
Neural Networks

doc. RNDr. Iveta Mrázová, CSc.

DEPARTMENT OF THEORETICAL COMPUTER SCIENCE AND MATHEMATICAL LOGIC
FACULTY OF MATHEMATICS AND PHYSICS, CHARLES UNIVERSITY IN PRAGUE

Neural Networks:

Associative Memories

doc. RNDr. Iveta Mrázová, CSc.

DEPARTMENT OF THEORETICAL COMPUTER SCIENCE AND MATHEMATICAL LOGIC
FACULTY OF MATHEMATICS AND PHYSICS, CHARLES UNIVERSITY IN PRAGUE

Neural Networks:

Contents:

- Multi-layered Neural Networks
- **Associative Memories**

Contents:

- Multi-layered Neural Networks
 - Multi-layered Neural Networks: Analysis of Their Properties
 - Multi-layered Neural Networks: an Application Example
- **Associative Memories**
 - Associative Memories: BAM – The Bidirectional Associative Memory
 - Associative Memories: Hopfield Networks

Neural Networks:

Contents:

- Multi-layered Neural Networks
- **Associative Memories**
- Associative Memories: BAM – The Bidirectional Associative Memory

Contents:

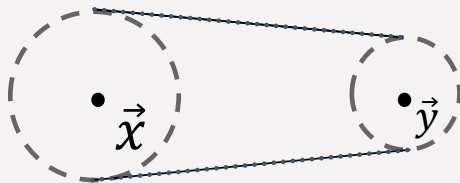
- Multi-layered Neural Networks
- **Associative Memories**
 - The Structure of Associative Memories
 - Eigenvector Automata
 - Hebbian Learning
 - The Capacity Problem
 - Pseudoinverse Matrix
- Associative Memories: BAM – The Bidirectional Associative Memory
 - Training and Recall
 - Energy Function
 - Convergence to Stable States

Associative Networks and Associative Memories (1)

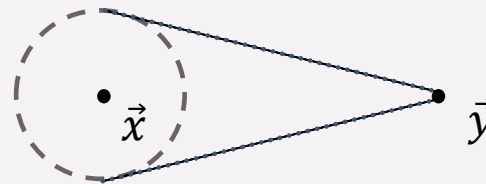
The goal of learning:

- ~ associate known input vectors with the given output vectors
- The neighborhood of a known input pattern \vec{x} should also be mapped to the image \vec{y} of \vec{x}
 - „noisy“ input vectors can then be associated with the correct output

BP-network



associative network



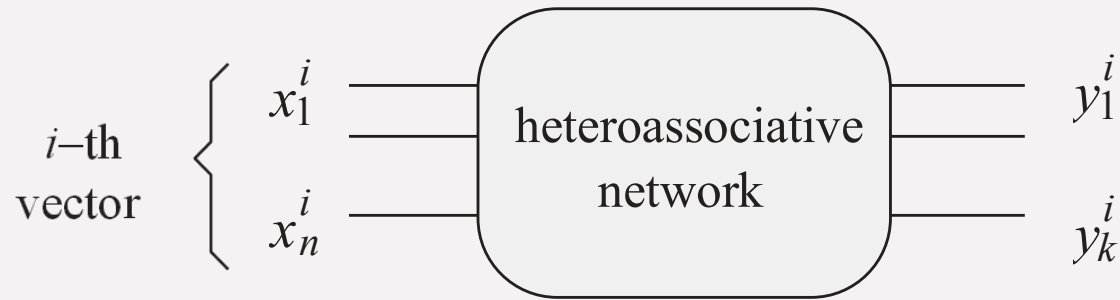
Associative Networks and Associative Memories (2)

- Associative memories can be implemented using networks with (or without) feedback
 - the simplest kind of feedback:
 - use the output of the network repetitively as a new input until the process converges to a stable state
- × but not all networks converge to a stable state after presenting a new input pattern
 - additional restrictions on the network architecture are necessary

The Function of an Associative Memory

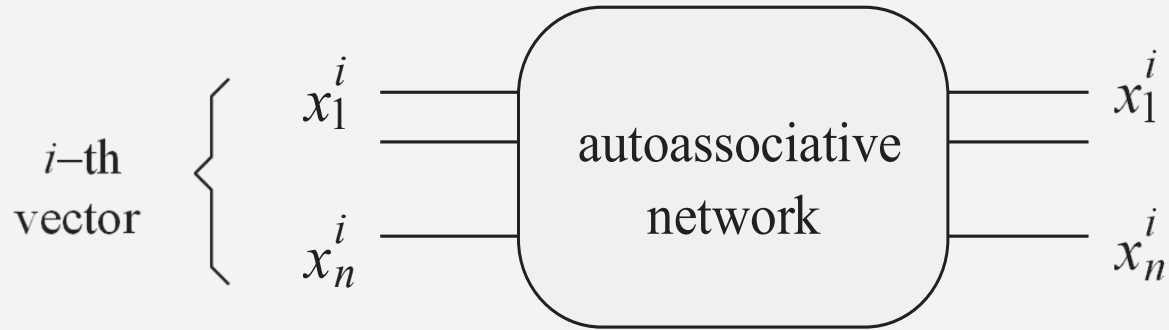
- Recognize previously learned input vectors, even if some noise has been added
- The response of each neuron is determined exclusively by the information flowing through its own weights (Hebbian learning)
- Three types of associative networks:
 - heteroassociative, autoassociative and pattern recognition networks

Heteroassociative Networks



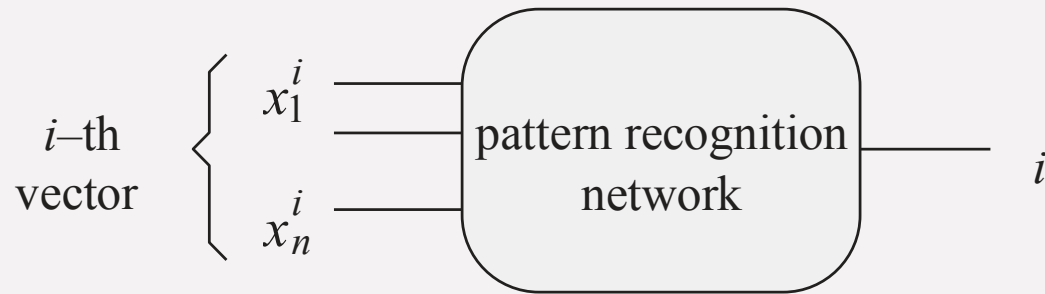
Map m input vectors $\vec{x}^1, \dots, \vec{x}^m$ from the n -dimensional space to m output vectors $\vec{y}^1, \dots, \vec{y}^m$ in the k -dimensional space, so that $\vec{x}^i \mapsto \vec{y}^i$. If $\|\tilde{\vec{x}} - \vec{x}^i\|^2 < \varepsilon$, then $\tilde{\vec{x}} \mapsto \vec{y}^i$ ($\varepsilon > 0$).

Autoassociative Networks



- A special subset of the heteroassociative networks (each vector is associated with itself: $\vec{y}^i = \vec{x}^i$ for $i = 1, \dots, m$).
- The function of autoassociative networks is to „correct noisy input patterns“.

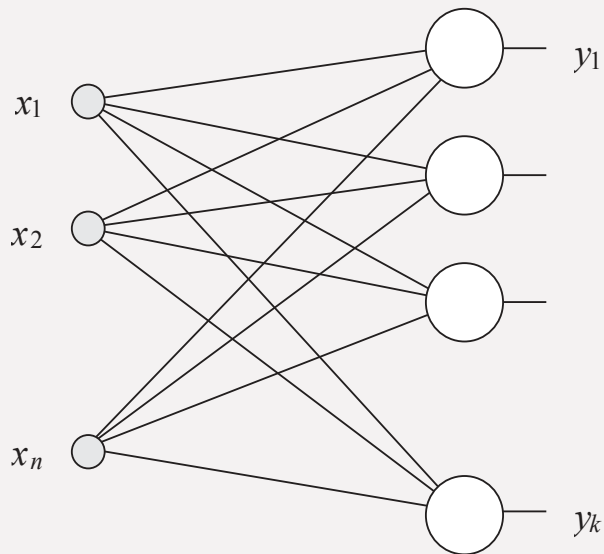
Pattern Recognition Networks



- A special type of heteroassociative networks (each vector \vec{x}^i is associated with the scalar value i).
- The goal is to identify the class of the input pattern

The Structure of an Associative Memory

Heteroassociative network without feedback



Associative memories can be implemented using a single layer of neurons

- Let: w_{ij} ... the weight between the input i and neuron j
 W ... the $n \times k$ weight matrix

→ vector $\vec{x} = (x_1, x_2, \dots, x_n)$ yields the excitation vector $\vec{e} = \vec{x} \cdot W$

→ Afterwards, the value of the transfer function is computed for each neuron

- For the identity, we get a linear associator and the output \vec{y} is just $\vec{x} \cdot W$

Structure of an Associative Memory (2)

In general: m different n -dimensional vectors $\vec{x}^1, \vec{x}^2, \dots, \vec{x}^m$ have to be associated with m k -dimensional vectors $\vec{y}^1, \dots, \vec{y}^m$

→ X $m \times n$ matrix (rows correspond to the respective input vectors)

Y $m \times k$ matrix (rows correspond to the output vectors)

→ *we are looking for such a weight matrix W , for which $X \cdot W = Y$*

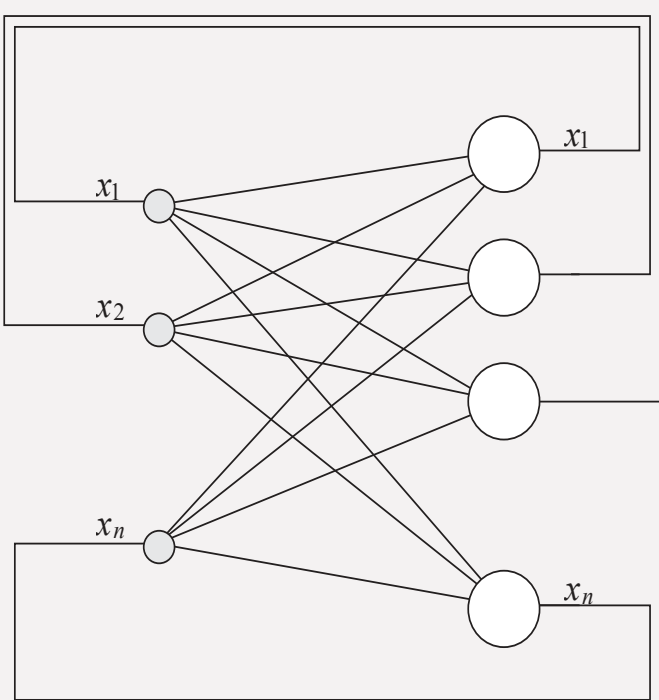
(and in the case of autoassociative memories: $X \cdot W = X$)

Remark: if $m = n$, then X is a square matrix

if it is invertible, the solution will be $W = X^{-1} \cdot Y$

Recurrent Associative Network

Autoassociative network with feedback



Network output is used as the new input

Assumption: all neurons compute their outputs simultaneously

→ in each step, the network is fed an input vector $\vec{x}(i)$ and produces a new output $\vec{x}(i + 1)$

Recurrent Associative Network (2)

Question: *is there a fixed point $\vec{\xi}$ such that $\vec{\xi} \cdot W = \vec{\xi}$?*

- the vector $\vec{\xi}$ is an eigenvector of the matrix W with the eigenvalue 1
- the network behaves as a first-order dynamical system, since each new state $\vec{x}(i + 1)$ is completely determined by its most recent predecessor

Eigenvector Automata

- Let W be the weight matrix of an autoassociative network, the individual neurons are linear associators

→ *look for fixed points of the dynamical system*

Remark: not all weight matrices lead to a stable state

Example: rotation by 90° in two-dimensional space:

$$W = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$$

→ cycles of length 4

Eigenvector Automata (2)

→ Quadratic matrices with a complete set of eigenvectors are more useful as memories

an $n \times n$ matrix W has at most n linearly independent eigenvectors and n eigenvalues

→ the eigenvectors $\vec{x}^1, \dots, \vec{x}^n$ then satisfy: $\vec{x}^i \cdot W = \lambda_i \vec{x}^i$
for $i = 1, \dots, n$ and the matrix eigenvalues $\lambda_1, \dots, \lambda_n$

Eigenvector Automata (3)

- Each weight matrix with a full set of eigenvectors defines an „eigenvector automaton“
 - given an initial input vector, the eigenvector with the largest eigenvalue can be found (if it exists)
- Assume, w.l.o.g., that λ_1 is the eigenvalue of W with the largest magnitude:

$$|\lambda_1| > |\lambda_i| \quad \forall i = 2, \dots, n$$

Eigenvector Automata (4)

- Let $\lambda_1 > 0$ and \vec{a}_0 be an n -dimensional randomly chosen non-zero vector

→ \vec{a}_0 can be expressed as a linear combination of the n eigenvectors of the matrix W :

$$\vec{a}_0 = \alpha_1 \vec{x}^1 + \alpha_2 \vec{x}^2 + \cdots + \alpha_n \vec{x}^n$$

- Assumption: all constants α_i are non-zero

→ After the first iteration with the matrix W we get:

$$\begin{aligned} \vec{a}_1 &= \vec{a}_0 \cdot W = (\alpha_1 \vec{x}^1 + \cdots + \alpha_n \vec{x}^n) \cdot W = \\ &= \alpha_1 \lambda_1 \vec{x}^1 + \alpha_2 \lambda_2 \vec{x}^2 + \cdots + \alpha_n \lambda_n \vec{x}^n \end{aligned}$$

Eigenvector Automata (5)

→ After t iterations the result is:

$$\vec{a}_t = \alpha_1 \lambda_1^t \vec{x}^1 + \alpha_2 \lambda_2^t \vec{x}^2 + \dots + \alpha_n \lambda_n^t \vec{x}^n$$

→ After a big enough number of iterations, the eigenvalue with the largest magnitude will dominate – λ_1

→ the vector \vec{a}_t can thus be brought arbitrarily close to the eigenvector \vec{x}^1 (with respect to the direction, not length)

→ in each iteration, the vector \vec{x}^1 attracts any other vector \vec{a}_0 with a non-zero component for α_1

→ \vec{x}^1 is an attractor

Eigenvector Automata (6)

Example:

- The matrix $W = \begin{pmatrix} 2 & 0 \\ 0 & 1 \end{pmatrix}$ has 2 eigenvectors, $(1, 0)$ and $(0, 1)$ with the respective eigenvalues 2 and 1.
- After t iterations, any initial vector (x_1, x_2) ; $x_1 \neq 0$ will be transformed into the vector $(2^t x_1, x_2)$.
 - For large enough t this vector will come arbitrarily close to $(1, 0)$
 - => the vector $(1, 0)$ is an attractor

Associative Learning

Goal: use associative networks as dynamical systems, whose attractors are exactly those vectors we would like to store in the memory

- **During network design, locate as many attractors in the input space as possible**
 - each one of them should have a well-defined and bounded influence region
 - ✗ in the case of the linear eigenvector automaton, just one vector absorbs almost the whole input space

Associative Learning (2)

→ a nonlinear dynamical systems

- *Nonlinear activation of neurons*

Hard-limiting transfer function:

$$\text{sgn}(x) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}$$

- *Bipolar coding* is better than the binary one
(bipolar vectors have a greater probability of being mutually orthogonal than binary vectors)

Hebbian Learning

Assumption:

- single-layer network of k neurons with the *sgn* transfer function

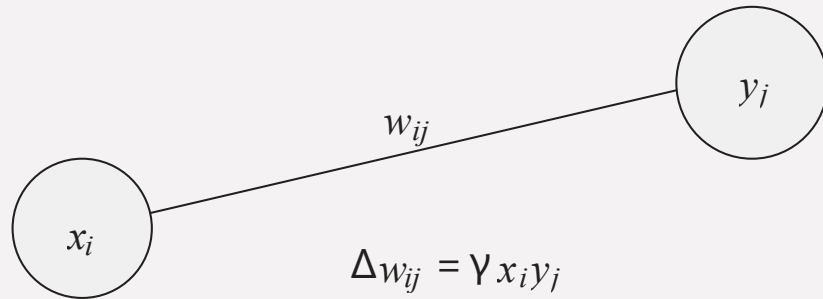
Goal:

- Find the appropriate weights to map the n -dimensional input vector \vec{x} to the k -dimensional output vector \vec{y}

Idea: (Donald Hebb – 1949)

- Two neurons, which are simultaneously active, should develop a degree of interaction higher than those neurons, whose activities are uncorrelated. In the latter case, the interaction between the elements should be very low or zero.

Hebbian Learning (2)



The Hebb rule: $\Delta w_{ij} = \gamma x_i y_j$

γ ... learning parameter

W ... weight matrix (initialized to zeroes)

The adjustment will be applied to all weights:

- at the input is the n -dimensional vector \vec{x}^1 , at the output is the k -dimensional vector \vec{y}^1
- the updated weight matrix W is the correlation matrix for these two vectors:
- $$W = [w_{ij}]_{n \times k} = [x_i^1 y_j^1]_{n \times k}$$

Hebbian Learning (3)

- the matrix W maps the non-zero vector \vec{x}^1 exactly to the vector \vec{y}^1

$$\begin{aligned}\vec{x}^1 \cdot W &= \left(y_1^1 \sum_{i=1}^n x_i^1 x_i^1, y_2^1 \sum_{i=1}^n x_i^1 x_i^1, \dots, y_k^1 \sum_{i=1}^n x_i^1 x_i^1 \right) = \\ &= \vec{y}^1 (\vec{x}^1 \cdot \vec{x}^1)\end{aligned}$$

- for $\vec{x}^1 \neq 0$, it holds that $\vec{x}^1 \cdot \vec{x}^1 > 0$ and the output of the network is:

$$\text{sgn}(\vec{x}^1 \cdot W) = (y_1^1, \dots, y_k^1) = \vec{y}^1$$

- for $-\vec{x}^1$, the output of the network is:

$$\text{sgn}(-\vec{x}^1 \cdot W) = -\text{sgn}(\vec{x}^1 \cdot W) = -\vec{y}^1$$

Hebbian Learning (4)

In general:

- If we want to associate m n -dimensional non-zero vectors $\vec{x}^1, \vec{x}^2, \dots, \vec{x}^m$ with m k -dimensional vectors $\vec{y}^1, \dots, \vec{y}^m$, we apply Hebbian learning to each INPUT/OUTPUT pair

- The resulting weight matrix W will have the form:

$$W = W^1 + W^2 + \dots + W^m,$$

where each matrix W^l is the $n \times k$ correlation matrix of the vectors \vec{x}^l and \vec{y}^l : $W^l = [x_i^l y_j^l]_{n \times k}$

Hebbian Learning (5)

- If the input to the network is the vector \vec{x}^p , the excitation vector of the network will be equal to:

$$\begin{aligned}\vec{x}^p \cdot W &= \vec{x}^p \cdot (W^1 + W^2 + \dots + W^m) = \\ &= \vec{x}^p \cdot W^p + \sum_{l \neq p}^m \vec{x}^p \cdot W^l = \\ &= \vec{y}^p \cdot (\vec{x}^p \cdot \vec{x}^p) + \sum_{l \neq p}^m \vec{y}^l \cdot (\vec{x}^l \cdot \vec{x}^p)\end{aligned}$$

- The excitation vector thus corresponds to \vec{y}^p (multiplied by a positive constant) plus a perturbation term $\sum_{l \neq p}^m \vec{y}^l \cdot (\vec{x}^l \cdot \vec{x}^p)$ that is called the CROSSTALK

Hebbian Learning (6)

- The network produces the desired vector \vec{y}^p as its output when the crosstalk is zero
 - ~ whenever the input patterns $\vec{x}^1, \vec{x}^2, \dots, \vec{x}^m$ are pairwise orthogonal
- The network can yield appropriate results even for non-zero crosstalks
 - × crosstalk should be smaller than $\vec{y}^p \cdot (\vec{x}^p \cdot \vec{x}^p)$
 - The output of the network is then equal to:

$$\text{sgn}(\vec{x}^p \cdot W) = \text{sgn} \left(\vec{y}^p \cdot (\vec{x}^p \cdot \vec{x}^p) + \sum_{l \neq p}^m \vec{y}^l \cdot (\vec{x}^l \cdot \vec{x}^p) \right)$$

Hebbian Learning (7)

- Since $\vec{x}^p \cdot \vec{x}^p$ is a positive constant, it holds that:

$$\text{sgn}(\vec{x}^p \cdot \mathbf{W}) = \text{sgn} \left(\vec{y}^p + \sum_{l \neq p}^m \vec{y}^l \cdot \frac{(\vec{x}^l \cdot \vec{x}^p)}{(\vec{x}^p \cdot \vec{x}^p)} \right)$$

- To produce the output \vec{y}^p , it must hold:

$$\vec{y}^p = \text{sgn} \left(\vec{y}^p + \sum_{l \neq p}^m \vec{y}^l \cdot \frac{(\vec{x}^l \cdot \vec{x}^p)}{(\vec{x}^p \cdot \vec{x}^p)} \right)$$

- This condition is satisfied, when the absolute value of all components of the perturbation term $\sum_{l \neq p}^m \vec{y}^l \cdot \frac{(\vec{x}^l \cdot \vec{x}^p)}{(\vec{x}^p \cdot \vec{x}^p)}$ is smaller than 1

Hebbian Learning (8)

- This means that the scalar product $\vec{x}^l \cdot \vec{x}^p$ must be smaller than the quadratic length of the vector \vec{x}^p (equal to n for n -dimensional bipolar vectors)
- If randomly selected bipolar vectors are associated with other also randomly selected bipolar vectors, the probability is high that they will be nearly pairwise orthogonal (as long as not too many of them are selected)
 - In such a case, the crosstalk will be small and Hebbian learning will lead to an efficient set of weights for the associative network

Geometric Interpretation of Hebbian Learning

- For the matrices W^i from $W = W^1 + W^2 + \dots + W^m$ it holds in an autoassociative networks:

$$W^i = (\vec{x}^i)^T \vec{x}^i$$

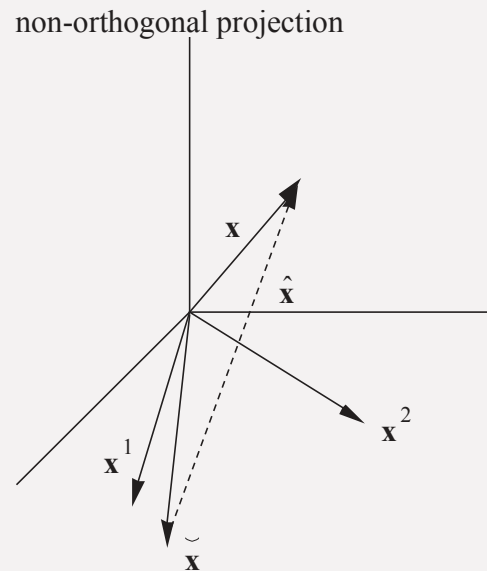
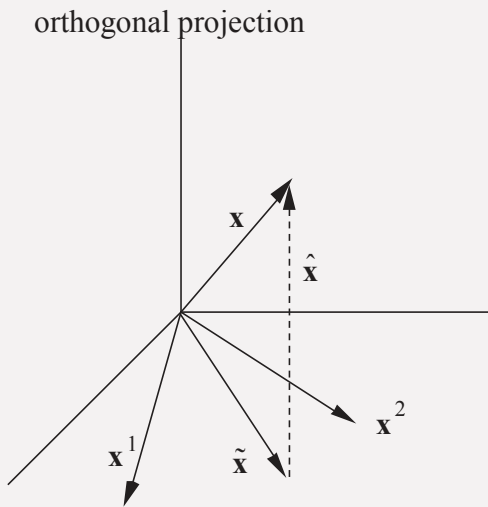
→ thus, for $W^1 = (\vec{x}^1)^T \vec{x}^1$, the input vector \vec{z} will be projected into the linear subspace L_1 spanned by the vector \vec{x}^1 , since

$$\vec{z} \cdot W^1 = \vec{z}(\vec{x}^1)^T \vec{x}^1 = (\vec{z}(\vec{x}^1)^T) \vec{x}^1 = c_1 \vec{x}^1$$

~ in general, a non-orthogonal projection of the vector \vec{z} into L_1
(c_1 represents a constant)

→ similarly for other weight matrices W^2, \dots, W^m

Geometric Interpretation of Hebbian Learning (2)



- The matrix $W = \sum_{i=0}^m W^i$ projects a vector \vec{z} into the linear subspace spanned by the vectors $\vec{x}^1, \vec{x}^2, \dots, \vec{x}^m$, because:

$$\begin{aligned}\vec{z} \cdot W &= \vec{z} \cdot W^1 + \vec{z} \cdot W^2 + \dots + \vec{z} \cdot W^m \\ &= c_1 \vec{x}^1 + c_2 \vec{x}^2 + \dots + c_m \vec{x}^m\end{aligned}$$

(in general, a non-orthogonal projection)

Associative Networks: Behavior Analysis

- Identification of attractors (fixed points of the system)
- The size of the basins of attraction
 - **Hamming distance**
 - ~ the number of different components in 2 bipolar vectors
 - Example: the Hamming distance of the vectors $(1 \quad \underline{-1} \quad \underline{1} \quad 1)$ and $(1 \quad \underline{1} \quad \underline{-1} \quad 1)$ is equal to 2
 - With the growing number of stored patterns, the basins of attraction become smaller \rightarrow spurious stable states
 - big crosstalk
 - for the patterns inverse to the stored ones:
$$\text{sgn}(-\vec{x} \cdot W) = -\text{sgn}(\vec{x} \cdot W) = -\vec{x}$$

Associative Networks: Behavior Analysis (2)

- Recurrent networks (use feedback)

- Improved convergence when compared to associative memories without feedback
- Wider basins of attraction
 - × not too many patterns can be stored
 - PROBLEM: *the capacity of the weight matrix*
- The sizes of the basins of attraction can be compared using an index

$$I = \sum_{h=0}^{n/2} h p_h ,$$

where p_h denotes the percentage of vectors with the Hamming distance h from a stored pattern they converged to

The Capacity Problem

- The basins of attraction of stored patterns deteriorate with every new pattern to be stored in the memory
- If the crosstalk term becomes too large, previously stored patterns can be even „forgotten“
- ✗ the probability that this could happen should be kept low

The Capacity Problem (2)

- Assess the number of patterns m , that can be stored safely in an autoassociative memory with a weight matrix $W_{n \times n}$
- Maximum capacity of the network: $m \sim 0.18 n$
 - The number of stored patterns should be smaller than $0.18 n$ (n is the dimension of the patterns)
 - If the patterns are correlated, even $m < 0.18 n$ can produce problems

Derivation of the Network Capacity: the Idea (1)

- Let us consider the weight matrices set as $W^i = \frac{1}{n} (\vec{x}^i)^T \vec{x}^i$
- Crosstalk for n -dimensional bipolar vectors and m patterns is in the case of an autoassociative network:

$$\frac{1}{n} \sum_{l \neq p}^m \vec{x}^l \left(\vec{x}^l \cdot \vec{x}^p \right)$$

- If the magnitude of the above crosstalk term is larger than 1 and the term has a sign opposite to the stored pattern, the considered pattern component can be flipped

Derivation of the Network Capacity: the Idea (2)

- Assume that the stored vectors are chosen randomly:
 - In this case the crosstalk term for bit i of the input vector is given by

$$\frac{1}{n} \sum_{l \neq p}^m x_i^l (\vec{x}^l \cdot \vec{x}^p) \quad (*)$$

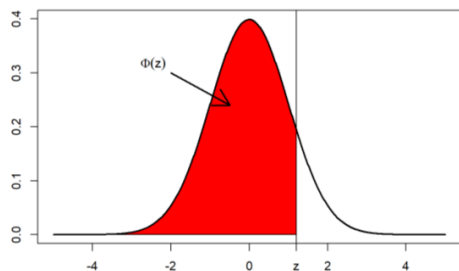
- Since the components of each pattern have been selected randomly, we can think of $m \cdot n$ random bit selections
- The expected value of this sum (*) is 0

Derivation of the Network Capacity: the Idea (3)

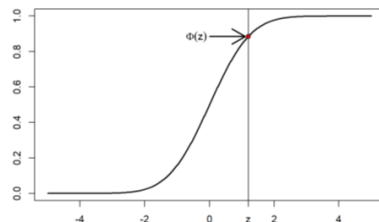
- The sum (*) has a binomial distribution and for large $m \cdot n$, we can approximate it by a normal distribution with the standard deviation $\sigma = \sqrt{m/n}$
- Probability of error P , that the sum (*) becomes larger than 1 (or smaller than -1), is given by

$$P = \frac{1}{\sqrt{2\pi}\sigma} \int_1^{\infty} e^{-x^2/(2\sigma^2)} dx$$

$\Phi(z)$ on probability density function



$\Phi(z)$ on cumulative distribution function



Derivation of the Network Capacity: the Idea (4)

That is: $P\{|(*)| > 1\} = 2 \left[1 - \Phi\left(\frac{1}{\sqrt{m/n}}\right) \right]$,

where $\Phi(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2/2} dt$

→ for the upper bound for one bit failure set to **0.01** we obtain:

$$0.01 = 2 \left[1 - \Phi\left(\frac{1}{\sqrt{m/n}}\right) \right]$$

→ therefore: $m \sim 0.18 n$

Neural Networks:

Contents:

- Multi-layered Neural Networks
- **Associative Memories**
- Associative Memories: BAM – The Bidirectional Associative Memory

Contents:

- Multi-layered Neural Networks
- **Associative Memories**
 - The Structure of Associative Memories
 - Eigenvector Automata
 - Hebbian Learning
 - The Capacity Problem
 - **Pseudoinverse Matrix**
- Associative Memories: BAM – The Bidirectional Associative Memory
 - Training and Recall
 - Energy Function
 - Convergence to Stable States

Associative Memories: the Pseudoinverse Matrix (1)

- Hebbian learning produces good results when the stored patterns are nearly orthogonal
 - ~ when m bipolar vectors are selected randomly from an n -dimensional space, n is „large enough“ and m is „much smaller“ than n
- × in real applications, the patterns are almost always correlated, and the crosstalk in the expression

$$\vec{x}^p \cdot W = \vec{y}^p \cdot (\vec{x}^p \cdot \vec{x}^p) + \sum_{l \neq p}^m \vec{y}^l \cdot (\vec{x}^l \cdot \vec{x}^p)$$

affects the recall process because the scalar products $\vec{x}^l \cdot \vec{x}^p$ are not small enough for $l \neq p$.

Associative Memories: the Pseudoinverse Matrix (2)

→ mutual correlation of the stored patterns causes reduction in the capacity of the associative network

~ the number of patterns, that can be stored and recalled

the stored patterns do not occupy the input space homogeneously, but concentrate around a small region

→ look for alternative learning methods capable of minimizing the crosstalk between stored patterns

→ use the pseudoinverse instead of the correlation matrix

Associative Memories: the Pseudoinverse Matrix (3)

Definition:

The *pseudoinverse* of a real $m \times n$ matrix is the real matrix \tilde{X} with the following properties:

1. $X\tilde{X}X = X$,
2. $\tilde{X}X\tilde{X} = \tilde{X}$,
3. $\tilde{X}X$ and $X\tilde{X}$ are symmetrical.

The pseudoinverse always exists and is unique.

Pseudoinverse Matrix: Properties

- Let $\vec{x}^1, \vec{x}^2, \dots, \vec{x}^m$ be n -dimensional vectors to be associated with m k -dimensional vectors $\vec{y}^1, \dots, \vec{y}^m$

→ matrix notation:

X Matrix $m \times n$

the rows are the vectors $\vec{x}^1, \vec{x}^2, \dots, \vec{x}^m$

Y Matrix $m \times k$

the rows are the vectors $\vec{y}^1, \dots, \vec{y}^m$

→ Look for a weight matrix W ; $XW = Y$

Pseudoinverse Matrix: Properties (2)

- Since in general $m \neq n$ and the vectors $\vec{x}^1, \vec{x}^2, \dots, \vec{x}^m$ are not necessarily linearly independent, the matrix X does not have to be invertible

→ look for a matrix W , which minimizes $\|XW - Y\|^2$

(~ the sum of the squares of all its elements)

minimization by means of $W = \tilde{X}Y$

\tilde{X} pseudoinverse of the matrix X

(~ the best approximation to an inverse of X

if X^{-1} exists, then $X^{-1} = \tilde{X}$)

Pseudoinverse Matrix: Properties (3)

Proposition:

Let X be an $m \times n$ real matrix and Y be an $m \times k$ real matrix.

The $n \times k$ matrix $W = \tilde{X}Y$ minimizes the quadratic norm $\|XW - Y\|^2$.

(At the same time, \tilde{X} minimizes $\|X\tilde{X} - I\|^2$.)

Proof:

Define $E = \|XW - Y\|^2$

→ E can be computed as $E = \text{tr}(S)$, where $S = (XW - Y)^T(XW - Y)$

($E \sim$ sum of all diagonal elements of S)

trace of a matrix



Pseudoinverse Matrix: Properties (4)

Proof (continue):

→ S can be rewritten as

$$S = (\tilde{X}Y - W)^T X^T X (\tilde{X}Y - W) + Y^T (I - X\tilde{X})Y$$

(because:

$$S = (\tilde{X}Y - W)^T (X^T X \tilde{X}Y - X^T X W) + Y^T (I - X\tilde{X})Y$$

Since the matrix $X\tilde{X}$ is symmetrical (def.), therefore:

$$S = (\tilde{X}Y - W)^T \left(\left(\underbrace{X \tilde{X} X}_{=X \text{ (def.)}} \right)^T Y - X^T X W \right) + Y^T (I - X\tilde{X})Y$$

Pseudoinverse Matrix: Properties (5)

Proof (continue):

$$\begin{aligned} \text{(it follows: } S &= (\tilde{X}Y - W)^T (X^T Y - X^T X W) + Y^T (I - X\tilde{X})Y = \\ &= (\tilde{X}Y - W)^T X^T (Y - XW) + Y^T (I - X\tilde{X})Y = \\ &= (X\tilde{X}Y - XW)^T (Y - XW) + Y^T (I - X\tilde{X})Y = \\ &= (-XW)^T (Y - XW) + Y^T X\tilde{X}(Y - XW) + Y^T (I - X\tilde{X})Y = \\ &= (-XW)^T (Y - XW) + Y^T (-XW) + Y^T Y = \\ &= (Y - XW)^T (Y - XW) \end{aligned}$$

→ E can be rewritten as

$$E = \text{tr} \left((\tilde{X}Y - W)^T X^T X (\tilde{X}Y - W) \right) + \underbrace{\text{tr} (Y^T (I - X\tilde{X})Y)}_{= \text{const.}}$$

→ $\min E$ for $W = \tilde{X}Y$, QED.

Pseudoinverse Matrix: Application

Motivation and application:

- The inverse matrix does not always exist
- An alternative represents the pseudoinverse matrix
 - Minimization of the mean squared error (e.g., with multi-layered neural networks)
 - The training set: $\{(\vec{x}_p, \vec{d}_p); p = 1, \dots, P\}$
 - \vec{x}_p Input pattern (n – dimensional)
 - \vec{d}_p Desired output (m – dimensional)
 - \vec{y}_p Actual output (m – dimensional)

Pseudoinverse Matrix: Application (2)

- Derivation: $E = \sum_{p=1}^P E_p = \sum_{p=1}^P \sum_{j=1}^m (d_{j,p} - y_{j,p})^2$
with \vec{y}_p to be determined according to: $y_{j,p} = \sum_{i=1}^n w_{ij} x_{i,p}$
 - Minimization of E with respect to the weights
→ the partial derivatives of E with respect to the weights:

$$\begin{aligned} \frac{\partial E}{\partial w_{ij}} &= \frac{\partial}{\partial w_{ij}} \left(\sum_{p=1}^P \sum_{j=1}^m \left(d_{j,p} - \sum_{i=1}^n w_{ij} x_{i,p} \right)^2 \right) = \\ &= -2 \sum_{p=1}^P \left(\sum_{i=1}^n d_{j,p} - w_{ij} x_{i,p} \right) x_{i,p} = 0 \end{aligned}$$

Pseudoinverse Matrix: Application (3)

Matrix notation: $WXX^T = DX^T$

- W matrix $m \times n$ with the elements w_{ij}
- X matrix $n \times P$ with the elements $x_{i,p}$
- D matrix $m \times P$ with the elements $d_{j,p}$

× in general, there does not have to exist an inverse matrix to the matrix XX^T

→ it might be not possible to solve the equation directly (and find the weight matrix W) if XX^T does not have m linearly independent rows

Pseudoinverse Matrix: Application (4)

There can be several solutions

→ an additional constraint on the weight values:

$$E = \lambda \sum_{i=1}^n \sum_{j=1}^m w_{ij}^2 \quad ; \quad \lambda > 0, \lambda = \text{const.}$$

Minimization by means of partial derivatives

$$W (X X^T + \lambda I) = D X^T$$

($\lambda > 0$ there exists an inverse matrix to $X X^T + \lambda I$)

$$W (X X^T + \lambda I) (X X^T + \lambda I)^{-1} = D X^T (X X^T + \lambda I)^{-1}$$

Pseudoinverse Matrix: Application (5)

For λ approaching in the limit zero ($\lambda \rightarrow 0$):

$$W = \lim_{\lambda \rightarrow 0} \left[DX^T (XX^T + \lambda I)^{-1} \right] = D\tilde{X}$$

- \tilde{X} ... Pseudoinverse matrix to the matrix X
- If there are more solutions, \tilde{X} will yield the lowest values

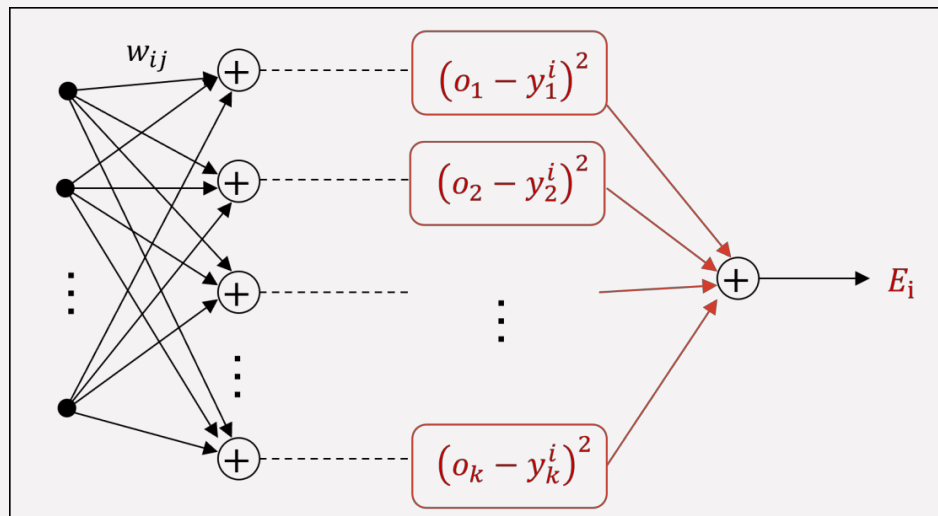
$$\sum_{i=1}^n \sum_{j=1}^m w_{ij}^2$$

- If the matrix inverse to X exists, it will hold $\tilde{X} = X^{-1}$



Computation of the Pseudoinverse

- Compute an approximation of the pseudoinverse using a backpropagation network
- The network used to find the weights for associative memories



o – network output

y – desired association

Computation of the Pseudoinverse (2)

- **The goal of training:** Find such a weight matrix W with the elements w_{ij} that produces the best mapping from the vectors $\vec{x}^1, \dots, \vec{x}^m$ to the vectors $\vec{y}^1, \dots, \vec{y}^m$
- For the i -th input vector, the actual network output will be compared with the vector \vec{y}^i , and E_i will be computed
- The total quadratic error $E = \sum_{i=1}^m E_i$ then corresponds to:
 $\|XW - Y\|^2$
- The backpropagation algorithm finds the matrix W , that is expected to minimize $\|XW - Y\|^2$

Neural Networks:

BAM – Bidirectional Associative Memory

doc. RNDr. Iveta Mrázová, CSc.

DEPARTMENT OF THEORETICAL COMPUTER SCIENCE AND MATHEMATICAL LOGIC
FACULTY OF MATHEMATICS AND PHYSICS, CHARLES UNIVERSITY IN PRAGUE

Neural Networks:

Contents:

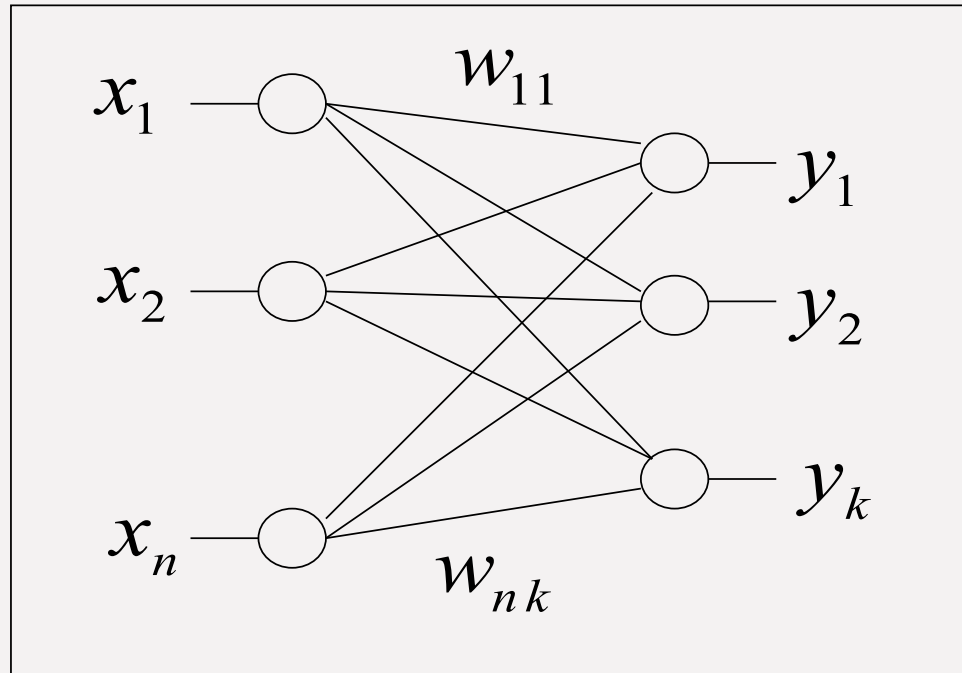
- Multi-layered Neural Networks
- **Associative Memories**
- Associative Memories: BAM – The Bidirectional Associative Memory

Contents:

- Multi-layered Neural Networks
- **Associative Memories**
 - The Structure of Associative Memories
 - Eigenvector Automata
 - Hebbian Learning
 - The Capacity Problem
 - Pseudoinverse Matrix
- Associative Memories: BAM – The Bidirectional Associative Memory
 - Training and Recall
 - Energy Function
 - Convergence to Stable States

Bidirectional Associative Memory

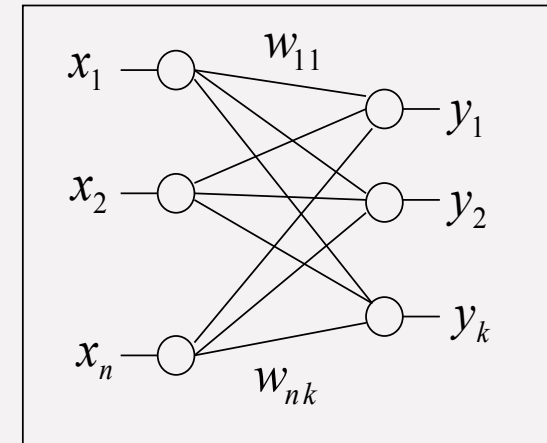
BAM – a synchronous associative model with bidirectional edges



Bidirectional Associative Memory (2)

- A recurrent associative memory

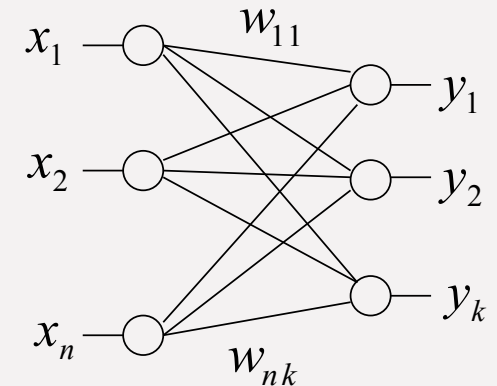
- Consists of 2 layers of units, which send information recursively between them.
- The input layer sends the result of its computation to the output layer by the weights.
- The output layer then returns the result of its computation back to the input layer – by the same weights.



- Question: Will the network achieve a stable state, in which the information being sent back and forth does not change after a few iterations?

Bidirectional Associative Memory (3)

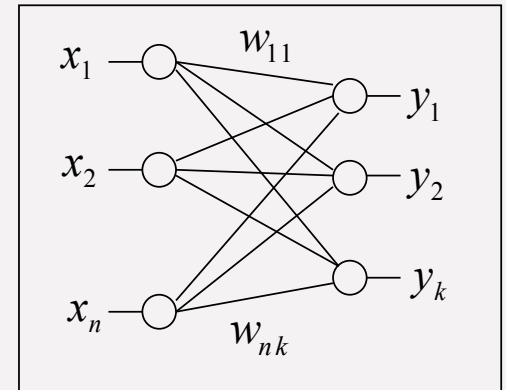
- a resonance network
- *sgn* transfer function
- information is coded using bipolar values
- the network maps an n -dimensional vector \vec{x}_0 to a k -dimensional vector \vec{y}_0
- the weight matrix of the network is the $n \times k$ matrix W .
 - after the first passage, we obtain: $\vec{y}_0 = \text{sgn}(\vec{x}_0 W)$
 - after the feedback passage, the input will be: $\vec{x}_1 = \text{sgn}(W \vec{y}_0^T)$
 - after the next forward passage, the output will be: $\vec{y}_1 = \text{sgn}(\vec{x}_1 W)$



Bidirectional Associative Memory (4)

- After m iterations, we have $m + 1$ pairs of vectors $(\vec{x}_0, \vec{y}_0), \dots, (\vec{x}_m, \vec{y}_m)$ that fulfill the condition:

$$\vec{y}_i = \text{sgn}(\vec{x}_i W) \quad \text{and} \quad \vec{x}_{i+1} = \text{sgn}(W \vec{y}_i^T)$$



- Question: Will the system find after some iterations a fixed point (\vec{x}, \vec{y}) such that it holds

$$\vec{y} = \text{sgn}(\vec{x} W) \quad \text{and} \quad \vec{x} = \text{sgn}(W \vec{y}^T) ?$$

Bidirectional Associative Memory (5)

→ if a pair of vectors (\vec{x}, \vec{y}) is given and we want to set the weights of a BAM such that this pair of vectors will represent its fixed point, Hebbian learning can be used to compute an adequate weight matrix: $W = \vec{x}^T \vec{y}$

$$\rightarrow \vec{y} = \text{sgn}(\vec{x}W) = \text{sgn}(\vec{x}\vec{x}^T\vec{y}) = \text{sgn}(\|\vec{x}\|^2\vec{y}) = \vec{y}$$

and also:

$$\vec{x}^T = \text{sgn}(W\vec{y}^T) = \text{sgn}(\vec{x}^T\vec{y}\vec{y}^T) = \text{sgn}(\vec{x}^T\|\vec{y}\|^2) = \vec{x}^T$$

Bidirectional Associative Memory (6)

- If we want to store several patterns $(\vec{x}_1, \vec{y}_1), \dots, (\vec{x}_m, \vec{y}_m)$, Hebbian learning will be more efficient, if the vectors $\vec{x}_1, \dots, \vec{x}_m$ and $\vec{y}_1, \dots, \vec{y}_m$ are pairwise orthogonal (negligible „crosstalk“)
 - for m pairs of vectors the matrix W will be set to:
$$W = \vec{x}_1^T \vec{y}_1 + \vec{x}_2^T \vec{y}_2 + \dots + \vec{x}_m^T \vec{y}_m$$
 - a bidirectional associative memory can be used also to build autoassociative networks, because the weight matrices produced by the Hebb rule (or when computing the pseudoinverse) are symmetric ($X = XW$ and $X^T = WX^T$)

Energy function for BAM

- Assume that a BAM is given for which the vector pair (\vec{x}, \vec{y}) is a stable state
- The initial vector presented to the network from the left is \vec{x}_0 (and after some iterations, the network shall converge to (\vec{x}, \vec{y}))
- The vector \vec{y}_0 is computed according to: $\vec{y}_0 = \text{sgn}(\vec{x}_0 W)$
- The output vector \vec{y}_0 is now used for a new iteration (from the right)

Energy function for BAM (2)

- Excitation of the neurons on the left is determined by the excitation vector $\vec{e}^T = W\vec{y}_0^T$
 - (\vec{x}_0, \vec{y}_0) corresponds to a stable state of the network,
if $\text{sgn}(\vec{e}) = \vec{x}_0$, i.e., if \vec{e} is close enough
 - the scalar product $\vec{x}_0 \vec{e}^T$ should be larger than for other vectors of the same length but further away from \vec{x}_0

Energy function for BAM (3)

- The product $E \cong - \vec{x}_0 \vec{e}^T = - \vec{x}_0 W \vec{y}_0^T$ is thus smaller, if the vector $W \vec{y}_0$ lies closer to \vec{x}_0
- a sort of an indicator of convergence to stable states of an associative memory

$E \sim \underline{\text{energy function}}$

- Local minima of the energy function correspond to stable states

Energy function for BAM (4)

Definition:

Let W be the weight matrix of a BAM and the output \vec{y}_i of the right layer of neurons is computed in the i -th iteration as $\vec{y}_i = \text{sgn}(\vec{x}_i W)$ and the output \vec{x}_{i+1} of the left layer of neurons is computed as $\vec{x}_{i+1} = \text{sgn}(W \vec{y}_i^T)$.

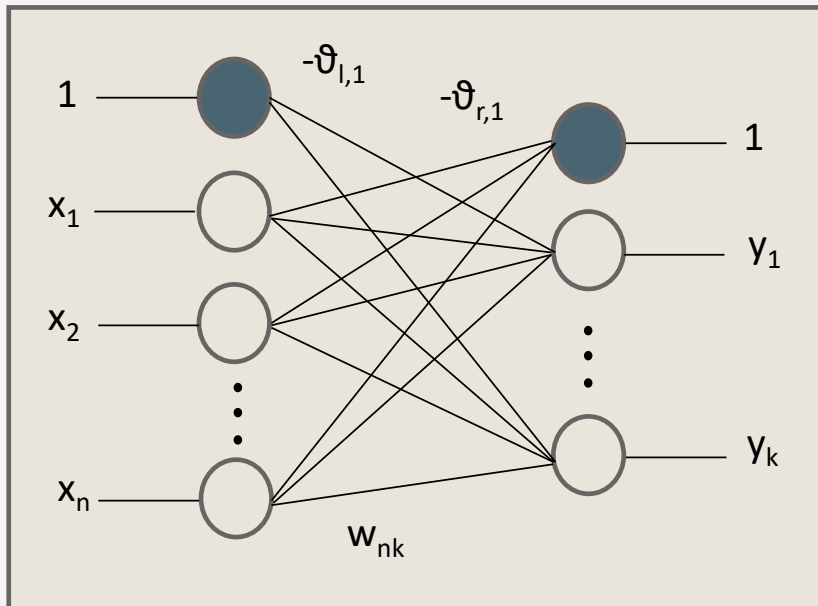
The energy function of a BAM is then given by:

$$E(\vec{x}_i, \vec{y}_i) = -\frac{1}{2} \vec{x}_i^T W \vec{y}_i$$

Generalized Energy Function

- Considers also the thresholds and a stepwise transfer function
 - Each n -dimensional vector \vec{x} will be transformed into the vector $(x_1, \dots, x_n, 1)$
 - Each k -dimensional vector \vec{y} , $y_j = \sum_i w_{ij}x_i - \vartheta_{l,j}$ will be transformed into the vector $(y_1, \dots, y_k, 1)$
 - The weight matrix W will be extended to a new matrix W' with an additional row and column

Generalized Energy Function (2)



- Negative thresholds of the neurons in the right layer of the BAM form the $(n + 1)$ -th row of W'
- Negative thresholds of the neurons in the left layer form the $(k + 1)$ -th column of W'
- The entry $(n + 1, k + 1)$ of the matrix W' is 0

Generalized Energy Function (3)

- The above transformation is equivalent to the introduction of an additional unit with output **1** into both layers
 - The weights of these additional neurons correspond to negative thresholds of neurons fed by this information

→ Energy function of the extended BAM:

$$E(\vec{x}_i, \vec{y}_i) = -\frac{1}{2}\vec{x}_i W \vec{y}_i^T + \frac{1}{2}\vec{\theta}_l \vec{y}_i^T + \frac{1}{2}\vec{x}_i \vec{\theta}_r^T$$

$\vec{\theta}_l$ the vector of thresholds of the k neurons (in the left layer)

$\vec{\theta}_r^T$ the vector of thresholds of the n neurons (in the right layer)

Asynchronous BAM-networks

- Each unit computes its excitation at random
- Changes its state to **1** or **-1** independently of the others (but according to the sign of its total excitation)
- The probability of two units firing simultaneously is zero
- Assumption: the state of a unit is not changed if the total excitation is zero

Asynchronous BAM-networks (2)

- BAM arrives at a stable state after a finite number of iterations (sequential choice of neurons)
 - a stable state \sim vector pair (\vec{x}, \vec{y}) ; $\vec{y} = \text{sgn}(\vec{x}W)$ and $\vec{x}^T = \text{sgn}(W\vec{y}^T)$
- Proposition:

A bidirectional associative memory with an arbitrary weight matrix W reaches a stable state in a finite number of iterations using either synchronous or asynchronous updates.

Asynchronous BAM-networks (3)

Proof:

- For the vectors $\vec{x} = (x_1, x_2, \dots, x_n)$, $\vec{y} = (y_1, y_2, \dots, y_k)$, and a $n \times k$ weight matrix $W = \{w_{ij}\}$, the energy function $E(\vec{x}, \vec{y})$ equals to:

$$E(\vec{x}, \vec{y}) = -\frac{1}{2} (x_1, \dots, x_n) \begin{pmatrix} w_{11} & w_{12} & \cdots & w_{1k} \\ w_{21} & w_{22} & \cdots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{n1} & w_{n2} & \cdots & w_{nk} \end{pmatrix} \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_k \end{pmatrix}$$

Asynchronous BAM-networks (4)

Proof (continue):

- Product of the i -th row of W and \vec{y}^T represents the excitation of the i -th neuron in the left layer g_i

→ then for the excitation vector of the left layer (g_1, \dots, g_n) :

$$E(\vec{x}, \vec{y}) = -\frac{1}{2} (x_1, \dots, x_n) \begin{pmatrix} g_1 \\ \vdots \\ g_n \end{pmatrix}$$

→ similarly, it holds for the right layer and its excitation vector (e_1, \dots, e_k) :

$$E(\vec{x}, \vec{y}) = -\frac{1}{2} (e_1, \dots, e_k) \begin{pmatrix} y_1 \\ \vdots \\ y_k \end{pmatrix}$$

Asynchronous BAM-networks (5)

Proof (continue):

- Energy function can be written in two equivalent forms:

$$E(\vec{x}, \vec{y}) = -\frac{1}{2} \sum_{i=1}^k e_i y_i$$

$$\text{and } E(\vec{x}, \vec{y}) = -\frac{1}{2} \sum_{i=1}^n g_i x_i$$

- In asynchronous networks at each iteration, we randomly select a neuron from the left or right layer:
 - For the selected neuron, excitation and new state are computed

Asynchronous BAM-networks (6)

Proof (continue):

- If the state remains the same, the energy of the network will not change
- The state of neuron i in the left layer will change only when the excitation g_i has a different sign than its present state x_i
- Since the other neurons do not change their states (asynchronous dynamics), the difference between the previous energy $E(\vec{x}, \vec{y})$ and the new energy $E(\vec{x}', \vec{y})$ will be:

$$E(\vec{x}, \vec{y}) - E(\vec{x}', \vec{y}) = -\frac{1}{2} g_i (x_i - x'_i)$$

Asynchronous BAM-networks (7)

Proof (continue):

- Since $x_i - x'_i$ has a different sign than g_i , it follows that:

$$E(\vec{x}, \vec{y}) - E(\vec{x}', \vec{y}) > 0$$

(if $x_i - x'_i$ would have the same sign like g_i , no change of the neuron state had been occurred)

→ The new state (\vec{x}', \vec{y}) has thus a lower energy than the original state (\vec{x}, \vec{y})

- Analogically for the neurons from the right layer:

$$E(\vec{x}, \vec{y}) - E(\vec{x}, \vec{y}') > 0$$

(Whenever the state of the neuron has been flipped.)

Asynchronous BAM-networks (8)

Proof (continue):

- Any update of the network state reduces the total energy
 - Since there are only a finite number of possible combinations of bipolar states, the process must stop at some state (\vec{a}, \vec{b}) whose energy cannot be further reduced
- the network has fallen into a local minimum of the energy function and the state (\vec{a}, \vec{b}) is an attractor of the system

QED

Asynchronous BAM-networks (9)

Remark:

- The proposition holds also for synchronous networks
- any given real matrix W possesses bidirectional stable bipolar states

Neural Networks:

Hopfield Networks

doc. RNDr. Iveta Mrázová, CSc.

DEPARTMENT OF THEORETICAL COMPUTER SCIENCE AND MATHEMATICAL LOGIC
FACULTY OF MATHEMATICS AND PHYSICS, CHARLES UNIVERSITY IN PRAGUE

Neural Networks:

Contents:

- Associative Memories
- Associative Memories: BAM – The Bidirectional Associative Memory
- Associative Memories: Hopfield Networks

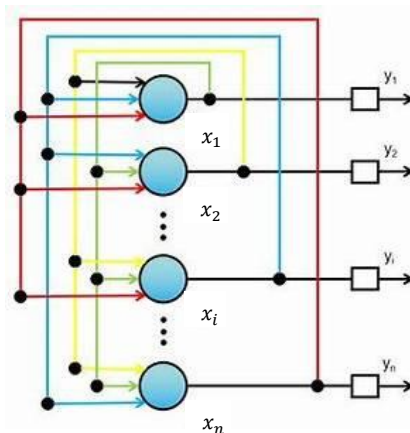
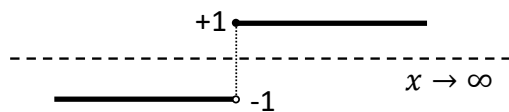
Contents:

- Associative Memories
 - The Structure of Associative Memories
 - Eigenvector Automata
 - Hebbian Learning
 - The Capacity Problem
 - Pseudoinverse Matrix
- Associative Memories: BAM – The Bidirectional Associative Memory
 - Training and Recall
 - Energy Function
 - Convergence to Stable States
- Associative Memories: Hopfield Networks
 - The Hopfield Model: Training and Recall
 - Energy Function and Convergence Analysis
 - Equivalence of Hebbian and Perceptron Learning for the Hopfield Model



„for foundational discoveries and inventions that enable machine learning with artificial neural networks“

Hard-limiting transfer function: f_h



Hopfield Networks

- n neurons with the hard-limiting transfer function
- Bipolar inputs and outputs $\{+1, -1\}$
- Synaptic weights w_{ij} (between all the neurons)
- m training patterns (classes)
- Supervised training
- Recall
- Applications:
 - Associative memory
 - Optimization tasks

The Hopfield Model (bipolar)

Step 1: *Training* – set the synaptic weights according to:

$$w_{ij} = \begin{cases} \sum_{s=1}^m x_i^s x_j^s & \text{for } i \neq j \\ 0 & \text{for } i = j \end{cases}$$

w_{ij} the synaptic weight between neuron i and j

$x_i^s \in \{-1, +1\}$ i -th component of the s -th pattern,
 $1 \leq i, j \leq n$

The Hopfield Model (bipolar) (2)

Step 2: *Initialization* – present a new input pattern:

$$y_i(0) = x_i \quad 1 \leq i \leq n$$

$y_i(t)$ the output of the neuron i at time t

$x_i \in \{-1, +1\}$ the i -th element of the presented pattern

Step 3: *Iteration*

$$y_j(t + 1) = f_h \left[\sum_{i=1}^n w_{ij} y_i(t) \right] \quad 1 \leq j \leq n$$

f_h the hard-limiting transfer function

The Hopfield Model (bipolar) (3)

The iterative process is repeated during recall until neuron outputs stabilize.

The neuron outputs then represent that stored pattern, which best corresponds to the presented (new) pattern.

Step 4: Go to Step 2.

The Hopfield Model (bipolar) (4)

Convergence (Hopfield):

- Symmetric weights: $w_{ij} = w_{ji}$
- Asynchronous output updates in single neurons

Drawbacks:

- Capacity ($m < 0.15 n$)
- Stability (\rightarrow orthogonalization)

The Hopfield Model: Example

Training:

- Patterns: $[-1, -1, 1, 1]$
 $[1, -1, 1, -1]$

- Weight setting:

$$w_{ij} = \sum_{m=1}^M x_i^{(m)} x_j^{(m)} \quad i \neq j$$

$$w_{ij} = 0 \quad i = j$$

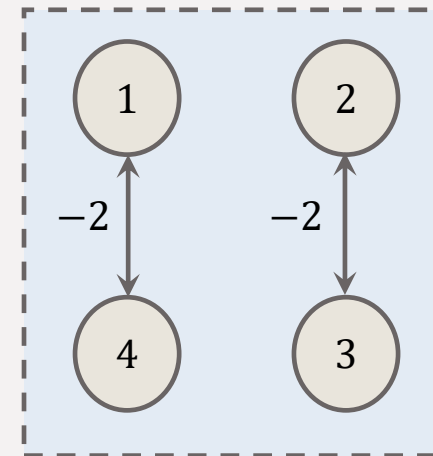
The Hopfield Model: Example (2)

- Weight matrix:

$$W = \begin{bmatrix} 0 & 0 & 0 & -2 \\ 0 & 0 & -2 & 0 \\ 0 & -2 & 0 & 0 \\ -2 & 0 & 0 & 0 \end{bmatrix}$$

Recall:

- Pattern: $[-1, -1, 1, -1]$
 \swarrow \searrow
 $[1, -1, 1, -1]$ $[-1, -1, 1, 1]$



The Hopfield Model: Recall

- When initialized with \vec{x}_1 , the vector of potentials is:

$$\begin{aligned}\vec{\xi} &= \vec{x}_1 \cdot W = \vec{x}_1 \cdot (\vec{x}_1^T \vec{x}_1 + \cdots + \vec{x}_m^T \vec{x}_m - mI) = \\ &= \underbrace{\vec{x}_1 \vec{x}_1^T}_{=n} \vec{x}_1 + \underbrace{\vec{x}_1 \vec{x}_2^T}_{=\alpha_{12}} \vec{x}_2 + \cdots + \underbrace{\vec{x}_1 \vec{x}_m^T}_{=\alpha_{1m}} \vec{x}_m - m\vec{x}_1 I = \\ &= (n - m)\vec{x}_1 + \underbrace{\sum_{j=2}^m \alpha_{1j} \vec{x}_j}_{PERTURBATION}\end{aligned}$$

The Hopfield Model: Recall (2)

$\alpha_{12}, \dots, \alpha_{1m}$ the scalar (dot) products of \vec{x}_1 with each of the (pattern) vectors $\vec{x}_2, \dots, \vec{x}_m$

→ State \vec{x}_1 is stable, if $m < n$ and the perturbation term $\sum_{j=2}^m \alpha_{1j} \vec{x}_j$ is small ($\Rightarrow \text{sgn}(\vec{\xi}) = \text{sgn}(\vec{x}_1)$)

→ **A small number of orthogonal patterns**

The Hopfield Model: Recall (3)

- States of neurons not selected for update remain the same
- Random selection for update
- Neurons are fully interconnected
- Symmetric weights: $w_{ij} = w_{ji}$
- $w_{ii} = 0$

→ Necessary conditions for convergence to a stable solution:

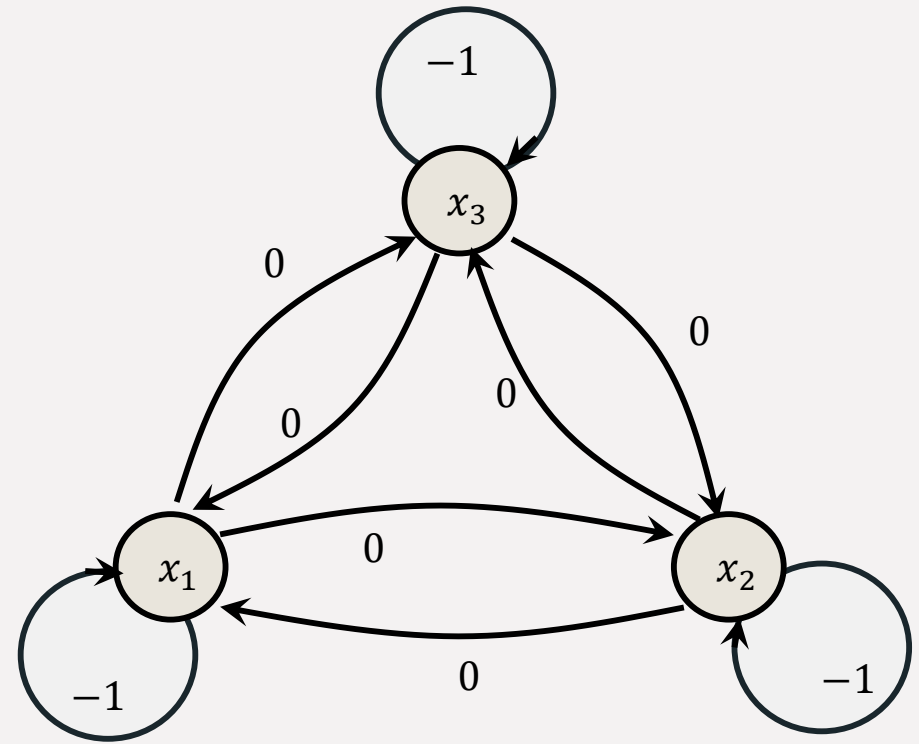
- symmetric weight matrix with zero diagonal
- and asynchronous dynamics

The Hopfield Model: Examples

- A weight matrix with nonzero diagonal does not have to yield stable states, e.g.:

$$W = \begin{pmatrix} -1 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

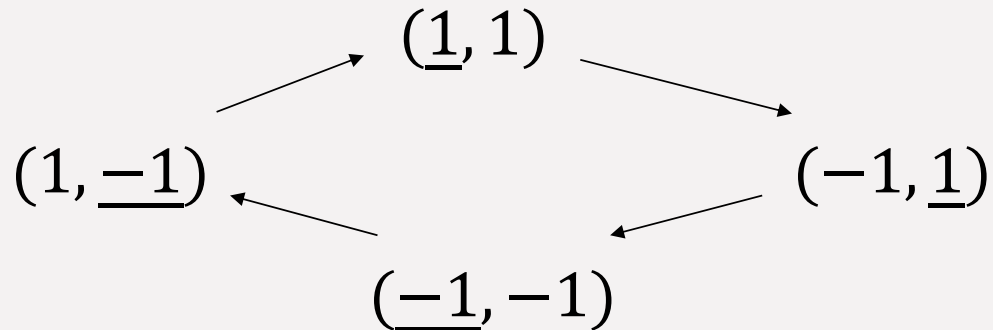
- Synchronous dynamics:
 $(-1, -1, -1) \leftrightarrow (1, 1, 1)$
- Asynchronous dynamics:
 - Random selection of one out of eight possible patterns



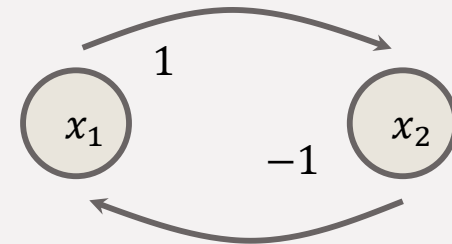
The Hopfield Model: Examples (2)

- Asymmetric weight matrix: $W = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix}$

- Asynchronous dynamics:



cyclic changes of states



Energy Function

- Energy function $E(\vec{x})$ of a Hopfield network with n neurons and the weight matrix W shows the energy of the network in state \vec{x} :

$$E(\vec{x}) = -\frac{1}{2} \vec{x} W \vec{x}^T = -\frac{1}{2} \sum_{j=1}^n \sum_{i=1}^n w_{ij} x_i x_j$$

(Similarly, also for networks with thresholds:

$$\begin{aligned} E(\vec{x}) &= -\frac{1}{2} \vec{x} W \vec{x}^T + \vec{\theta} \vec{x}^T = \\ &= -\frac{1}{2} \sum_{j=1}^n \sum_{i=1}^n w_{ij} x_i x_j + \sum_{i=1}^n \vartheta_i x_i .) \end{aligned}$$

Energy Function (2)

Proposition:

A Hopfield network with n neurons and asynchronous dynamics, which starts from any given network state, eventually reaches a stable state at a local minimum of the energy function.

Proof (idea):

- Initial state:
 - Presented pattern: $\vec{x} = (x_1, \dots, \underline{x_k}, \dots, x_n)$ and the energy is evaluated

Energy Function (3)

Proof (idea - continue):

according to: $E(\vec{x}) = -\frac{1}{2} \sum_{j=1}^n \sum_{i=1}^n w_{ij} x_i x_j$

- Neuron k is selected for adjustment
 - if k does not change its state \rightarrow the value of $E(\vec{x})$ does not change, too
 - if k changes its state \rightarrow the vector $\vec{x}' = (x_1, \dots, \underline{x'_k}, \dots, x_n)$ yields the energy:

$$\begin{aligned} E(\vec{x}') = & -\frac{1}{2} \sum_{\substack{j=1 \\ j \neq k}}^n \sum_{\substack{i=1 \\ i \neq k}}^n w_{ij} x_i x_j - \frac{1}{2} \sum_{\substack{i=1 \\ i \neq k}}^n w_{ik} x_i x'_k - \\ & - \frac{1}{2} \sum_{\substack{j=1 \\ j \neq k}}^n w_{ik} x'_k x_j - \frac{1}{2} w_{kk} x'_k x'_k \end{aligned}$$

Energy Function (4)

Proof (idea - continue):

$$E(\vec{x}') = -\frac{1}{2} \sum_{\substack{j=1 \\ j \neq k}}^n \sum_{\substack{i=1 \\ i \neq k}}^n w_{ij} x_i x_j - \sum_{i=1}^n w_{ik} x_i x'_k$$

because of symmetric weights


$$-\sum_{i=1}^n w_{ik} x_i x'_k = \begin{cases} -\frac{1}{2} \sum_{j=1}^n w_{kj} x'_k x_j \\ -\frac{1}{2} \sum_{i=1}^n w_{ik} x_i x'_k \end{cases}$$

and a zero feed-back $w_{kk} = 0$

Energy Function (5)

Proof (idea - continue):

- The difference between the old and new energies:

$$\begin{aligned} E(\vec{x}) - E(\vec{x}') &= -\sum_{i=1}^n w_{ik} x_i x_k - (-\sum_{i=1}^n w_{ik} x_i x'_k) = \\ &= -\underbrace{(x_k - x'_k)}_{\text{POTENTIAL}} \underbrace{\sum_{i=1}^n w_{ik} x_i}_{\text{POTENTIAL}} > 0 \end{aligned}$$


Both $x_k - x'_k$ and the new potential value have a different sign than the potential (otherwise no change of state would occur)

Energy Function (6)

Proof (idea - continue):

- *every time the state of a neuron is altered, the total energy of the network is reduced*
- due to the finite number of possible (bipolar) states
 - *the network must eventually reach a stable state for which the energy cannot be further reduced*

QED

Equivalence of the Hebbian and Perceptron Learning for the Hopfield Model

- Sometimes Hebbian learning cannot find a weight matrix for which m given vectors are stable states (although such a matrix exists)
 - if the vectors to be stored lie near each other, the perturbation term can grow too large
 - **worse results of Hebbian learning**
- Alternative: *Perceptron learning for Hopfield networks*

Equivalence of the Hebbian and Perceptron Learning for the Hopfield Model (2)

Perceptron learning in Hopfield networks

- Hopfield networks are composed of neurons with a non-zero threshold and the hard-limiting transfer function
 - Neurons adopt state **1** for potentials greater than **0**
 - Neurons adopt state **-1** for potentials smaller than or equal to **0**

Equivalence of the Hebbian and Perceptron Learning for the Hopfield Model (3)

- Let us consider a Hopfield network:
 - n the number of neurons
 - $W = \{w_{ij}\}$ the $n \times n$ weight matrix
 - ϑ_i the threshold of the neuron i
- If a vector $\vec{x} = (x_1, \dots, x_n)$ is given to be „imprinted“ in the network, this vector will correspond to a stable state only if the network does not change its state after presenting this vector at its input

Equivalence of the Hebbian and Perceptron Learning for the Hopfield Model (4)

→ neuron potentials should have the same sign like their previous states

- Minus sign will be assigned to zero values
- The following inequalities should therefore hold:

$$\text{Neuron 1: } \text{sgn}(x_1) (0 + x_2 w_{12} + \dots + x_n w_{1n} - \vartheta_1) > 0$$

$$\text{Neuron 2: } \text{sgn}(x_2) (x_1 w_{21} + 0 + \dots + x_n w_{2n} - \vartheta_2) > 0$$

...

...

...

...

$$\text{Neuron n: } \text{sgn}(x_n) (x_1 w_{n1} + x_2 w_{n2} + \dots + 0 - \vartheta_1) > 0$$

Equivalence of the Hebbian and Perceptron Learning for the Hopfield Model (5)

- $w_{ij} = w_{ji} \rightarrow n \cdot (n - 1)/2$ non-zero elements of the weight matrix and n thresholds

→ let \vec{v} denote a vector of dimension $n + n \cdot (n - 1)/2$

(the components of \vec{v} correspond to the elements w_{ij} above the diagonal of the matrix W ; $i < j$, and the n thresholds with minus sign)

$$\vec{v} = (\underbrace{w_{12}, w_{13}, \dots, w_{1n}}_{n-1 \text{ components}}, \underbrace{w_{23}, w_{24}, \dots, w_{2n}}_{n-2 \text{ components}}, \dots, \underbrace{w_{n-1,n}}_{1 \text{ component}}, \underbrace{-\vartheta_1, \dots, -\vartheta_n}_{n \text{ components}})$$

Equivalence of the Hebbian and Perceptron Learning for the Hopfield Model (6)

→ transformation of the vector \vec{x} into n „auxiliary“ vectors $\vec{z}_1, \vec{z}_2, \dots, \vec{z}_n$ of the dimension $n + \frac{n(n-1)}{2}$:

$$\begin{aligned}
 \vec{z}_1 &= (\underbrace{x_2, x_3, \dots, x_n}_{n-1 \text{ components}}, \underbrace{0, 0, \dots, 0, 1, 0, \dots, 0}_n) \\
 \vec{z}_2 &= (\underbrace{x_1, 0, \dots, 0}_{n-1 \text{ components}}, \underbrace{x_3, \dots, x_n}_{n-2 \text{ components}}, \underbrace{0, 0, \dots, 0, 1, \dots, 0}_n) \\
 &\quad \vdots \qquad \qquad \qquad \vdots \qquad \qquad \qquad \vdots \\
 \vec{z}_n &= (\underbrace{0, 0, \dots, x_1}_{n-1 \text{ components}}, \underbrace{0, 0, \dots, x_2}_{n-2 \text{ components}}, \underbrace{0, 0, \dots, 1}_n)
 \end{aligned}$$

Equivalence of the Hebbian and Perceptron Learning for the Hopfield Model (7)

- the components of the vectors $\vec{z}_1, \vec{z}_2, \dots, \vec{z}_n$ allow to write the above inequalities as:

$$\begin{array}{llll} \text{Neuron 1:} & \text{sgn}(x_1) & \vec{z}_1 \cdot \vec{v} & > 0 \\ \text{Neuron 2:} & \text{sgn}(x_2) & \vec{z}_2 \cdot \vec{v} & > 0 \\ & \vdots & \vdots & \vdots \\ \text{Neuron n:} & \text{sgn}(x_n) & \vec{z}_n \cdot \vec{v} & > 0 \end{array}$$

Equivalence of the Hebbian and Perceptron Learning for the Hopfield Model (8)

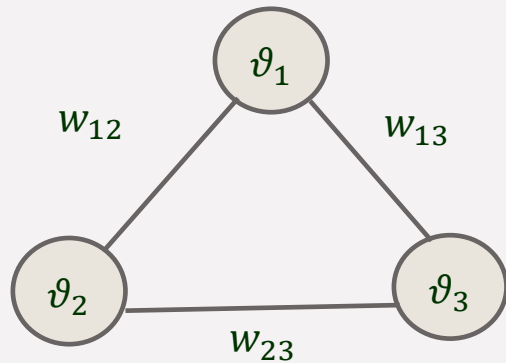
- to linearly separate the vectors $\vec{z}_1, \vec{z}_2, \dots, \vec{z}_n$ (based on $\text{sgn}(x_i) - \vec{z}_i$ should be positive if $\text{sgn } x_i = 1$), we can use perceptron learning
- Compute the weight vector \vec{v} needed for the linear separation of the vectors $\vec{z}_1, \vec{z}_2, \dots, \vec{z}_n$ and set the weight matrix W
- If the Hopfield network has to „remember“ m vectors $\vec{x}_1, \vec{x}_2, \dots, \vec{x}_m$, we have to use the above transformation for everyone of them

Equivalence of the Hebbian and Perceptron Learning for the Hopfield Model (9)

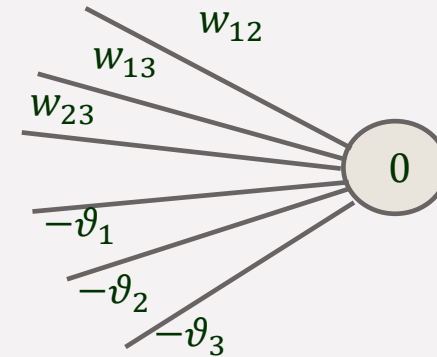
- $m \cdot n$ „auxiliary“ vectors, which must be linearly separated
- If the (auxiliary) vectors are actually linearly separable, perceptron learning will find the solution to the problem, encoded as the vector \vec{v}

Equivalence of the Hebbian and Perceptron Learning for the Hopfield Model (10)

Example:



Training of a Hopfield network with n neurons



Perceptron learning with the input space dimension
 $n + n \cdot (n - 1)/2 \quad (= n \cdot (n + 1)/2)$

Remark: „local application of the delta-rule“ instead of perceptron learning algorithm