Recap: Data-based modelling

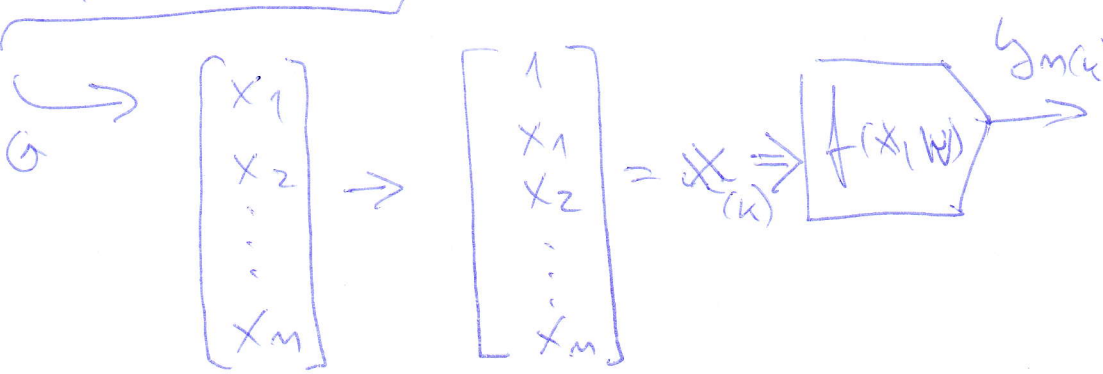
$\vec{x}(k)$ ... input vector at time k

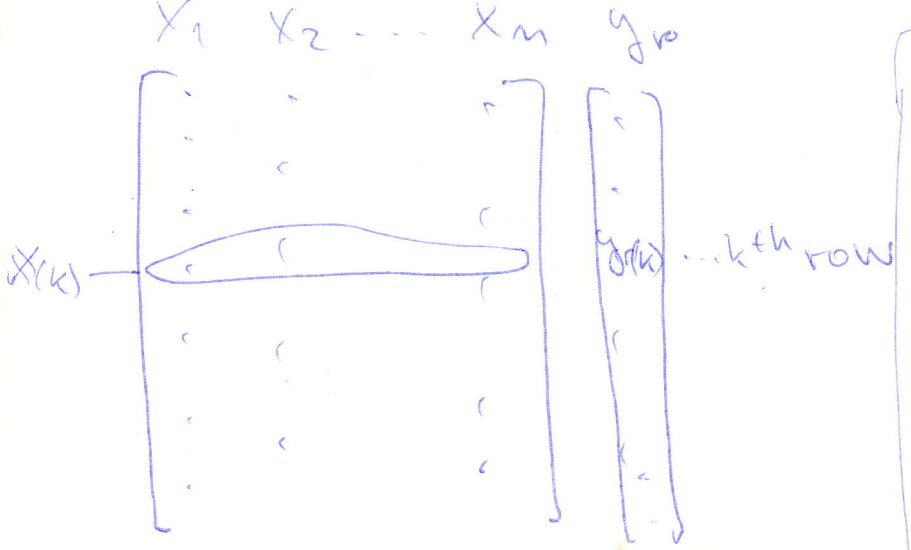
$y_m(k)$ ... model output at time k

$y_r(k)$ ... real value at time k (the "Target")

Measured data are in a table:

k	x_1	x_2	...	x_m	y_r
0	#	#		#	
1	#	#		#	
...					
	$x_1(k)$	$x_2(k)$		$x_m(k)$	$y_r(k)$
...					
N					





X — Matrix of all input data

$\vec{y}_r$ — vector of all targets

By now, we learned gradient sample-by-sample adaptation; i.e.:

1) Set random weights
2) for every row: (one-by-one)
   - calculate model output
   - calculate error
   - update weights
   - at the end of table start again (another epoch)

Summary on gradient descent (that we learned)

a) We learned G.D. only for static models

b) The principle is applicable to dynamic models (RTRL ... real time recurrent learning)

c) The GD we learned, is a basis for most powerful gradient based learning (optimization), i.e. for BPTT ... BackPropagation Through Time

d) BPTT is the most efficient gradient-based training
- when we do not want to use much more complicated and computationally demanding algorithms
(- when we do not have too large model) (too high-level network)

e) BPTT can be implemented as combination of GD and Levenberg-Marquardt algorithm (L-M)

Linear model

$$y_m(k) = w_0 + w_1 x_1 + w_2 x_2 =$$

$$= [w_0 \; w_1 \; w_2] * \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} =$$

$$= W * X(k)$$

Update:

$$\Delta W = \mu \cdot e(k) * X(k) =$$

$$= [\Delta w_0 \; \Delta w_1 \; \Delta w_2]^T$$

$W = W + \Delta W$ ... all weights at once

Polynomial model

$$y_m(k) = w_{00} + w_{01} x_1 + w_{02} x_2 +$$

$$+ w_{11} x_1^2 + w_{12} x_1 x_2 + w_{22} x_2^2 =$$

$$= [w_{00} \; w_{01} \; w_{02} \; w_{11} \; w_{12} \; w_{22}]$$

$$* \begin{bmatrix} x_0^2 \\ x_0 x_1 \\ x_0 x_2 \\ x_1 x_1 \\ x_1 x_2 \\ x_2^2 \end{bmatrix} = W * X(k)$$

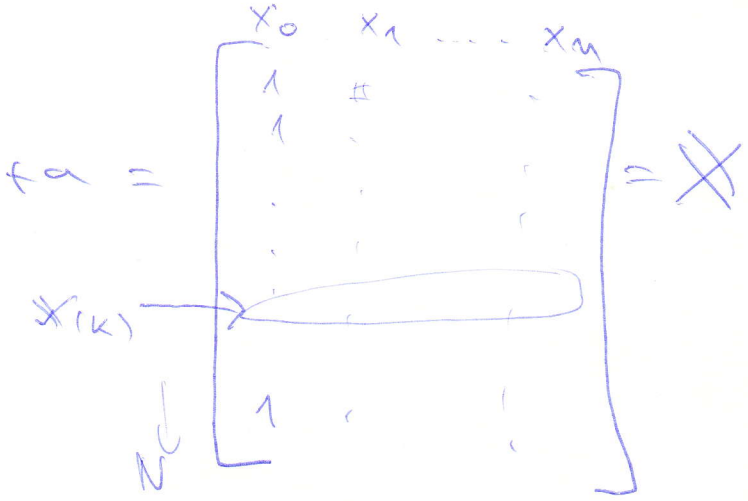
where $x_0 = 1$

Update: $\Delta W = \mu \cdot e(k) \cdot X(k)$

Levenberg - Marquardt algorithm (batch training)

- Gradient Descent adaptation calculated weight updates in sample-by-sample manner
- L-M algorithm calculate weight updates while considering all data at once (batch training)

L-M algorithm:

X data = 

X ... matrix of all input data

y_r ... vector of all targets (the same table of data as for GD)

$$y_r = \begin{bmatrix} y_r(k=0) \\ y_r(k=1) \\ \vdots \\ \end{bmatrix} \quad N\times$$

L-M formula:

for a single weight:

$$\Delta w_{ij} = (J_i * J_i + \frac{1}{\mu})^{-1} J_i^T * e$$

e ... column vector of all errors

J_i ... ith column of Jacobian

for multiple weights:

$$[\Delta w_i \dots \Delta w_j] = (J_{i\dots j}^T * J_{ij} + \frac{1}{\mu} * \mathbb{1})^{-1} J_{i\dots j}^T * e$$

$$J_{i\dots j} = \begin{bmatrix} \frac{\partial y_m(1)}{\partial w_i} & \dots & \frac{\partial y_m(1)}{\partial w_j} \\ \vdots & & \\ \frac{\partial y_m(N)}{\partial w_i} & \dots & \frac{\partial y_m(N)}{\partial w_j} \end{bmatrix} \Leftrightarrow i\dots j \text{ columns of Jacobian}$$

Jacobians for L-M

Linear model

$$y_n = w_0 + w_1 x_1 + w_2 x_2$$

$$J = \begin{bmatrix} 1 & x_1(1) & x_2(1) \\ 1 & x_1(2) & x_2(2) \\ \vdots & \vdots & \vdots \\ 1 & x_1(N) & x_2(N) \end{bmatrix}$$

Polynomial model

$$y_n = w_{00} + w_{01} x_1 + w_{02} x_2 + w_{11} x_1^2 + w_{12} x_1 x_2 + w_{22} x_2^2 =$$

$$= [w_{00}\ w_{01}\ w_{02}\ w_{11}\ w_{12}\ w_{22}] * \begin{bmatrix} 1 \\ x_1 \\ x_2 \\ x_1^2 \\ x_1 x_2 \\ x_2^2 \end{bmatrix}$$

$$J = \begin{bmatrix} 1 & x_1(1) & x_2 & x_1^2 & x_1 x_2 & x_2^2 \\ 1 & \vdots & \vdots & & & \\ \vdots & \vdots & \vdots & & & \\ 1 & x_1(N) & x_2(N) & \dots & & \end{bmatrix}$$

For a general model:

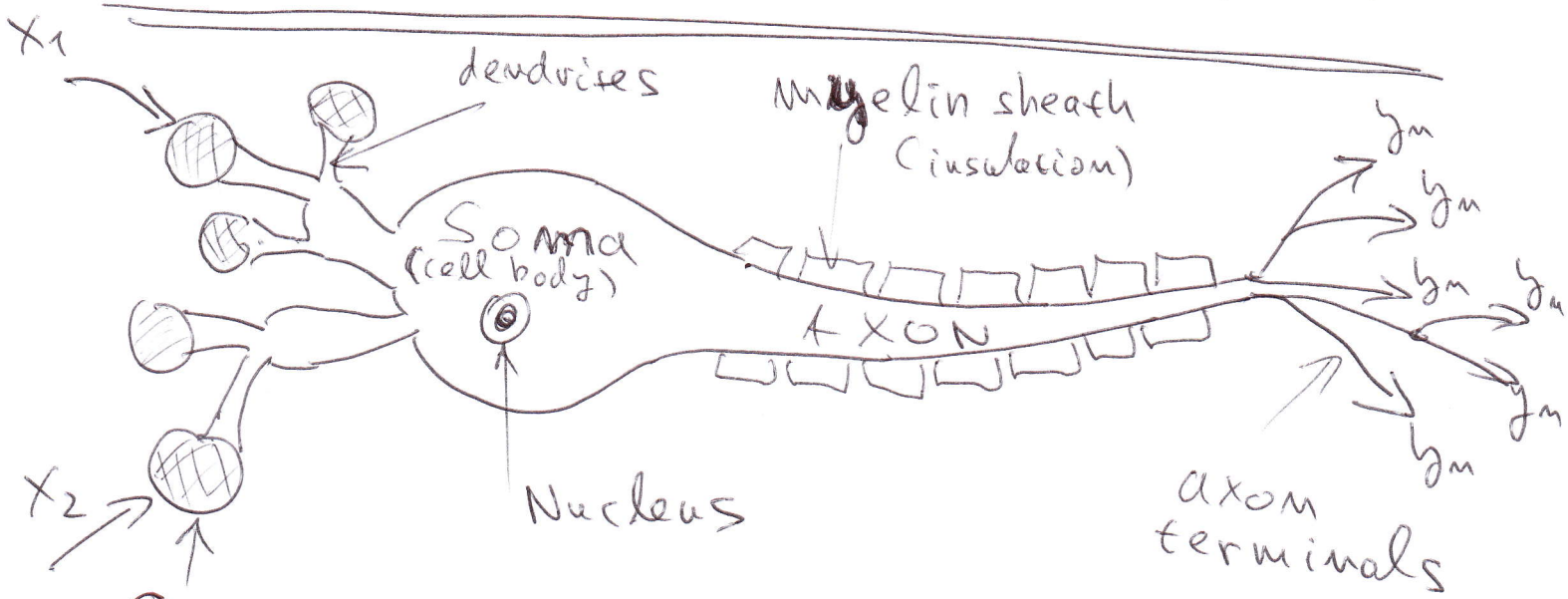
$$y_n = f(x, w) \longrightarrow \text{weights}$$

$$J = k \begin{matrix} 1 \\ \downarrow \\ N \end{matrix} \begin{bmatrix} \frac{\partial f(k=1)}{\partial w_0} & \frac{\partial f(k=1)}{\partial w_1} & \dots & \frac{\partial f(k=1)}{\partial w_n} \\ \frac{\partial f(k=2)}{\partial w_0} & \frac{\partial f(k=2)}{\partial w_1} & \ddots & \vdots \\ \vdots & \vdots & & \\ \frac{\partial f(k=N)}{\partial w_0} & \frac{\partial f(k=N)}{\partial w_1} & \dots & \frac{\partial f(k=N)}{\partial w_n} \end{bmatrix}$$

Artificial Neuron (Perceptron)

Static (Feed-Forward) Neural Network

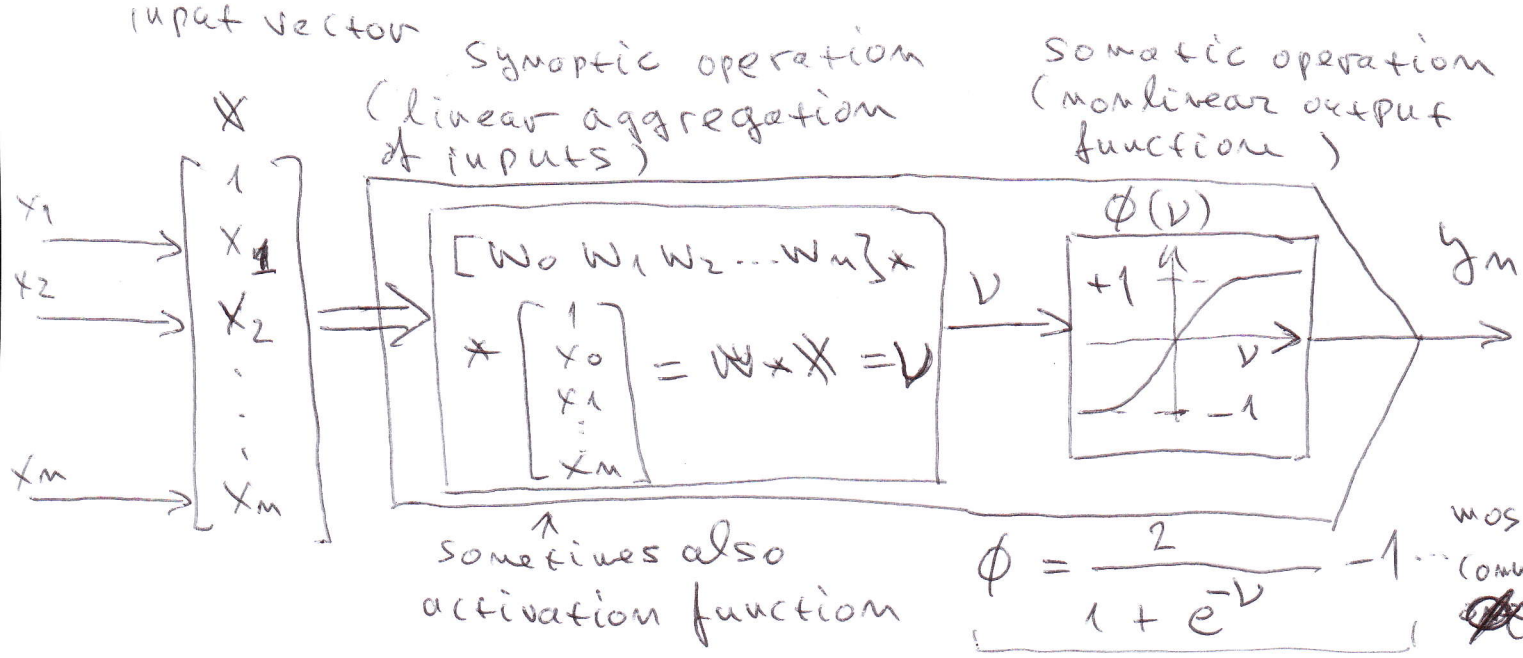
Linear model x polynomial x perceptron



x_1, x_2 ... signals from other neurons (or receptors, ...)

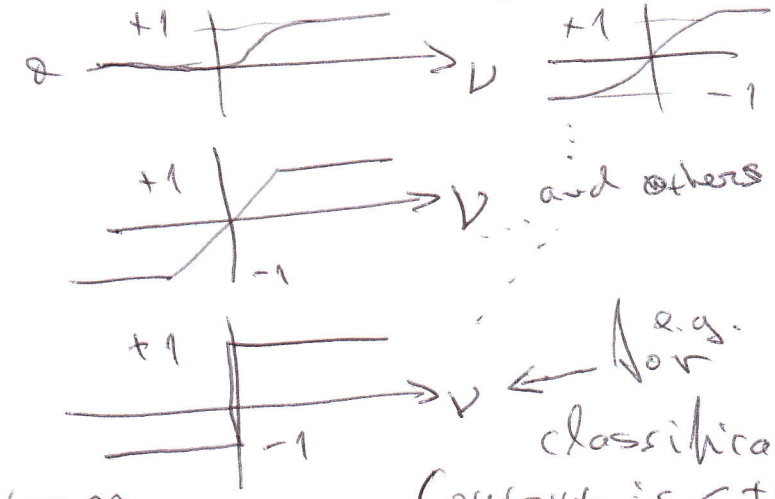
A mathematical model of a neuron has been widely adopted from fundamental works of McCulloch & Pitts 1940', Rosenblatt 1950', Widrow & Hoff 1960':

Perceptron type neuron:



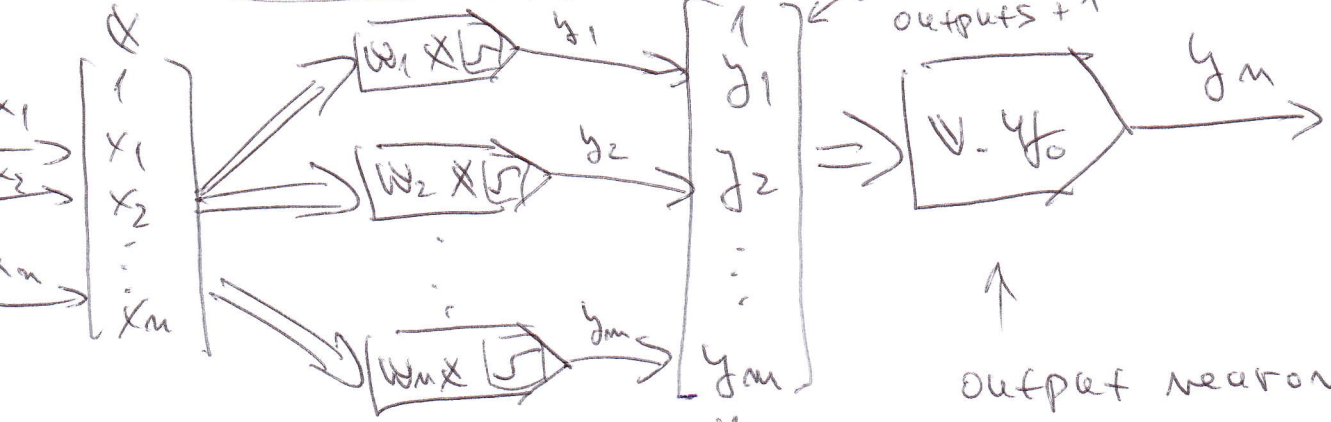
w_0 ... bias (or treshold)

$\phi(v)$ ... somatic function, can be many types



The above perceptron neuron is a basic element of perceptron neural networks

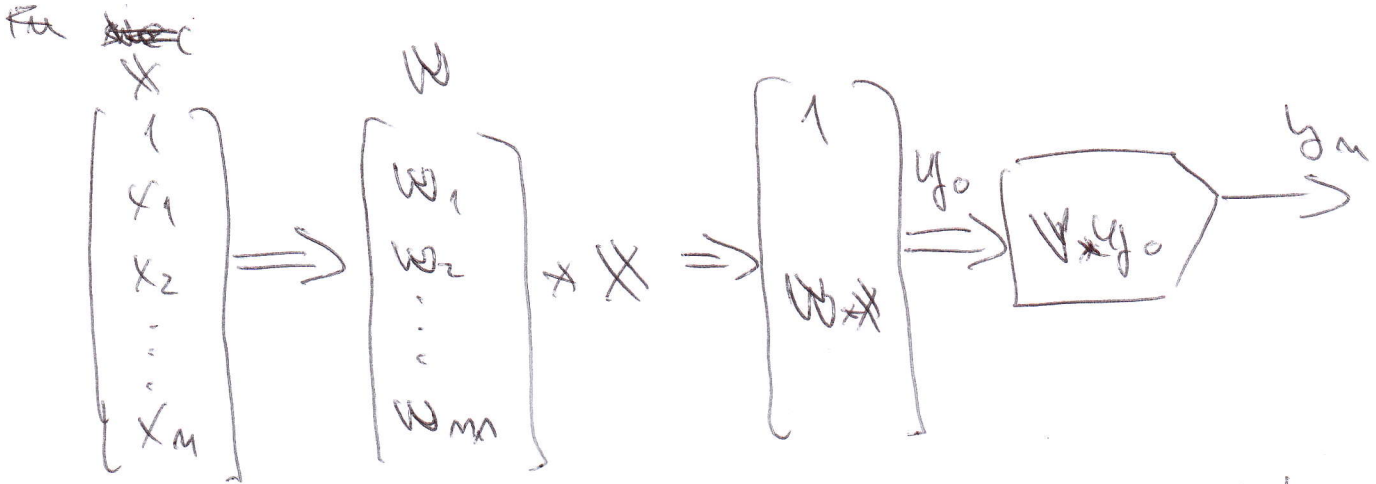
MLP ... multi-layer perceptron Neural Network



1st layer of m hidden neurons

output neuron usually without $\phi()$ (only linear aggregation)

$V = [v_0\ v_1\ v_2 \dots v_m]$



$$W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_m \end{bmatrix} = \begin{bmatrix} w_{10} & w_{11} & \dots & w_{1m} \\ w_{20} & w_{21} & \dots & w_{2m} \\ \vdots & & & \\ w_{m0} & w_{m1} & \dots & w_{mm} \end{bmatrix} \begin{matrix} \leftarrow \text{weights of neuron 1} \\ \leftarrow \text{weights of neuron 2} \\ \\ \leftarrow \text{weights of neuron } m \end{matrix}$$

The above is a Multilayer Perceptron (MLP) Neural Network (its simplest case with a single hidden layer of neurons with sigmoidal output (somatic) function)

The complete formula is:

$$y_m = V * \phi(W * X)$$

Network output

- V: weight vector of output neuron
- ϕ: somatic operation of first layer neurons
- W: weight matrix of first layer (rows correspond to neurons)
- X: input vector augmented with "1"

in pseudo code:

Synaptic operation: $nu = W * X$ ... matrix multiplication

Somatic operation: $phi = \frac{2}{1+e^{-nu}} - 1$

network output: $y_m = V * NU$ ... vector multiplication

sum(V*nu)

multiply(V, nu)

Linear x Polynomial x MLP network

$$y_{lin} = w_0 + w_1 x_1 + w_2 x_2 + \dots + w_m x_m$$

$$y_{pol} = w_{00} + w_{01} x_1 + w_{02} x_2 + w_{11} x_1^2 + w_{12} x_1 x_2 + \dots$$

$$y_{MLP} = V_0 + V_1 * \phi(W_1 X) + V_2 * \phi(W_2 X) + \dots$$

(W_1: 1st neuron, W_2: 2nd neuron)

Training i.e., optimization means to substitute input and output data with measured values.

Which of y_{lin}, y_{pol}, y_{MLP} has linear optimization problem?

1. 2.

Case study: For a given table of data, implement and train and test a model.

1) Preprocess data
2) Design a model (start with linear)
3) Derive and program learning rule (Gradient Descent, or Levenberg-Marquardt)
4) After training, test how your model works for values that are not in the original training table
5) Before trying more complicated model, consider if you can improve the training data. (6) Do not use it for real medicine dosage)

Data: (Panadol Baby)

body weight [kg]	age [years]	dosage [ml]
6-8	0.3 - 0.5	4.0
8-10	0.5 - 1	5.0
10-13	1 - 2	7.0
13-15	2 - 3	9.0
15-21	3 - 6	10.0
21-29	6 - 9	14.0
29-42	9 - 12	20.0

original data are given as intervals (good for Fuzzy logic or RBF networks)

data preprocessing for math. model

Preprocessed data (one option)

x_1	x_2	y_m
7	0.4	4
9	0.75	5
11.5	1.5	7
14	2.5	9
18	4.5	10
25	7.5	14
35.5	10.5	20

Linear model:

$$y_m = [w_0 \; w_1 \; w_2] * \begin{bmatrix} 1 \\ x_1 \\ x_2 \end{bmatrix} =$$

$$= w_0 + w_1 x_1 + w_2 x_2$$

a) by Gradient Descent:

k	x_1	x_2	y_r
0	7	0.4	4
1	9	0.75	5
⋮	⋮	⋮	⋮
6	35.5	10.5	20

update at $k=0$:

$X = (1, x_1(0), x_2(0))$ # build X at every k

$X = \text{array}(X)$ # X has to be array

$y_m = \text{sum}(w * X)$ # vector multiplication if w and X are "arrays"

$e = y_r - y_m$

$w = w + \mu * e * X$

$k=1$:

⋮

b) by Levenberg-Marquardt (batch training)

- weight update is calculated considering all data at once (one epoch ≡ one update)

$$J = \begin{bmatrix} 1 & x_1(1) & x_2(1) \\ 1 & x_1(2) & x_2(2) \\ \vdots & \vdots & \vdots \\ 1 & x_1(7) & x_2(7) \end{bmatrix} = \left[\begin{bmatrix} 1 \\ 1 \\ \vdots \\ 1 \end{bmatrix} X \right]$$ ... Jacobian for the linear model

x_1	x_2	y_r
7	0.4	4
9	0.75	5
11.5	1.5	7
⋮	⋮	⋮
35.5	10.5	20

y_r

$e = y_r - y_m = y_r - w * X$ ... errors for all k

$$\Delta W = \begin{bmatrix} \Delta w_0 \\ \Delta w_1 \\ \Delta w_2 \end{bmatrix} = (J^T * J + \frac{1}{\mu} \cdot \mathbb{1})^{-1} * J^T * e \qquad \mathbb{1} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$W = W + \Delta W^T$$