Introduction to Neuroinformatics: Winner-takes-all Networks

Prof. Dr. Martin Riedmiller

University of Osnabrück

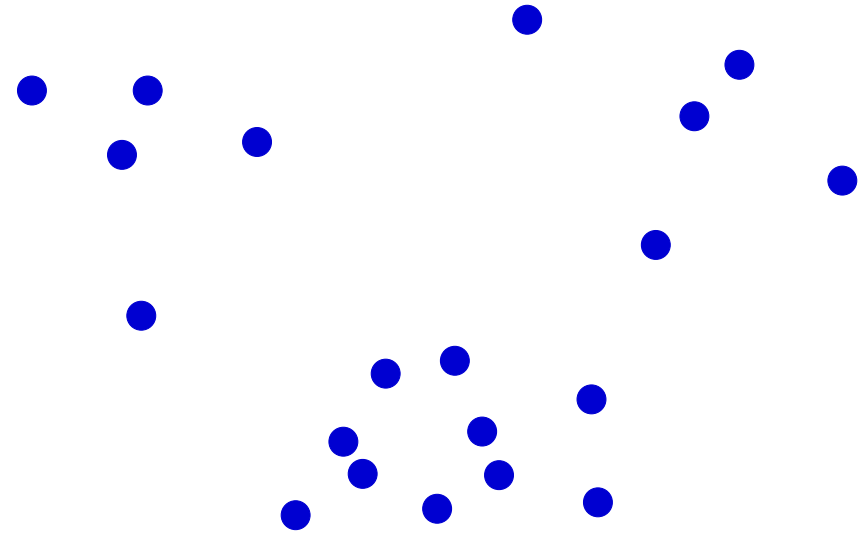
Institute of Computer Science and Institute of Cognitive Science

Outline

- winner-takes-all networks (WTAN), general principle
- WTAN for unsupervised learning
- k-means clustering
- WTAN for classification
- WTAN for structure learning

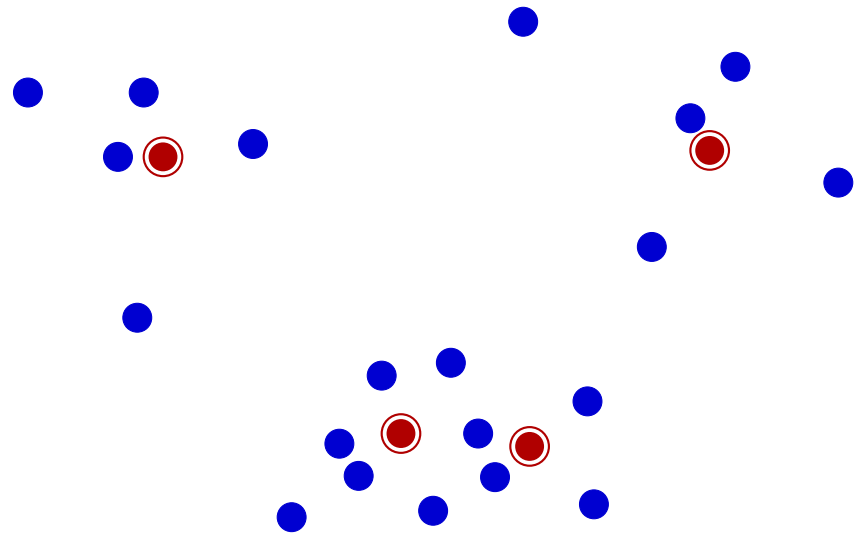
Principle task

- given: a set of points (patterns) in $\mathbb{R}^n$, no labels, no target values



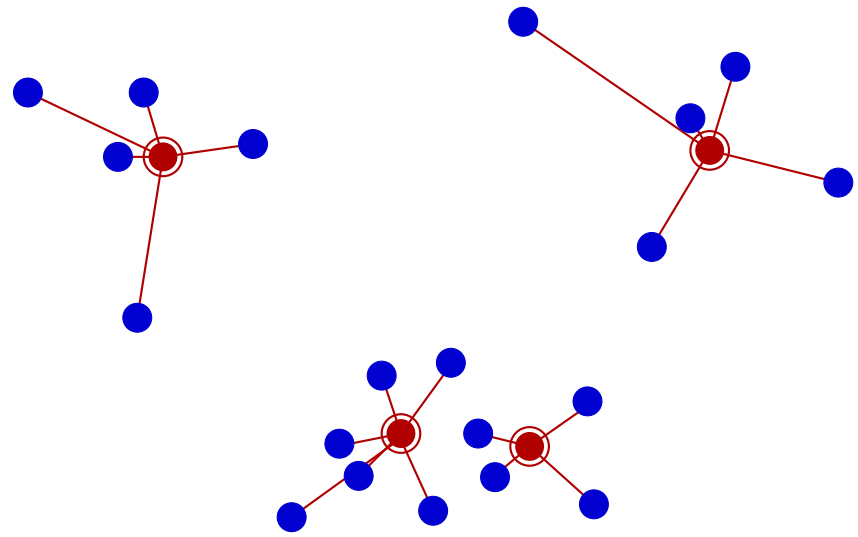
Principle task

- given: a set of points (patterns) in $\mathbb{R}^n$, no labels, no target values
- goal: find some representatives (points in $\mathbb{R}^n$, prototypes) that are closest to the patterns



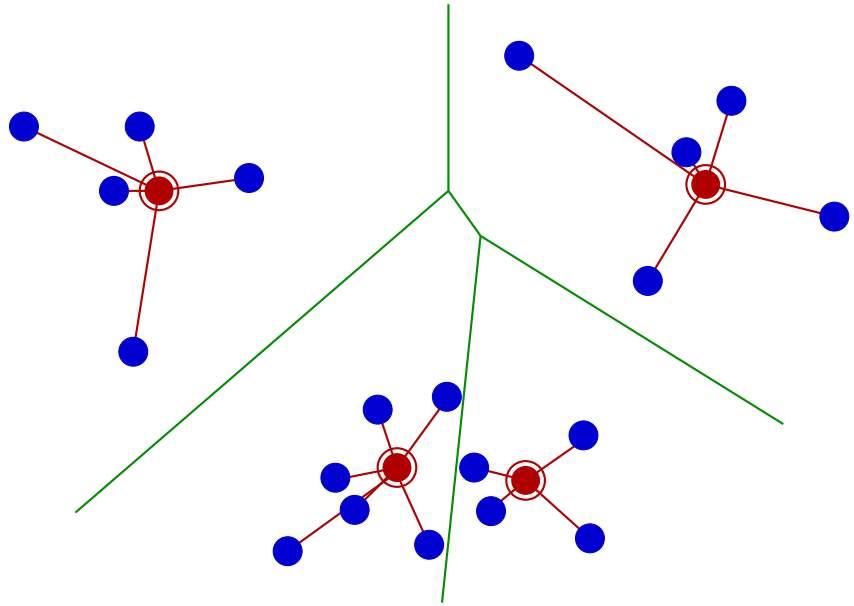
Principle task

- given: a set of points (patterns) in $\mathbb{R}^n$, no labels, no target values
- goal: find some representatives (points in $\mathbb{R}^n$, prototypes) that are closest to the patterns
- position of prototypes should minimize the distance from patterns to closest prototype



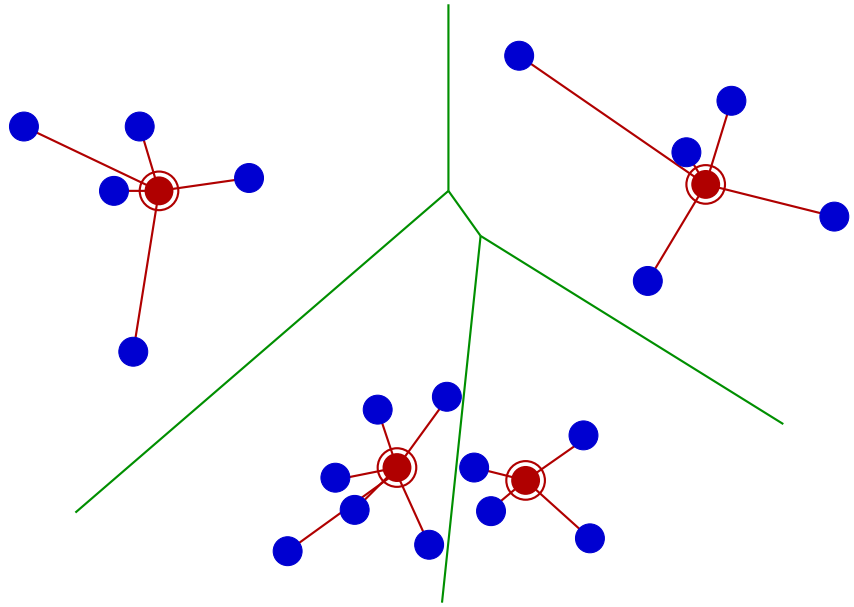
Principle task

- given: a set of points (patterns) in $\mathbb{R}^n$, no labels, no target values
- goal: find some representatives (points in $\mathbb{R}^n$, prototypes) that are closest to the patterns
- position of prototypes should minimize the distance from patterns to closest prototype
- each prototype has an “influence area”, all points in $\mathbb{R}^n$ which are closer to it than to any other prototype



Principle task

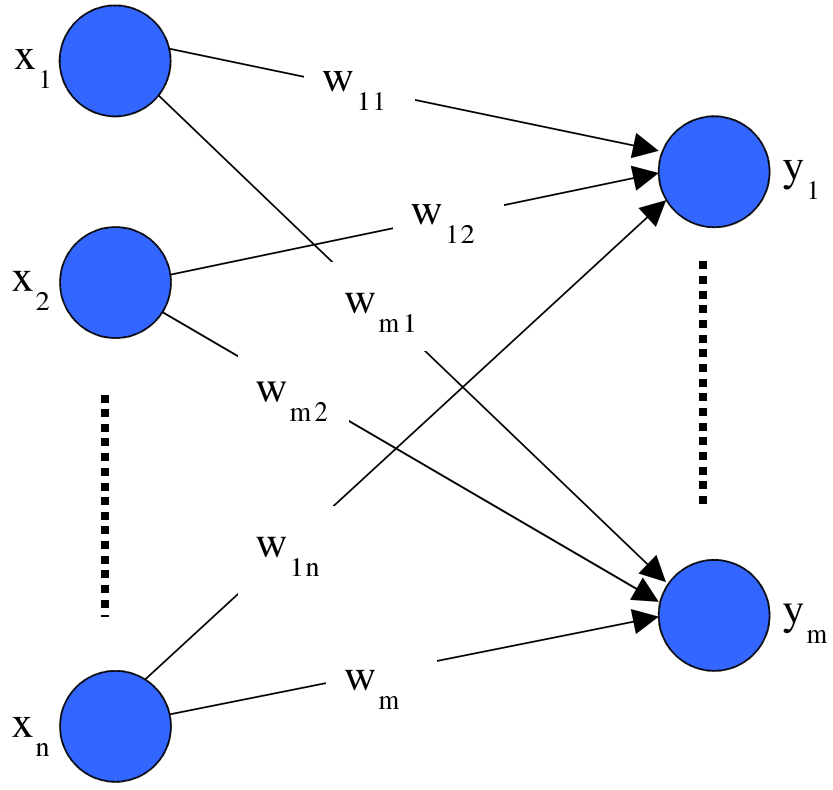
- given: a set of points (patterns) in $\mathbb{R}^n$, no labels, no target values
- goal: find some representatives (points in $\mathbb{R}^n$, prototypes) that are closest to the patterns
- position of prototypes should minimize the distance from patterns to closest prototype
- each prototype has an “influence area”, all points in $\mathbb{R}^n$ which are closer to it than to any other prototype



- this principle is called vector quantization, it is a kind of clustering
- the influence areas are called Voronoi cells

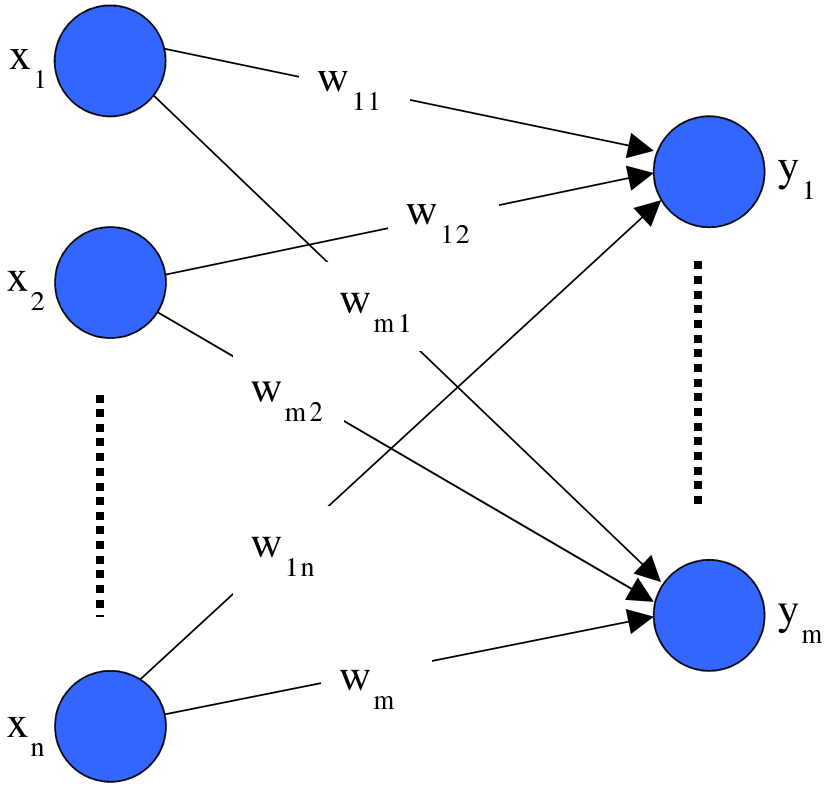
Winner takes all networks

- Winner-takes-all networks (WTAN) are a neural representation of the idea of vector quantization.



Winner takes all networks

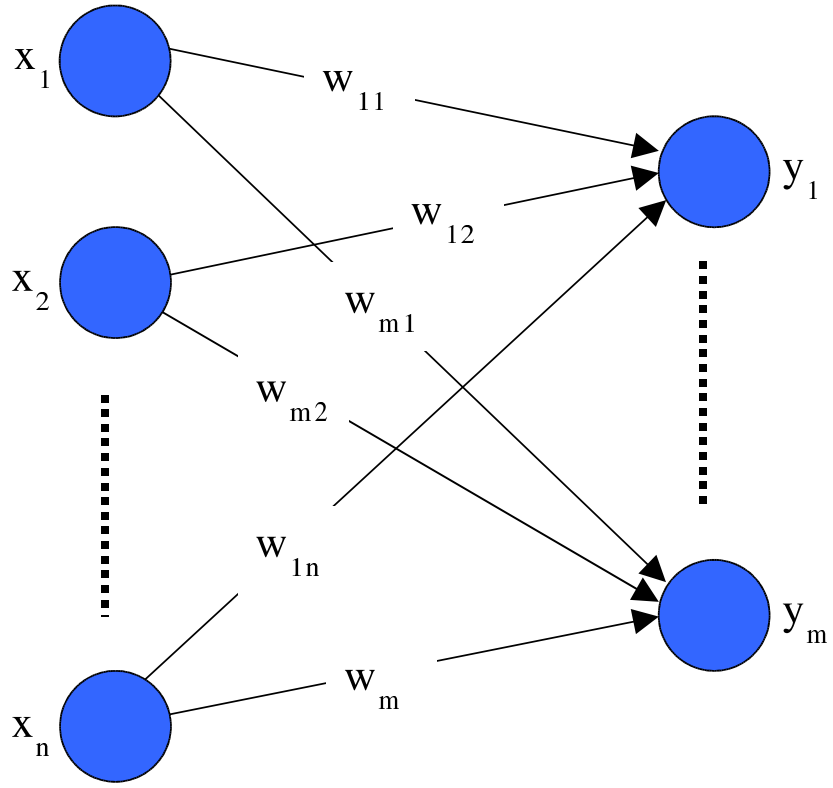
- Winner-takes-all networks (WTAN) are a neural representation of the idea of vector quantization.
- two layers: input layer, output layer
- weighted connections from each input neuron to each output neuron, no bias weights
- the i-th output neuron represents the i-th prototype vector $\vec{w}^{(i)} = (w_{i,1}, \ldots, w_{i,n})^T$



Winner takes all networks (cont.)

- network input of i-th output neuron: $net_i = ||\vec{x} - \vec{w}^{(i)}||^2$, Euclidean distance between input pattern and i-th prototype

$$||\vec{x} - \vec{w}^{(i)}||^2 = \left\langle \vec{x} - \vec{w}^{(i)}, \vec{x} - \vec{w}^{(i)} \right\rangle$$



Winner takes all networks (cont.)

- network input of i-th output neuron: $net_i = ||\vec{x} - \vec{w}^{(i)}||^2$, Euclidean distance between input pattern and i-th prototype

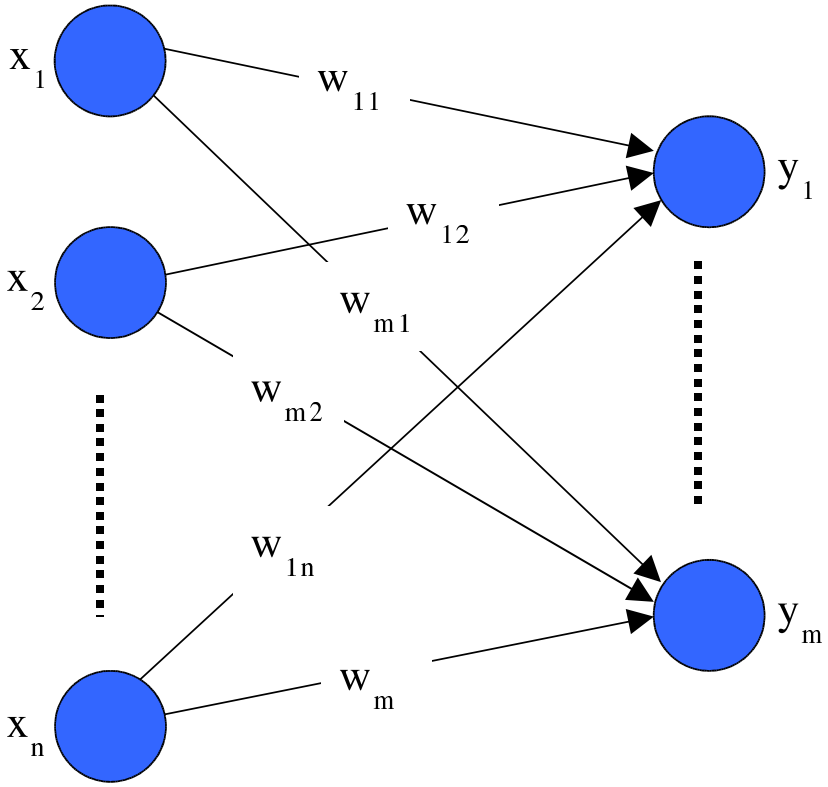
$$||\vec{x} - \vec{w}^{(i)}||^2 = \left\langle \vec{x} - \vec{w}^{(i)}, \vec{x} - \vec{w}^{(i)} \right\rangle$$

- activation of i-th neuron:

$$a_i = \begin{cases} 1 & \text{if } i = \arg\min_j net_j \\ 0 & \text{otherwise} \end{cases}$$

winner-takes-all principle,

1-out-of-m coding



Winner takes all networks (cont.)

- network input of i-th output neuron: $net_i = ||\vec{x} - \vec{w}^{(i)}||^2$, Euclidean distance between input pattern and i-th prototype

  $$||\vec{x} - \vec{w}^{(i)}||^2 = \left\langle \vec{x} - \vec{w}^{(i)}, \vec{x} - \vec{w}^{(i)} \right\rangle$$

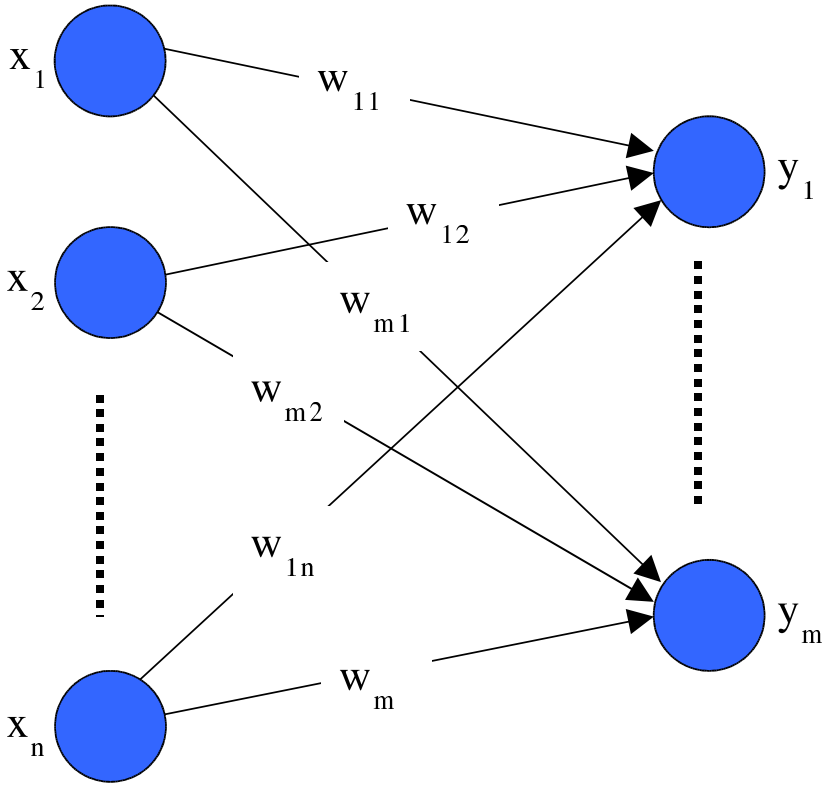
- activation of i-th neuron:

  $$a_i = \begin{cases} 1 & \text{if } i = \arg\min_j net_j \\ 0 & \text{otherwise} \end{cases}$$

  winner-takes-all principle,

  1-out-of-m coding

- applying a pattern $\vec{x}$ the WTAN determines the prototype $\vec{w}^{(i)}$ with the smallest distance to $\vec{x}$



Winner takes all networks (cont.)

- example:

prototypes: $\vec{w}^{(1)} = \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}, \vec{w}^{(2)} = \begin{pmatrix} -3 \\ -2 \\ 0 \end{pmatrix}, \vec{w}^{(3)} = \begin{pmatrix} 0 \\ 2 \\ 4 \end{pmatrix}$

pattern: $\vec{x} = \begin{pmatrix} -1 \\ 2 \\ 1 \end{pmatrix}$

Winner takes all networks (cont.)

- example:

prototypes: $\vec{w}^{(1)} = \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}, \vec{w}^{(2)} = \begin{pmatrix} -3 \\ -2 \\ 0 \end{pmatrix}, \vec{w}^{(3)} = \begin{pmatrix} 0 \\ 2 \\ 4 \end{pmatrix}$

pattern: $\vec{x} = \begin{pmatrix} -1 \\ 2 \\ 1 \end{pmatrix}$

squared distances:

$||\vec{x} - \vec{w}^{(1)}||^2 = 8$

$||\vec{x} - \vec{w}^{(2)}||^2 = 21$

$||\vec{x} - \vec{w}^{(3)}||^2 = 10$

Winner takes all networks (cont.)

- example:

prototypes: $\vec{w}^{(1)} = \begin{pmatrix} 1 \\ 2 \\ -1 \end{pmatrix}, \vec{w}^{(2)} = \begin{pmatrix} -3 \\ -2 \\ 0 \end{pmatrix}, \vec{w}^{(3)} = \begin{pmatrix} 0 \\ 2 \\ 4 \end{pmatrix}$

pattern: $\vec{x} = \begin{pmatrix} -1 \\ 2 \\ 1 \end{pmatrix}$

squared distances:

$||\vec{x} - \vec{w}^{(1)}||^2 = 8$

$||\vec{x} - \vec{w}^{(2)}||^2 = 21$

$||\vec{x} - \vec{w}^{(3)}||^2 = 10$

winner is first prototype, network output is $(1, 0, 0)$

Vector quantization: unsupervised case

- given: training patterns $\mathcal{D} = \{\vec{x}^{(1)}, \ldots, \vec{x}^{(p)}\}$, number of prototypes m
- task: find m prototypes that represent the training set optimally

Vector quantization: unsupervised case

- given: training patterns $\mathcal{D} = \{\vec{x}^{(1)}, \ldots, \vec{x}^{(p)}\}$, number of prototypes m
- task: find m prototypes that represent the training set optimally
- idea: learning by pushing the prototypes towards the patterns

Vector quantization: unsupervised case

- given: training patterns $\mathcal{D} = \{\vec{x}^{(1)}, \ldots, \vec{x}^{(p)}\}$, number of prototypes m
- task: find m prototypes that represent the training set optimally
- idea: learning by pushing the prototypes towards the patterns
- (naive) VQ algorithm:

  1: **loop**
  2: **for all** $\vec{x} \in \mathcal{D}$ **do**
  3: calculate closest prototype j
  4: push $\vec{w}^{(j)}$ towards $\vec{x}$: $\vec{w}^{(j)} \leftarrow \vec{w}^{(j)} + \epsilon(\vec{x} - \vec{w}^{(j)})$
  5: **end for**
  6: **end loop**

  $\epsilon > 0$ is the learning rate, decreasing

Vector quantization: unsupervised case (cont.)

- analyzing the VQ algorithm update rule:

$$\vec{w}^{(j)} \leftarrow \vec{w}^{(j)} + \epsilon(\vec{x} - \vec{w}^{(j)})$$

resembles a gradient descent update rule if we interpret $-(\vec{x} - \vec{w}^{(j)})$ as the gradient of an error function that we want to minimize

Vector quantization: unsupervised case (cont.)

- analyzing the VQ algorithm update rule:

$$\vec{w}^{(j)} \leftarrow \vec{w}^{(j)} + \epsilon(\vec{x} - \vec{w}^{(j)})$$

  resembles a gradient descent update rule if we interpret $-(\vec{x} - \vec{w}^{(j)})$ as the gradient of an error function that we want to minimize

- by integration with respect to $\vec{w}^{(j)}$ we get:

$$e(\vec{x}; \vec{w}^{(j)}) = \frac{1}{2}||\vec{x} - \vec{w}^{(j)}||^2$$

  ($\longrightarrow$ check that for this function the gradient is $-(\vec{x} - \vec{w}^{(j)})$)

Vector quantization: unsupervised case (cont.)

- analyzing the VQ algorithm update rule:

$$\vec{w}^{(j)} \leftarrow \vec{w}^{(j)} + \epsilon(\vec{x} - \vec{w}^{(j)})$$

  resembles a gradient descent update rule if we interpret $-(\vec{x} - \vec{w}^{(j)})$ as the gradient of an error function that we want to minimize

- by integration with respect to $\vec{w}^{(j)}$ we get:

$$e(\vec{x}; \vec{w}^{(j)}) = \frac{1}{2}||\vec{x} - \vec{w}^{(j)}||^2$$

  ($\longrightarrow$ check that for this function the gradient is $-(\vec{x} - \vec{w}^{(j)})$)

- by periodically applying all training patterns VQ realizes a learning by pattern gradient descent approach

Vector quantization: unsupervised case (cont.)

- the corresponding learning by epoch approach would minimize the error term:

$$E(\mathcal{D}; \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\}) = \sum_{i=1}^{p} e(\vec{x}^{(i)}; \vec{w}^{(closest(i))})$$

$$= \frac{1}{2} \sum_{i=1}^{p} ||\vec{x}^{(i)} - \vec{w}^{(closest(i))}||^2$$

(problem in detail: what happens on boundaries of Voronoi cells?)

Vector quantization: unsupervised case (cont.)

- the corresponding learning by epoch approach would minimize the error term:

$$E(\mathcal{D}; \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\}) = \sum_{i=1}^{p} e(\vec{x}^{(i)}; \vec{w}^{(closest(i))})$$

$$= \frac{1}{2} \sum_{i=1}^{p} ||\vec{x}^{(i)} - \vec{w}^{(closest(i))}||^2$$

  (problem in detail: what happens on boundaries of Voronoi cells?)

- hence, VQ performs stochastic gradient descent minimizing the quantization error $E(\mathcal{D}; \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\})$.

Vector quantization: unsupervised case (cont.)

- practical problems of “vanilla” VQ for unsupervised learning:
  - result heavily depends on initial prototypes
  - easily gets stuck in local minima of E
  - some prototypes are never moved/do not represent any pattern
  - prototypes may be located half-way between several clusters
  - the number of prototypes must be defined in advance
  - an appropriate learning rate/decrease of learning rate is difficult to find

  $\Rightarrow$ several modifications to improve learning

k-means: speeding up VQ

- observation: assume the assignment from patterns to prototypes is fix: $h(i)$. Then the error becomes:

$$E(\mathcal{D}; \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\}) = \frac{1}{2} \sum_{i=1}^{p} ||\vec{x}^{(i)} - \vec{w}^{(h(i))}||^2$$

  In this case we can analytically find the minimum:

$$\vec{w}^{(j)} = \frac{1}{|\{i|h(i) = j\}|} \sum_{i|h(i)=j} \vec{x}^{(i)}$$

  i.e. the mean of the patterns solves the problem

- we can replace gradient descent by analytical calculations

k-means: speeding up VQ (cont.)

- k-means algorithm:

1: **repeat**
2: **for all** $\vec{x}^{(i)} \in \mathcal{D}$ **do**
3: calculate closest prototype $h(i)$
4: **end for**
5: **for all prototypes** $\vec{w}^{(j)}$ **do**
6: calculate mean of assigned patterns $\vec{w}^{(j)} \leftarrow \frac{1}{|\{i|h(i)=j\}|} \sum_{i|h(i)=j} \vec{x}^{(i)}$
7: **end for**
8: **until** assignments did not change

k-means: speeding up VQ (cont.)

- Lemma (convergence):

  The k-means algorithm always converges within finite time. The quantization error E decreases during learning until the algorithm has found a local minimum of E.

k-means: speeding up VQ (cont.)

- Lemma (convergence):

  The k-means algorithm always converges within finite time. The quantization error E decreases during learning until the algorithm has found a local minimum of E.

  Proof:

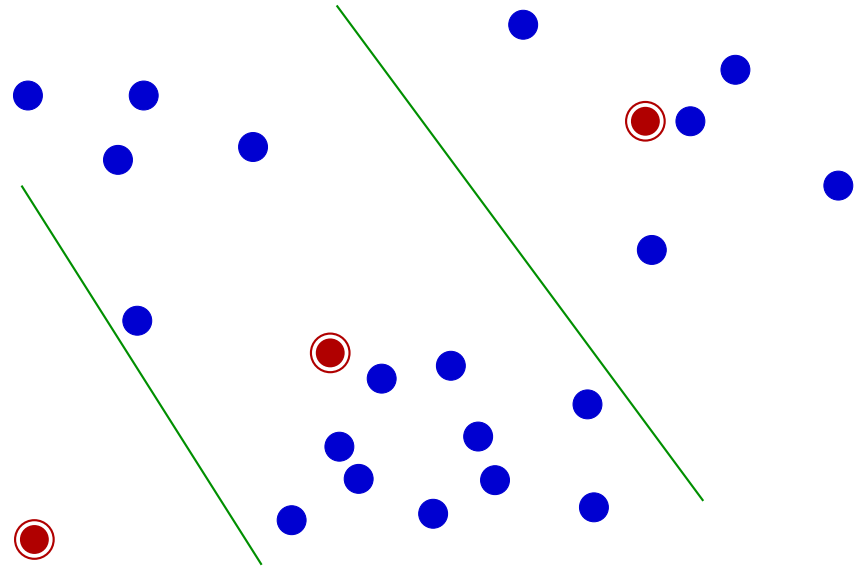
  We have already seen that the mean is the optimal position for a prototype representing a fixed set of patterns. This argument applys to all Voronoi cells, i.e. the optimal place for a prototype representing all patterns within the cell is the mean of the patterns. Hence, moving prototypes to the mean decreases E. Since the number of possible partitionings of the pattern set is finite and E decreases, the algorithm stops after a finite numer of iterations. Finally, the prototypes are in the optimal place with respect to the resulting partitioning.

VQ and k-means

- both approaches minimize the quantization error E
- both approaches get stuck in local optima
- k-means is a batch mode type algorithm
- VQ is a stochastic gradient descent approach
- k-means is much faster than VQ and does not need a learning rate
- typically, k-means is used when all patterns are given in advance, VQ is used when patterns continuously arrive over time, e.g. for online processing of an incoming data stream.

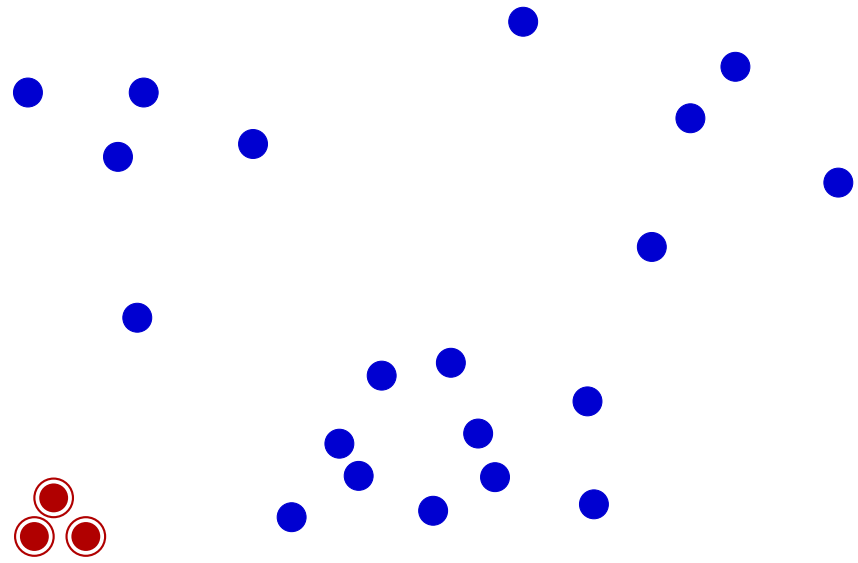
Neural Gas

- common problem with VQ: some prototypes are unused while others represent several clusters
- principle “only update the closest prototyp” fails, results depend heavily on initialization



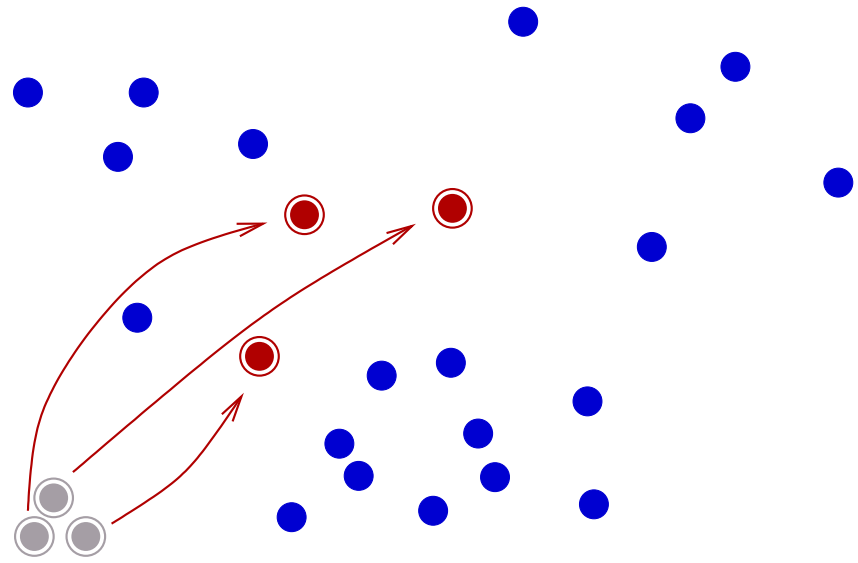
Neural Gas

- common problem with VQ: some prototypes are unused while others represent several clusters
- principle “only update the closest prototyp” fails, results depend heavily on initialization
- what we need: first, prototypes must be moved into the interesting area, then they should concentrate on clusters within the interesting area



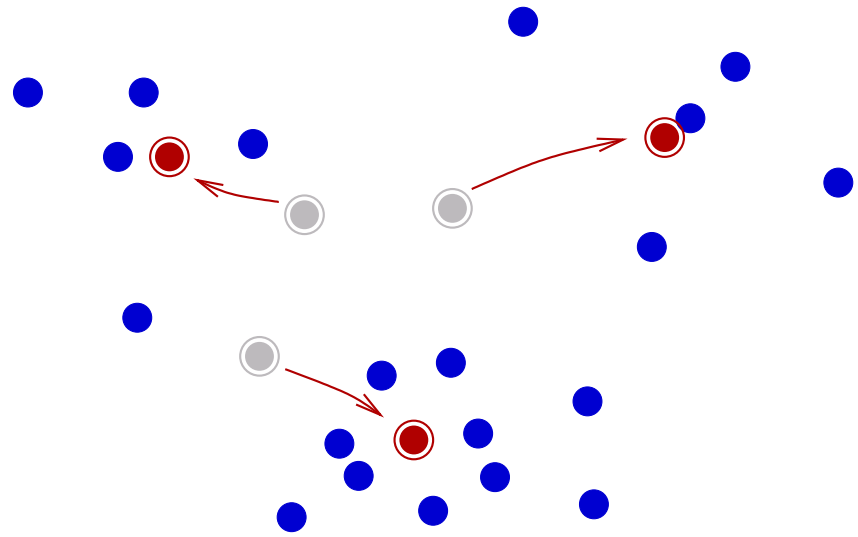
Neural Gas

- common problem with VQ: some prototypes are unused while others represent several clusters
- principle “only update the closest prototyp” fails, results depend heavily on initialization
- what we need: first, prototypes must be moved into the interesting area, then they should concentrate on clusters within the interesting area



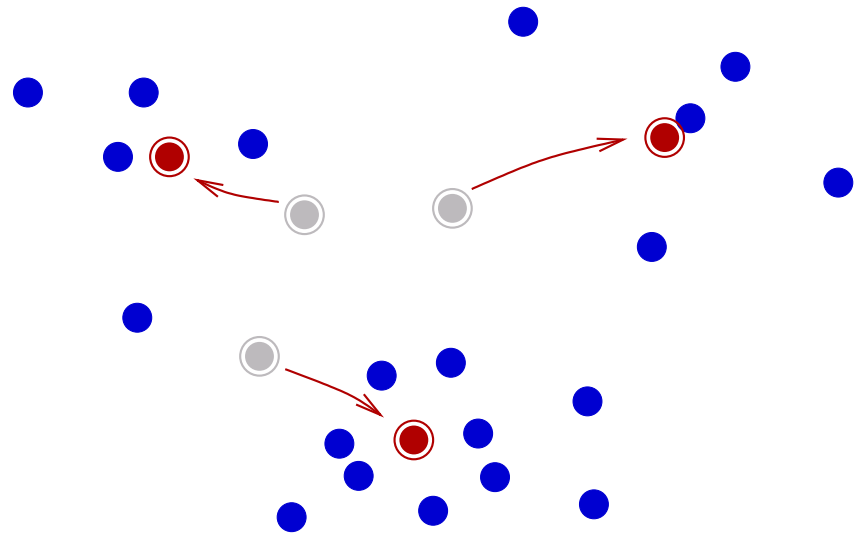
Neural Gas

- common problem with VQ: some prototypes are unused while others represent several clusters
- principle “only update the closest prototyp” fails, results depend heavily on initialization
- what we need: first, prototypes must be moved into the interesting area, then they should concentrate on clusters within the interesting area



Neural Gas

- common problem with VQ: some prototypes are unused while others represent several clusters
- principle “only update the closest prototyp” fails, results depend heavily on initialization
- what we need: first, prototypes must be moved into the interesting area, then they should concentrate on clusters within the interesting area
- Neural gas (Martinetz&Schulten 1991)



Neural gas (cont.)

- idea: always move all prototypes depending on the ranking of distances

1: **loop**
2: **for all** $\vec{x} \in \mathcal{D}$ **do**
3: **for all** prototypes $\vec{w}^{(j)}$ **do**
4: calculate squared distance $d_j \leftarrow ||\vec{x} - \vec{w}^{(j)}||^2$
5: **end for**
6: sort $d_1, \ldots, d_m$ in ascending order
7: let be r_j rank within sorted list of distances
8: **for all** prototypes $\vec{w}^{(j)}$ **do**
9: push $\vec{w}^{(j)}$ towards $\vec{x}$: $\vec{w}^{(j)} \leftarrow \vec{w}^{(j)} + \epsilon e^{-\frac{r_j - 1}{\lambda}} (\vec{x} - \vec{w}^{(j)})$
10: **end for**
11: **end for**
12: **end loop**

$\epsilon > 0$ is the learning rate, decreasing, $\lambda > 0$ parameter, decreasing

Neural gas (cont.)

- example:

$$\vec{w}^{(1)} = (1, 2, -1)^T,$$
$$\vec{w}^{(2)} = (-3, -2, 0)^T,$$
$$\vec{w}^{(3)} = (0, 2, 4)^T$$

$$\vec{x} = (-1, 2, 1)^T$$

Neural gas (cont.)

- example:

  $\vec{w}^{(1)} = (1, 2, -1)^T,$
  $\vec{w}^{(2)} = (-3, -2, 0)^T,$
  $\vec{w}^{(3)} = (0, 2, 4)^T$

  $\vec{x} = (-1, 2, 1)^T$

  squared distances:

  $||\vec{x} - \vec{w}^{(1)}||^2 = 8$
  $||\vec{x} - \vec{w}^{(2)}||^2 = 21$
  $||\vec{x} - \vec{w}^{(3)}||^2 = 10$

  ranking: $r_1 = 1, r_2 = 3, r_3 = 2$

Neural gas (cont.)

- example:

$$\vec{w}^{(1)} = (1, 2, -1)^T,$$
$$\vec{w}^{(2)} = (-3, -2, 0)^T,$$
$$\vec{w}^{(3)} = (0, 2, 4)^T$$

$$\vec{x} = (-1, 2, 1)^T$$

squared distances:

$$||\vec{x} - \vec{w}^{(1)}||^2 = 8$$
$$||\vec{x} - \vec{w}^{(2)}||^2 = 21$$
$$||\vec{x} - \vec{w}^{(3)}||^2 = 10$$

ranking: $r_1 = 1, r_2 = 3, r_3 = 2$

updates ($\epsilon = 0.1, \lambda = 1$):

$$\vec{w}^{(1)} \leftarrow \vec{w}^{(1)} + 0.1 \cdot e^{-\frac{0}{1}}(\vec{x} - \vec{w}^{(1)})$$
$$= (0.8, 2, -0.8)^T$$

$$\vec{w}^{(2)} \leftarrow \vec{w}^{(2)} + 0.1 \cdot e^{-\frac{2}{1}}(\vec{x} - \vec{w}^{(2)})$$
$$= (-2.97, -1.95, 0.01)^T$$

$$\vec{w}^{(3)} \leftarrow \vec{w}^{(3)} + 0.1 \cdot e^{-\frac{1}{1}}(\vec{x} - \vec{w}^{(3)})$$
$$= (-0.04, 1.85, 3.89)^T$$

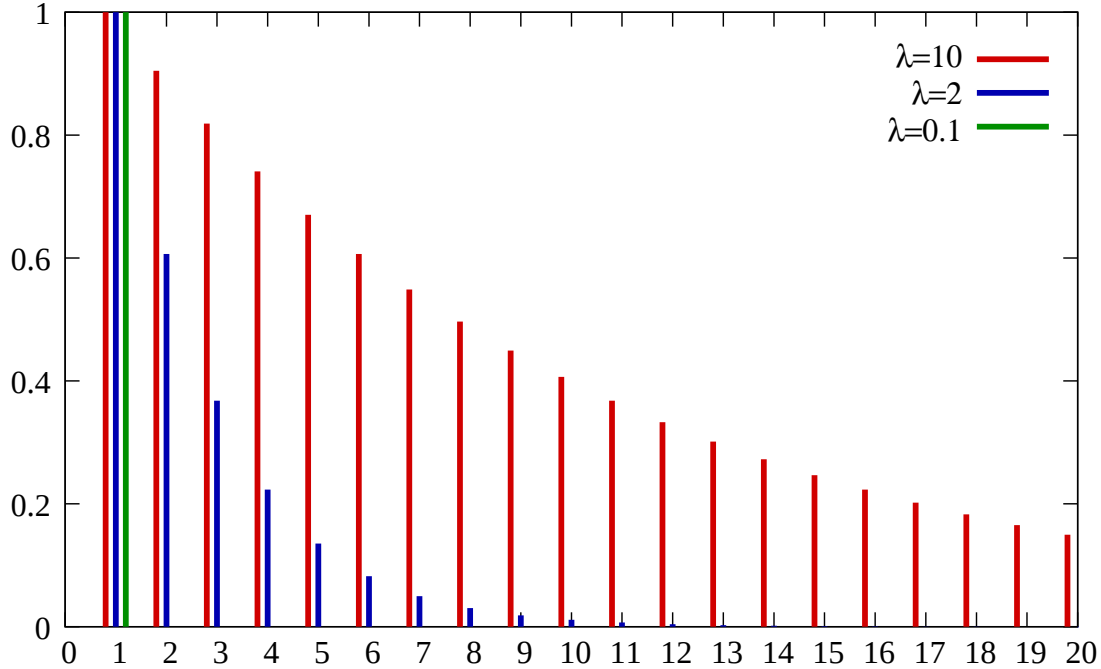
Neural gas (cont.)

- the factor $e^{-\frac{r_j - 1}{\lambda}}$

λ controls how much prototypes with rank ≥ 2 are moved towards $\vec{x}$

large $\lambda \Rightarrow$ all prototypes are moved considerably

small $\lambda \Rightarrow$ only low rank prototypes are moved considerably



Neural gas (cont.)

- parameter λ (connectivity): typically start with large value and decrease over time
- parameter ϵ (learning rate): same as for VQ, decrease over time

Neural gas (cont.)

- parameter λ (connectivity): typically start with large value and decrease over time
- parameter ϵ (learning rate): same as for VQ, decrease over time
- error function:

$$E(\mathcal{D}; \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\}) = \frac{1}{2} \sum_{i=1}^{p} \sum_{j=1}^{m} e^{-\frac{r_j(i)-1}{\lambda}} ||\vec{x}^{(i)} - \vec{w}^{(j)}||^2$$

  converges to error function of VQ for $\lambda \longrightarrow 0$
- a batch learning algorithm (similar to k-means) exists

Learning vector quantization: classification

- VQ can be extended for classification tasks
  $\Rightarrow$ Learning vector quantization (LVQ)
- main idea: each prototype is provided with a class label
  - patterns attract prototypes of the same class
  - patterns repel prototypes of other class

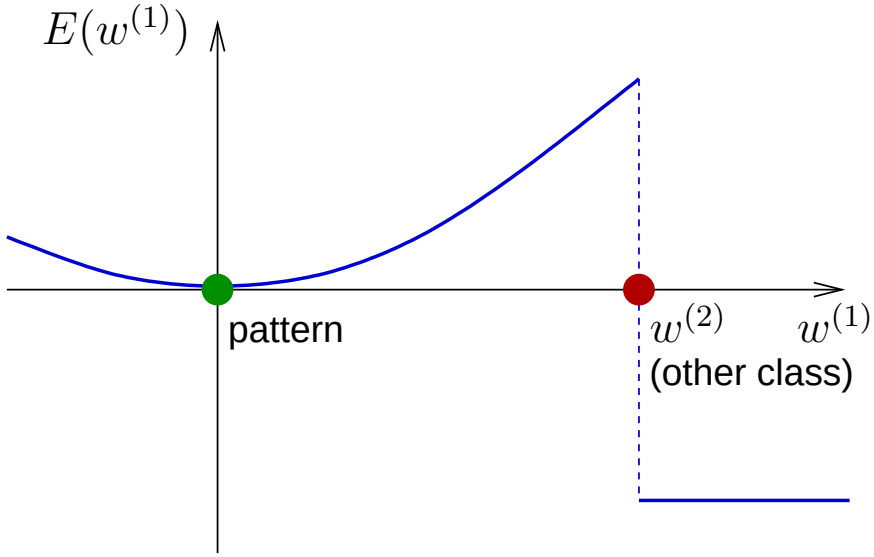
Learning vector quantization: classification

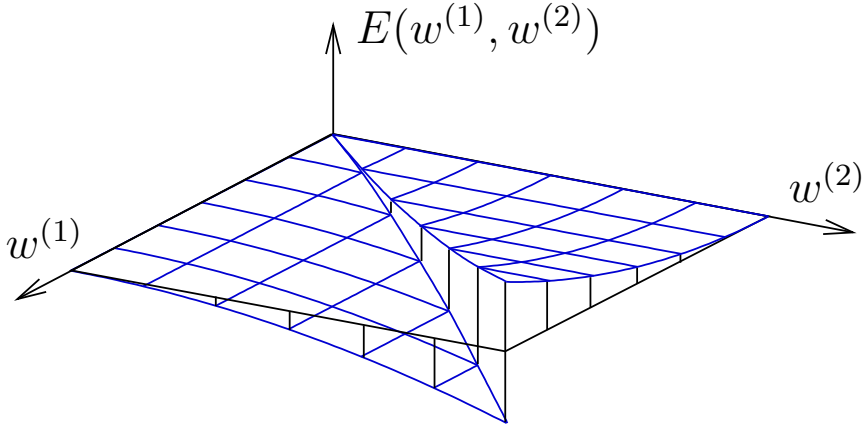
- VQ can be extended for classification tasks
  $\Rightarrow$ Learning vector quantization (LVQ)
- main idea: each prototype is provided with a class label
  - patterns attract prototypes of the same class
  - patterns repel prototypes of other class

1: **loop**
2: **for all** $(\vec{x}, d) \in \mathcal{D}$ **do**
3: calculate closest prototype j
4: **if** $d =$ class label of prototype j **then**
5: $\vec{w}^{(j)} \leftarrow \vec{w}^{(j)} + \epsilon(\vec{x} - \vec{w}^{(j)})$
6: **else**
7: $\vec{w}^{(j)} \leftarrow \vec{w}^{(j)} - \epsilon(\vec{x} - \vec{w}^{(j)})$
8: **endif**
9: **end for**
10: **end loop**

Learning vector quantization: classification (cont.)

- what is the error function that is minimized with this algorithm?
- simple example: one pattern, two prototypes: $\vec{w}^{(1)}$ of same class, $\vec{w}^{(2)}$ of different class





strange error function, discontinuous at boundaries of Voronoi cells, not bounded below, becomes small if closest prototype is of wrong class but far away

Generalized LVQ

- we cannot simply interpret vanilla LVQ as stochastic minimization of an error function, but we can try to design an error function:
  - for correct classification it must yield a minimum
  - wrong classification must be penalized
  - function must be differentiable
  - each pattern should contribute additively

Generalized LVQ

- we cannot simply interpret vanilla LVQ as stochastic minimization of an error function, but we can try to design an error function:
  - for correct classification it must yield a minimum
  - wrong classification must be penalized
  - function must be differentiable
  - each pattern should contribute additively
- generic form:

$$E(\mathcal{D}; \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\}) = \sum_{i=1}^{p} e((\vec{x}^{(i)}, d^{(i)}); \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\})$$

with e: a differentiable error term for a single pattern

Generalized LVQ (cont.)

- attempt with step function:

$$e((\vec{x}, d); \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\}) = \begin{cases} 1 & \text{if } \delta^+ > \delta^- \\ 0 & \text{if } \delta^+ < \delta^- \end{cases}$$

δ^+: squared distance of $\vec{x}$ to closest prototype of same class
δ^-: squared distance of $\vec{x}$ to closest prototype of other class

problem: error term is not differentiable, even not continuous

Generalized LVQ (cont.)

- attempt with step function:

$$e((\vec{x}, d); \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\}) = \begin{cases} 1 & \text{if } \delta^+ > \delta^- \\ 0 & \text{if } \delta^+ < \delta^- \end{cases}$$

δ^+: squared distance of $\vec{x}$ to closest prototype of same class
δ^-: squared distance of $\vec{x}$ to closest prototype of other class

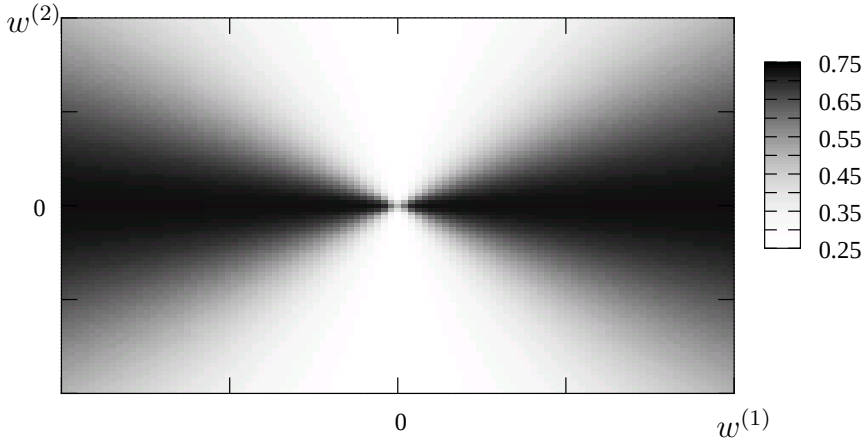
problem: error term is not differentiable, even not continuous

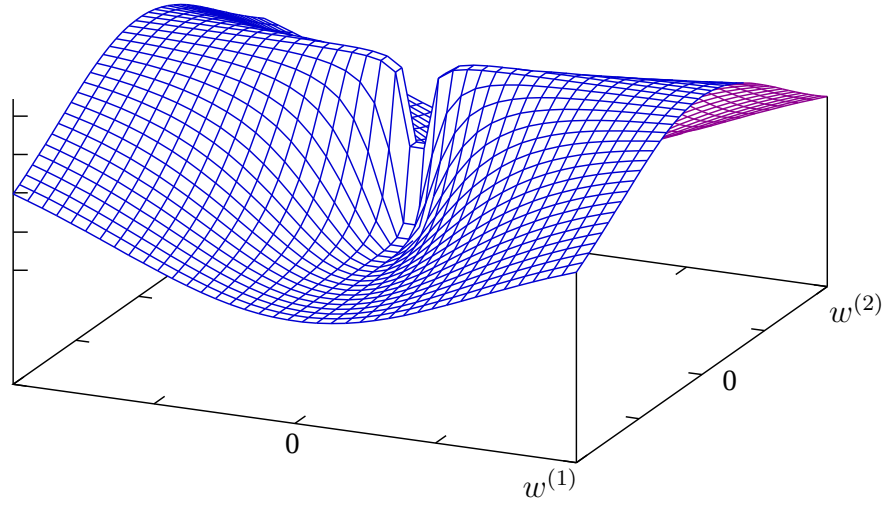
- idea: relax step function, use logistic function as approximation instead:

$$e((\vec{x}, d); \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\}) = f_{log}\left(\frac{\delta^+ - \delta^-}{\delta^+ + \delta^-}\right)$$

Generalized LVQ (cont.)

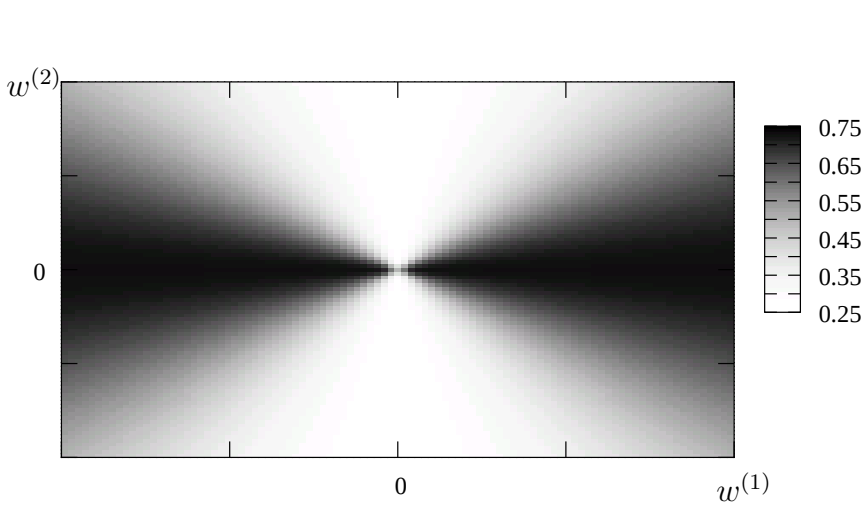
- example: one pattern, two prototypes: $\vec{w}^{(1)}$ of same class, $\vec{w}^{(2)}$ of different class

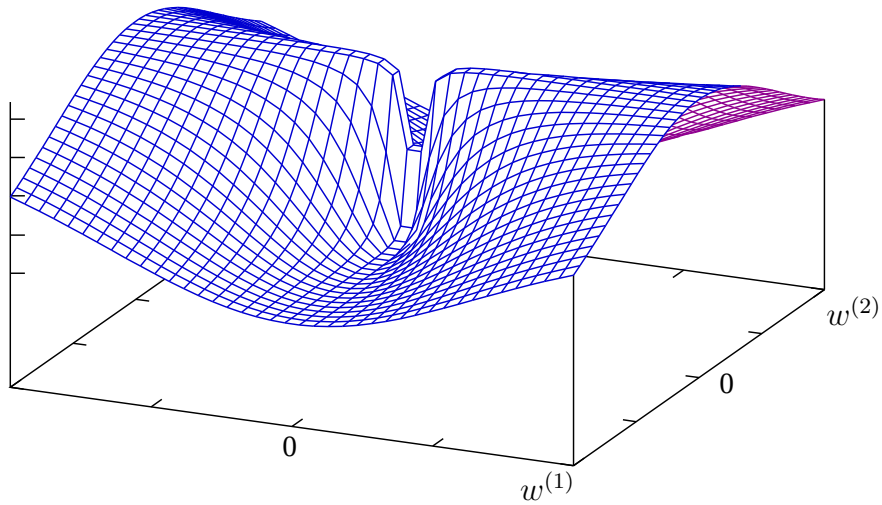




Generalized LVQ (cont.)

- example: one pattern, two prototypes: $\vec{w}^{(1)}$ of same class, $\vec{w}^{(2)}$ of different class





- error function is continuous and differentiable, except:
  - on the boundaries of Voronoi cells it is continuous but non-differentiable (same as with all other prototype based approaches)
  - if $\delta^+ = \delta^- = 0$ the error term is undefined. Possible solution: add a very small positive number to the denominator of $\frac{\delta^+ - \delta^-}{\delta^+ + \delta^-}$

Generalized LVQ (cont.)

- update rule for GLVQ: calculate the partial derivatives

$$e((\vec{x}, d); \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\}) = f_{log}\left(\frac{\delta^+ - \delta^-}{\delta^+ + \delta^-}\right)$$

denote with $\vec{w}^{(+)}$ the closest prototype of class d and $\vec{w}^{(-)}$ the closest prototype of other class

Generalized LVQ (cont.)

- update rule for GLVQ: calculate the partial derivatives

$$e((\vec{x}, d); \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\}) = f_{log}\left(\frac{\delta^+ - \delta^-}{\delta^+ + \delta^-}\right)$$

denote with $\vec{w}^{(+)}$ the closest prototype of class d and $\vec{w}^{(-)}$ the closest prototype of other class

$$\frac{\partial e}{\partial w_i^{(+)}} = f'_{log}\left(\frac{\delta^+ - \delta^-}{\delta^+ + \delta^-}\right) \cdot \frac{\frac{\partial \delta^+}{\partial w_i^{(+)}} \cdot (\delta^+ + \delta^-) - (\delta^+ - \delta^-) \cdot \frac{\partial \delta^+}{\partial w_i^{(+)}}}{(\delta^+ + \delta^-)^2}$$

$$\frac{\partial \delta^+}{\partial w_i^{(+)}} = -(x_i - w_i^{(+)})$$

$$f'_{log}(z) = f_{log}(z)(1 - f_{log}(z))$$

Generalized LVQ (cont.)

$$\frac{\partial e}{\partial w_i^{(-)}} = f'_{log}\left(\frac{\delta^+ - \delta^-}{\delta^+ + \delta^-}\right) \cdot \frac{-\frac{\partial \delta^-}{\partial w_i^{(-)}} \cdot (\delta^+ + \delta^-) - (\delta^+ - \delta^-) \cdot \frac{\partial \delta^-}{\partial w_i^{(-)}}}{(\delta^+ + \delta^-)^2}$$

$$\frac{\partial \delta^-}{\partial w_i^{(-)}} = -(x_i - w_i^{(-)})$$

Generalized LVQ (cont.)

$$\frac{\partial e}{\partial w_i^{(-)}} = f'_{log}\left(\frac{\delta^+ - \delta^-}{\delta^+ + \delta^-}\right) \cdot \frac{-\frac{\partial \delta^-}{\partial w_i^{(-)}} \cdot (\delta^+ + \delta^-) - (\delta^+ - \delta^-) \cdot \frac{\partial \delta^-}{\partial w_i^{(-)}}}{(\delta^+ + \delta^-)^2}$$

$$\frac{\partial \delta^-}{\partial w_i^{(-)}} = -(x_i - w_i^{(-)})$$

gradient descent update (negative gradient):

$$\vec{w}^{(+)} \leftarrow \vec{w}^{(+)} + \epsilon f'_{log}\left(\frac{\delta^+ - \delta^-}{\delta^+ + \delta^-}\right) \frac{2\delta^-}{(\delta^+ + \delta^-)^2}(\vec{x} - \vec{w}^{(+)})$$

$$\vec{w}^{(-)} \leftarrow \vec{w}^{(-)} - \epsilon f'_{log}\left(\frac{\delta^+ - \delta^-}{\delta^+ + \delta^-}\right) \frac{2\delta^+}{(\delta^+ + \delta^-)^2}(\vec{x} - \vec{w}^{(-)})$$

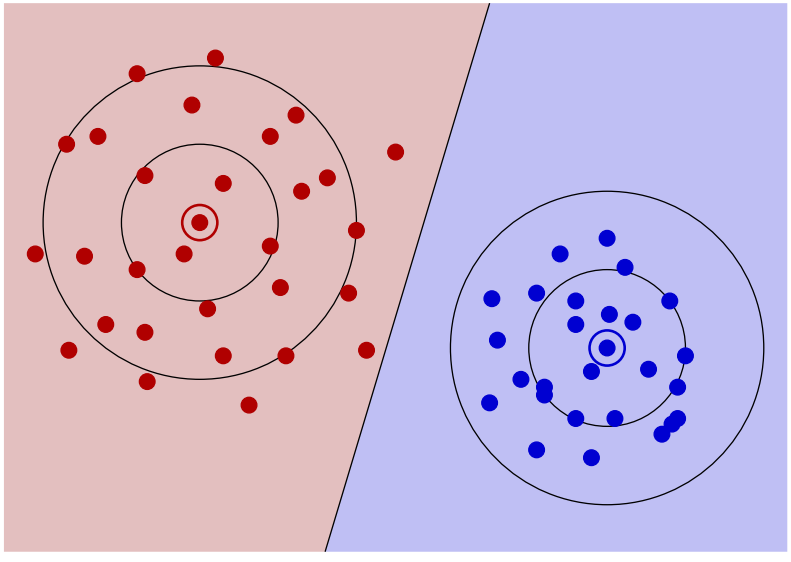
ϵ: learning rate

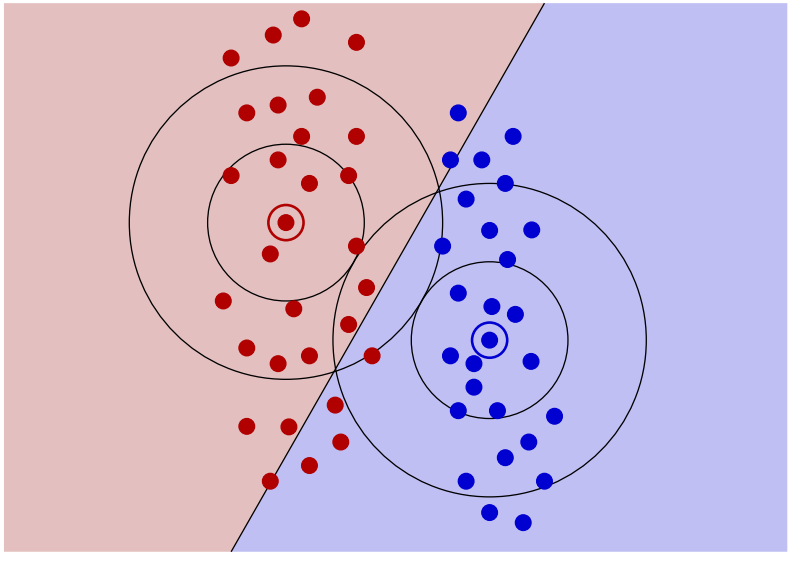
Generalized LVQ (cont.)

- Generalized LVQ (GLVQ) (Sato&Yamada 1995)
- even more general generic form that allows a wide variety of error functions. The presented one is the form most frequently used

Relevance learning

- results of LVQ/GLVQ heavily depend on scaling of the data and the distance function used
- euclidean distance prefers circular Voronoi cells





works well for round clusters but not well for elongated clusters = round clusters after rescaling

Relevance learning (cont.)

- Relevance learning: adapt the distance measure during learning

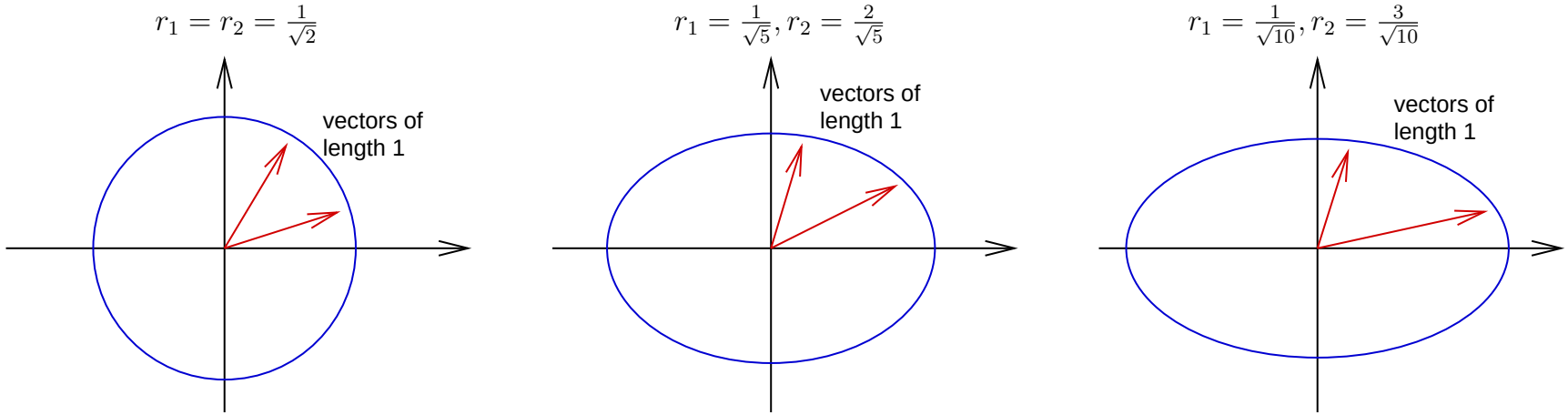
  (Bojer, Hammer, Schunk, Tluk von Toschanowitz 2001)

- idea: introduce weights for each dimension

$$||\vec{z}||^2_{\vec{r}} = \sum_{i=1}^{d} (r_i z_i^2)$$

  $\vec{r}$ is the vector of relevances, $r_i \geq 0$, $\sum_{i=1}^{n} r_i^2 = 1$

- the larger r_i, the more important the i-th dimension becomes



Relevance learning (cont.)

- relevance learning: update relevance vector $\vec{r}$ by gradient descent:

$$r_i \leftarrow \max\{0, r_i - \epsilon_r \frac{\partial e((\vec{x}, d); \{\vec{w}^{(1)}, \ldots, \vec{w}^{(m)}\})}{\partial r_i}\}$$

  $\epsilon_r > 0$ is learning rate, decreasing.
  $\epsilon_r \neq \epsilon$!
  to meet the condition $\sum_{i=1}^{d} r_i^2 = 1$, we have to rescale $\vec{r}$ after each update:

$$\vec{r} \leftarrow \frac{1}{\sqrt{\sum_{i=1}^{d} r_i^2}} \vec{r}$$

- we get two interleaved gradient descent processes:
  - updating the prototype positions using learning rate ϵ
  - updating the relevance vector $\vec{r}$ using learning rate ϵ_r

Relevance learning (cont.)

- relevance learning can be combined with any supervised WTAN approach:
  - RLVQ: relevance learning combined with LVQ
  - GRLVQ: relevance learning combined with GLVQ

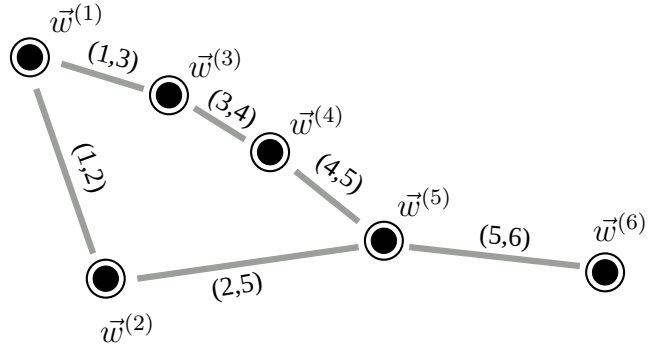
Structure learning

- using VQ we find a set of representatives for all patterns
- we lose information about neighborhood between patterns/prototypes

  $\Rightarrow$ structure learning: learn prototypes and the spatial neighborhood between prototypes

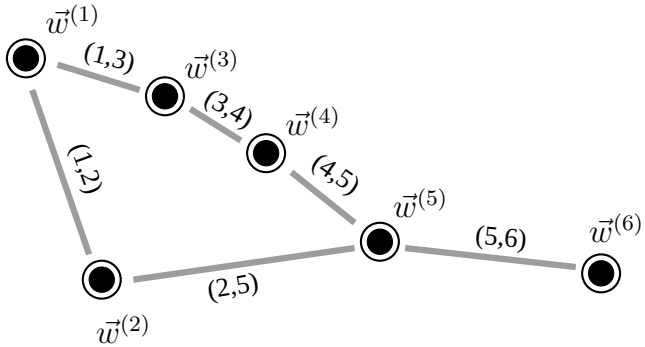
Structure learning (cont.)

- add neighborhood relationship of prototypes
  - set of edges $C \subseteq P \times P$ (P: set of prototypes, C set of edges)



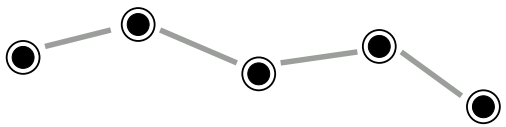
Structure learning (cont.)

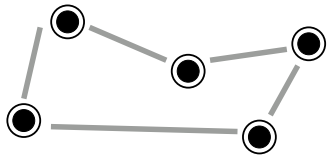
- add neighborhood relationship of prototypes
  - set of edges $C \subseteq P \times P$ (P: set of prototypes, C set of edges)



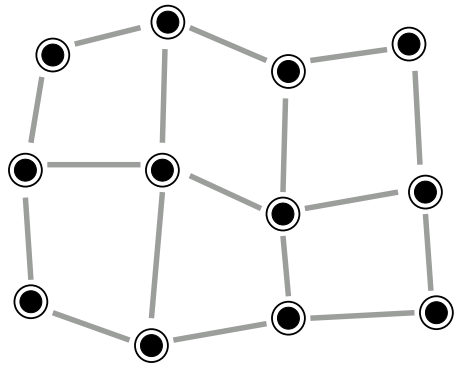
  - typical neighborhood structures:

chain

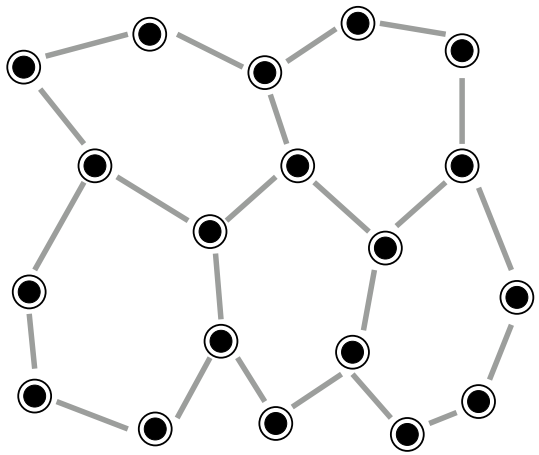




ring



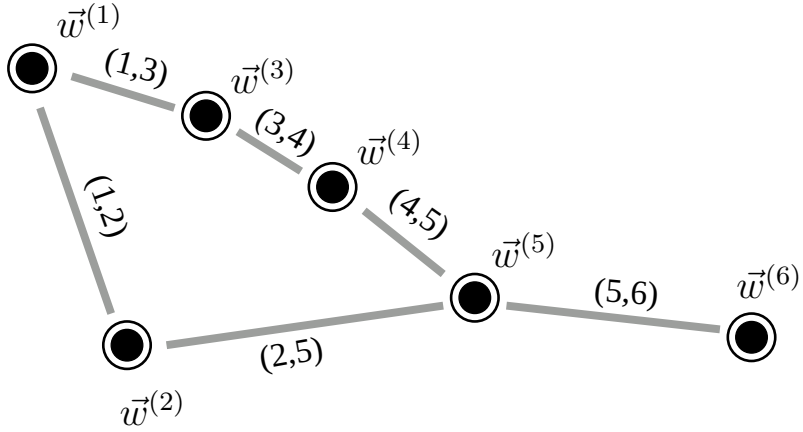
2−dim. rectangular grid



2−dim. hexagonal grid

Structure learning (cont.)

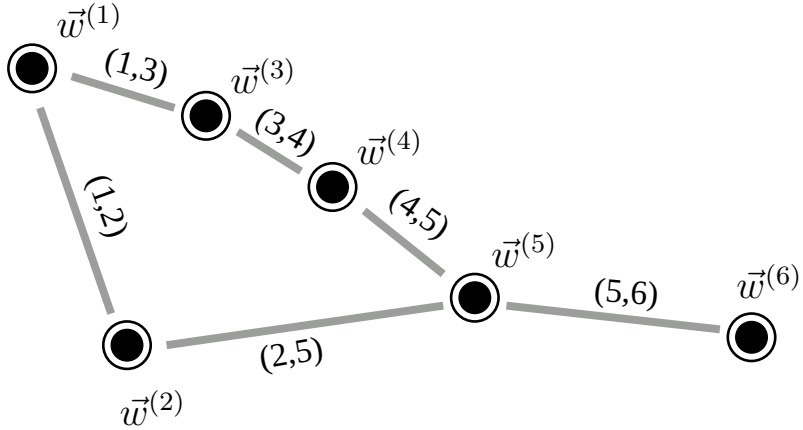
- direct and indirect neighborhood of prototypes



neighborhood distance δ between two protoypes: count the number of edges on the shortest path between both prototypes. e.g.:

Structure learning (cont.)

- direct and indirect neighborhood of prototypes



neighborhood distance δ between two protoypes: count the number of edges on the shortest path between both prototypes. e.g.:

$$\delta(\vec{w}^{(1)}, \vec{w}^{(2)}) = 1$$
$$\delta(\vec{w}^{(1)}, \vec{w}^{(4)}) = 2$$
$$\delta(\vec{w}^{(1)}, \vec{w}^{(5)}) = 2$$
$$\delta(\vec{w}^{(2)}, \vec{w}^{(3)}) = 2$$
$$\delta(\vec{w}^{(1)}, \vec{w}^{(1)}) = 0$$

Self organizing maps

- unsupervised learning with a given topology
- self organizing maps (SOM), Kohonen maps (Kohonen 1982)

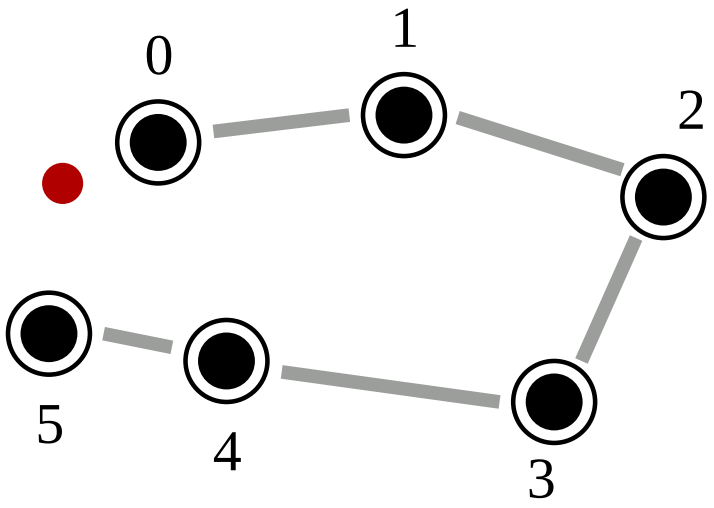
1: start with given topology
2: **loop**
3: **for all** $\vec{x} \in \mathcal{D}$ **do**
4: calculate closest prototype k
5: **for all** prototypes $\vec{w}^{(j)}$ **do**
6: $\vec{w}^{(j)} \leftarrow \vec{w}^{(j)} + \epsilon e^{-\frac{\delta(j,k)}{\lambda}} \left(\vec{x} - \vec{w}^{(j)}\right)$
7: **end for**
8: **end for**
9: **end loop**

$\epsilon > 0$ is the learning rate, decreasing, $\lambda > 0$ parameter, decreasing

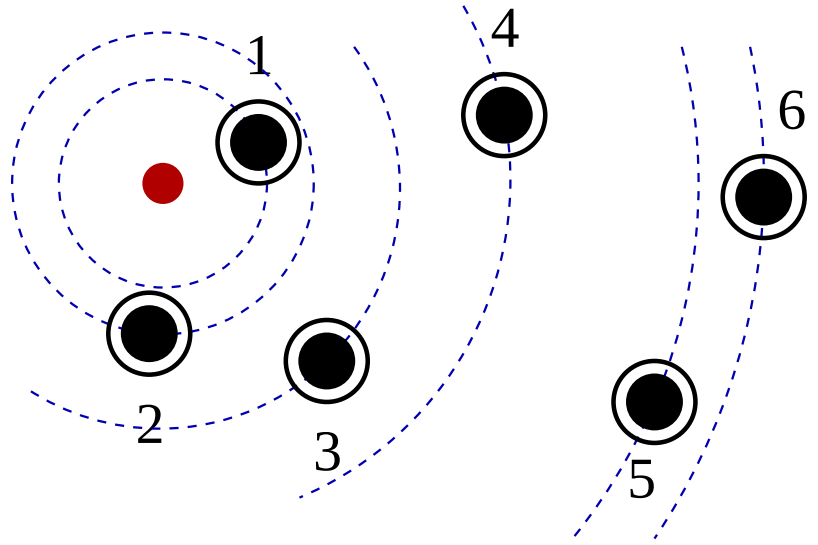
in (6) it is possible to replace $e^{-\frac{\delta(j,k)}{\lambda}}$ by any positive, decreasing function

Self organizing maps (cont.)

- comparison: SOM and Neural gas
  - both approaches push all prototypes depending on their distance to the winning prototype, but:
  - SOM uses the neighborhood distance δ on the predefined topology
  - Neural gas uses the Euclidean distance $||\cdot||$ of the prototype vectors



neighborhood distance



ranking of Euclidean distance

Self organizing maps (cont.)

- Neural gas adapts better to the data
- SOM forces the prototypes in a predefined structure
- SOM can be understood as embedding a structure into a pattern space
- dimension of structure may differ from dimension of pattern space

Growing neural gas

- SOM: topology does not adapt to the data
- Neural gas: uses ranks but does not create a topology

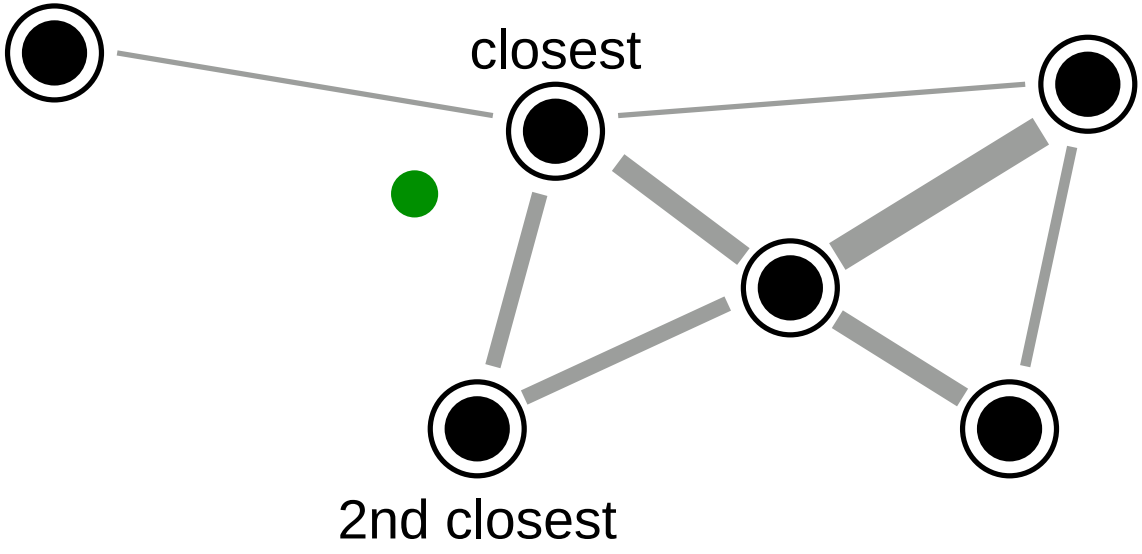
Growing neural gas

- SOM: topology does not adapt to the data
- Neural gas: uses ranks but does not create a topology

  $\Rightarrow$ Growing Neural Gas (Fritzke 1994) combines both
  - creates topology data dependent
  - adds and prunes prototypes
  - bottom-up approach: start with two prototypes and add new prototypes periodically

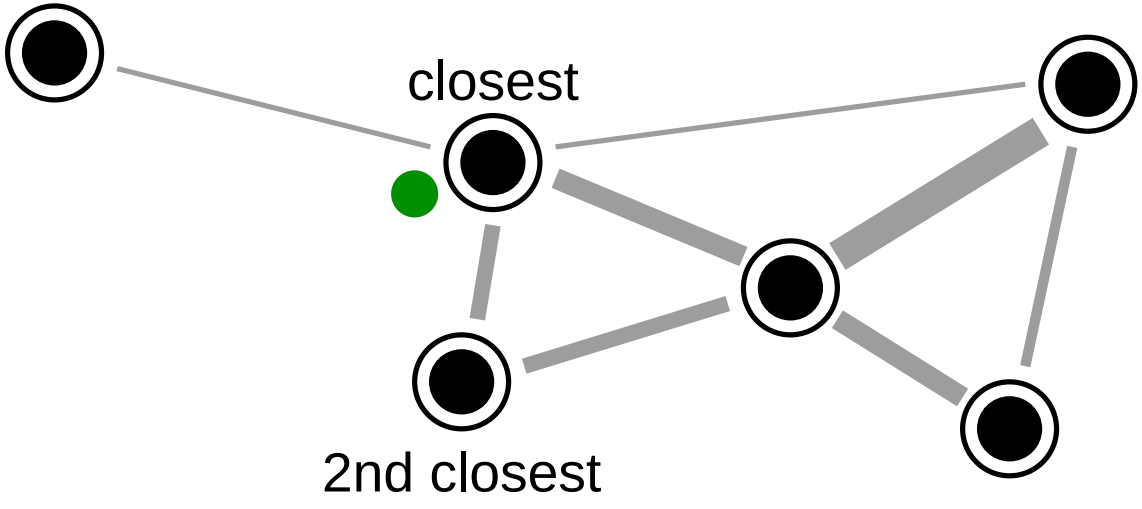
Growing neural gas (cont.)

- moving closest and second closest prototype towards pattern



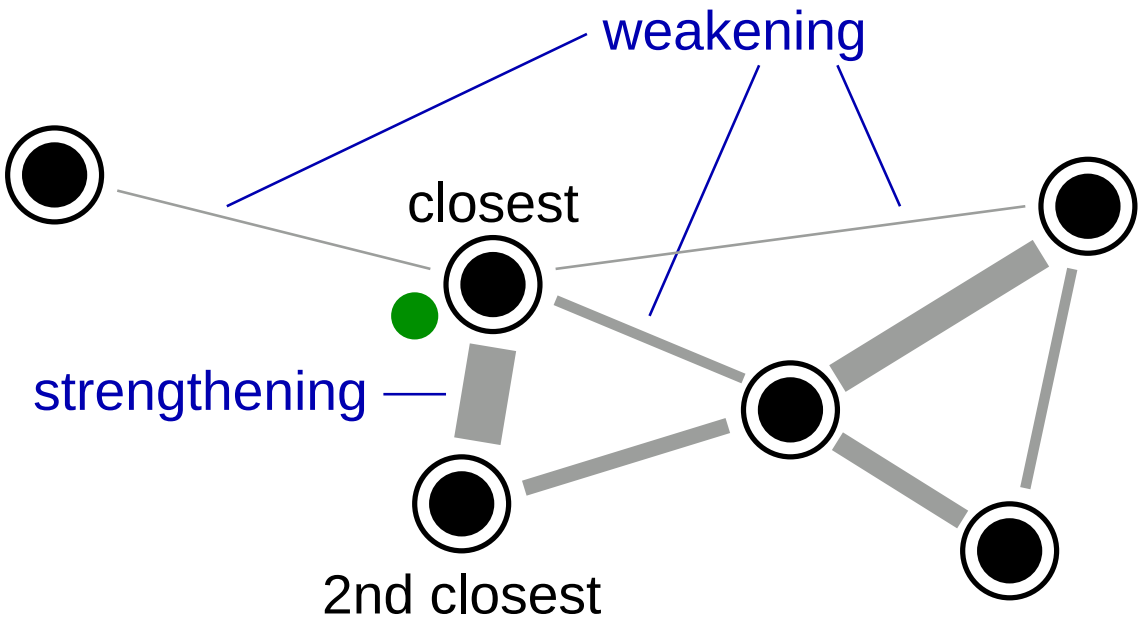
Growing neural gas (cont.)

- moving closest and second closest prototype towards pattern



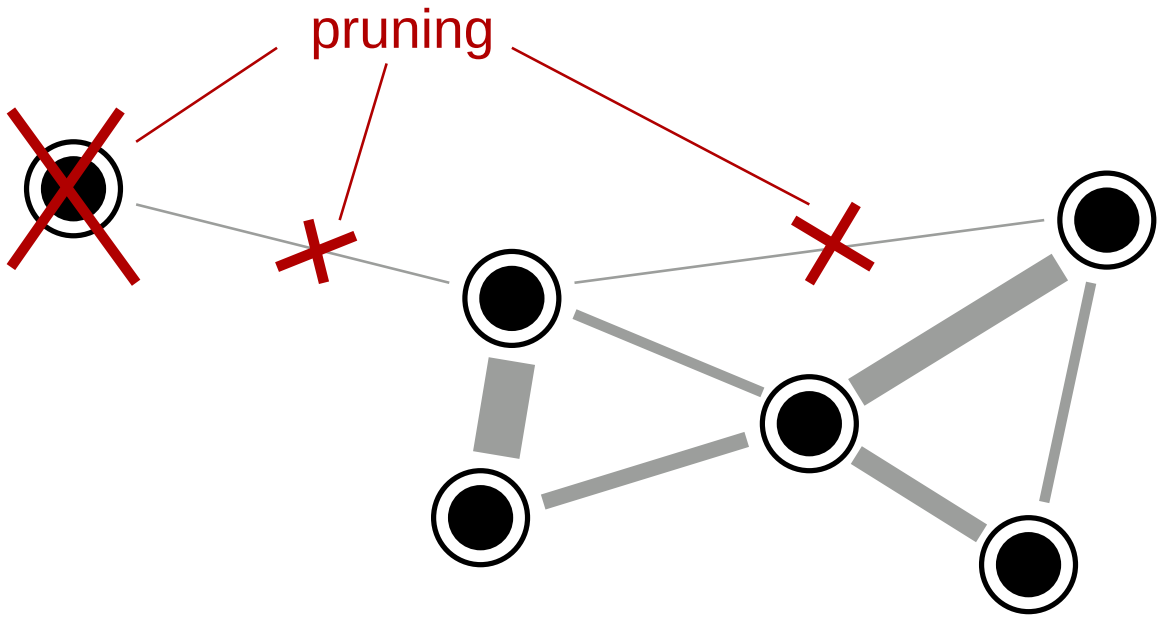
Growing neural gas (cont.)

- moving closest and second closest prototype towards pattern
- strengthen connection between closest prototypes, weaken connections between closest prototypes and prototypes far away



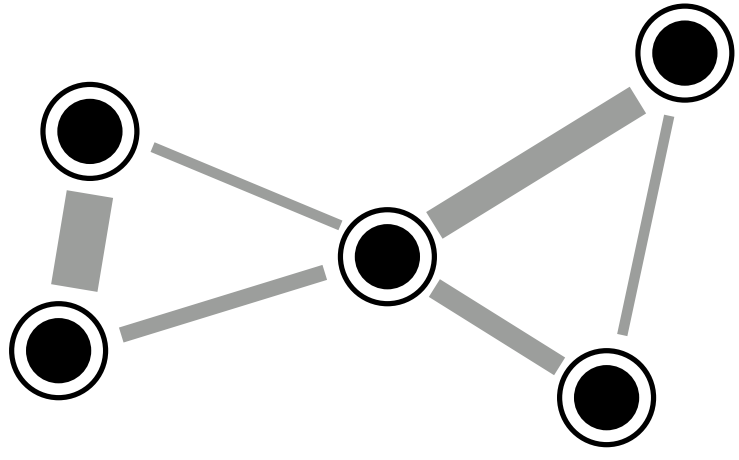
Growing neural gas (cont.)

- moving closest and second closest prototype towards pattern
- strengthen connection between closest prototypes, weaken connections between closest prototypes and prototypes far away
- prune weak connections, prune isolated protoypes



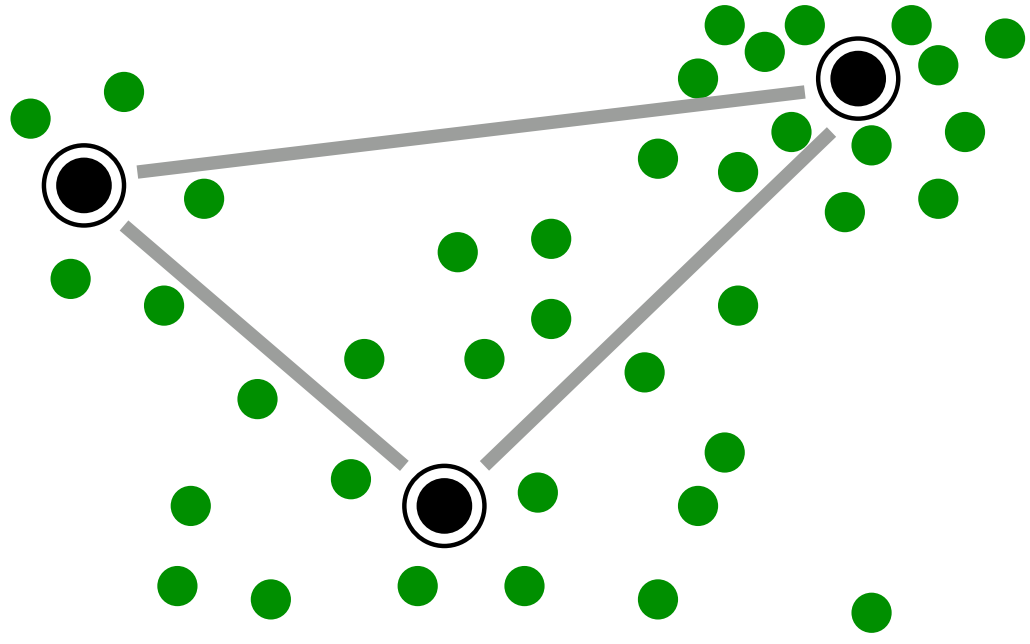
Growing neural gas (cont.)

- moving closest and second closest prototype towards pattern
- strengthen connection between closest prototypes, weaken connections between closest prototypes and prototypes far away
- prune weak connections, prune isolated protoypes



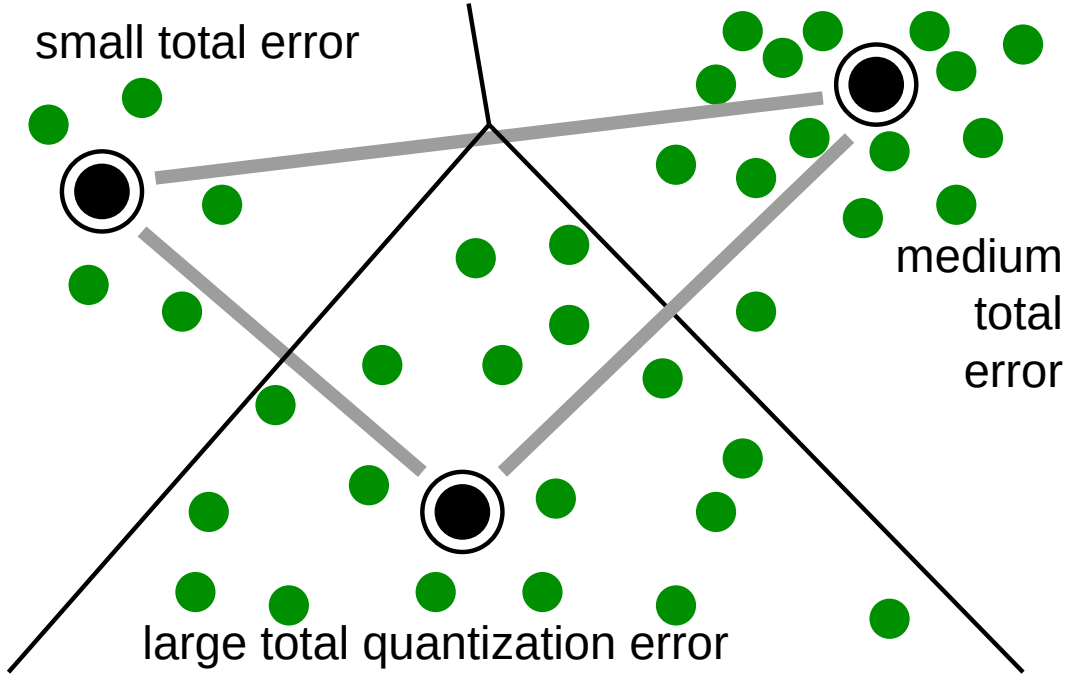
Growing neural gas (cont.)

- adding prototypes:
  - every prototype is assigned with a (local) quantization error
  - adding protoypes makes sense in areas of large quantization error



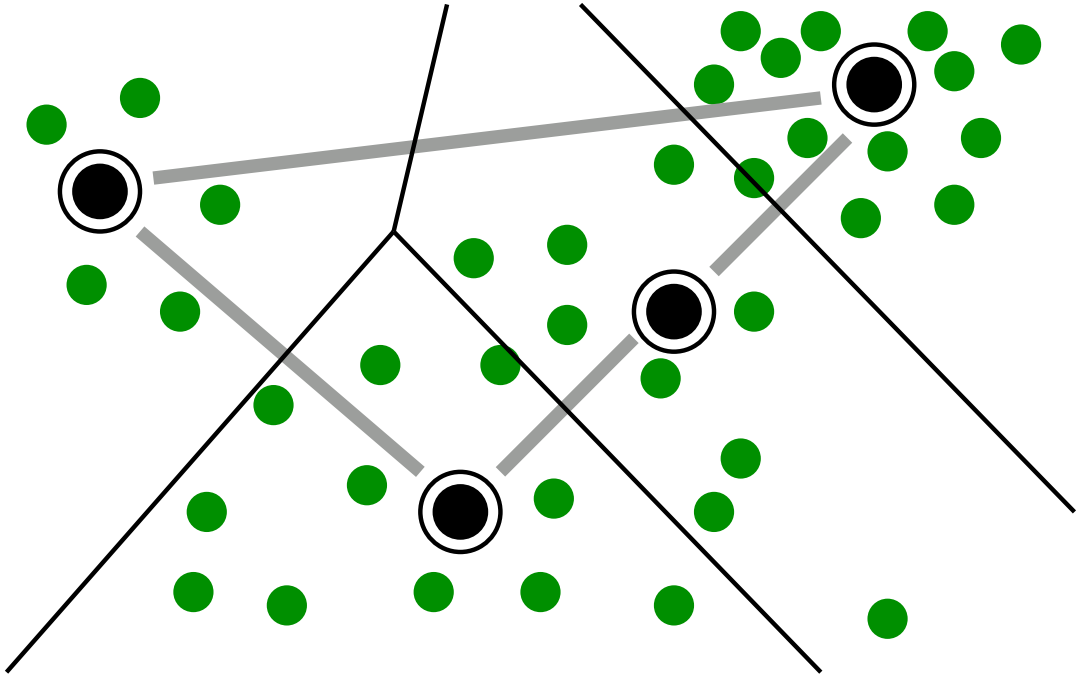
Growing neural gas (cont.)

- adding prototypes:
  - every prototype is assigned with a (local) quantization error
  - adding protoypes makes sense in areas of large quantization error



Growing neural gas (cont.)

- adding prototypes:
  - every prototype is assigned with a (local) quantization error
  - adding protoypes makes sense in areas of large quantization error
  - a prototype is added halfway between prototype with largest error and adjacent prototype with second largest error



Growing neural gas (cont.)

- bringing together these ideas yields the Growing neural gas algorithm
- variables:
  - set of prototypes P, prototype vectors are denoted $\vec{w}^{(i)}$
  - set of connections C: $C \subseteq P \times P$
  - age function that assigns to each edge in C a number: $age : C \rightarrow \mathbb{N}_0$
  - local error functions L that assign to each prototype a real number: $L : P \rightarrow \mathbb{R}$
  - some parameters of the algorithm: learning rates $\epsilon_1 \geq \epsilon_2 > 0$,maximal age $a_{max} \in \mathbb{N}$, real numbers α, β: $0 \leq \alpha, \beta \leq 1$

Growing neural gas (cont.)

- main loop

Require: $|P| = 2, C = \emptyset, L(1) = L(2) = 0$

1: **loop**
2: **for all** patterns $\vec{x} \in \mathcal{D}$ **do**
3: determine closest prototype s_1 to $\vec{x}$ and second closest prototype s_2
4: update weight vector s_1: $\vec{w}^{(s_1)} \leftarrow \vec{w}^{(s_1)} + \epsilon_1\big(\vec{x} - \vec{w}^{(s_1)}\big)$
5: update weight vector s_2: $\vec{w}^{(s_2)} \leftarrow \vec{w}^{(s_2)} + \epsilon_2\big(\vec{x} - \vec{w}^{(s_2)}\big)$
6: call procedure update_edge$\big(s_1, s_2\big)$
7: call procedure update_age$\big(s_1\big)$
8: update local error of s_1: $L\big(s_1\big) \leftarrow L\big(s_1\big) + \big|\big|\vec{x} - \vec{w}^{(s_1)}\big|\big|^2$
9: call procedure decay_local_error()
10: **end for**
11: sometimes call procedure add_prototype()
12: **end loop**

Growing neural gas (cont.)

- procedure update_edge(s_1, s_2)

  1: create edge from s_1 to s_2 if it does not exist: $C \leftarrow C \cup \{(s_1, s_2)\}$
  2: reset age of connection $age(s_1, s_2) \leftarrow 0$

- procedure update_age(s_1)

  1: **for all** $i \in P | (s_1, i) \in C$ **do**
  2: increment age of connections: $age(s_1, i) \leftarrow age(s_1, i) + 1$
  3: **if** $age(s_1, i) > a_{max}$ **then**
  4: remove edge: $C \leftarrow C \setminus \{(s_1, i)\}$
  5: **if** prototype i has become isolated **then**
  6: remove prototype i: $P \leftarrow P \setminus \{i\}$
  7: **end if**
  8: **end if**
  9: **end for**

Growing neural gas (cont.)

- procedure decay_local_error()

  1: **for all** $p \in P$ **do**
  2: decrease local error: $L(p) \leftarrow (1 - \beta)L(p)$
  3: **end for**

Growing neural gas (cont.)

- procedure add_prototype()
  1: determine prototype with largest local error: $p_1 \leftarrow \arg\max_{p \in P} L(p)$
  2: determine adjacent prototype of p_1 with largest local error: $p_2 \leftarrow \arg\max_{p|(p,p_1) \in C} L(p)$
  3: create new prototype q with prototype vector $\vec{w}^{(q)}$: $P \leftarrow P \cup \{q\}$
  4: set new prototype vector halfway between p_1 and p_2: $\vec{w}^{(q)} \leftarrow \frac{1}{2}(\vec{w}^{(p_1)} + \vec{w}^{(p_2)})$
  5: replace link between p_1 and p_2 by links between q and p_1, p_2: $C \leftarrow (C \setminus \{(p_1, p_2)\}) \cup \{(p_1, q), (p_2, q)\}$
  6: update local error: $L(p_1) \leftarrow (1 - \alpha) L(p_1)$
  7: update local error: $L(p_2) \leftarrow (1 - \alpha) L(p_2)$
  8: update local error: $L(q) \leftarrow \frac{1}{2}(L(p_1) + L(p_2))$

Growing neural gas (cont.)

- there is not much theory on GNG
- we can presume that
  - an error term exists which is minimized as long as no prototypes are created or removed
  - a batch algorithm can be built which works somehow similar to GNG and that may simplify calculations, especially the calculation of local errors
- varying the model size opens all problems of flexible models: overfitting, model selection, theoretical problems
- adjusting the parameters of GNG may need much experience
- nonetheless, GNG can be used successfully in practice

Survey of methods

- table of WTAN methods discussed in this lecture:

method	type	error function?	batch learning method?
VQ	unsupervised	exists	k-means
NG	unsupervised	exists	similar to k-means
LVQ	supervised	not sensefull	no
GLVQ	supervised	exists	no
SOM	structure learning (fixed topology)	exists	similar to k-means
GNG	structure learning (flexible topology)	maybe	maybe

each of the supervised methods can be combined with relevance learning to adapt the distance measure (e.g. RLVQ, GRLVQ)

Survey of methods (cont.)

- area of ongoing research activities

Survey of methods (cont.)

- area of ongoing research activities
- many other methods and variants exist like
  - neural gas for supervised learning
  - approaches that add and prune prototypes
  - approaches that can be used to learn sequential data (e.g. timeseries)
  - approaches that converge quicker/are more robust w.r.t. intitialization
  - prototype individual learning rate (OLVQ)
  - heuristics to control the learning rate

Survey of methods (cont.)

- area of ongoing research activities
- many other methods and variants exist like
  - neural gas for supervised learning
  - approaches that add and prune prototypes
  - approaches that can be used to learn sequential data (e.g. timeseries)
  - approaches that converge quicker/are more robust w.r.t. intitialization
  - prototype individual learning rate (OLVQ)
  - heuristics to control the learning rate
- prototypes are also often called codebook vectors. The set of prototypes is called a codebook

Survey of methods (cont.)

- area of ongoing research activities
- many other methods and variants exist like
  - neural gas for supervised learning
  - approaches that add and prune prototypes
  - approaches that can be used to learn sequential data (e.g. timeseries)
  - approaches that converge quicker/are more robust w.r.t. intitialization
  - prototype individual learning rate (OLVQ)
  - heuristics to control the learning rate
- prototypes are also often called codebook vectors. The set of prototypes is called a codebook
- Important: results of prototype based methods depend heavily on scaling of data and on the distance measure used. Be aware of it!

Application examples

- color quantization
- similarity of objects
- motor maps

Application examples (cont.)

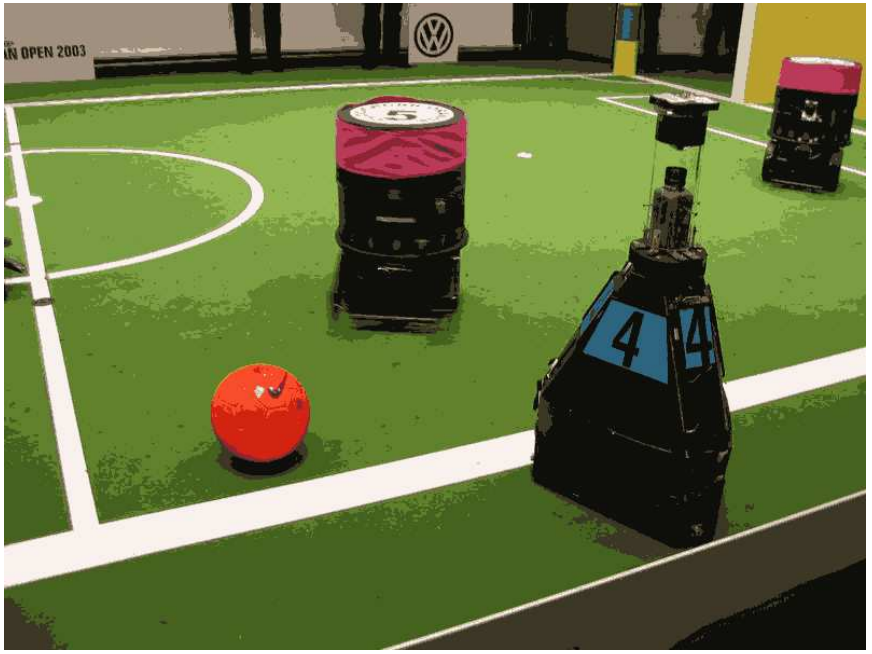
- color quantization: given a picture, find a small number of representative colors within the picture
- each pixel yields one 3-dim. pattern (RGB color values)

original image

reduced color (16 different colors) using k-means

Application examples (cont.)

original image



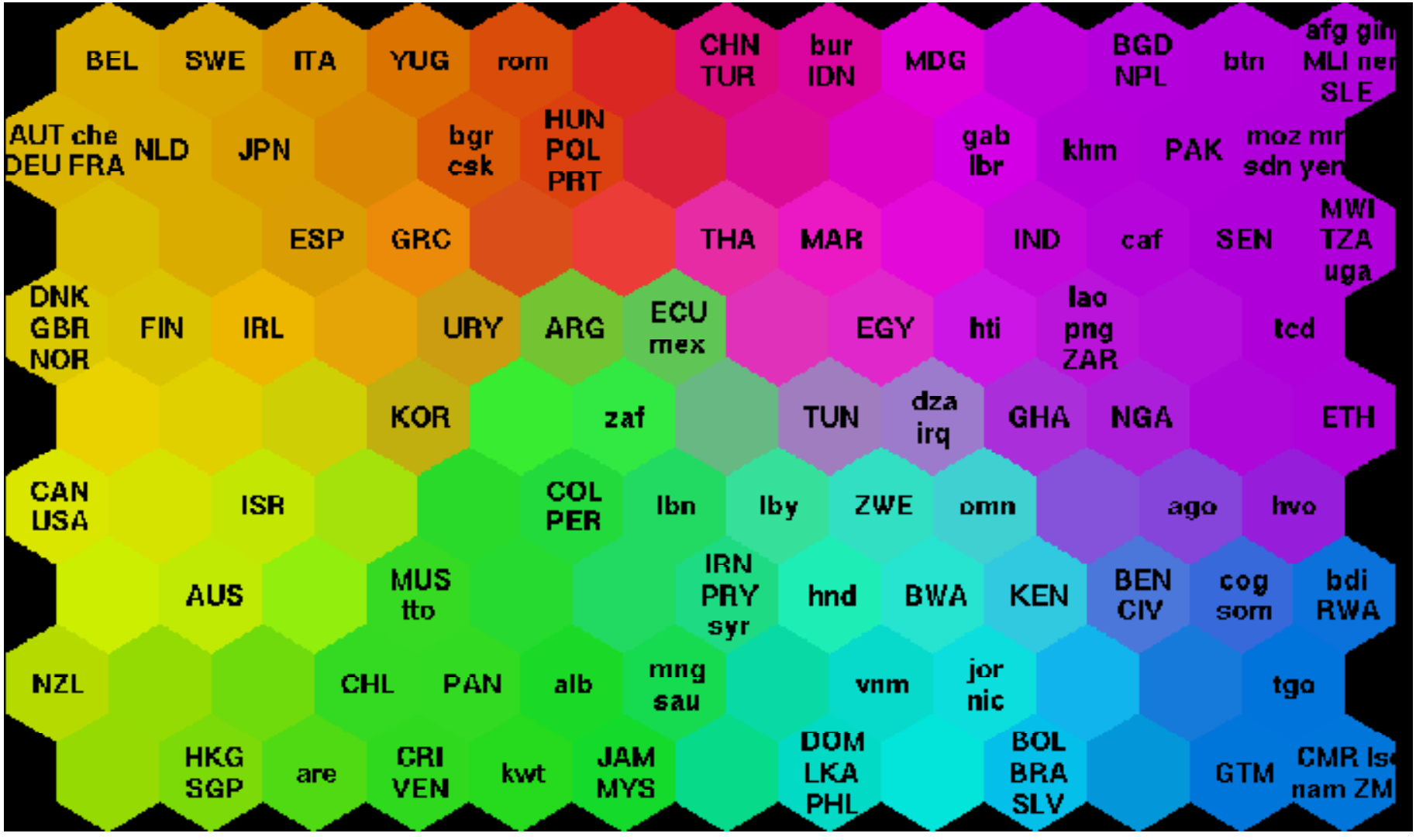
reduced color (16 different colors)

- application domains
  - satellite image analysis
  - image and video compression

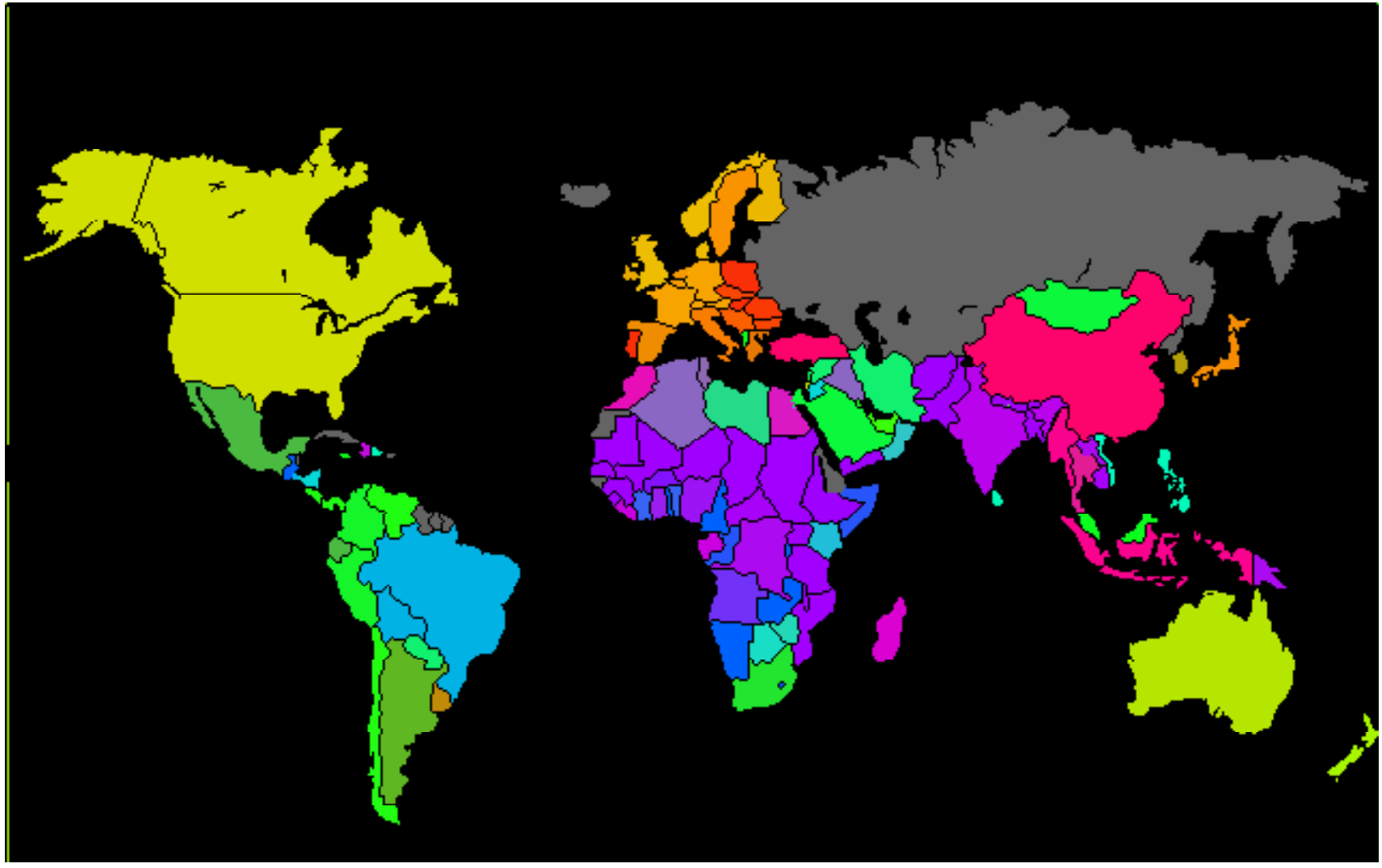
Application examples (cont.)

- similarities of objects: given a set of objects described by numeric vectors, e.g. the countries of the world described by economical, cultural and political figures. Find a grouping of theses countries and a neighborhood relationship (concerning the economical, political and cultural situation, not the geographical position)
- train a SOM on these data
- if you want to determine how similar different countries are:
  - look for the closest prototype for each country
  - determine the neighborhood distance of the prototypes
- similar applications: document retrievel (similarity of text documents, images, etc.)
- more examples can be found of Kohonen's website: http://www.cis.hut.fi/

Application examples (cont.)

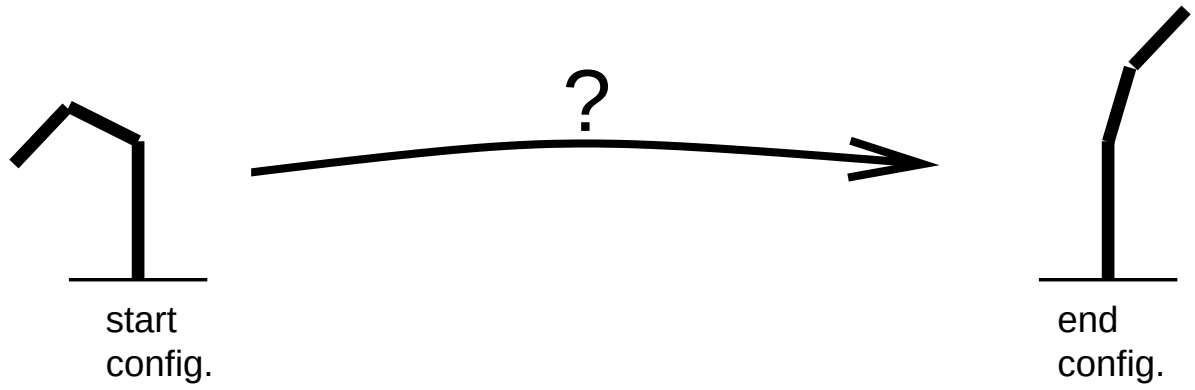


Application examples (cont.)



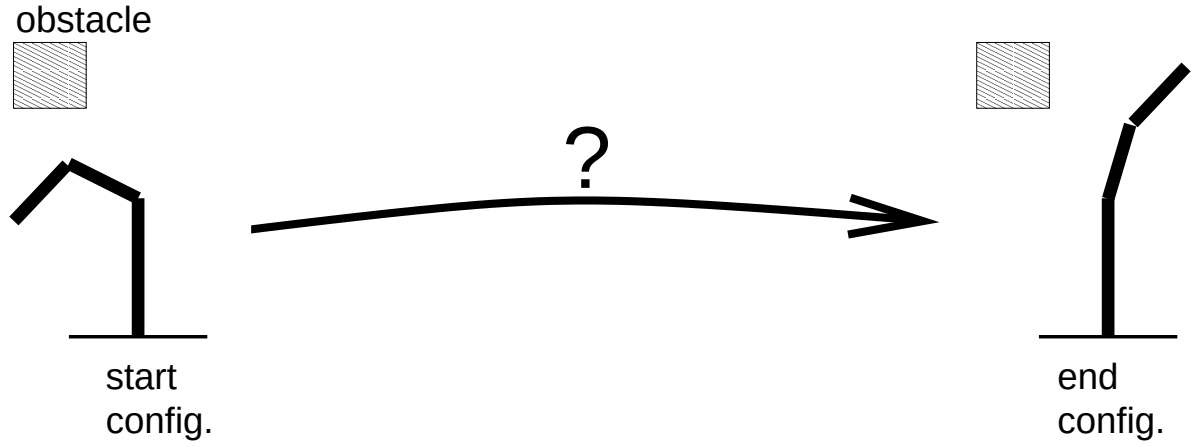
Application examples (cont.)

- motor maps: a map that refers to the cofigurations of a robot



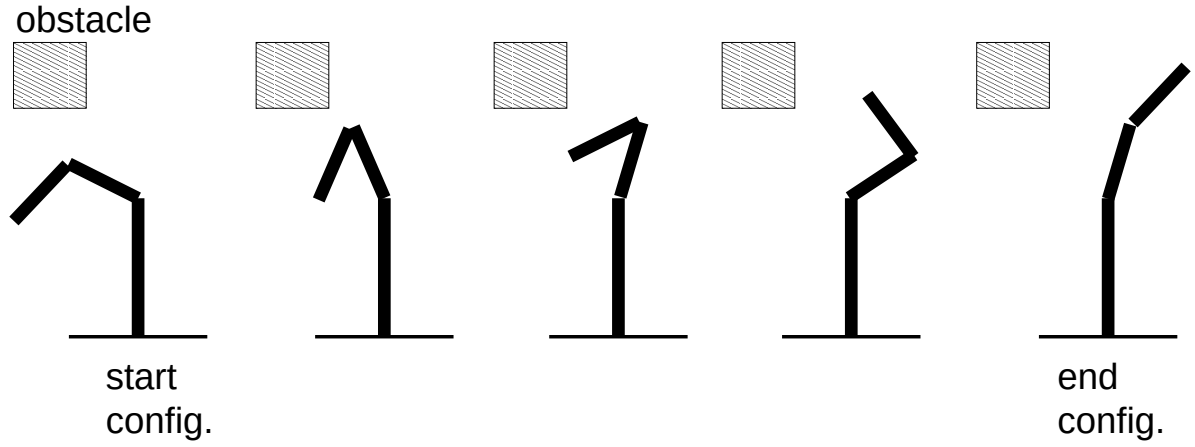
Application examples (cont.)

- motor maps: a map that refers to the cofigurations of a robot



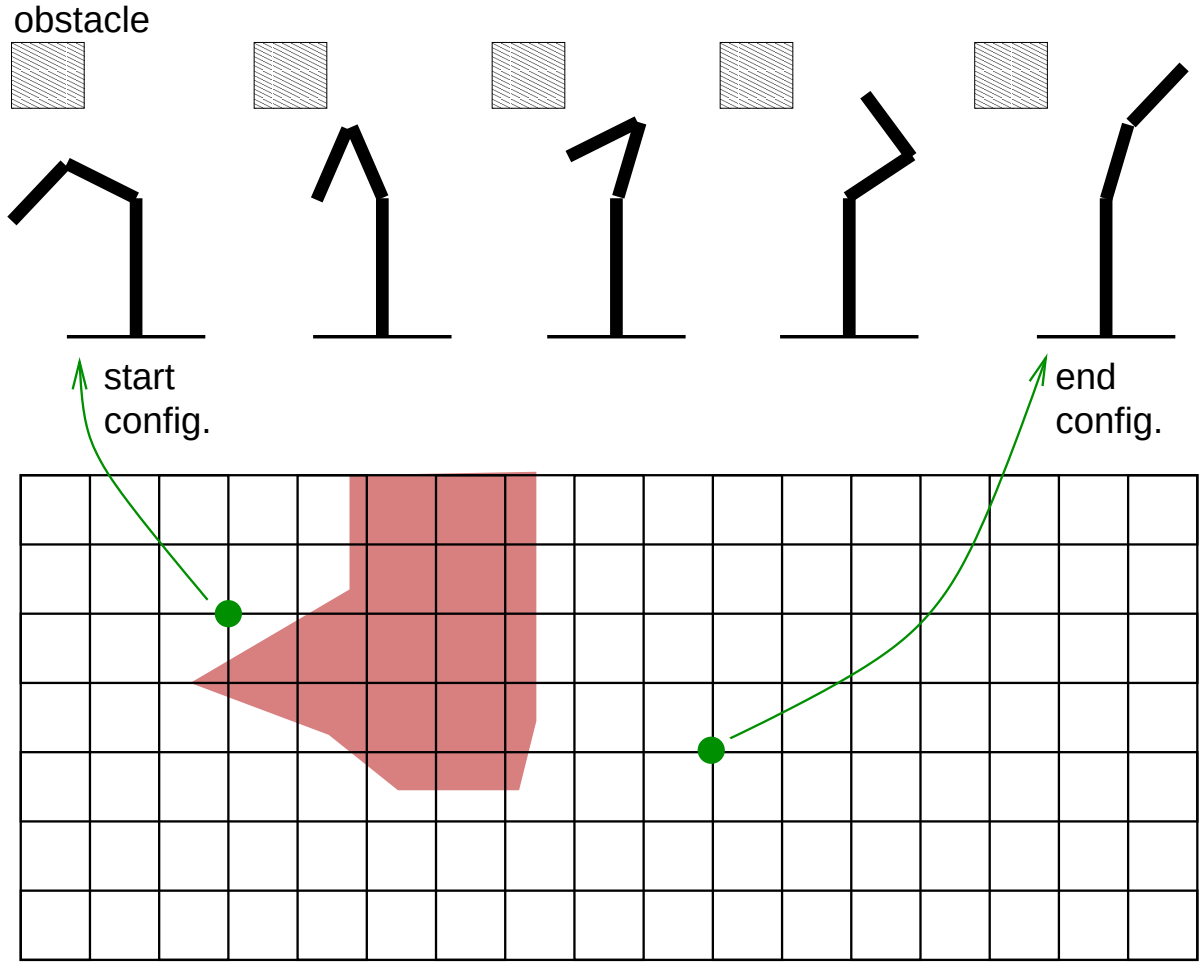
Application examples (cont.)

- motor maps: a map that refers to the cofigurations of a robot



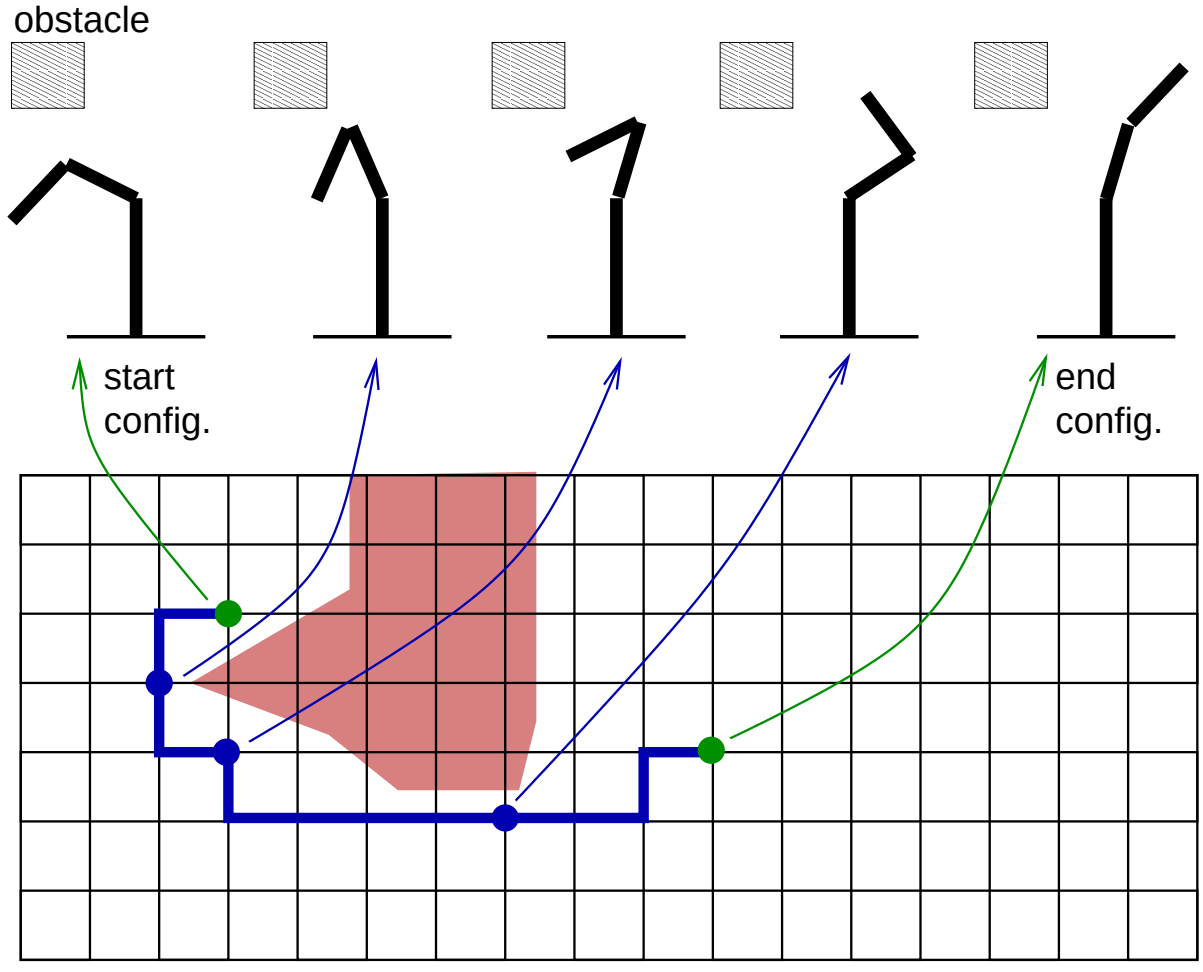
Application examples (cont.)

- motor maps: a map that refers to the cofigurations of a robot



Application examples (cont.)

- motor maps: a map that refers to the cofigurations of a robot



shortest path through motor map without collision

simplification of path planning, working in finite SOM grid instead of infinite space of configurations

Last Slide