Multilayer feedforward networks and backpropagation

Johan Suykens

KU Leuven, ESAT-STADIUS
Kasteelpark Arenberg 10
B-3001 Leuven (Heverlee), Belgium
Email: johan.suykens@esat.kuleuven.be
http://www.esat.kuleuven.be/stadius

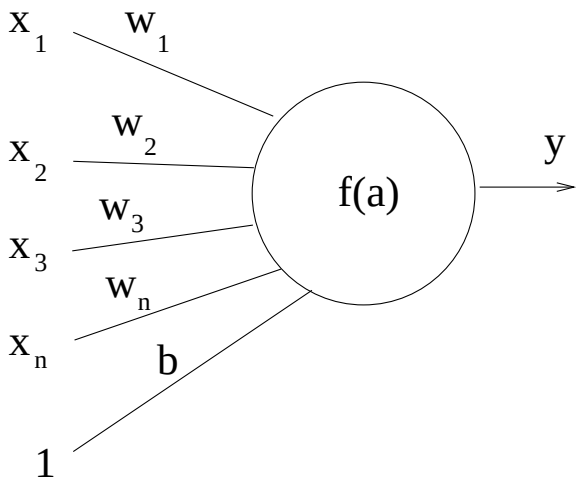
Lecture 2

Overview

- neuron, activation functions
- multilayer perceptron
- radial basis function network
- universal approximation
- curse of dimensionality
- backpropagation algorithm
- overfitting and early stopping
- limitations of perceptron, Cover's theorem
- example: alphabet recognition

McCulloch-Pitts model of a neuron

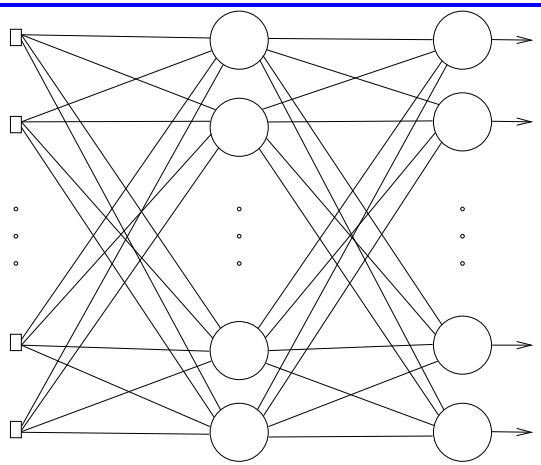
Incoming signals x_i
Interconnection weights w_i
Bias term (threshold value) b
Activation a
Nonlinearity $f(\cdot)$
Output y



$$a = w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b$$

$$y = f(a)$$

Multilayer perceptron (MLP)



Input $x \in \mathbb{R}^m$, output $y \in \mathbb{R}^l$, hidden layer: n_h hidden neurons
Interconnection weight matrices: $W \in \mathbb{R}^{l \times n_h}$, $V \in \mathbb{R}^{n_h \times m}$
Bias vector (thresholds of hidden neurons): $\beta \in \mathbb{R}^{n_h}$

Matrix-vector notation

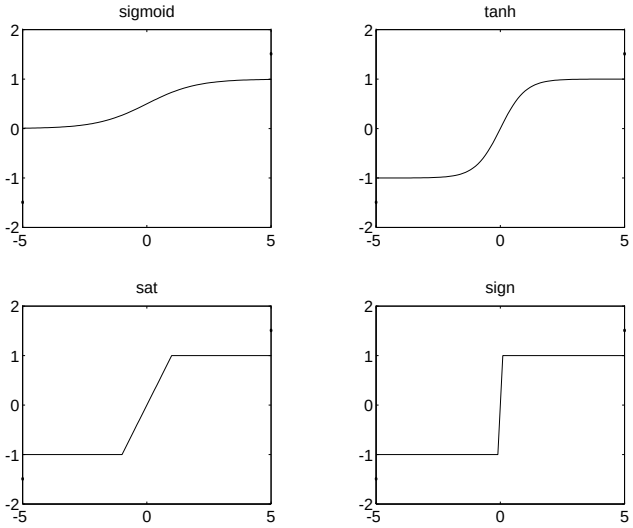
$$y = W\,\sigma(Vx + \beta)$$

Elementwise

$$y_i = \sum_{r=1}^{n_h} w_{ir}\,\sigma(\sum_{j=1}^{m} v_{rj}x_j + \beta_r),\ i = 1, ..., l$$

Activation functions $\sigma(\cdot)$

- Some examples:

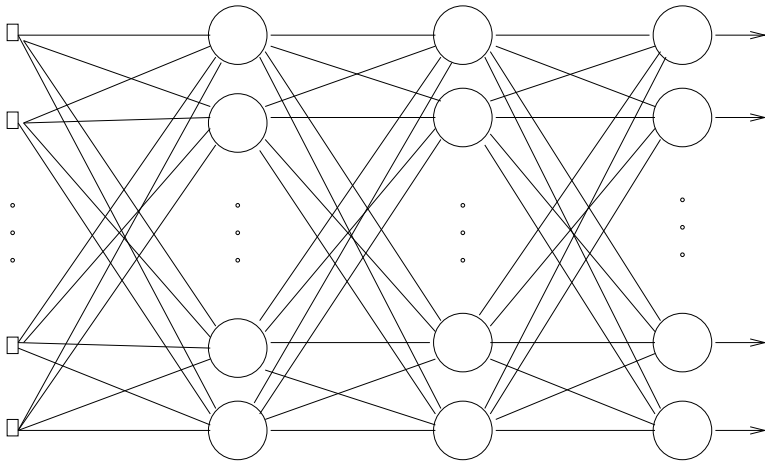


- The choice of the activation function depends on the specific application (classification, regression, ...). For neurons of input and output layer a linear characteristic is often chosen.
- Derivatives of $\sigma(\cdot)$:

sigmoid: $\sigma' = \sigma(1-\sigma) \quad [\sigma(x) = \frac{1}{1+\exp(-x)}]$

tanh: $\sigma' = 1-\sigma^2 \quad [\tanh(x) = \frac{1-\exp(-2x)}{1+\exp(-2x)}]$

MLP with two hidden layers



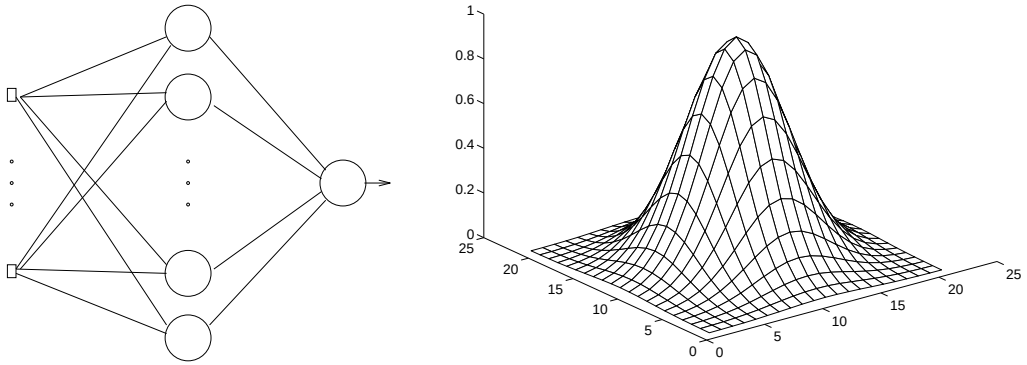
Input $x \in \mathbb{R}^m$, output $y \in \mathbb{R}^l$, 2 hidden layers: n_{h_1}, n_{h_2} hidden neurons
Interconnection matrices: $W \in \mathbb{R}^{l \times n_{h_2}}$, $V_2 \in \mathbb{R}^{n_{h_2} \times n_{h_1}}$, $V_1 \in \mathbb{R}^{n_{h_1} \times m}$
Bias vectors: $\beta_2 \in \mathbb{R}^{n_{h_2}}$, $\beta_1 \in \mathbb{R}^{n_{h_1}}$

Matrix-vector notation

$$y = W\,\sigma(V_2\,\sigma(V_1 x + \beta_1) + \beta_2)$$

Elementwise: $y_i = \sum_{r=1}^{n_{h_2}} w_{ir}\,\sigma(\sum_{s=1}^{n_{h_1}} v_{rs}^{(2)}\,\sigma(\sum_{j=1}^{m} v_{sj}^{(1)} x_j + \beta_s^{(1)}) + \beta_r^{(2)}),\ i = 1, ..., l$

Radial basis function (RBF) network



RBF network:

$$y = \sum_{i=1}^{n_h} w_i \, h(\|x - c_i\|)$$

with input $x \in \mathbb{R}^m$, output $y \in \mathbb{R}$, n_h hidden neurons, output weights w_i, centers $c_i \in \mathbb{R}^m$ and Gaussian activation function

$$y = \sum_{i=1}^{n_h} w_i \exp(-\frac{\|x - c_i\|_2^2}{\sigma_i^2})$$

Universal approximation by neural nets (1)

- **Hilbert's 13th problem**

  In 1900 Hilbert formulated a list of 23 challenging mathematical problems for the century to come.

- Famous 13th problem:

  Conjecture: There are analytical functions of three variables which cannot be represented as a finite superposition of continuous functions of only two variables.

  This conjecture was refuted by Kolmogorov and Arnold (1957).

Universal approximation by neural nets (2)

- **Kolmogorov Theorem (1957)**
  Any continuous function $f(x_1, ..., x_n)$ defined on $[0,1]^n$ with $(n \geq 2)$ can be represented in the form

$$f(x) = \sum_{j=1}^{2n+1} \chi_j (\sum_{i=1}^{n} \phi_{ij}(x_i))$$

  where χ_j, ϕ_{ij} are continuous functions and ϕ_{ij} are also monotone.

- **Example:** $n = 3$

$$\begin{aligned} f(x_1, x_2, x_3) &= \sum_{j=1}^{7} \chi_j (\sum_{i=1}^{3} \phi_{ij}(x_i)) \\ &= \sum_{j=1}^{7} \chi_j (\phi_{1j}(x_1) + \phi_{2j}(x_2) + \phi_{3j}(x_3)) \end{aligned}$$

Universal approximation by neural nets (3)

- **Sprecher Theorem (1965) [refined version of Kolmogorov]**
  There exists a real monotone increasing function $\phi(x) : [0,1] \rightarrow [0,1]$ depending on n $(n \geq 2)$ having the following property: $\forall \delta > 0, \exists \epsilon$ $(0 < \epsilon < \delta)$ such that every real continuous function $f(x) : [0,1]^n \rightarrow \mathbb{R}$ can be represented as

$$f(x) = \sum_{j=1}^{2n+1} \chi[\sum_{i=1}^{n} \lambda^i \phi(x_i + \epsilon(j-1)) + j - 1]$$

  where χ is a real continuous function and λ is a constant.

- **Hecht-Nielsen Theorem (1987) [based on Sprecher Theorem]**
  Any continuous mapping $f(x) : [0,1]^n \rightarrow \mathbb{R}^m$ can be represented by a three layer NN (two hidden layers).

Universal approximation by neural nets (4)

- **Hornik (1989)**
  Regardless of the activation function (can be discontinuous), the dimension of the input space, a NN with one hidden layer can approximate *any* continuous function arbitrarily well (in a certain metric).
  The results extend to NN's with multiple outputs.

- **Leshno et al. (1993)**
  A standard multilayer feedforward NN with locally bounded piecewise continuous activation function can approximate any continuous function to any degree of accuracy *if and only if* the network's activation function is not a polynomial.

- **Universal approximation of RBF networks:**
  Park and Sandberg (1991) showed that it is possible to approximate any continuous nonlinear function by means of an RBF network.

Universal approximation by neural nets (5)

Important remarks:

The proofs of the universal approximation Theorems are **not constructive** in the following sense:

1. Given a nonlinear function to be approximated, the proof does not provide us with an algorithm for determination of the interconnection weights of the NN.

2. It is not clear how many hidden neurons are sufficient to approximate a given nonlinear function on a compact interval.

Curse of dimensionality (1)

- **Curse of dimensionality - Barron (1993)**
  Neural networks avoid the curse of dimensionality in the sense that the approximation error becomes independent from the dimension of the input space (under certain conditions), which is not the case e.g. for polynomial expansions.

- Approximation error:

  One hidden layer MLP: $\mathcal{O}(1/n_h)$
  Polynomial expansion: $\mathcal{O}(1/n_p^{2/n})$

  with

  n_h number of hidden units
  n dimension of input space
  n_p number of terms in expansion

Curse of dimensionality (2)

Example: $y = f(x_1, x_2, x_3)$

- Polynomial expansion (many terms needed!):

$$\begin{aligned} y \quad = \quad & a_1x_1 + a_2x_2 + a_3x_3 + a_{11}x_1^2 + a_{22}x_2^2 + a_{33}x_3^2 + \\ & a_{12}x_1x_2 + a_{13}x_1x_3 + a_{23}x_2x_3 + a_{111}x_1^3 + a_{222}x_2^3 + \\ & a_{333}x_3^3 + ... + a_{1111111}x_1^7 + a_{2222222}x_2^7 + ... \end{aligned}$$

- MLP:

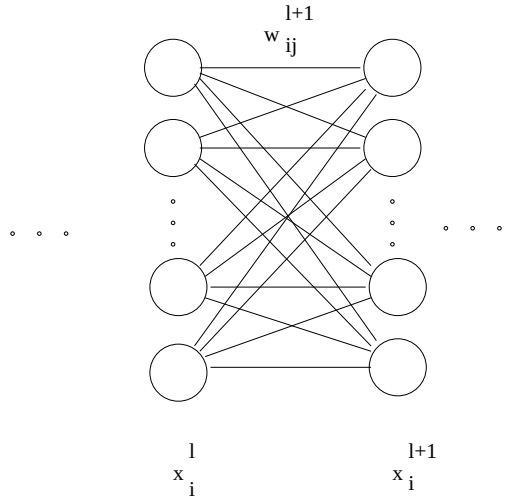
$$y = w^T \tanh(V \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} + \beta)$$

Backpropagation algorithm (1)

Description of multilayer feedforward neural network

- L layers ($L-1$ hidden layers) [index $l = 1, ..., L$]
- P input patterns and corresponding desired outputs [index $p = 1, ..., P$]
- N_l neurons in layer l
- Relation between layer l and layer $l+1$:

$$x_{i,p}^{l+1} = \sigma(\xi_{i,p}^{l+1}), \qquad \xi_{i,p}^{l+1} = \sum_{j=1}^{N_l} w_{ij}^{l+1} x_{j,p}^{l}$$



Backpropagation algorithm (2)

- **Objective**

$$\min_{w_{ij}^l} E = \frac{1}{P}\sum_{p=1}^{P} E_p \quad \text{with} \quad E_p = \frac{1}{2}\sum_{i=1}^{N_L}(x_{i,p}^{desired} - x_{i,p}^{L})^2$$

  where $x_p^{desired}$ is the desired output for the p-th input pattern of the training set. E denotes the MSE (Mean Squared Error).

- Define so-called δ variables: $\delta_{i,p}^l = \frac{\partial E_p}{\partial \xi_{i,p}^l}$

Backpropagation algorithm (3)

- **Generalized delta rule**

$$\begin{cases} \Delta w_{ij}^{l} = \eta\, \delta_{i,p}^{l}\, x_{j,p}^{l-1} = -\eta\, \dfrac{\partial E_p}{\partial w_{ij}^{l}} \\ \\ \delta_{i,p}^{L} = (x_{i,p}^{desired} - x_{i,p}^{L})\, \sigma'(\xi_{i,p}^{L}) \\ \\ \delta_{i,p}^{l} = (\displaystyle\sum_{r=1}^{N_{l+1}} \delta_{r,p}^{l+1} w_{ri}^{l+1})\, \sigma'(\xi_{i,p}^{l}), \qquad l = 1, ..., L-1 \end{cases}$$

  where η is the learning rate.

- Using a momentum term:

$$\Delta w_{ij}^{l}(k+1) = \eta\, \delta_{i,p}^{l}\, x_{j,p}^{l-1} + \alpha\, \Delta w_{ij}^{l}(k)$$

  where k is the iteration step and $0 < \alpha < 1$.

Backpropagation algorithm (4)

On-line learning Backpropagation: adapt weights each time after presenting a new pattern p.

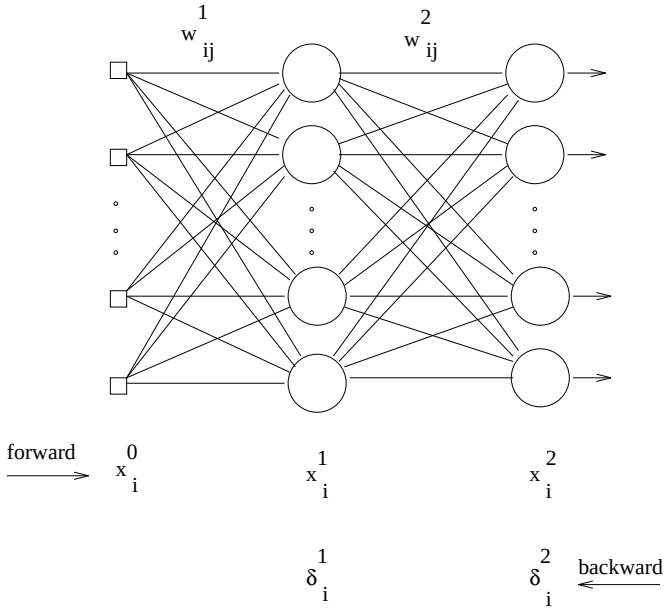
Off-line learning Backpropagation: update of weights after presenting all the training patterns to the network.

Note: more advanced methods for on-line learning exist, e.g. based on extended Kalman filtering.

Backpropagation algorithm (5)

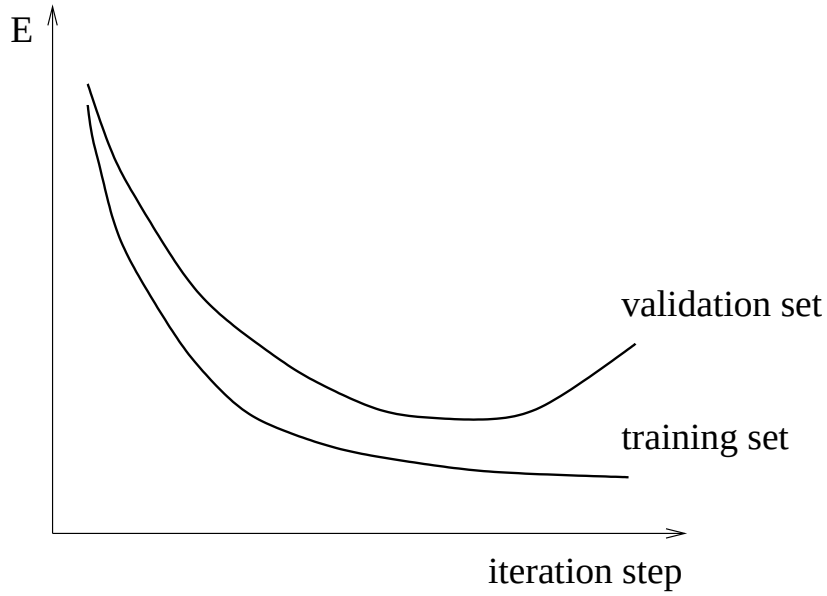
- **Example:** $L = 2$ (1 hidden layer)

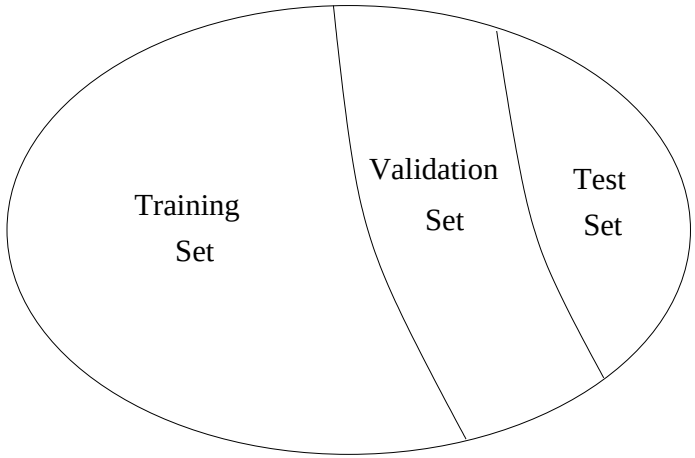
$$\begin{cases} \Delta w_{ij}^2 = \eta\, \delta_{i,p}^2\, x_{j,p}^1 & \Delta w_{ij}^1 = \eta\, \delta_{i,p}^1\, x_{j,p}^0 \\ \delta_{i,p}^2 = (x_{i,p}^{desired} - x_{i,p}^2)\, \sigma'(\xi_{i,p}^2) & \delta_{i,p}^1 = (\sum_{r=1}^{N_2} \delta_{r,p}^2 w_{ri}^2)\, \sigma'(\xi_{i,p}^1) \end{cases}$$



Forward propagation of input patterns
Backward propagation of errors and δ variables

Overfitting and early stopping

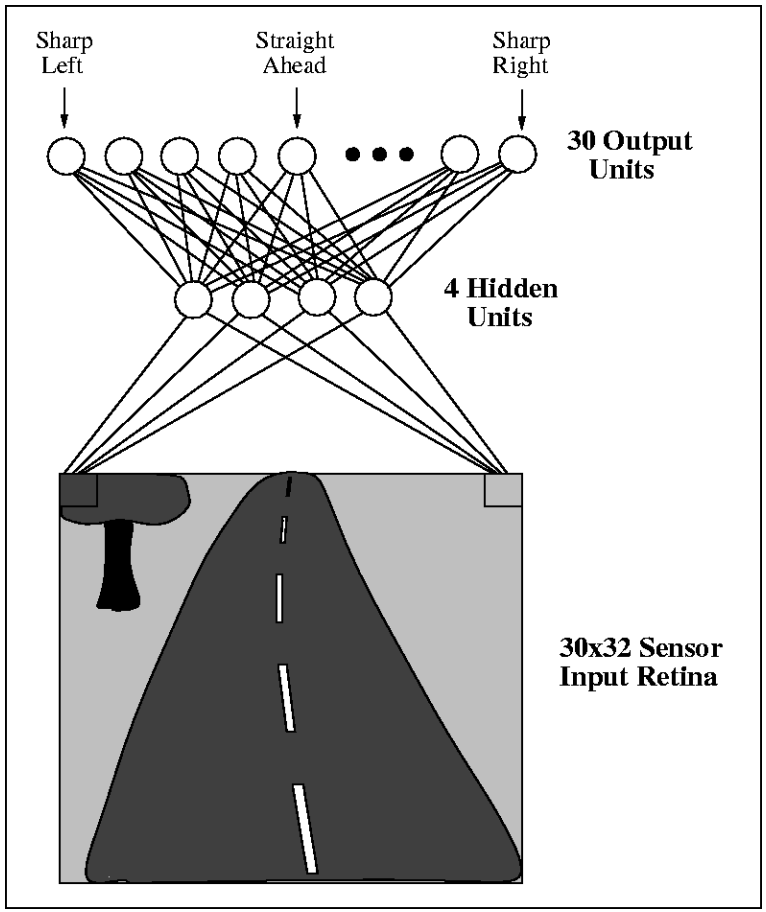




- Training set: used for training
- Validation set: used for stopping
- Test set

Stop when minimal error on validation set is reached

ALVINN example



Perceptron

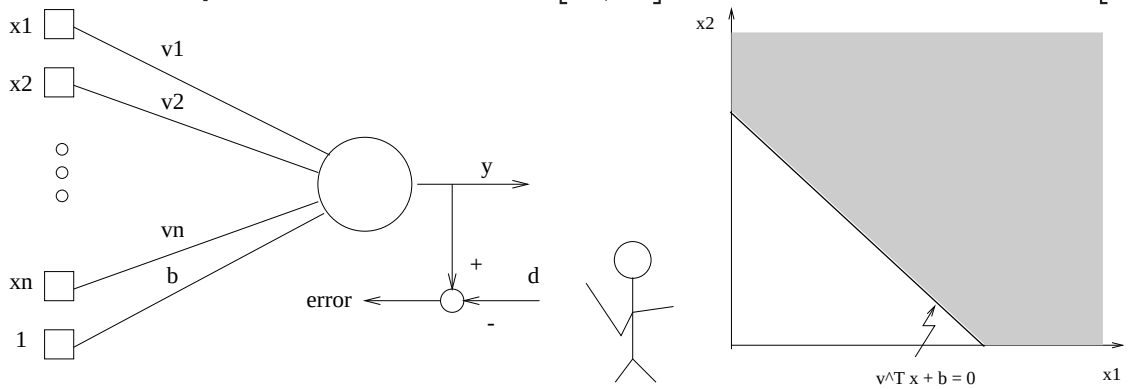
- Perceptron: consists of one neuron with sign activation function

- **Perceptron training algorithm:**
  Given training set $\{x^{(i)}, d^{(i)}\}_{i=1}^{N}$
  Input $x \in \mathbb{R}^n$, output $y \in \mathbb{R}$, desired outputs $d^{(i)}$

  Perceptron network:

$$y = \text{sign}(v^T x + b) = \text{sign}(w^T z)$$

  with augmented input vector $z = [x; 1] \in \mathbb{R}^{n+1}$ and $w = [v; b]$



Perceptron algorithm (1)

1. Choose $c > 0$

2. Initialize small random weights w.
   Set $k = 1, i = 1$ and set cost function $E = 0$.

3. The training cycle begins here.
   Present i-th input and compute corresponding output:

$$y^{(i)} = \text{sign}(w^T z^{(i)})$$

4. Update weights:

$$w := w + \frac{1}{2} c \left[d^{(i)} - y^{(i)} \right] z^{(i)}$$

Perceptron algorithm (2)

5. Compute cycle error:

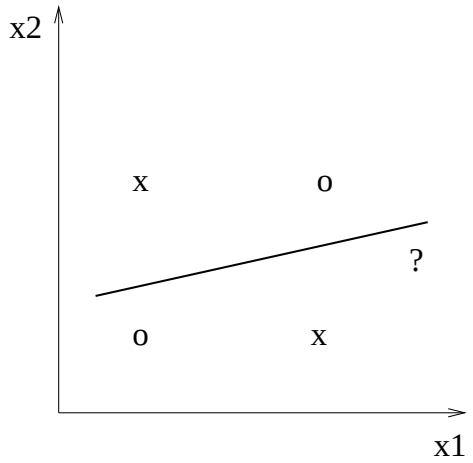
$$E := E + \frac{1}{2}[d^{(i)} - y^{(i)}]^2$$

6. If $i < N$, then $i := i + 1$ and $k := k + 1$ and go to step 3.
   Otherwise, go to step 7.

7. If $E = 0$ then stop the training.
   If $E > 0$ then set $E = 0, i = 1$ and enter a new training cycle by going to step 3.

Linear separability (1)

- Example of a not-linearly separable problem:
  exclusive-OR (XOR) problem



This problem cannot be solved by a perceptron.
A multilayer perceptron with one hidden layer can solve it.

Linear separability (2)

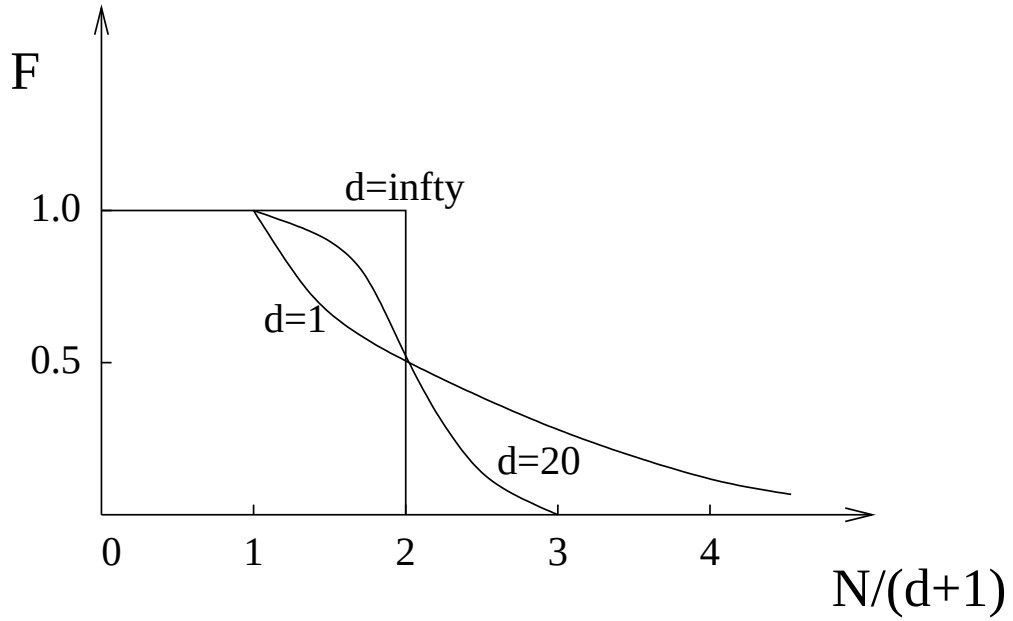
- Consider the case of continuous input variables (d dimensions).
  Question: what is the probability that a random set of patterns will be linearly separable ?
  Assign randomly points to classes $\mathcal{C}_1, \mathcal{C}_2$ with equal probability. Each possible assignment for the complete data set is called a *dichotomy*.
  For N points there are 2^N possible dichotomies.

- **Cover's Theorem [Cover, 1965]**
  The fraction $F(N, d)$ of dichotomies that is linearly separable:

$$F(N,d) = \begin{cases} 1 & , \quad N \leq d+1 \\ \frac{1}{2^{N-1}} \sum_{i=0}^{d} \begin{pmatrix} N-1 \\ i \end{pmatrix} & , \quad N \geq d+1 \end{cases}$$

  with $\begin{pmatrix} N \\ M \end{pmatrix} = \frac{N!}{(N-M)!M!}$ the number of combinations of M objects selected from a total of N.

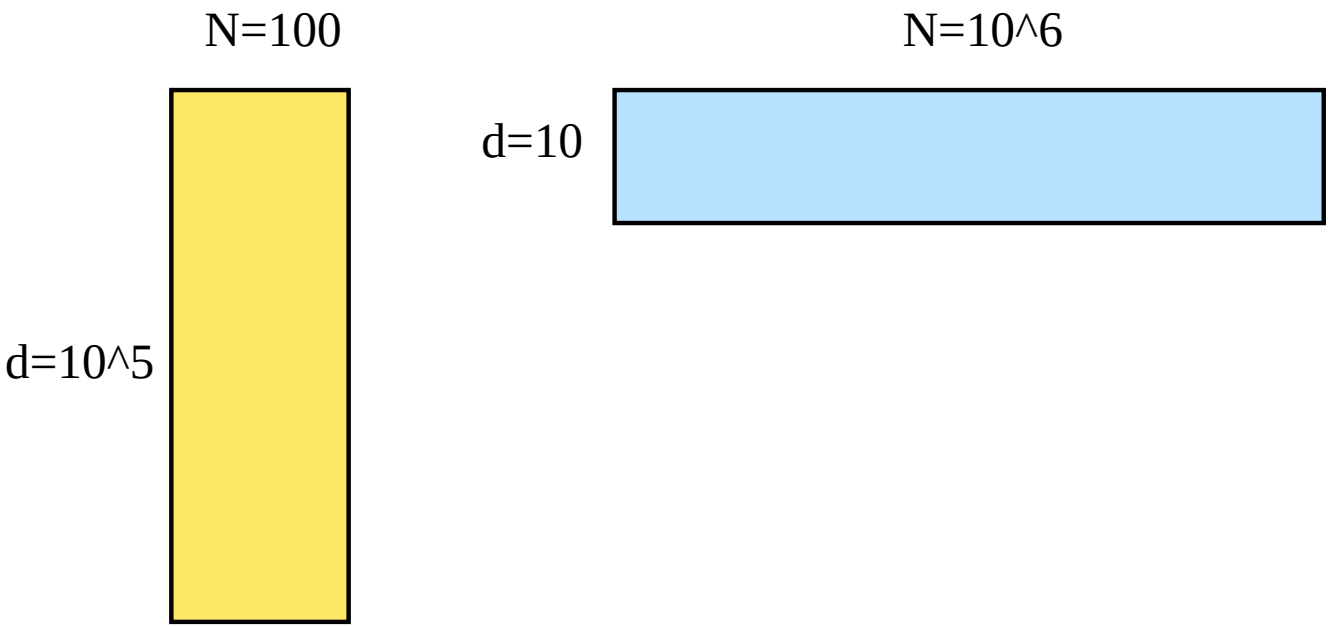
Linear separability (3)



If $N < d+1$ then any labelling of points will always lead to linear separability.

Important conclusion:
For larger d it becomes likely that more dichotomies are linearly separable.

Linear separability (4)



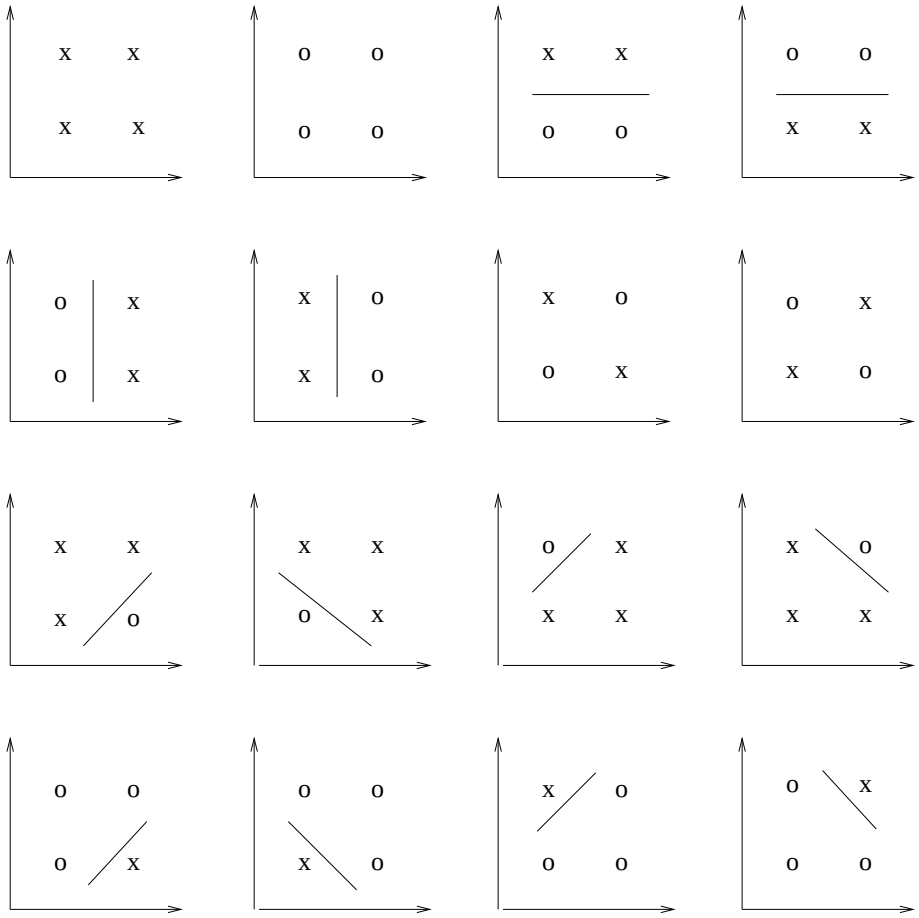
Example:
- $d = 10^5$, $N = 100$: e.g. microarray data set
- $d = 10$, $N = 10^6$: e.g. fraud detection data set

Linear separability (5)

Example:

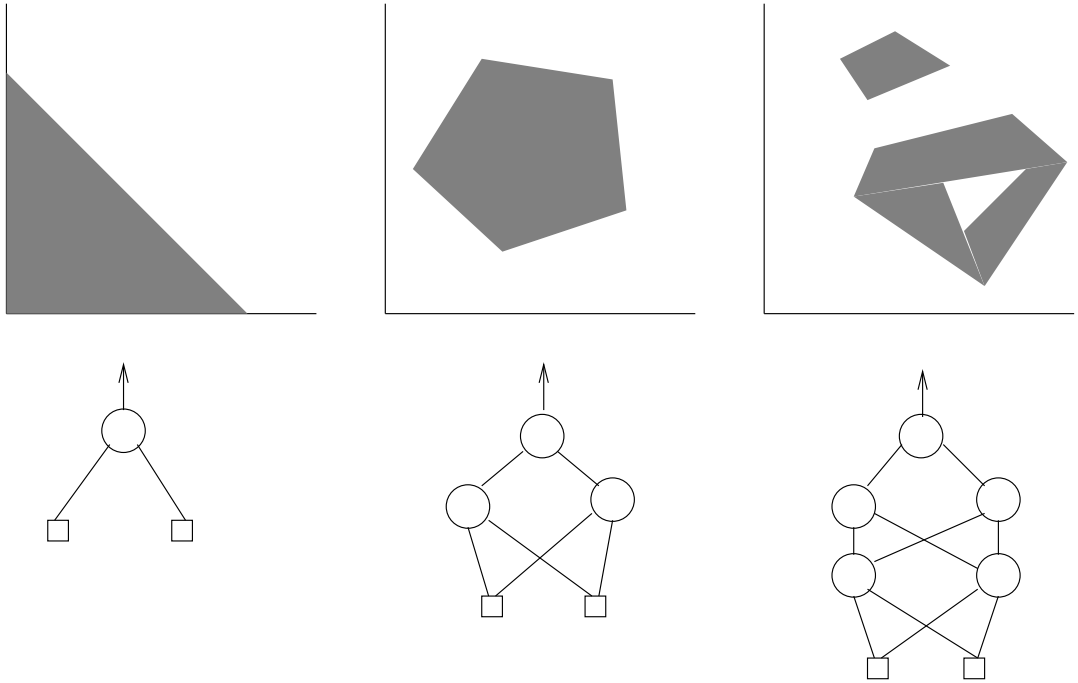
$N = 4, d = 2$: $2^4 = 16$ dichotomies

14 dichotomies are linearly separable (everything but XOR)



Nonlinear separation with MLPs

From perceptron to multilayer perceptron (MLP) classifier:



One neuron (perceptron): Linear separation
One hidden layer: Realization of convex regions
Two hidden layers: Realization of non-convex regions

Classification problems (1)

- Popular approach: solve classification problem as a regression problem with class labels as targets.
  Consider MLP with input $x \in \mathbb{R}^m$ and output $y \in \mathbb{R}^l$:

$$y = W \tanh(Vx + \beta).$$

  Work with outputs $\{-1, +1\}$ (or $\{0, 1\}$) to encode the classes.
  With l outputs one could encode 2^l classes, e.g. for $l = 2$:

	y_1	y_2
Class 1	$+1$	$+1$
Class 2	$+1$	-1
Class 3	-1	$+1$
Class 4	-1	-1

Classification problems (2)

On the other hand one can also use one output per class:

	y_1	y_2	y_3	y_4
Class 1	$\boxed{+1}$	-1	-1	-1
Class 2	-1	$\boxed{+1}$	-1	-1
Class 3	-1	-1	$\boxed{+1}$	-1
Class 4	-1	-1	-1	$\boxed{+1}$

Training can be done in the same way as for regression with the class labels as target values for the outputs. After training of the classifier, decisions are made by the classifier as follows:

$$y = \text{sign}[W \tanh(Vx + \beta)].$$

If the output does not correspond to one of the defined class codes, one can assign it to the class which is closest in terms of Hamming distance.

Alphabet recognition

Example: Alphabet recognition