

# ELEC2870 - Machine learning: regression and dimensionality reduction

## *Nonlinear regression with Multi-Layer Perceptrons*

Michel Verleysen

Machine Learning Group

Université catholique de Louvain

Louvain-la-Neuve, Belgium

michel.verleysen@uclouvain.be

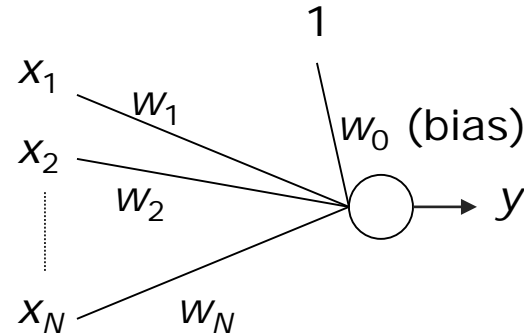
# Outline

- Motivation
- Single-layer nonlinear regression
- Multi-layer perceptron
  - Model
  - MLP with threshold units
  - Number of layers
  - Learning
    - Error back-propagation
    - Weight adjustment
- Applications

# Nonlinear regression: motivation

- Remember the linear model

$$y = \mathbf{w}^T \mathbf{x}$$



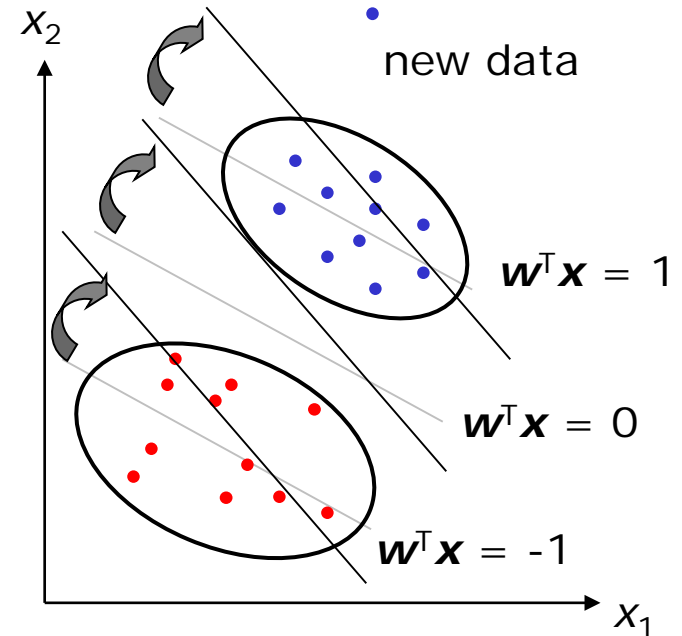
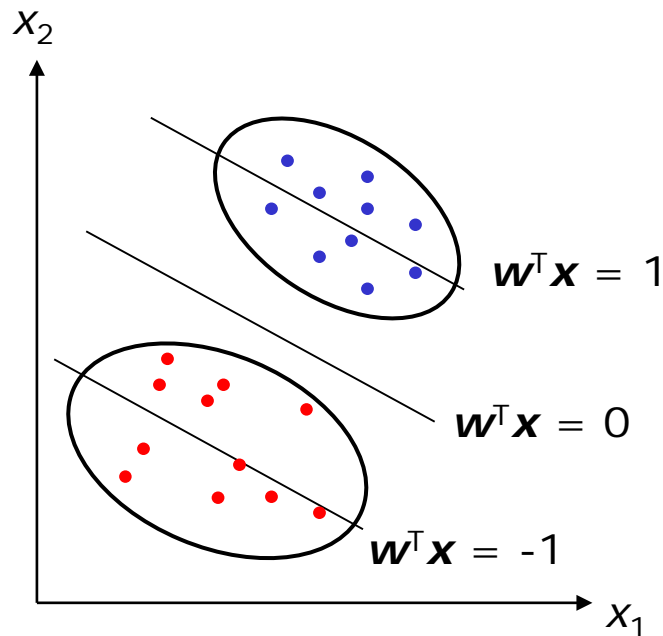
- and the sum-of-squares criterion

$$E = \frac{1}{P} \sum_{p=1}^P (t^p - y^p)^2 = \frac{1}{P} \sum_{p=1}^P (t^p - \mathbf{w}^T \mathbf{x}^p)^2$$

- The influence of a *large* single error on this criterion is *very large* (because the error is squared)

# Nonlinear regression: motivation

- When the dataset is quite small, a single outlier may have a dramatic influence:

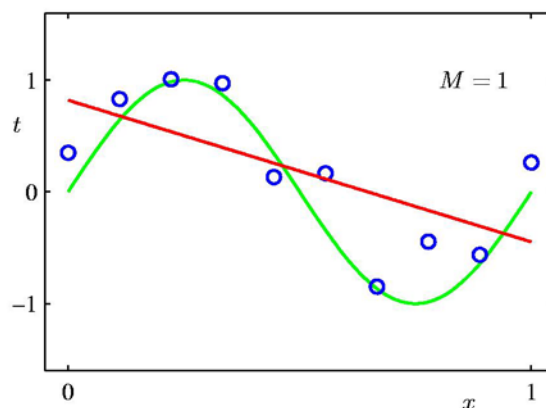


- This is true both in classification and regression!

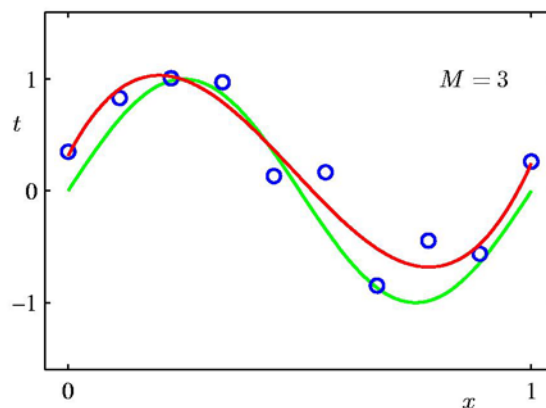
# Nonlinear regression: motivation

- Remember also that a linear model simply does not the expected job...

– Linear model



– Nonlinear model



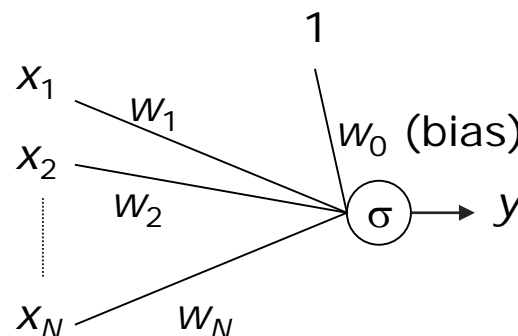
# Outline

- Motivation
- Single-layer nonlinear regression
- Multi-layer perceptron
  - Model
  - MLP with threshold units
  - Number of layers
  - Learning
    - Error back-propagation
    - Weight adjustment
- Applications

# Single-layer nonlinear regression

- Model identical to linear regression, but with *nonlinear activation function*

$$y = \sigma(\mathbf{w}^T \mathbf{x})$$



- The error function becomes

$$E = \frac{1}{P} \sum_{k=1}^P \left( t^k - \sigma(\mathbf{w}^T \mathbf{x}^k) \right)^2$$

- And the stochastic gradient descent rule

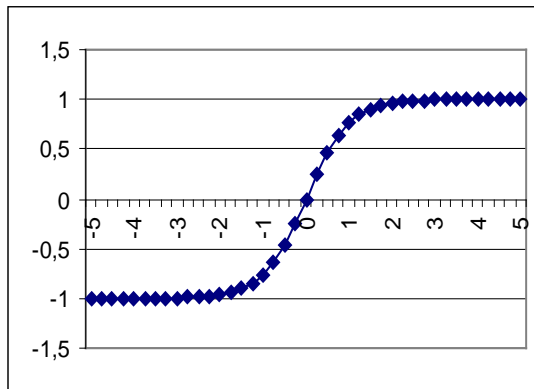
$$\mathbf{w}(t+1) = \mathbf{w}(t) + \frac{2}{P} \alpha \left( t^k - \sigma(\mathbf{w}(t)^T \mathbf{x}^k) \right) \mathbf{x}^k \frac{\partial \sigma}{\partial p} \bigg|_{p=\mathbf{w}(t)^T \mathbf{x}^k}$$

(sometimes called the *generalized delta rule*)

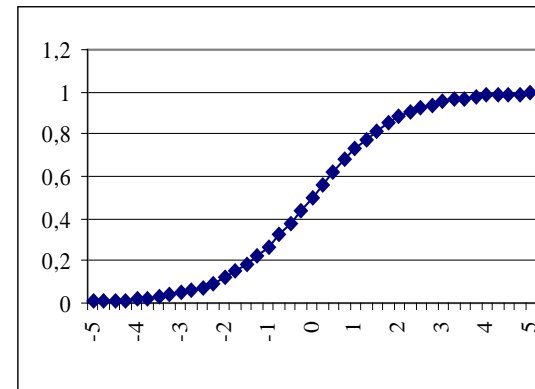
# Nonlinear activation functions

- Commonly used non-linear activation function

hyperbolic tangent



logistic function

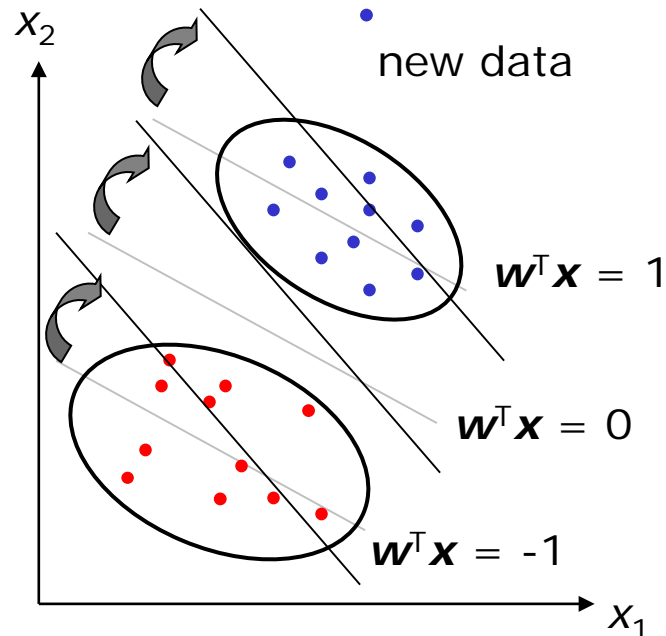


- any slope
- slope can be a parameter
- logistic function: used to estimate posterior probabilities

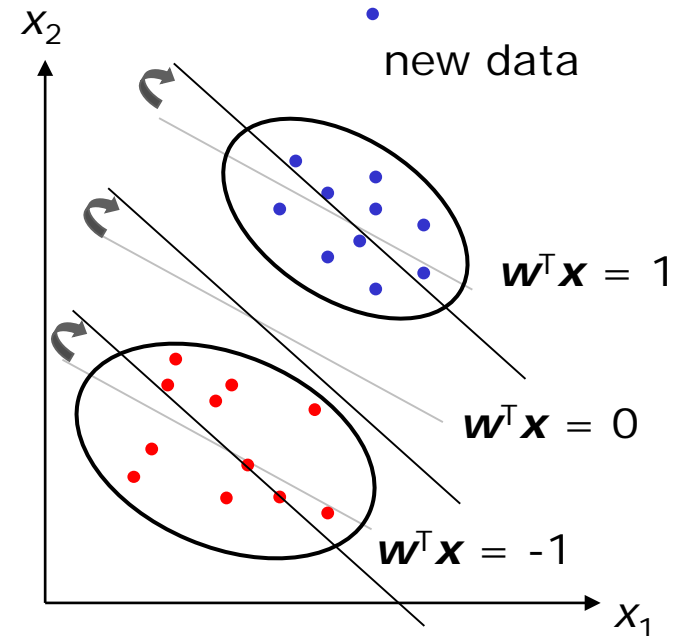


# Nonlinear effect on outliers

without non-linearity



with non-linearity



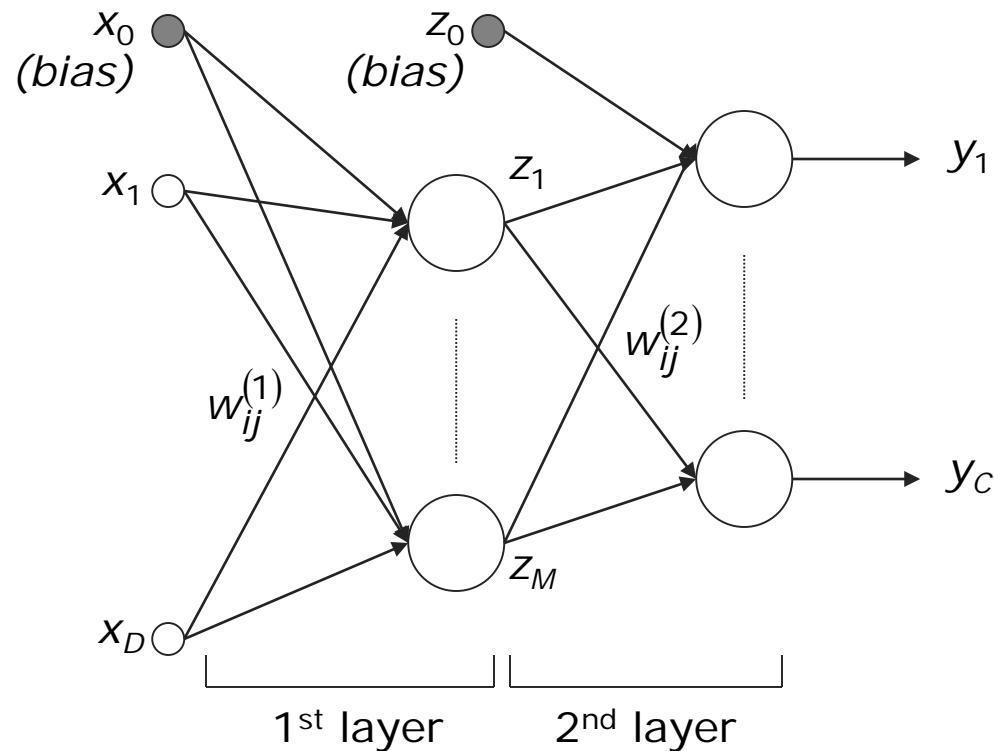
- Why? Because  $|\sigma(\mathbf{w}^T \mathbf{x}^k)|$  never exceeds one, therefore  $t^k - \sigma(\mathbf{w}^T \mathbf{x}^k)$  is almost 0 for well-classified points
- Not so simple in regression

# Outline

- Motivation
- Single-layer nonlinear regression
- Multi-layer perceptron
  - Model
  - MLP with threshold units
  - Number of layers
  - Learning
    - Error back-propagation
    - Weight adjustment
- Applications

# Multi-Layer Perceptron (MLP)

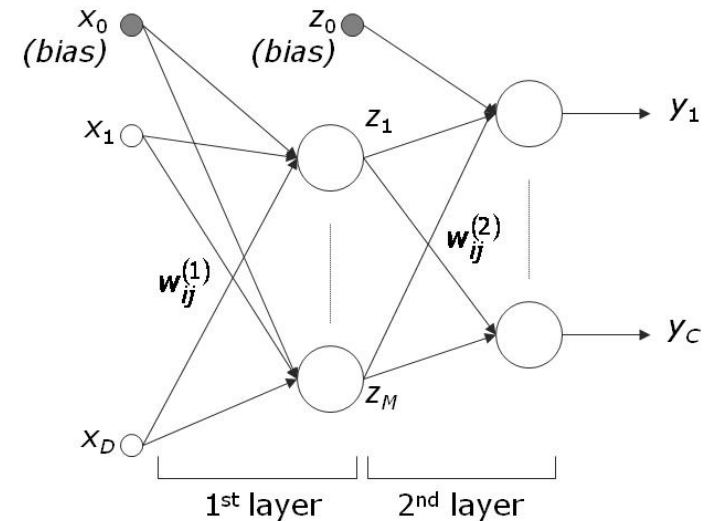
- several layers of weights and activation units



# Multi-Layer Perceptron

$$y_k(\mathbf{x}) = h \left( \sum_{i=0}^M w_{ki}^{(2)} g \left( \sum_{j=0}^D w_{ij}^{(1)} x_j \right) \right)$$

$$\mathbf{y}(\mathbf{x}) = \mathbf{h}(\mathbf{w}^{(2)} \mathbf{g}(\mathbf{w}^{(1)} \mathbf{x}))$$



- Convention: 2 layers of weights (in literature: sometimes 3 layers of *units* or *neurons*)
- $g$  and  $h$  can be threshold (sign) units or continuous ones
- $h$  can be linear but not  $g$  (otherwise only one layer)

# Multi-Layer Perceptron

- How many layers
  - can we use ?
  - should we use ?
- In theory:
  - Any number of layers (see back-propagation algorithm)
- In practice:
  - More layers means more  $w_{ij}^{(l)}$  parameters
  - Is it really needed?
  - How many layers do we need to approximate any function?

# Outline

- Motivation
- Single-layer nonlinear regression
- Multi-layer perceptron
  - Model
  - MLP with threshold units
  - Number of layers
  - Learning
    - Error back-propagation
    - Weight adjustment
- Applications

# MLP with threshold units

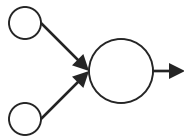
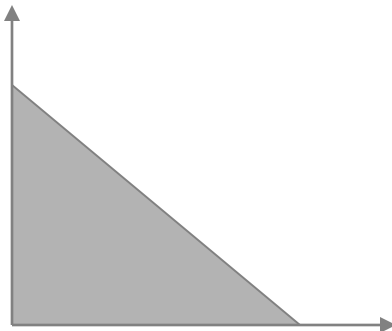
Warning: this is NOT the traditional MLP !

- We discuss the *MLP with threshold units* first to get an intuition about the number of layers
- But then we forget it immediately!
- *Why? Because threshold units means discontinuities, so no algorithm...*
- Outputs:
  - threshold units (all layers) → binary outputs
- Inputs:
  - binary inputs: Boolean function network
    - 2 layers (max.) for any function
    - look-up table without generalisation
  - continuous inputs (classification)
    - data become binary after 1<sup>st</sup> layer
    - shape of decision boundaries depend on # layers

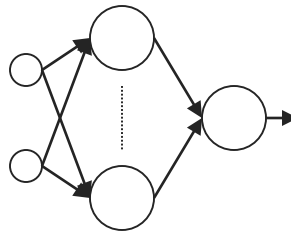
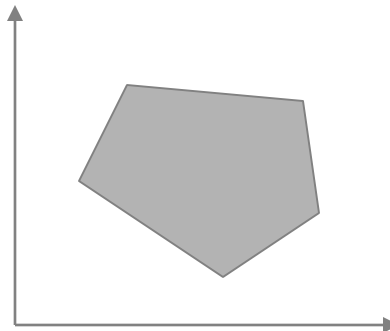
# MLP with threshold units

Warning: this is NOT the traditional MLP !

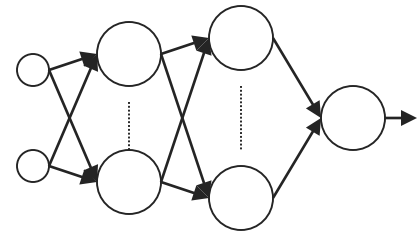
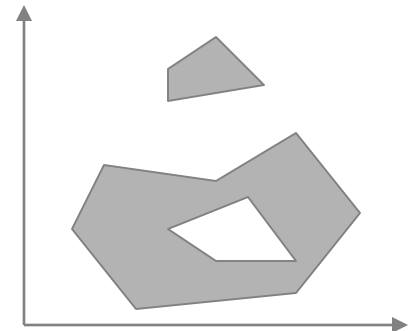
linear



convex



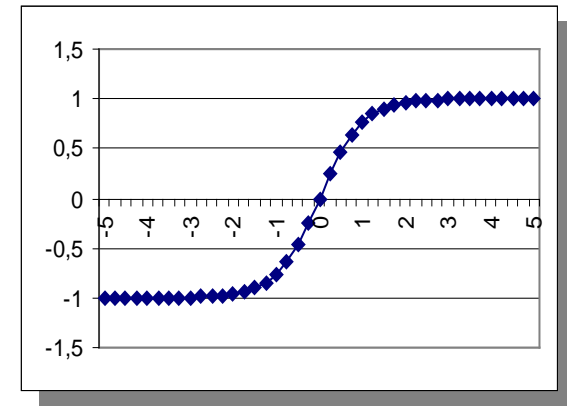
any





# Multi-Layer Perceptron

- General case (no more threshold units!)
- non-linear activation function: sigmoid or hyperbolic tangent
  - at least for “hidden” layers
  - output layer can be linear (otherwise limited output range)
- used for
  - approximation of functions
  - classification

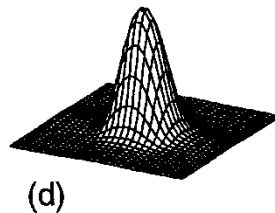
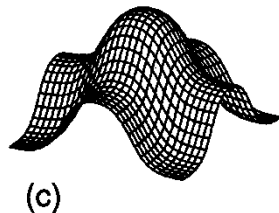
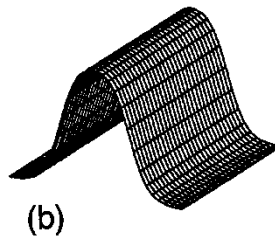
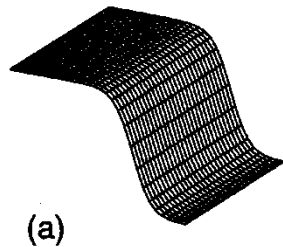


# Outline

- Motivation
- Single-layer nonlinear regression
- Multi-layer perceptron
  - Model
  - MLP with threshold units
  - Number of layers
  - Learning
    - Error back-propagation
    - Weight adjustment
- Applications

# Number of layers

- A 3-layers network can approximate
  - any function
  - with any precision
- Indeed:
  - A 2-layers network can approximate a local function



2 layers:

(a) sigmoid

(b) sum of 2 sigmoids

(c) sum of 4 sigmoids

(d) sigmoid of sum  
(bell-shaped local  
function)

- A 3-layers network can thus approximate any sum of local functions

## Number of layers

- That was intuition...
- Mathematically, it can be proven that

A **2**-layer MLP can approximate arbitrarily well any (functional) continuous mapping, provided the number  $M$  of hidden units is sufficiently large

- This is called the *universal approximation property*
- It is theory: in practice, if  $M$  is too high, overfitting!
- It also valid for decision boundaries (classification)

# Outline

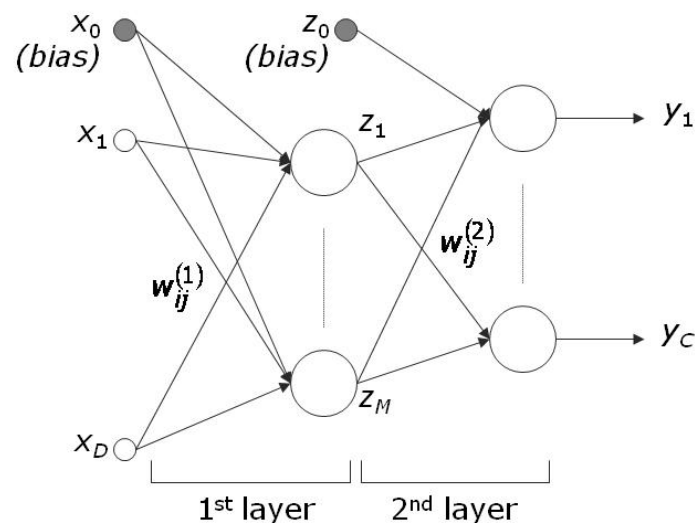
- Motivation
- Single-layer nonlinear regression
- Multi-layer perceptron
  - Model
  - MLP with threshold units
  - Number of layers
  - Learning
    - Error back-propagation
    - Weight adjustment
- Applications

# Learning in MLP

- Learning =
  - definition of an error criterion  $E$
  - evaluation of derivatives of  $E$  w.r.t. parameters  $w$
  - adjustments of parameters  $w$  according to derivatives

- Error criterion 
$$E = \frac{1}{P} \sum_{p=1}^P E^p(y_1, \dots, y_C)$$

- Batch / stochastic learning
  - Batch learning (all samples together):  $E$
  - on-line, stochastic (one sample at each iteration):  $E^p$
- In the following, we omit  $p$  (stochastic learning)

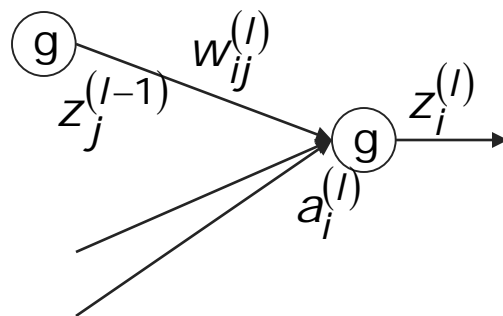


# Outline

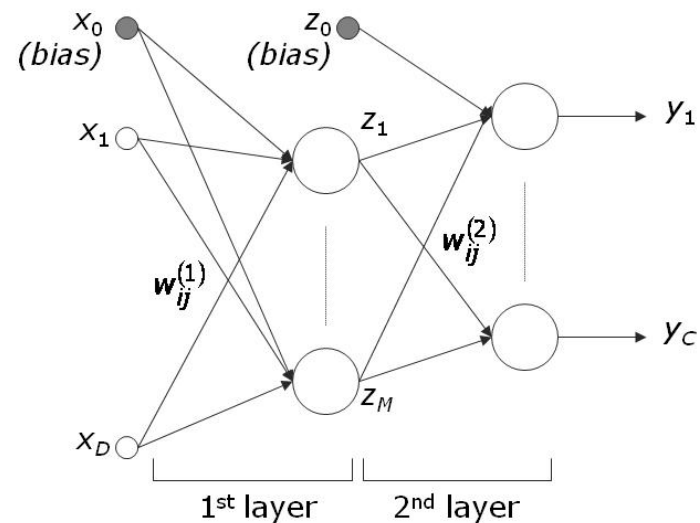
- Motivation
- Single-layer nonlinear regression
- Multi-layer perceptron
  - Model
  - MLP with threshold units
  - Number of layers
  - Learning
    - Error back-propagation
    - Weight adjustment
- Applications

# Error back-propagation

- Some notations



$$a_i^{(l)} = \sum_j w_{ij}^{(l)} z_j^{(l-1)}$$

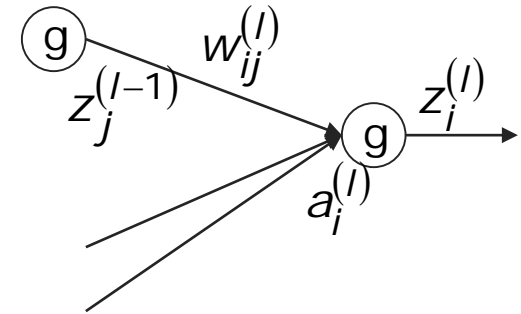




# Error back-propagation

$$a_i^{(l)} = \sum_j w_{ij}^{(l)} z_j^{(l-1)}$$

- Gradient descent: evaluation of  $\frac{\partial E}{\partial w_{ij}^{(l)}}$



$$\begin{aligned} \frac{\partial E}{\partial w_{ij}^{(l)}} &= \frac{\partial E}{\partial a_i^{(l)}} \frac{\partial a_i^{(l)}}{\partial w_{ij}^{(l)}} \\ &= \delta_i^{(l)} z_j^{(l-1)} \end{aligned} \quad \leftarrow \quad \left( \delta_i^{(l)} \equiv \frac{\partial E}{\partial a_i^{(l)}} \text{ to evaluate} \right)$$

# Error back-propagation

$$\delta_i^{(l)} \equiv \frac{\partial E}{\partial a_i^{(l)}} \quad \text{to evaluate}$$

- For output units

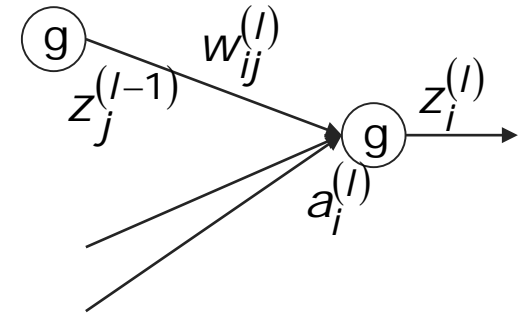
$$\delta_i^{(l)} = \frac{\partial E}{\partial a_i^{(l)}} = \frac{\partial E}{\partial y_i^{(l)}} \frac{\partial y_i^{(l)}}{\partial a_i^{(l)}}$$

$$= \frac{\partial E}{\partial y_i^{(l)}} g'(a_i^{(l)})$$

known

derivative of error criterion (known):  $E = (t_i - y_i)^2$

$$\Rightarrow \frac{\partial E}{\partial y_i} = -2(t_i - y_i)$$

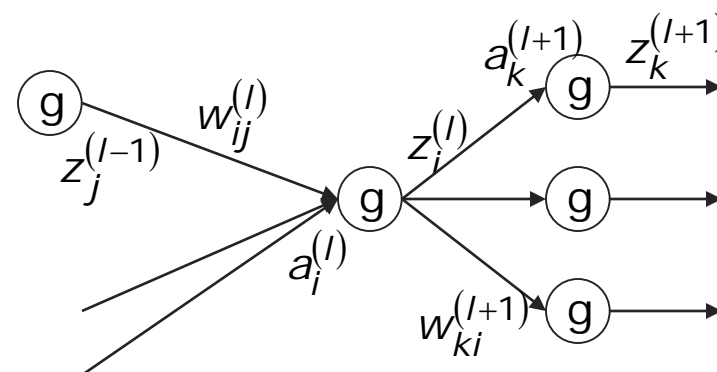


# Error back-propagation

$$\delta_i^{(l)} \equiv \frac{\partial E}{\partial a_i^{(l)}} \quad \text{to evaluate}$$

- For hidden units

$$\begin{aligned} \delta_i^{(l)} &= \frac{\partial E}{\partial a_i^{(l)}} = \sum_k \frac{\partial E}{\partial a_k^{(l+1)}} \frac{\partial a_k^{(l+1)}}{\partial a_i^{(l)}} \\ &= \sum_k \left( \delta_k^{(l+1)} \frac{\partial \left( \sum_{j \in (l)} w_{kj}^{(l+1)} z_j^{(l)} \right)}{\partial a_i^{(l)}} \right) \\ &= \sum_k \left( \delta_k^{(l+1)} w_{ki}^{(l+1)} g'(a_i^{(l)}) \right) = g'(a_i^{(l)}) \sum_k \left( \delta_k^{(l+1)} w_{ki}^{(l+1)} \right) \end{aligned}$$



The error term ( $\delta$ ) is expressed as a combination of errors in the next layer

# Error back-propagation

- Algorithm:
  - Apply an input vector  $\mathbf{x}^k$  and propagate it through the network to evaluate all activations  $z_i^{(l)}$  and neuron outputs  $a_i^{(l)}$
  - Evaluate error terms  $\delta_i^{(o)}$  in output layer
  - Back-propagate error terms  $\delta_i^{(l)}$  to find error terms  $\delta_i^{(l-1)}$
  - Evaluate all derivatives  $\frac{\partial E}{\partial w_{ij}^{(l)}} = \delta_i^{(l)} z_j^{(l-1)}$
  - Adjust weights according to derivatives and a gradient descent scheme

# Outline

- Motivation
- Single-layer nonlinear regression
- Multi-layer perceptron
  - Model
  - MLP with threshold units
  - Number of layers
  - Learning
    - Error back-propagation
    - Weight adjustment
- Applications

# Weight adjustment

- Gradient descent is nice, but...
  - it can be very slow (when there are many parameters)
  - it is easily stuck in local minima
  - it is quite easy to do better...
- Now we have the derivatives  $\frac{\partial E}{\partial w_{ij}^{(l)}}$ , how to adjust weights?
- Notations:

– weight update  $\delta \mathbf{w} \equiv \mathbf{w}(t+1) - \mathbf{w}(t)$

– gradient  $\left( \frac{\partial E}{\partial \mathbf{w}} \right)^T \equiv \left( \frac{\partial E}{\partial w_1} \frac{\partial E}{\partial w_2} \dots \frac{\partial E}{\partial w_D} \right)$

– Hessian

$$H \equiv \left( \frac{\partial^2 E}{\partial \mathbf{w}^2} \right) \equiv \begin{pmatrix} \frac{\partial^2 E}{\partial w_1^2} & \frac{\partial^2 E}{\partial w_1 \partial w_2} & \dots & \frac{\partial^2 E}{\partial w_1 \partial w_D} \\ \vdots & \vdots & & \vdots \\ \frac{\partial^2 E}{\partial w_D \partial w_1} & \frac{\partial^2 E}{\partial w_D \partial w_2} & \dots & \frac{\partial^2 E}{\partial w_D^2} \end{pmatrix}$$

Indices are changed for simplicity of notations:

$$w_{ij}^{(l)}, \forall i, j, l$$

↓

$$w_i, 1 \leq i \leq D$$

# Weight adjustment

- First-order methods :

$$\begin{aligned}
 E(\mathbf{w}(t+1)) &= E(\mathbf{w}(t)) + \left( \frac{\partial E}{\partial \mathbf{w}} \bigg|_{\mathbf{w}(t)} \right)^T (\mathbf{w}(t+1) - \mathbf{w}(t)) \\
 &= E(\mathbf{w}(t)) + \left( \frac{\partial E}{\partial \mathbf{w}} \bigg|_{\mathbf{w}(t)} \right)^T \delta \mathbf{w}
 \end{aligned}$$

- For a  $\delta \mathbf{w}$  of a given magnitude, largest  $|\delta E|$  is found when gradient and weight update vectors are parallel

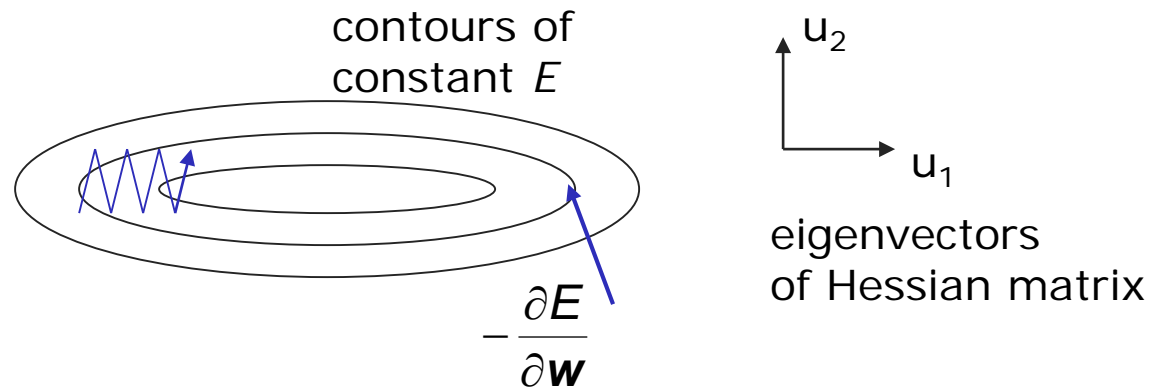
- Adaptation rule:  $\mathbf{w}(t+1) = \mathbf{w}(t) - \alpha \frac{\partial E}{\partial \mathbf{w}} \bigg|_{\mathbf{w}(t)}$

- This is gradient descent; nice, isn't it?

# Weight adjustment

- First-order methods

- Note that  $-\frac{\partial E}{\partial \mathbf{w}}$  does not point towards the minimum of  $E$  !





# Weight adjustment

- First-order methods: improvements
  - Momentum : to avoid sharp changes in gradient direction (stochastic scheme, outliers, etc.)

$$\mathbf{w}(t+1) = \mathbf{w}(t) - \alpha \frac{\partial E}{\partial \mathbf{w}} \bigg|_{\mathbf{w}(t)} + \beta (\mathbf{w}(t) - \mathbf{w}(t-1))$$

$$\beta = \sim 0.9$$

# Weight adjustment

- First-order methods: improvements
  - adaptive learning rate  $\alpha$

$$w_{ij}(t+1) = w_{ij}(t) - \alpha_{ij}(t) \left. \frac{\partial E}{\partial w_{ij}} \right|_{\mathbf{w}(t)}$$

$$\delta w_{ij}(t-1) \delta w_{ij}(t) > 0 \Rightarrow \alpha_{ij}(t+1) = \alpha_{ij}(t) + \kappa$$

$$\delta w_{ij}(t-1) \delta w_{ij}(t) < 0 \Rightarrow \alpha_{ij}(t+1) = \alpha_{ij}(t) - \kappa$$

- Need for maximum safe step size
- This is more or less the « delta-bar-delta » rule

# Weight adjustment

- Second-order methods (Newton's method)

$$E(\mathbf{w}(t+1)) = E(\mathbf{w}(t)) + \left( \frac{\partial E}{\partial \mathbf{w}} \Big|_{\mathbf{w}(t)} \right)^T \delta \mathbf{w} + \frac{1}{2} \delta \mathbf{w}^T \frac{\partial^2 E}{\partial \mathbf{w}^2} \Big|_{\mathbf{w}(t)} \delta \mathbf{w}$$

- Stationary point (zero derivative) of this quadratic form:

$$\delta \mathbf{w} = - \left( \frac{\partial^2 E}{\partial \mathbf{w}^2} \Big|_{\mathbf{w}(t)} \right)^{-1} \frac{\partial E}{\partial \mathbf{w}} \Big|_{\mathbf{w}(t)}$$

- Adaptation rule:
  - minimum: move in this direction
  - maximum or saddle point: move in conjugate direction
  - Many "conjugate gradient" rules (some are line searches)

# Weight adjustment

- Second-order methods (Newton's method)

$$\delta \mathbf{w} = - \left( \frac{\partial^2 E}{\partial \mathbf{w}^2} \Big|_{\mathbf{w}(t)} \right)^{-1} \frac{\partial E}{\partial \mathbf{w}} \Big|_{\mathbf{w}(t)}$$

- Hessian matrix difficult to compute → various approximations
  - Successive estimations
  - Diagonal terms only (quasi-Newton)
- Levenberg–Marquardt is another efficient algorithm

# Weight adjustment

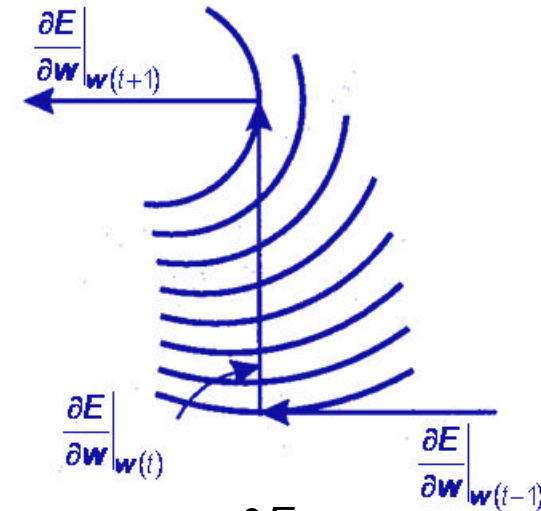
- What about the size of the step?
- Stochastic way: small  $\alpha$ 
  - but if too small: slow
  - and if too large: jumps over the minimum
- Other solution: line search  $\mathbf{w}(t+1) = \mathbf{w}(t) + \lambda(t)\mathbf{d}(t)$ 
  - go as far as possible in the chosen direction  $\mathbf{d}(t)$
  - Implicit use of the Hessian matrix
  - If  $d(t)$  is the gradient at each time step:  $\mathbf{d}(t) = \left. \frac{\partial E}{\partial \mathbf{w}} \right|_{\mathbf{w}(t)}$

$$\frac{\partial}{\partial \lambda} E(\mathbf{w}(t) + \lambda \mathbf{d}(t)) = 0 \quad \text{thus} \quad \left. \frac{\partial E}{\partial \mathbf{w}} \right|_{\mathbf{w}(t+1)} \mathbf{d}(t) = 0$$

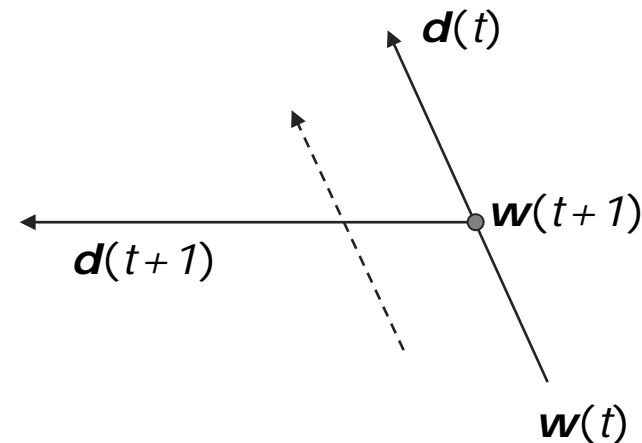
- So successive directions are always orthogonal!

# Weight adjustment

- Successive directions are orthogonal (line search)
  - Not good for speed of convergence



- Use *conjugate* gradients:  
New direction is chosen so that the component of the gradient  $\frac{\partial E}{\partial \mathbf{w}}$  parallel to the previous direction remains 0
  - Conjugate gradients make implicit use of the Hessian matrix



# Outline

- Motivation
- Single-layer nonlinear regression
- Multi-layer perceptron
  - Model
  - MLP with threshold units
  - Number of layers
  - Learning
    - Error back-propagation
    - Weight adjustment
- Applications

# Applications

- Some (old...) applications of MLP
- In fact some applications of nonlinear regression with machine learning! (not specific to MLP)
  1. A standard nonlinear model
  2. Several ways to use similar models for the same goal
  3. The interpolation-extrapolation question



## Application: function approximation

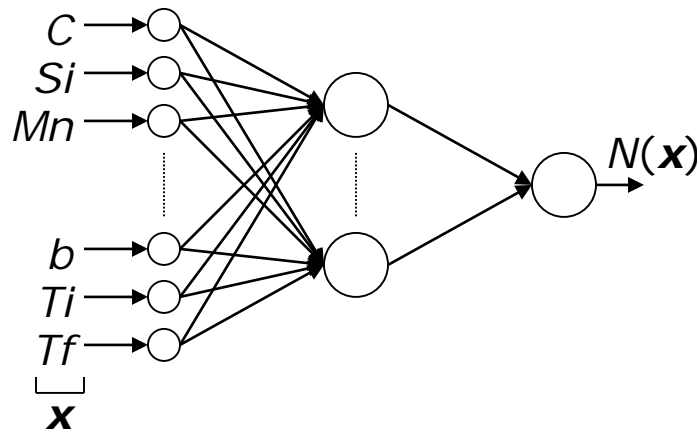
- Standard multivariate regression (function approximation)
- Application to process optimisation (estimation of physical properties, based on chemical composition and process parameters)

$$\alpha = f(C, Si, Mn, P, S, Al, N, Cu, Cr, Ni, Sn, V, Mo, Ti, Nb, B, d, b, Ti, Tf)$$

- “relative yield stress”  $\alpha$  of different steel qualities
- inputs:
  - 16 chemical additives
  - plate’s thickness  $d$  and width  $b$
  - temperatures before ( $T_i$ ) and after ( $T_f$ ) rolling

From “Estimating material properties for process optimization”, T. Poppe & T. Martinetz (Siemens AG), in ICANN’93 (Amsterdam, The Netherlands) proceedings, S. Gielen & B. Kappen eds., Springer-Verlag, 1993

# Application: function approximation

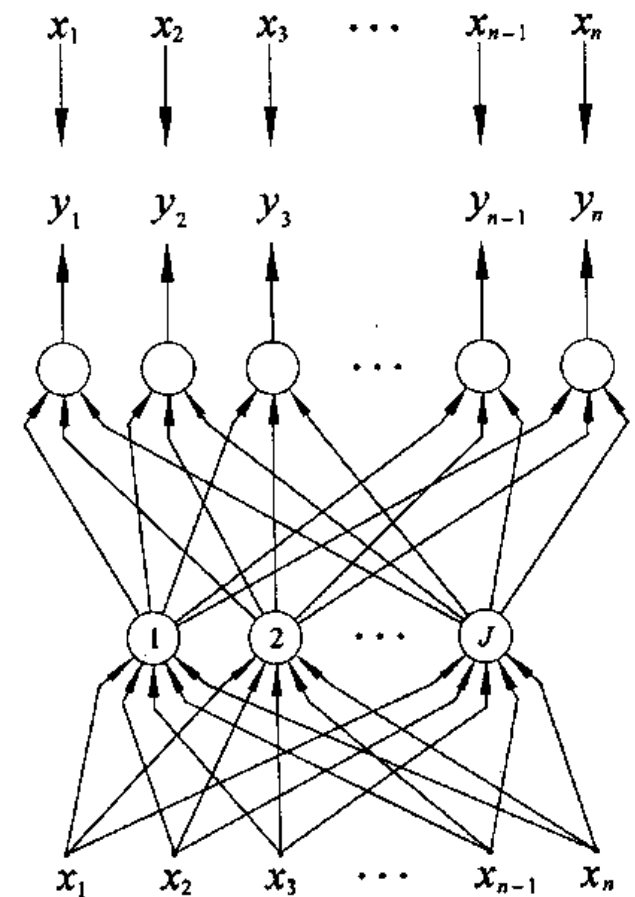


$$E = \sum_{p=1}^P E^p(\mathbf{x}) = \sum_{p=1}^P \left( \alpha^p - N(\mathbf{x}^p) \right)^2$$

- 1 hidden layer
  - 10 hidden units
  - training set: 9000 pairs
  - test set: 3000 pairs
  - on-line training
- 
- Results:
    - Physical characterisation: RMS = 53.6 %
    - MLP learning: RMS = 34.9 %

# Application: image compression 1

- lossy compression scheme (example)
- original image split into 8x8 pixel regions
- each 64-features vector sent in auto-associative MLP
- hidden layer: 16 neurons
- compression ration: 16/64



From: M. Hassoun, Fundamentals of artificial neural networks, MIT Press, 1995

# Application: image compression 1

- result on training image
  - (a) original
  - (b) compressed
  - (c) compressed + quantized (1.5 bit/pixel)
  - (d) compressed + quantized (1 bit/pixel)



(a)



(b)



(c)

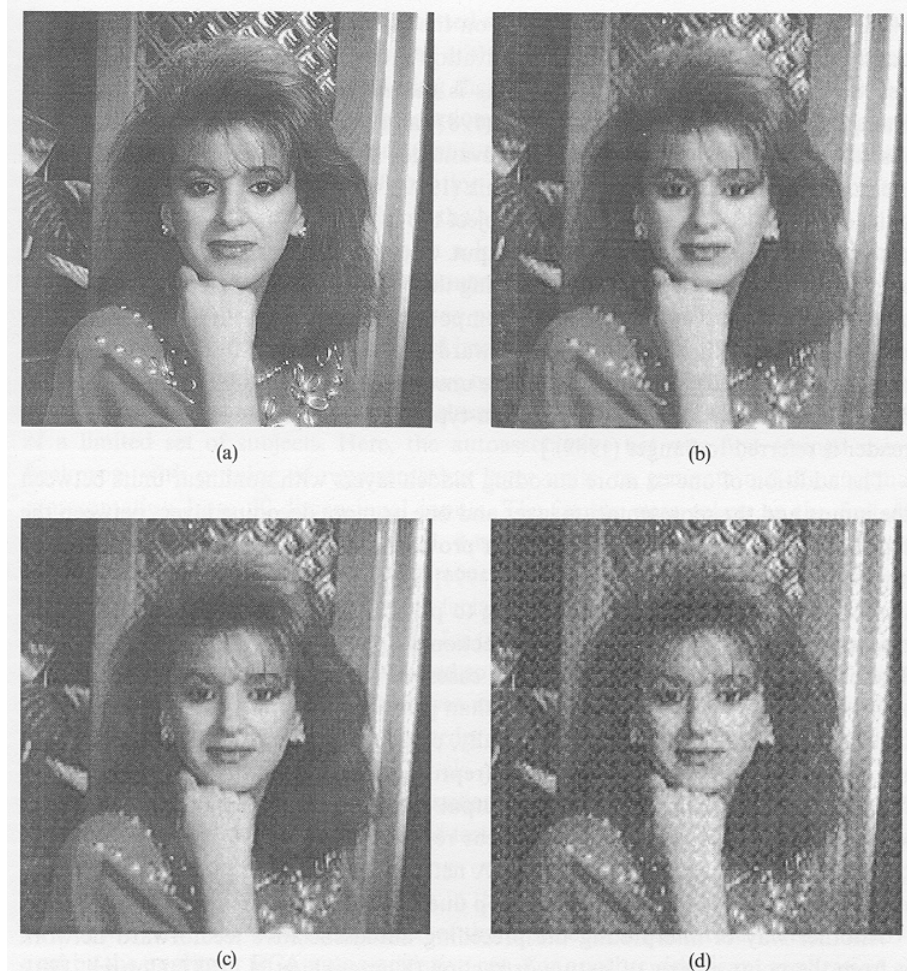


(d)

From: M. Hassoun, Fundamentals of artificial neural networks, MIT Press, 1995

# Application: image compression 1

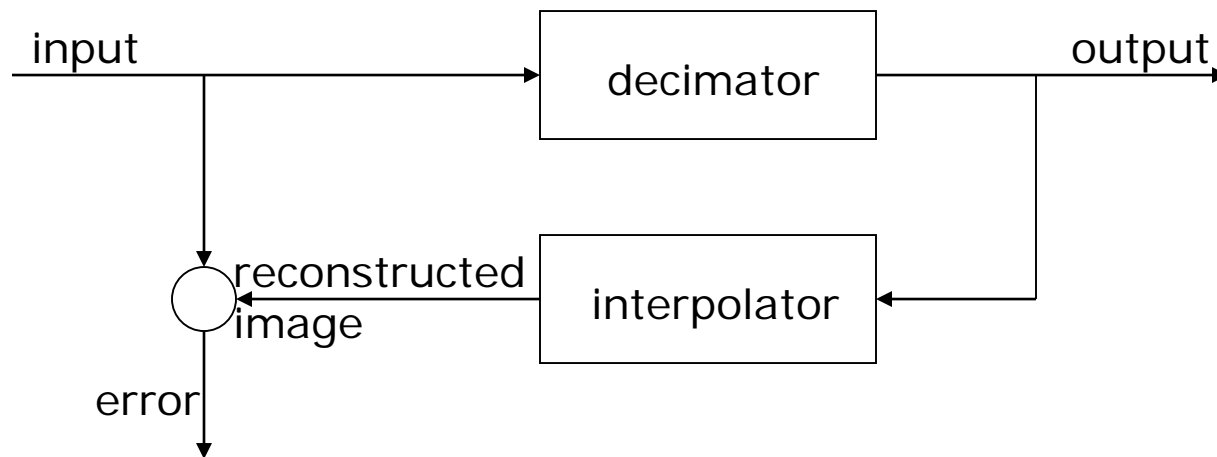
- result on testimage
  - (a) original
  - (b) compressed
  - (c) compressed + quantized (1.5 bit/pixel)
  - (d) compressed + quantized (1 bit/pixel)



From: M. Hassoun, Fundamentals of artificial neural networks, MIT Press, 1995

## Application: image compression 2

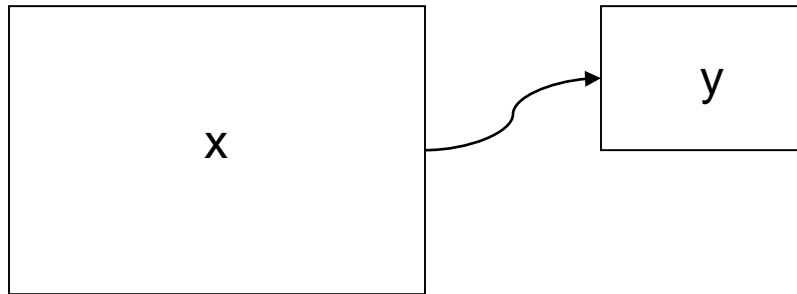
- lossless compression scheme (example)
- *Laplacian pyramids*



- If output and error are transmitted: lossless compression ! (interpolator function also needed of course)

## Application: image compression 2

- Decimator: (half-band filter) + downsampler  $\leftarrow$  anti-aliasing



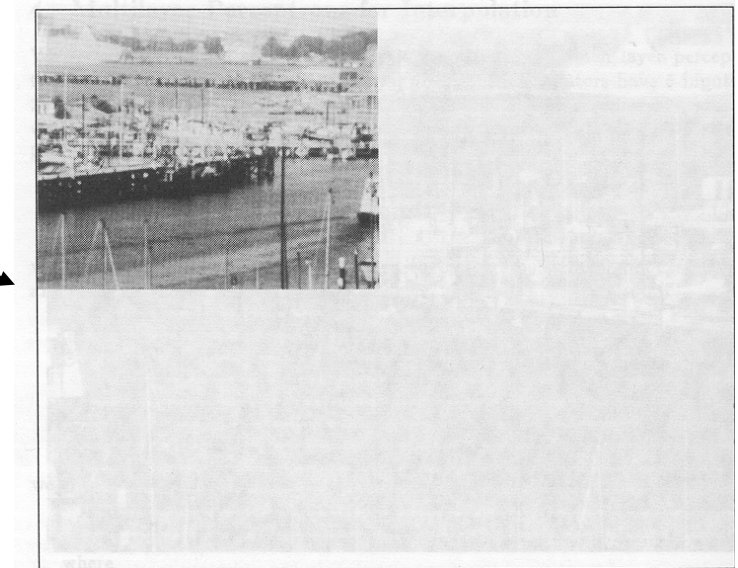
- Interpolator: upsampler + (half-band filter)  $\leftarrow$  smoothing

## Application: image compression 2

- Traditional Laplace pyramids: decimator and interpolator are linear
- MLP: interpolator is non-linear
  - better reconstruction
  - smaller error
  - better compression ratio for error with entropic coder
- entropic coder: better ratio if more zeroes



# Application: image compression 2



From "Laplacian pyramid with multilayer perceptrons interpolators", B. Simon, B. Macq, M. Verleysen, in ESANN'93 (Bruges, Belgium) proceedings, D-Facto publications, 1993

# Application: image compression 2

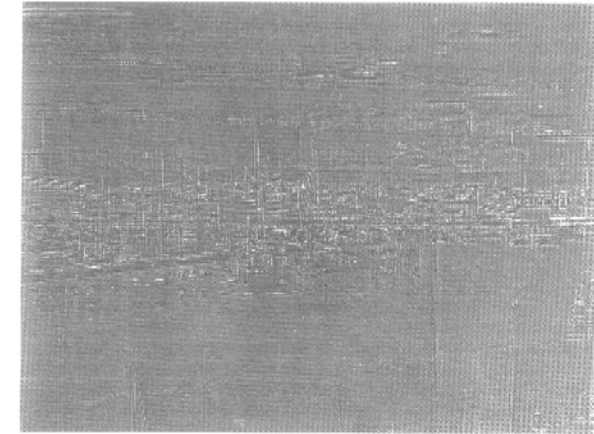
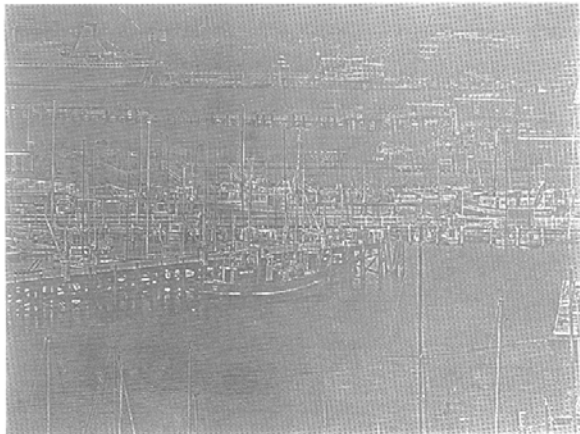
linear



MLP



reconstructed  
images

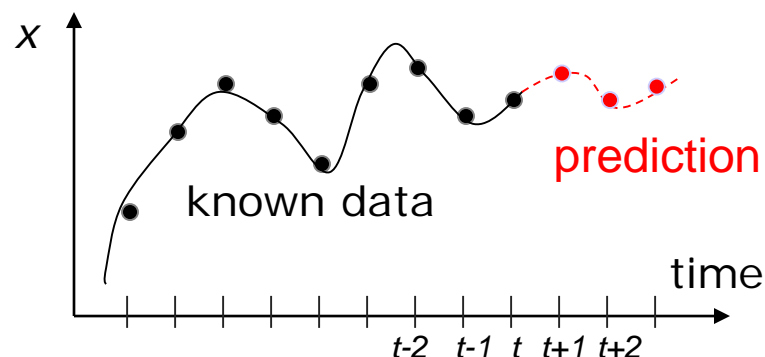


differences

From "Laplacian pyramid with multilayer perceptrons interpolators", B. Simon, B. Macq, M. Verleysen, in ESANN'93 (Bruges, Belgium) proceedings, D-Facto publications, 1993

# Application: time series forecasting

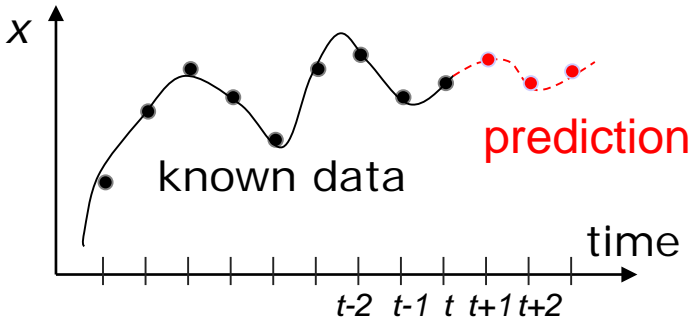
- Time series:



- Applications:
  - finance (stock market index, exchange rates, etc.)
  - electricity / gas consumption
  - etc.

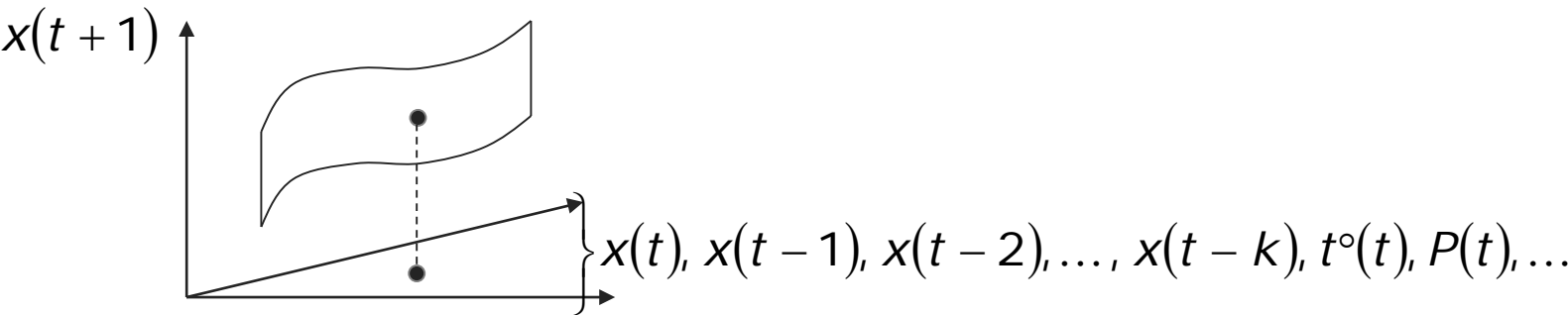
# Application: time series forecasting

- Time series forecasting = function approximation



$$x(t + 1) = f(\underbrace{x(t), x(t - 1), x(t - 2), \dots, x(t - k)}_{\text{past values}}, \underbrace{t^\circ(t), P(t), \dots}_{\text{inputs (exogenous variables)}})$$

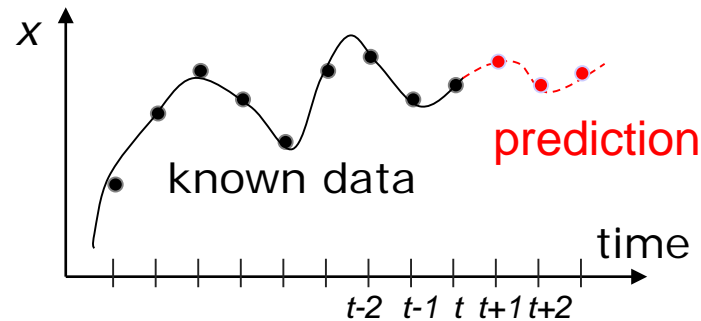
- New point in *inside* the surface



- So it is really interpolation, not extrapolation!

# Application: time series forecasting

- Long-term forecasting



- To predict  $x(t+2)$  we can use
  - the true value
  - the estimated value of  $x(t+1)$