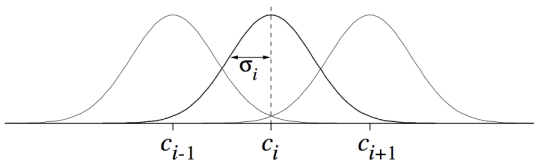
Generalization and Function Approximation

Suggested reading:

Chapter 8 in R. S. Sutton, A. G. Barto: Reinforcement Learning: An Introduction
MIT Press, 1998.

Generalization and Function Approximation



Contents:

Problems with tables

- value functions are represented as a table with one entry for each state or for each state-action pair
- it is limited to tasks with small numbers of states and actions
- the state or action spaces include continuous variables or complex sensations (visual image)

Value Prediction with Function Approximation

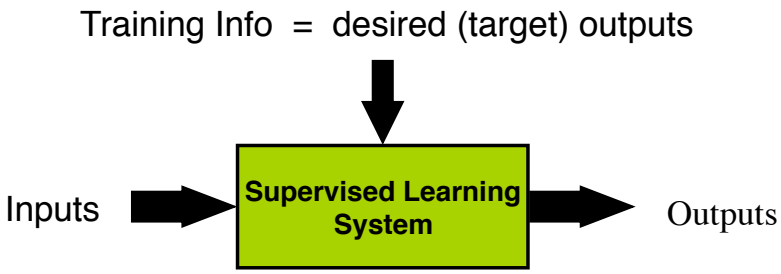
As usual: Policy Evaluation (the prediction problem):
for a given policy π, compute the state-value function V^π

In earlier chapters, value functions were stored in lookup tables.

Here, the value function estimate V_t at time t, is a parameterized functional form with parameter vector $\boldsymbol{\theta}_t$ and only the parameter vector is updated.

For example, V_t might be the function computed by an artificial neural network, with $\boldsymbol{\theta}_t$ the vector of connection weights.

Adapt Supervised Learning Algorithms



Training example = {input, target output}

Error = (target output – actual output)

Performance Measures

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

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

estimate

- *P* is a distribution weighting the errors of different states
- Let us assume that *P* is always the distribution of states from which the training examples are drawn, and thus the one at which backups are done.
- The **on-policy distribution**: the distribution created while following the policy being evaluated (describes the frequency with which states are encountered).

Ideal Goal

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

$$V_t(s) = V(s,\theta_t)$$

$$\theta^* \longrightarrow \qquad MSE(\theta^*) \leq MSE(\theta) \qquad \forall \theta$$

global optimum

complex function approximators may seek
to converge instead to a *local optimum*

Backups as Training Examples

Training example:

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

input

target output

Backups as Training Examples

We don't know the real target values, but we can use estimates:

$$\text{e.g., the TD(0) backup:}$$

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

As a training example:

$$\{\text{description of } s_t,\ r_{t+1} + \gamma V(s_{t+1})\}$$

input

target output

Any FA Method?

- In principle, yes:
  - artificial neural networks
  - decision trees
  - multivariate regression methods
  - etc.
- But RL has some special requirements:
  - usually want to learn while interacting
  - ability to handle non-stationarity
  - other?

Gradient Descent Methods

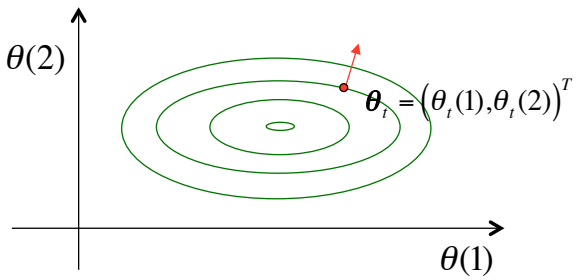
$$\boldsymbol{\theta}_t = \left(\theta_t(1), \theta_t(2), \ldots, \theta_t(n)\right)^T$$

Assume V_t is a (sufficiently smooth) differentiable function of θ_t, for all $s \in S$.

Recall: Gradient Descent

Let f be any function of the parameter space.
Its gradient at any point $\boldsymbol{\theta}_t$ in this space is:

$$\nabla_{\boldsymbol{\theta}} f(\boldsymbol{\theta}_t) = \left(\frac{\partial f(\boldsymbol{\theta}_t)}{\partial \theta(1)}, \frac{\partial f(\boldsymbol{\theta}_t)}{\partial \theta(2)}, \ldots, \frac{\partial f(\boldsymbol{\theta}_t)}{\partial \theta(n)}\right)^T.$$



Iteratively move down the gradient:

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

Gradient Descent Cont.

For the MSE given above and using the chain rule:

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

Gradient Descent Cont.

Alternatively, use just the **sample gradient** instead:

$$\begin{aligned}\theta_{t+1} &= \theta_t - \frac{1}{2}\alpha\nabla_\theta\Big[V^\pi(s_t) - V_t(s_t)\Big]^2 \\ &= \theta_t + \alpha\Big[V^\pi(s_t) - V_t(s_t)\Big]\nabla_\theta V_t(s_t),\end{aligned}$$

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.

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

But We Don't have these Targets

Suppose we just have targets v_t instead:

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

If each v_t is an unbiased estimate of $V^{\pi}(s_t)$,
i.e., $E\{v_t\} = V^{\pi}(s_t)$, then gradient descent converges
to a local minimum (provided α decreases appropriately).

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

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

What about TD(λ) Targets?

$$\theta_{t+1} = \theta_t + \alpha\left[R_t^{\lambda} - V_t(s_t)\right]\nabla_{\theta} V_t(s_t)$$

Not for $\lambda < 1$

But we do it anyway, using the backwards view:

$\theta_{t+1} = \theta_t + \alpha\,\delta_t\,\boldsymbol{e}_t$,

where:

$\delta_t = r_{t+1} + \gamma V_t(s_{t+1}) - V_t(s_t)$, as usual, and

$\boldsymbol{e}_t = \gamma\,\lambda\,\boldsymbol{e}_{t-1} + \nabla_{\theta} V_t(s_t)$

On-Line Gradient-Descent TD(λ)

Initialize $\vec{\theta}$ arbitrarily
Repeat (for each episode):
  $\vec{e} = 0$
  $s \leftarrow$ initial state of episode
  Repeat (for each step of episode):
    $a \leftarrow$ action given by π for s
    Take action a, observe reward, r, and next state, s'
    $\delta \leftarrow r + \gamma V(s') - V(s)$
    $\vec{e} \leftarrow \gamma\lambda\vec{e} + \nabla_{\vec{\theta}} V(s)$
    $\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
    $s \leftarrow s'$
  until s is terminal

Linear Methods

Represent states as feature vectors:

for each $s \in S$:

$$\boldsymbol{\phi}_s = \left(\phi_s(1), \phi_s(2), \ldots, \phi_s(n)\right)^T$$

$$V_t(s) = \boldsymbol{\theta}_t^T \boldsymbol{\phi}_s = \sum_{i=1}^{n} \theta_t(i)\,\phi_s(i)$$

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

Nice Properties of Linear FA Methods

- The gradient is very simple: $\nabla_{\theta} V_t(s) = \boldsymbol{\phi}_s$
- For MSE, the error surface is simple: quadratic surface with a single minumum.
- Linear gradient descent TD(λ) converges:
  - Step size decreases over time
  - On-line sampling (states sampled from the on-policy distribution)
  - Converges to parameter vector $\boldsymbol{\theta}_{\infty}$ with property:

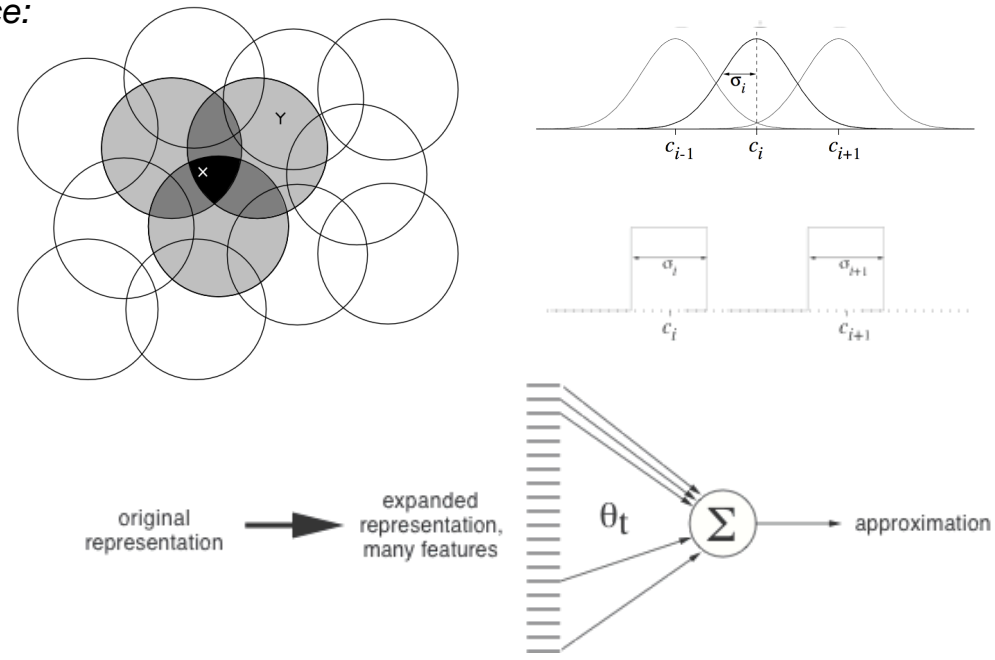
$$MSE(\boldsymbol{\theta}_{\infty}) \leq \frac{1-\gamma\lambda}{1-\gamma} MSE(\boldsymbol{\theta}^*)$$

best parameter vector

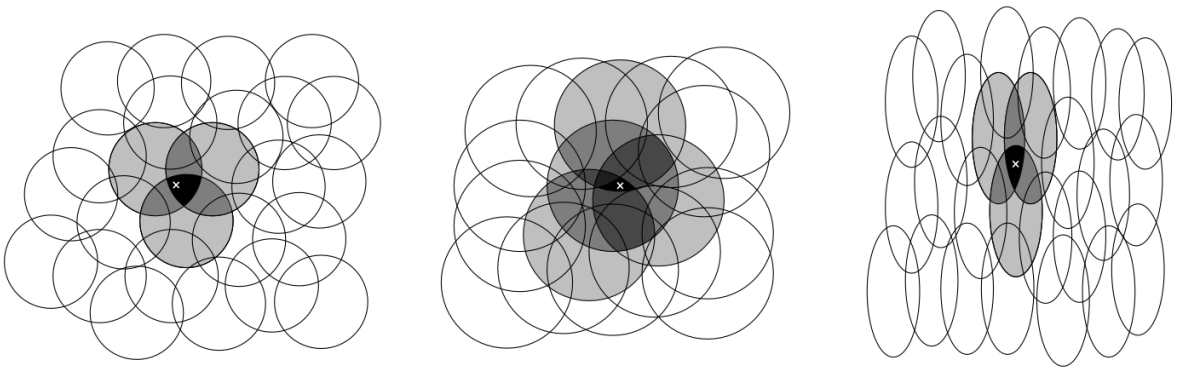
(Tsitsiklis & Van Roy, 1997)

Coarse Coding

Consider a task in which the state set is continuous and two-dimensional. A state in this case is a point in 2-space, a vector with two real components. One kind of feature for this case is those corresponding to *circles in state space:*



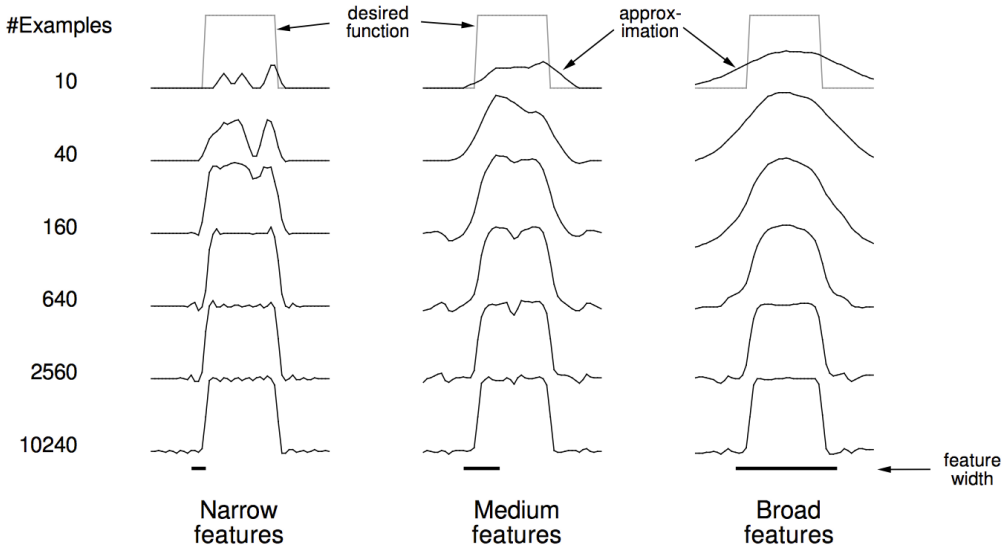
Shaping Generalization in Coarse Coding



a) Narrow generalization b) Broad generalization c) Asymmetric generalization

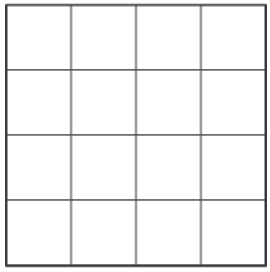
Learning and Coarse Coding

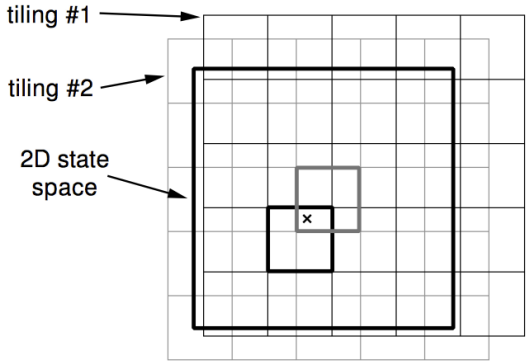
Linear function approximation based on coarse coding was used to learn a one-dimensional square-wave function. The values of this function were used as targets. With just one dimension, the receptive fields were intervals rather than circles.



With broad features, the generalization tended to be broad. However, the final function learned was affected only slightly by the width of the features.

Tile Coding





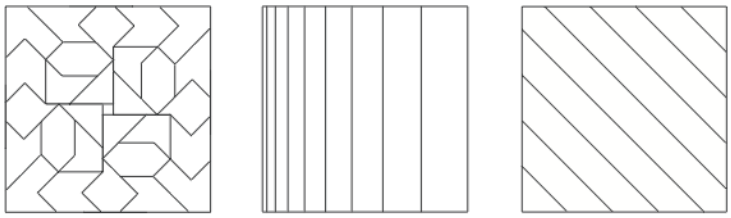
- Binary feature for each tile
- Number of features present at any one time is constant
- Binary features means weighted sum easy to compute
- Easy to compute indices of the features present

Shape of tiles ⇒ Generalization

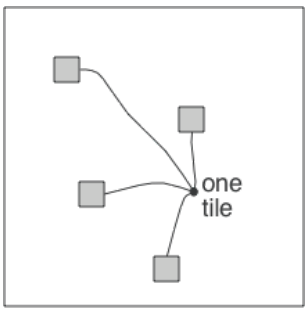
#Tilings ⇒ Resolution of final approximation

Tile Coding Cont.

Irregular tilings



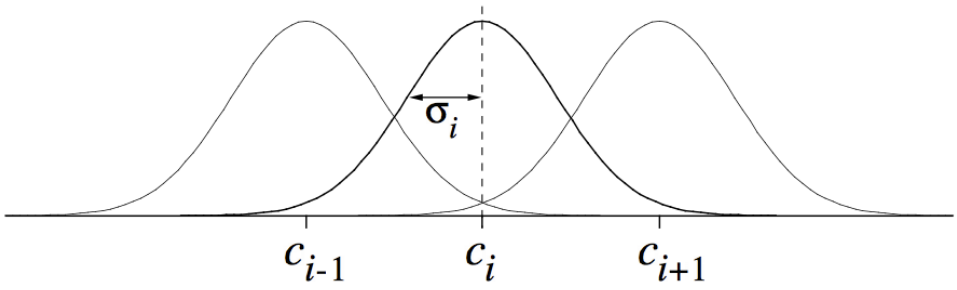
Hashing



Radial Basis Functions (RBFs)

e.g., Gaussians

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



Can you beat 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 bunch of binary **prototypes**
  - Use hamming distance as distance measure
  - Dimensionality is no longer a problem, only complexity

Control with Function Approximation

- Learning state-action values

  Training examples of the form:

$$\{\text{description of } (s_t, a_t),\ v_t\}$$

- The general gradient-descent rule:

$$\theta_{t+1} = \theta_t + \alpha\left[v_t - Q_t(s_t,a_t)\right]\nabla_{\theta} Q(s_t,a_t)$$

- Gradient-descent Sarsa(λ) (backward view):

$$\theta_{t+1} = \theta_t + \alpha\,\delta_t\,\boldsymbol{e}_t$$

where

$$\delta_t = r_{t+1} + \gamma\, Q_t(s_{t+1},a_{t+1}) - Q_t(s_t,a_t)$$

$$\boldsymbol{e}_t = \gamma\,\lambda\,\boldsymbol{e}_{t-1} + \nabla_{\boldsymbol{\theta}} \boldsymbol{Q}_t(s_t,a_t)$$

Continuous action spaces are a topic of ongoing research. Here, stick with discrete action spaces.

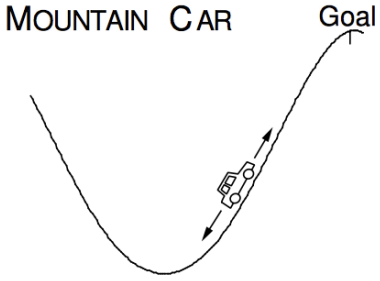
GPI with Linear Gradient Descent Sarsa(λ)

Initialize $\vec{\theta}$ arbitrarily
Repeat (for each episode):
  $\vec{e} = \vec{0}$
  $s, a \leftarrow$ initial state and action of episode
  $\mathcal{F}_a \leftarrow$ set of features present in s, a
  Repeat (for each step of episode):
    For all $i \in \mathcal{F}_a$:
      $e(i) \leftarrow e(i) + 1$ (accumulating traces)
      or $e(i) \leftarrow 1$ (replacing traces)
    Take action a, observe reward, r, and next state, s
    $\delta \leftarrow r - \sum_{i \in \mathcal{F}_a} \theta(i)$
    With probability $1 - \varepsilon$:
      For all $a \in \mathcal{A}(s)$:
        $\mathcal{F}_a \leftarrow$ set of features present in s, a
        $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
      $a \leftarrow \arg\max_a Q_a$
    else
      $a \leftarrow$ a random action $\in \mathcal{A}(s)$
      $\mathcal{F}_a \leftarrow$ set of features present in s, a
      $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
    $\delta \leftarrow \delta + \gamma Q_a$
    $\vec{\theta} \leftarrow \vec{\theta} + \alpha\delta\vec{e}$
    $\vec{e} \leftarrow \gamma\lambda\vec{e}$
  until s is terminal

GPI Linear Gradient Descent Watkins' Q(λ)

Initialize $\vec{\theta}$ arbitrarily
Repeat (for each episode):
  $\vec{e} = \vec{0}$
  $s, a \leftarrow$ initial state and action of episode
  $\mathcal{F}_a \leftarrow$ set of features present in s, a
  Repeat (for each step of episode):
    For all $i \in \mathcal{F}_a$: $e(i) \leftarrow e(i) + 1$
    Take action a, observe reward, r, and next state, s
    $\delta \leftarrow r - \sum_{i \in \mathcal{F}_a} \theta(i)$
    For all $a \in \mathcal{A}(s)$:
      $\mathcal{F}_a \leftarrow$ set of features present in s, a
      $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
    $\delta \leftarrow \delta + \gamma \max_a Q_a$
    $\vec{\theta} \leftarrow \vec{\theta} + \alpha \delta \vec{e}$
    With probability $1 - \varepsilon$:
      For all $a \in \mathcal{A}(s)$:
        $Q_a \leftarrow \sum_{i \in \mathcal{F}_a} \theta(i)$
      $a \leftarrow \arg\max_a Q_a$
      $\vec{e} \leftarrow \gamma \lambda \vec{e}$
    else
      $a \leftarrow$ a random action $\in \mathcal{A}(s)$
      $\vec{e} \leftarrow 0$
  until s is terminal

Mountain-Car Task



Example: Driving an underpowered car up a steep mountain road.
The difficulty is that gravity is stronger than the car's engine. The only solution is to first move away from the goal and up the opposite slope on the left.

Reward: -1 on all time steps until the car reaches the goal, which ends the episode.

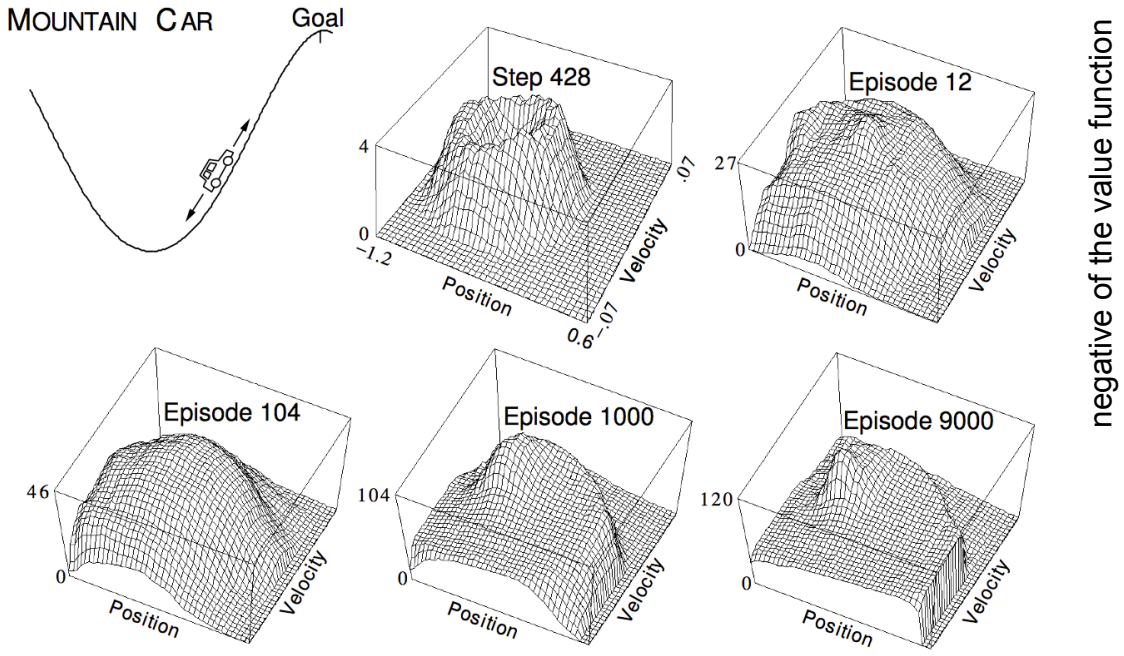
3 actions: full throttle forward, full throttle reverse, and zero throttle.

States: Position, Velocity

To convert the two continuous state variables to binary features, ten 9x9 gridtilings were used, each offset by a random fraction of a tile width.

The initial action values were all zero, which was optimistic (all true values are negative in this task), causing extensive exploration to occur.

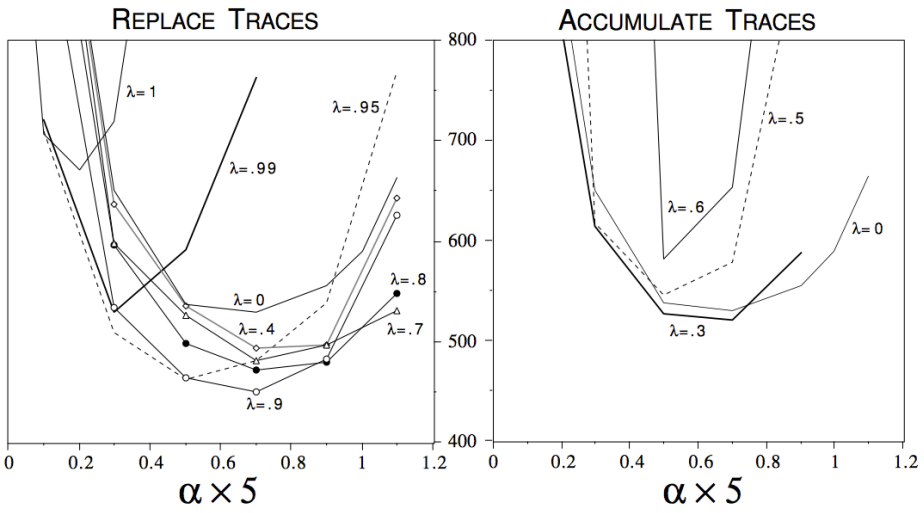
Mountain-Car Task



The Sarsa algorithm (using replace traces and the optional clearing) solved this task, learning a near optimal policy within 100 episodes.

Mountain-Car Results

Steps per episode
averaged over
first 20 trials
and 30 runs



Off policy bootstrapping

Bootstrapping methods are more difficult to combine with function approximation than are non-bootstrapping methods.

In value prediction with linear, gradient-descent function approximation non-bootstrapping methods find minimal MSE solutions for any distribution of training examples, whereas bootstrapping methods find only near-minimal MSE solutions, and only for the on-policy distribution.

Moreover, the quality of the MSE bound for TD(λ) gets worse the farther λ strays from 1, that is, the farther the method moves from its nonbootstrapping form.

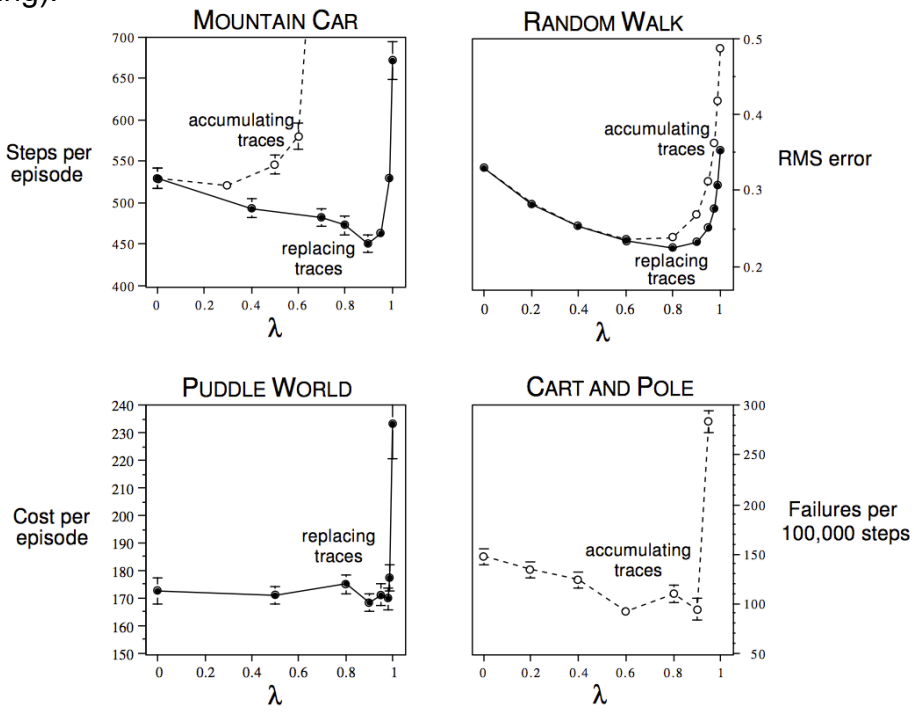
$$MSE(\boldsymbol{\theta}_\infty) \leq \frac{1-\gamma\lambda}{1-\gamma} MSE(\boldsymbol{\theta}^*)$$

Off-policy bootstrapping combined with function approximation can lead to divergence and infinite MSE.

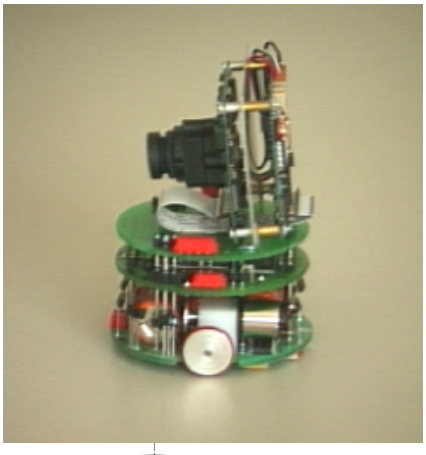
Off-policy control methods do not backup states (or state-action pairs) with exactly the same distribution with which the states would be encountered following the estimation policy (the policy whose value function they are estimating).

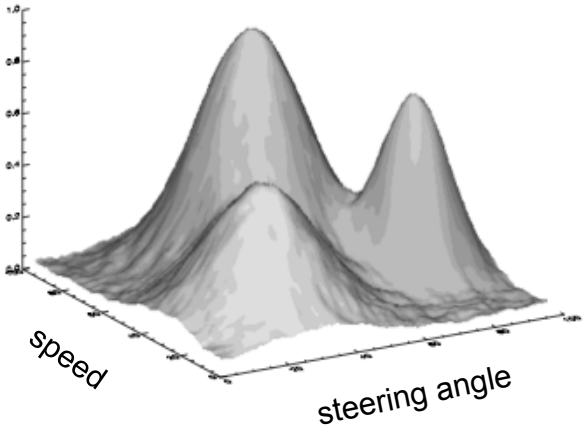
Should We Bootstrap?

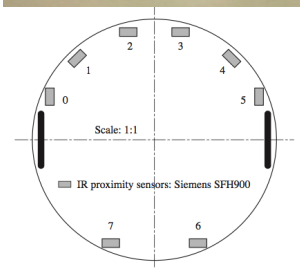
The performance of bootstrapping can be investigated by using a TD method with eligibility traces and vary λ from 0 (pure bootstrapping) to 1 (pure non-bootstrapping).



Example: Topological RL in continuous action spaces



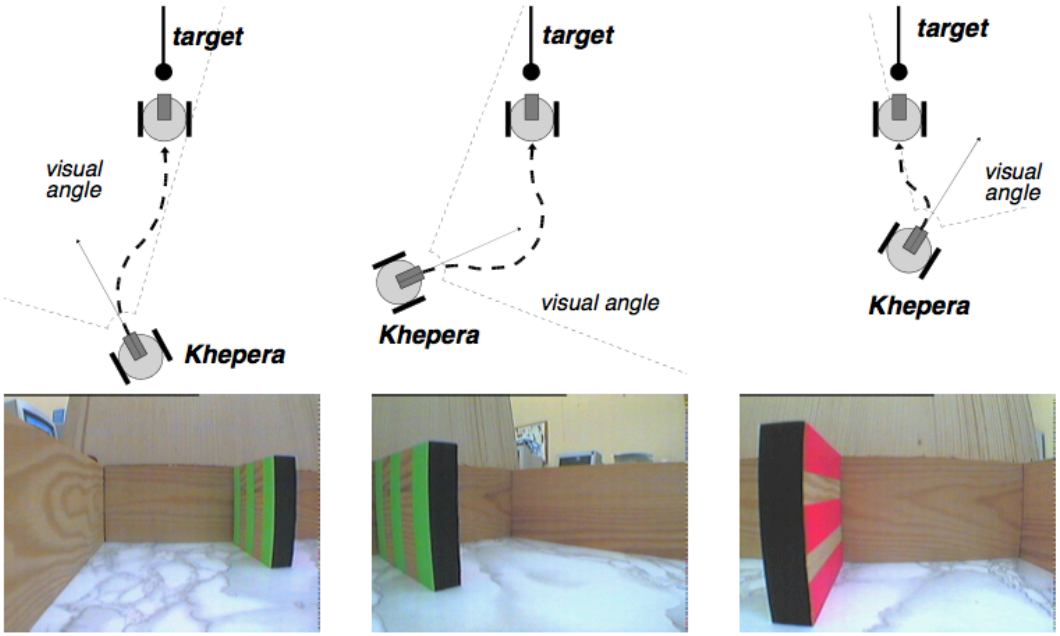




Eight infrared sensors are placed around the robot.

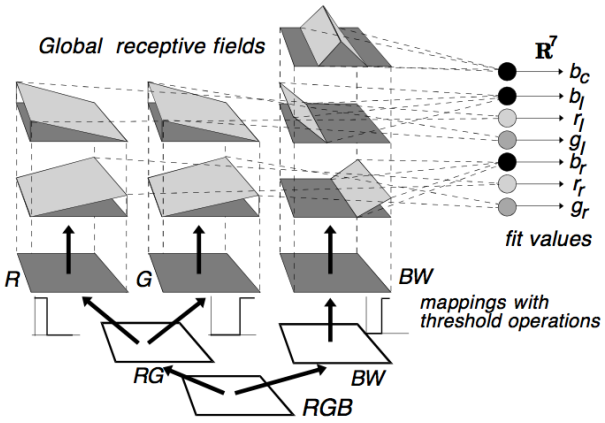
Example: Topological RL in continuous action spaces

Goal: Docking at a target using the camera image.

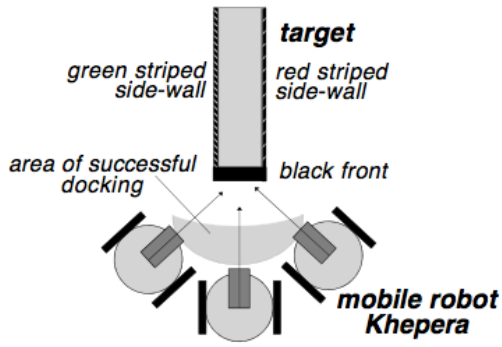


Example: Topological RL in continuous action spaces

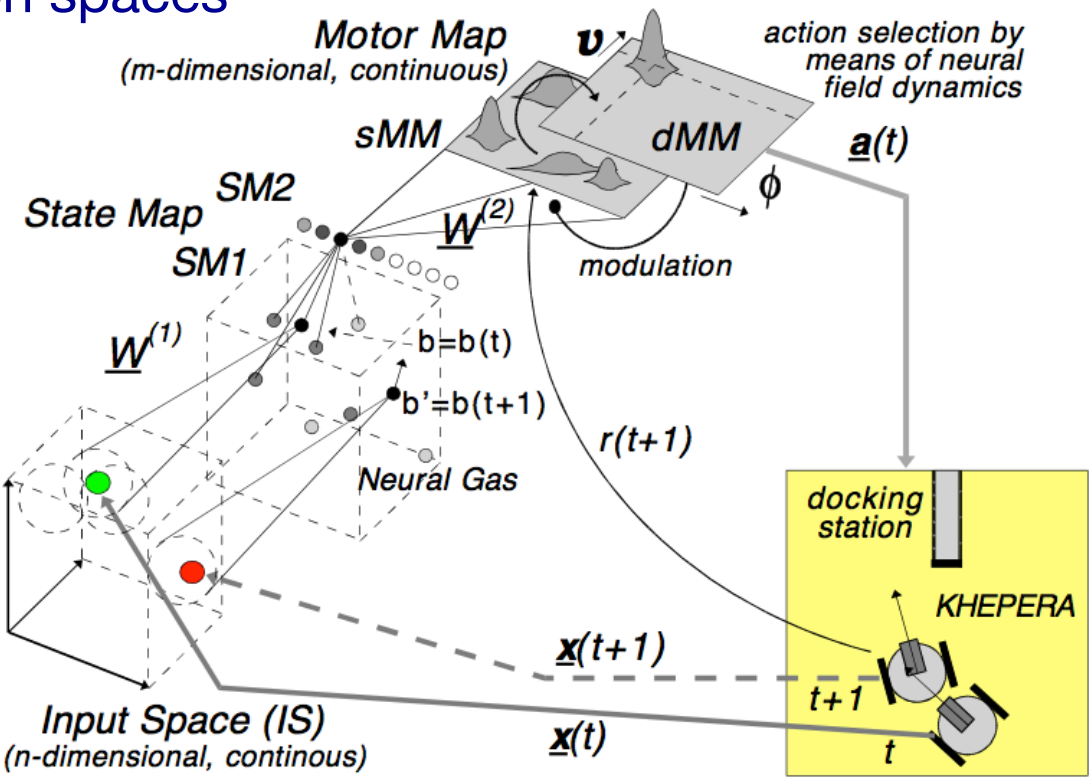
Sensor Input



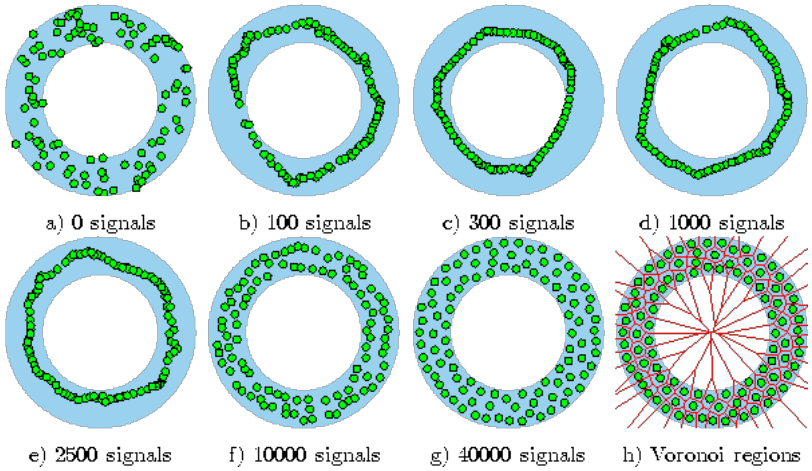
Reward



Example: Topological RL in continuous action spaces



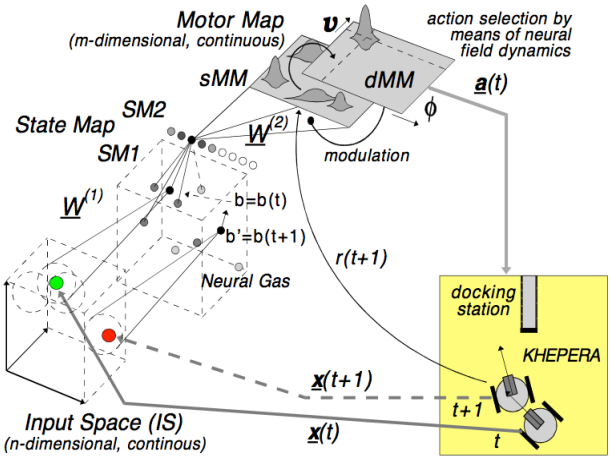
Neural Gas



The **neural gas algorithm** sorts for each input signal the units of the network according to the distance of their reference vectors to the input. Based on this "rank order" a certain number of units is adapted towards the input. Both the number of adapted units and the adaptation strength are decreased according to a fixed schedule.

T. M. Martinetz, S. G. Berkovich, and K. J. Schulten. Neural-gas network for vector quantization and its application to time-series prediction. IEEE Transactions on Neural Networks, 4(4):-569, 1993.

Example: Topological RL in continuous action spaces



1. Initialize the weights $\underline{\mathbf{W}}^{(1)}$ between the *State Map (SM)* and the *Input Space (IS)* and $\underline{\mathbf{W}}^{(2)}$ between the *Motor Map (MM)* and *SM*.

2. Perceive the current sensory situation $\underline{\mathbf{x}}(t)$.

3. Determine the current state by computing the activity $y_s^{sm1}(t) \in \underline{\mathbf{y}}^{sm1}(t)$ of the neurons s in the first layer of the *State Map (SM1)*, a "Neural Gas" vector quantizer [7], according to a specific neighborhood function that considers similarities in the input space:

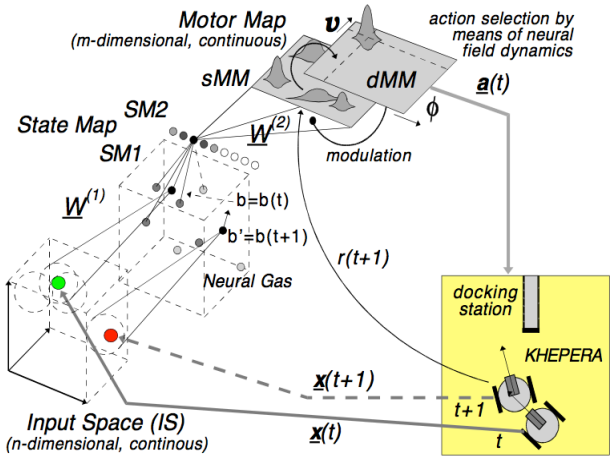
$$y_s^{sm1}(t) = e^{-(k_i/\sigma(t))} \qquad (2)$$

k_i results of a "neighborhood ranking" of the reference vectors $\underline{\mathbf{w}}_s^{(1)}$ for the current input $\underline{\mathbf{x}}(t)$; $\sigma(t)$ is a time-dependent adaptation range.

4. Compute the activity $y_s^{sm2}(t) \in \underline{\mathbf{y}}^{sm2}(t)$ of the neurons s in *SM2* that project the action-values (Q-values) to the static layer of the *Motor Map (sMM)*. Best results have been achieved by the following interpolating activation that considers the responsibility of all *SM1*-neurons for the current input $\underline{\mathbf{x}}(t)$:

$$y_s^{sm2}(t) = \frac{1/\left(\|\underline{\mathbf{x}}(t) - \underline{\mathbf{w}}_s^{(1)}(t)\|\right)}{\sum_{i \in SM1} 1/\left(\|\underline{\mathbf{x}}(t) - \underline{\mathbf{w}}_i^{(1)}(t)\|\right)} \qquad (3)$$

Example: Topological RL in continuous action spaces



5. Compute the activity $y_{\underline{\mathbf{r}}}^{smm}(t)$ of the neurons $\underline{\mathbf{r}}$ in the static layer of the *Motor Map* (sMM) on the basis of the interpolated Q-value mapping from $SM2$ to sMM.

$$y_{\underline{\mathbf{r}}}^{smm}(t) = \sum_s w_{\underline{\mathbf{r}}_s}^{(2)}(t) \cdot y_s^{sm2}(t) \qquad (4)$$

6. Add a randomly positioned Gaussian activity blob $\underline{\mathbf{y}}^{stoch}$ to the activity distribution in sMM. This corresponds to a neural field implementation of a random exploration strategy (Boltzmann exploration):

$$\underline{\mathbf{y}}^{smm}(t) := T(t) \cdot \underline{\mathbf{y}}^{stoch} + (1 - T(t)) \cdot \underline{\mathbf{y}}^{smm}(t) \quad (5)$$

T is a control parameter that can be decreased over time to decrease the exploration rate ($T \in [0.0, 1.0]$).

7. Compute the neural field dynamics in the dynamic layer of the *Motor Map* (dMM) and find the winner-blob in $\underline{\mathbf{y}}^{dmm}(t)$

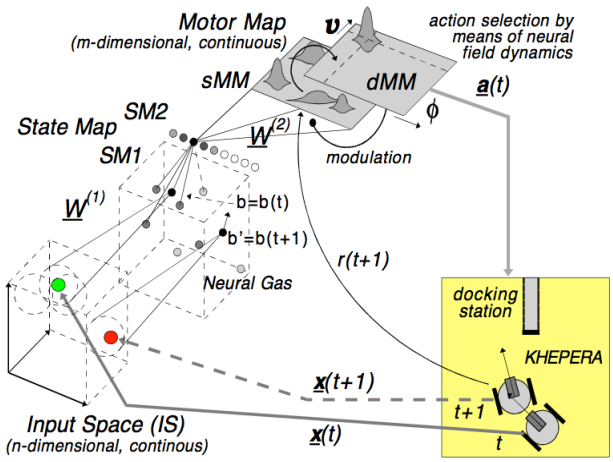
8. Select the corresponding action $\underline{\mathbf{a}}(t) = f(\underline{\mathbf{y}}^{dmm}(t))$ by determination of the center of gravity within dMM.

9. Execute the chosen action $\underline{\mathbf{a}}(t)$, this yields a new situation $\underline{\mathbf{x}}(t+1)$

10. Evaluate the applied action with the immediate reward $r(t+1)$

11. Compute the activity $y_s^{sm1}(t+1)$ of the neurons s in $SM1$ for the new sensory situation $\underline{\mathbf{x}}(t+1)$ according to equation (2).

Example: Topological RL in continuous action spaces



12. Compute the activity $y_s^{sm2}(t+1)$ of the neurons s in $SM2$ according to equation (3).

13. Compute the activity $y_{\underline{\mathbf{r}}}^{smm}(t+1)$ of the neurons $\underline{\mathbf{r}}$ in sMM according to equation (4).

14. Update the weights $\underline{\mathbf{W}}^{(1)}$ of all neurons s in $SM1$ controlled by the activation $y_s^{sm1}(t)$:

$$\begin{aligned} \underline{\mathbf{w}}_s^{(1)}(t+1) &:= \underline{\mathbf{w}}_s^{(1)}(t) + \Delta\underline{\mathbf{w}}_s^{(1)}(t) \\ \Delta\underline{\mathbf{w}}_s^{(1)}(t) &= \eta(t) y_s^{sm1}(t) \left(\underline{\mathbf{x}}(t) - \underline{\mathbf{w}}_s^{(1)}(t)\right) \end{aligned} \quad (6)$$

15. Update the weights $\underline{\mathbf{W}}^{(2)}$ (Q-values) between the neurons $\underline{\mathbf{r}}$ of the dMM-winner-blob and s in $SM2$:

$$\begin{aligned} w_{\underline{\mathbf{r}}_s}^{(2)}(t+1) &:= w_{\underline{\mathbf{r}}_s}^{(2)}(t) + \Delta w_{\underline{\mathbf{r}}_s}^{(2)}(t) \\ \Delta w_{\underline{\mathbf{r}}_s}^{(2)}(t) &= y_{\underline{\mathbf{r}}}^{dmm}(t) y_s^{sm2}(t) \alpha \Big[r(t+1) + \\ &\quad \gamma \max_{\underline{\mathbf{p}}} y_{\underline{\mathbf{p}}}^{smm}(t+1) - w_{\underline{\mathbf{r}}_s}^{(2)}(t) \Big] \end{aligned} \quad (7)$$

$\underline{\mathbf{y}}^{dmm}(t)$ and $\underline{\mathbf{y}}^{sm2}(t)$ serve as gating functions that control the topological action-value learning between the neurons of the dMM-winner-blob and the neighboring $SM2$-neurons representing similar states. α is a constant learning rate (no "freezing") because the agent has to retain its plasticity to cope with a changing environment; γ is the factor discounting future reinforcements ([11]).

16. Switch between time levels:
$\underline{\mathbf{x}}(t) := \underline{\mathbf{x}}(t+1)$; $\underline{\mathbf{y}}^{sm1}(t) := \underline{\mathbf{y}}^{sm1}(t+1)$;
$\underline{\mathbf{y}}^{sm2}(t) := \underline{\mathbf{y}}^{sm2}(t+1)$; $\underline{\mathbf{y}}^{smm}(t) := \underline{\mathbf{y}}^{smm}(t+1)$

17. If $\underline{\mathbf{x}}(t)$ is a final state then terminate else go to **7.**

Example: Topological RL in continuous action spaces

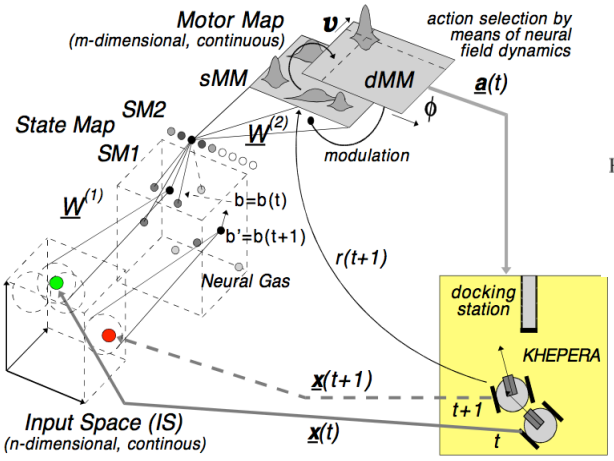


Fig. 6. *Left:* Initial activity distribution in dMM as a result of superimposed action-value mappings from $SM2$ to sMM. *Right:* winner-blob selected by the neural field dynamics within dMM: x-direction codes steering angle $\phi \in [-\phi_{max}, +\phi_{max}]$, y-direction codes speed $v \in [0, v_{max}]$.

Example: Topological RL in continuous action spaces

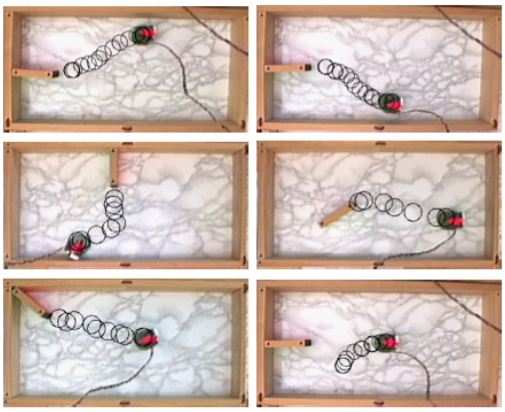


Fig. 7. Temporal traces of successful docking maneuvers to the target located at various positions in the arena (1.-5.) compared to a non-successful docking approach based on a random action selection (bottom right).

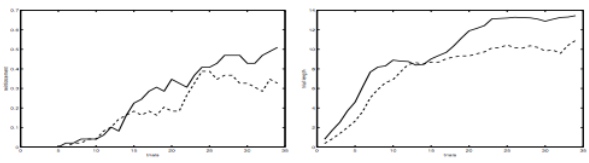
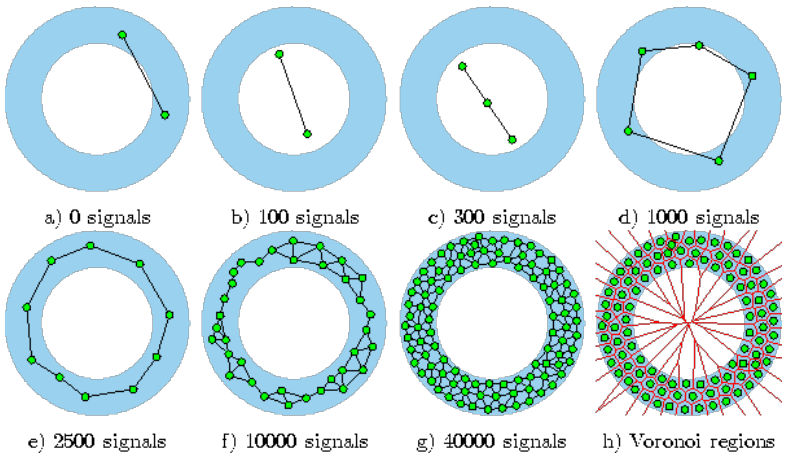


Fig. 8. Temporal evolution of the average reinforcement *(left)* and the mean number of action steps per trial required to achieve the target or done until truncation *(right)* for the neural field based Q-learning (solid line) and the Q-learning model with a discrete action space (dashed line). The diagrams show the mean values of seven independent Q-learning experiments, each of them consists of 35 docking trials.

Growing Neural Gas



The **growing neural gas (GNG) algorithm** starts with very few units and new units are inserted successively. To determine where to insert new units, local error measures are gathered during the adaptation process. Each new unit is inserted near the unit which has accumulated the highest error.

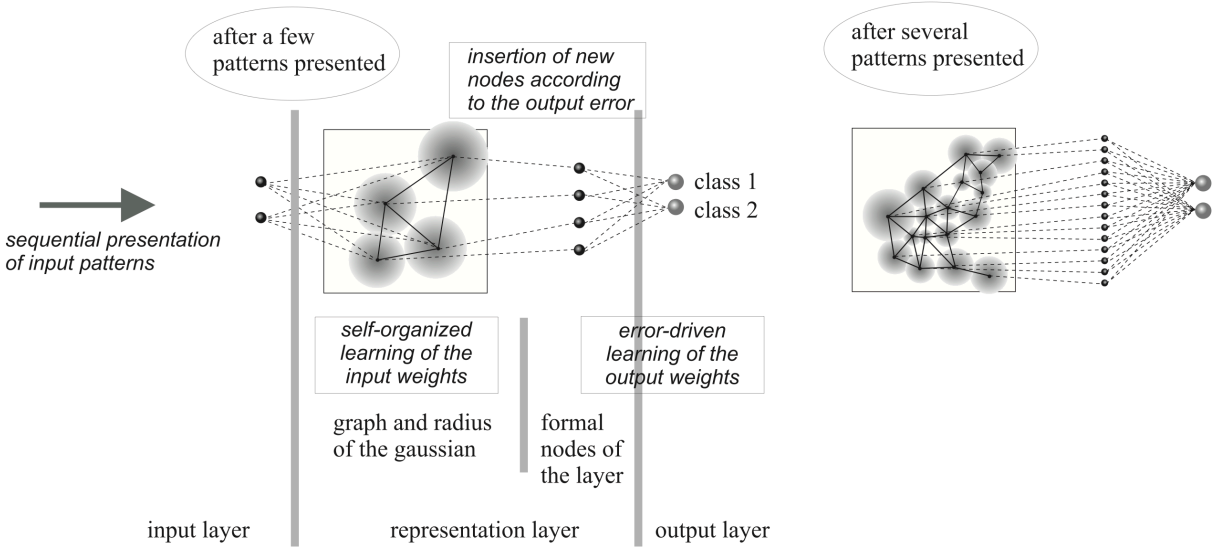
B. Fritzke. A growing neural gas network learns topologies. In G. Tesauro, D. S. Touretzky, and T. K. Leen, editors, Advances in Neural Information Processing Systems 7, pages 625-632. MIT Press, Cambridge MA, 1995.

Stability-Plasticity Dilemma

The problem: learning in a non-stationary environment

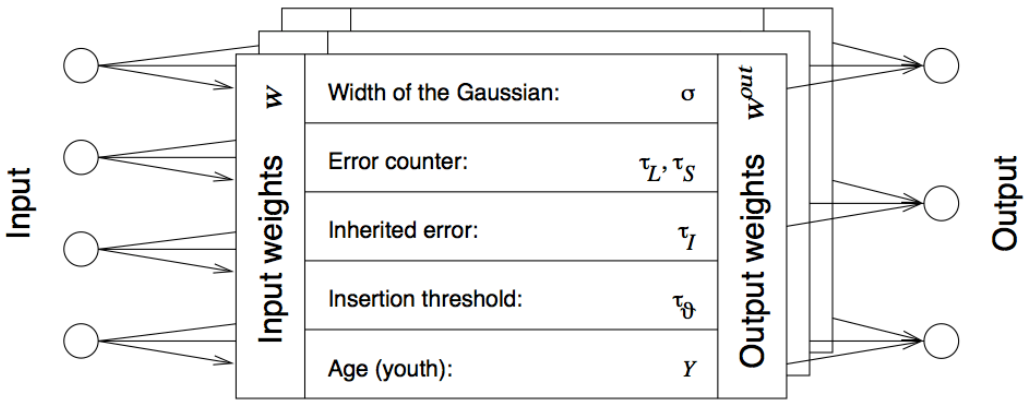
Learning in artificial neural networks inevitably implies forgetting. Later input patterns tend to wash out previous knowledge. A purely stable network is unable to absorb new knowledge from its interactions, whereas a purely plastic network cannot preserve its knowledge.

Life-long learning cell structures



Hamker, F.H. (2001) Life-long Learning Cell Structures – continuously learning without catastrophic interference. Neural Networks, 14:551-573.

Life-long learning cell structures



A node of the Life-long Learning Cell Structures.

Besides the width of the Gaussian, each node has error and age counters. In contrast to the inherited error, which remains fixed until the node is selected for insertion, the error counters are defined as moving averages according to their individual time constant.

Life-long learning cell structures

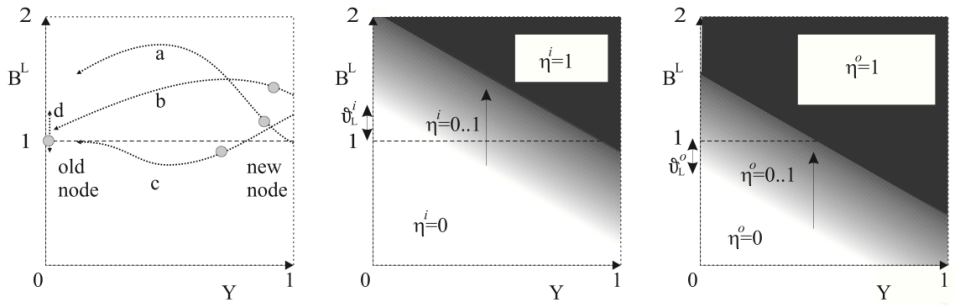
Repeat for each pattern:

Adaptation of the representation layer

- locate the node *b*, which best matches the input pattern by applying a distance measure. Find also the second best node *s*.
- determine the quality measure for learning B^L for *b* and its neighbors *c* separately, determine the individual input learning rate η^i of *b* and of its neighbors *c*.

$$B^{\mathrm{L}}_{(b/c)} = \frac{\tau_{\mathrm{S}(b/c)} + 1}{\tau_{\mathrm{L}(b/c)} + 1} \qquad \forall c \in N_b$$

- move the input weights of the node *b* and its neighbors toward the presented pattern according to the individual learning rate.



Life-long learning cell structures

Repeat for each pattern:

Adaptation of the output layer

- calculate the activation of all nodes in the representation layer by a RBF.
- determine the individual output learning rate for all nodes in the representation layer.
- adapt the output weights to match the desired output by applying the delta rule.

Insertion of nodes in the representation layer

If the algorithm was presented a sufficient number of patterns since the last insertion criterion was applied.

Check insertion

•find the node *q* with the highest value of the insertion criterion. The insertion criterion depends on each quality measure for insertion B^I and on the age of each node.

$$0 < K_{\mathrm{ins},q} = \max_{i \in G} (K_{\mathrm{ins},i}); \qquad K_{\mathrm{ins},i} = B^{\mathrm{I}}_i - Y_i$$

$$B^{\mathrm{I}}_i = \tau_{\mathrm{L}i} - \tau_{\vartheta i}(1 + \vartheta_{\mathrm{ins}}) \qquad \forall i \in G$$

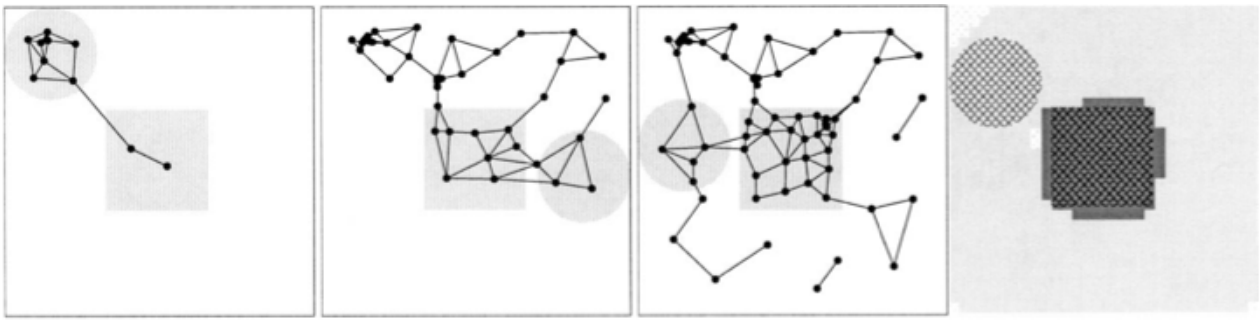
Life-long learning cell structures

- Insert a new node if the insertion criterion is satisfied.
- *Evaluate the previous insertion.*
  - If

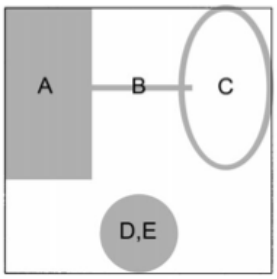
$$\tau_{\mathrm{L}i} \geq \tau_{\mathrm{I}i}(1 - \vartheta_{\mathrm{ins}})$$

The last insertion was not successful
increase the insertion threshold:

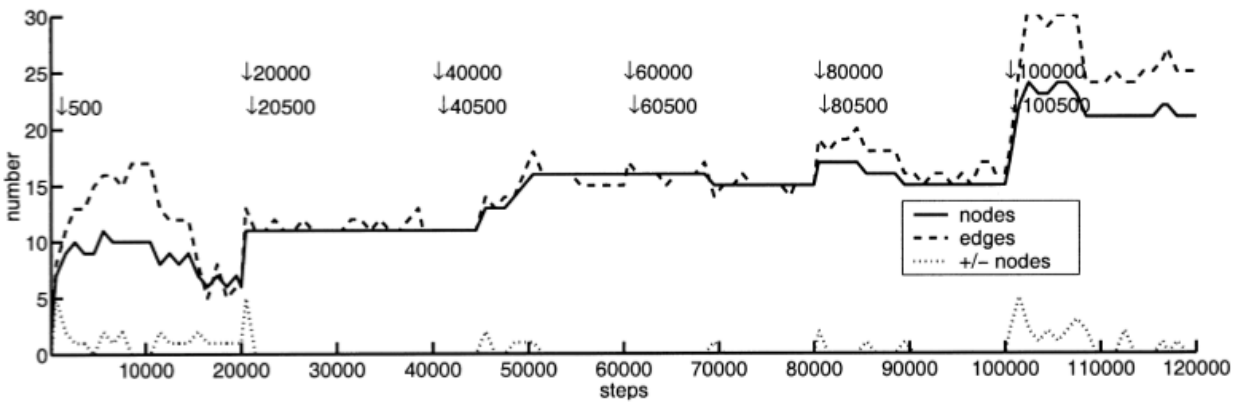
$$\tau_{\vartheta i} := \tau_{\vartheta i} + \eta_{\vartheta}(\tau_{\mathrm{L}i} - \tau_{\vartheta i}(1 - \vartheta_{\mathrm{ins}}))$$



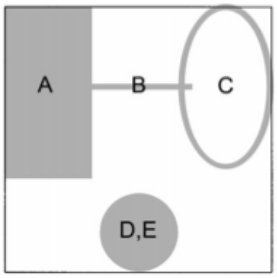
Life-long learning cell structures



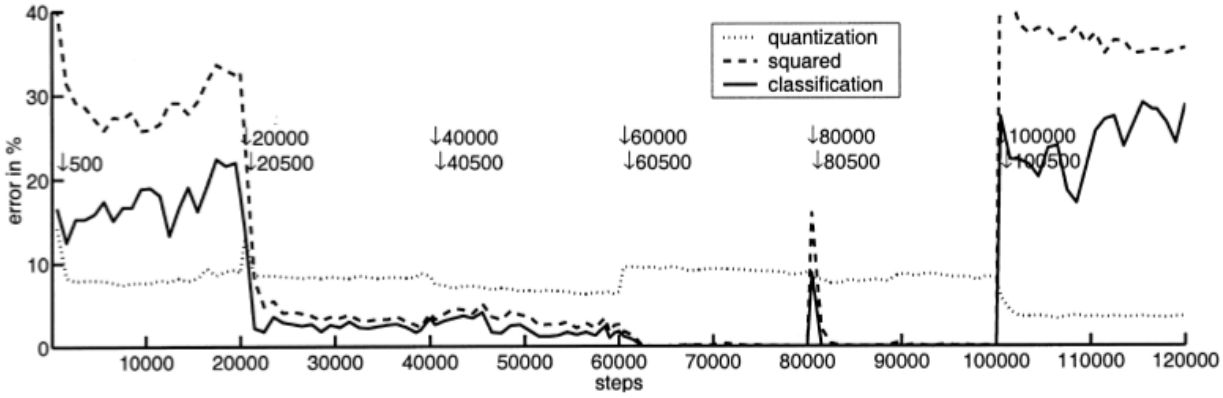
	Class		Environment (Frequency)					
			1	2	3	4	5	6
A	1	Rectangle	1	1	0	1	1	0
B	1	Line	1	1	1	0	0	1
C	2	Ellipse	0	1	1	1	1	0
D	3	Circular area	1	0	0	0	1	1
E	2	Circular area	1	1	1	0	0	1



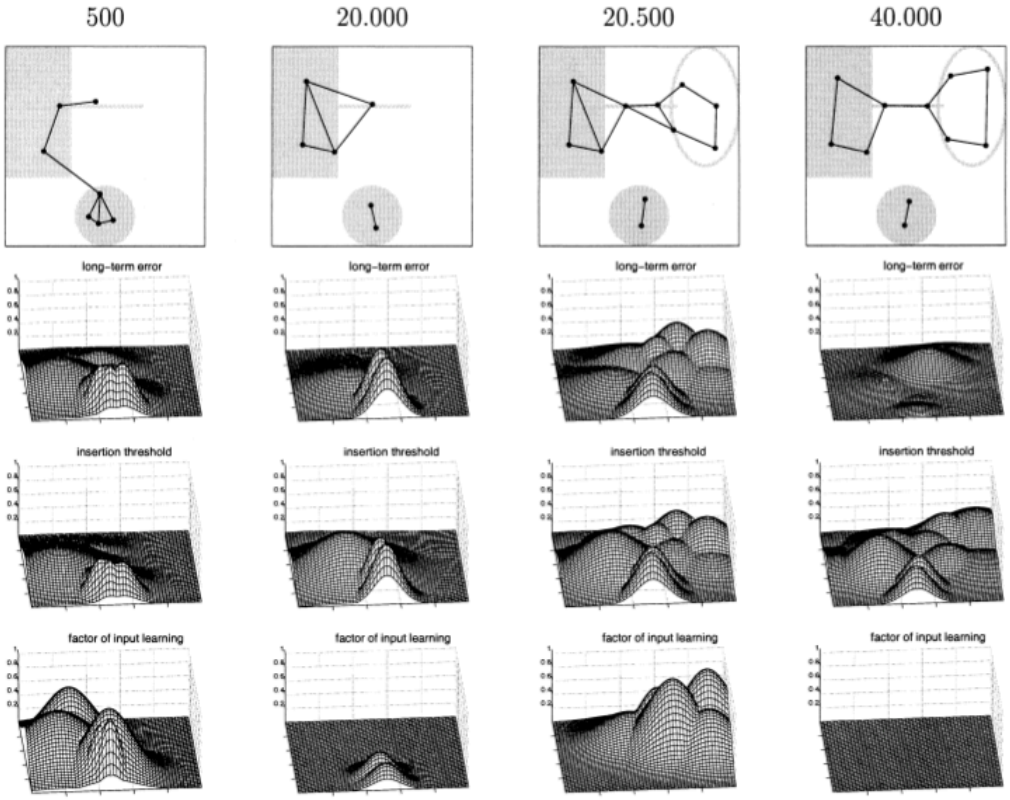
Life-long learning cell structures



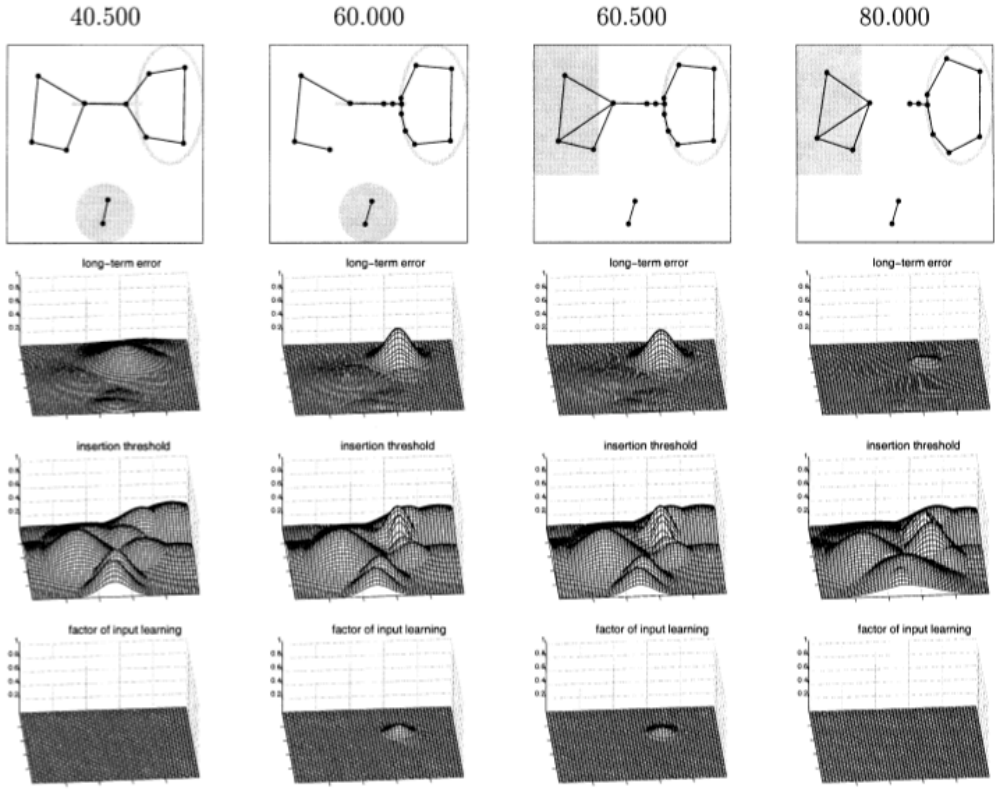
	Class		Environment (Frequency)					
			1	2	3	4	5	6
A	1	Rectangle	1	1	0	1	1	0
B	1	Line	1	1	1	0	0	1
C	2	Ellipse	0	1	1	1	1	0
D	3	Circular area	1	0	0	0	1	1
E	2	Circular area	1	1	1	0	0	1



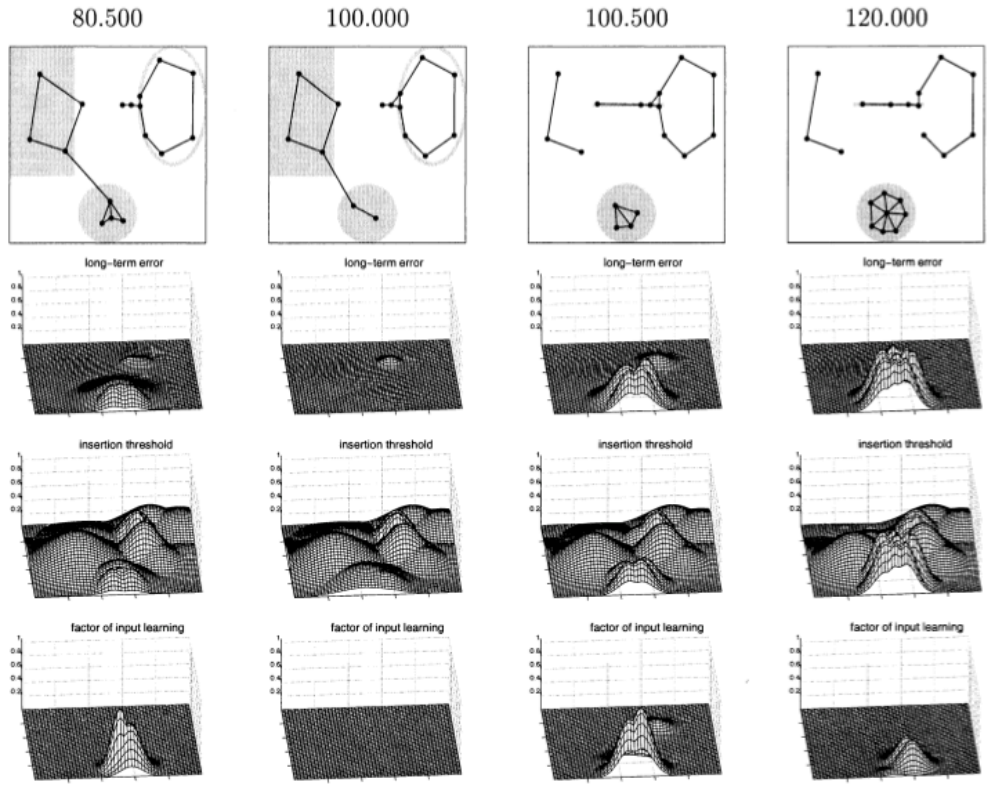
Life-long learning cell structures



Life-long learning cell structures



Life-long learning cell structures



Summary

- Generalization
- Adapting supervised-learning function approximation methods
- Gradient-descent methods
- Linear gradient-descent methods
  - Radial basis functions
  - Tile coding
  - Kanerva coding
- Nonlinear gradient-descent methods? Backpropation?
- Subtleties involving function approximation, bootstrapping and the on-policy/off-policy distinction

References

Tsitsiklis, J. N. and Van Roy, B. (1997). An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control.*