



UNIVERSITY OF TECHNOLOGY  
IN THE EUROPEAN CAPITAL OF CULTURE  
CHEMNITZ

# Deep Reinforcement Learning

Function approximation

Julien Vitay

Professur für Künstliche Intelligenz - Fakultät für Informatik

<https://tu-chemnitz.de/informatik/KI/edu/deepri>

# 1 - Limits of tabular RL

# Tabular reinforcement learning

- All the methods seen so far belong to **tabular RL**.
- Q-learning necessitates to store in a **Q-table** one Q-value per state-action pair  $(s, a)$ .

Game Board:



Current state (s):  
0 0 0  
0 1 0

Q Table:

$\gamma = 0.95$

	0 0 0 1 0 0	0 0 0 0 1 0	0 0 0 0 0 1	1 0 0 0 0 0	0 1 0 0 0 0	0 0 1 0 0 0
↑	0.2	0.3	1.0	-0.22	-0.3	0.0
↓	-0.5	-0.4	-0.2	-0.04	-0.02	0.0
→	0.21	0.4	-0.3	0.5	1.0	0.0
←	-0.6	-0.1	-0.1	-0.31	-0.01	0.0

Source: <https://towardsdatascience.com/qrash-course-deep-q-networks-from-the-ground-up-1bbda41d3677>

# Tabular reinforcement learning

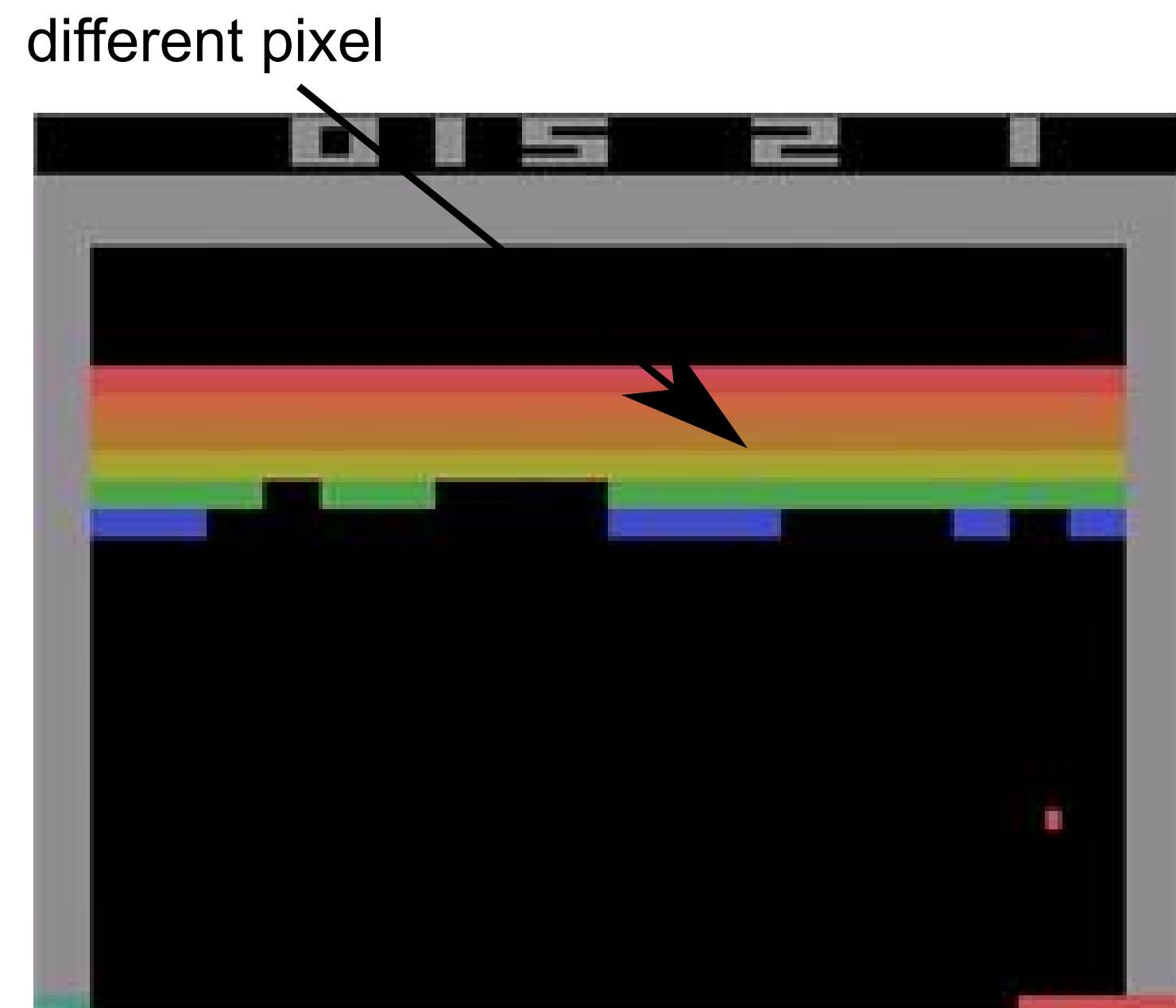
- If a state has never been visited during learning, the Q-values will still be at their initial value (0.0), no policy can be derived.

Visited state



Optimal action: left

Not visited state

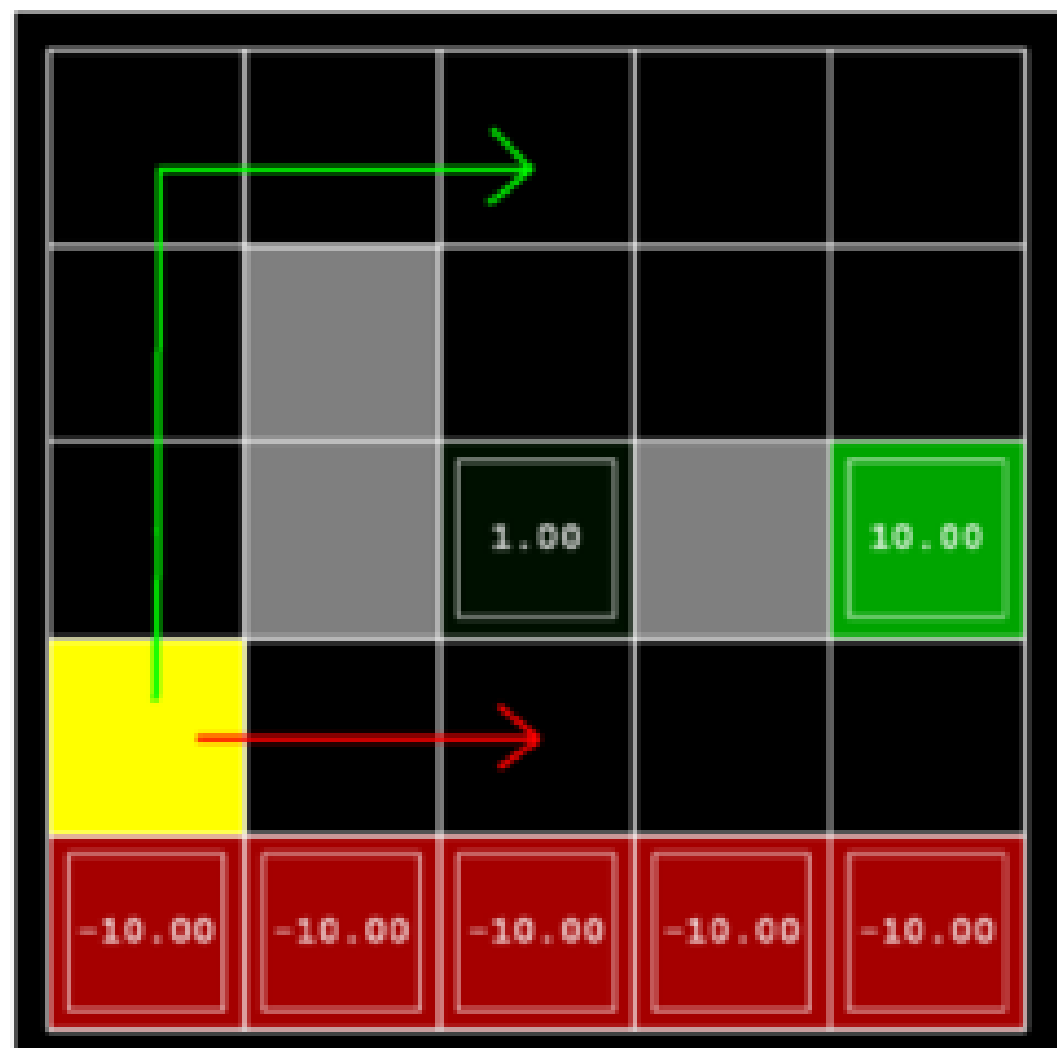


Optimal action: ?

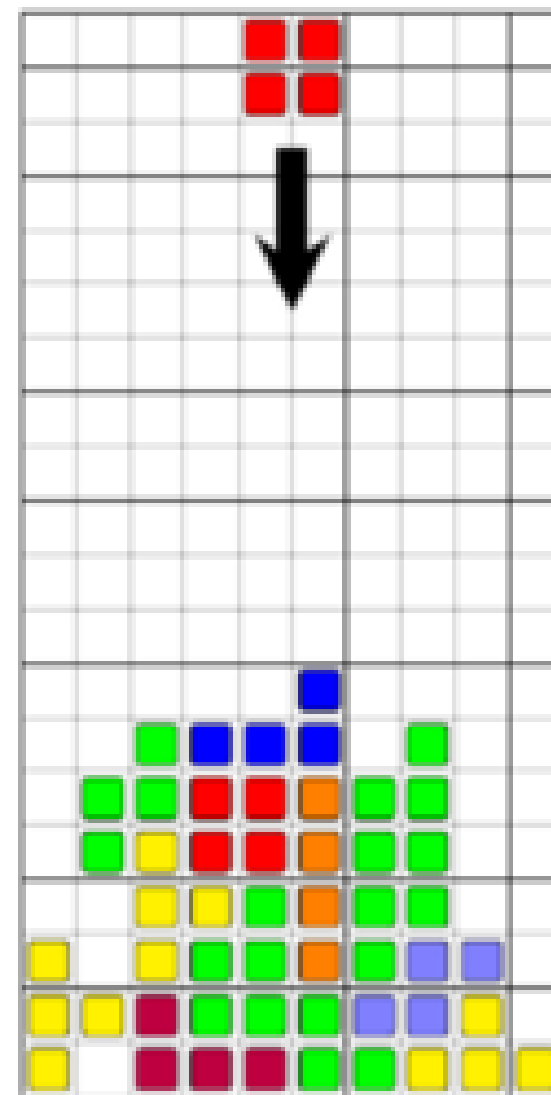
- Similar states likely have the same optimal action: we want to be able to **generalize** the policy between states.

# Tabular reinforcement learning

- For most realistic problems, the size of the Q-table becomes quickly untractable.



Gridworld  
 $10^1$



Tetris  
 $10^{60}$

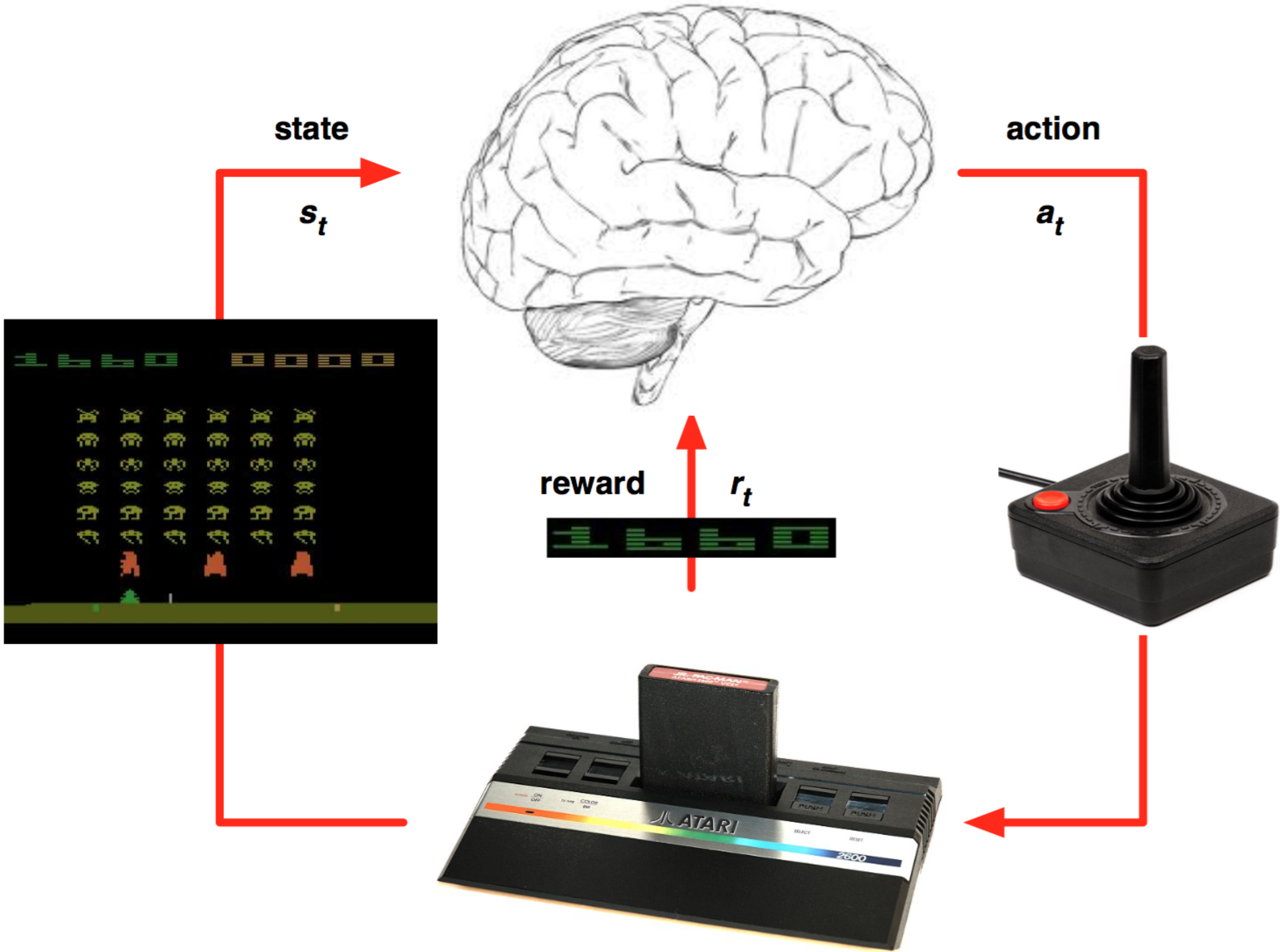


Atari  
 $10^{308}$  (ram)  $10^{16992}$  (pixels)

Source: <https://medium.com/@tw446/a-summary-of-deep-reinforcement-learning-rl-bootcamp-lecture-2-c3a15db5934e>

- If you use black-and-white 256x256 images as inputs, you have  $2^{256 \times 256} = 10^{19728}$  possible states!
- Tabular RL** is limited to toy problems.

# Tabular RL cannot learn to play video games





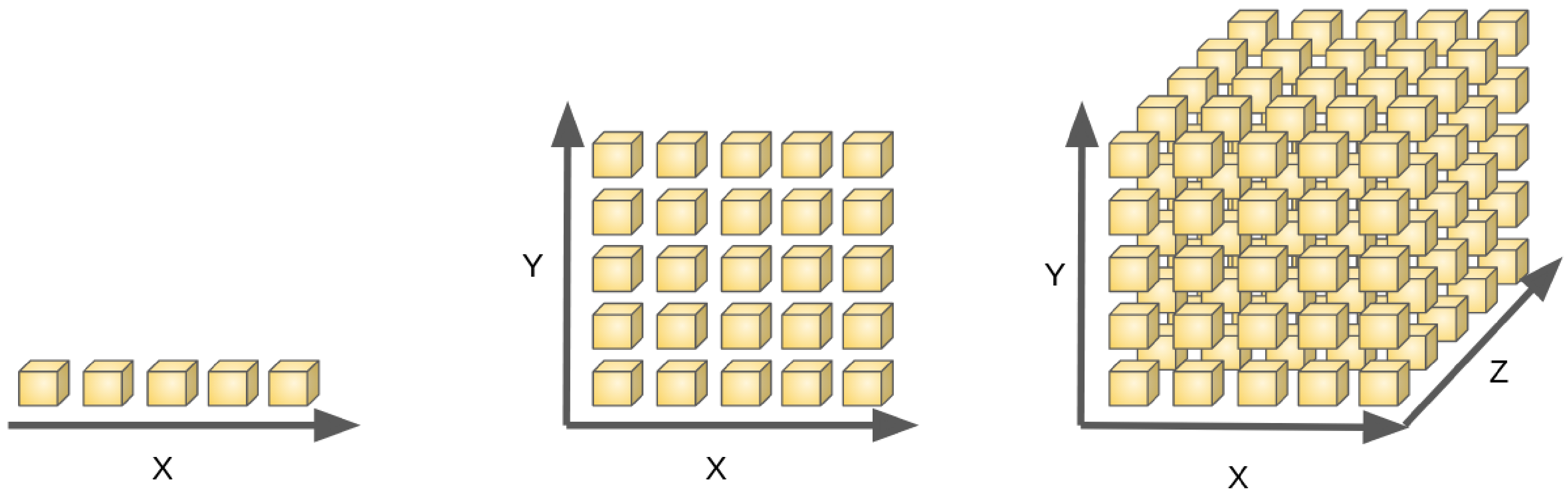
# Continuous action spaces

- Tabular RL only works for small **discrete action spaces**.
- Robots have **continuous action spaces**, where the actions are changes in **joint angles** or **torques**.
- A joint angle could take any value in  $[0, \pi]$ .



# Continuous action spaces

- A solution would be to **discretize** the action space (one action per degree), but we would fall into the **curse of dimensionality**.



- The more degrees of freedom, the more discrete actions, the more entries in the Q-table...
- Tabular RL cannot deal with continuous action spaces, unless we approximate the policy with an **actor-critic** architecture.



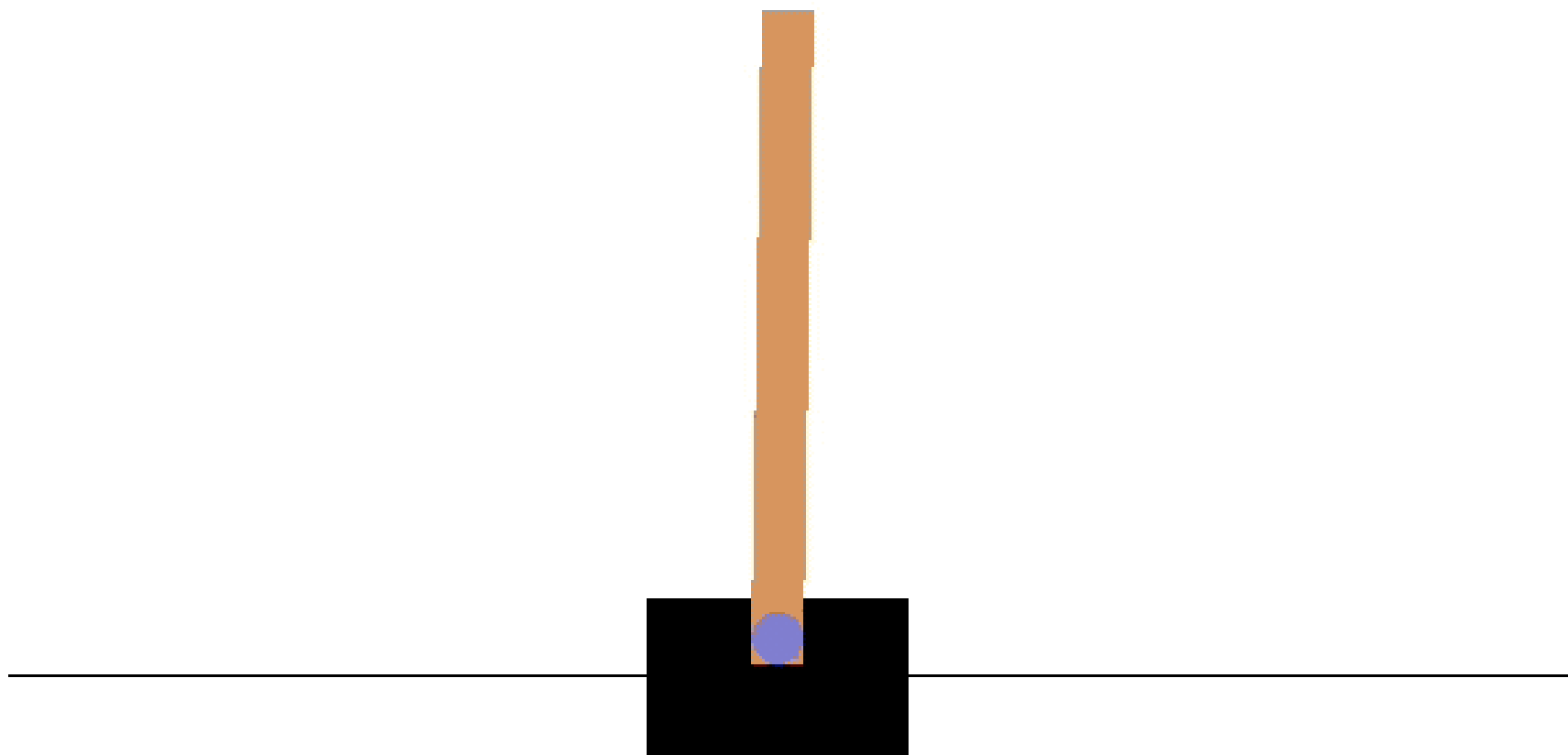
## 2 - Function approximation

# Feature vectors

- Let's represent a state  $s$  by a vector of  $d$  **features**  
 $\phi(s) = [\phi_1(s), \phi_2(s), \dots, \phi_d(s)]^T$ .
- For the cartpole, the feature vector would be:

$$\phi(s) = \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}$$

- $x$  is the position,  $\theta$  the angle,  $\dot{x}$  and  $\dot{\theta}$  their derivatives.
- We are able to represent **any state**  $s$  using these four variables.



## Feature vectors

- For more complex problems, the feature vector should include all the necessary information (Markov property).



$$\phi(s) = \begin{bmatrix} x \text{ position of the paddle} \\ x \text{ position of the ball} \\ y \text{ position of the ball} \\ x \text{ speed of the ball} \\ y \text{ speed of the position} \\ \text{presence of brick 1} \\ \text{presence of brick 2} \\ \vdots \end{bmatrix}$$

- In deep RL, we will **learn** these feature vectors, but let's suppose for now that we have them.

## Feature vectors

- Note that we can always fall back to the tabular case using **one-hot encoding** of the states:

$$\phi(s_1) = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \dots \\ 0 \end{bmatrix} \quad \phi(s_2) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ \dots \\ 0 \end{bmatrix} \quad \phi(s_3) = \begin{bmatrix} 0 \\ 0 \\ 1 \\ \dots \\ 0 \end{bmatrix} \quad \dots$$

- But the idea is that we can represent states with much less values than the number of states:

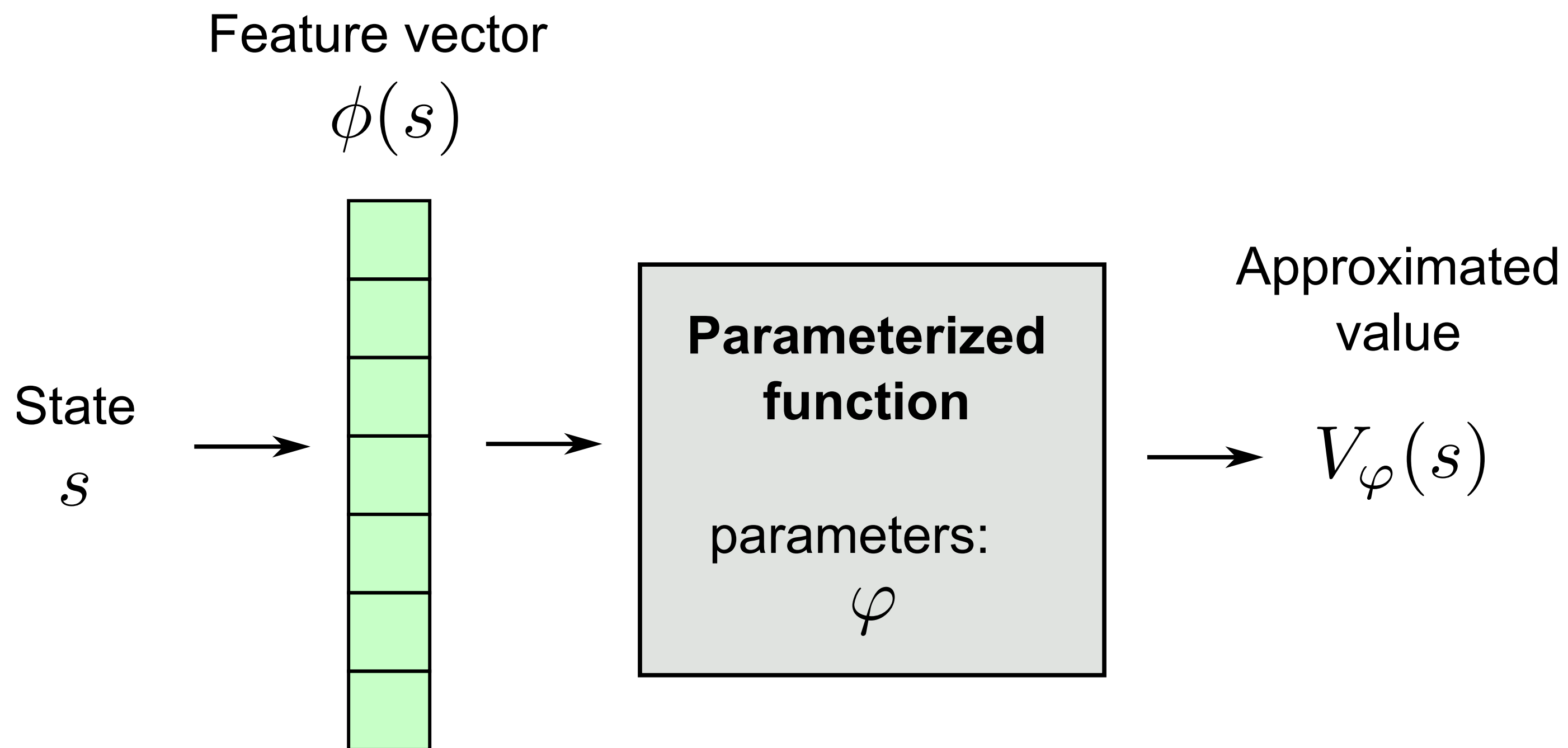
$$d \ll |\mathcal{S}|$$

- We can also represent **continuous state spaces** with feature vectors.

# State value approximation

- In **state value approximation**, we want to approximate the state value function  $V^\pi(s)$  with a **parameterized function**  $V_\varphi(s)$ :

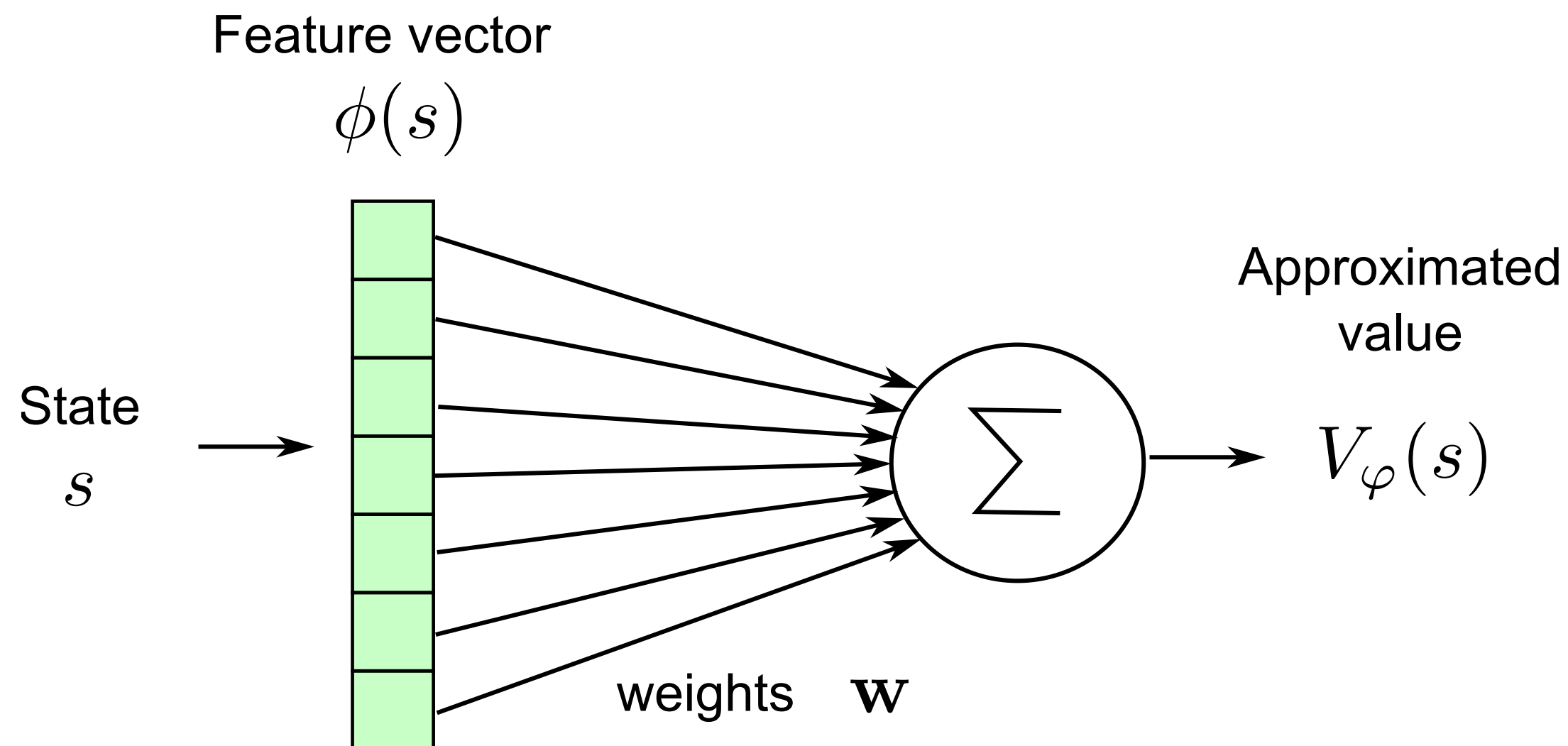
$$V_\varphi(s) \approx V^\pi(s)$$



- The parameterized function can have any form. It has a set of parameters  $\varphi$  used to transform the feature vector  $\phi(s)$  into an approximated value  $V_\varphi(s)$ .

# Linear approximation of state value functions

- The simplest function approximator (FA) is the **linear approximator**.

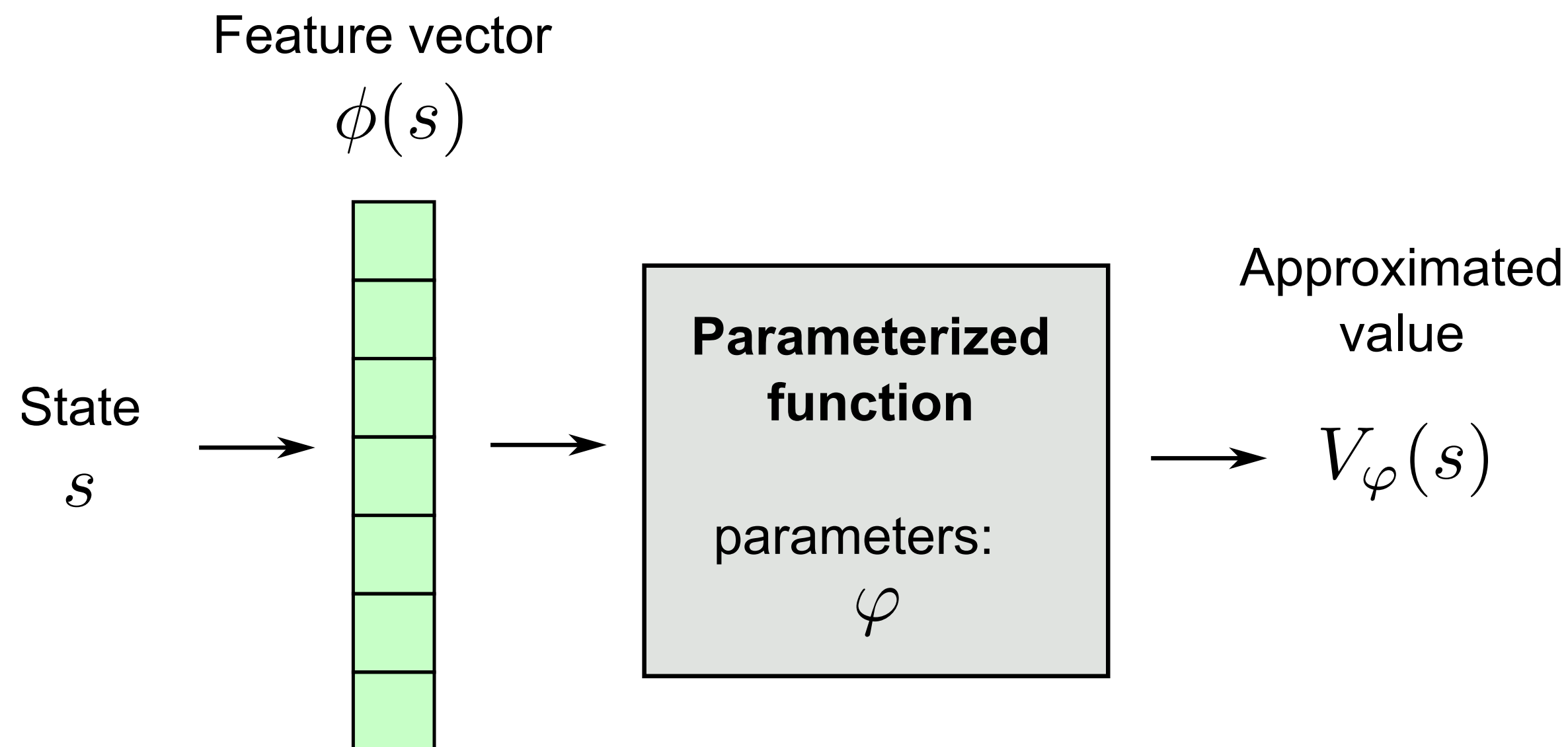


- The approximated value is a linear combination of the features:

$$V_\varphi(s) = \sum_{i=1}^d w_i \phi_i(s) = \mathbf{w}^T \times \phi(s)$$

- The **weight vector**  $\mathbf{w} = [w_1, w_2, \dots, w_d]^T$  is the set of parameters  $\varphi$  of the function.
- A linear approximator is a single **artificial neuron** (linear regression) without a bias.

# Learning the state value approximation



- Regardless the form of the function approximator, we want to find the parameters  $\varphi$  making the approximated values  $V_\varphi(s)$  as close as possible from the true values  $V^\pi(s)$  for all states  $s$ .
  - This is a **regression** problem.
- We want to minimize the **mean-square error** between the two quantities:

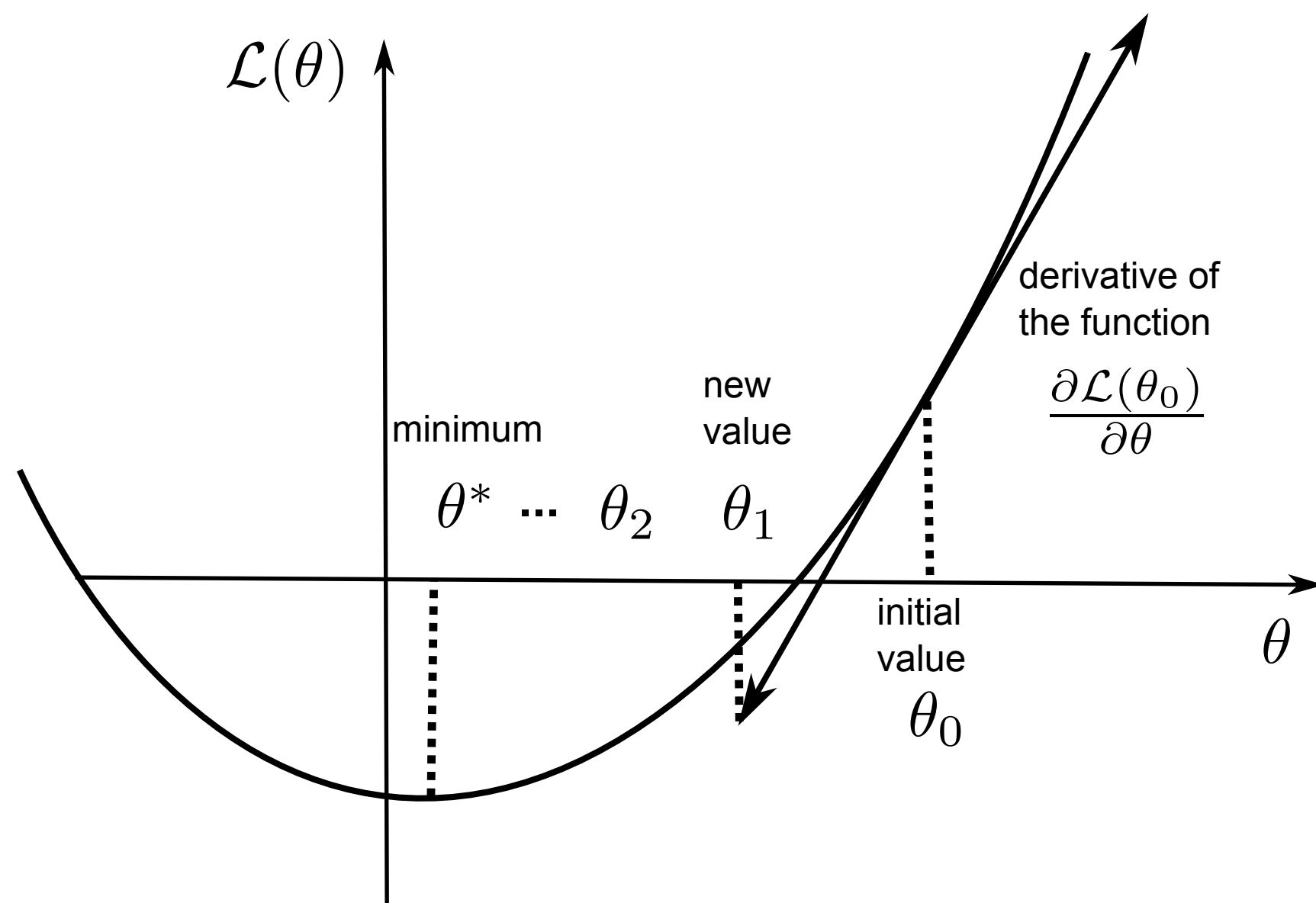
$$\min_{\varphi} \mathcal{L}(\varphi) = \mathbb{E}_{s \in \mathcal{S}} [(V^\pi(s) - V_\varphi(s))^2]$$

- The **loss function**  $\mathcal{L}(\varphi)$  is minimal when the predicted values are close to the true ones on average.



# Learning the state value approximation

- Let's suppose that we know the true state values  $V^\pi(s)$  for all states and that the parametrized function is **differentiable**.



- We can find the minimum of the loss function by applying **gradient descent** (GD) iteratively:

$$\Delta\varphi = -\eta \nabla_{\varphi} \mathcal{L}(\varphi)$$

- $\nabla_{\varphi} \mathcal{L}(\varphi)$  is the gradient of the loss function w.r.t to the parameters  $\varphi$ .

$$\nabla_{\varphi} \mathcal{L}(\varphi) = \begin{bmatrix} \frac{\partial \mathcal{L}(\varphi)}{\partial \varphi_1} \\ \frac{\partial \mathcal{L}(\varphi)}{\partial \varphi_2} \\ \vdots \\ \frac{\partial \mathcal{L}(\varphi)}{\partial \varphi_K} \end{bmatrix}$$

- When applied repeatedly, GD converges to a local minimum of the loss function.

# Learning the state value approximation

- To minimize the mean square error,

$$\min_{\varphi} \mathcal{L}(\varphi) = \mathbb{E}_{s \in \mathcal{S}} [(V^{\pi}(s) - V_{\varphi}(s))^2]$$

we will iteratively modify the parameters  $\varphi$  according to:

$$\begin{aligned} \Delta\varphi &= \varphi_{k+1} - \varphi_n = -\eta \nabla_{\varphi} \mathcal{L}(\varphi) \\ &= -\eta \nabla_{\varphi} \mathbb{E}_{s \in \mathcal{S}} [(V^{\pi}(s) - V_{\varphi}(s))^2] \\ &= \mathbb{E}_{s \in \mathcal{S}} [-\eta \nabla_{\varphi} (V^{\pi}(s) - V_{\varphi}(s))^2] \\ &= \mathbb{E}_{s \in \mathcal{S}} [\eta (V^{\pi}(s) - V_{\varphi}(s)) \nabla_{\varphi} V_{\varphi}(s)] \end{aligned}$$

- As it would be too slow to compute the expectation on the whole state space (**batch algorithm**), we will sample the quantity:

$$\delta_{\varphi} = \eta (V^{\pi}(s) - V_{\varphi}(s)) \nabla_{\varphi} V_{\varphi}(s)$$

and update the parameters with **stochastic gradient descent** (SGD).

# Learning the state value approximation

- Gradient of the mse:

$$\Delta\varphi = \mathbb{E}_{s \in \mathcal{S}} [\eta (V^\pi(s) - V_\varphi(s)) \nabla_\varphi V_\varphi(s)]$$

- If we sample  $K$  states  $s_i$  from the state space:

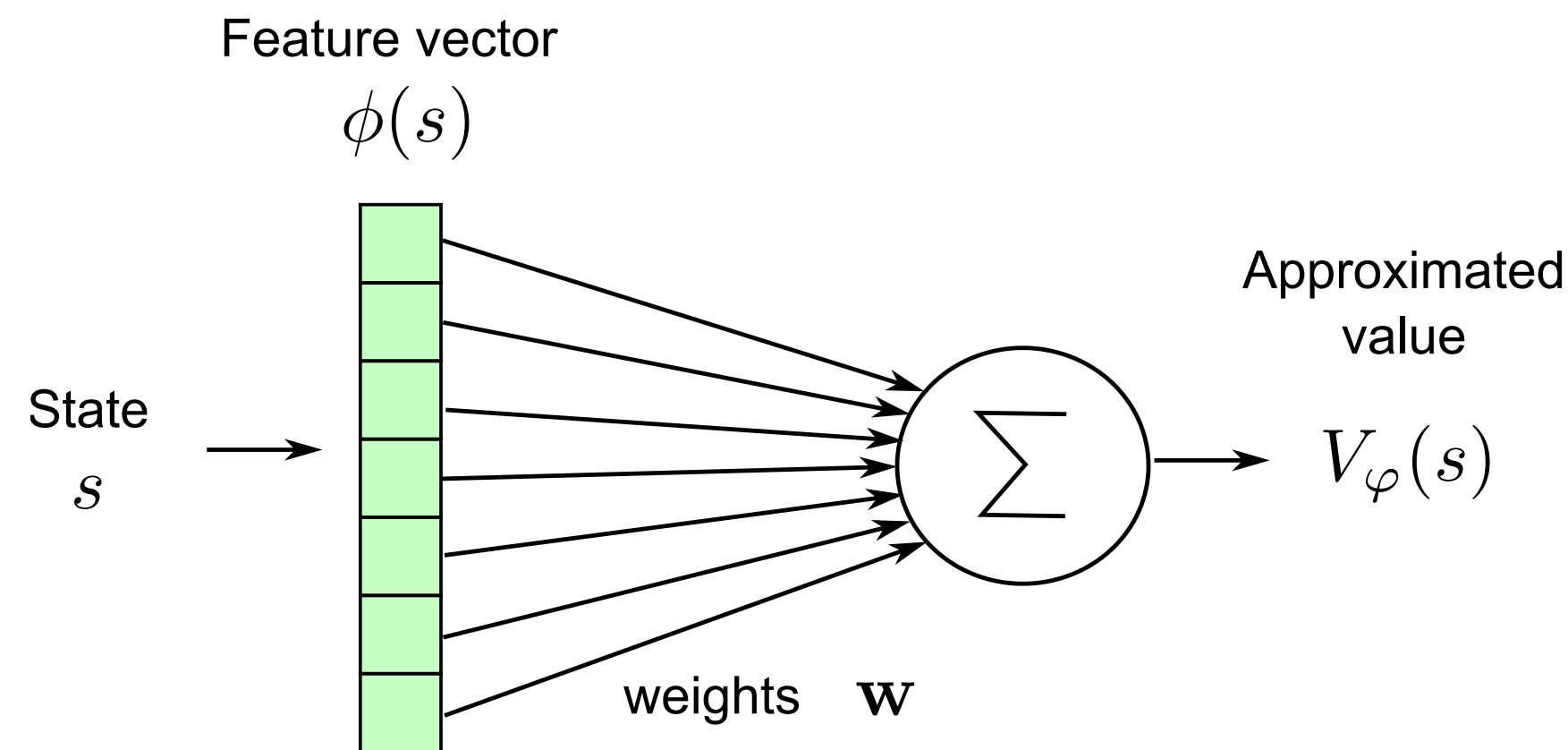
$$\Delta\varphi = \eta \frac{1}{K} \sum_{k=1}^K (V^\pi(s_k) - V_\varphi(s_k)) \nabla_\varphi V_\varphi(s_k)$$

- We can also sample a single state  $s$  (online algorithm):

$$\Delta\varphi = \eta (V^\pi(s) - V_\varphi(s)) \nabla_\varphi V_\varphi(s)$$

- Unless stated otherwise, we will sample single states in this section, but the parameter updates will be noisy (high variance).

# Linear approximation



- The approximated value is a linear combination of the features:

$$V_\varphi(s) = \sum_{i=1}^d w_i \phi_i(s) = \mathbf{w}^T \times \phi(s)$$

- The weights are updated using stochastic gradient descent:

$$\Delta \mathbf{w} = \eta (V^\pi(s) - V_\varphi(s)) \phi(s)$$

- That is the **delta learning rule** of linear regression and classification, with  $\phi(s)$  being the input vector and  $V^\pi(s) - V_\varphi(s)$  the prediction error.

# Function approximation with sampling

- The rule can be used with any function approximator, we only need to be able to differentiate it:

$$\Delta\varphi = \eta (V^\pi(s) - V_\varphi(s)) \nabla_\varphi V_\varphi(s)$$

- The problem is that we do not know  $V^\pi(s)$ , as it is what we are trying to estimate.
- We can replace  $V^\pi(s)$  by a sampled estimate using Monte-Carlo or TD:

- **Monte-Carlo** function approximation:

$$\Delta\varphi = \eta (R_t - V_\varphi(s)) \nabla_\varphi V_\varphi(s)$$

- **Temporal Difference** function approximation:

$$\Delta\varphi = \eta (r_{t+1} + \gamma V_\varphi(s') - V_\varphi(s)) \nabla_\varphi V_\varphi(s)$$

- Note that for Temporal Difference, we actually want to minimize the TD reward-prediction error for all states, i.e. the surprise:

$$\mathcal{L}(\varphi) = \mathbb{E}_{s \in \mathcal{S}} [(r_{t+1} + \gamma V_\varphi(s') - V_\varphi(s))^2] = \mathbb{E}_{s \in \mathcal{S}} [\delta_t^2]$$

# Gradient Monte Carlo Algorithm for value estimation

- Algorithm:

- Initialize the parameter  $\varphi$  to 0 or randomly.

- **while** not converged:

- 1. Generate an episode according to the current policy  $\pi$  until a terminal state  $s_T$  is reached.

$$\tau = (s_o, a_o, r_1, s_1, a_1, \dots, s_T)$$

- 2. For all encountered states  $s_0, s_1, \dots, s_{T-1}$ :

- 1. Compute the return  $R_t = \sum_k \gamma^k r_{t+k+1}$ .

- 2. Update the parameters using function approximation:

$$\Delta\varphi = \eta (R_t - V_\varphi(s_t)) \nabla_\varphi V_\varphi(s_t)$$

- Gradient Monte-Carlo has no bias (real returns) but a high variance.

# Semi-gradient Temporal Difference Algorithm for value estimation

- Algorithm:

- Initialize the parameter  $\varphi$  to 0 or randomly.
- **while** not converged:
  - Start from an initial state  $s_0$ .
  - **foreach** step  $t$  of the episode:
    - Select  $a_t$  using the current policy  $\pi$  in state  $s_t$ .
    - Observe  $r_{t+1}$  and  $s_{t+1}$ .
    - Update the parameters using function approximation:

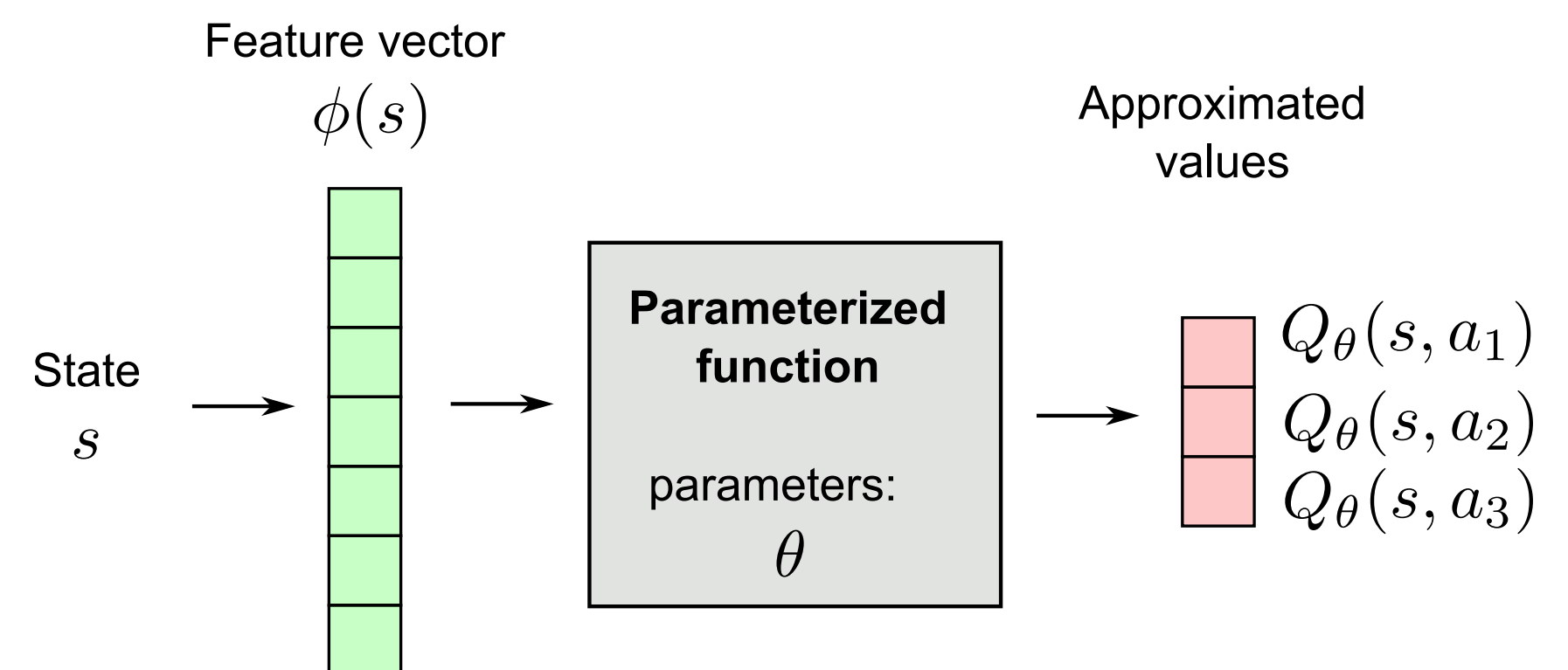
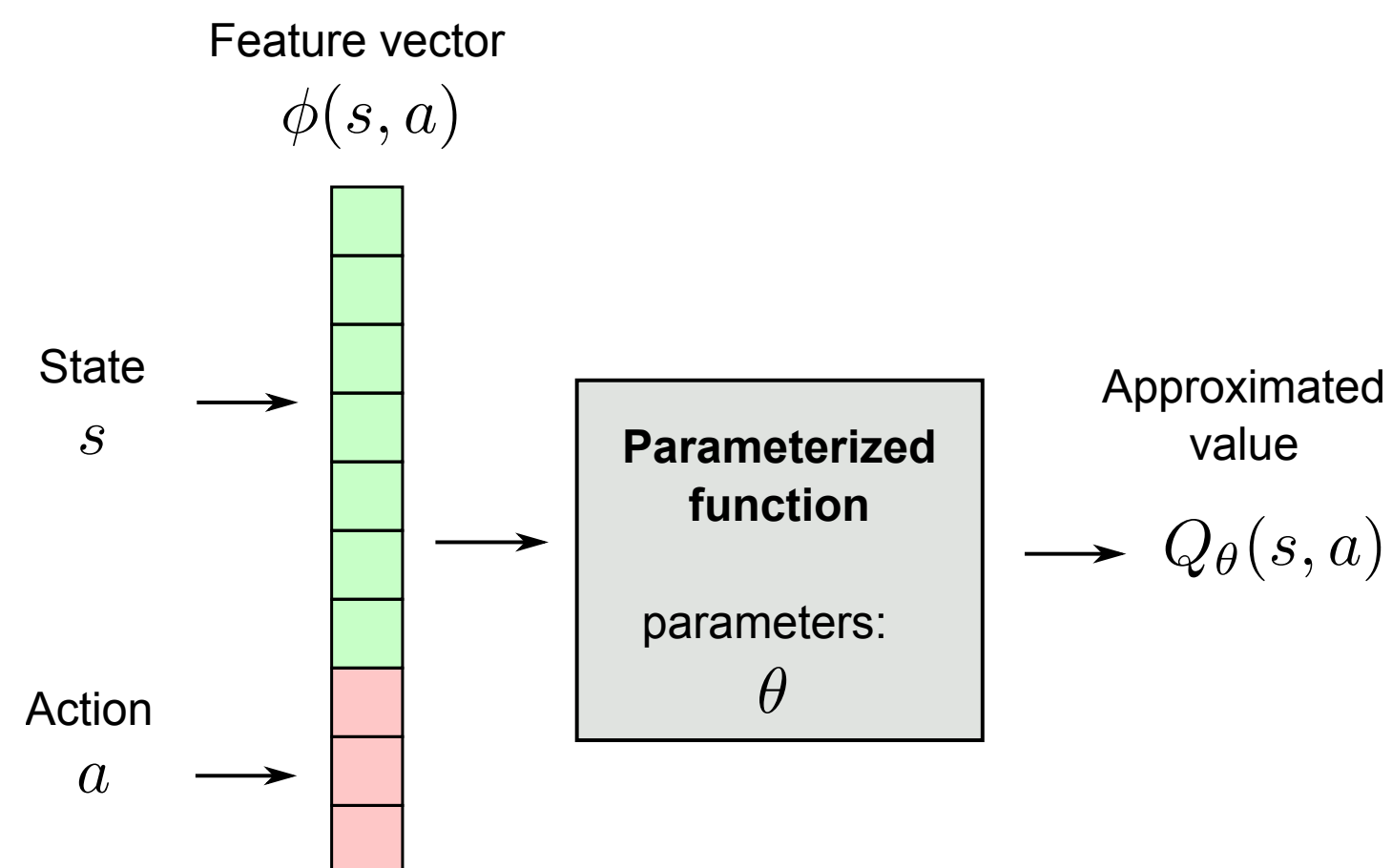
$$\Delta\varphi = \eta (r_{t+1} + \gamma V_{\varphi}(s_{t+1}) - V_{\varphi}(s_t)) \nabla_{\varphi} V_{\varphi}(s_t)$$

- **if**  $s_{t+1}$  is terminal: **break**
- Semi-gradient TD has less variance, but a significant bias as  $V_{\varphi}(s_{t+1})$  is initially wrong. You can never trust these estimates completely.



# Function approximation for Q-values

- Q-values can be approximated by a parameterized function  $Q_{\theta}(s, a)$  in the same manner.
- There are basically two options for the structure of the function approximator:
  - The FA takes a feature vector for both the state  $s$  and the action  $a$  (which can be continuous) as inputs, and outputs a single Q-value  $Q_{\theta}(s, a)$ .
  - The FA takes a feature vector for the state  $s$  as input, and outputs one Q-value  $Q_{\theta}(s, a)$  per possible action (the action space must be discrete).



- In both cases, we minimize the mse between the true value  $Q^{\pi}(s, a)$  and the approximated value  $Q_{\theta}(s, a)$ .

# Q-learning with function approximation

- Initialize the parameters  $\theta$ .
- **while** True:
  - Start from an initial state  $s_0$ .
  - **foreach** step  $t$  of the episode:
    - Select  $a_t$  using the behavior policy  $b$  (e.g. derived from  $\pi$ ).
    - Take  $a_t$ , observe  $r_{t+1}$  and  $s_{t+1}$ .
    - Update the parameters  $\theta$ :

$$\Delta\theta = \eta (r_{t+1} + \gamma \max_a Q_\theta(s_{t+1}, a) - Q_\theta(s_t, a_t)) \nabla_\theta Q_\theta(s_t, a_t)$$

- Improve greedily the learned policy:

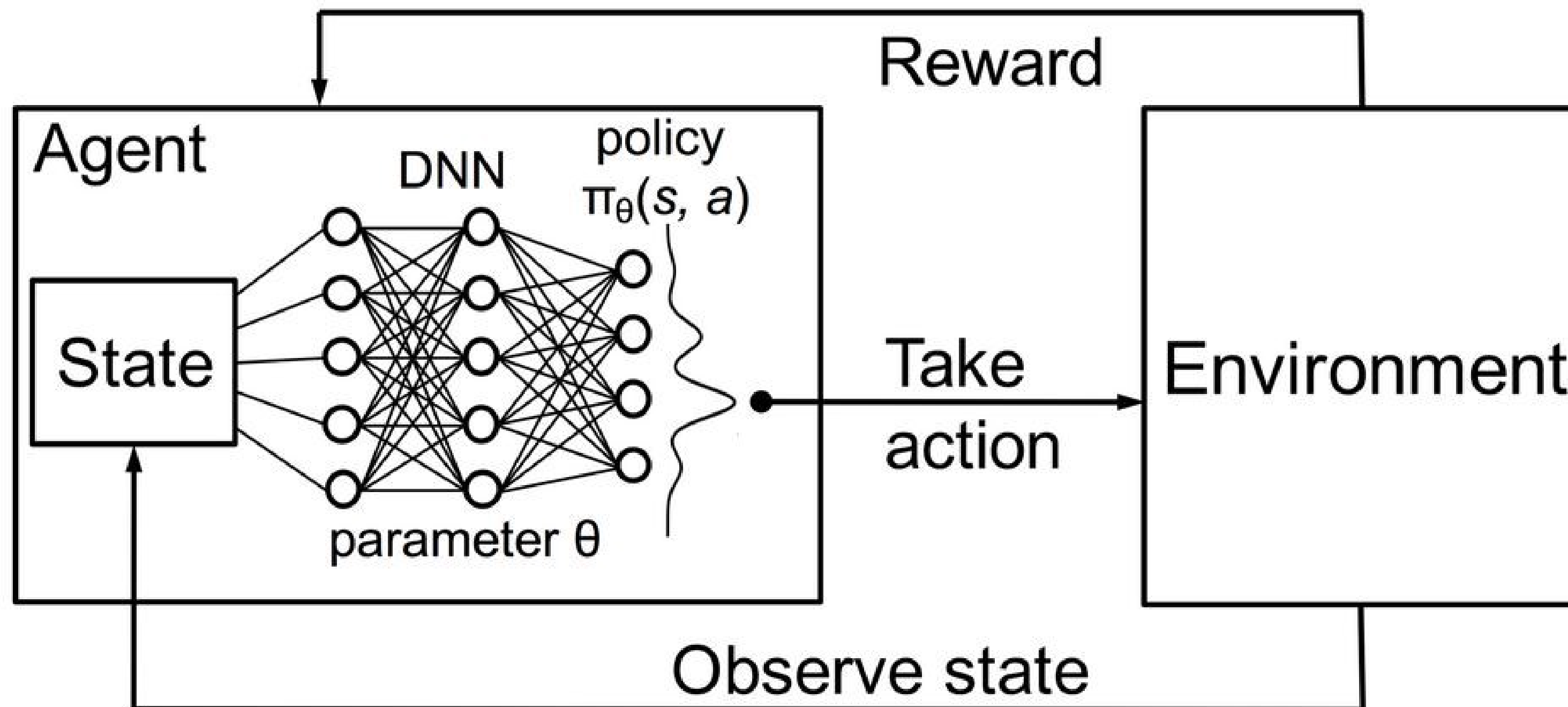
$$\pi(s_t, a) = \text{Greedy}(Q_\theta(s_t, a))$$

- **if**  $s_{t+1}$  is terminal: **break**

## 3 - Feature construction

# Feature construction

- Before we dive into deep RL (i.e. RL with non-linear FA), let's see how we can design good **feature vectors** for linear function approximation.

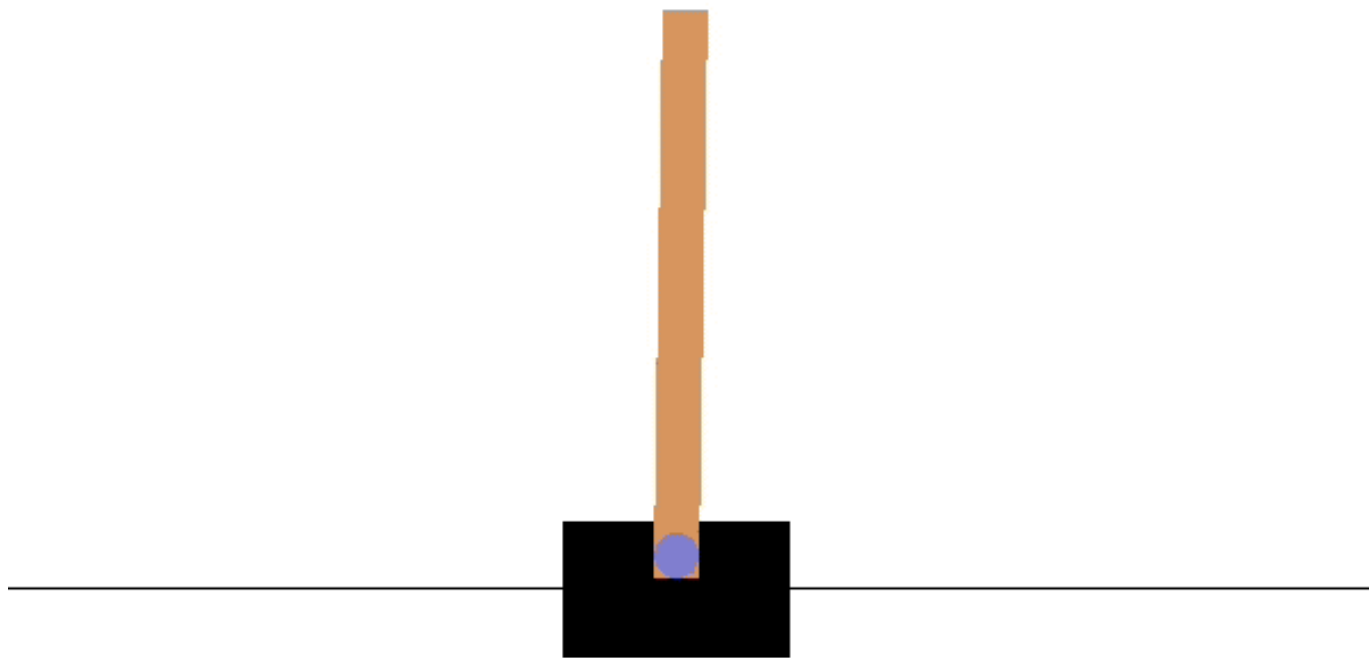


- The problem with deep NN is that they need a lot of samples to converge, what worsens the fundamental problem of RL: **sample efficiency**.
- By engineering the right features, we could use linear approximators, which converge much faster.
- The convergence of linear FA is **guaranteed**, not (always) non-linear ones.

# Why do we need to choose features?

- For the cartpole, the feature vector  $\phi(s)$  could be:

$$\phi(s) = \begin{bmatrix} x \\ \dot{x} \\ \theta \\ \dot{\theta} \end{bmatrix}$$



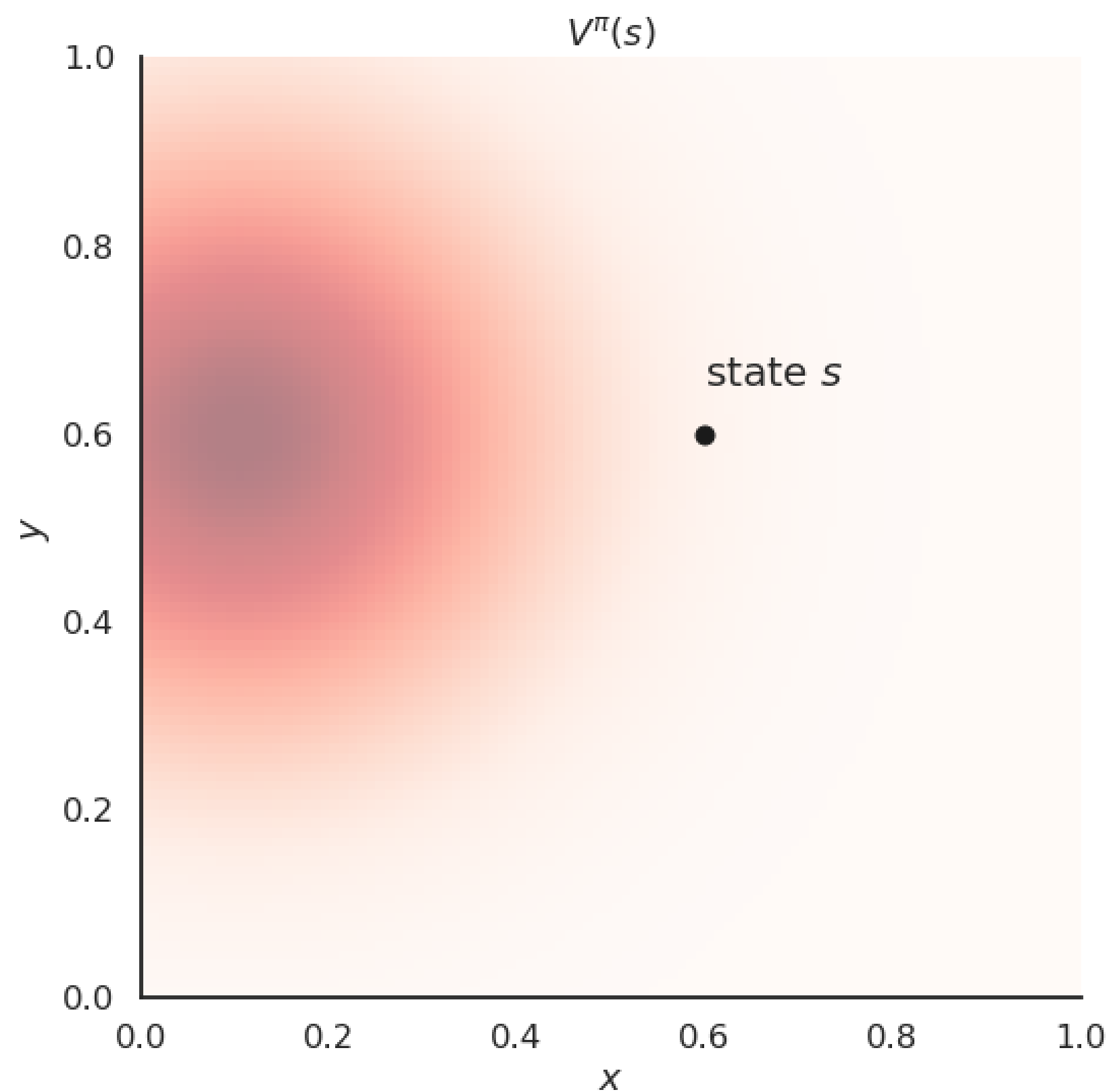
- $x$  is the position,  $\theta$  the angle,  $\dot{x}$  and  $\dot{\theta}$  their derivatives.
- Can we predict the value of a state **linearly**?

$$V_{\varphi}(s) = \sum_{i=1}^d w_i \phi_i(s) = \mathbf{w}^T \times \phi(s)$$

- No, a high angular velocity  $\dot{\theta}$  is good when the pole is horizontal (going up) but bad if the pole is vertical (will not stop).
- The value would depend linearly on something like  $\dot{\theta} \sin \theta$ , which is a non-linear combination of features.

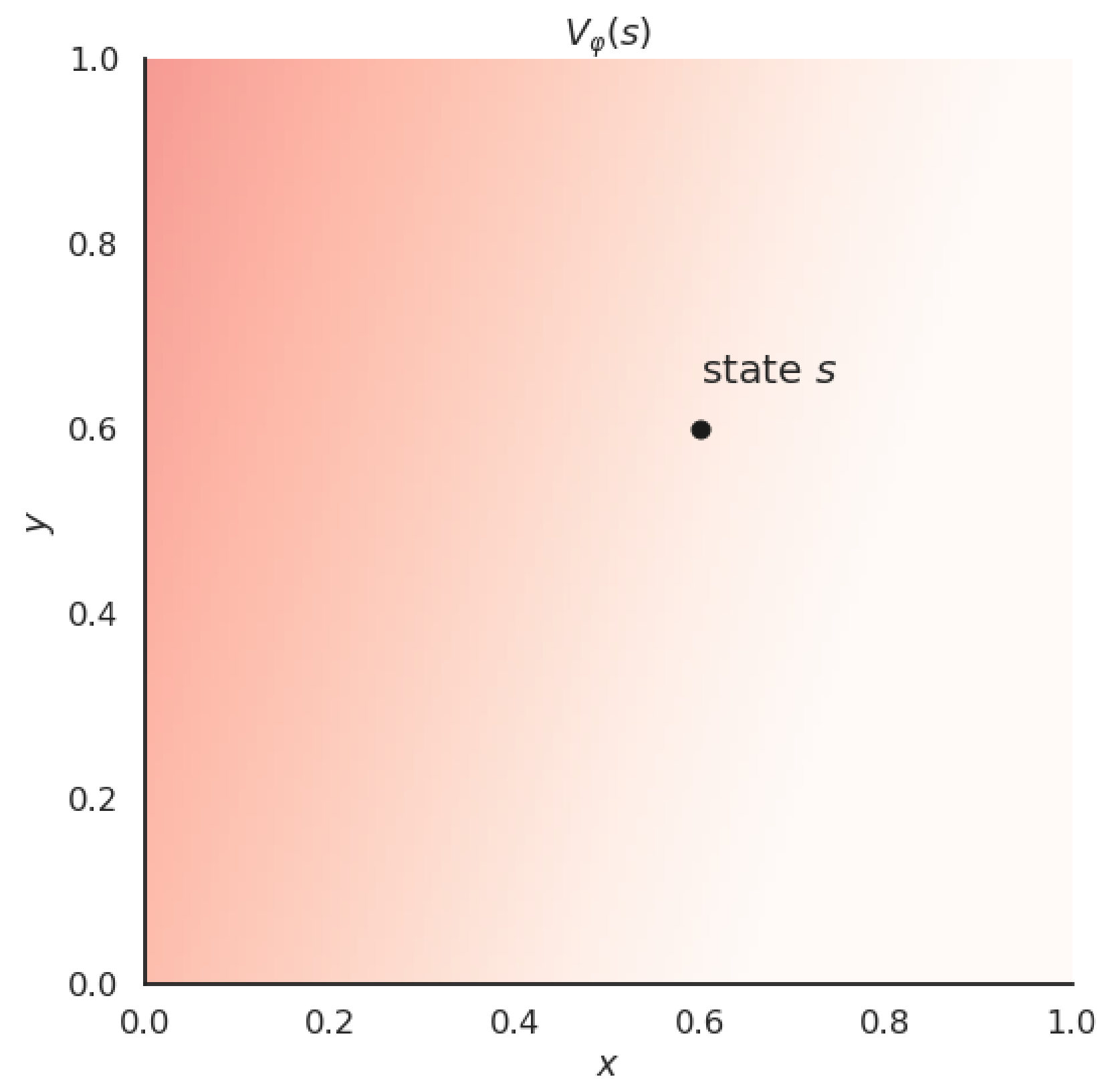
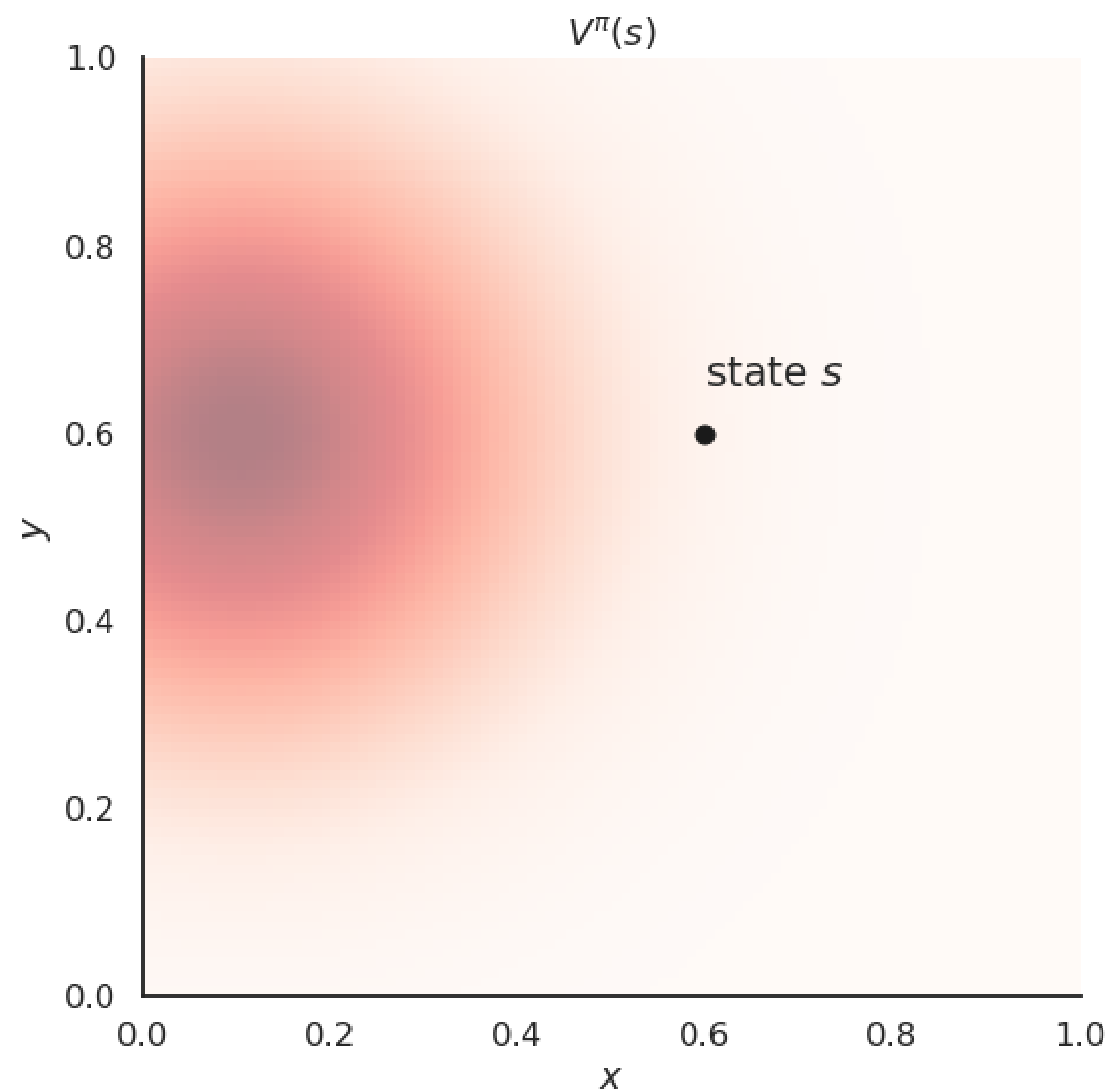
# Feature coding

- Let's suppose we have a simple problem where the state  $s$  is represented by two continuous variables  $x$  and  $y$ .
- The true value function  $V^\pi(s)$  is a non-linear function of  $x$  and  $y$ .



# Linear approximation

- If we apply linear FA directly on the feature vector  $[x, y]$ , we catch the tendency of  $V^\pi(s)$  but we make a lot of bad predictions:
  - **high bias** (underfitting).





# Polynomials

- To introduce non-linear relationships between continuous variables, a simple method is to construct the feature with **polynomials** of the variables.
- Example with polynomials of order 2:

$$\phi(s) = [1 \quad x \quad y \quad xy \quad x^2 \quad y^2]^T$$

- We transform the two input variables  $x$  and  $y$  into a vector with 6 elements. The 1 (order 0) is there to learn the offset.
- Example with polynomials of order 3:

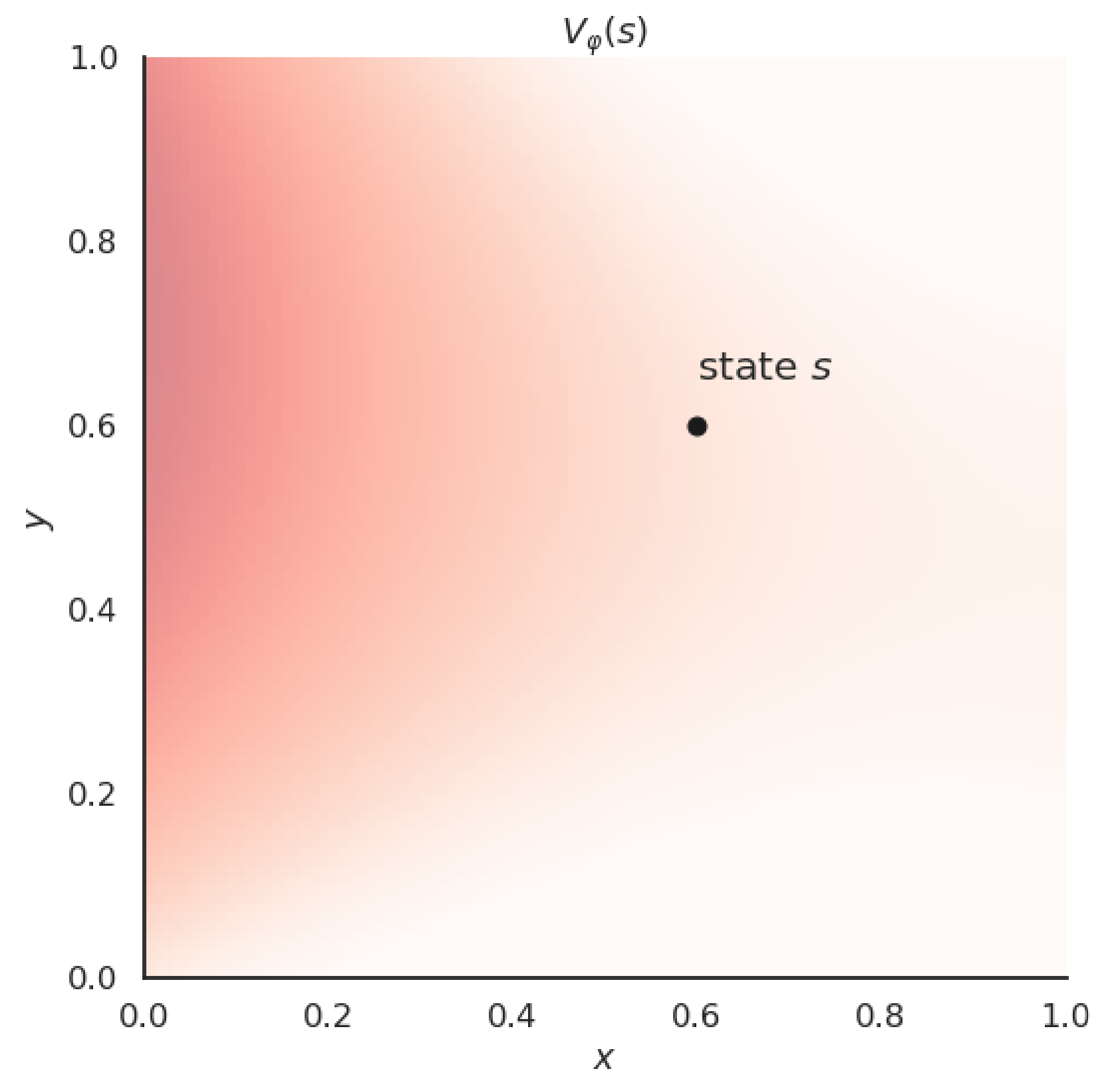
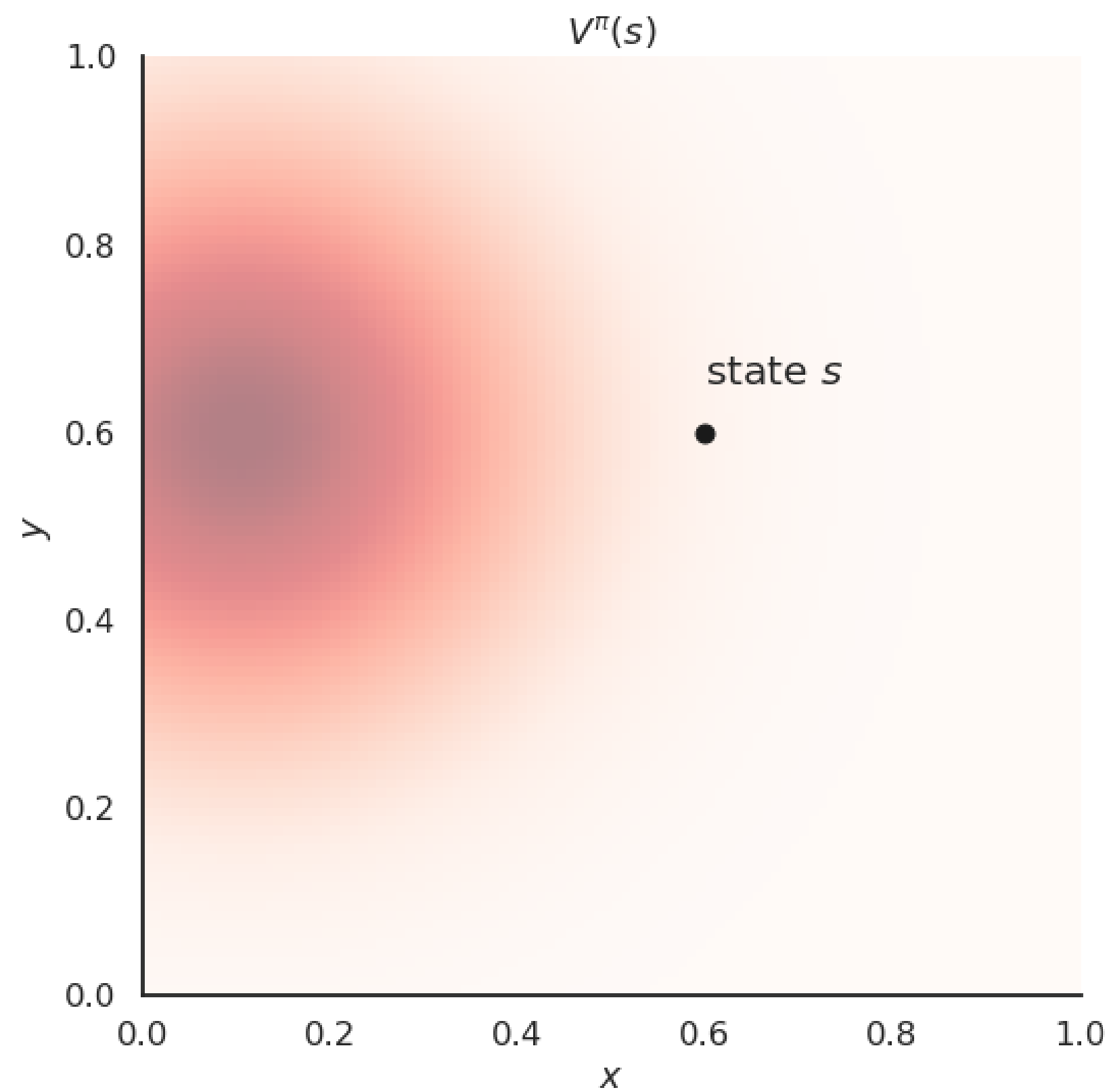
$$\phi(s) = [1 \quad x \quad y \quad xy \quad x^2 \quad y^2 \quad x^2y \quad xy^2 \quad x^3 \quad y^3]^T$$

- And so on. We then just need to apply linear FA on these feature vectors (**polynomial regression**).

$$V_\varphi(s) = w_0 + w_1 x + w_2 y + w_3 xy + w_4 x^2 + w_5 y^2 + \dots$$

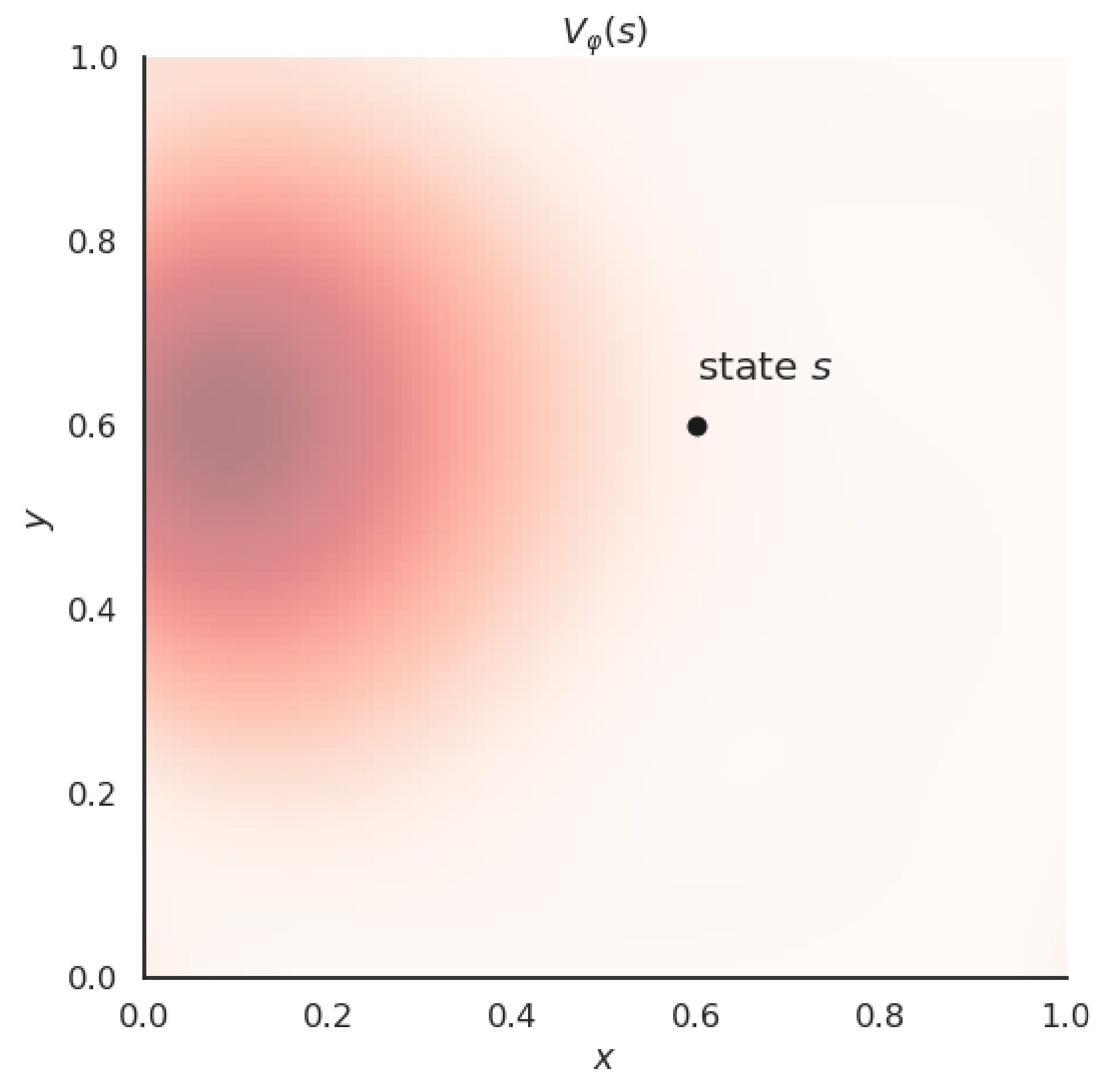
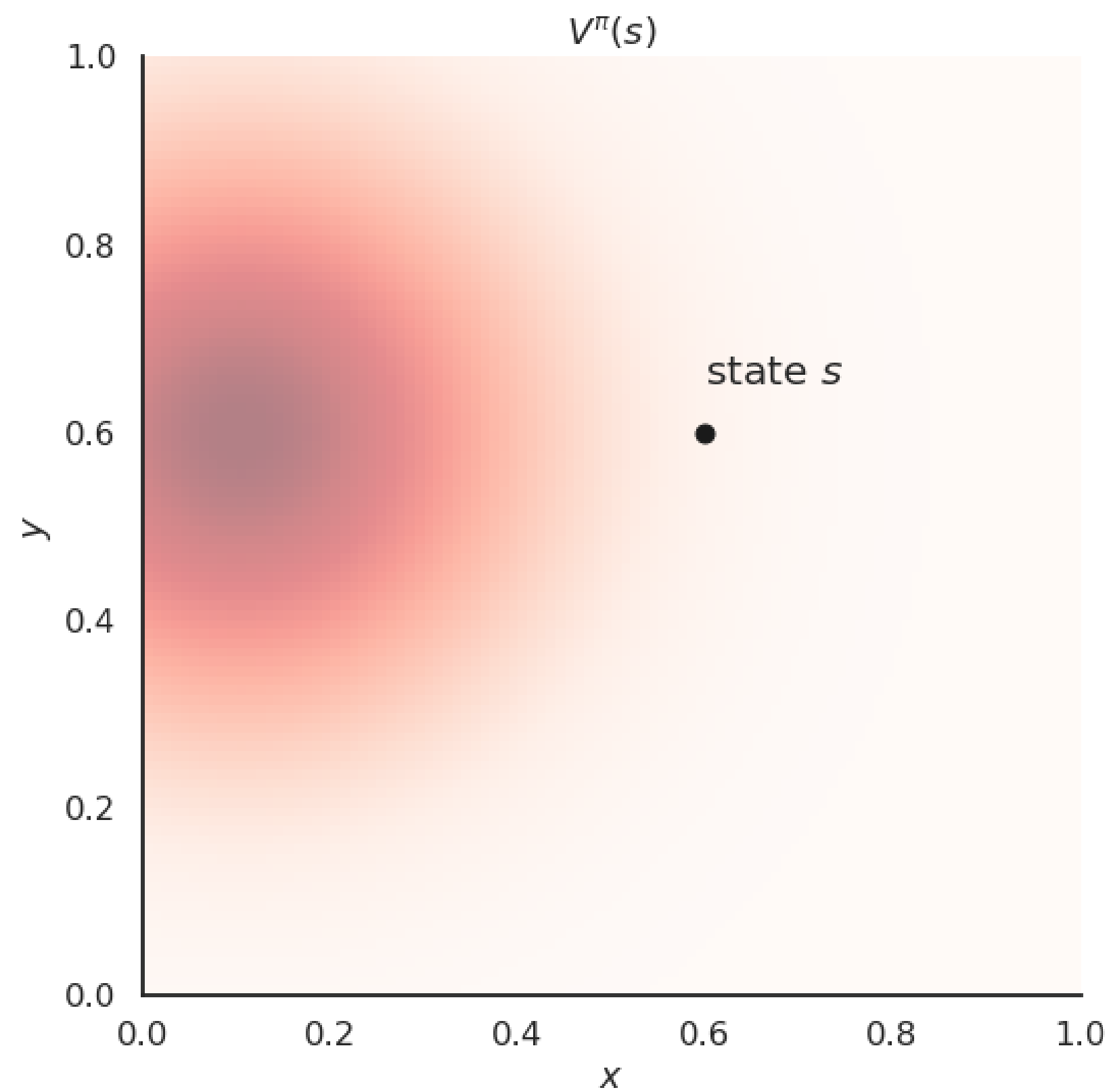
# Polynomials

- Polynomials of order 2 already allow to get a better approximation.



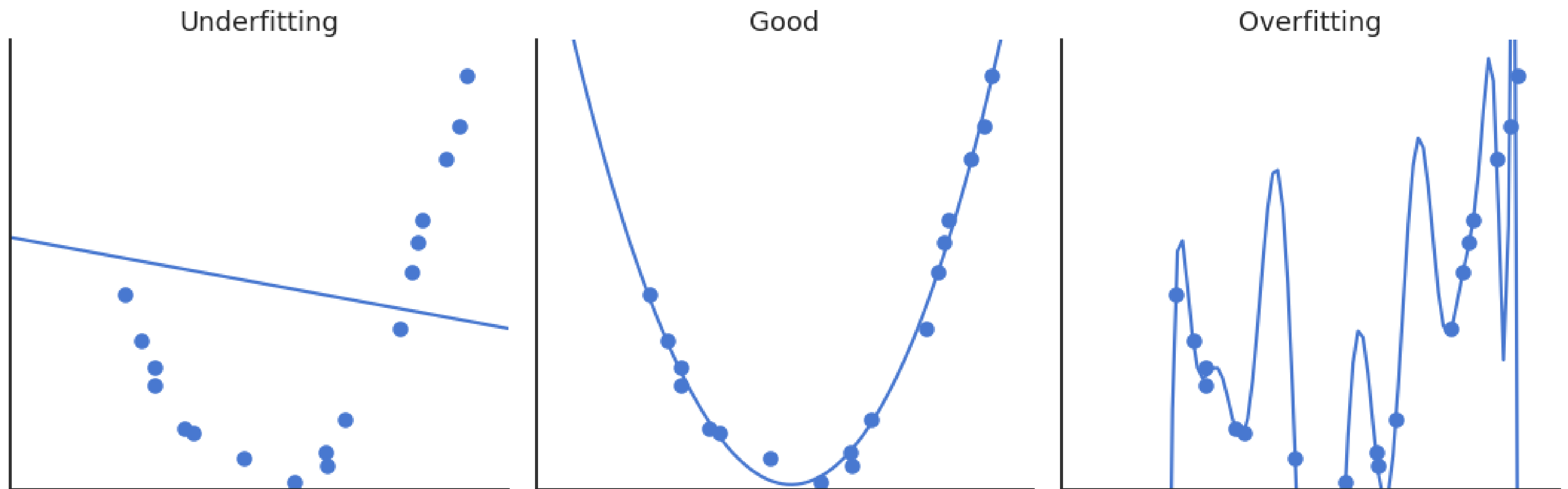
# Polynomials

- Polynomials of order 6 are an even better fit for our problem.



# Polynomials

- The higher the degree of the polynomial, the better the fit, but the number of features grows exponentially.
  - Computational complexity.
  - **Overfitting**: if we only sample some states, high-order polynomials will not interpolate correctly.



# Fourier transforms

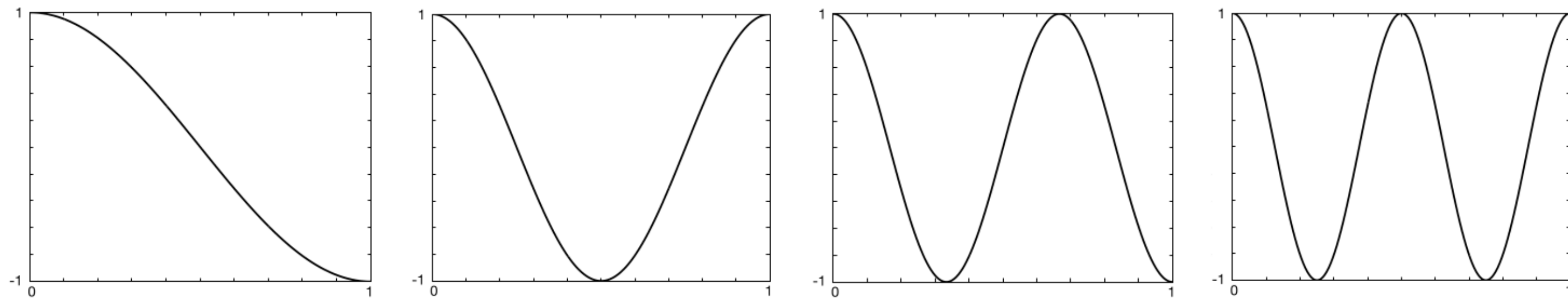
- Instead of approximating a state variable  $x$  by a polynomial:

$$V_\varphi(s) = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + \dots$$

- we could also use its **Fourier decomposition** (here DCT, discrete cosine transform):

$$V_\varphi(s) = w_0 + w_1 \cos(\pi x) + w_2 \cos(2\pi x) + w_3 \cos(3\pi x) + \dots$$

- Fourier tells us that, if we take enough frequencies, we can reconstruct the signal  $V_\varphi(s)$  perfectly.

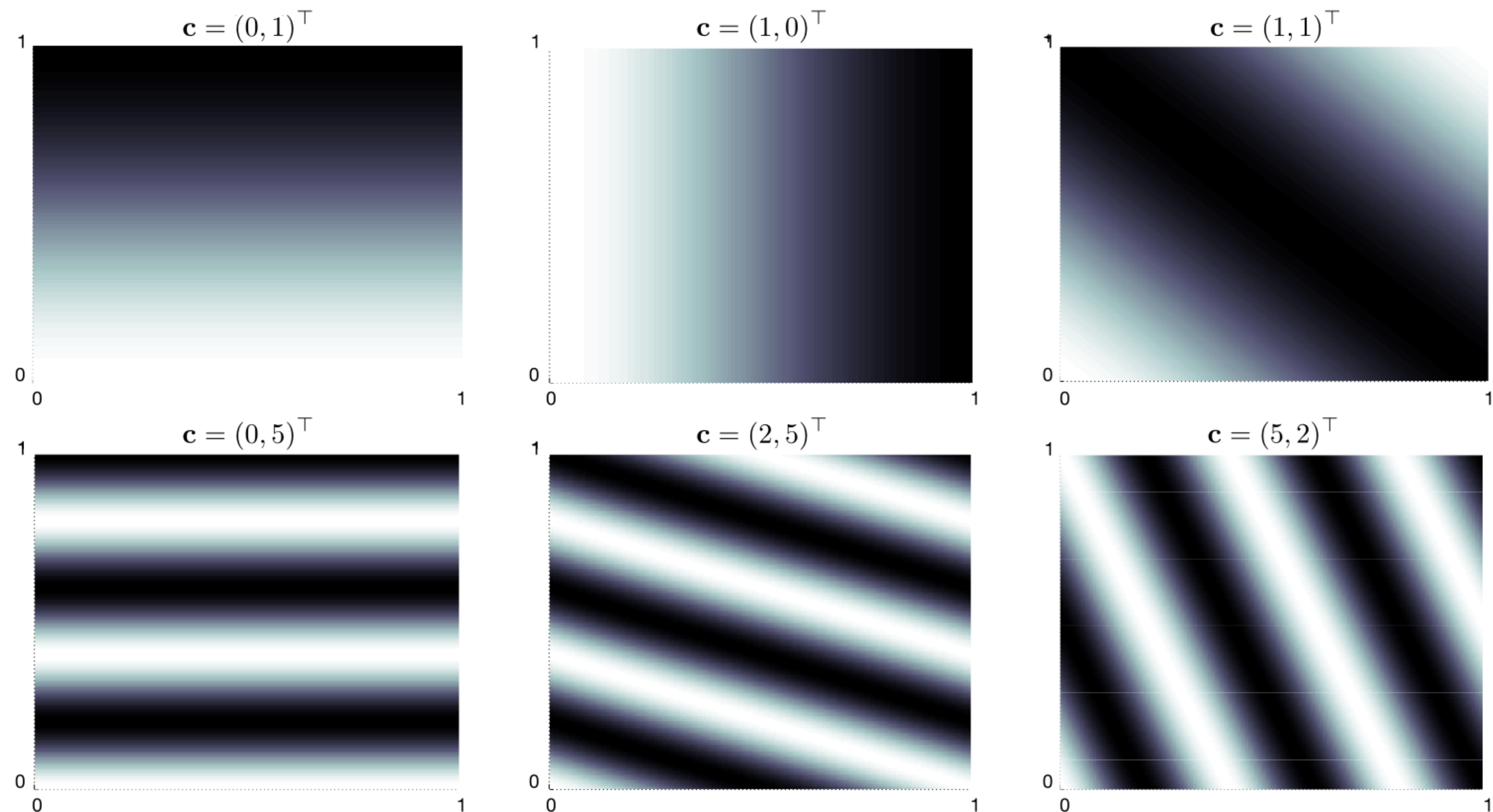


**Figure 9.3:** One-dimensional Fourier cosine-basis features  $x_i$ ,  $i = 1, 2, 3, 4$ , for approximating functions over the interval  $[0, 1]$ . After Konidaris et al. (2011).

- It is just a change of basis, the problem stays a linear regression to find  $w_0, w_1, w_2$ , etc.

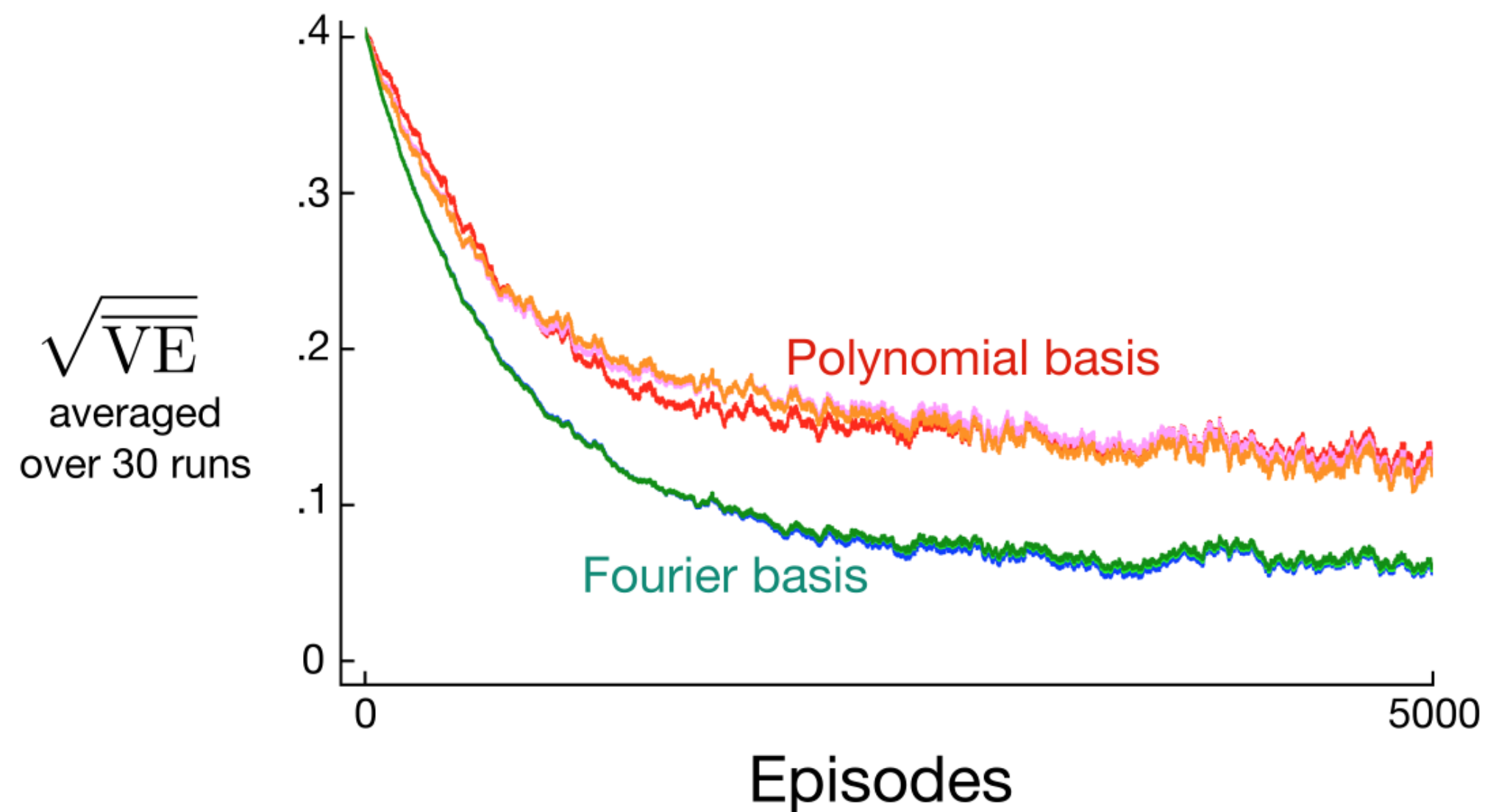
# Fourier transforms

- Fourier transforms can be applied on multivariate functions as well.



**Figure 9.4:** A selection of six two-dimensional Fourier cosine features, each labeled by the vector  $\mathbf{c}^i$  that defines it ( $s_1$  is the horizontal axis, and  $\mathbf{c}^i$  is shown with the index  $i$  omitted). After Konidaris et al. (2011).

# Polynomial vs. Fourier basis



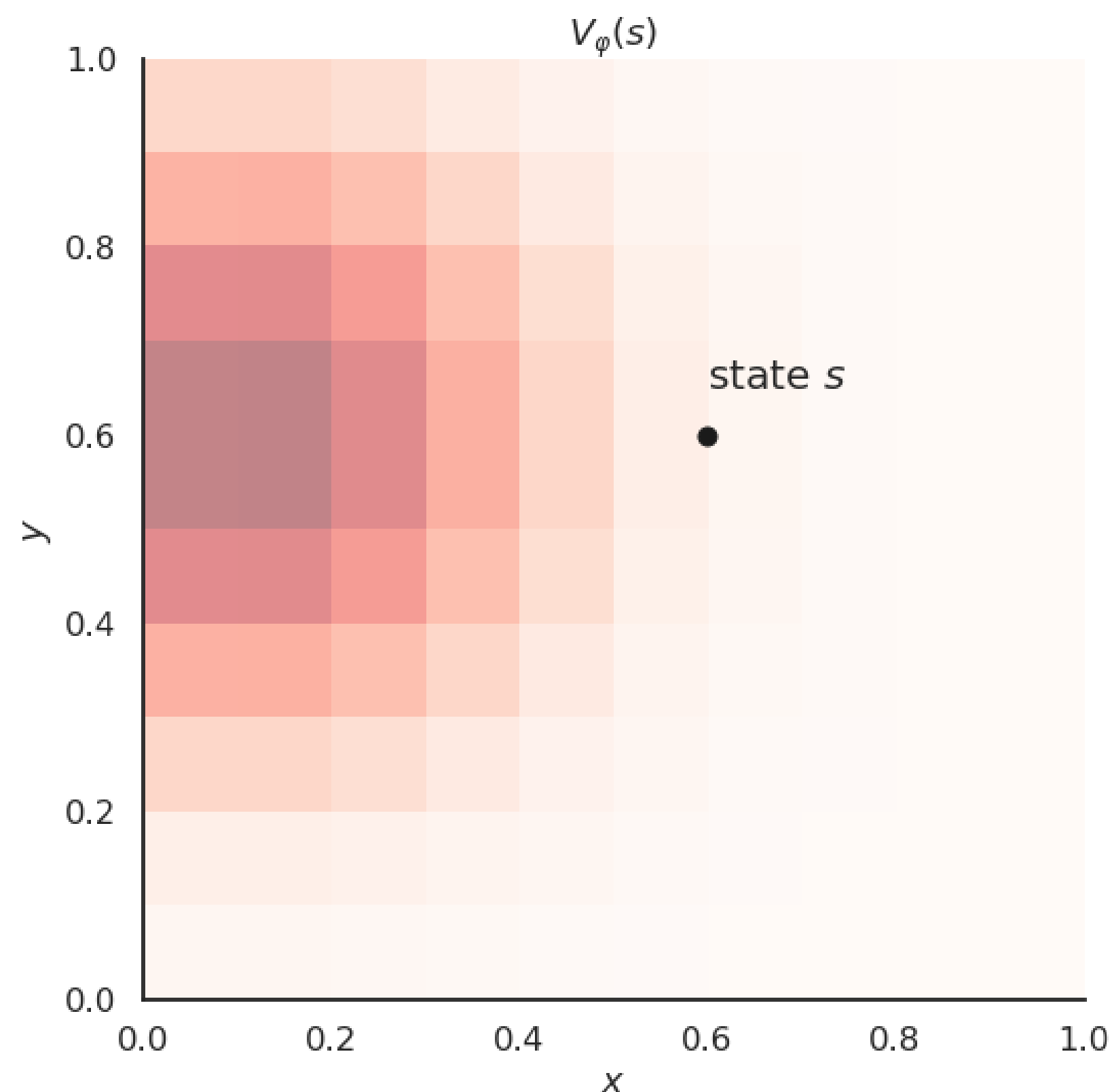
**Figure 9.5:** Fourier basis vs polynomials on the 1000-state random walk. Shown are learning curves for the gradient Monte Carlo method with Fourier and polynomial bases of order 5, 10, and 20. The step-size parameters were roughly optimized for each case:  $\alpha = 0.0001$  for the polynomial basis and  $\alpha = 0.00005$  for the Fourier basis. The performance measure (y-axis) is the root mean squared value error (9.1).

- A Fourier basis tends to work better than a polynomial basis.
- The main problem is that the number of features increases very fast with:
  - the number of input dimensions.
  - the desired precision (higher-order polynomials, more frequencies).



# Discrete coding

- An obvious solution for continuous state variables is to **discretize** the input space.
- The input space is divided into a grid of non-overlapping **tiles**.

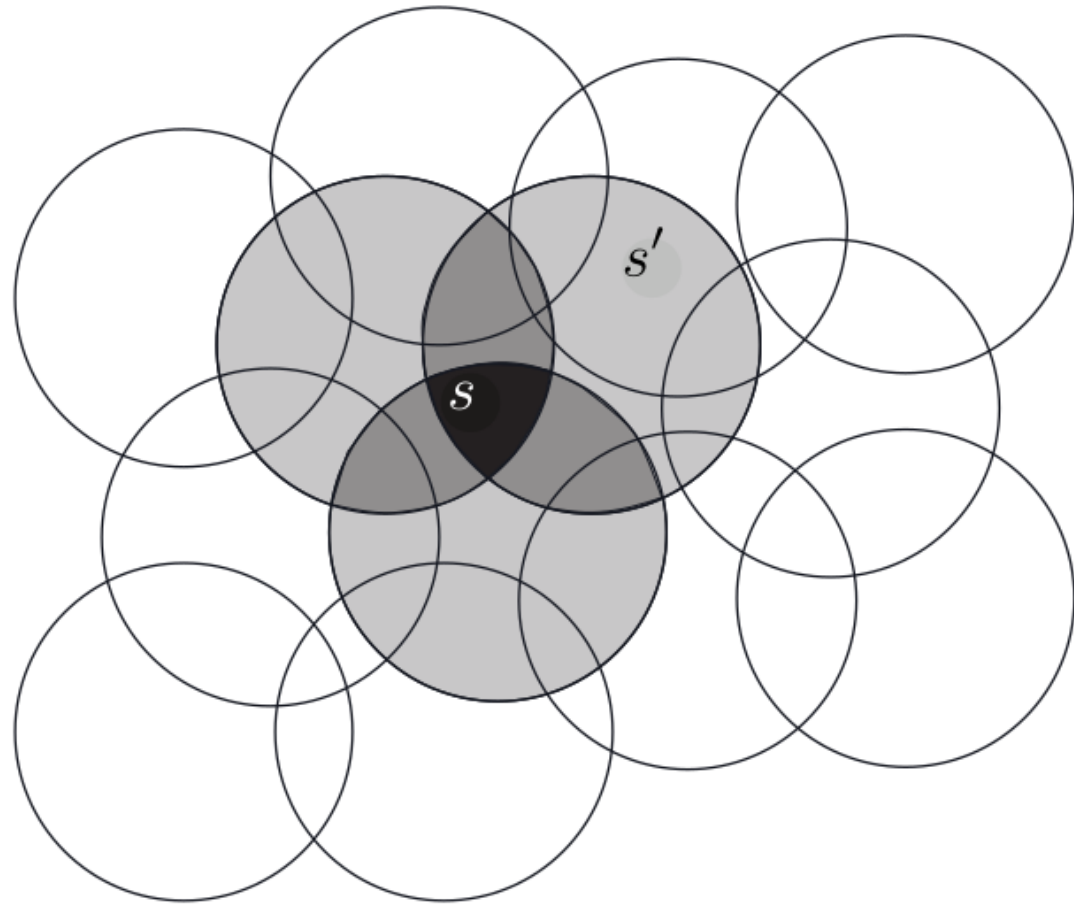


- The feature vector is a **binary** vector with a 1 when the input is inside a tile, 0 otherwise.

$$\phi(s) = [0 \quad 0 \quad \dots \quad 0 \quad 1 \quad 0 \quad \dots \quad 0]^T$$

- This ensures **generalization** inside a tile: you only need a couple of samples inside a tile to know the mean value of all the states.
- Drawbacks:
  - the value function is step-like (discontinuous).
  - what is the correct size of a tile?
  - curse of dimensionality.

# Coarse coding

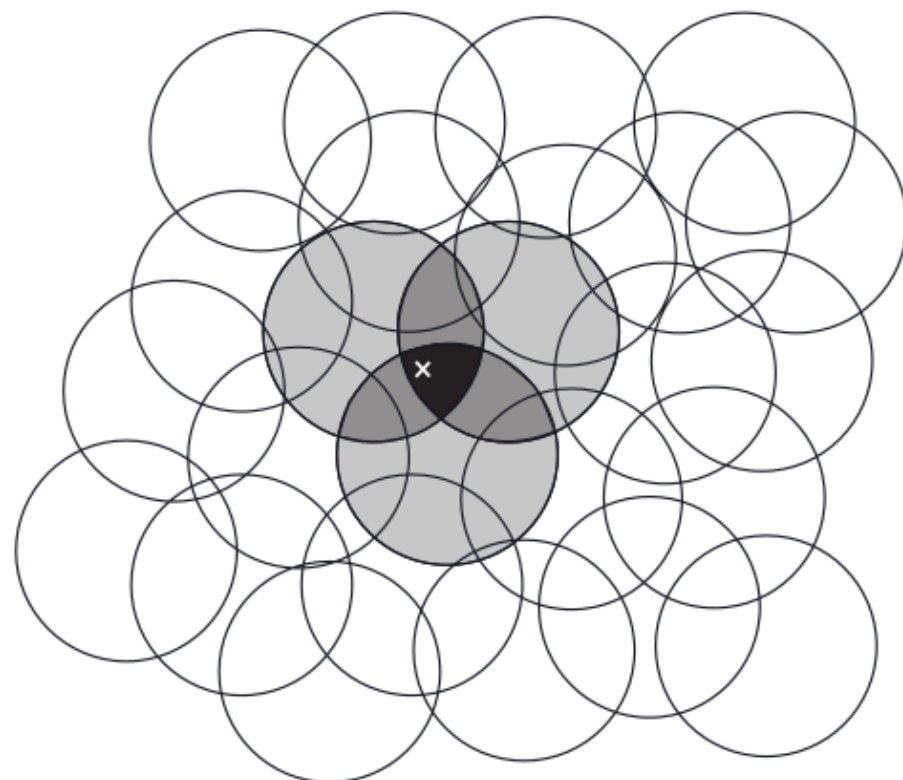


- A more efficient solution is **coarse coding**.
- The tiles (rectangles, circles, or what you need) need to **overlap**.
- A state  $s$  is encoded by a **binary vector**, but with several 1, for each tile it belongs to.

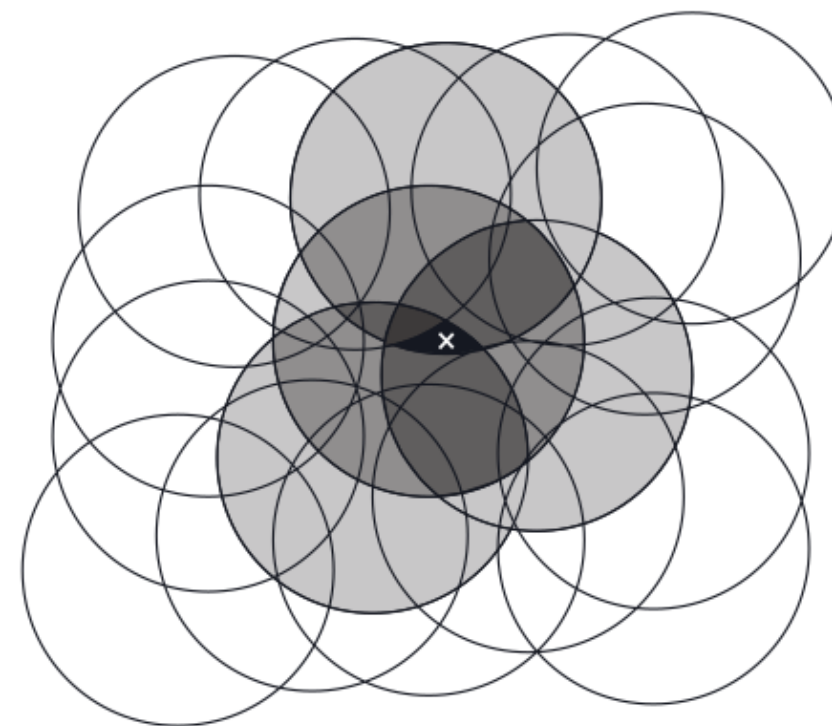
$$\phi(s) = [0 \quad 1 \quad 0 \quad \dots \quad 1 \quad 1 \quad 0 \quad \dots \quad 0]^T$$

- This allows generalization inside a tile, but also **across tiles**.

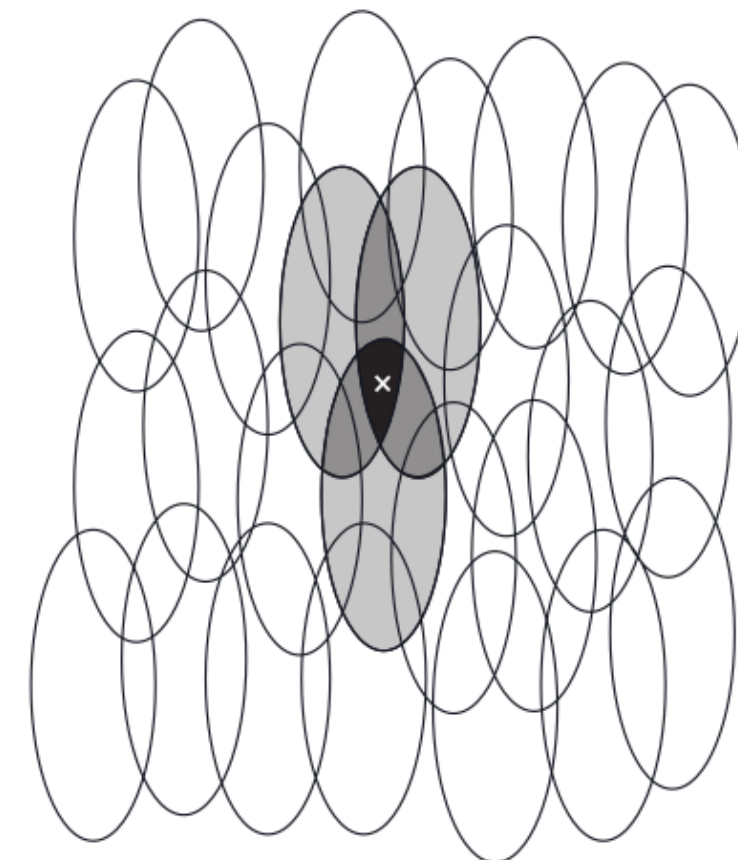
- The size and shape of the “**receptive field**” influences the generalization properties.



Narrow generalization



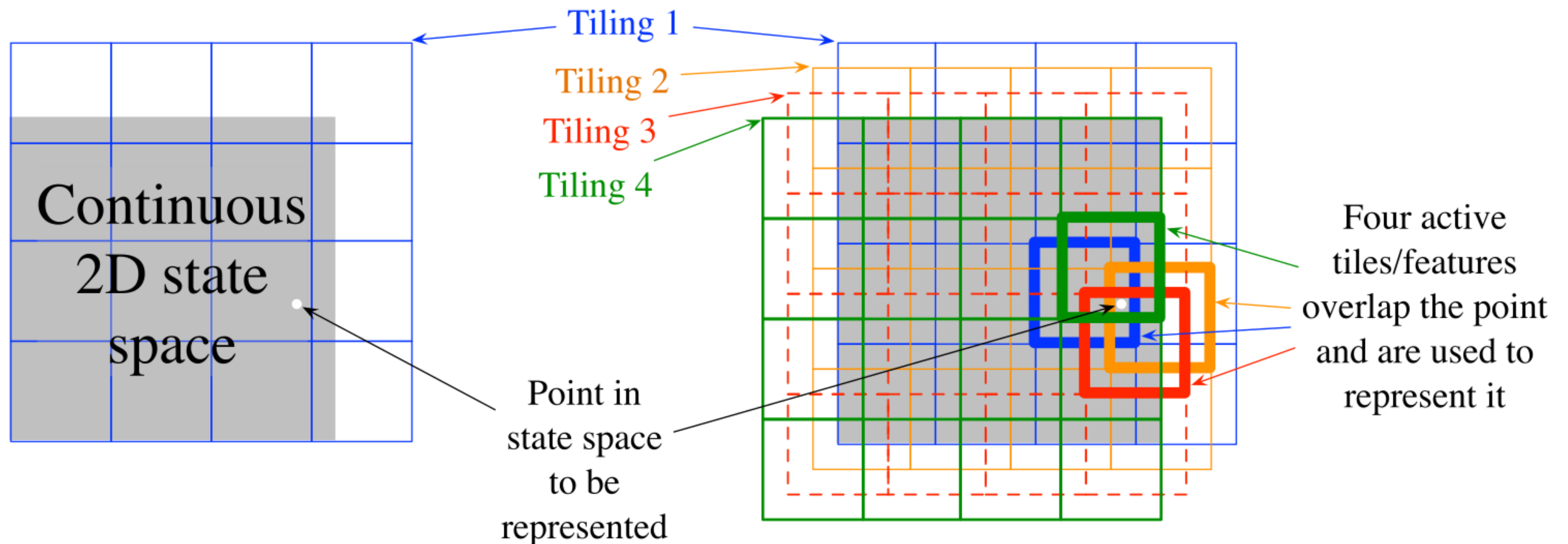
Broad generalization



Asymmetric generalization

# Tile coding

- A simple way to ensure that tiles overlap is to use several regular grids with an **offset**.
- Each tiling will be **coarse**, but the location of a state will be quite precise as it may belong to many tiles.

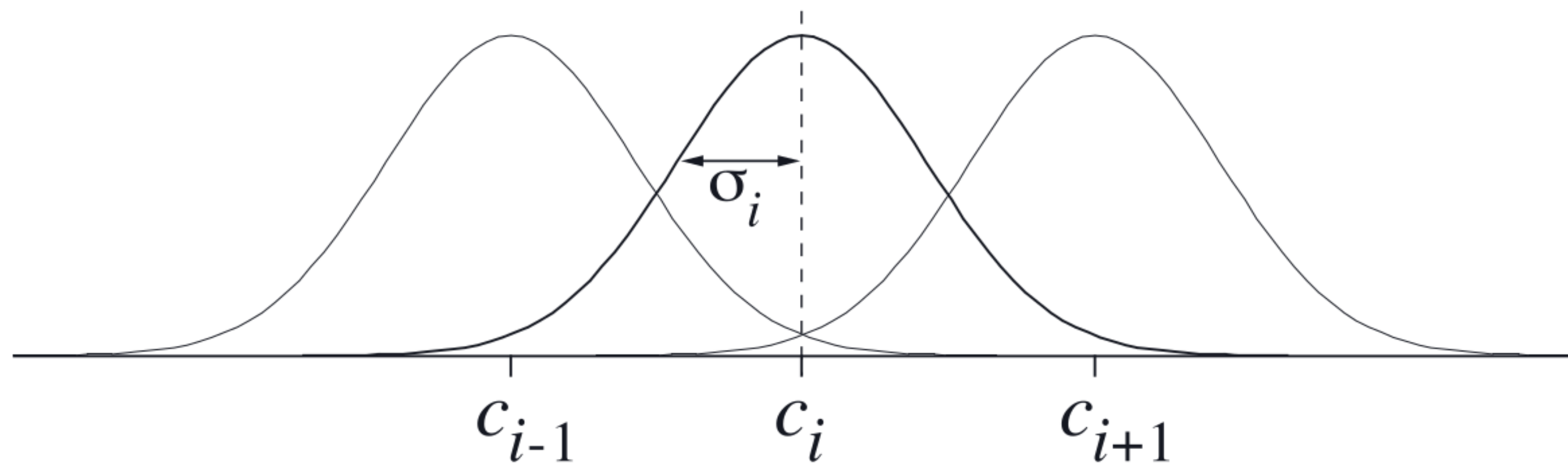


**Figure 9.9:** Multiple, overlapping grid-tilings on a limited two-dimensional space. These tilings are offset from one another by a uniform amount in each dimension.

- This helps against the curse of dimensionality: high precision, but the number of tiles does not grow exponentially.

# Radial-basis functions (RBF)

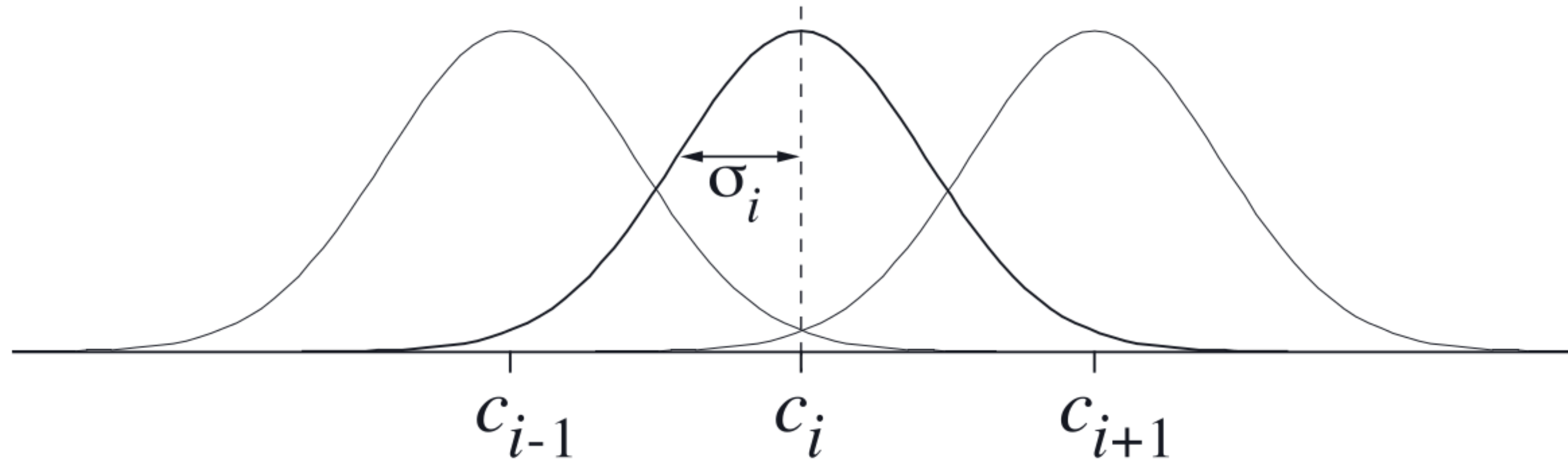
- The feature vector in tile coding is a binary vector: there will be **discontinuous jumps** in the approximated value function when moving between tiles.
- We can use **radial-basis functions** (RBF) such as Gaussians to map the state space.



- We set a set of centers  $\{c_i\}_{i=1}^K$  in the input space on a regular grid (or randomly).
- Each element of the feature vector will be a Gaussian function of the distance between the state  $s$  and one center:

$$\phi_i(s) = \exp \frac{-(s - c_i)^2}{2 \sigma_i^2}$$

# Radial-basis functions (RBF)



- The approximated value function now represents **continuously** the states:

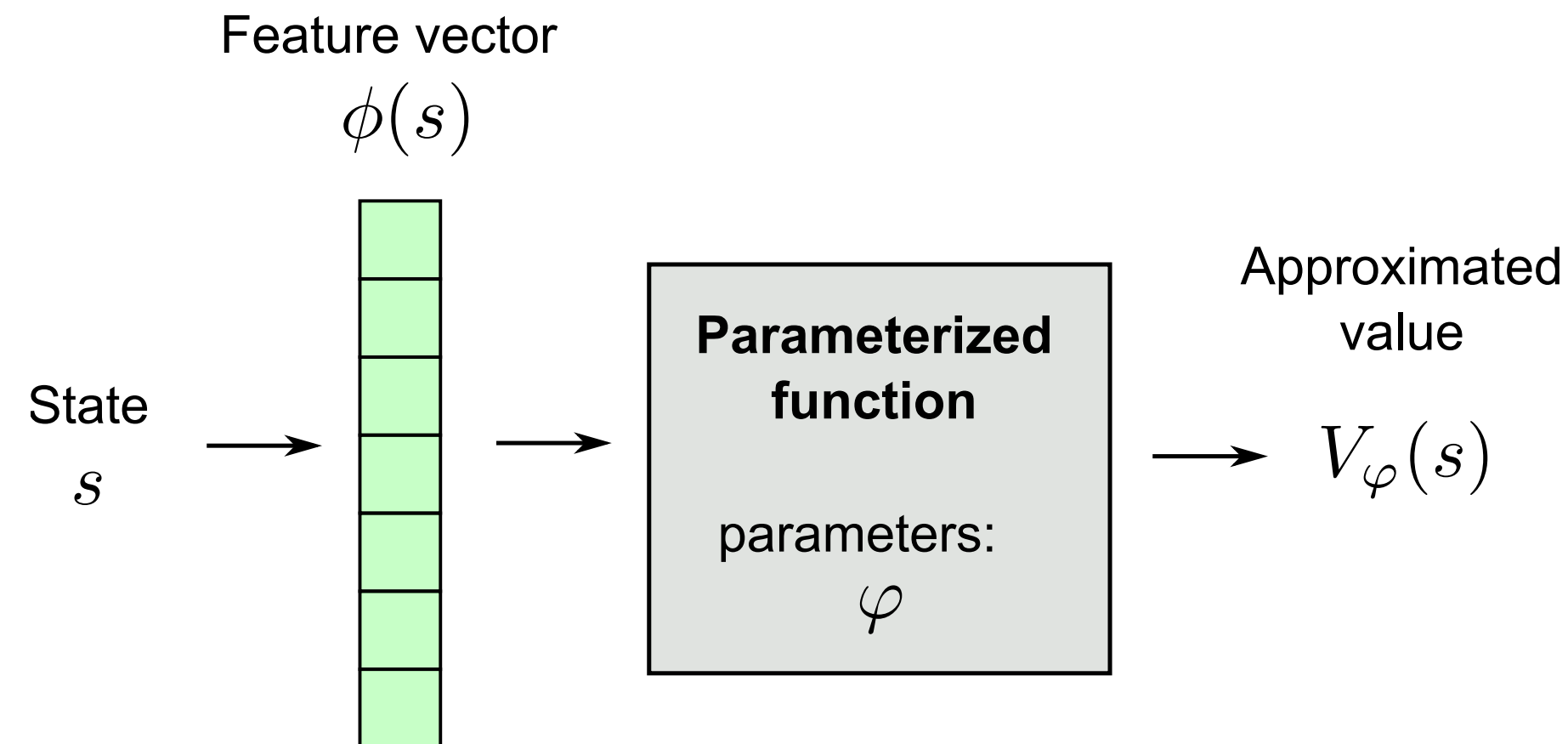
$$V_{\varphi}(s) = \sum_{i=1}^d w_i \phi_i(s) = \sum_{i=1}^d w_i \exp \frac{-(s - c_i)^2}{2 \sigma_i^2}$$

- If you have enough centers and they overlap sufficiently, you can even **decode** the original state perfectly:

$$\hat{s} = \sum_{i=1}^d \phi_i(s) c_i$$



# Summary of function approximation



- In FA, we project the state information into a **feature space** to get a better representation.
- We then apply a linear approximation algorithm to estimate the value function:

$$V_\varphi(s) = \mathbf{w}^T \phi(s)$$

- The linear FA is trained using some variant of gradient decent:

$$\Delta \mathbf{w} = \eta (V^\pi(s) - V_\varphi(s)) \phi(s)$$

- **Deep neural networks** are the most powerful function approximators in supervised learning.
- Do they also work with RL?