Chapter 9:

On-policy Prediction with Approximation

$$\cancel{V(s) \approx} \; v_\pi(s) \tag{1.71}$$

$$\mathbf{w} \in \mathbb{R}^d, \quad e.g., \quad \mathbf{w} = \begin{bmatrix} 2.1 \\ 0.01 \\ -1.1 \\ 1.2 \\ -0.1 \\ 0.01 \\ 4.93 \\ 0.5 \end{bmatrix}, \qquad \mathbf{x}(s) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{x} : \mathcal{S} \to \mathbb{R}^d$$

parameter vector

feature vector

$$\cancel{Q(s,a) \approx} \; q_\pi(s,a)$$

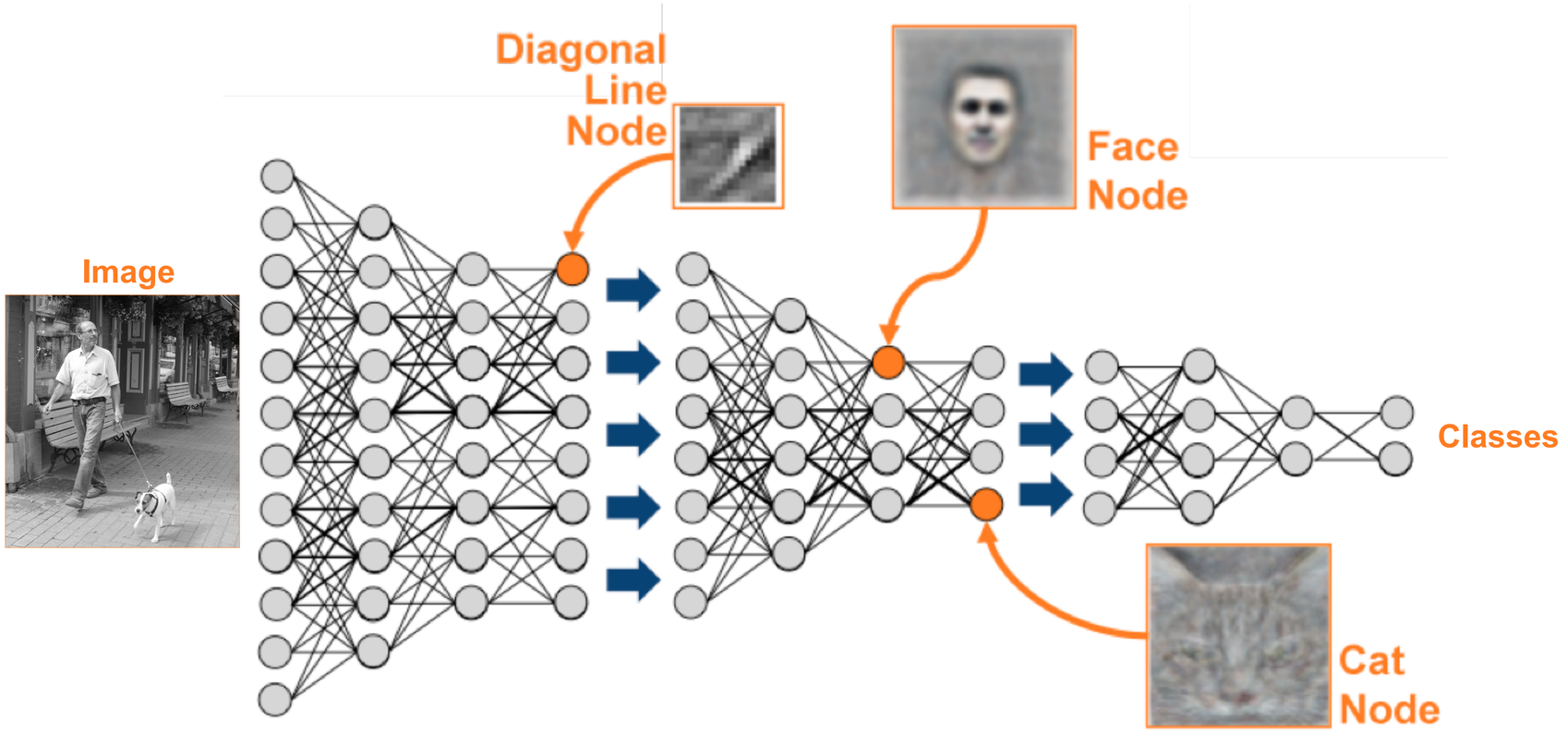
3 waves of neural networks

- First explored in the 1950-60s: Perceptron, Adaline…
  - only one learnable layer
- Revived in the 1980-90s as Connectionism, Neural Networks
  - exciting multi-layer learning using backpropagation (SGD); many successful applications; remained popular in engineering
- Revived again in ~2010 as Deep Learning
  - dramatically improved over state-of-the-art in speech recognition and visual object recognition, *transforming these fields*
  - the best algorithms were essentially the same as in the 1980s, except with faster computers and larger training sets

i.e., NNs won (eventually) because their performance scaled with Moore's law, whereas competing methods did not

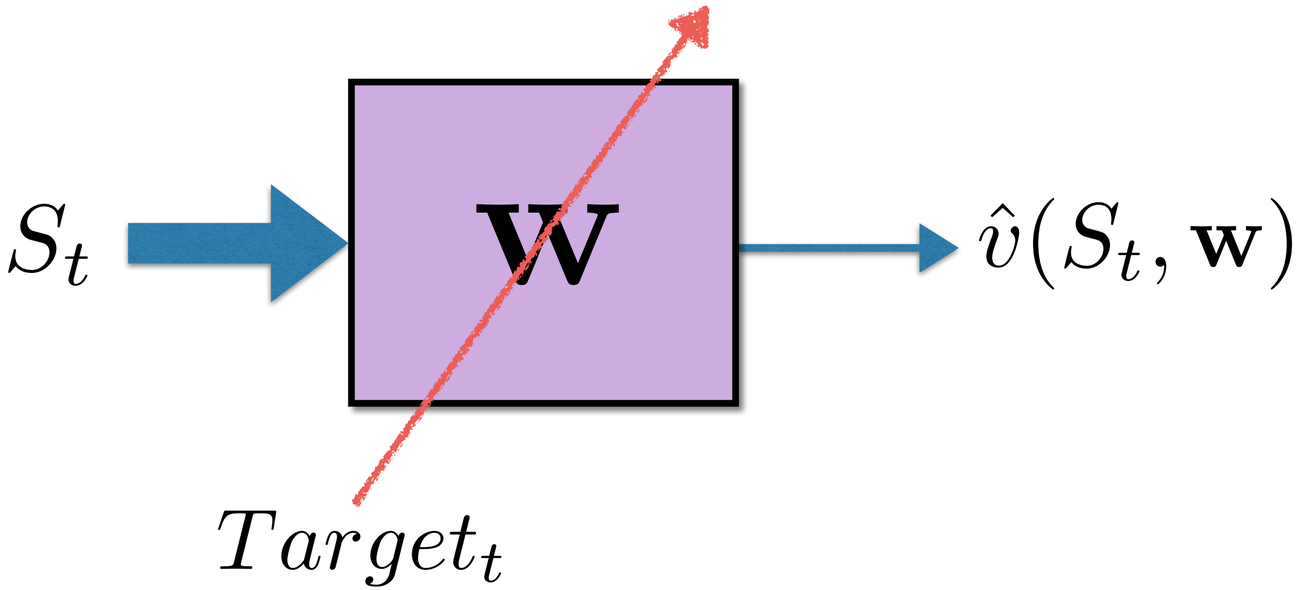
Deep learning

≡ multi-layer neural networks with many layers



- Each line has a learned connection weight
- Each node combines its weighted inputs, then applies a nonlinear transformation
- For each image, the network produces class labels as output, and true class labels are provided by people (supervised learning)
- Then each weight is incremented so as to reduce the squared error (stochastic gradient descent, backpropagation)

Value function approximation (VFA) replaces the table with a general parameterized form



Stochastic Gradient Descent (SGD) is the idea behind most approximate learning

General SGD: $\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} Error_t^2$

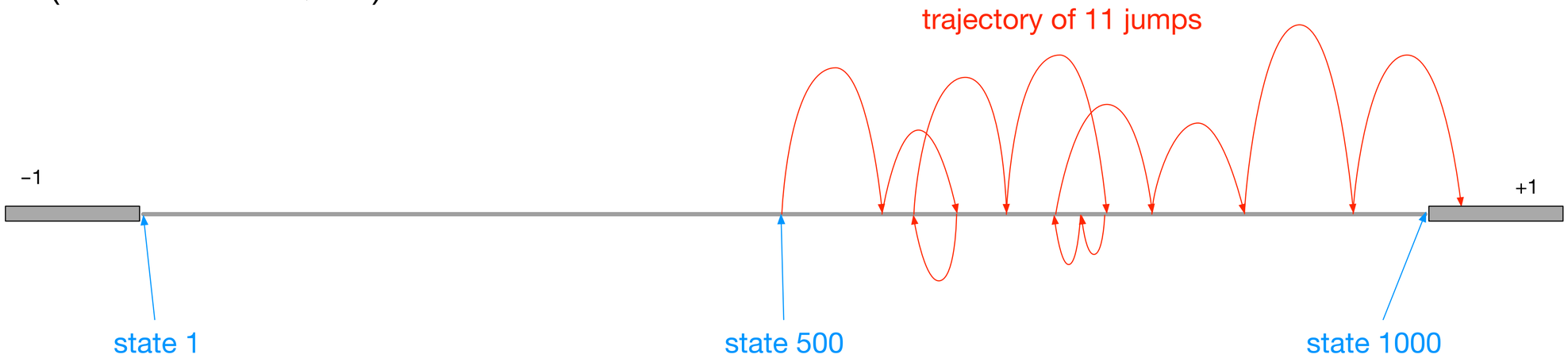
Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$

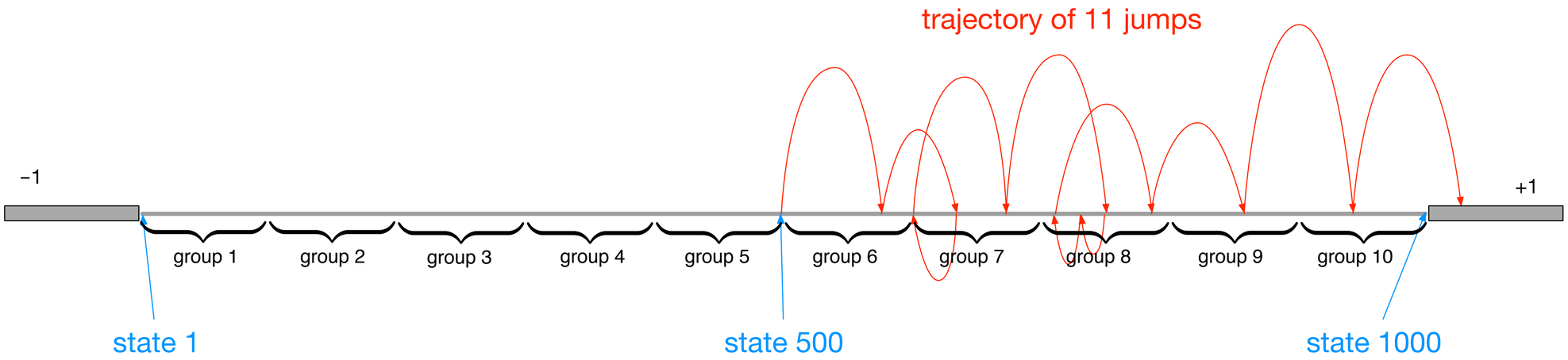
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ as appropriate (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat forever:
    Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
    For $t = 0, 1, \ldots, T-1$:
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

The 1000-state random walk example

- States are numbered 1 to 1000
- Walks start in the near middle, at state 500 $S_0 = 500$
- At each step, *jump* to one of the 100 states to the right, or to one of the 100 states to the left $S_1 \in \{400..499\} \cup \{501..600\}$
- If the jump goes beyond 1 or 1000, terminates with a reward of −1 or +1 (otherwise R_t=0)



State aggregation into 10 groups of 100



The whole value function over 1000 states will be approximated with 10 numbers!

State aggregation is the simplest kind of VFA

- States are partitioned into disjoint subsets (groups)
- One component of $\mathbf{w}$ is allocated to each group

$$\hat{v}(s, \mathbf{w}) \doteq w_{group(s)}$$

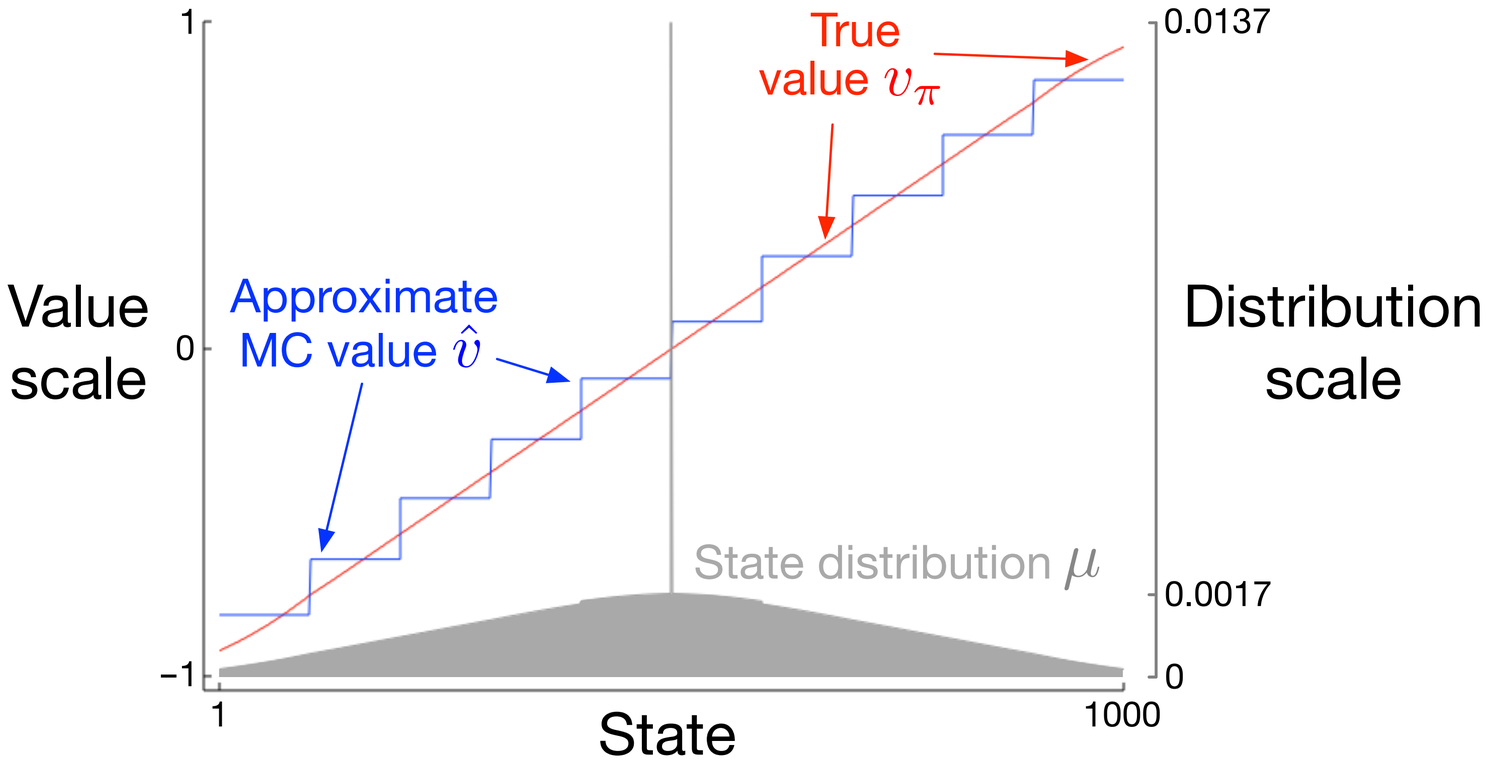
$$\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w}) \doteq [0, 0, \ldots, 0, 1, 0, 0, \ldots, 0]$$

Recall: $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[Target_t - \hat{v}(S_t, \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$

Gradient MC works well on the 1000-state random walk using *state aggregation*

- 10 groups of 100 states
- after 100,000 episodes
- α = 2 x 10^{-5}

- state distribution affects accuracy



A natural objective in VFA
is to minimize the Mean Square Value Error

$$\text{MSVE}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \Big[v_\pi(s) - \hat{v}(s, \mathbf{w}) \Big]^2$$

where $\mu(s)$ is the fraction of time steps spent in state s

True SGD will converge to a local minimum of the error objective

In *linear* VFA, there is only one minimum: local=global

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

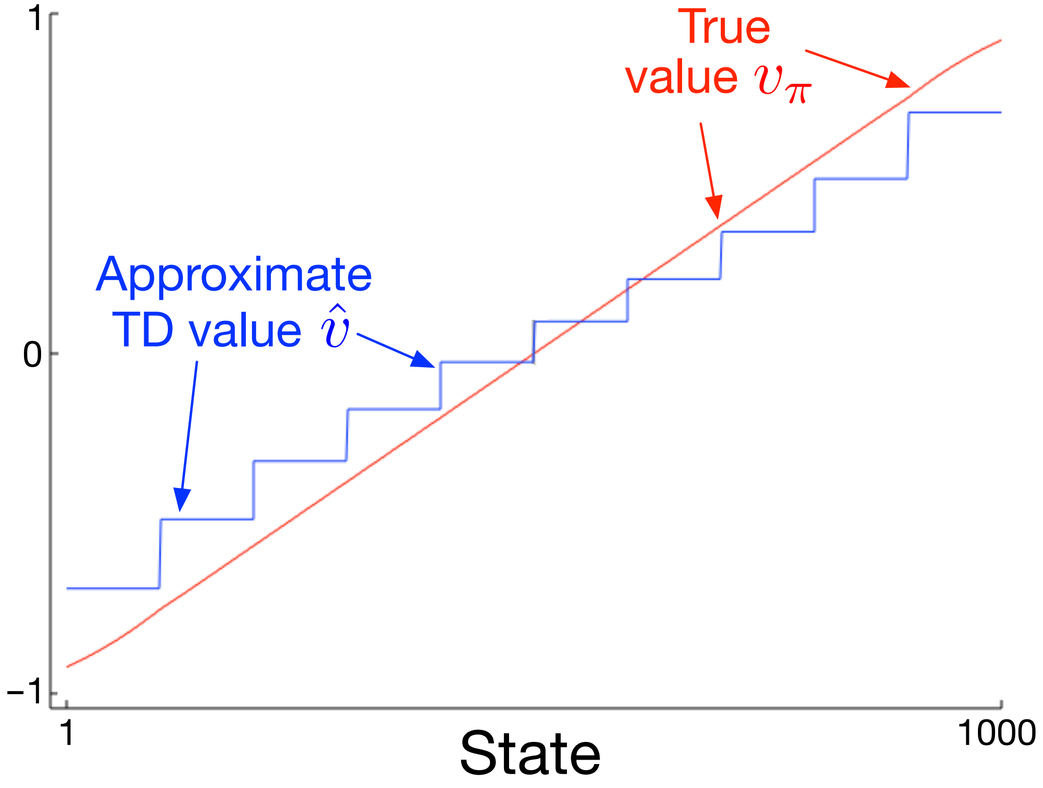
Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose $A \sim \pi(\cdot|S)$
        Take action A, observe R, S'
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
        $S \leftarrow S'$
    until S' is terminal

Gradient TD is less accurate than MC on the 1000-state random walk using state aggregation

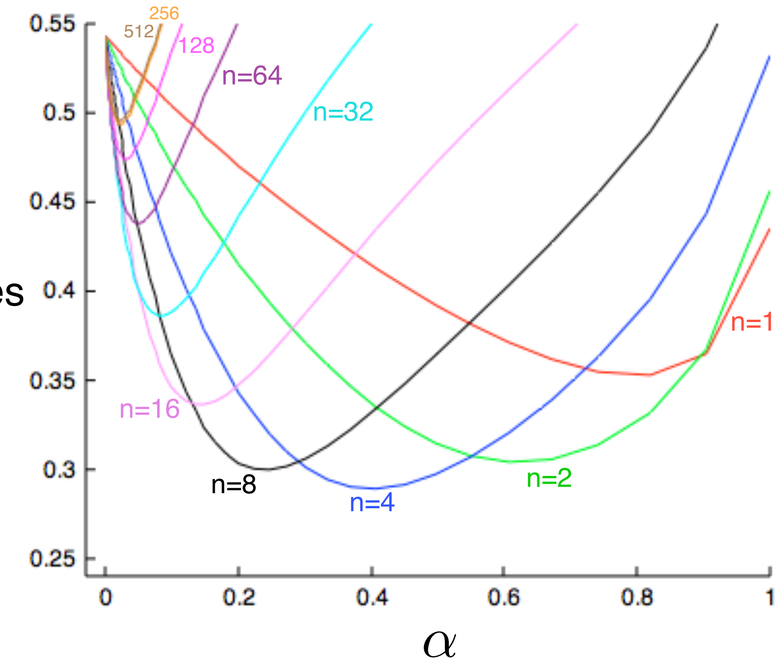
- 10 groups of 100 states
- after 100,000 episodes
- α = 2 x 10^{-5}

Relative values are still pretty accurate



Bootstrapping still greatly speeds learning

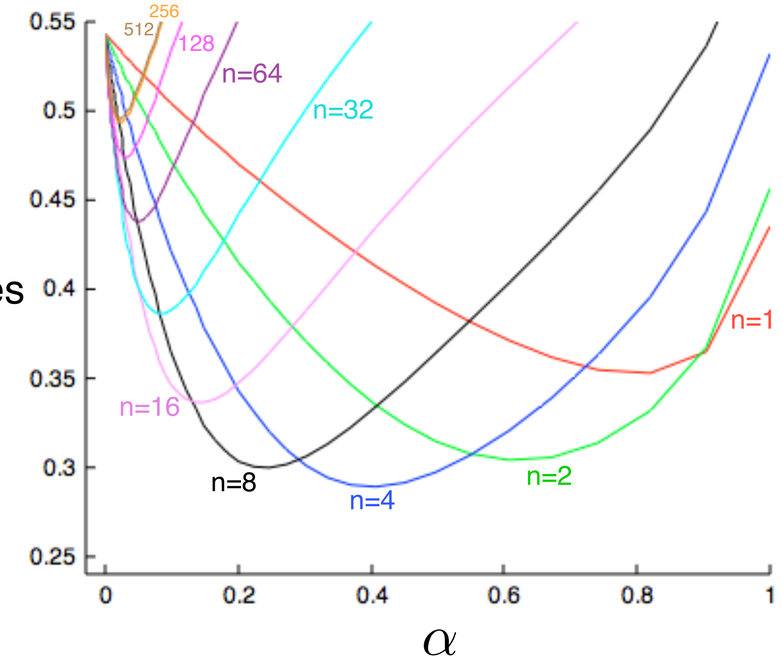
Average
RMS error
over 1000 states
and first 10
episodes



1000 states aggregated
into 20 groups of 50

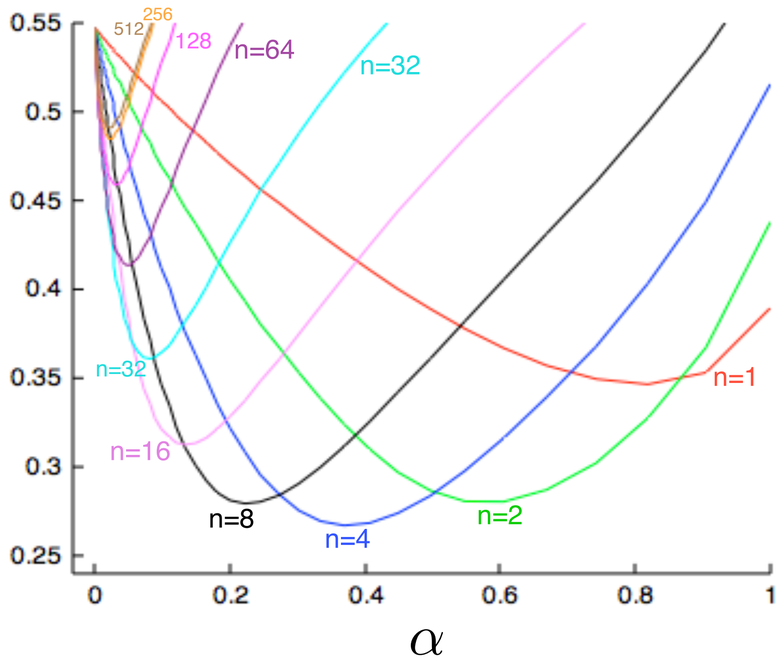
Bootstrapping still greatly speeds learning very much like the tabular 19-state walk

Average RMS error over 1000 states and first 10 episodes



1000 states aggregated into 20 groups of 50

Average RMS error over 19 states and first 10 episodes



19 states tabular (from Chapter 7)

TD converges to the TD fixedpoint, $\boldsymbol{\theta}_{TD}$, a biased but interesting answer

TD(0) update:

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

Fixedpoint analysis:

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

In expectation:

$$\mathbb{E}[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t),$$

where

$$\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d \quad \text{and} \quad \mathbf{A} \doteq \mathbb{E}\Big[\mathbf{x}_t\big(\mathbf{x}_t - \gamma \mathbf{x}_{t+1}\big)^\top\Big] \in \mathbb{R}^d \times \mathbb{R}^d$$

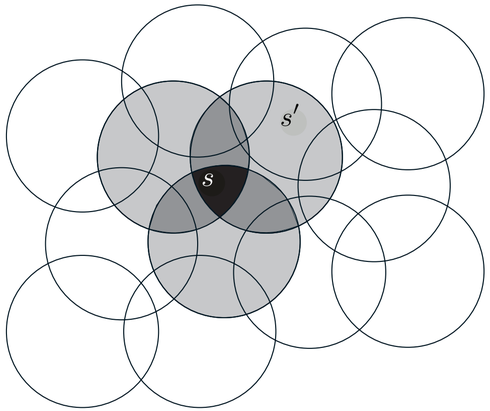
Guarantee:

$$\mathrm{MSVE}(\mathbf{w}_{TD}) \leq \frac{1}{1-\gamma} \min_{\mathbf{w}} \mathrm{MSVE}(\mathbf{w})$$

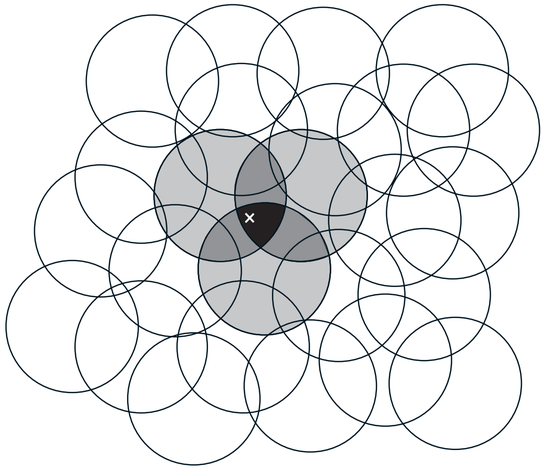
Constructing features, for linear FA

- All fast learning is linear learning
  - The original perception
  - The Least-Mean-Square (LMS) algorithm
  - Support Vector Machines (SVMs)
- Even in deep learning (artificial neural networks), learning is linear in the critical last layer

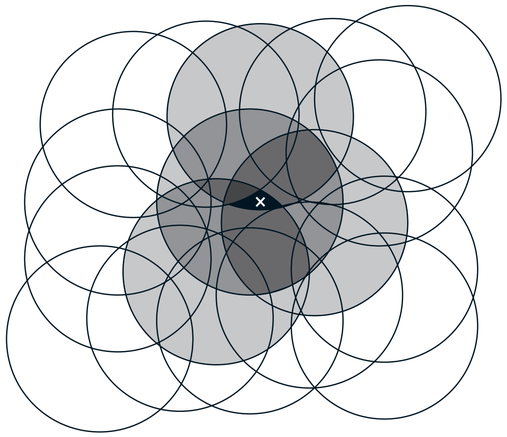
With binary features, a continuous state space can be coarsely coded, adding *generalization*



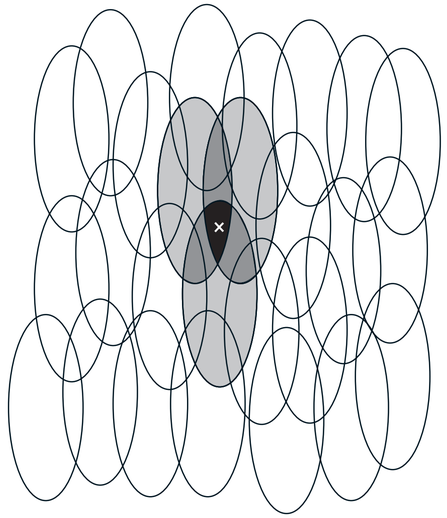
Idea



a) Narrow generalization



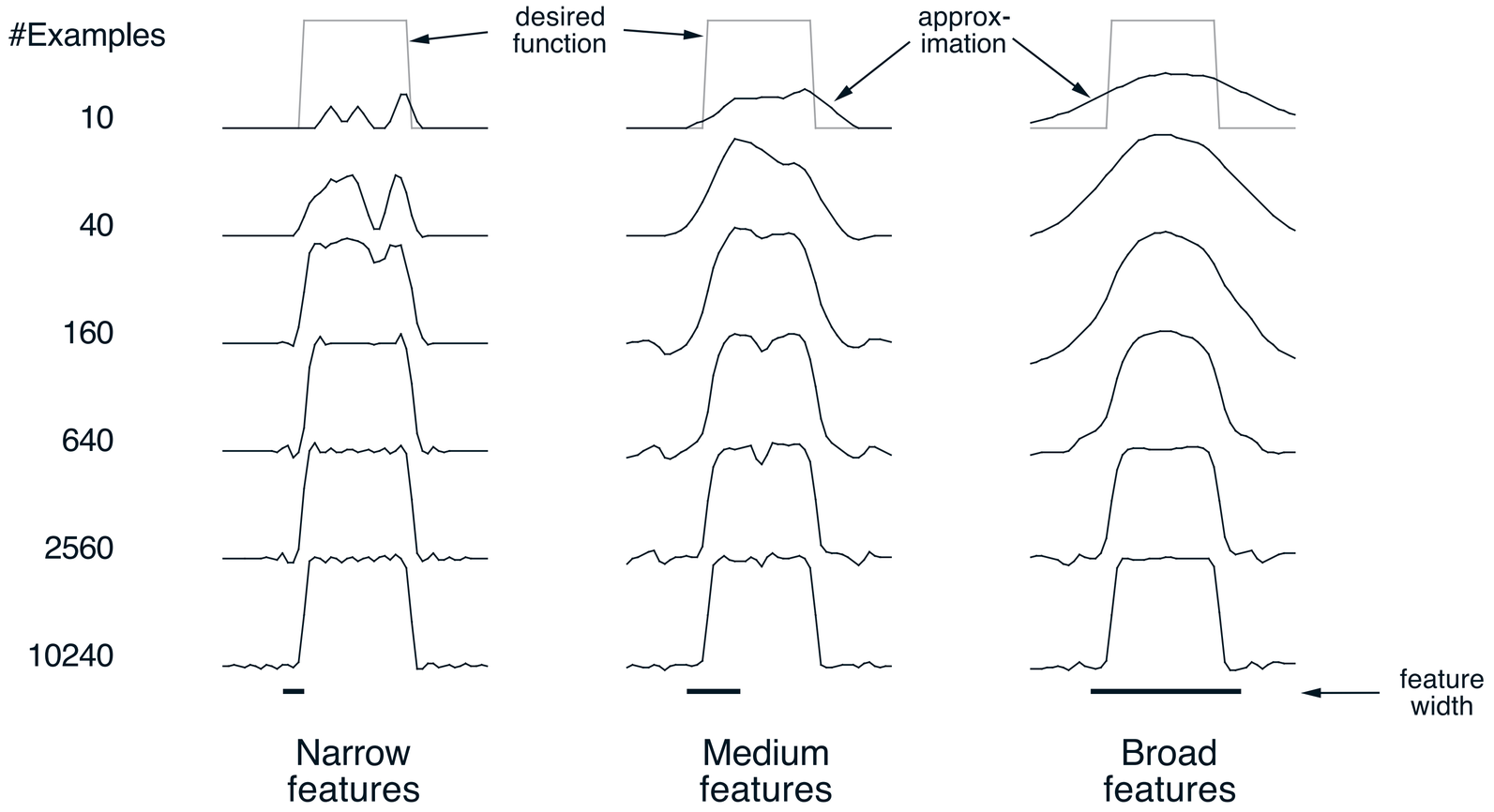
b) Broad generalization



c) Asymmetric generalization

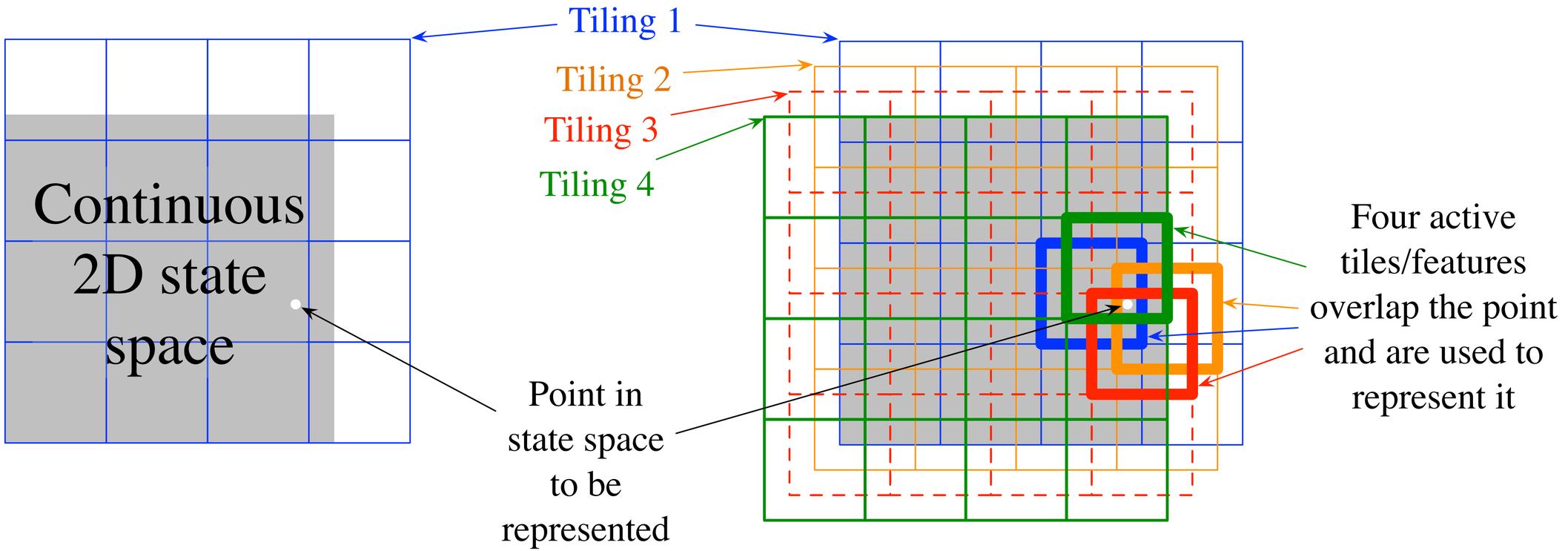
The width of the receptive fields determines breadth of generalization

1D example, supervised training



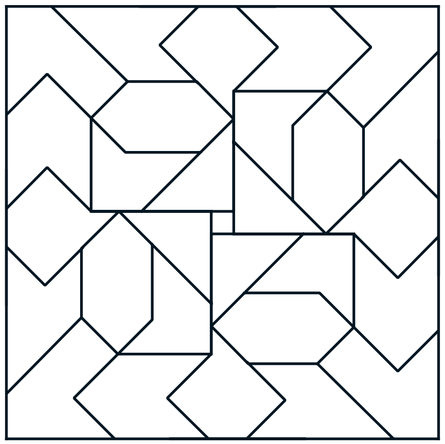
Tile coding is coarse coding for digital computers, with rectangular receptive fields, controlled overlap

2D example

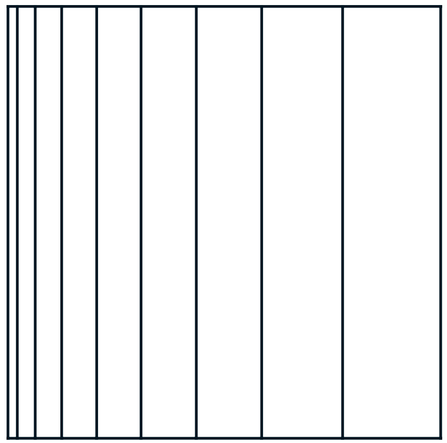


Nevertheless, tile coding is very flexible

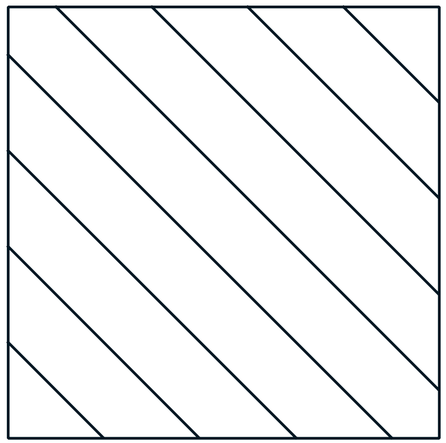
Non-traditional tilings:



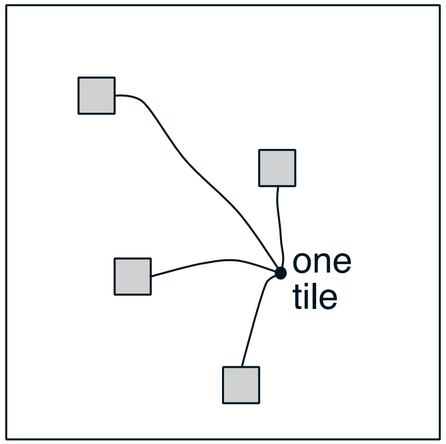
a) Irregular



b) Log stripes

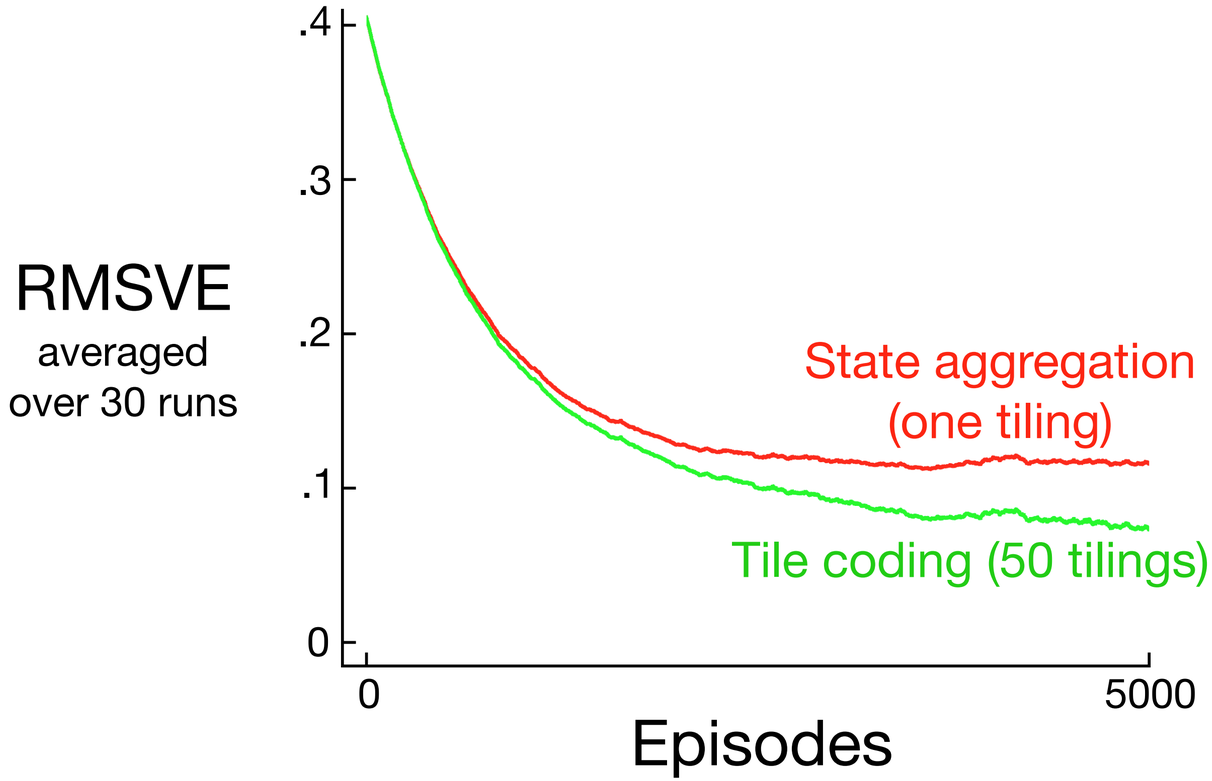


c) Diagonal stripes



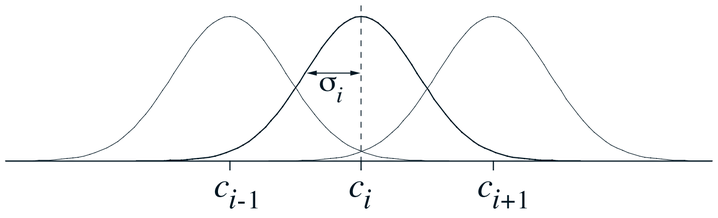
Hashing

Tile coding works better than state aggregation on the 1000-state random walk

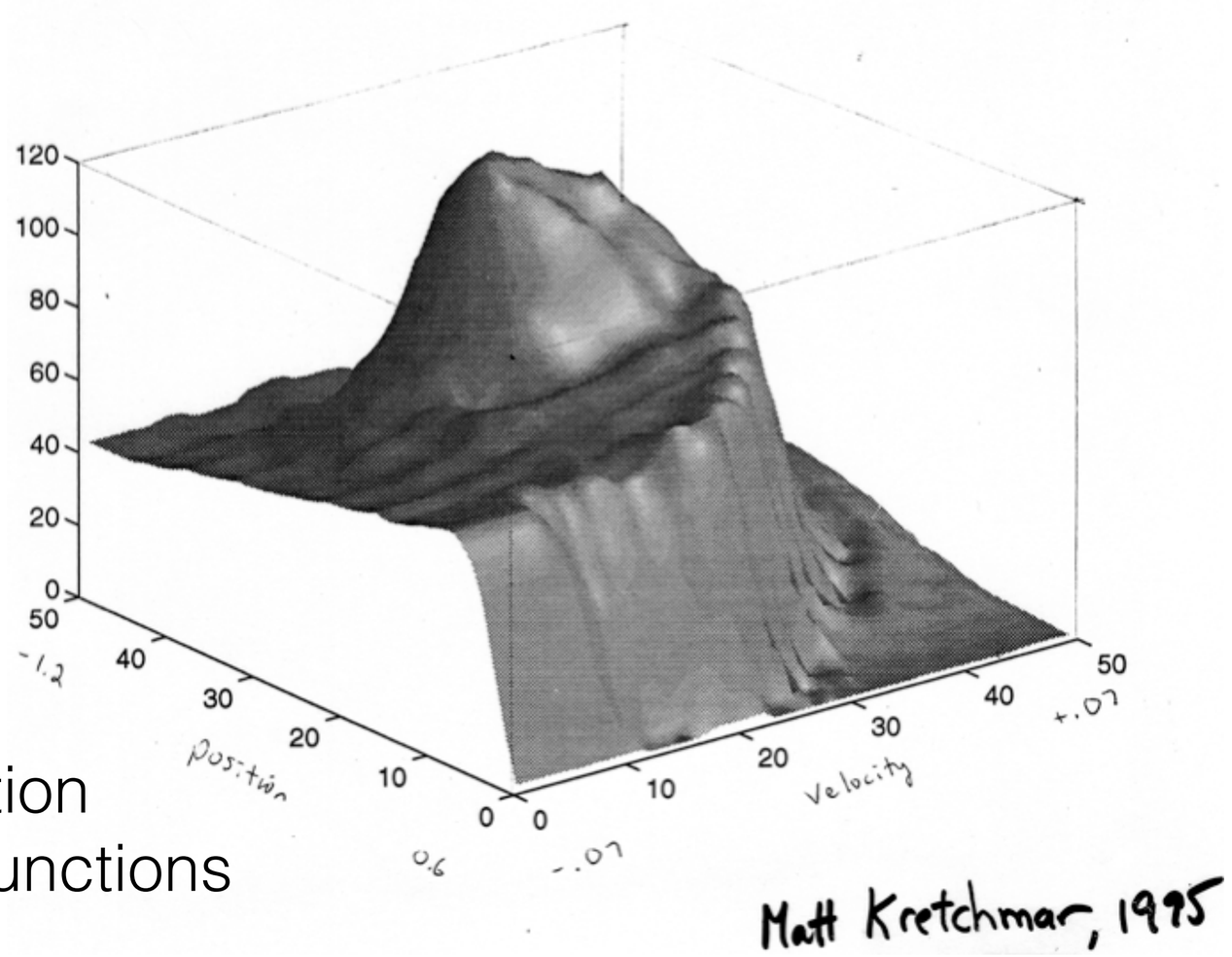


- groups/tiles of 200 states
- α set so that initial learning rate is the same for both methods

Smooth-edged receptive fields are little different but increase computational complexity



1D radial basis functions



A 2D approx. value function
learned with 2D radial basis functions

Conclusions

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