Vector quantization

- Assume we are given a probability density function $p(\vec{x})$ on input vectors $\vec{x} \in \mathbb{R}^n$.

  I.e. assume that the inputs are randomly generated according to $p(\vec{x})$.
- Our goal is to approximate $p(\vec{x})$ using finitely many **centres** $\vec{w}_i \in \mathbb{R}^n$ where $i = 1, \ldots, h$.

  Roughly speaking: We want more centres in areas of higher density and less in areas of low density.
- Formally: To every input $\vec{x}$ we assign its *closest* centre $\vec{w}_{c(\vec{x})}$:

  $$c(\vec{x}) = \arg \min_{i=1,\ldots,h} \left\{ \left\| \vec{x} - \vec{w}_i \right\| \right\}$$

  and then minimize the error

  $$E = \int \left\| \vec{x} - \vec{w}_{c(\vec{x})} \right\|^2 p(\vec{x}) d\vec{x}$$

  Caution! $c(\vec{x})$ depends on $\vec{x}$.

Vector quantization

In practice, $p(\vec{x})$ is obtained by *sampling uniformly* from a given training (multi)set:

$$\mathcal{T} = \{\vec{x}_j \in \mathbb{R}^n \mid j = 1, \ldots, \ell\}$$

The error then corresponds to

$$E = \frac{1}{\ell} \sum_{j=1}^{\ell} \left\| \vec{x}_j - \vec{w}_{c(\vec{x}_j)} \right\|^2$$

(keep in mind that $c(\vec{x}_j) = \arg\min_{i=1,\ldots,h} \left\{ \left\| \vec{x}_j - \vec{w}_i \right\| \right\}$.)

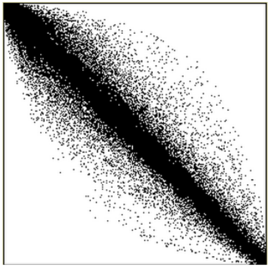
If $\mathcal{T}$ has been randomly selected according to $p(\vec{x})$ and ℓ is large eough, then

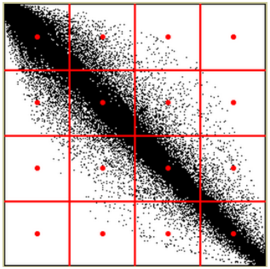
$$\frac{1}{\ell} \sum_{j=1}^{\ell} \left\| \vec{x}_j - \vec{w}_{c(\vec{x}_j)} \right\|^2 \approx \int \left\| \vec{x} - \vec{w}_{c(\vec{x})} \right\|^2 p(\vec{x}) d\vec{x}$$

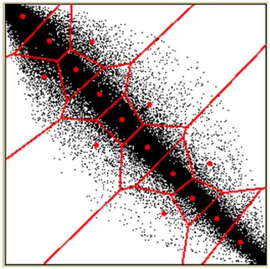
Example – image compression

- Every pixel has 256 shades of grey,
- each pair of neighbouring pixels is a two-dimensional vector from $\{0, \dots, 255\} \times \{0, \dots, 255\}$,
- our compression finds a small set of centres that will encode shades of grey of *pairs of pixels*,
- image is then encoded by simple substitution of pairs of pixels with their centres.

Example – image compression


pair distribution


naive quantization


smart quantization

Lloyd's algorithm

Assume a finite training set: $\mathcal{T} = \{\vec{x}_j \in \mathbb{R}^n \mid j = 1, \ldots, \ell\}$

The algorithm moves centres closer to the centres of mass of closest points.

In the step t computes $\vec{w}_1^{(t)}, \ldots, \vec{w}_h^{(t)}$ as follows:

- for every $k = 1, \ldots, h$ compute a set $\mathcal{T}_k$ of all vectors of $\mathcal{T}$ to which $\vec{w}_k^{(t-1)}$ is the closest centre:

$$\mathcal{T}_k = \left\{ \vec{x}_j \in \mathcal{T} \quad \middle| \quad k = \arg\min_{i=1,\ldots,h} \left\{ \left\| \vec{x}_j - \vec{w}_i^{(t-1)} \right\| \right\} \right\}$$

- compute $\vec{w}_k^{(t)}$ as the centre of mass of $\mathcal{T}_k$:

$$\vec{w}_k^{(t)} = \frac{1}{|\mathcal{T}_k|} \sum_{\vec{x} \in \mathcal{T}_k} \vec{x}$$

We may stop the computation when, e.g. the error E is sufficiently small.

Kohonen's learning

Disadvantage of Lloyd's algorithm: It is not online!

The following Kohonen's algorithm is online (i.e. the inputs may be generated one by one and the centres are adapted online):

In step t, consider the input $\vec{x}_t$ and compute $\vec{w}_k^{(t)}$ as follows:

If $\vec{w}_k^{(t-1)}$ is the closest centre to $\vec{x}_t$, i.e.
$k = \arg\min_i \left\| \vec{x}_t - \vec{w}_i^{(t-1)} \right\|$ **then**

$$\vec{w}_k^{(t)} = \vec{w}_k^{(t-1)} + \theta \cdot (\vec{x}_t - \vec{w}_k^{(t-1)})$$

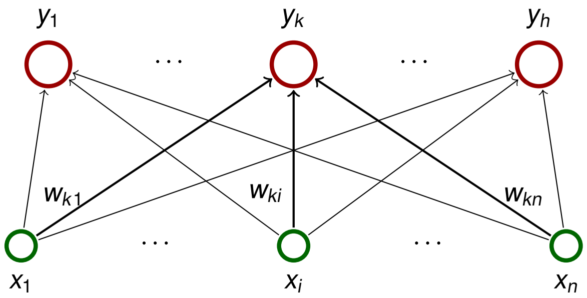
else $\vec{w}_k^{(t)} = \vec{w}_k^{(t-1)}$

$0 < \theta \leq 1$ determines how much to move the centre towards the input.

Let us formulate this algorithm in the language of neural networks.

Kohonen's learning – neural network

Architecture: Single layer



Activity: For an input $\vec{x} \in \mathbb{R}^n$ and $k = 1, \ldots, h$:

$$y_k = \begin{cases} 1 & k = \arg\min_{i=1,\ldots,h} \left\| \vec{x} - \vec{w}_i \right\| \\ 0 & \text{otherwise} \end{cases}$$

Kohonen's learning

In step t, consider the input $\vec{x}_t$ and compute $\vec{w}_k^{(t)}$ as follows:

If $\vec{w}_k^{(t-1)}$ is the closest center to $\vec{x}_t$, i.e.
$k = \arg\min_i \left\| \vec{x}_t - \vec{w}_i^{(t-1)} \right\|$ **then**

$$\vec{w}_k^{(t)} = \vec{w}_k^{(t-1)} + \theta \cdot (\vec{x}_t - \vec{w}_k^{(t-1)})$$

else $\vec{w}_k^{(t)} = \vec{w}_k^{(t-1)}$

$0 < \theta \leq 1$ determines how much to move the center towards the input.

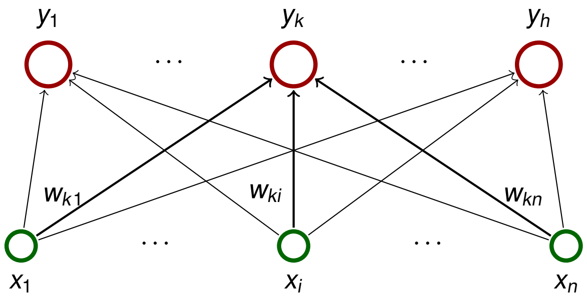
Kohonen's learning – efficiency

- Works well if most input vectors evenly distributed in a convex area.
- In case of two (or more) separated clusters, the density may not correspond to $p(\vec{x})$ at all:
  - Ex. Two separated areas with the same density.
  - Assume that the centres are initially in one of the areas.
  - The second then "drags" only one of the centres (which always wins the competition).
  - Result: One of the areas will be covered by a single centre even though it contains half of the mass of the input examples.

Solution: We tie centres together so that they have to move together.

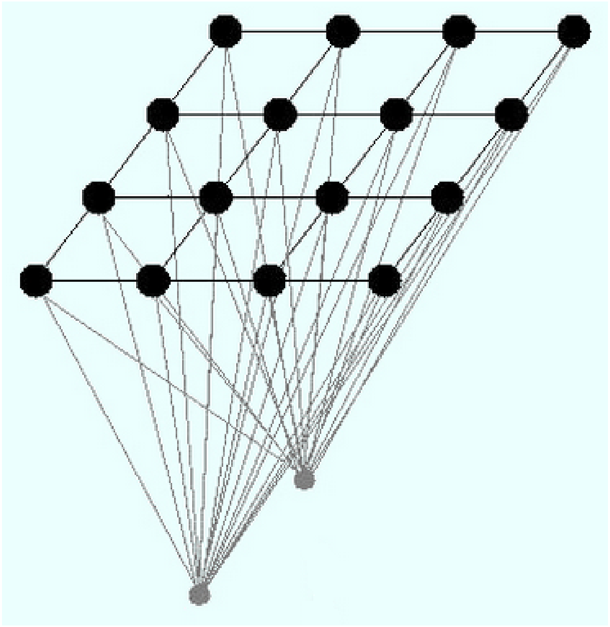
Kohonen's map

Architecture: Single layer



- **Topological structure**: neurons connected by edges so that they are nodes in an undirected graph.
- In most cases, this structure is either a one dimensional sequence or a two dimensional grid.

Kohonen's map – illustration



Kohonen's map – bio motivation

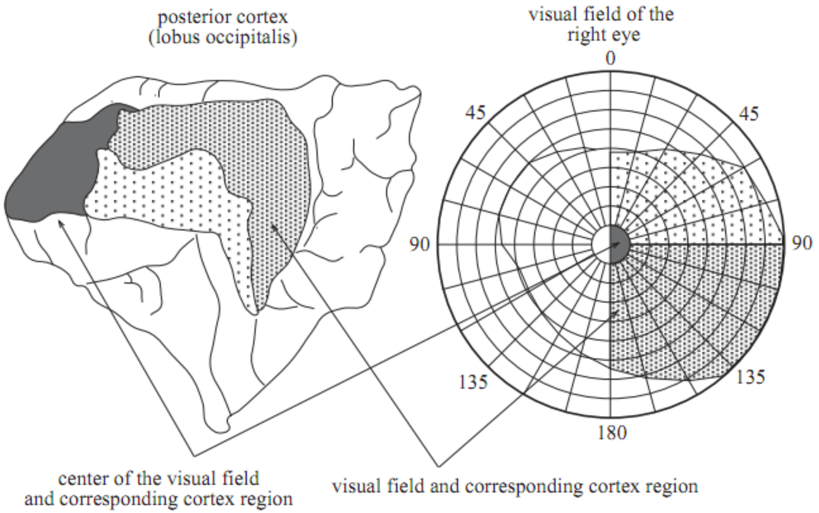

posterior cortex
(lobus occipitalis)
visual field of the
right eye
0
45
45
90
90
135
135
180
center of the visual field
and corresponding cortex region
visual field and corresponding cortex region

Fig. 15.2. Mapping of the visual field on the cortex

Source: Neural Networks - A Systematic Introduction, Raul Rojas, Springer, 1996

Kohonen's map

Activity: Given an input vector $\vec{x} \in \mathbb{R}^n$ and $k = 1, \ldots, h$:

$$y_k = \begin{cases} 1 & k = \arg\min_{i=1,\ldots,h} \left\| \vec{x} - \vec{w}_i \right\| \\ 0 & \text{jinak} \end{cases}$$

Learning: We use the topological structure.

- Denote by $d(c, k)$ the length of the shortest path from neuron c to neuron k in the *topological structure*.
- For every neuron c and a given $s \in \mathbb{N}_0$ define **topological neighbourhood** of the neuron c of size s : $N_s(c) = \{k \mid d(c, k) \leq s\}$

In step t, given training example $\vec{x}_t$ adapt $\vec{w}_k$ as follows:

$$\vec{w}_k^{(t)} = \begin{cases} \vec{w}_k^{(t-1)} + \theta \cdot \left(\vec{x}_t - \vec{w}_k^{(t-1)} \right) & k \in N_s(c(\vec{x}_t)) \\ \vec{w}_k^{(t-1)} & \text{otherwise} \end{cases}$$

where $c(\vec{x}_t) = \arg\min_{i=1,\ldots,h} \left\| \vec{x}_t - \vec{w}_i^{(t-1)} \right\|$ and $\theta \in \mathbb{R}$ and $s \in \mathbb{N}_0$ are parameters that may change during training.

Kohonen's map – learning

More general version:

$$\vec{w}_k^{(t)} = \vec{w}_k^{(t-1)} + \Theta(c(\vec{x}_t), k) \cdot \left(\vec{x}_t - \vec{w}_k^{(t-1)}\right)$$

where $c(\vec{x}_t) = \arg\min_{i=1,\dots,h} \left\| \vec{x}_t - \vec{w}_i^{(t-1)} \right\|$. The previous case then corresponds to

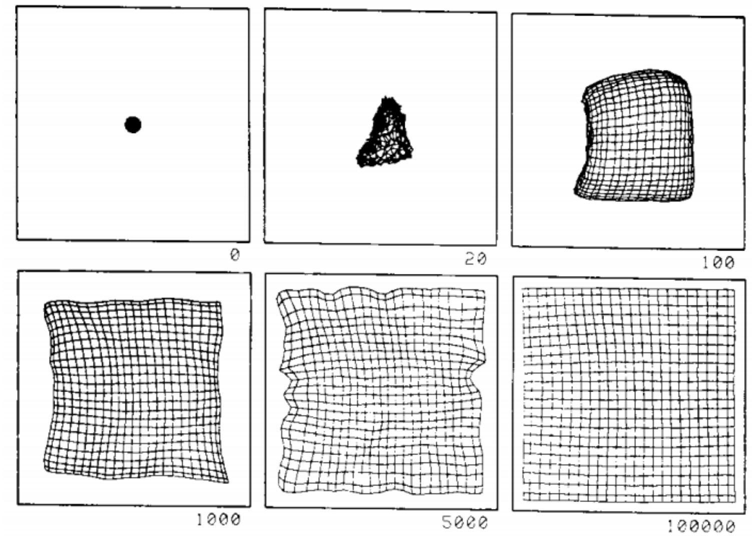
$$\Theta(c(\vec{x}_t), k) = \begin{cases} \theta & k \in N_s(c(\vec{x}_t)) \\ 0 & \text{jinak} \end{cases}$$

A smoother version:

$$\Theta(c(\vec{x}_t), k) = \theta_0 \cdot \exp\left(\frac{-d(c(\vec{x}_t), k)^2}{\sigma^2}\right)$$

where $\theta_0 \in \mathbb{R}$ is a learning rate and $\sigma \in \mathbb{R}$ is the width (both parameters may change during training).

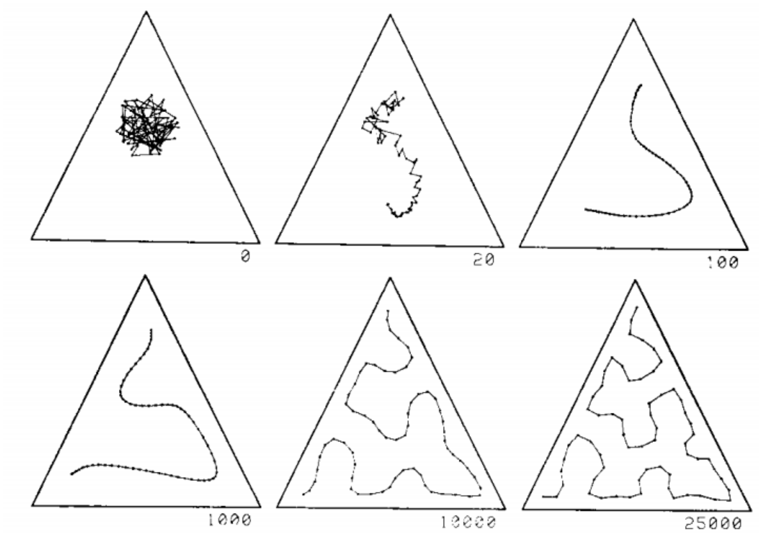
Example 1



Inputs uniformly distributed in a rectangle.

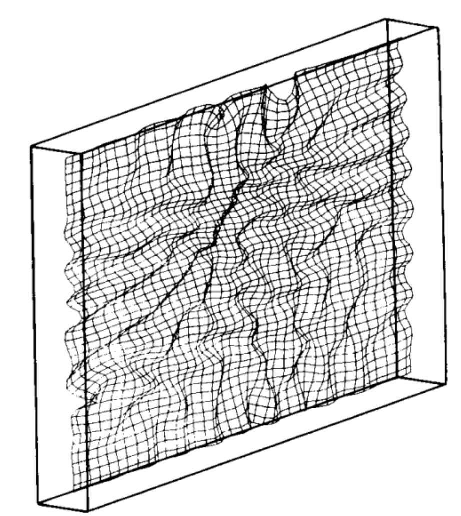
Zdroj obrázku: Neural Networks - A Systematic Introduction, Raul Rojas, Springer, 1996

Example 2



Inputs uniformly distributed in a triangle. Zdroj obrázku: Neural Networks - A Systematic Introduction, Raul Rojas, Springer, 1996

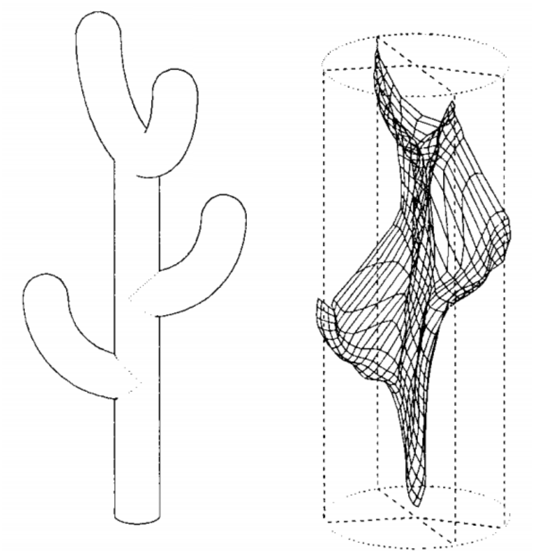
Example 3



Inputs uniformly distributed in a cuboid.

Zdroj obrázku: Neural Networks - A Systematic Introduction, Raul Rojas, Springer, 1996

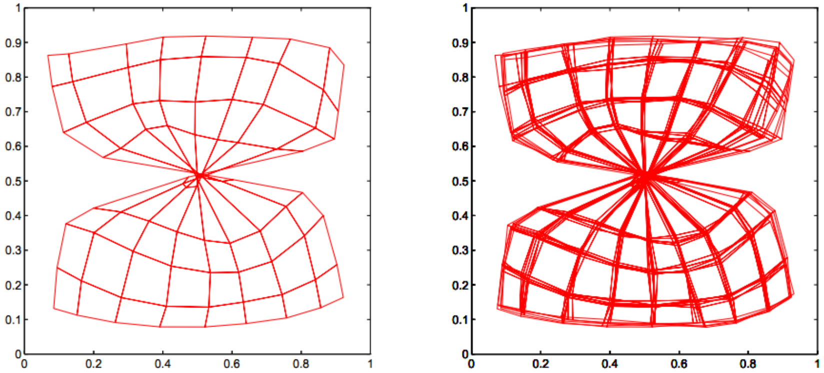
Example 4



Inputs uniformly distributed in a cactus.

Zdroj obrázku: Neural Networks - A Systematic Introduction, Raul Rojas, Springer, 1996

Example – defect



Topological defect – twisted network.

Zdroj obrázku: Neural Networks - A Systematic Introduction, Raul Rojas, Springer, 1996

Kohonen's map – practical approach

By Kohonen's paper: Inital weights are not so important, should be different from each other.

Two phase learning:

coarse phase:

- Approx. 1000 steps
- learning rate θ: start with 0.1 and steadily decrement to 0.01
- topological neighbourhood of every neuron (determined by *s* or by the width σ) should be large at the beginning (i.e. contain most neurons) and should shrink to few neurons at the end

fine tuning:

- number of steps: approx. 500 times the number of neurons
- θ close to 0.01 (otherwise topological defects are likely to occur)
- neighbourhood of each neuron should contain just few other neurons

Kohonen's map – theory

- Convergence to "ordered" state has been proved only for one dimensional maps and special cases of the distribution $p(\vec{x})$ (uniform), fixed neighbourhoods of size 1, and a fixed learning rate.

  There are simple counterexamples disproving convergence in case these assumptions are not satisfied.
- In more than one dimension there are no guarantees at all, convergence depends on several factors:
  - initial distribution of neurons (centres)
  - size of the neighbourhood
  - learning rate
- What dimension to choose? Typically one or two dimensional map is used (as a coarse version of dimensionality reduction).

LVQ – classification using Kohonen's map

Assume randomly generated training examples of the form $(\vec{x}_t, d_t)$ where $\vec{x}_t \in \mathbb{R}^n$ is **feature vector** and $d_t \in \{C_1, \ldots, C_q\}$ corresponds to one of the q **classes**.

Our goal is to classify objects based on our knowledge of their features, i.e. to every $\vec{x}_t$ assign a class so that the probability of error is minimized.

Ex.: Conveyor belt with fruits, apples and oranges: Formally, $(\vec{x}_t, d_t)$ where

- $\vec{x}_t \in \mathbb{R}^2$, here the first component is the weight and the second the diameter.
- d_t is either A or O depending on whether the given object is an apple or an orange.

We allow apples and oranges with the same features.

The goal is to sort out the fruits based on their weight and diameter.

Classification using Kohonen's map

We use Kohonen's map as follows:

1. Train the map on feature vectors $\vec{x}_t$ where $t = 1, \ldots, \ell$ (ignore the classes for now).
2. Label neurons with classes. The class v_c of a given neuron c is determined as follows:

   For every neuron c and every class C_i count the number $\#(c, C_i)$ of training examples $\vec{x}_t$ with class C_i for which the neuron c returns 1 (i.e. is the closest to them).

   To c, assign the class v_c satisfying

   $$v_c = argmax_{C_i} \#(c, C_i)$$

3. Fine tune the network using LVQ (see later)

The trained network is used as follows: Given a feature vector $\vec{x}$, evaluate the network with $\vec{x}$ as the input. A single neuron c has the value 1, return v_c as the class of $\vec{x}$.

LVQ

Iterate over training examples. For $(\vec{x}_t, d_t)$ find the closes neuron c

$$c = \arg \min_{i=1,\dots,h} \left\|\vec{x}_t - \vec{w}_i\right\|$$

Adjust weights of c as follows:

$$\vec{w}_c^{(t)} = \begin{cases} \vec{w}_c^{(t-1)} + \alpha(\vec{x}_t - \vec{w}_c^{(t-1)}) & d_t = v_c \\ \vec{w}_c^{(t-1)} - \alpha(\vec{x}_t - \vec{w}_c^{(t-1)}) & d_t \neq v_c \end{cases}$$

The parameter α should be small right from the beginning (approx. 0.01 – 0.02) and go to 0 steadily.

By Kohonen: The border between classes should be a good approximation of the Bayes decision boundary.

What is it??

Bayes classifier

For simplicity, consider two classes C_0 and C_1 (e.g. A and O).

Let $P(C_i \mid \vec{x})$ be the probability that the object belongs to C_i assuming that it has features $\vec{x}$.

(e.g. $P(A \mid (a, b))$ is the probability that a fruit with weight a and diameter b is an apple.)

Bayes classifier assigns to $\vec{x}$ the class C_i which satisfies $P(C_i \mid \vec{x}) \geq P(C_{1-i} \mid \vec{x})$.

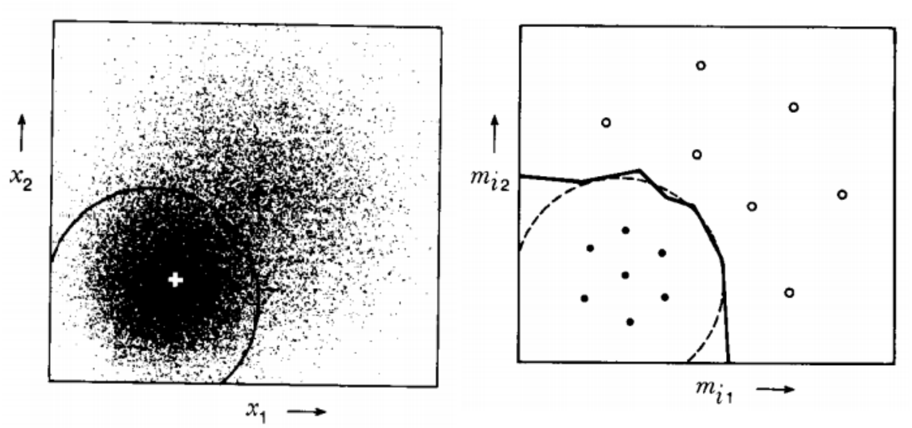
Denote by R_0 the set of all $\vec{x}$ satisfying $P(C_0 \mid \vec{x}) \geq P(C_1 \mid \vec{x})$ and $R_1 = \mathbb{R}^n \smallsetminus R_0$.

Bayes classifier minimizes the error probability:

$$P(\vec{x} \in R_0 \wedge C_1) + P(\vec{x} \in R_1 \wedge C_0)$$

Bayes decision boundary is the boundary between the sets R_0 and R_1.

Bayes decision boundary vs LVQ

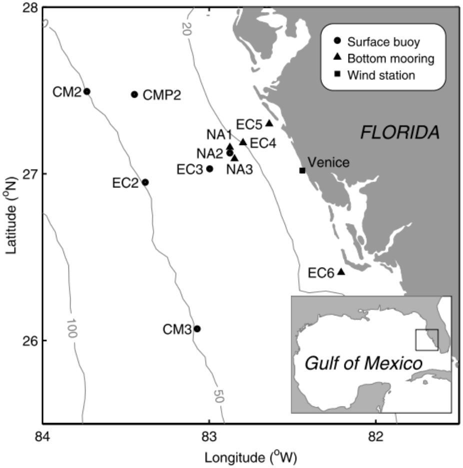


Zdroj obrázku: The Self-Organizing Map, Teuvo Kohonen, IEEE, 1990

Oceanographic data

Source: Patterns of ocean current variability on the West Florida Shelf using the self-organizing map. Y. Liu a R. H. Weisberg, JOURNAL OF GEOPHYSICAL RESEARCH, 2005

Investigates currents in the ocean around Florida.



Oceanographic data

- 11 measuring stations, 3 depths (surface, bottom, in between).
- data: 2D velocity vectors of the current
- measured by every hour, for 25585 hours

Thus we have 25585 data samples, 66 dimensions.

Kohonen's map:

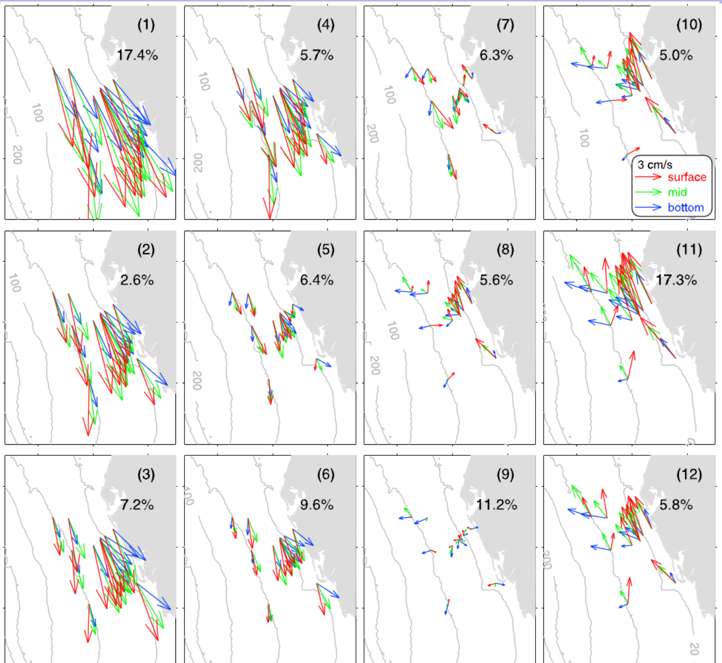
- grid 3×4
- neighbourhoods given by Gaussian functions

$$\Theta(c,k) = \theta_0 \cdot \exp\left(\frac{-d(c,k)^2}{\sigma^2}\right)$$

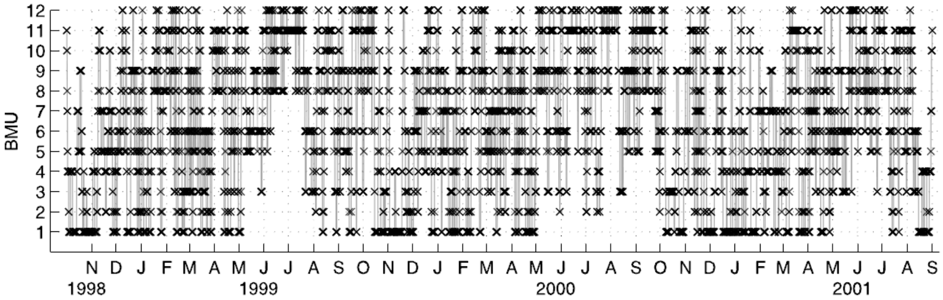
shrinking width

(linearly decreasing learning rate)

Oceanographic data



Oceanographic data



- crosses are winning neurons)
- influenced by local fluctuations
- observable trend:
  - winter: neurons 1-6 (south-east)
  - summer: neurons 10-12 (north-west)

Grimm's fairy tales

Zdroj: Contextual Relations of Words in Grimm Tales, Analyzed by Self-Organizing Map. T. Kohonen, T. Honkela a V. Pulkki, ICANN, 1995

Our goal is to visualize syntactic and semantic categories of words in fairy tales (depending on context).

Input: Grimm's fairy tales (understandably encoded using a stream of 270-dimensional vectors)

- triples of words (predecessor, key, successor)
- every component in the triple encoded using a randomly generated 90 dimensional real vector

Network: Kohonen's map, 42×36 neurons, weights of the form $w = (w_p, w_k, w_n)$ where $w_p, w_k, w_n \in \mathbb{R}^{90}$.

Grimm's fairy tales

Learning:

Trained on triples of successive words in fairy tales

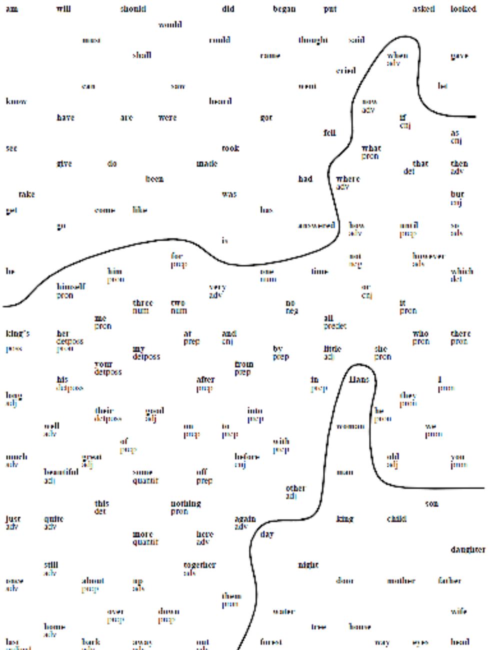
The training set consisted of 150 most common words, with "average" context.

Coarse training: 600 000 iterations; Fine tuning: 400 000

In the end, 150 most common words labelled neurons:

A word u labels a neuron with weights $w = (w_p, w_k, w_n)$ when w_k is closest to the code of u.

Grimm's fairy tales



Great summary – models

We have considered several models of neural networks:

- ADALINE (aka linear regression)
- Multilayer Perceptron
- Hopfield Networks
- Restricted Boltzmann Machines and Deep Belief Networks
- Convolutional Networks
- Recurrent Networks (LSTM)
- Kohonen's Maps

Great summary – algorithms

Gradient descent!

The only exception were Kohonen's maps (Kohonen learning) and Hopfield (Hebb's learning).

The gradient computed using

- Backpropagation: MLP, Convolutional, Recurrent (LSTM)
- Simulations: RBM

Deeper thoughts

- Most neural network models are universal approximators (i.e. capable of approximating any reasonable function), but it is difficult to find the appropriate configuration $\rightarrow$ such configuration can be learned efficiently (without guarantees of course)
- Depth is stronger than size: deep networks are more succinct in their representation but are harder to train: Do not forget the vanishin/exploding gradient problem!
- The way how backprop is derived: Unification of all neurons using indices, backprop for models then differs very little, only in specification of neurons with tied weights!
- Weight tying = single most effective trick in the history of neural networks!