Purpose:

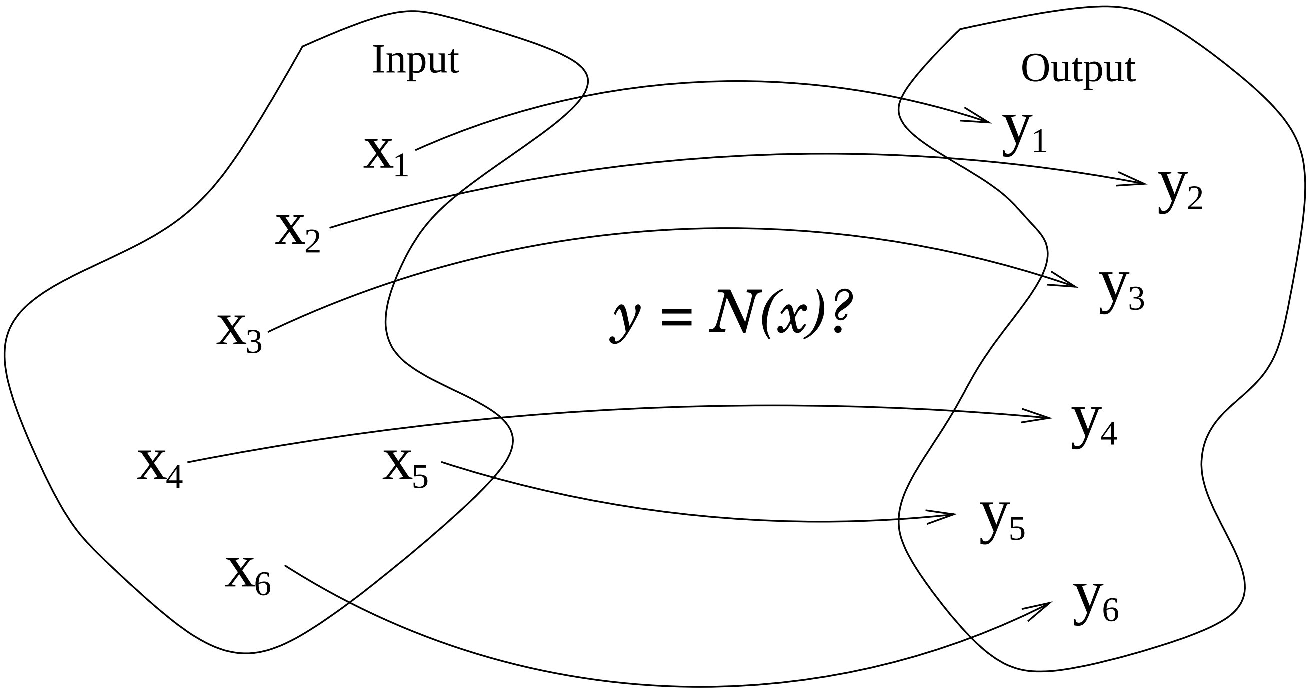
How to train an MLP neural network in MATLAB environment!

that is

For good computations,
we need good formulae
for good algorithms;
and good visualization
for good illustration
and proper testing
of good methods
and succesfull applications!

About Neural Networks

- supervised learning of NN: nonlinear regression approximation based on given input-output -vector pairs $\{\mathbf{x}_i, \mathbf{y}_i\}_{i=1}^N$, $\mathbf{x}_i \in \mathbf{R}^{n_0}$ and $\mathbf{y}_i \in \mathbf{R}^{n_2}$



- here, instead of *backpropagation*, i.e.

  $i)$ applying the chain-rule for sensitivity analysis

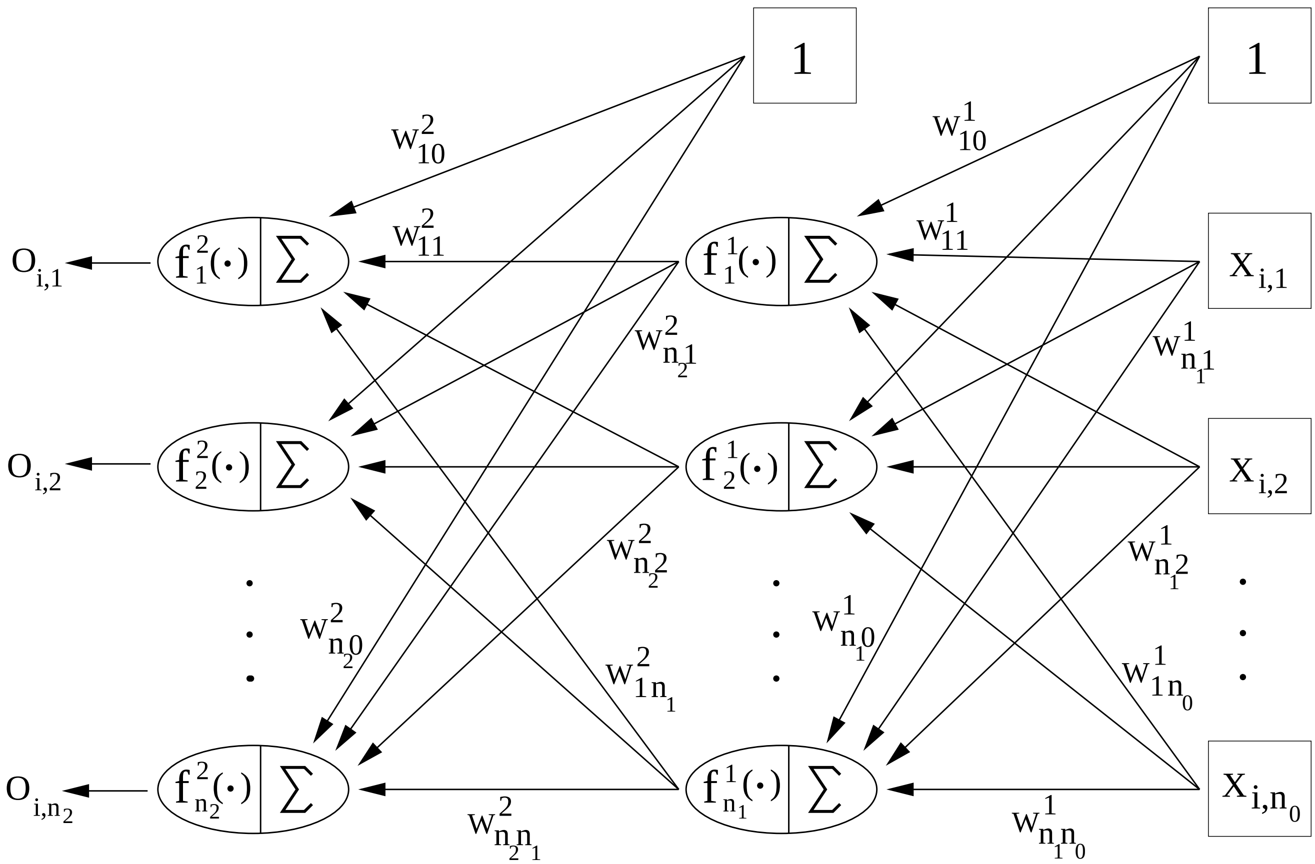
  $ii)$ applying basic gradient method -type training algorithm

- we utilize

  $i)$ layered-wise representation of network architecture and corresponding *calculus*

  $ii)$ more advanced optimization methods for training

What is MLP?



For activation:

1. **architecture:** how to present the network (compactly)
2. **Learning data:** given set of input-output -pairs
3. **Learning problem:** optimization problem for deriving unknown weights
4. **Training method:** a way to solve the optimization problem

Learning Data

- input-output vectors:

$$\mathbf{x}_i \simeq \mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_{n_0} \end{bmatrix} \in \mathbf{R}^{n_0}$$

  and

$$\mathbf{y}_i \simeq \mathbf{y} = \begin{bmatrix} y_1 \\ \vdots \\ y_{n_2} \end{bmatrix} \in \mathbf{R}^{n_2}$$

  for all $i = 1, \ldots, N$, where N is the number of given vectors N

- components $(\mathbf{x}_i)_j, j = 1, \ldots, n_0$, of input-vectors $\{\mathbf{x}_i\}$ are called *features* (cf. pattern recognition)

- Whole data can be stored - surprise, surprise - to the following matrices:

$$\mathbf{X} = \begin{bmatrix} \mathbf{x}_1^T \\ \vdots \\ \mathbf{x}_N^T \end{bmatrix} = \begin{bmatrix} (\mathbf{x}_1)_1 & \ldots & (\mathbf{x}_1)_{n_0} \\ \vdots & \ddots & \vdots \\ (\mathbf{x}_N)_1 & \ldots & (\mathbf{x}_N)_{n_0} \end{bmatrix} \in \mathbf{R}^{N \times n_0}$$

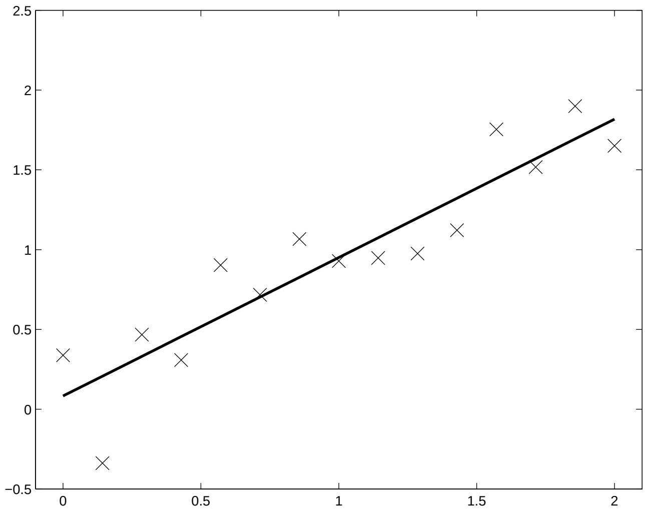
  and

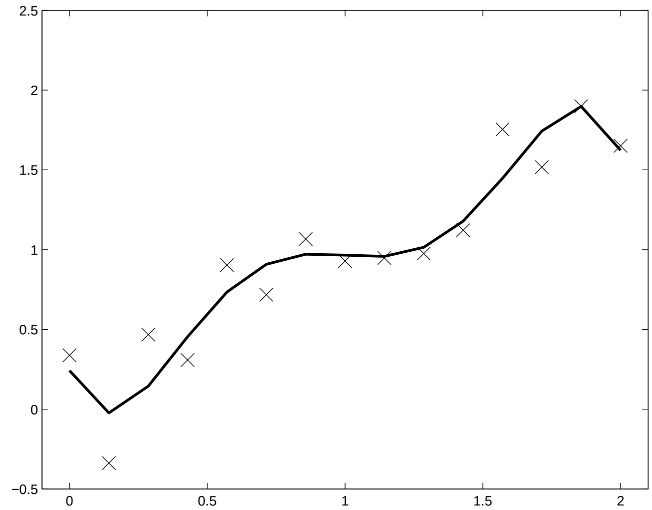
$$\mathbf{Y} = \begin{bmatrix} \mathbf{y}_1^T \\ \vdots \\ \mathbf{y}_N^T \end{bmatrix} = \begin{bmatrix} (\mathbf{y}_1)_1 & \ldots & (\mathbf{y}_1)_{n_2} \\ \vdots & \ddots & \vdots \\ (\mathbf{y}_N)_1 & \ldots & (\mathbf{y}_N)_{n_2} \end{bmatrix} \in \mathbf{R}^{N \times n_2}.$$

For succesfull application:

1. learning data representing stationary function and having suitable error distribution
2. proper architecture of MLP by means of complexity of unknown function

Example:





Economics: “Are we building the right product?”
“Are we building the product right?”

Software Engineering: “Are we building the right software?”
“Are we building the software right?”

Neural Networks: “Are we training the right network?”
“Are we training the network right?”

Optimization: “Are we minimizing the right functional?”
“Are we minimizing the functional right?”

Layerwise description of MLP-mapping:

1. from line to surface to hypersurface

$$a\left(\mathbf{x}\right) = w_0 + w_1 x_1 + \cdots + w_{n-1} x_{n-1} + w_n x_n = \mathbf{w}^T \hat{\mathbf{x}},$$

   where

$$\mathbf{w} = \begin{bmatrix} w_0 \\ w_1 \\ \vdots \\ w_n \end{bmatrix} \quad \text{and} \quad \hat{\mathbf{x}} = \begin{bmatrix} 1 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

   w_0 the so-called bias-term shifting the origin ($a\left(0\right) = w_0$)

2. Linear transformation (= linear perceptron):

$$\begin{bmatrix} w_{10}^1 + w_{11}^1 x_1 + \cdots + w_{1,n}^1 x_n \\ \vdots \\ w_{i,0}^1 + w_{i,1}^1 x_1 + \cdots + w_{i,n}^1 x_n \\ \vdots \\ w_{n_1,0}^1 + w_{n_1,1}^1 x_1 + \cdots + w_{n_1,n}^1 x_n \end{bmatrix} = \begin{bmatrix} \mathbf{w}_1^{1,T} \hat{\mathbf{x}} \\ \vdots \\ \mathbf{w}_i^{1,T} \hat{\mathbf{x}} \\ \vdots \\ \mathbf{w}_m^{1,T} \hat{\mathbf{x}} \end{bmatrix} = \mathbf{W}^1 \hat{\mathbf{x}}$$

3. Nonlinear activation with diagonal function-matrix:

$$\mathcal{F}(\cdot) = \begin{bmatrix} f_1(\cdot) & 0 & \ldots & 0 \\ 0 & f_2(\cdot) & \ldots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & \ldots & 0 & f_m(\cdot) \end{bmatrix}$$

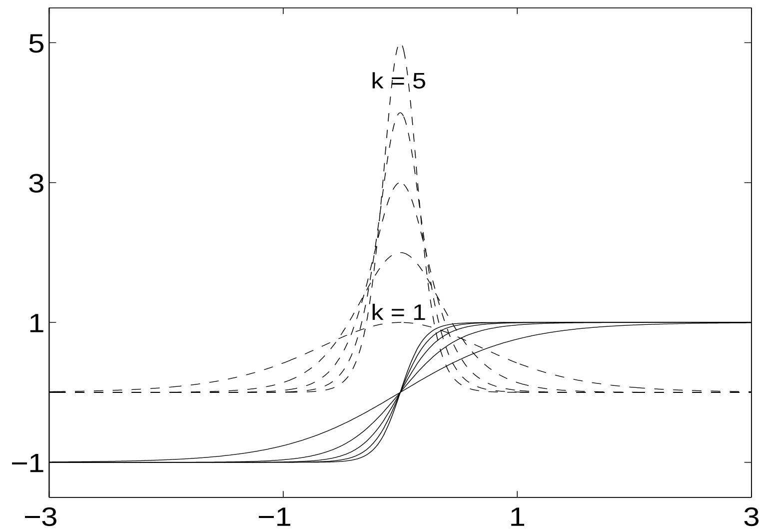
   $\Rightarrow o^1 = \mathcal{F}(\mathbf{W}^1 \hat{\mathbf{x}})$.

   Note: MLP could be generalized by using nondiagonal $\mathcal{F}$

4. One layer not enough for proper nonlinearity. To quarantee (simplified) *nonlinear* action we introduce second layer:

$$\mathcal{N}(\mathbf{x}) = \mathbf{W}^2 \widehat{\mathbf{o}}^1 = \mathbf{W}^2 \widehat{\mathcal{F}}(\mathbf{W}^1 \hat{\mathbf{x}})$$

About activation functions:



- originally step-function was used, but its *nondifferentiability* prevents efficient training
- mathematical properties *non-polynomiality* and *monotonicity* (i.e., squashing function)
- most popular choices

$$s(a) = \frac{1}{1+\exp(-a)} \quad \text{logistic sigmoid}$$

$$s_k(a) = \frac{1}{1+\exp(-k\,a)}, \quad k = 1, 2, \ldots \quad \text{'k-sig'}$$

$$t_k(a) = \frac{\exp(k\,a) - \exp(-k\,a)}{\exp(k\,a) + \exp(-k\,a)} = \frac{2}{1+\exp(-2\,k\,a)} - 1 = 2\,s_k(2\,a) - 1 \quad \text{'k-tanh'}$$

for which $t_k(-a) = -t_k(a)\ \forall a \geq 0$.

- …with derivatives

$$s_k^{'}(a) = k\ \exp(k\,a)/(1+\exp(k\,a))^2 = k\,s_k(a)\,(1 - s_k(a))$$

$$t_k^{'}(a) = 4\,k\ \exp(2\,k\,a)/(1+\exp(2\,k\,a))^2 = k\,(1+t_k(a))\,(1-t_k(a)) = k\,(1-t_k(a)^2)$$

About approximation capability:

Theorem 1. Let $\varphi(\cdot)$ be a nonconstant, bounded, and monotone-increasing continuous function. Let I_m denote the m_0-dimensional unit hypercube $[0,1]^{m_0}$. The space of continuous functions on I_{m_0} is denoted by $C(I_{m_0})$. Then, given any function $f \ni C(I_{m_0})$ and $\varepsilon > 0$, there exists an integer m_1 and sets of real constants α_i, β_i, and w_{ij}, where $i = 1, \ldots, m_1$ and $j = 1, \ldots, m_0$ such that we may define

$$F(x_1, \ldots, x_{m_0}) = \sum_{i=1}^{m_1} \alpha_i \, \varphi(\sum_{j=1}^{m_0} w_{ij} x_j + b_i)$$

as an approximate realization of function $f(\cdot)$; that is,

$$|F(x_1, \ldots, x_{m_0}) - f(x_1, \ldots, x_{m_0})| < \varepsilon$$

for all $x_1, \ldots, x_{m_0}$ that lie in the input space.

- merely about existence, nothing about how to fix unknown coefficients

MLP-transformation using MatLab:

```
%
n0 = 3; n1 = 4; n2 = 2; k = 1;
x = zeros(1,n0); w1 = zeros(n1,n0+1); w2 = zeros(n2,n1+1);
% ...
o1 = w1*[1; x'];
o1 = 2./(1 + exp(-2*k*o1)) - 1;
o = w2*[1; o1];
%
% Isn't it simple?!
%
```