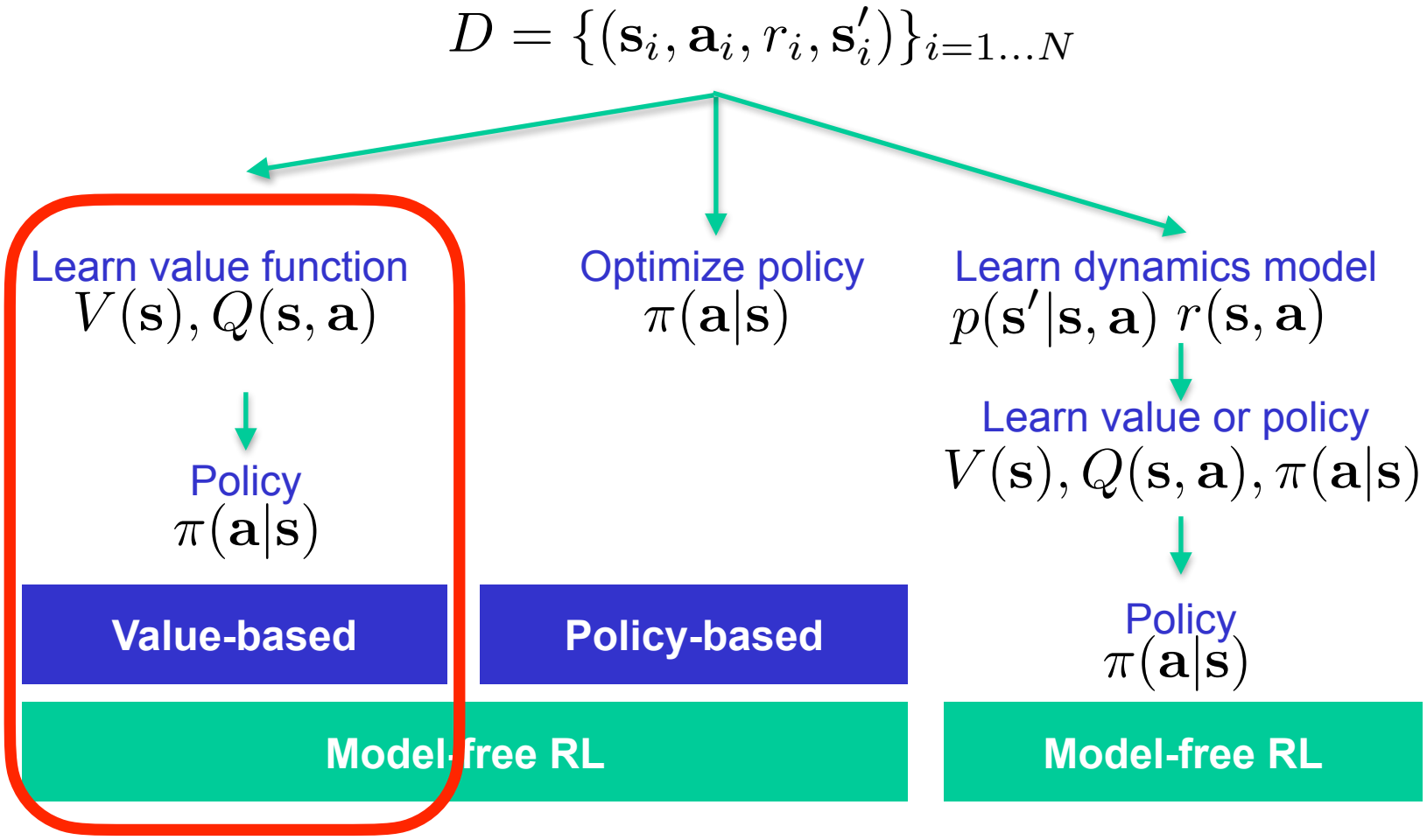
Back to the big picture

Main problems:

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

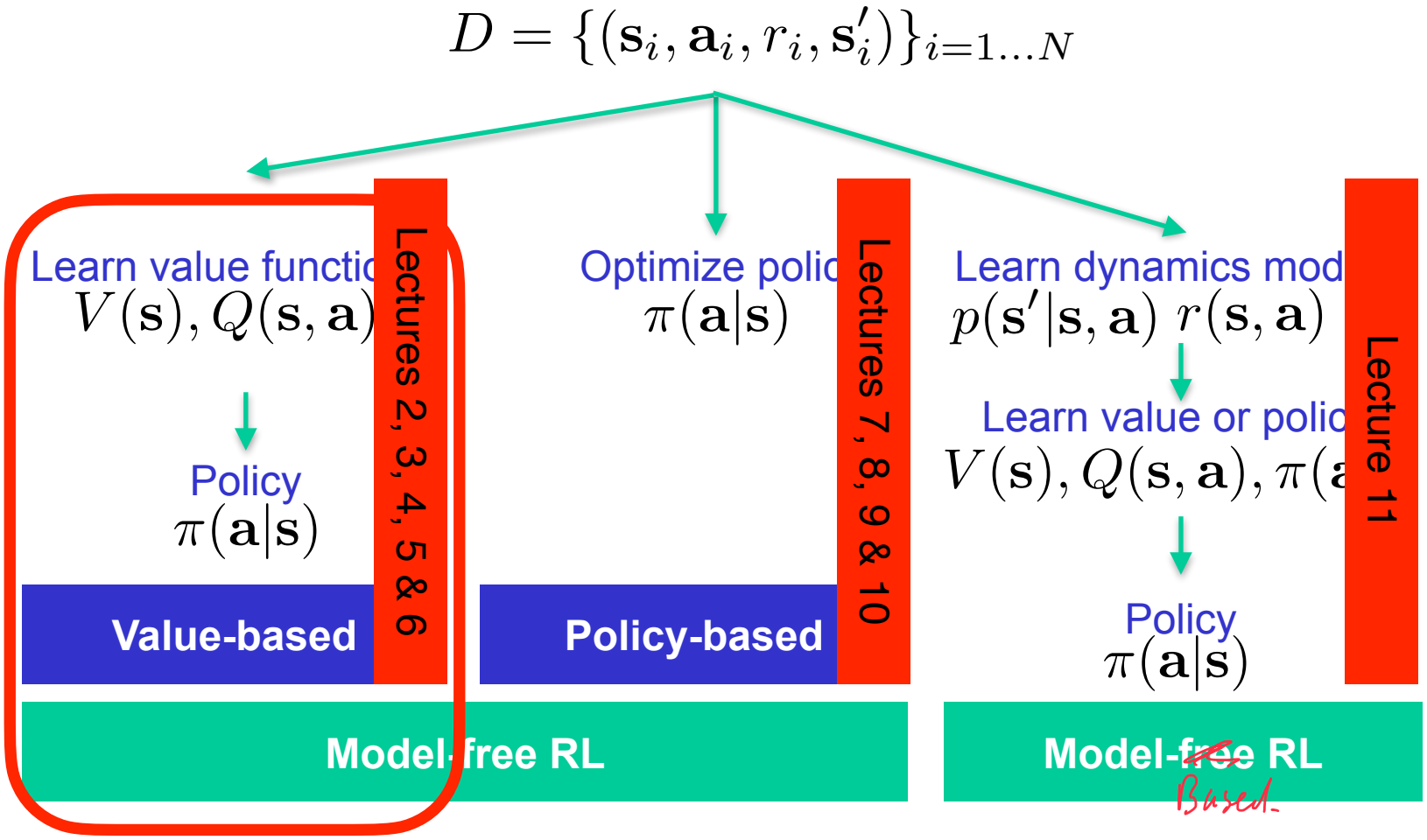
Tabular (Exact)	Approximate
lecture 2, 3& 4	Today's lecture (5)
lecture 2, 3& 4	Next lecture (6)
lecture 2, 3& 4	Next lecture (6)

Back to the big picture



Thanks to Jan Peters

Back to the big picture



Thanks to Jan Peters

Feedback?

h.c.vanhoof@uva.nl

Extra: Aggregation and tiling

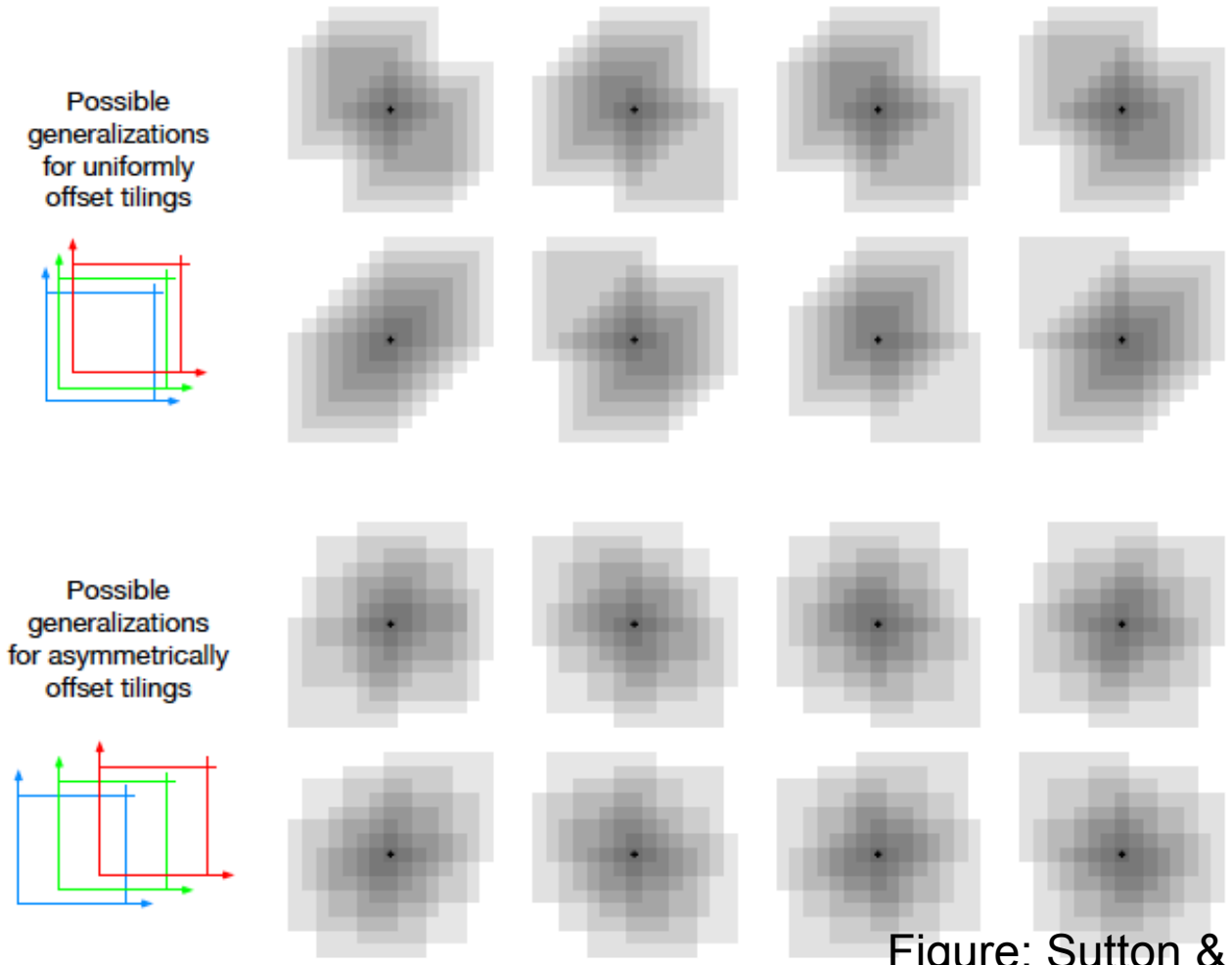


Figure: Sutton & Barto. RL:AI

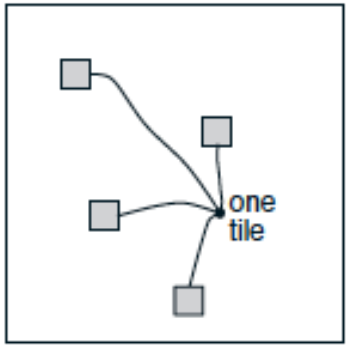
Extra: Aggregation and tiling

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

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

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

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

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

$$\vdots$$

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

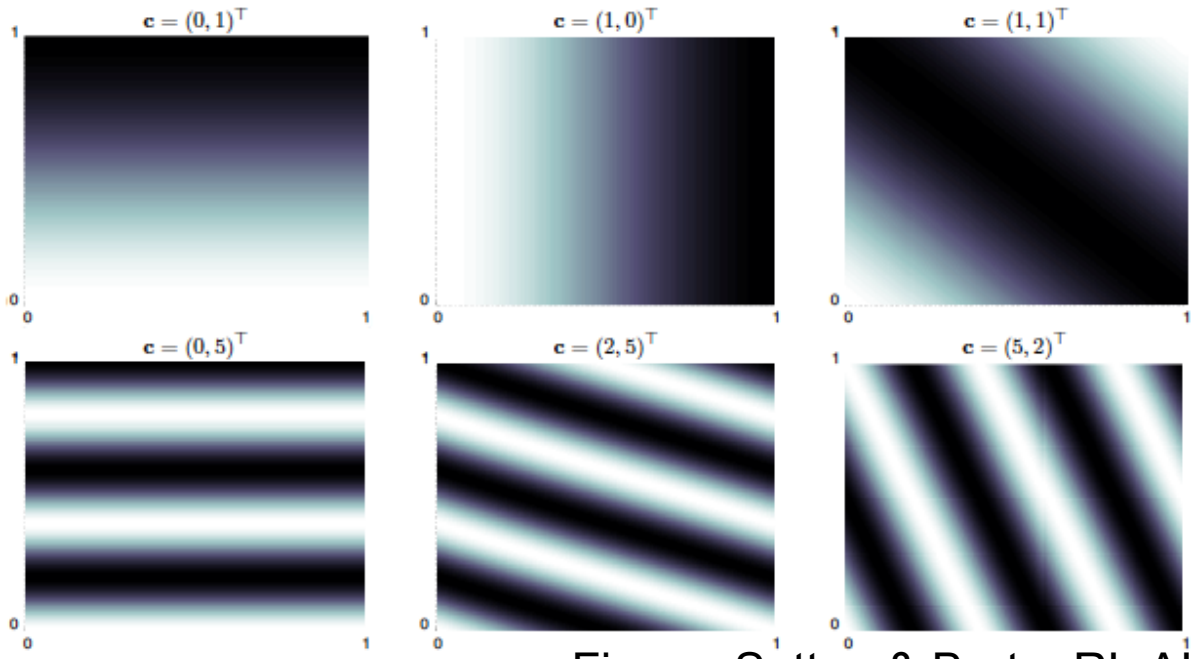


Figure: Sutton & Barto. RL:AI

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

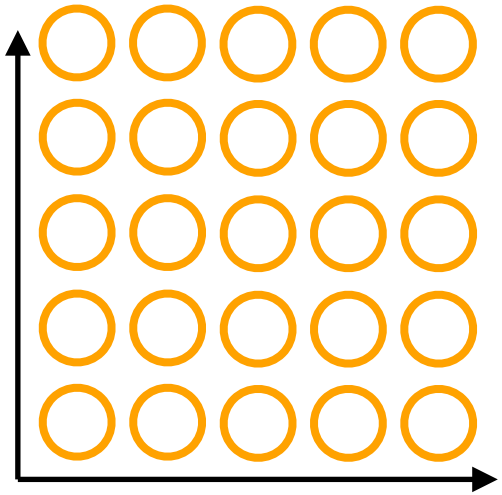
However, finding the nearest neighbours at every step is usually more expensive than evaluating a parametric function

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

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

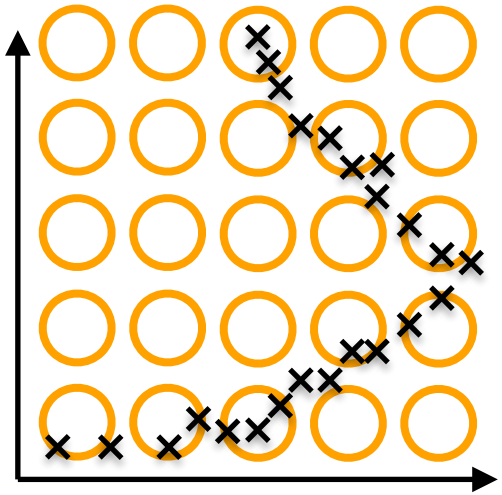
data points

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

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

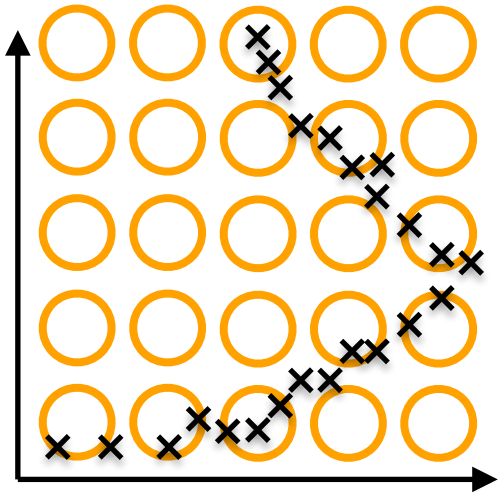
data points

Extra: Non-parametric approximation

Radial basis functions

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

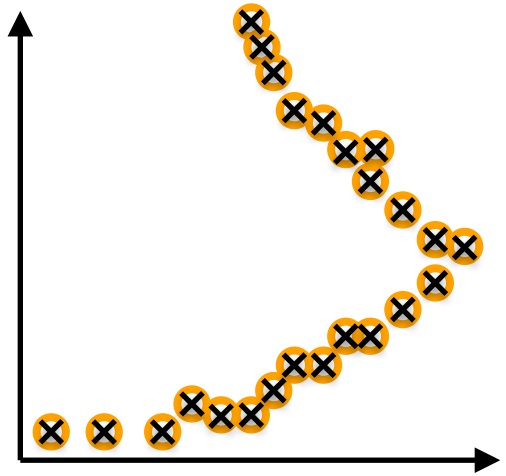
predefined features



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

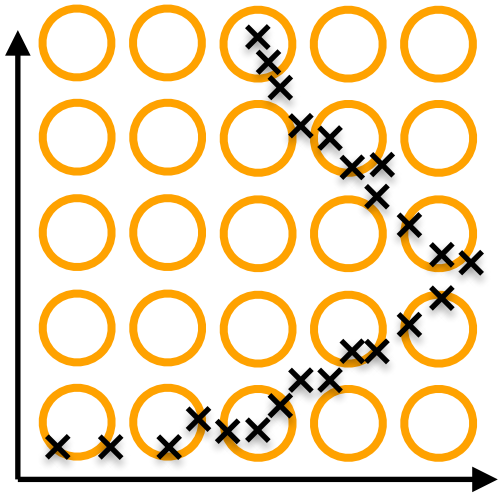


Extra: Non-parametric approximation

Radial basis functions

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

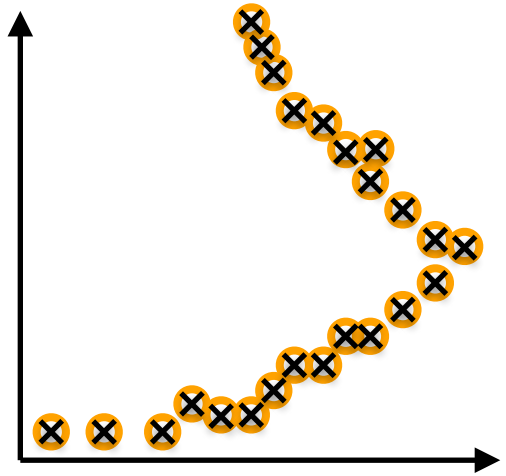
predefined features



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)