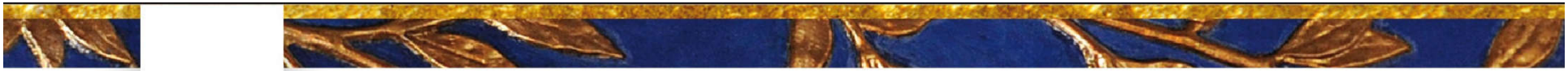


UPPSALA
UNIVERSITET

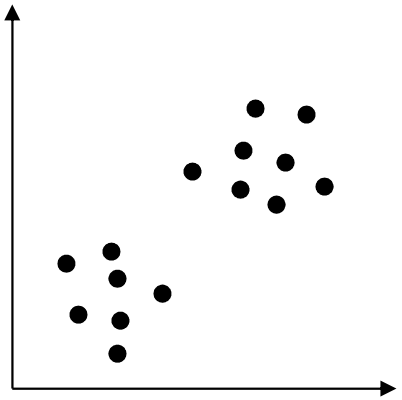
Natural Computation Methods in Machine Learning (NCML)

Lecture 9: Unsupervised Learning

UPPSALA UNIVERSITET

Unsupervised Learning

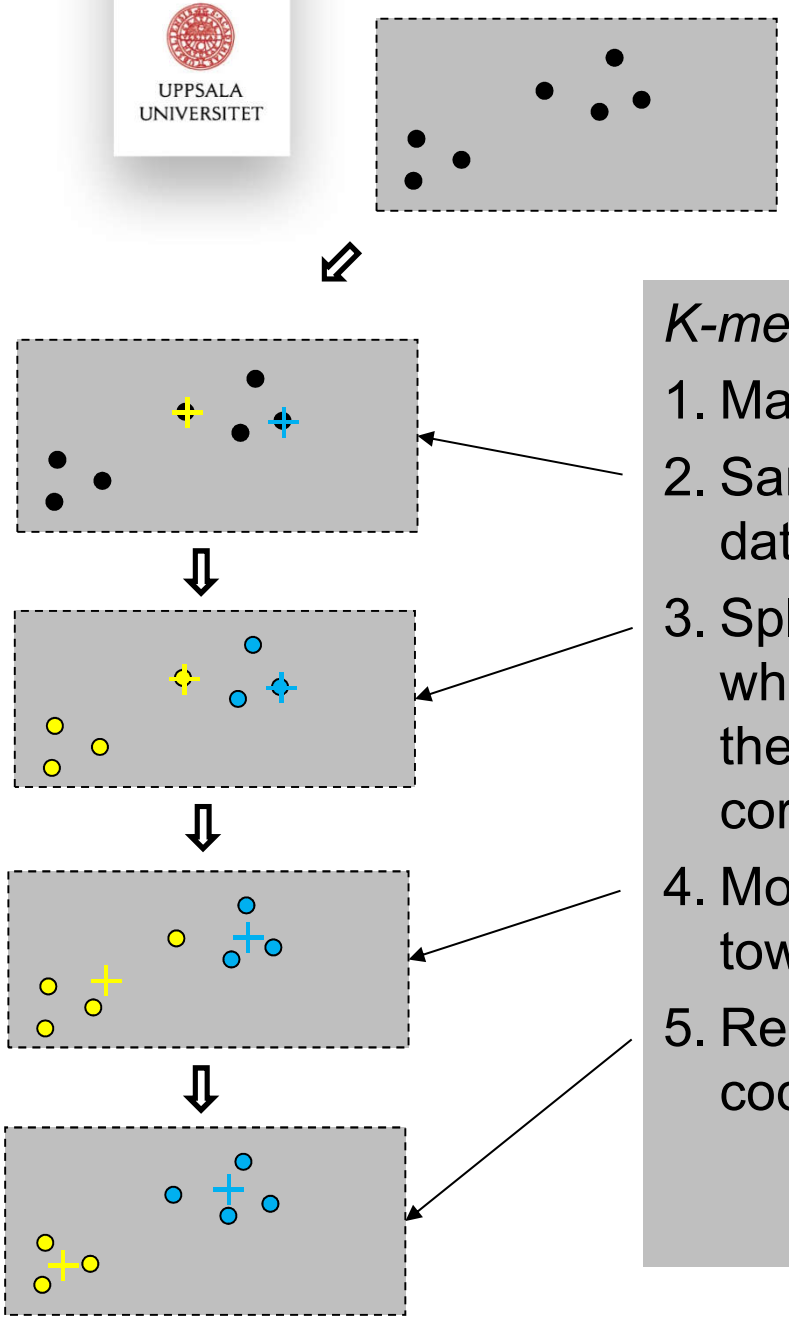
- Learning to classify (usually) from unlabelled data
  - Only inputs, no target information, no 'rewards'
- Requires that class membership can be decided by structural properties (features) in the data
  - and that the learning system can find those features
- We typically assume that data close to each other belong to the same class
  - Clustering



UPPSALA UNIVERSITET

K-means

Steinhaus 1956, MacQueen 1967



K-means, for K=2

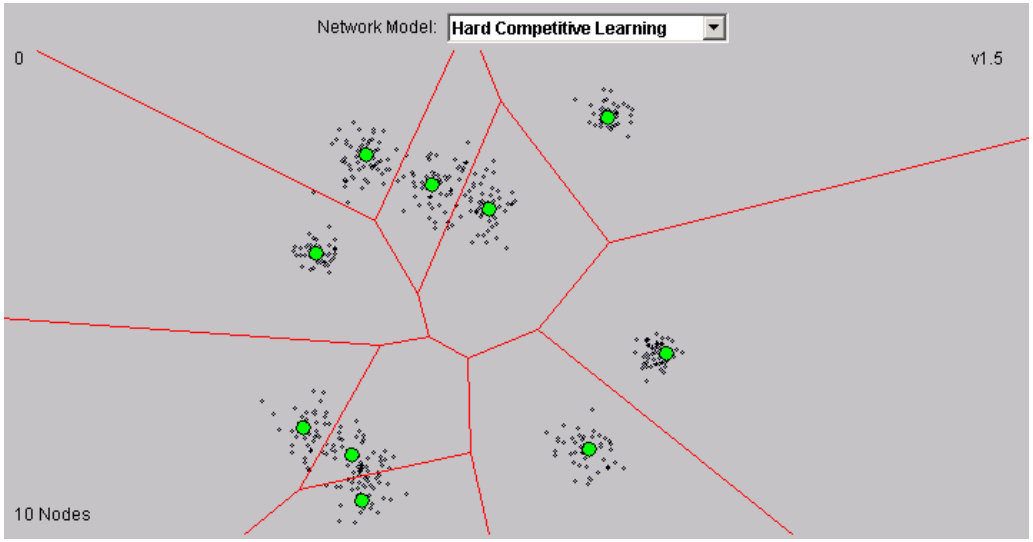
1. Make a 'codebook' of two vectors, c_1 and c_2
2. Sample (at random) two vectors from the data as initial values of c_1 and c_2
3. Split the data in two subsets, D_1 and D_2, where D_1 is the set of all points with c_1 as their closest codebook vector, and D_2 is the corresponding set for c_2
4. Move c_1 towards the mean in D_1 and c_2 towards the mean in D_2
5. Repeat from 3 until convergence (until the codebook vectors stop moving)

(codebook vector = centroid)

UPPSALA UNIVERSITET

Voronoi Regions

- *K*-means define *Voronoi regions* in the input space
- The Voronoi region of a codebook vector c_i is the region in which c_i would be the closest one



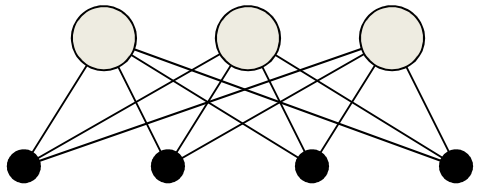
Voronoi regions around 10 codebook vectors (green) in Euclidean space

UPPSALA UNIVERSITET

Competitive Learning

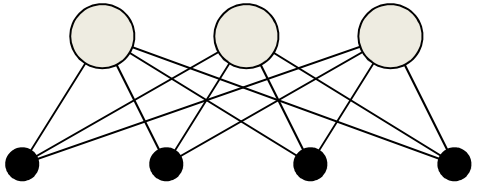
LVQ-I without neighbourhood function

- Let's say we want to classify N-dimensional data into M classes
  - i.e. identify M clusters
- Create a layer of M nodes, fully connected to the inputs (as usual)



- We can think of the nodes as neurons
  - though they (usually) don't compute weighted sums
- A weight vector can be plotted as a position in space
  - in this case the same N-dimensional space as the inputs, since there are no threshold/bias weights
  - When we change weights, we move that position around
  - Each node represents a codebook vector (as in K-means)

UPPSALA UNIVERSITET



Competitive Learning

- Given an input vector, $\overline{x}$, find the closest node, k
  - The node with the smallest distance between its weight vector and the input vector
  - We call node k the "winner"
- Works with weighted sums too! (if you wish)
  - If the weight vectors are normalized to unit length, the closest node is the one with the greatest weighted sum
- We want to make node k even more likely to win for the same input vector, next time it shows up
  - Move it towards the input vector!

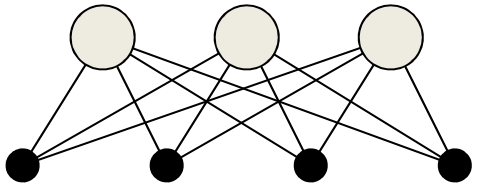
$$\Delta\overline{w}_k = \eta(\overline{x} - \overline{w}_k) = \eta(x_i - w_i), \text{ where } 1 \leq i \leq N$$

  - This is the *Standard Competitive Learning* rule
  - Note that only the winner is moved!

UPPSALA UNIVERSITET

Competitive Learning

Summary



Usual interpretation

1. Present a pattern (sample), $\overline{x}$
2. Find the closest node, k, i.e. the node with the closest weight vector to $\overline{x}$
3. The weights of node k, *the winner*, is moved towards the input vector $\overline{x}$. All other weights are left unchanged

Neural interpretation

1. Present a pattern (sample), $\overline{x}$
2. Find the node, k, with the greatest weighted sum for $\overline{x}$ (no thresholds)
3. Update the weights of node k to increase the weighted sum for $\overline{x}$. All other weights are left unchanged

The standard competitive learning rule

$$\Delta\overline{w}_k = \eta(\overline{x} - \overline{w}_k)$$

weight vector = codebook vector

Competitive learning + epoch learning = *K*-means!

UPPSALA UNIVERSITET

The 'winner takes all' scenario

- Problem: If a single node wins a lot, it may become invincible!

In this example, the first node to win, will always win

- Solution 1: Initialize the weight vectors by drawing vectors at random from the data (as in K-means)
  - If we draw them at random, they will all be close to origo, and that's not necessarily where the data is
- Solution 2: Modify the distance measure to include the frequency of winning

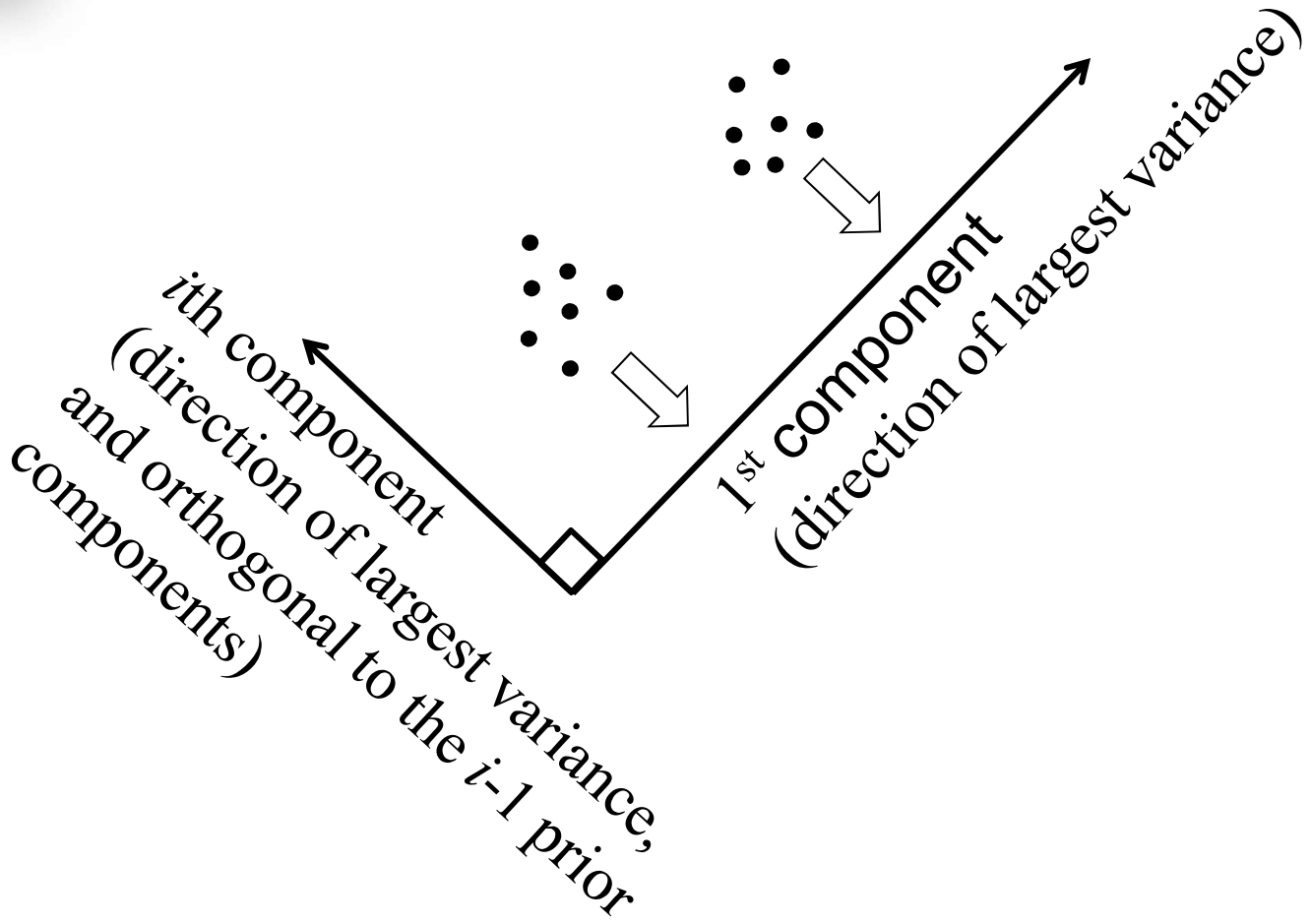
UPPSALA UNIVERSITET

Reducing dimensionality

- Sometimes we have data of high dimensionality which we want to reduce
  - As a preprocessing stage for later classification
  - Or just to make the data presentable to humans (2D)
- We want to project the data down to some sub-space of lower dimensionality
  - hopefully without destroying class information
- Principal Component Analysis
- Kohonen's Self-Organizing Feature Maps

UPPSALA UNIVERSITET

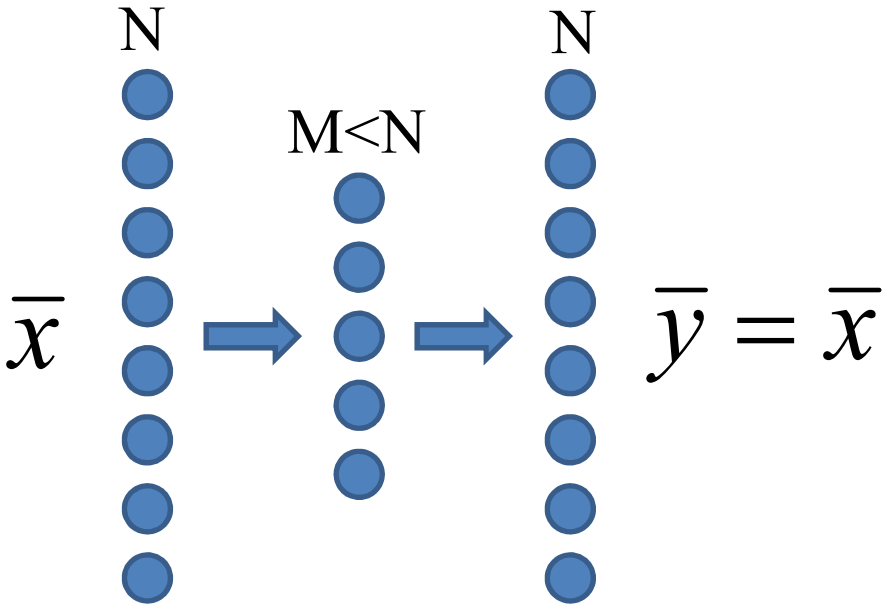
Principal Component Analysis



The principal components are eigenvectors of the correlation matrix of the input data, corresponding to the *M* largest eigenvalues.

UPPSALA UNIVERSITET

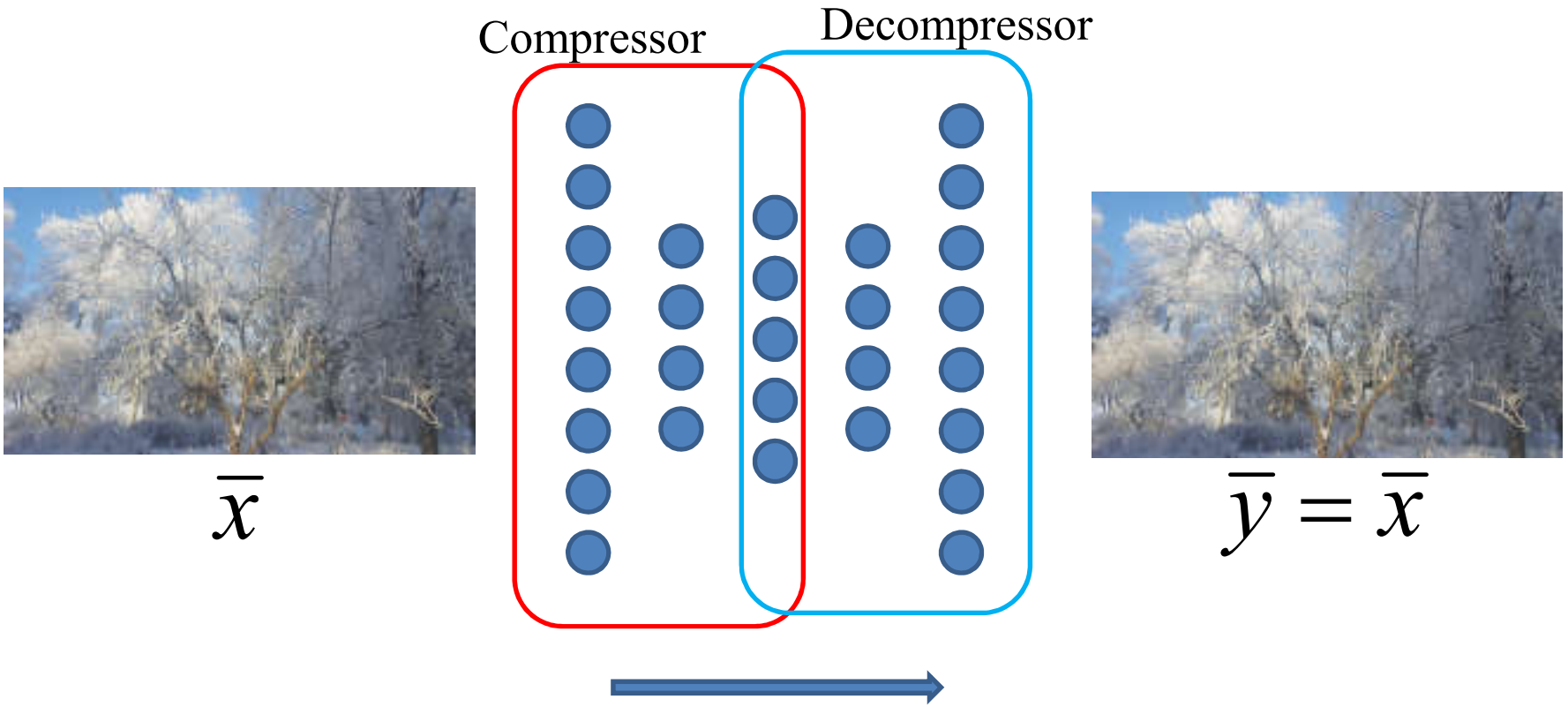
Auto-encoders (Neural PCA)



- M <u>linear</u> hidden nodes → M first principal components
- Non-linear hidden nodes → more complex mappings
- Compression/decompression, encryption/decryption
- Novelty detection
- Pre-training of hidden layers in Deep Learning

UPPSALA UNIVERSITET

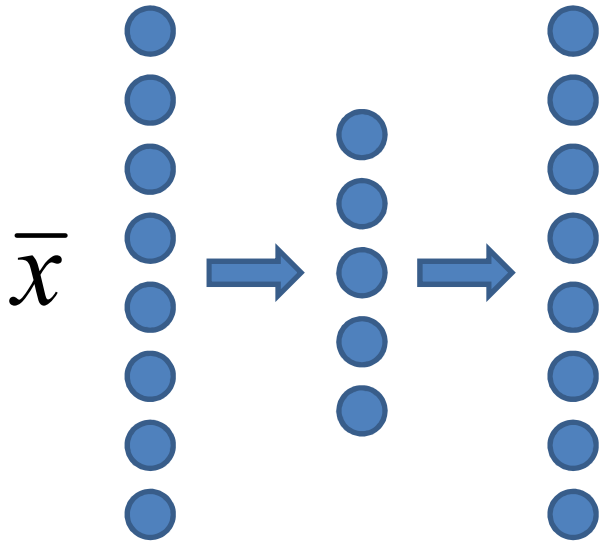
Auto-encoders for compression (and encryption)



Requires at least 3 hidden layers
(the two parts need at least one hidden layer each)

UPPSALA UNIVERSITET

Auto-encoders for novelty detection



$$\bar{y} = \bar{x}$$

$$Err = \sum_{i=0}^{N} (x_i - y_i)^2$$

- Train on normal cases only
- Then measure error for test cases
- Unexpected case ➔ Greater error
- If error > limit ➔ Alarm!

UPPSALA UNIVERSITET

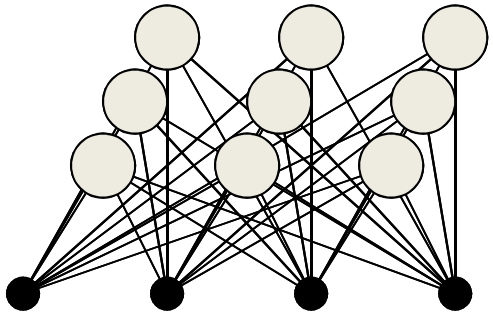
Self-Organizing Feature Maps

Kohonen, 1984

- Based on two observation of the human cortex
  - We humans reason using the cerebral cortex, which is essentially two dimensional. Still we can reason in more dimensions than two!
    - Indicates some form of dimension reduction is going on
  - Cells in the auditory cortex, which respond to certain frequencies, are located in frequency order
    - Topological preservation / topographic map
- SOFM = Non-linear, topologically preserving, dimension reduction
  - Like a fish-eye lens photo

UPPSALA UNIVERSITET

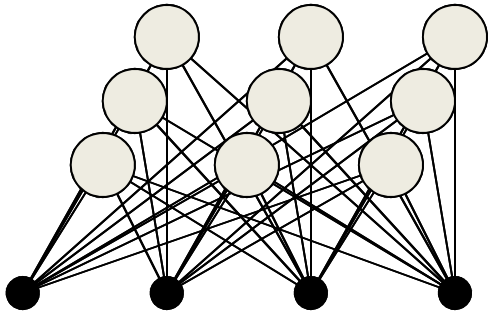
Self-Organizing Feature Maps



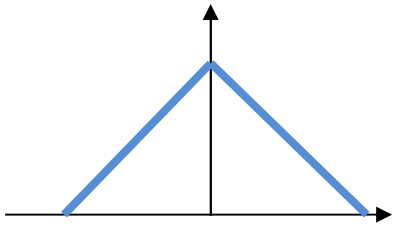
- A 2D (usually) grid of 'neurons' with common inputs. The map.
- Extend Competitive Learning to update not only the winner's weights, but also it's closest neighbour nodes (on the map)
- Note that we now must compute distances in two different (but commonly confused) spaces:
  - The winner is found by measuring distances between weight vectors and input vectors in the input space
  - Which nodes to update, with the winner, is decided by computing distances on the map
- Topologically preserving = when two vectors that are close in the input space are close also on the map

UPPSALA UNIVERSITET

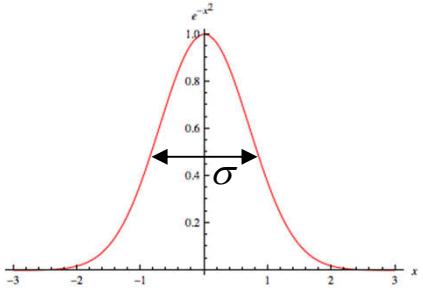
Self-Organizing Feature Maps



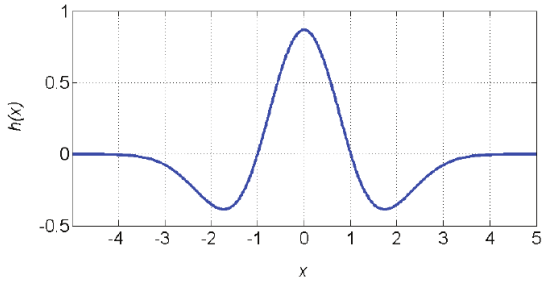
- We probably should not move a neighbour node as much as the winner (node k)
- Define a *neighbourhood function* $f(j, k)$ which is 1 for the winner itself ($j = k$) and then decreases with the distance (on the map) from node k.
- Examples:



Linear



Gaussian



'Mexican hat'

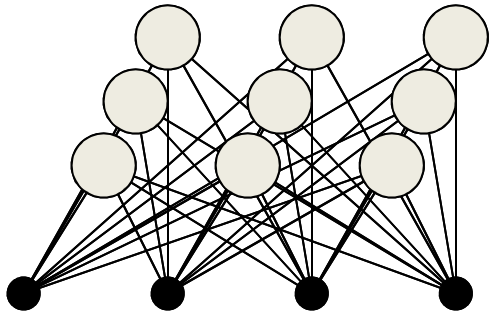
UPPSALA UNIVERSITET

Self-Organizing Feature Maps

- The neighbourhood function is used to modify the step length
- Extend the competitive learning rule:

$$\Delta \overline{w}_j = \eta f(j,k)\left(\overline{x} - \overline{w}_j\right), \text{ for all nodes } j$$

- In your course book, the gain factor, η, may seem to be missing, but it is embedded in the definition of $f()$



UPPSALA UNIVERSITET

Self-Organizing Feature Maps

Implementation

1. Initialize the network (set all weights to small random values)
2. Present input vector, $\overline{x}$
3. For each node, j, compute the distance, d_j, between its weight vector and the input vector

$$d_j = \sum_{i=1}^{N} \left(x_i - w_{ji}\right)^2$$

where

N is the number of inputs,

x_i is the value of input i

w_{ji} is the weight from input i to node j

UPPSALA UNIVERSITET

Self-Organizing Feature Maps

Implementation

4. Find the node, k, which is closest to the current input vector (lowest value of d)
5. Update the weights of all nodes by

$$w_{ji} := w_{ji} + \eta f(j,k)\left(x_i - w_{ji}\right)$$

where

η is the learning rate

$f(j, k)$ is the neighbourhood function, usually Gaussian

6. Repeat from 2

In summary:

1. Find the closest matching node to the input
2. Increase the similarity of this node, and those in its neighbourhood, to the input

UPPSALA UNIVERSITET

Self-Organizing Feature Maps

- Good idea to decrease both step length (η) and neighbourhood radius/width (σ) over time
- SOFM is often trained in two phases:
  1. A coarse *Ordering phase*, to find the number of classes and their approximate locations on the map.

     Start with a large η and a large neighbourhood. The radius is typically reduced from the diameter of the map to 1
  2. A *Tuning phase*, to decide the exact location and form of the classes on the map

     Start with low value of η. Radius typically reduced from 3 to 1. About 10x longer training time than in the ordering phase
- After training, the active areas on the map can be labelled by presenting patterns for which the generating class is known

UPPSALA UNIVERSITET

Self-Organizing Feature Maps

"Topologically preserving"

=

Input vectors that are close (in that space), usually activate nodes that are close on the map

=

The weights form a density function of the data

=

Weight vectors are distributed as the data

(for example, uniformly distributed input vectors ➔ uniformly distributed weight vectors

=

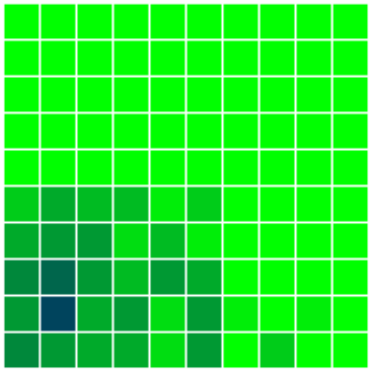
Garbage in ➔ Garbage out

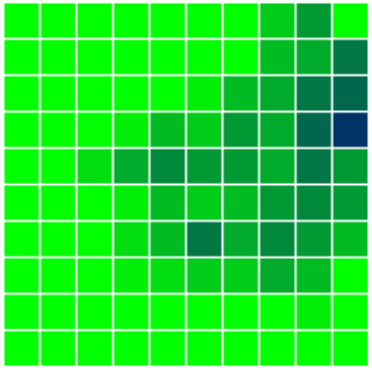
UPPSALA UNIVERSITET

Self-Organizing Feature Maps

Applications

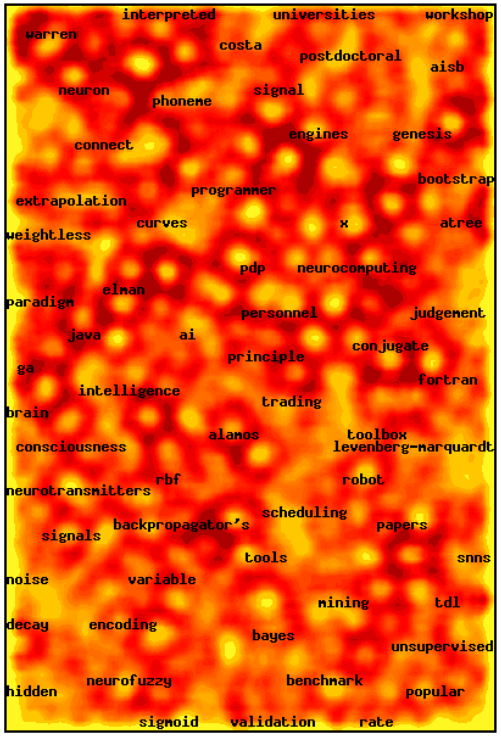
"Online"





- Visualisation
- Image analysis
- Recommender systems
- Fraud detection
- Kohonen's "phonetic typewriter"
- Preprocessing for MLP, to fill in missing values
- Preprocessing for classification to decide the number of classes

"Offline"

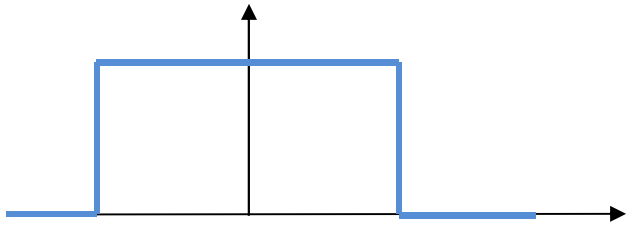


UPPSALA UNIVERSITET

SOFM in Matlab

Lab 3

- The map is hexagonal, not square
- Epoch learning, instead of pattern learning
- The neighbourhood function is a tophat function!



- Step length, η, is 1!
  - This means that all neighbours in a radius around the winner, are moved to the input (not just towards it)!
  - At first glance, this should not work!
  - But in combination with the use of epoch learning, it does! (since weight changes are accumulated over the whole training set, before actually applied)