Reinforcement Learning

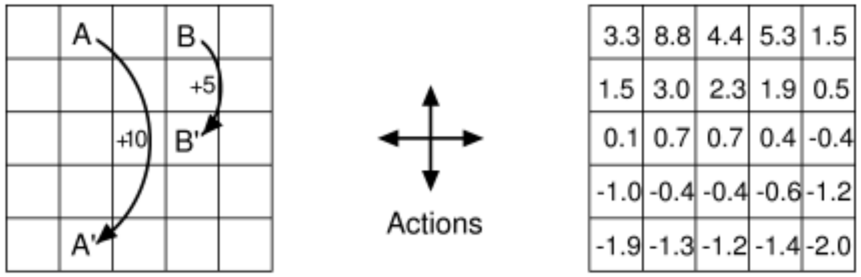
Generalization and Function Approximation

Subramanian Ramamoorthy
School of Informatics

28 February, 2017

Core RL Task: Estimate Value Function

- We are interested in determining V^{π} from experience generated using a policy, π.
- So far, we have considered situations wherein V^{π} is represented in a table.



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

- We would now like to represent V^{π} using a parameterized functional form.

Need for Generalization

- Large state/action spaces
- Continuous valued states and actions
- Most states may not be experienced exactly before
- Many considerations:
  - Memory
  - Time
  - Data

How can experience with a small part of state space be used to produce good behaviour over large part of state space?

Function Approximation

- Use a weight vector θ to parameterize the functional form.
- V^{π} is approximated by another function $V(s, \theta)$
- This function $V(s, \theta)$ could be a linear function in features of the state s
  - θ would then be the feature weights
- The function could also be computed by a neural network
  - θ would be the vector of connection weights in all layers
  - More expressive and could capture many function forms
- Another example is to compute V with a decision tree
  - θ would be numbers defining split points and leaf values

Feature Vectors

$$\vec{\phi}_s = \begin{pmatrix} \text{redness} \\ \text{greenness} \\ \text{roundness} \\ \text{starness} \\ \text{size} \end{pmatrix} = \begin{pmatrix} 25 \\ 3 \\ 2 \\ 15 \\ 25 \end{pmatrix}$$

- "Redness" = say closeness to 111111110000000000000000 (RGB, R=255, G=0, B=0)
- "Roundness" = say distance of points from enclosing circle
- "Starness" = say some combination of number of points, template matching to a star shape, high spatial frequency components of boundary

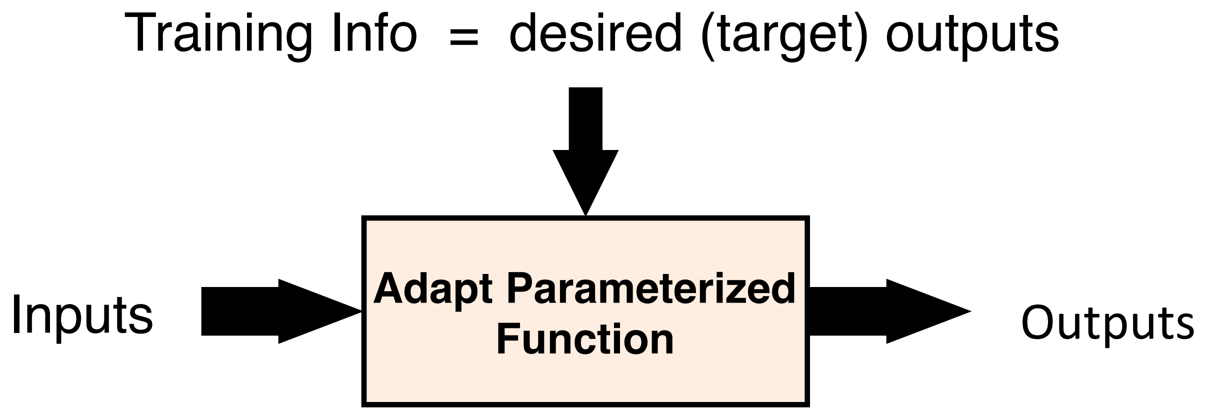
This gives you a summary of state,
e.g., state ⇔ weighted linear combination of features

Issues with Function Approximation

- Number of weights (number of components of θ) is much less than the number of states $(n \ll |\mathcal{S}|)$
- Changing any component of the weight vector will have an effect on more than one state at a time
  - In contrast, with a tabular representation, backups were computed state by state independently

- This is generalization
  - Potentially more powerful
  - May need to be managed with care

Supervised Learning Approach


Training Info = desired (target) outputs
Inputs
Adapt Parameterized Function
Outputs

Training example = {input, target output}

Error = $\mathcal{L}$(target output − actual output)

Value Prediction - Backups

- All of the value prediction methods we have looked at can be understood as ‘backups’
- Updates to an existing value function that shift its value at particular states towards ‘backed-up value’
- Function mapping $V^{\pi}(s)$ to a goal value towards which $V^{\pi}(s)$ is then shifted.

- In the case of Monte Carlo prediction, goal value is return R_t
- The goal value in the case of TD(0) is $r_{t+1} + \gamma V^{\pi}(s_{t+1})$

Using Backups as Training Examples

e.g., the TD(0) backup:

$$V(s_t) \leftarrow V(s_t) + \alpha\left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right]$$

As a training example:

$$\left\{\underbrace{\text{description of } s_t}_{\substack{\text{Input}\\\text{(e.g., feature vector)}}},\quad \underbrace{r_{t+1} + \gamma V(s_{t+1})}_{\text{Target Output}}\right\}$$

What Kind of Function Approximation?

- Neural networks, decision trees, multivariate regression ...
- Use of statistical clustering for state aggregation
- Can swap in/out your favourite function approximation method as long as they can deal with:
  - learning while interacting, *online* (not just batch methods)
  - non-stationarity induced by policy changes
    - As learning proceeds, target function may change (e.g., in the case of TD(0))
- So, combining gradient descent methods with RL requires care (especially regarding convergence)

Prediction Objective

- In the tabular case, updates were decoupled. Eventually, we would arrive at the correct V^{π} for all states.
- When approximating value function, we need a measure of quality of prediction
  - May not be possible to get exactly correct prediction in all states
  - Because we have more states than weights
- What else could we do?
  - Decide which states we care about the most
  - Weight error with a distribution $P(s)$

Performance Measures

- Many are applicable but…
- a common and simple one is the mean-squared error (MSE) over a distribution *P* :

$$MSE(\theta_t) = \sum_{s \in S} \boxed{P(s)} \left[V^{\pi}(s) - V_t(s) \right]^2$$

Value obtained, e.g., by backup updates *Approximated 'surface'*

- Why minimize MSE?

  Real objective is of course a better policy, but unclear how else to get at that other than value prediction.

- Why *P* ?

  Consider, e.g., the case where *P* is always the distribution of states at which backups are done.

Choosing $P(s)$

- One natural definition: fraction of time spent in s while following the target policy π.
- In continuous tasks, this is the the stationary distribution under π
- In episodic tasks, this depends on how the initial states of episodes are drawn

- The on-policy distribution: the distribution created while following the policy being evaluated. Stronger results are available for this distribution.

Linear Methods

Represent states as feature vectors;

for each $s \in S$:

$\vec{\phi}_s = \left(\phi_1, \phi_2, \ldots, \phi_n\right)_s^T$

$$V_t(s) = \vec{\theta}_t^T \vec{\phi}_s = \sum_{i=1}^{n} (\theta_i)_t (\phi_i)_s$$

$$\nabla_{\vec{\theta}} V_t(s) = \quad ?$$

What are we learning? From what?

Update the weight vector:

$$\vec{\theta}_t = \left(\theta_1, \theta_2, \ldots, \theta_n\right)_t^T$$

Assume V_t is a (sufficiently smooth) differentiable function of $\vec{\theta}_t$, for all $s \in S$.

Assume, for now, training examples of this form:

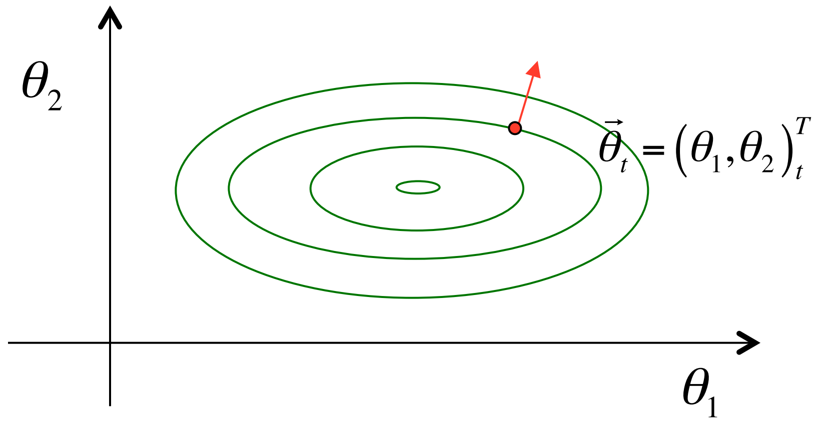
$$\left\{\text{description of } s_t,\ V^\pi(s_t)\right\}$$

Concept: Gradient Descent

Let f be any function of the parameter space.

Its gradient at any point $\vec{\theta}_t$ in this space is:

$$\nabla_{\vec{\theta}} f(\vec{\theta}_t) = \left(\frac{\partial f(\vec{\theta}_t)}{\partial \theta_1}, \frac{\partial f(\vec{\theta}_t)}{\partial \theta_2}, \ldots, \frac{\partial f(\vec{\theta}_t)}{\partial \theta_n} \right)^T$$



Iteratively move down the gradient:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \alpha \nabla_{\vec{\theta}} f(\vec{\theta}_t)$$

Gradient Descent for Weights – Basic Setup

For the MSE given earlier and using the chain rule:

$$
\begin{aligned}
\vec{\theta}_{t+1} &= \vec{\theta}_t - \frac{1}{2}\alpha \nabla_{\vec{\theta}} MSE(\vec{\theta}_t) \\
&= \vec{\theta}_t - \frac{1}{2}\alpha \nabla_{\vec{\theta}} \sum_{s \in S} P(s)\left[V^{\pi}(s) - V_t(s)\right]^2 \\
&= \vec{\theta}_t + \alpha \sum_{s \in S} P(s)\left[V^{\pi}(s) - V_t(s)\right] \nabla_{\vec{\theta}} V_t(s)
\end{aligned}
$$

Gradient Computation

In practice, could just use the **sample gradient** (as we are acting based on the same distribution, *P(s)*:

$$\vec{\theta}_{t+1} = \vec{\theta}_t - \frac{1}{2}\alpha\nabla_{\vec{\theta}}\left[V^{\pi}(s_t) - V_t(s_t)\right]^2$$
$$= \vec{\theta}_t + \alpha\left[V^{\pi}(s_t) - V_t(s_t)\right]\nabla_{\vec{\theta}}V_t(s_t),$$

Since each sample gradient is an **unbiased estimate** of the true gradient, this converges to a local minimum of the MSE if α decreases appropriately with *t*.

$$E\left[V^{\pi}(s_t) - V_t(s_t)\right]\nabla_{\vec{\theta}}V_t(s_t) = \sum_{s\in S} P(s)\left[V^{\pi}(s) - V_t(s)\right]\nabla_{\vec{\theta}}V_t(s)$$

But We Don't have these Targets

Suppose we just have targets v_t instead:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\left[v_t - V_t(s_t)\right]\nabla_{\vec{\theta}} V_t(s_t)$$

If each v_t is an unbiased estimate of $V^\pi(s_t)$,

i.e., $E\left\{v_t\right\} = V^\pi(s_t)$, then gradient descent converges

to a local minimum (provided α decreases appropriately).

e.g., the Monte Carlo target $v_t = R_t$:

$$\vec{\theta}_{t+1} = \vec{\theta}_t + \alpha\left[R_t - V_t(s_t)\right]\nabla_{\vec{\theta}} V_t(s_t)$$

State aggregation is the simplest kind of Value Function Approximation

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

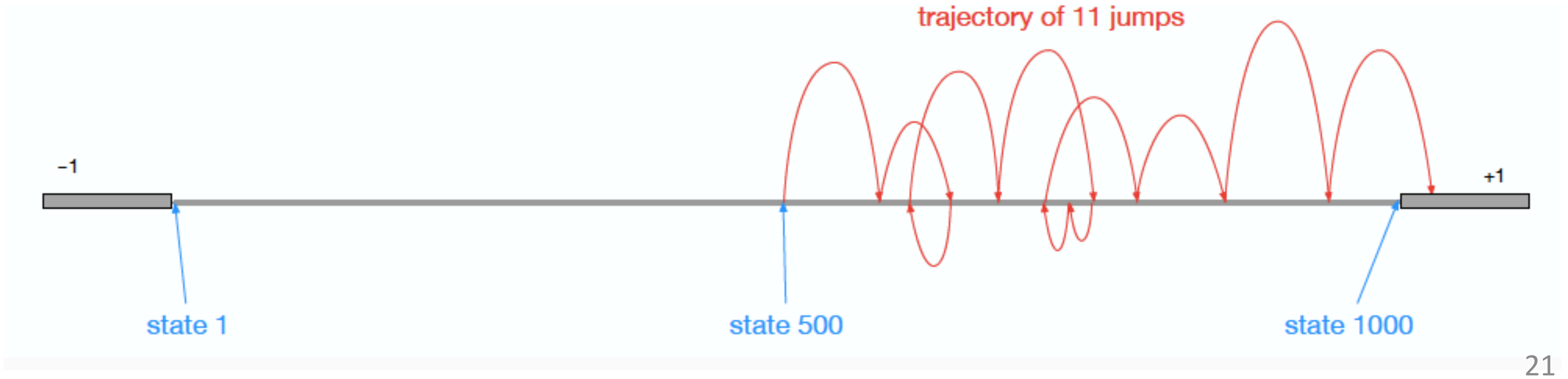
$$V_t(s) = \theta_{group}(s)$$

$$\nabla_\theta V_t(s) = [0, 0, ..., 1, 0, 0, ..., 0]$$

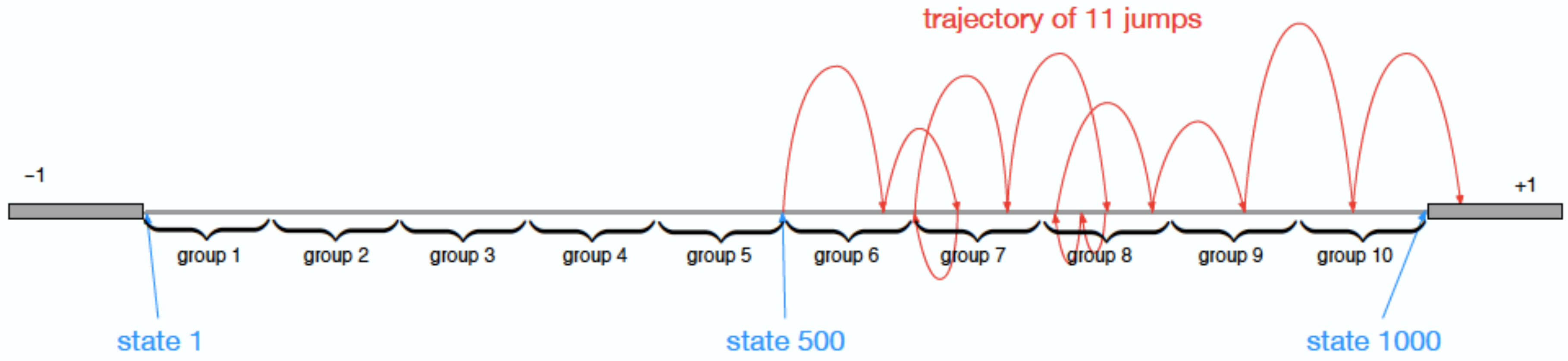
Recall: $\theta \leftarrow \theta + \alpha[Target_t - V_t(s_t)]\nabla_\theta V_t(s_t)$

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

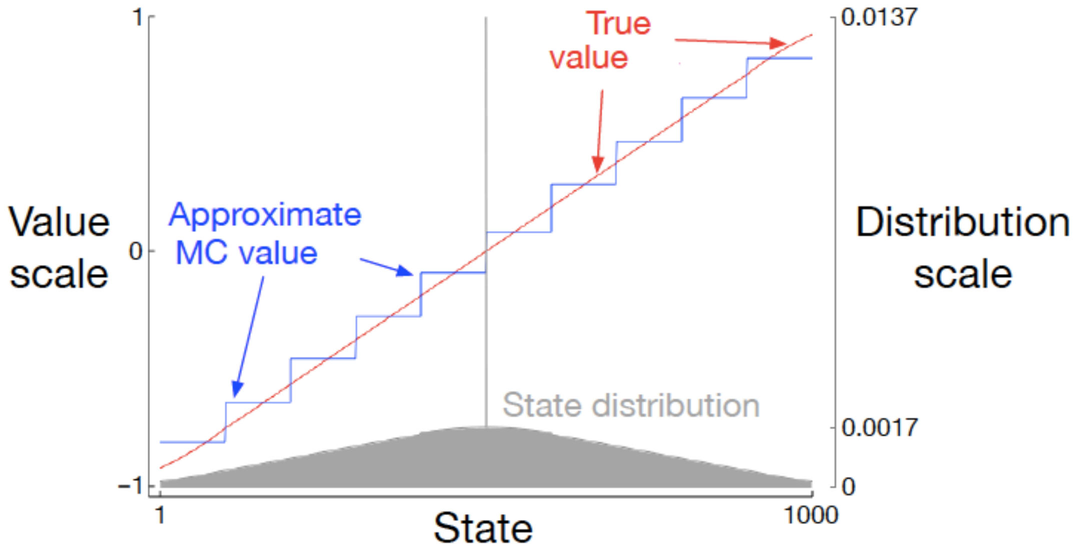


The whole value function over 1000 states will be approximated with 10 numbers!

VF Computed with Gradient MC

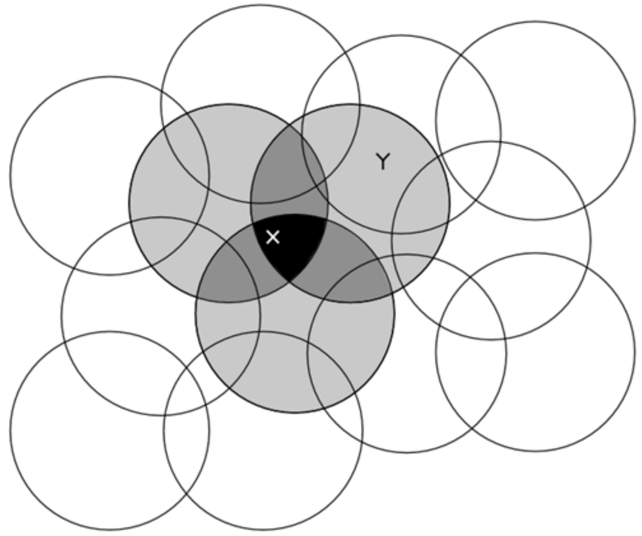
- 10 groups of 100 states
- after 100,000 episodes
- $\alpha = 2 \times 10^{-5}$

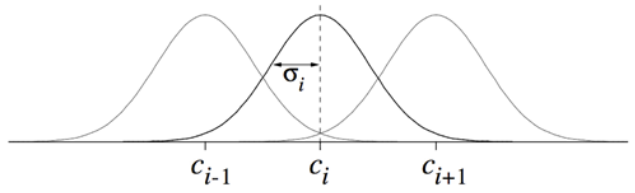
- state distribution affects accuracy

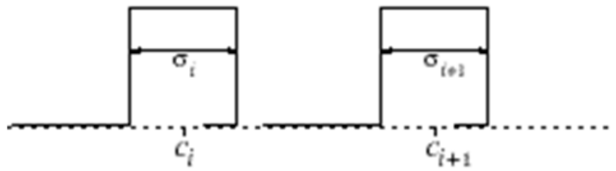


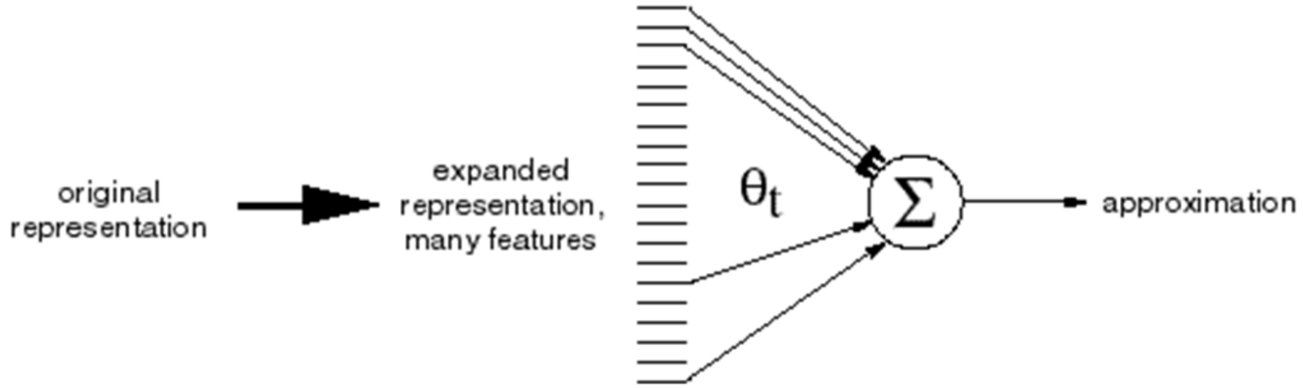
On Basis Functions: Coarse Coding

Many ways to achieve "coding":



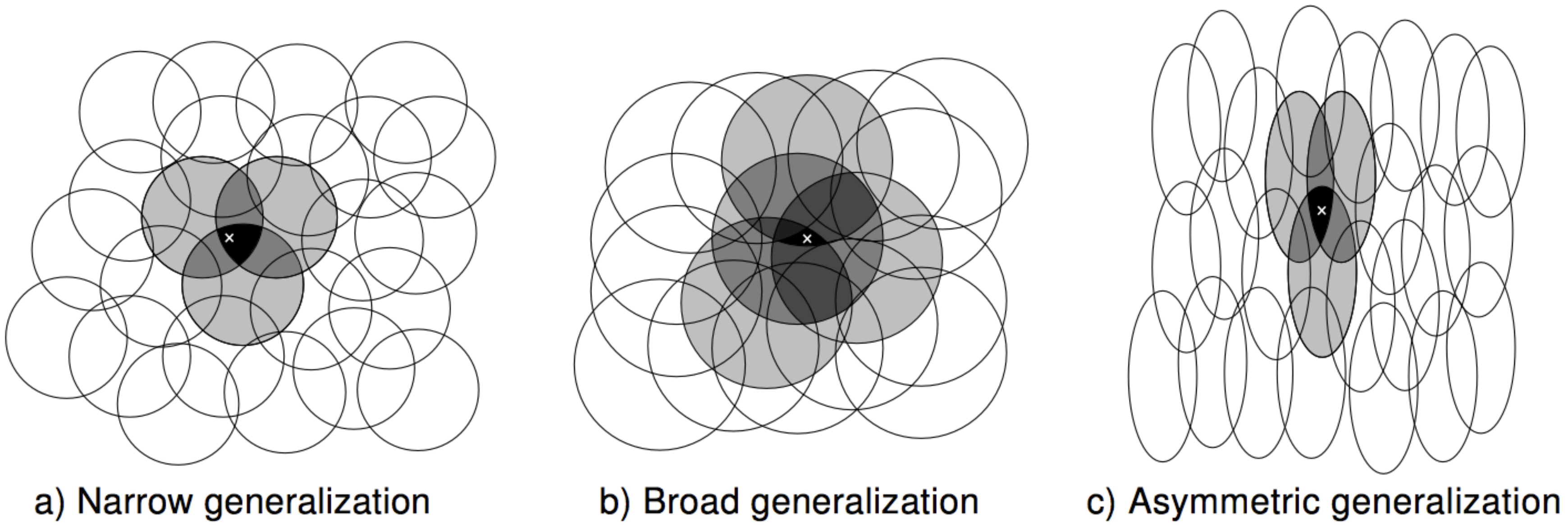




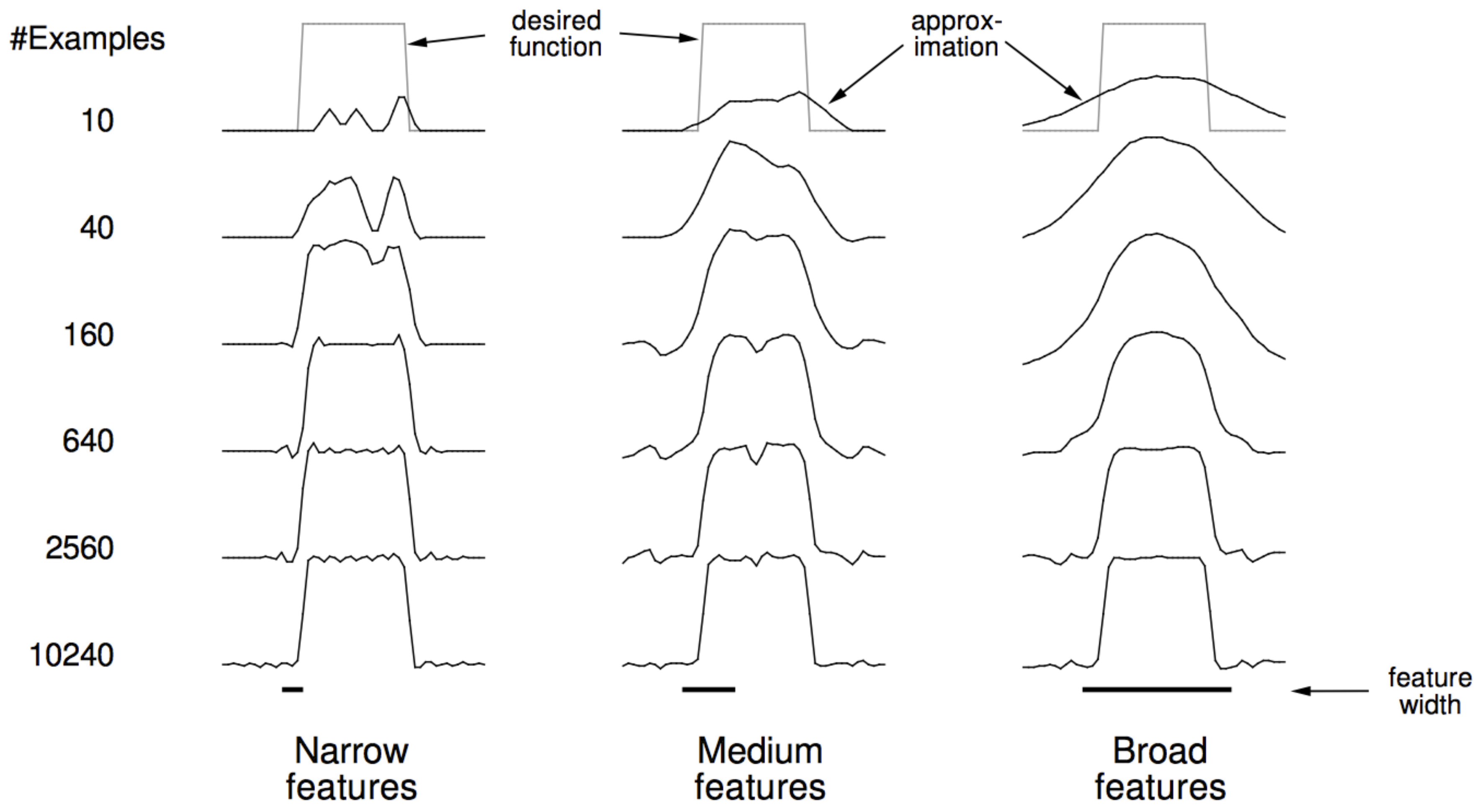


Shaping Generalization in Coarse Coding

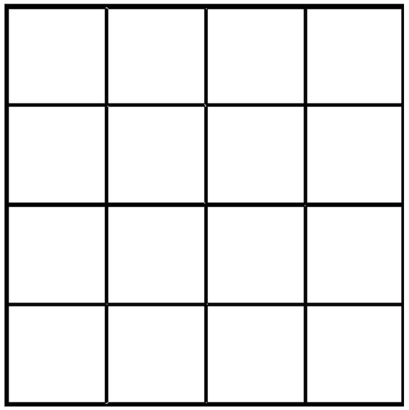
Binary features defined by overlap between receptive fields.



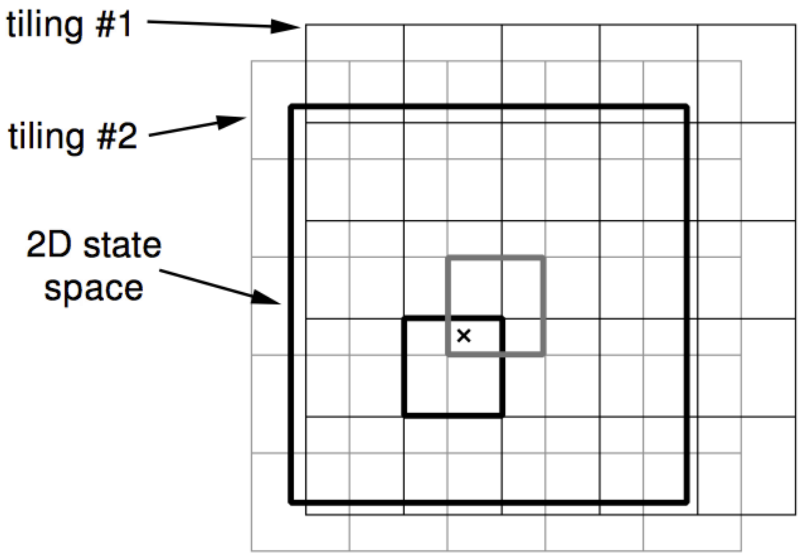
Learning and Coarse Coding



Tile Coding



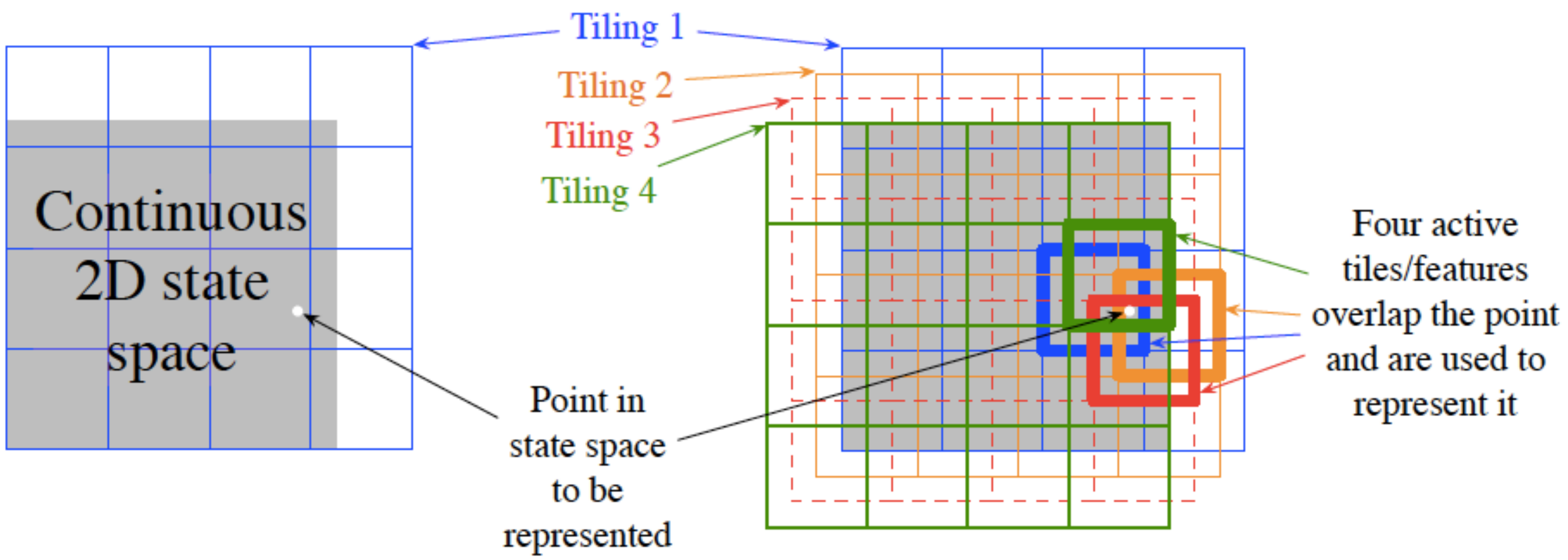
- Binary feature for each tile
- Number of features present at any one time is constant
- Binary features means weighted sum easy to compute
- Easy to compute indices of the features present



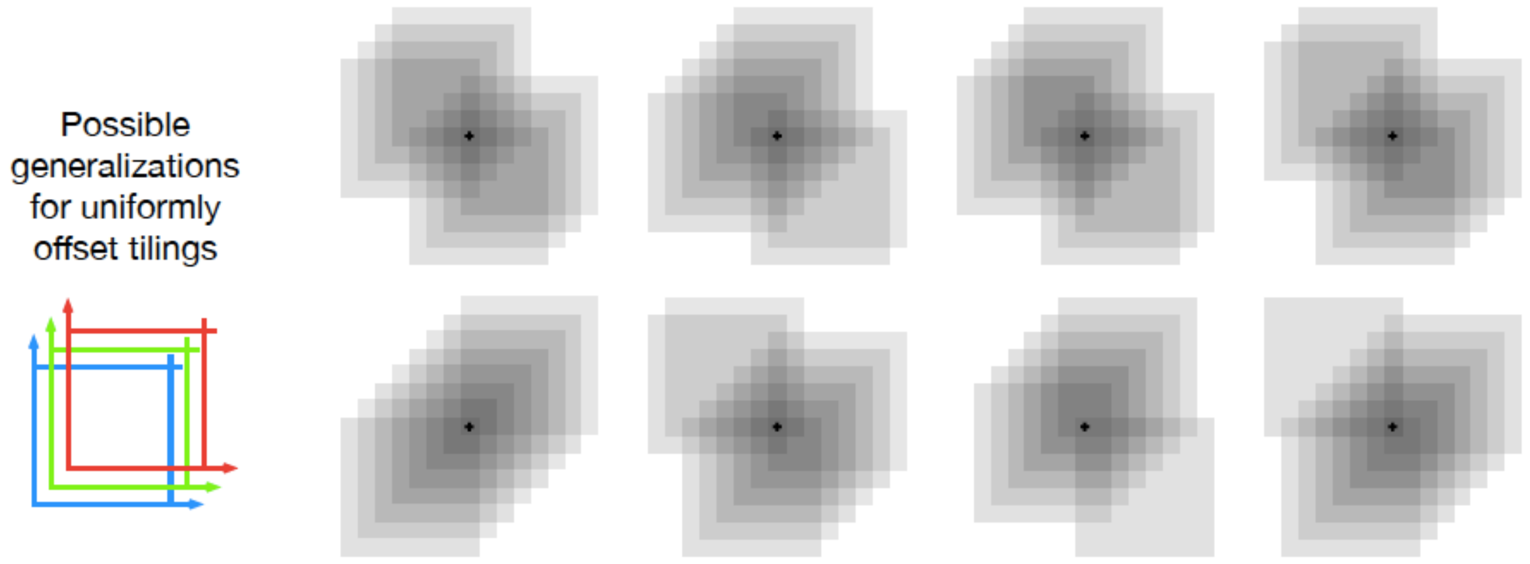
Shape of tiles $\Rightarrow$ Generalization

#Tilings $\Rightarrow$ Resolution of final approximation

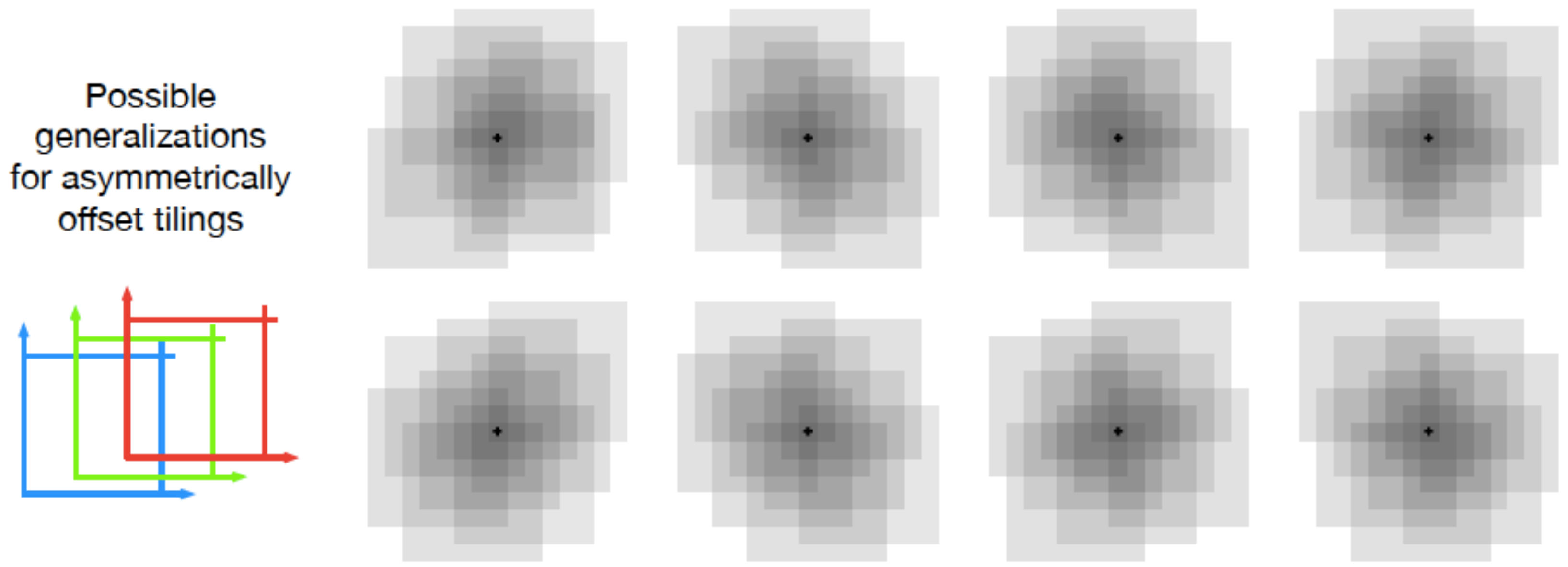
Encoding 2D Space with Many Tiles


Tiling 1
Tiling 2
Tiling 3
Tiling 4
Continuous 2D state space
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it

Generalizing with Uniformly Offset Tiles

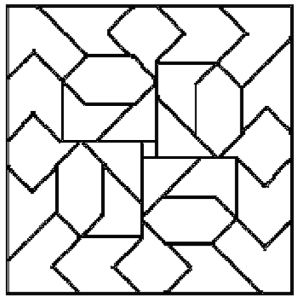


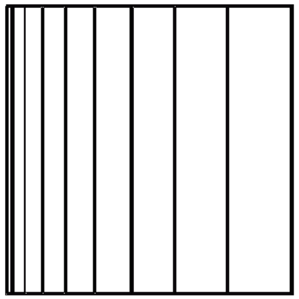
Generalizing with Asymmetrically Offset Tiles

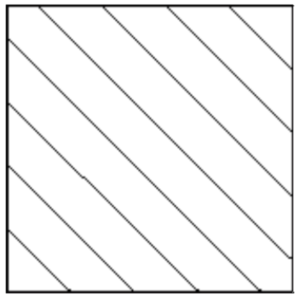


Tile Coding, Contd.

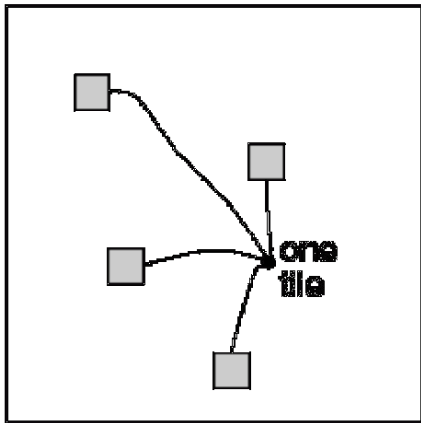
Irregular tilings


a) Irregular


b) Log stripes


c) Diagonal stripes

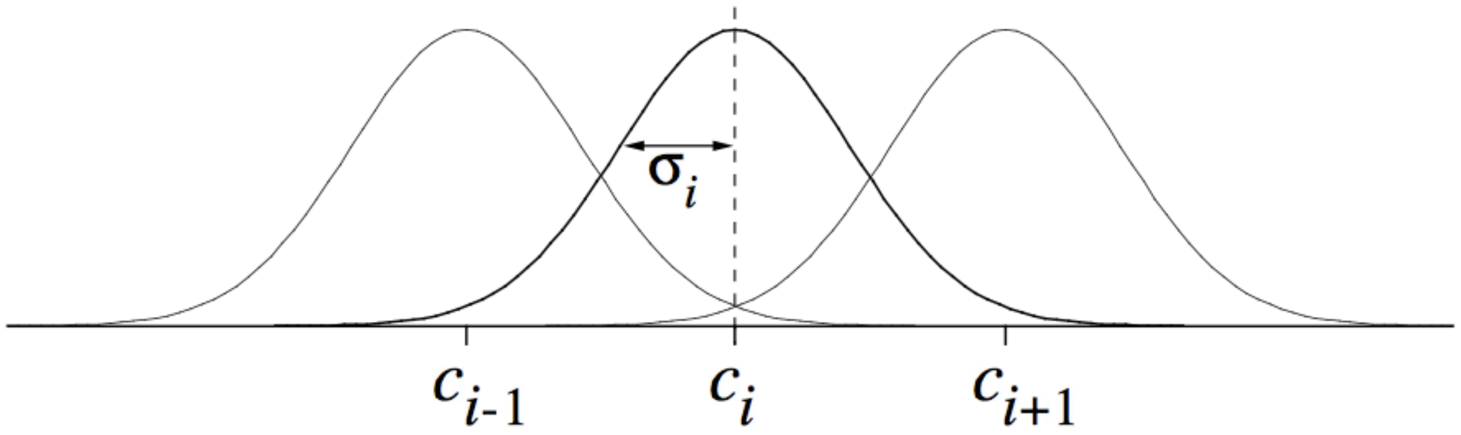
Hashing



Radial Basis Functions (RBFs)

e.g., Gaussians

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



Beating the "Curse of Dimensionality"

- Can you keep the number of features from going up exponentially with the dimension?
- Function complexity, not dimensionality, is the problem.
- Kanerva coding:
  - Select a set of binary prototypes
  - Use Hamming distance as distance measure
  - Dimensionality is no longer a problem, only complexity
- "Lazy learning" schemes:
  - Remember all the data
  - To get new value, find nearest neighbors & interpolate
  - e.g., (nonparametric) locally-weighted regression

Going from Value Prediction to GPI

- So far, we've only discussed policy evaluation where the value function is represented as an approximated function

- In order to extend this to a GPI-like setting,
  1. Firstly, we need to use the action-value functions
  2. Combine that with the policy improvement and action selection steps

Gradient Descent Update for Action-Value Function Prediction

$$\overrightarrow{\theta}_{t+1} = \overrightarrow{\theta}_t + \alpha[v_t - Q_t(s_t, a_t)] \nabla_{\overrightarrow{\theta}_t} Q_t(s_t, a_t)$$

$$\text{e.g., } R_t = r_{t+1} + \gamma Q_t(s_{t+1}, a_{t+1})$$

How to Plug-in Policy Improvement or Action Selection?

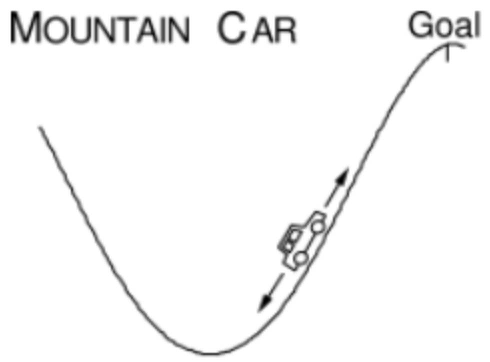
- If spaces are very large, or continuous, this is an active research topic
- For manageable discrete spaces,
  - For each action, a, available at a state, s_t, compute $Q_t(s_t,a)$ and find the greedy action according to it

$$a_t^* = \arg\max_a Q_t(s_t, a)$$

  - Then, one could use this as part of an ε-greedy action selection or as the estimation policy in off-policy methods

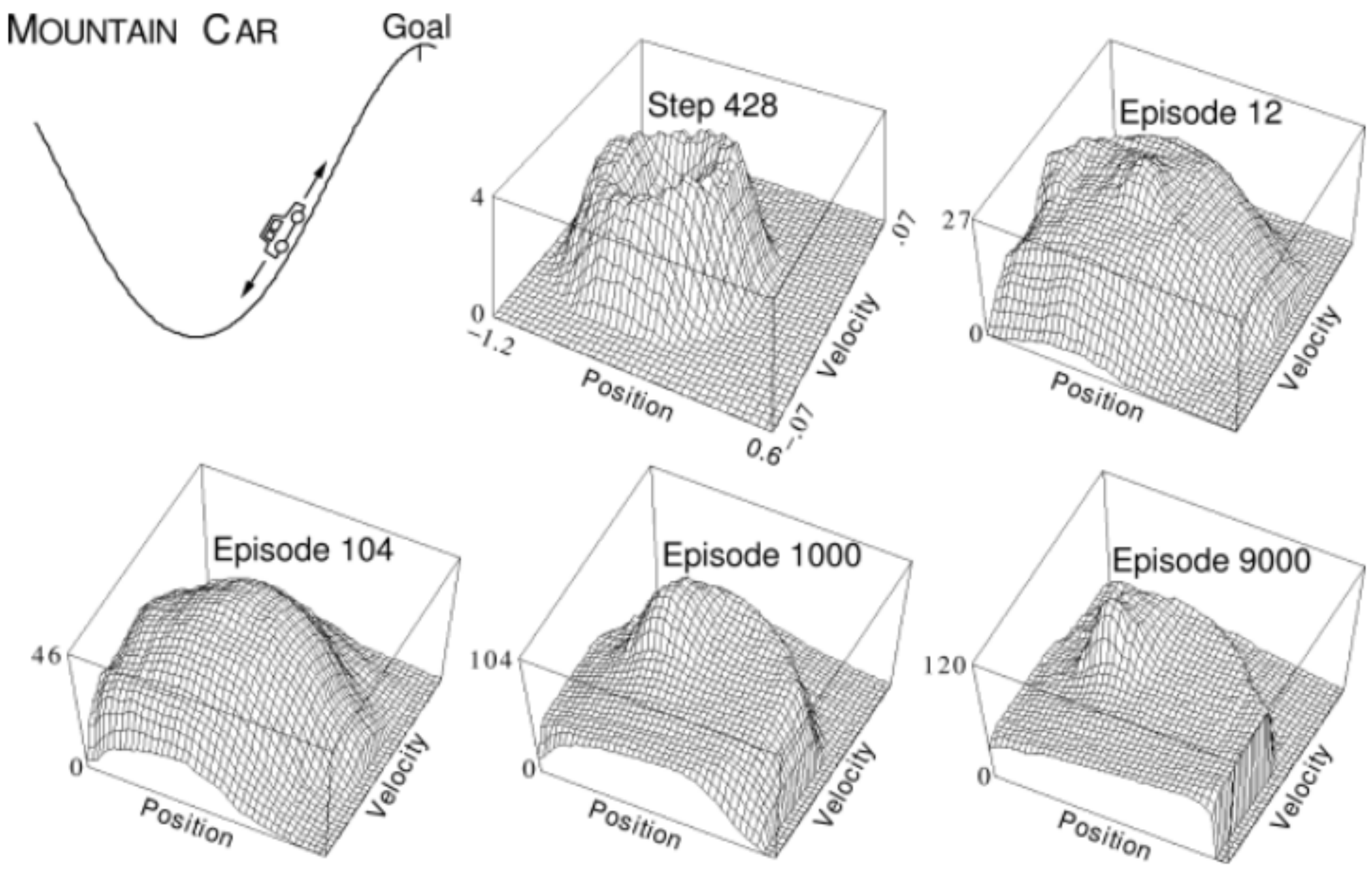
Example: Mountain-car Task

- Drive an underpowered car up a steep mountain road
- Gravity is stronger than engine (like in cart-pole example)
- Example of a continuous control task where system must move away from goal first, then converge to goal
- Reward of -1 until car 'escapes'
- Actions: +τ, -τ, 0

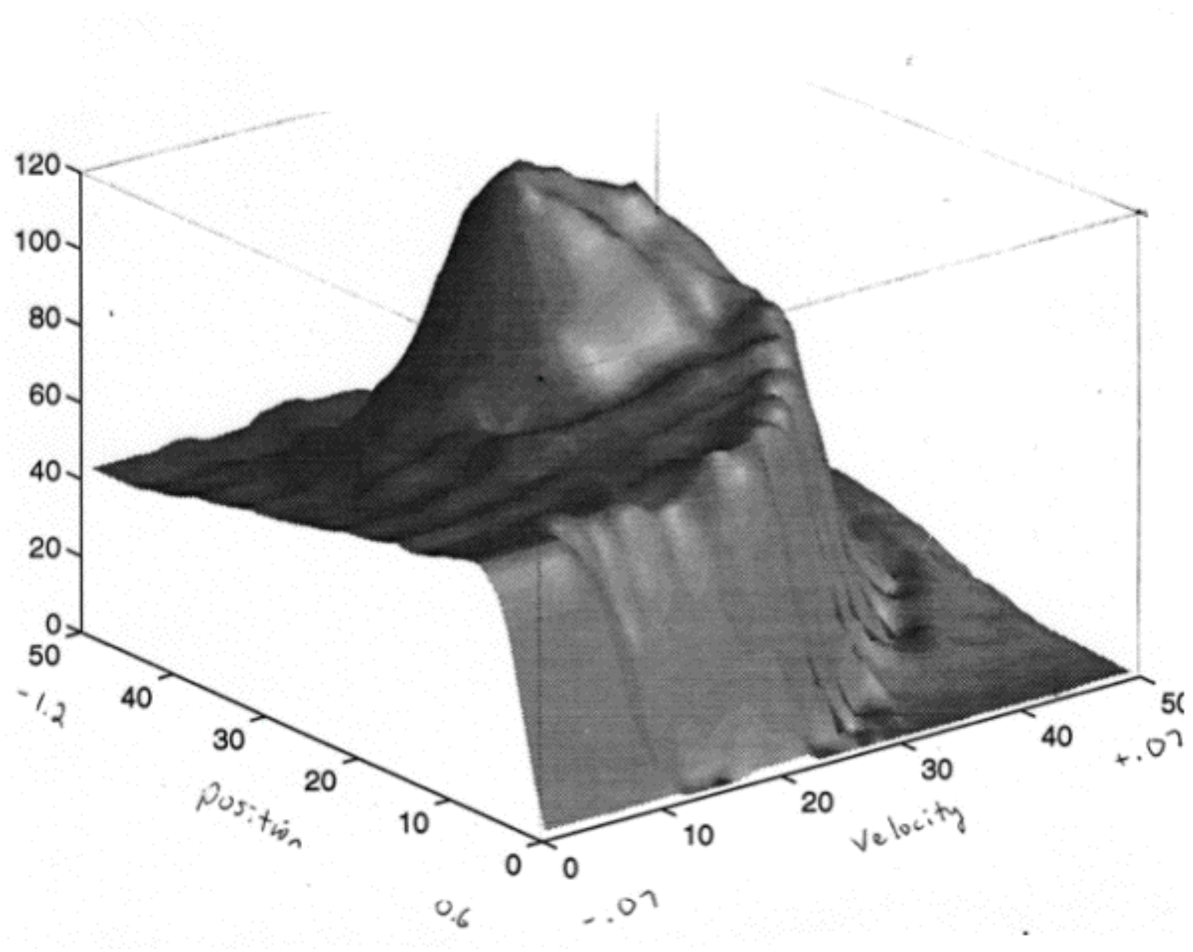


$$x_{t+1} = bound\left[x_t + \dot{x}_{t+1}\right]$$
$$\dot{x}_{t+1} = bound\left[\dot{x}_t + 0.001a_t + -0.0025\cos(3x_t)\right]$$

Mountain-car Example: Cost-to-go Function (SARSA solution)



Mountain Car Solution with RBFs



[Computed by M. Kretchmar]