Neuronale Netze

Prof. Dr. Rudolf Kruse

Computational Intelligence
Institut für Intelligente Kooperierende Systeme
Fakultät für Informatik
rudolf.kruse@ovgu.de

Lernende Vektorquantisierung

(engl. Learning Vector Quantization)

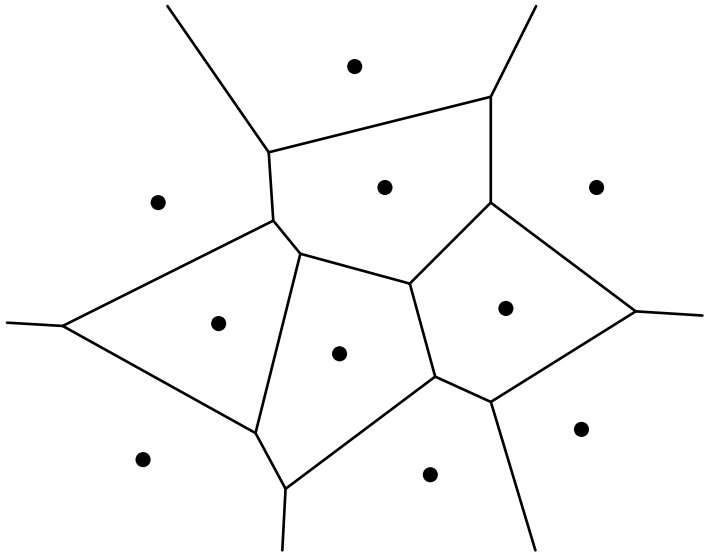
Motivation

Bisher: festes Lernen, jetzt freies Lernen, d.h. es existieren keine festgelegten Klassenlabels oder Zielwerte für jedes Lernbeispiel

Grundidee: ähnliche Eingaben führen zu ähnlichen Ausgaben

Ähnlichkeit zum Clustering: benachbarte (ähnliche) Datenpunkte im Eingaberaum liegen auch im Ausgaberaum benachbart

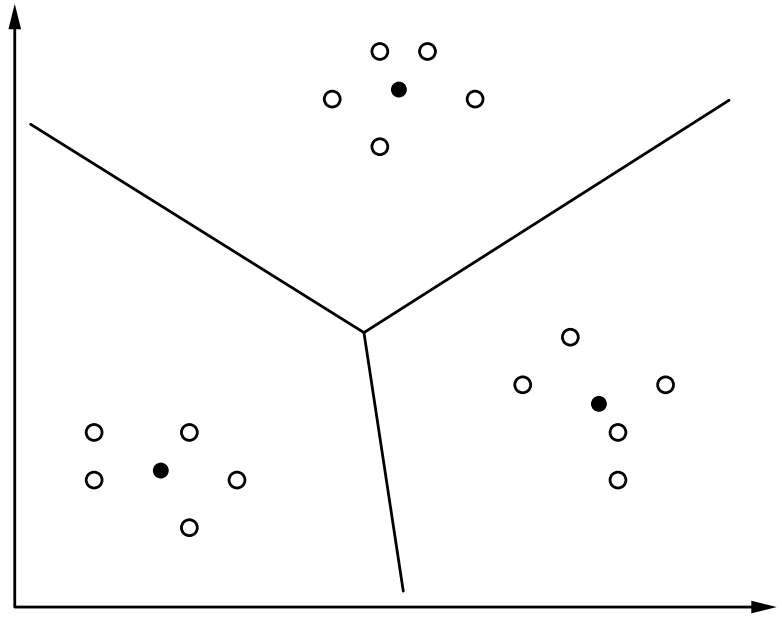
Voronoidiagramm einer Vektorquantisierung



Punkte repräsentieren Vektoren, die zur Quantisierung der Fläche genutzt werden.

Linien sind die Grenzen der Regionen, deren Punkte am nächsten zu dem dargestellten Vektor liegen.

Finden von Clustern in einer gegebenen Menge von Punkten



Datenpunkte werden durch leere Kreise dargestellt (○).

Clusterzentren werden durch gefüllte Kreise dargestellt (●).

Lernende Vektorquantisierung, Netzwerk

Ein **Lernendes Vektorquantisierungsnetzwerk (LVQ)** ist ein neuronales Netz mit einem Graphen $G = (U, C)$, das die folgenden Bedingungen erfüllt:

(i) $U_{\text{in}} \cap U_{\text{out}} = \emptyset$, $U_{\text{hidden}} = \emptyset$

(ii) $C = U_{\text{in}} \times U_{\text{out}}$

Die Netzeingabefunktion jedes Ausgabeneurons ist eine **Abstandsfunktion** zwischen Eingabe- und Gewichtsvektor, d.h.

$$\forall u \in U_{\text{out}}: \qquad f_{\text{net}}^{(u)}(\vec{w}_u, \vec{\text{in}}_u) = d(\vec{w}_u, \vec{\text{in}}_u),$$

wobei $d : \mathbb{R}^n \times \mathbb{R}^n \to \mathbb{R}_0^+$ eine Funktion ist, die $\forall \vec{x}, \vec{y}, \vec{z} \in \mathbb{R}^n$:

$$\begin{array}{lll} (i) & d(\vec{x}, \vec{y}) = 0 \quad \Leftrightarrow \quad \vec{x} = \vec{y}, & \\ (ii) & d(\vec{x}, \vec{y}) = d(\vec{y}, \vec{x}) & \text{(Symmetrie)}, \\ (iii) & d(\vec{x}, \vec{z}) \leq d(\vec{x}, \vec{y}) + d(\vec{y}, \vec{z}) & \text{(Dreiecksungleichung)} \end{array}$$

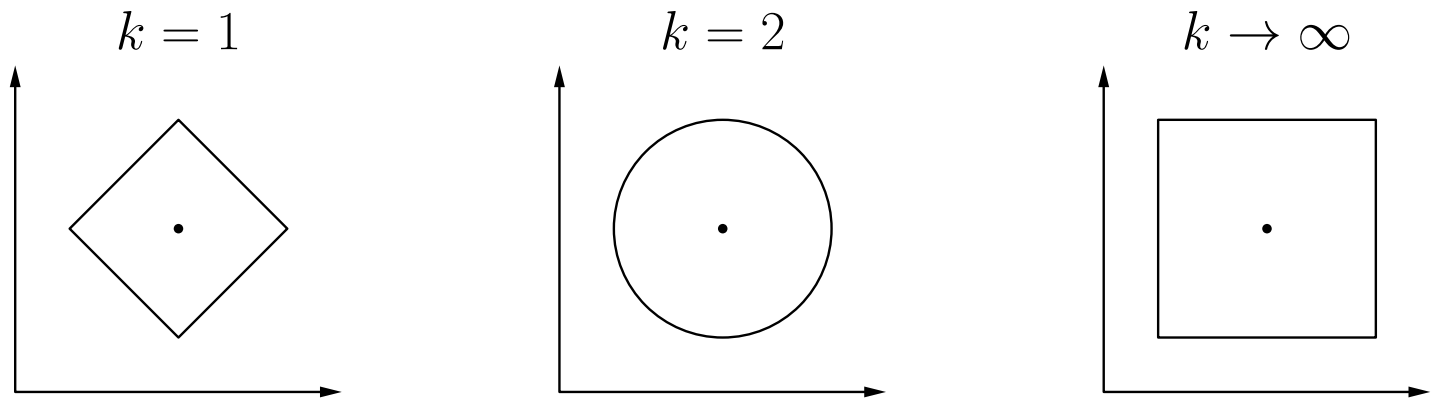
erfüllt.

Veranschaulichung von Abstandsfunktionen

$$d_k(\vec{x}, \vec{y}) = \left(\sum_{i=1}^{n} (x_i - y_i)^k \right)^{\frac{1}{k}}$$

Bekannte Spezialfälle:

$k = 1$: Manhattan- oder City-Block-Abstand,
$k = 2$: Euklidischer Abstand,
$k \to \infty$: Maximum-Abstand, d.h. $d_\infty(\vec{x}, \vec{y}) = \max_{i=1}^{n} |x_i - y_i|$.



Lernende Vektorquantisierung

Die Aktivierungsfunktion jedes Ausgabeneurons ist eine sogenannte **radiale Funktion**, d.h. eine monoton fallende Funktion

$$f : \mathbb{R}_0^+ \to [0, \infty] \quad \text{mit} \quad f(0) = 1 \quad \text{und} \quad \lim_{x \to \infty} f(x) = 0.$$

Manchmal wird der Wertebereich auf das Intervall $[0, 1]$ beschränkt.
Durch die spezielle Ausgabefunktion ist das allerdings unerheblich.

Die Ausgabefunktion jedes Ausgabeneurons ist keine einfache Funktion der Aktivierung des Neurons. Sie zieht stattdessen alle Aktivierungen aller Ausgabeneuronen in Betracht:

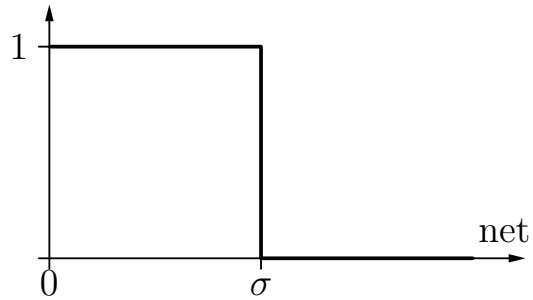
$$f_{\text{out}}^{(u)}(\text{act}_u) = \begin{cases} 1, & \text{falls } \text{act}_u = \max\limits_{v \in U_{\text{out}}} \text{act}_v, \\ 0, & \text{sonst.} \end{cases}$$

Sollte mehr als ein Neuron die maximale Aktivierung haben, wird ein zufällig gewähltes Neuron auf die Ausgabe 1 gesetzt, alle anderen auf Ausgabe 0:
Winner-Takes-All-Prinzip.

Radiale Aktivierungfunktionen

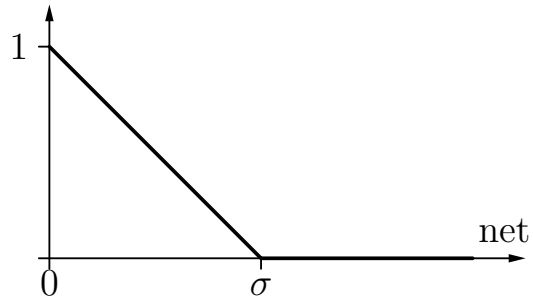
Rechteckfunktion:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{falls net} > \sigma, \\ 1, & \text{sonst.} \end{cases}$$



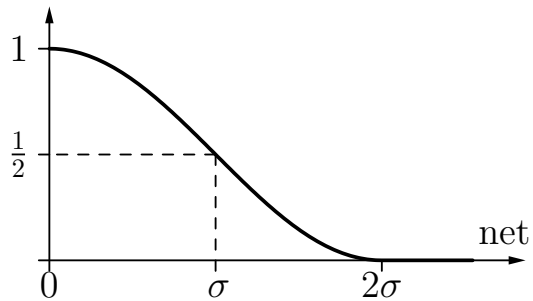
Dreiecksfunktion:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{falls net} > \sigma, \\ 1 - \frac{\text{net}}{\sigma}, & \text{sonst.} \end{cases}$$



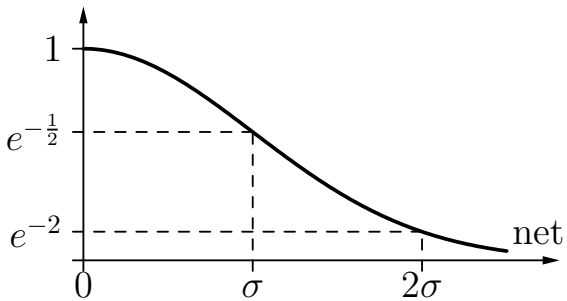
Kosinus bis Null:

$$f_{\text{act}}(\text{net}, \sigma) = \begin{cases} 0, & \text{falls net} > 2\sigma, \\ \frac{\cos\left(\frac{\pi}{2\sigma}\,\text{net}\right)+1}{2}, & \text{sonst.} \end{cases}$$



Gauß-Funktion:

$$f_{\text{act}}(\text{net}, \sigma) = e^{-\frac{\text{net}^2}{2\sigma^2}}$$



Lernende Vektorquantisierung

Anpassung der Referenzvektoren (Codebuch-Vektoren)

Bestimme zu jedem Trainingsbeispiel den nächsten Referenzvektor.

Passe nur diesen Referenzvektor an (Gewinnerneuron).

Für Klassifikationsprobleme kann die Klasse genutzt werden:
Jeder Referenzvektor wird einer Klasse zugeordnet.

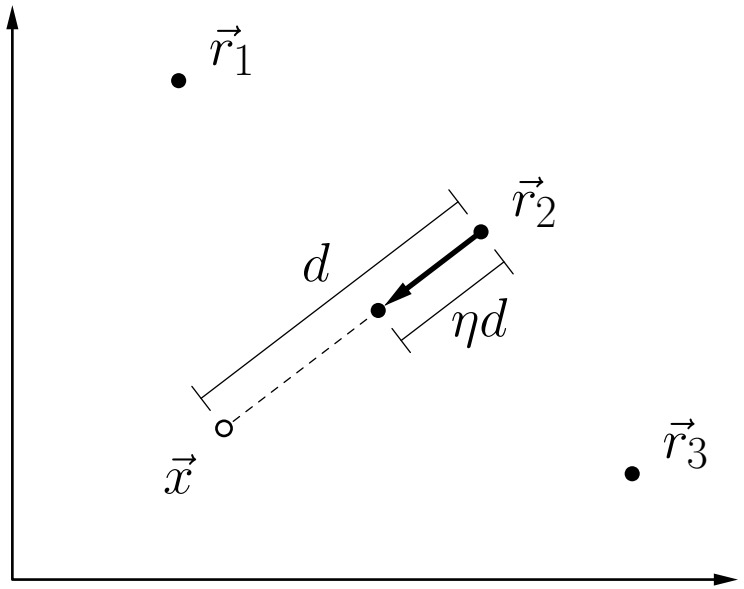
Anziehungsregel (Datenpunkt und Referenzvektor haben dieselbe Klasse)

$$\vec{r}^{\,(\text{new})} = \vec{r}^{\,(\text{old})} + \eta(\vec{x} - \vec{r}^{\,(\text{old})}),$$

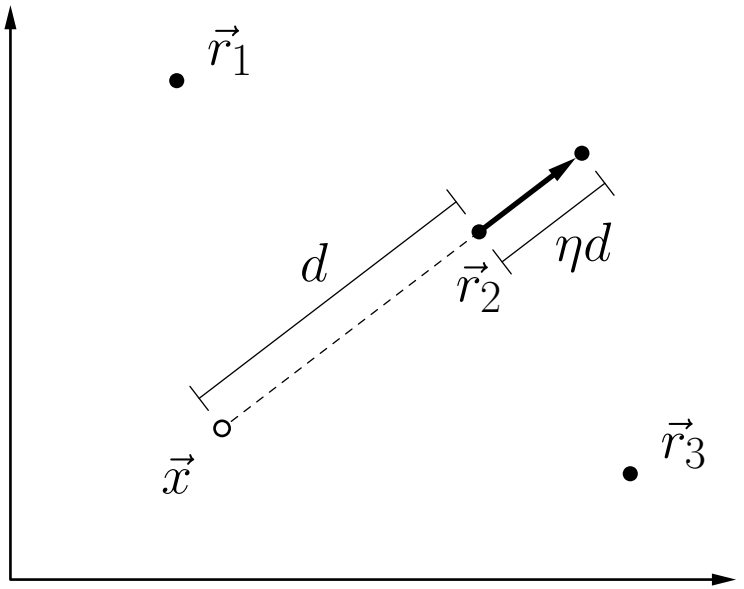
Abstoßungsregel (Datenpunkt und Referenzvektor haben verschiedene Klassen)

$$\vec{r}^{\,(\text{new})} = \vec{r}^{\,(\text{old})} - \eta(\vec{x} - \vec{r}^{\,(\text{old})}).$$

Anpassung der Referenzvektoren



Anziehungsregel

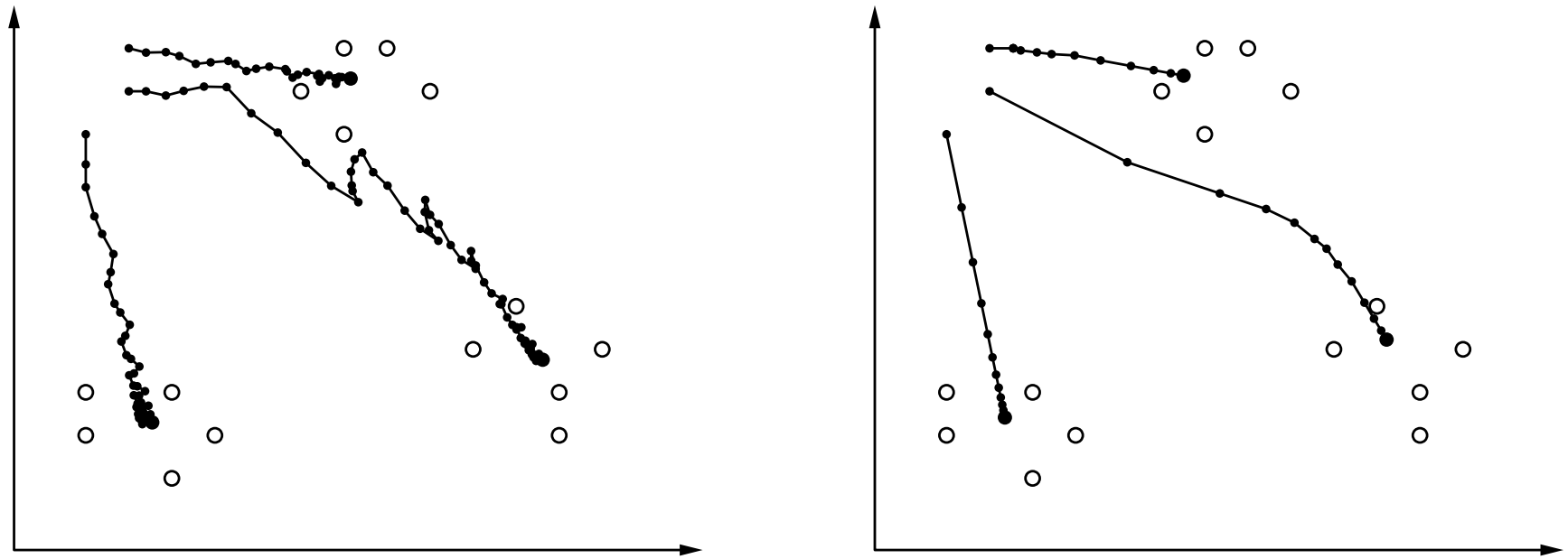


Abstoßungsregel

$\vec{x}$: Datenpunkt, $\vec{r}_i$: Referenzvektor
$\eta = 0.4$ (Lernrate)

Lernende Vektorquantisierung: Beispiel

Anpassung der Referenzvektoren

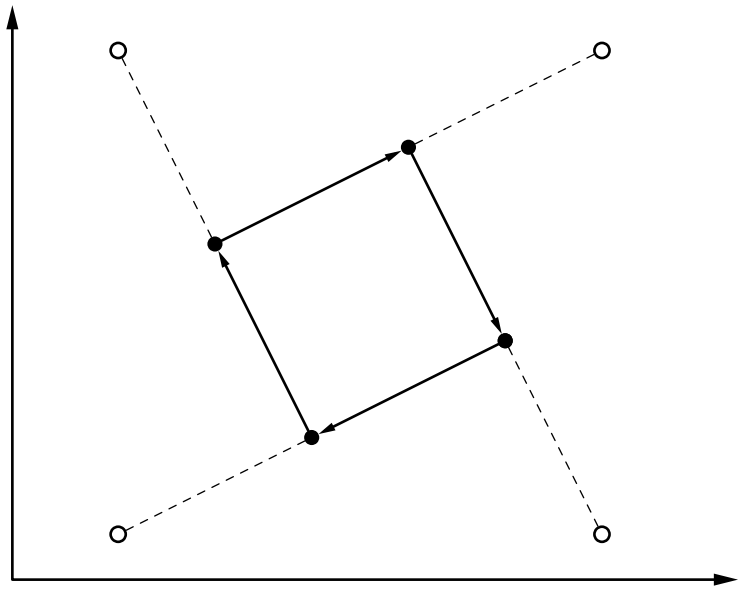


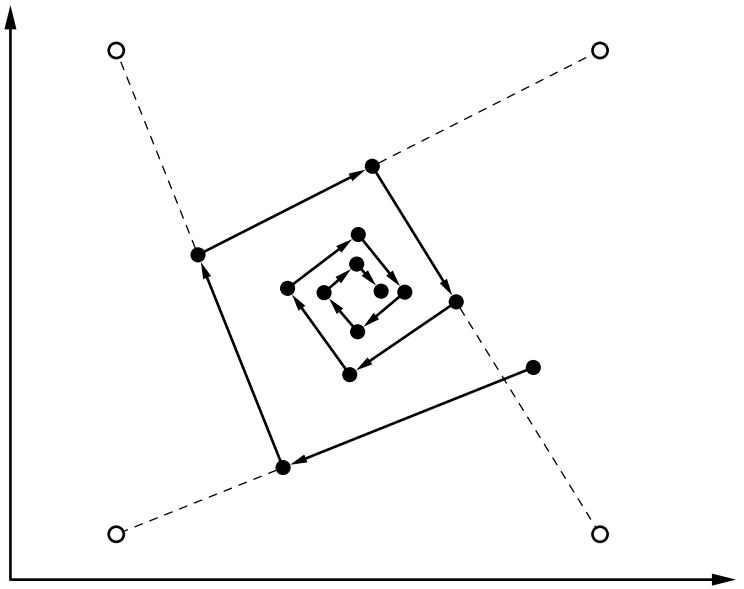
Links: Online-Training mit Lernrate $\eta = 0.1$,

Rechts: Batch-Training mit Lernrate $\eta = 0.05$.

Lernende Vektorquantisierung: Verfall der Lernrate

Problem: feste Lernrate kann zu Oszillationen führen





Lösung: **zeitabhängige Lernrate**

$$\eta(t) = \eta_0 \alpha^t, \quad 0 < \alpha < 1, \qquad \text{oder} \qquad \eta(t) = \eta_0 t^\kappa, \quad \kappa > 0.$$

Verbesserte Anpassungsregel für klassifizierte Daten

Idee: Passe nicht nur den Referenzvektor an, der am nächsten zum Datenpunkt liegt (das Gewinnerneuron), sondern passe **die zwei nächstliegenden Referenzvektoren**.

Sei $\vec{x}$ der momentan bearbeitete Datenpunkt und c seine Klasse.
Seien $\vec{r}_j$ und $\vec{r}_k$ die zwei nächstliegenden Referenzvektoren und z_j sowie z_k ihre Klassen.

Referenzvektoren werden nur angepasst, wenn $z_j \neq z_k$ und entweder $c = z_j$ oder $c = z_k$. (o.B.d.A. nehmen wir an: $c = z_j$.)

Die **Anpassungsregeln** für die zwei nächstgelegenen Referenzvektoren sind:

$$\begin{aligned} \vec{r}_j^{\,(\text{new})} &= \vec{r}_j^{\,(\text{old})} + \eta(\vec{x} - \vec{r}_j^{\,(\text{old})}) \qquad \text{und} \\ \vec{r}_k^{\,(\text{new})} &= \vec{r}_k^{\,(\text{old})} - \eta(\vec{x} - \vec{r}_k^{\,(\text{old})}), \end{aligned}$$

wobei alle anderen Referenzvektoren unverändert bleiben.

Lernende Vektorquantisierung: "Window Rule"

In praktischen Experimenten wurde beobachtet, dass LVQ in der Standardausführung die Referenzvektoren immer weiter voneinander wegtreibt.

Um diesem Verhalten entgegenzuwirken, wurde die **window rule** eingeführt: passe nur dann an, wenn der Datenpunkt $\vec{x}$ in der Nähe der Klassifikationsgrenze liegt.

"In der Nähe der Grenze" wird formalisiert durch folgende Bedingung:

$$\min\left(\frac{d(\vec{x},\vec{r}_j)}{d(\vec{x},\vec{r}_k)}, \frac{d(\vec{x},\vec{r}_k)}{d(\vec{x},\vec{r}_j)}\right) > \theta, \qquad \text{wobei} \qquad \theta = \frac{1-\xi}{1+\xi}.$$

ξ ist ein Parameter, der vom Benutzer eingestellt werden muss.

Intuitiv beschreibt ξ die "Größe" des Fensters um die Klassifikationsgrenze, in dem der Datenpunkt liegen muss, um zu einer Anpassung zu führen.

Damit wird die Divergenz vermieden, da die Anpassung eines Referenzvektors nicht mehr durchgeführt wird, wenn die Klassifikationsgrenze weit genug weg ist.

Soft LVQ

Idee: Benutze weiche Zuordnungen anstelle von *Winner-Takes-All*.

Annahme: Die Daten wurden aus einer Mischung von Normalverteilungen gezogen. Jeder Referenzvektor beschreibt eine Normalverteilung.

Ziel: Maximiere das Log-Wahrscheinlichkeitsverhältnis der Daten, also

$$\ln L_{\text{ratio}} = \sum_{j=1}^{n} \ln \sum_{\vec{r} \in R(c_j)} \exp\left(-\frac{(\vec{x}_j - \vec{r})^\top (\vec{x}_j - \vec{r})}{2\sigma^2}\right) - \sum_{j=1}^{n} \ln \sum_{\vec{r} \in Q(c_j)} \exp\left(-\frac{(\vec{x}_j - \vec{r})^\top (\vec{x}_j - \vec{r})}{2\sigma^2}\right).$$

Hierbei ist σ ein Parameter, der die "Größe" jeder Normalverteilung angibt.
$R(c)$ ist die Menge der Referenzvektoren der Klasse c und $Q(c)$ deren Komplement.

Intuitiv: für jeden Datenpunkt sollte die Wahrscheinlichkeitsdichte für seine Klasse so groß wie möglich sein, während die Dichte für alle anderen Klassen so klein wie möglich sein sollte.

Anpassungsregel abgeleitet aus Maximum-Log-Likelihood-Ansatz:

$$\vec{r}_i^{\,(\text{new})} = \vec{r}_i^{\,(\text{old})} + \eta \cdot \begin{cases} u_{ij}^{\oplus} \cdot (\vec{x}_j - \vec{r}_i^{\,(\text{old})}), & \text{if } c_j = z_i, \\ -u_{ij}^{\ominus} \cdot (\vec{x}_j - \vec{r}_i^{\,(\text{old})}), & \text{if } c_j \neq z_i, \end{cases}$$

wobei z_i die dem Referenzvektor $\vec{r}_i$ zugehörige Klasse ist und

$$u_{ij}^{\oplus} = \frac{\exp\left(-\frac{1}{2\sigma^2}(\vec{x}_j - \vec{r}_i^{\,(\text{old})})^\top (\vec{x}_j - \vec{r}_i^{\,(\text{old})})\right)}{\sum_{\vec{r} \in R(c_j)} \exp\left(-\frac{1}{2\sigma^2}(\vec{x}_j - \vec{r}^{\,(\text{old})})^\top (\vec{x}_j - \vec{r}^{\,(\text{old})})\right)} \quad \text{und}$$

$$u_{ij}^{\ominus} = \frac{\exp\left(-\frac{1}{2\sigma^2}(\vec{x}_j - \vec{r}_i^{\,(\text{old})})^\top (\vec{x}_j - \vec{r}_i^{\,(\text{old})})\right)}{\sum_{\vec{r} \in Q(c_j)} \exp\left(-\frac{1}{2\sigma^2}(\vec{x}_j - \vec{r}^{\,(\text{old})})^\top (\vec{x}_j - \vec{r}^{\,(\text{old})})\right)}.$$

$R(c)$ ist die Menge der Referenzvektoren, die zu Klasse c gehören und $Q(c)$ ist deren Komplement.

Hard LVQ

Idee: Leite ein Schema mit scharfen Zuordnungen aus der unscharfen Version ab.

Ansatz: Lasse den Größenparameter σ der Gaußfunktion gegen Null streben.

Die sich ergebende Anpassungsregel ist somit:

$$\vec{r}_i^{\,(\text{new})} = \vec{r}_i^{\,(\text{old})} + \eta \cdot \begin{cases} u_{ij}^{\oplus} \cdot (\vec{x}_j - \vec{r}_i^{\,(\text{old})}), & \text{if } c_j = z_i, \\ -u_{ij}^{\ominus} \cdot (\vec{x}_j - \vec{r}_i^{\,(\text{old})}), & \text{if } c_j \neq z_i, \end{cases}$$

wobei

$$u_{ij}^{\oplus} = \begin{cases} 1, & \text{falls } \vec{r}_i = \underset{\vec{r} \in R(c_j)}{\operatorname{argmin}}\, d(\vec{x}_j, \vec{r}), \\ 0, & \text{sonst,} \end{cases} \qquad u_{ij}^{\ominus} = \begin{cases} 1, & \text{falls } \vec{r}_i = \underset{\vec{r} \in Q(c_j)}{\operatorname{argmin}}\, d(\vec{x}_j, \vec{r}), \\ 0, & \text{sonst.} \end{cases}$$

$\vec{r}_i$ ist der nächstgelegene Vektor derselben Klasse

$\vec{r}_i$ ist der nächstgelegene Vektor einer anderen Klasse

Diese Anpassungsregel ist stabil, ohne dass eine *window rule* die Anpassung beschränken müsste.

Lernende Vektorquantisierung: Erweiterungen

Frequency Sensitive Competitive Learning

- Der Abstand zu einem Referenzvektor wird modifiziert, indem berücksichtigt wird, wieviele Datenpunkte diesem Referenzvektor zugewiesen sind.

Fuzzy LVQ

- Nutzt die enge Verwandschaft zum Fuzzy-Clustering aus.
- Kann als Online-Version des Fuzzy-Clustering angesehen werden.
- Führt zu schnellerem Clustering.

Größen- und Formparameter

- Weise jedem Referenzvektor einen Clusterradius zu.
  Passe diesen Radius in Abhängigkeit von der Nähe der Datenpunkte an.
- Weise jedem Referenzvektor eine Kovarianzmatrix zu.
  Passe diese Matrix abhängig von der Verteilung der Datenpunkte an.

Selbstorganisierende Karten

(engl. Self-Organizing Maps (SOMs))

Selbstorganisierende Karten

Eine **selbstorganisierende Karte** oder **Kohonen-Merkmalskarte** ist ein neuronales Netz mit einem Graphen $G = (U, C)$ das folgende Bedingungen erfüllt:

(i) $U_{\text{hidden}} = \emptyset$, $U_{\text{in}} \cap U_{\text{out}} = \emptyset$,

(ii) $C = U_{\text{in}} \times U_{\text{out}}$.

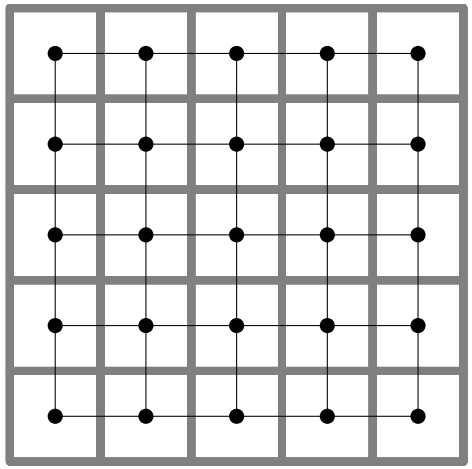
Die Netzeingabefunktion jedes Ausgabeneurons ist eine **Abstandsfunktion** zwischen Eingabe- und Gewichtsvektor. Die Aktivierungsfunktion jedes Ausgabeneurons ist eine **radiale Funktion**, d.h. eine monoton fallende Funktion

$$f : \mathbb{R}_0^+ \to [0, 1] \quad \text{mit} \quad f(0) = 1 \quad \text{und} \quad \lim_{x \to \infty} f(x) = 0.$$

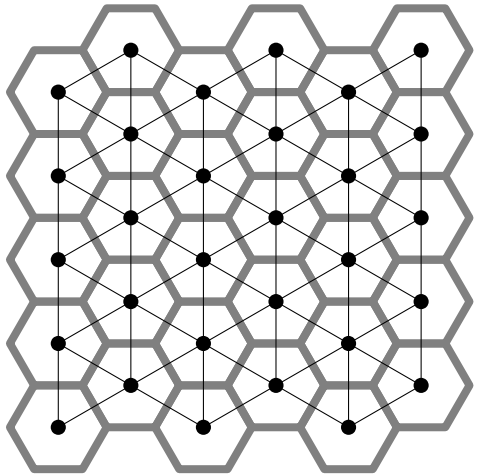
Die Ausgabefunktion jedes Ausgabeneurons ist die Identität.
Die Ausgabe wird oft per "**Winner-Takes-All**"-Prinzip diskretisiert.
Auf den Ausgabeneuronen ist eine **Nachbarschaftsbeziehung** definiert:

$$d_{\text{neurons}} : U_{\text{out}} \times U_{\text{out}} \to \mathbb{R}_0^+ .$$

Nachbarschaft der Ausgabeneuronen: Neuronen bilden ein Gitter



quadratisches Gitter

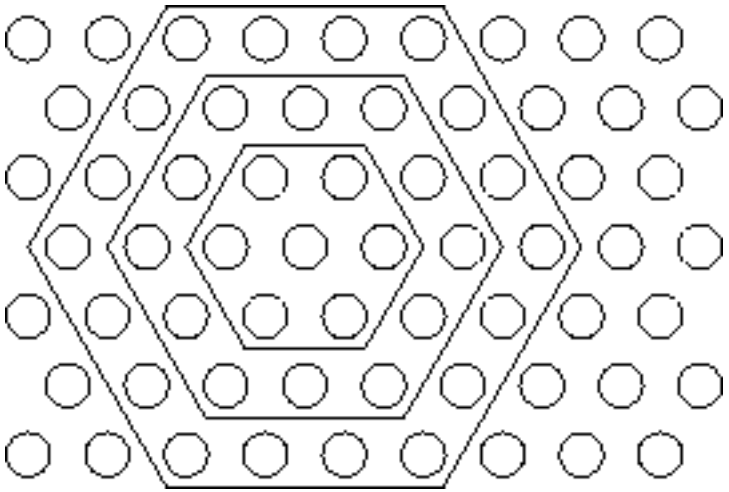


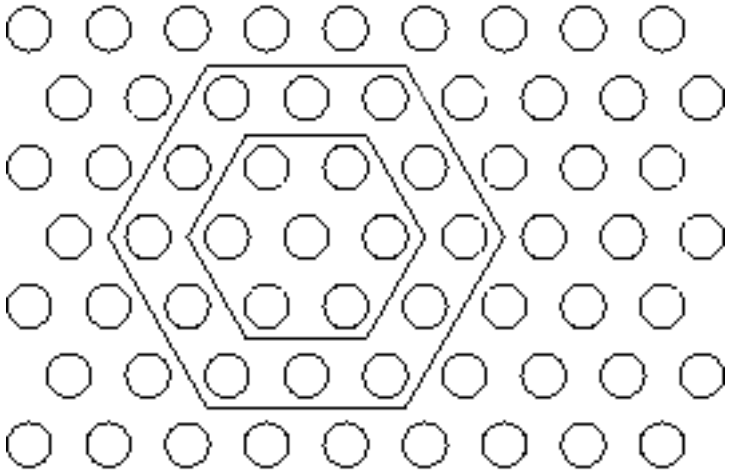
hexagonales Gitter

Dünne schwarze Linien: Zeigen nächste Nachbarn eines Neurons.

Dicke graue Linien: Zeigen Regionen, die einem Neuron zugewiesen sind.

Nachbarschaft des Gewinnerneurons

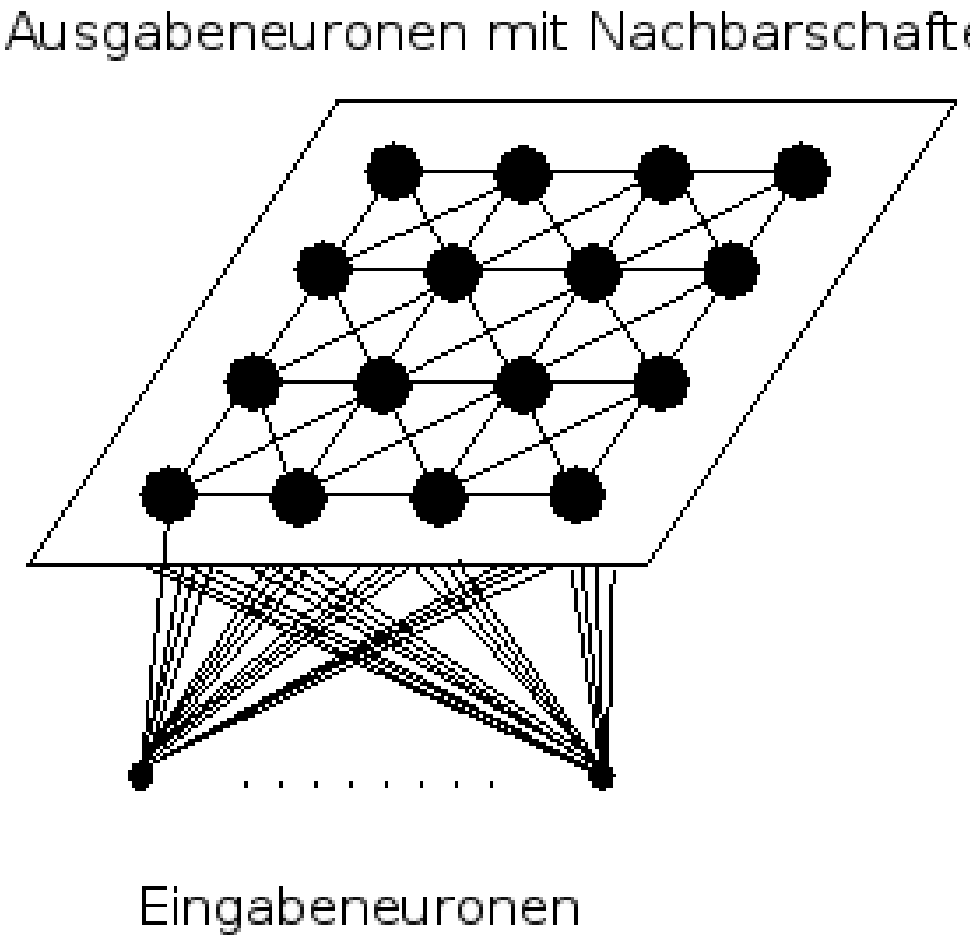




Der Nachbarschaftsradius wird im Laufe des Lernens kleiner.

Selbstorganisierende Karten: Struktur

Die “Karte” stellt die Ausgabeneuronen mit deren Nachbarschaften dar.



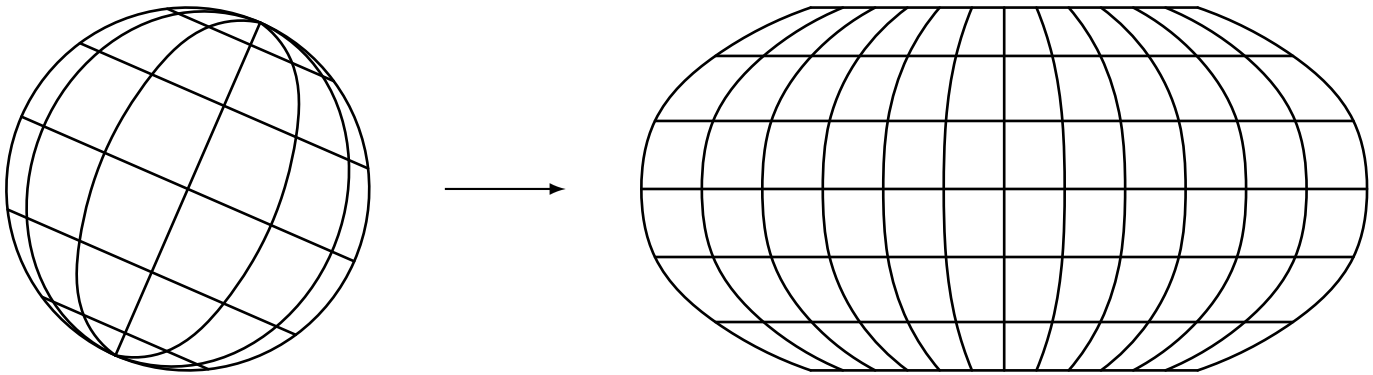
Ablauf des SOM-Lernens

1. Initialisierung der Gewichtsvektoren der Karte
2. zufällige Wahl des Eingabevektors aus der Trainingsmenge
3. Bestimmung des Gewinnerneurons über Abstandsfunktion
4. Bestimmung des zeitabhängigen Radius und der im Radius liegenden Nachbarschaftsneuronen des Gewinners
5. Zeitabhängige Anpassung dieser Nachbarschaftsneuronen, weiter bei 2.

Topologieerhaltende Abbildung

Abbildungen von Punkten, die im Originalraum nah beieinander sind, sollen im Bildraum ebenfalls nah beieinander sein.

Beispiel: **Robinson-Projektion** der Oberfläche einer Kugel



Die Robinson-Projektion wird häufig für Weltkarten genutzt.

$\rightarrow$ eine SOM realisiert eine topologieerhaltende Abbildung.

Finde topologieerhaltende Abbildung durch Beachtung der Nachbarschaft

Anpassungsregel für Referenzvektor:

$$\vec{r}_u^{\,(\text{new})} = \vec{r}_u^{\,(\text{old})} + \eta(t) \cdot f_{\text{nb}}(d_{\text{neurons}}(u, u_*), \varrho(t)) \cdot (\vec{x} - \vec{r}_u^{\,(\text{old})}),$$

u_* ist das Gewinnerneuron (Referenzvektor am nächsten zum Datenpunkt).

Die Funktion f_{nb} ist eine radiale Funktion.

Zeitabhängige Lernrate

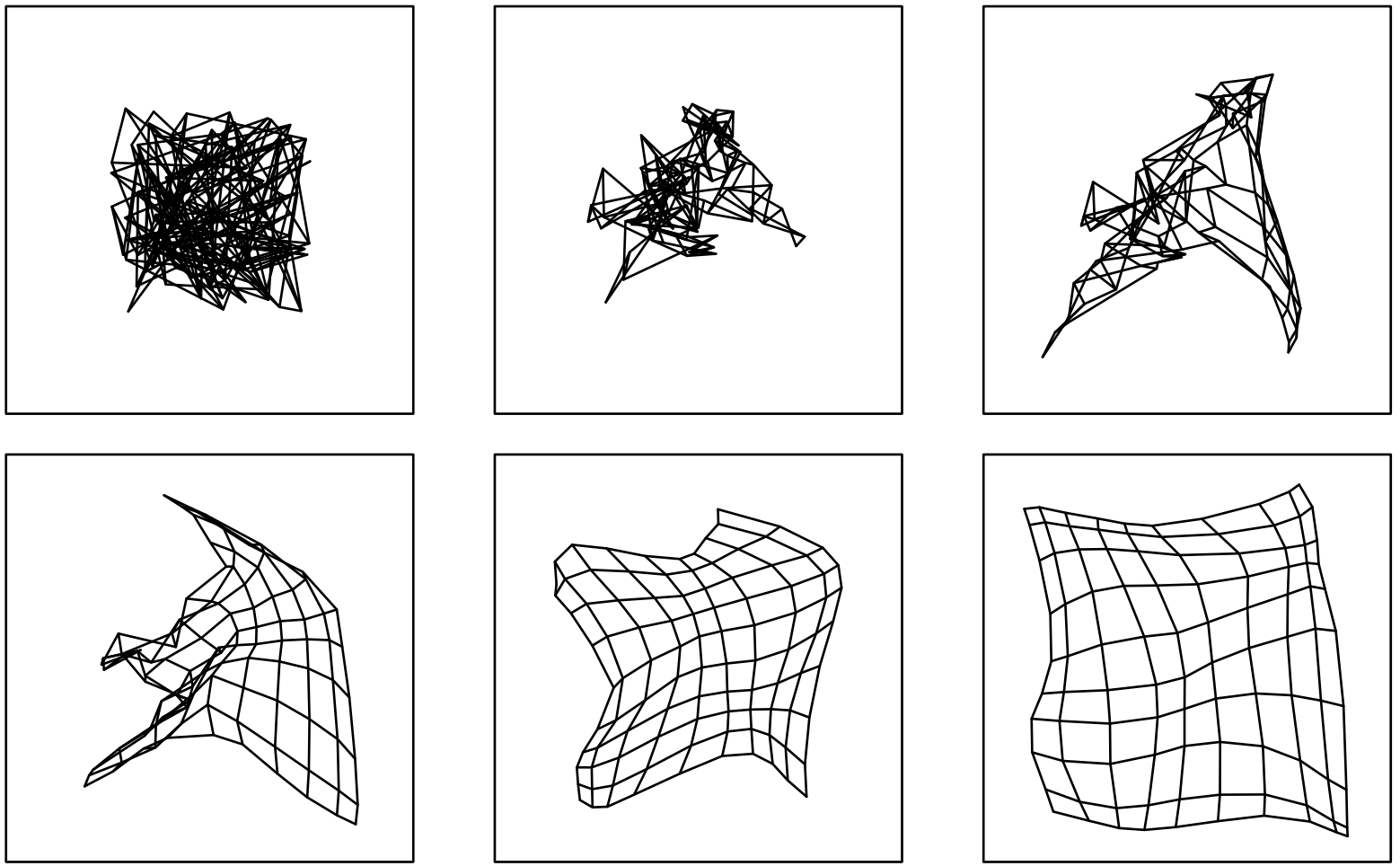
$$\eta(t) = \eta_0 \alpha_\eta^t, \quad 0 < \alpha_\eta < 1, \qquad \text{oder} \qquad \eta(t) = \eta_0 t^{\kappa_\eta}, \quad \kappa_\eta > 0.$$

Zeitabhängiger Nachbarschaftsradius

$$\varrho(t) = \varrho_0 \alpha_\varrho^t, \quad 0 < \alpha_\varrho < 1, \qquad \text{oder} \qquad \varrho(t) = \varrho_0 t^{\kappa_\varrho}, \quad \kappa_\varrho > 0.$$

Selbstorganisierende Karten: Beispiele

Beispiel: Entfalten einer zweidimensionalen SOM



Die Abbildungen (von links nach rechts, von oben nach unten) zeigen den jeweiligen Zustand der SOM (des Eingaberaums) nach 10, 20, 40, 80 und 160 Lernschritten. In jedem Schritt wird ein Trainingsmuster verarbeitet.

Selbstorganisierende Karten: Beispiele

Beispiel: Entfalten einer zweidimensionalen SOM (Erläuterungen)

Entfaltung einer 10x10-Karte, die mit zufälligen Mustern aus $[-1, 1] \times [-1, 1]$ trainiert wird

Initialisierung mit Referenzvektoren aus $[-0.5, 0.5]$

Linien verbinden direkte Nachbarn (Gitter/Grid)

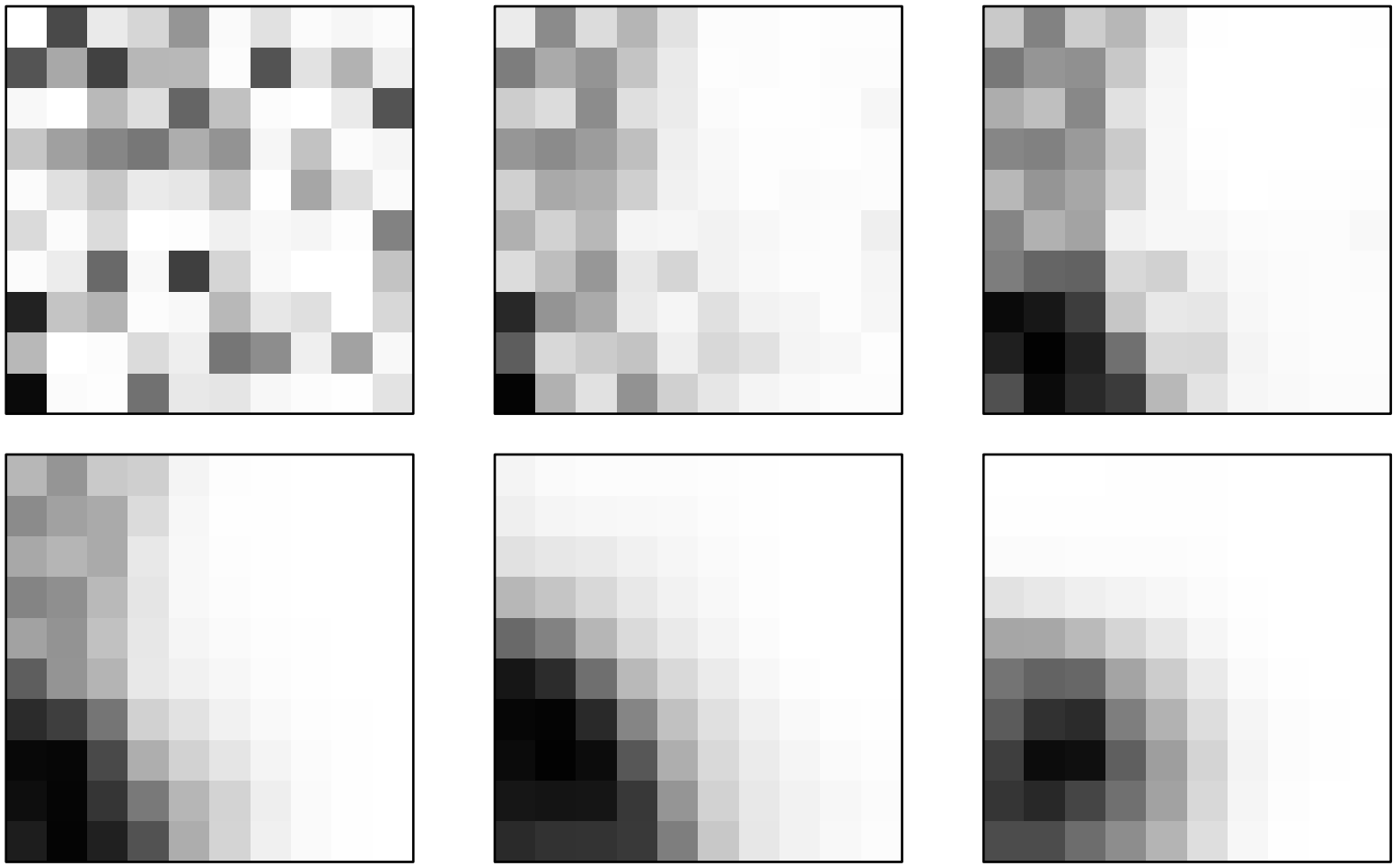
Lernrate $\eta(t) = 0.6 * t$

Gaußsche Nachbarschaftsfunktion f_{nb}

Radius $\rho(t) = 2.5 * t^{-0.1}$

Selbstorganisierende Karten: Beispiele

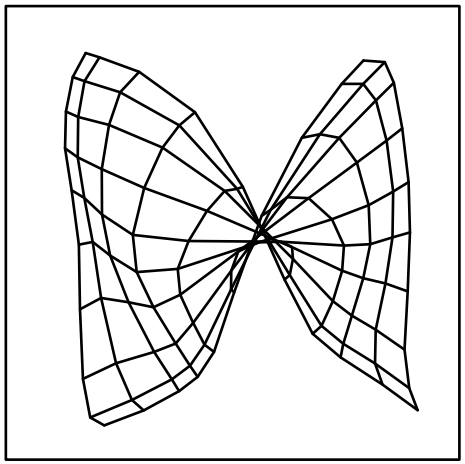
Beispiel: Entfalten einer zweidimensionalen SOM



Die Abbildungen verdeutlichen die Gitterstruktur der Ausgabeneuronen. Jedes Neuron wird durch ein kleines Quadrat dargestellt. Die Graustufe kodiert die Aktivierung des Ausgabeneurons für das Muster $(-0.5, -0.5)$ bei Gaußscher Aktivierungsfunktion.

Selbstorganisierende Karten: Beispiele

Beispiel: Entfalten einer zweidimensionalen SOM

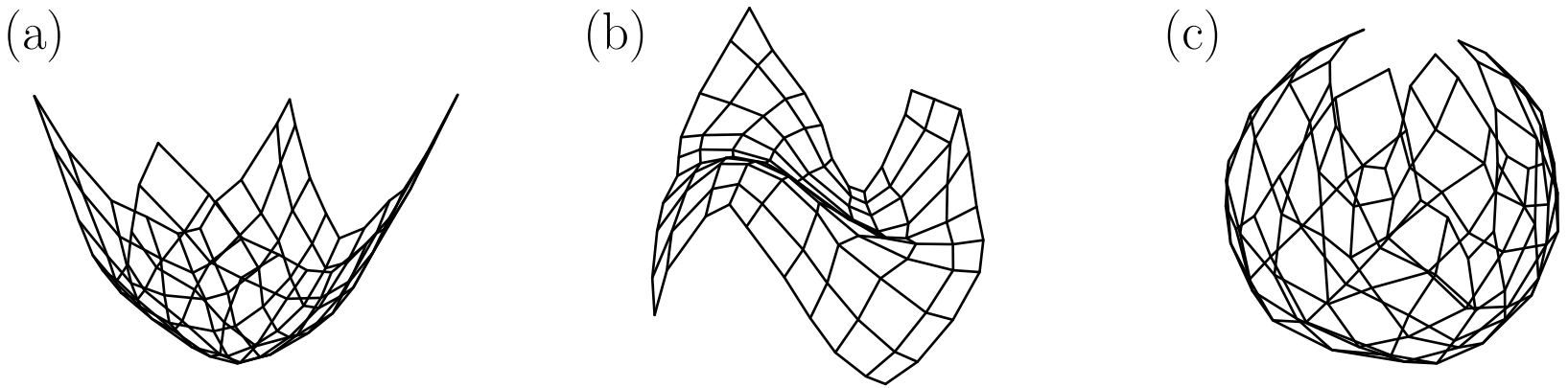


Das Trainieren einer SOM kann u.a. fehlschlagen, falls

- die Initialisierung ungünstig ist oder
- die (anfängliche) Lernrate zu klein gewählt ist oder
- die (anfängliche) Nachbarschaft zu klein gewählt ist.

Selbstorganisierende Karten: Beispiele

Beispiel: Entfalten einer zweidimensionalen SOM, Dimensionsreduktion



Als Lernstichprobe werden zufällige Punkte der Oberfläche einer Rotationsparabel (bzw. kubische Funktion, Kugel) gewählt, also drei Eingabeneuronen (x,y,z-Koordinaten).

Eine Karte mit 10×10 Ausgabeneuronen wird trainiert.

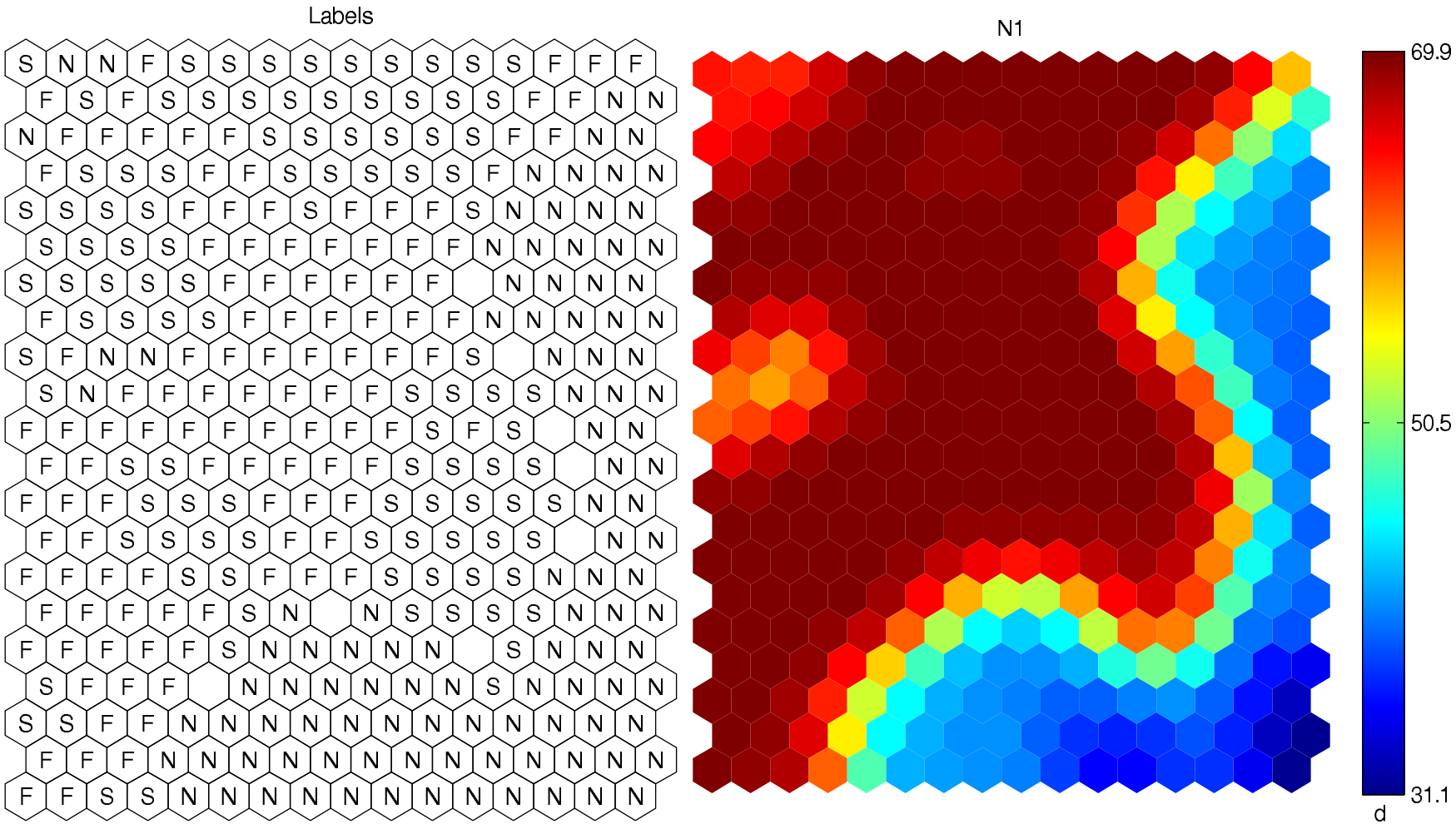
Die 3D-Referenzvektoren der Ausgabeneuronen (mit Gitter) werden dargestellt.

Wegen 2D-Fläche (gekrümmt) klappt die Anpassung sehr gut.

In diesen Fällen haben Originalraum und Bildraum unterschiedliche Dimensionen.

Selbstorganisierende Karten können zur Dimensionsreduktion genutzt werden.

SOM, Beispiel Clustering von Feldbearbeitungsstrategien



Links: selbstorganisierende Karte mit eingezeichneten Klassenlabels, rechts eine der zum Lernen der Karte verwendeten Variablen [Ruß; 2012]

SOM, Phonemkarte des Finnischen

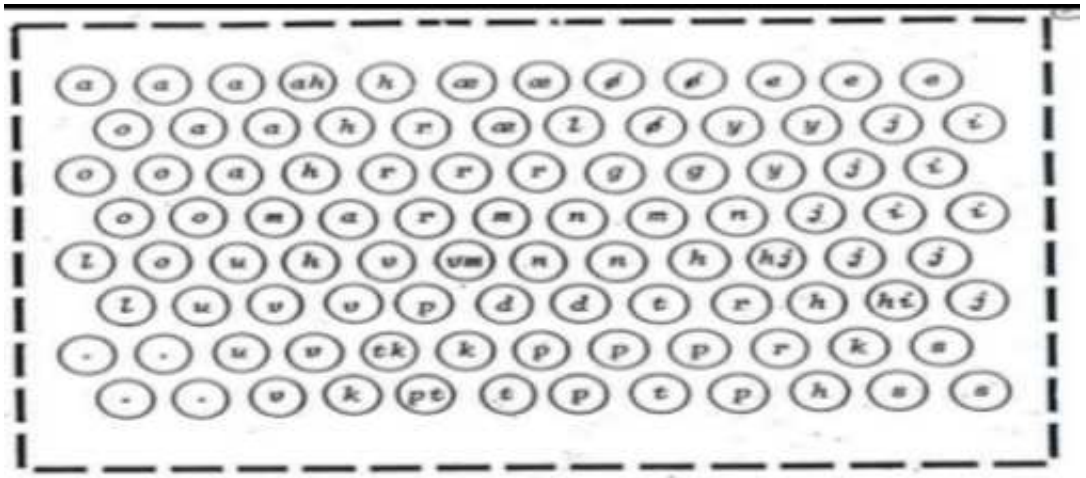


Abb. 2.6.7 Phonemkarte des Finnischen (nach [KOH88])

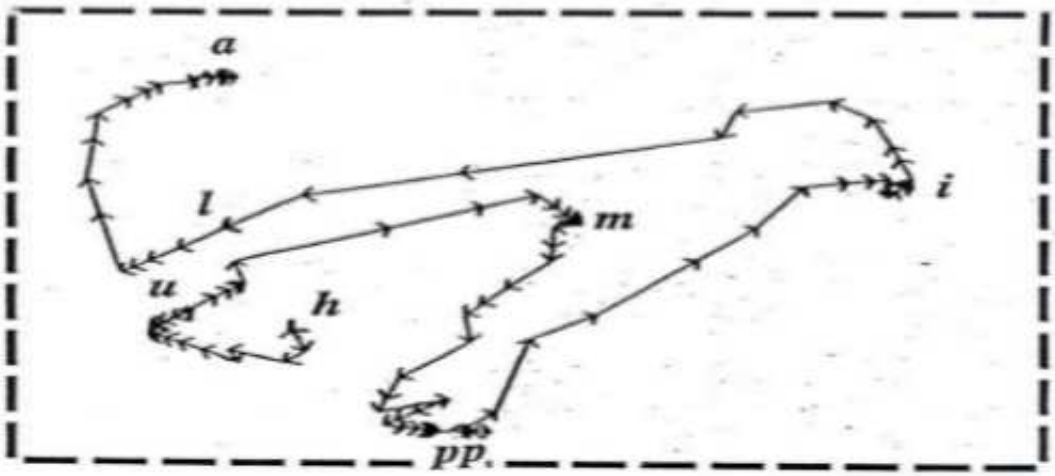
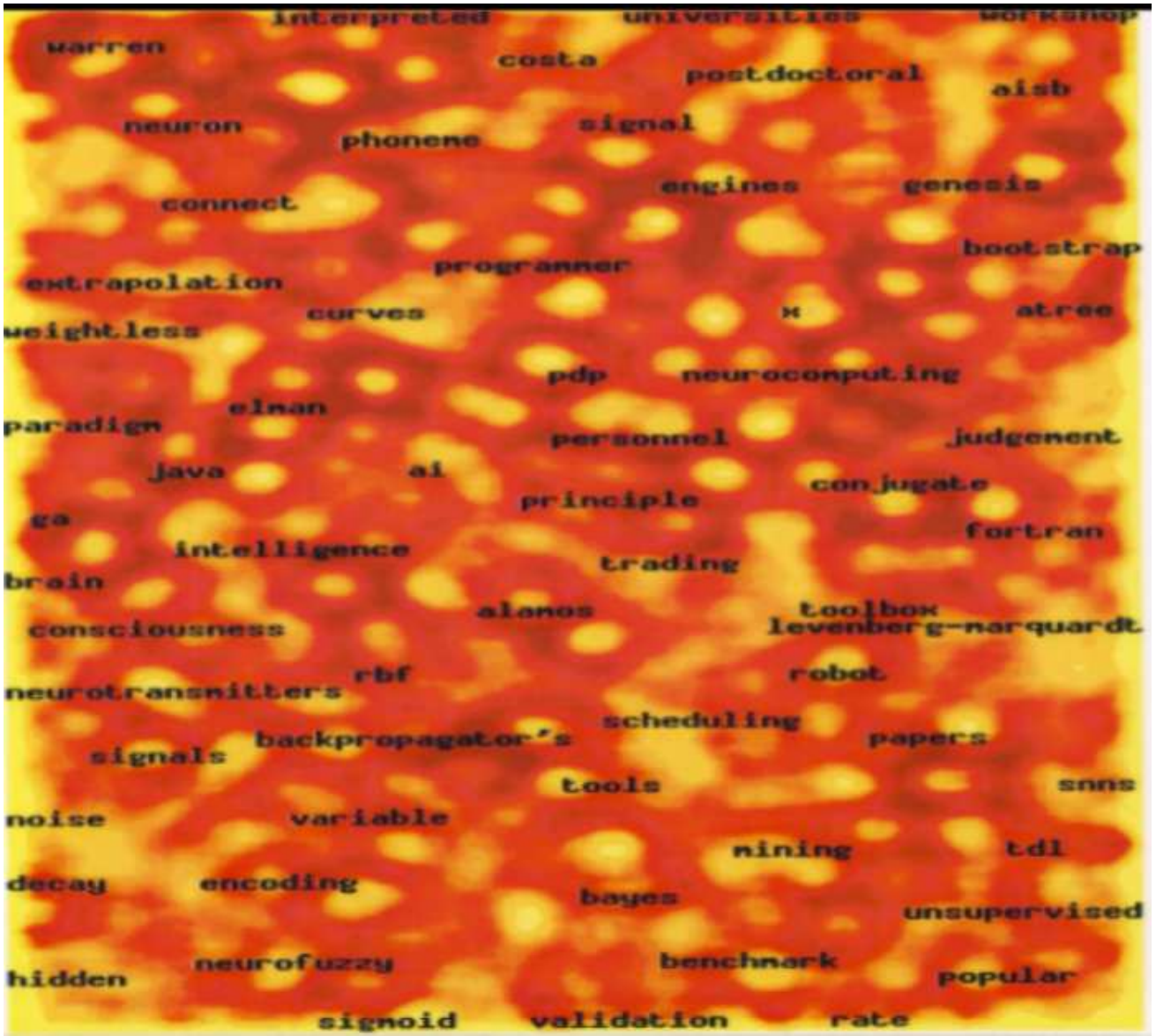
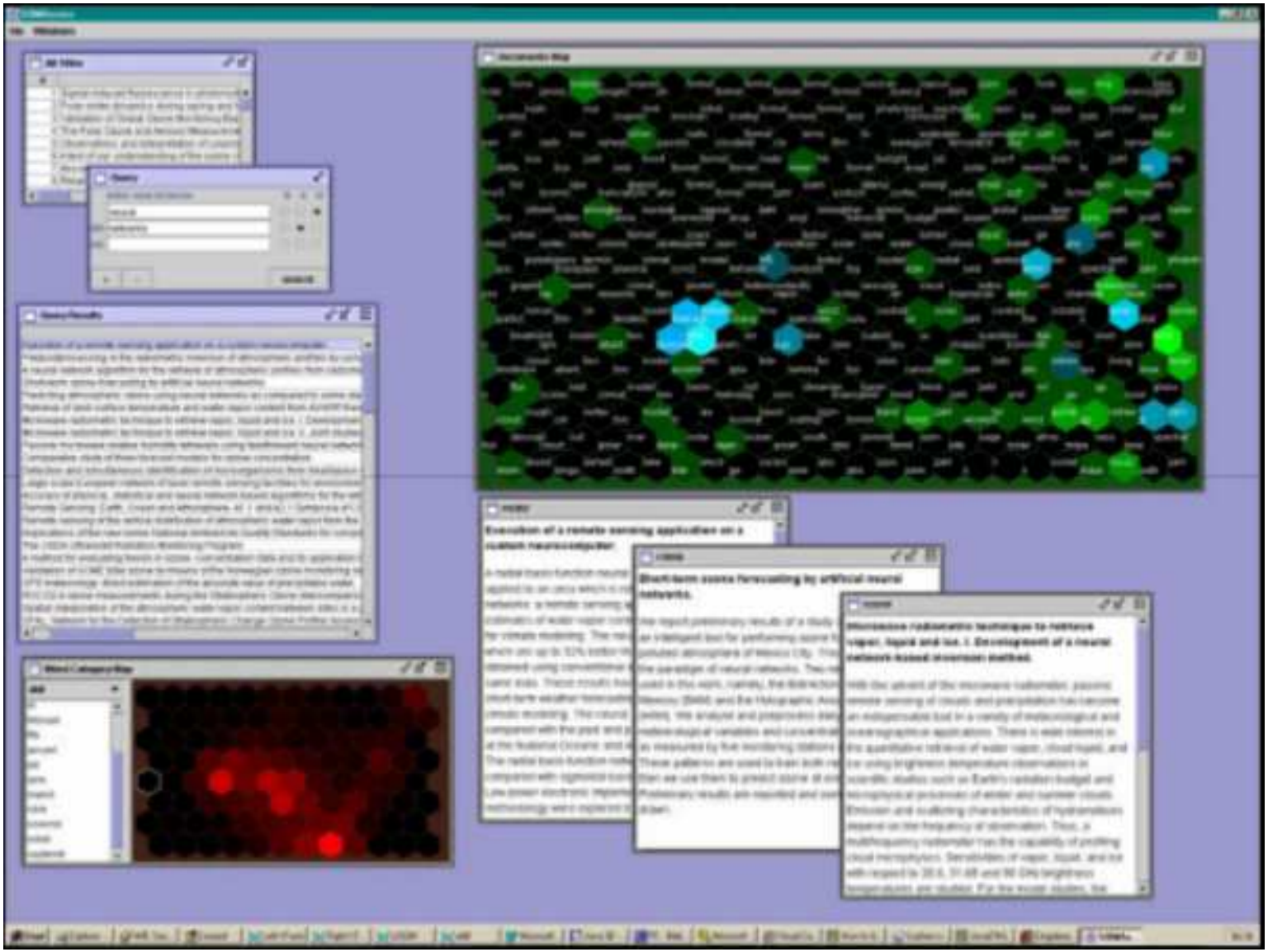


Abb. 2.6.8 Phonemsequenz für */humppila/* (nach [KOH88])

SOM, Websom


interpreted
universities
workshop
warren
costa
postdoctoral
aisb
neuron
phonene
signal
engines
genesis
connect
bootstrap
programmer
extrapolation
curves
atree
weightless
pdp
neuroconputing
elman
paradigm
personnel
judgement
java
ai
conjugate
principle
ga
fortran
intelligence
trading
brain
alanos
toolbox
consciousness
levenberg-marquardt
rbf
robot
neurotransmitters
scheduling
backpropagator's
papers
signals
tools
snns
noise
variable
nining
tdl
decay
encoding
bayes
unsupervised
neurofuzzy
benchmark
popular
hidden
signoid
validation
rate

SOM, Websom, [Nürnberger, Klose, Kruse; 2003]



SOM, MusicMiner [Mörchen, Ultsch u.a.; 2005]