Perceptrons Vol. 2

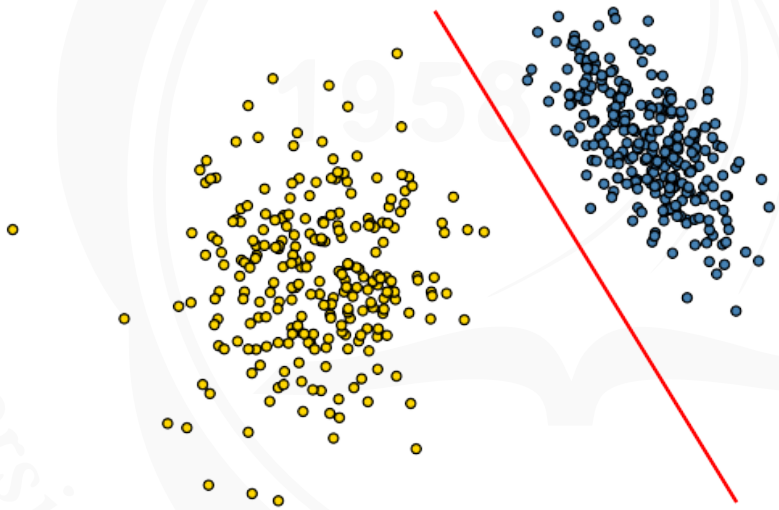
Meet your first neural network

Devert Alexandre
School of Software Engineering of USTC

Table of Contents

Linear classification

Iterating the *Delta rule* gives separating hyperplanes



Delta rule

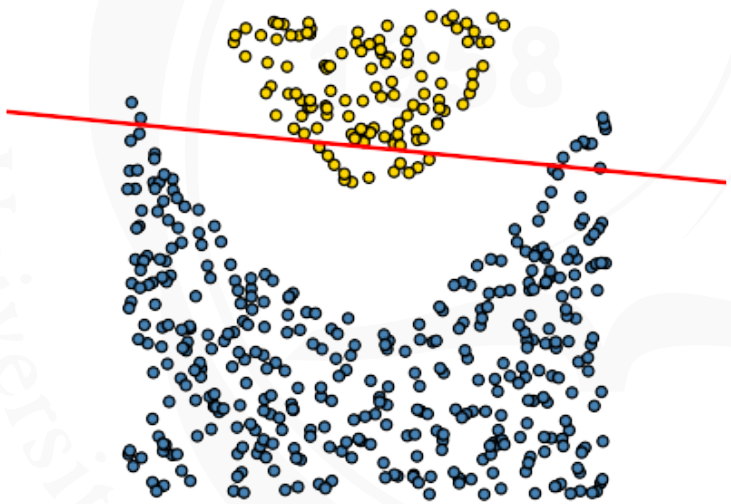
The *Delta rule* update the parameters of an hyperplane in $\mathbb{R}^n$

$$\Delta v_i = \eta(z - t)x_i f' \left(\sum_{i=0}^{n} v_i x_i \right)$$

- v_i are the parameters of the hyperplane
- Δv_i is the change to add to v_i
- x_i are the coordinate of an example point X
- t is the class of the point X
- z is the class predicted by the hyperplane
- f is the *transfer function*
- η is the learning rate

Linear separability

A linear separator does not separate well data with a non-linear frontier

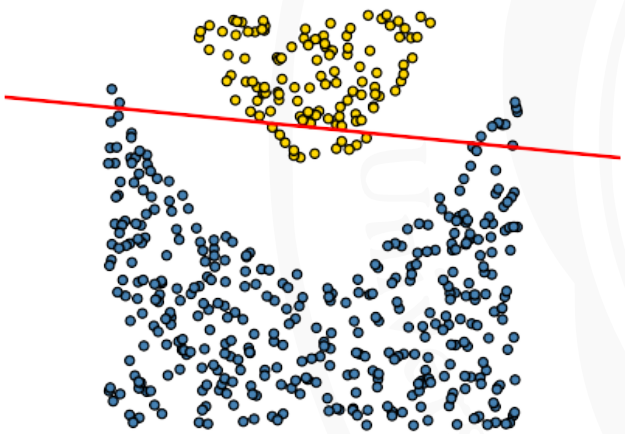


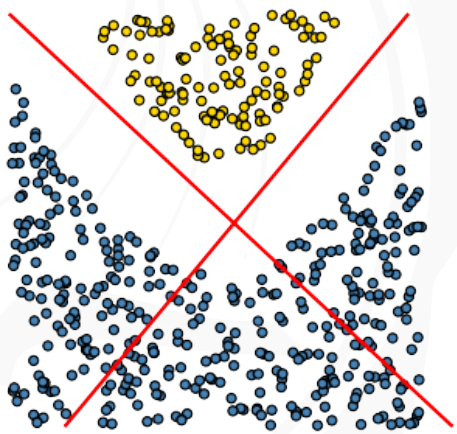
How to separate this ? Incredible suspense !

Table of Contents

Multi-linear classification

What about using 2, 3, . . . , many hyperplanes ?

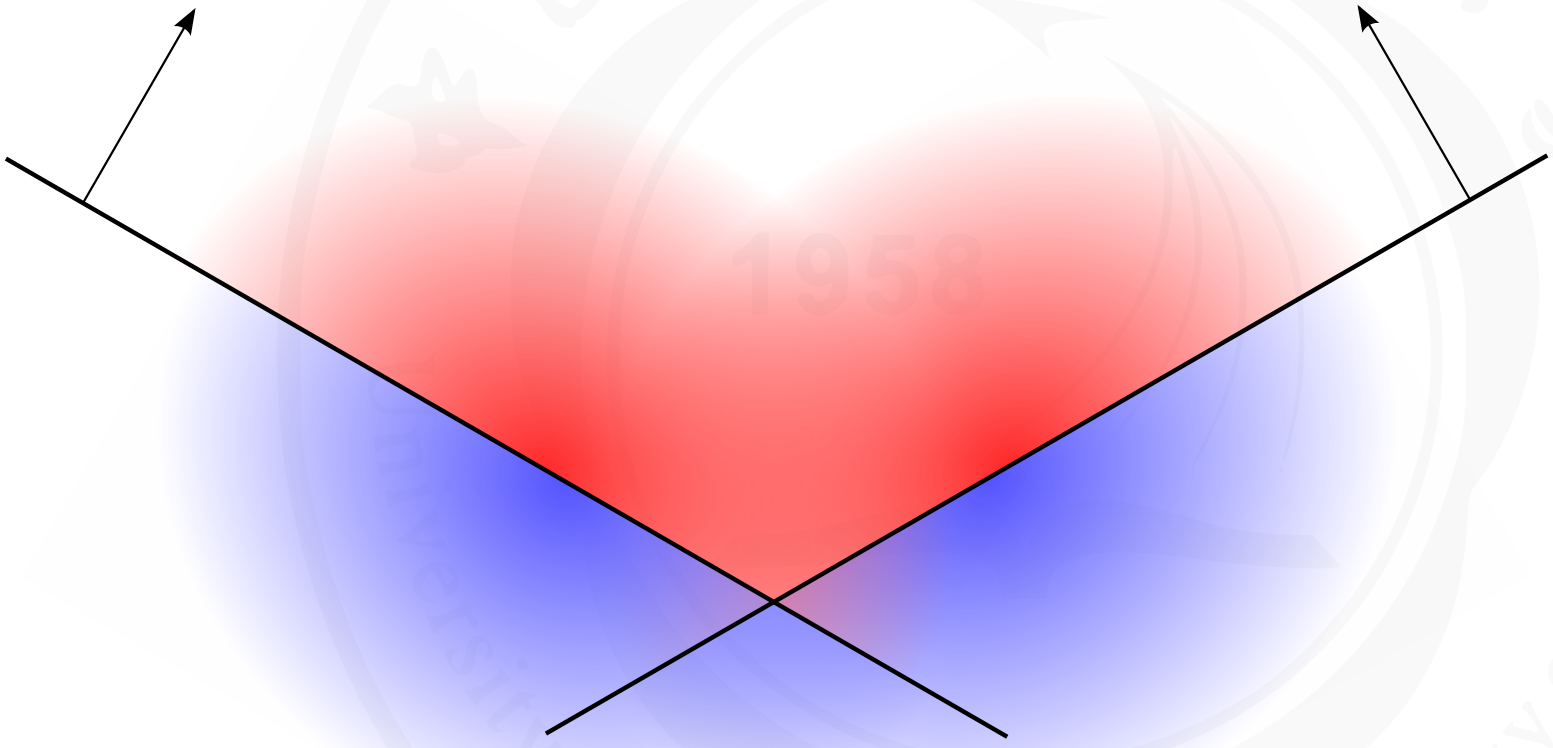




Here, 2 hyperplanes can be combined to separate the 2 classes

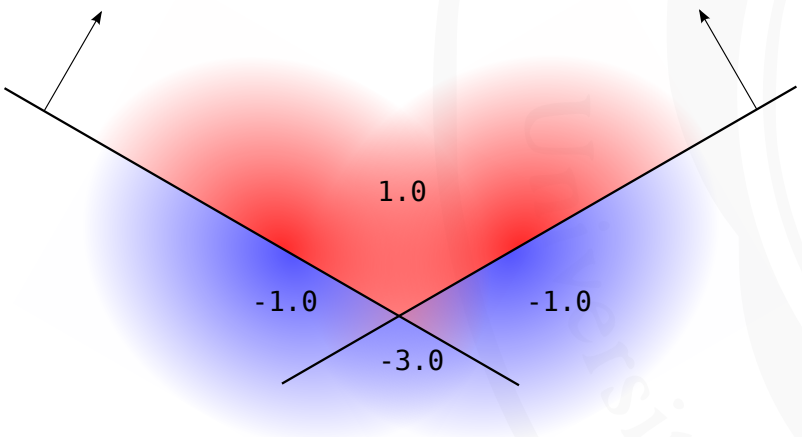
Multi-linear classification

Combining 2 hyperplanes to classify data



Multi-linear classification

Hyperplane A and B, with outputs z_A and z_B. Let's compute $z_{A \cap B} = tanh(z_A + z_B - 1)$



Sign of for a point X

- if X is in positive side of A *and* B, $z_{A \cap B} > 0$
- if X is in negative side of A *or* B, $z_{A \cap B} < 0$

Multi-linear classification

We can build a multi-linear classifier by

- take h hyperplanes as decision boundaries
- do an *affine* combination of the h hyperplanes decisions

The h hyperplanes and an *affine* combination of their decision can be built by $h + 1$ neurons

Multi-linear classification

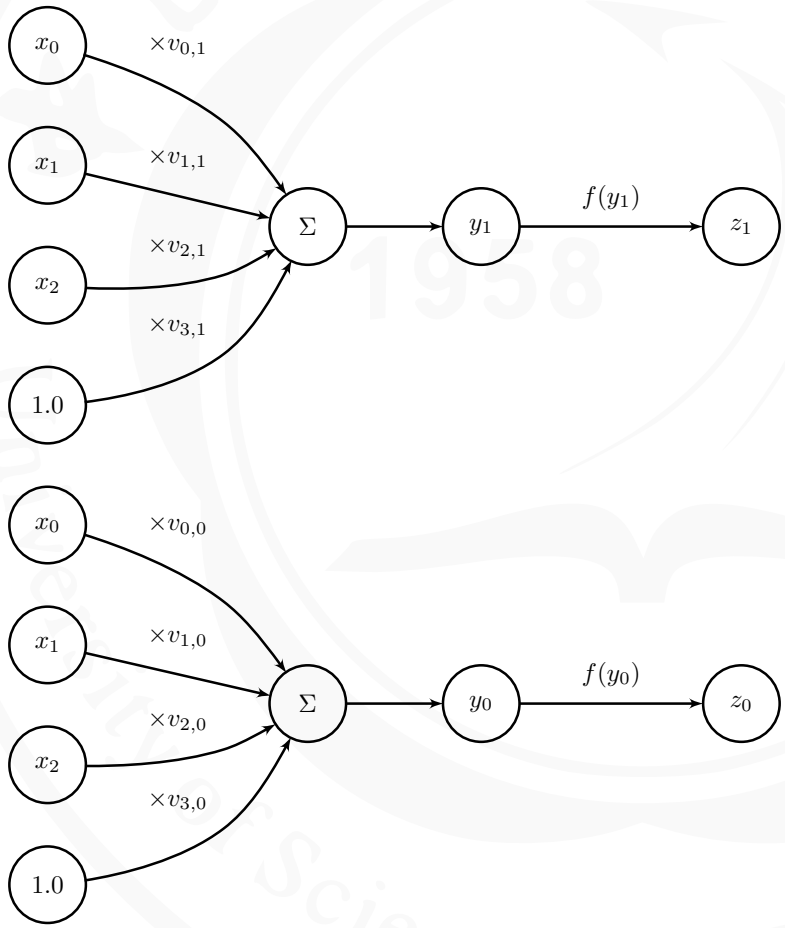
A multi-linear classifier can classify any datasets !

- Any decision boundary can be approximated by several hyperplanes
- It's called a *piece-wise linear* approximation
- By using more hyperplanes, we can approximate as close as we wish

Table of Contents

Combining neurons

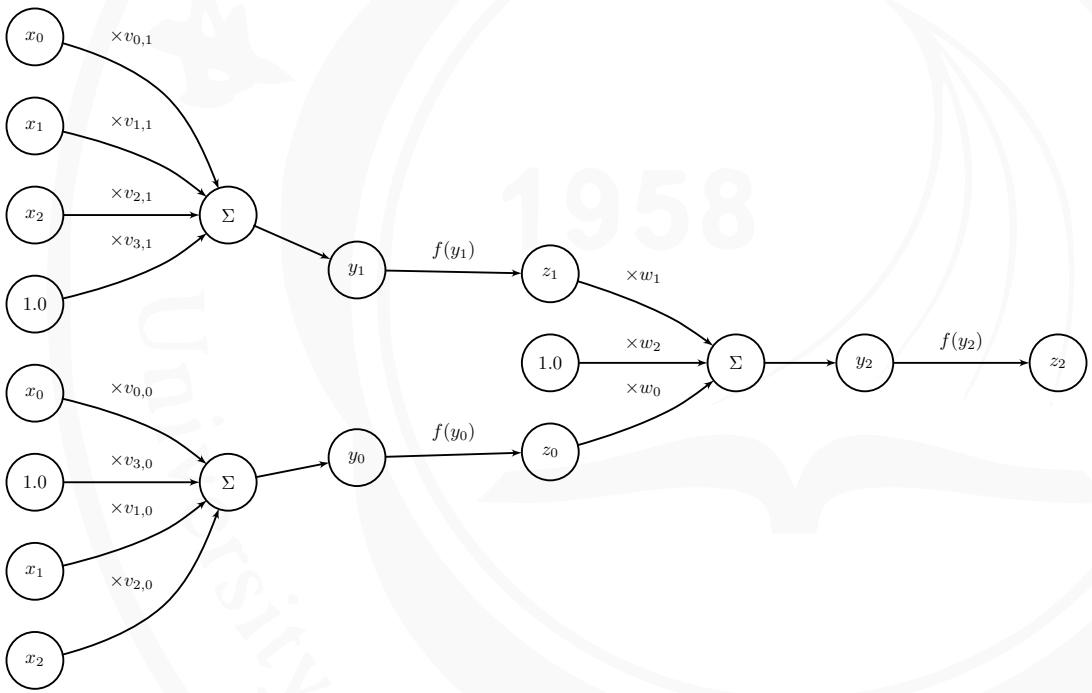
2 neurons getting input X, one neuron getting the output of the 2 first neurons



2 neurons, to have 2 hyperplanes for classifying data

Combining neurons

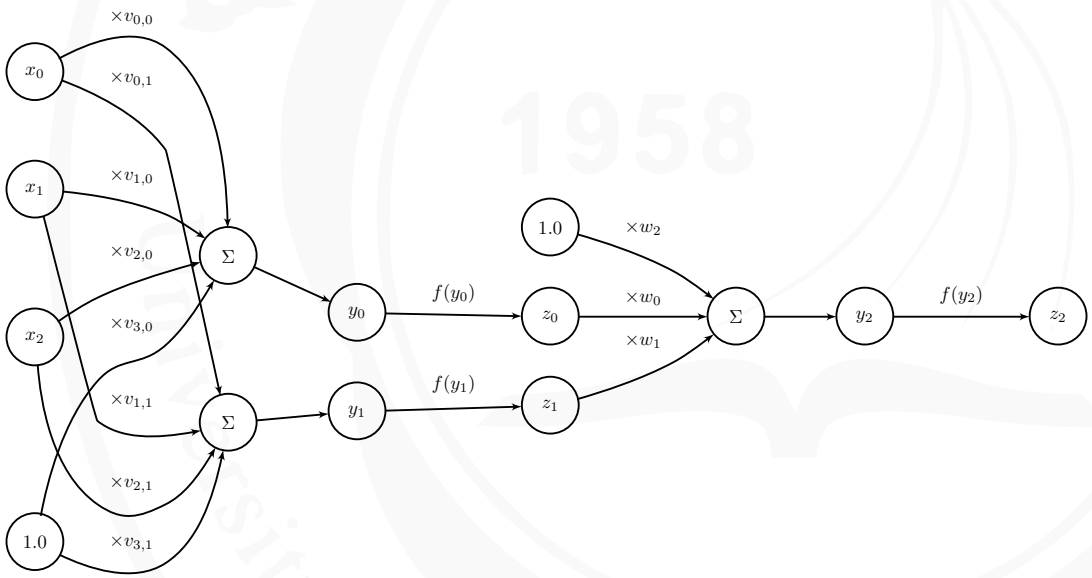
2 neurons getting input X, one neuron getting the output of the 2 first neurons



1 neuron to combine the answers of the 2 neurons connected to inputs

Combining neurons

2 neurons getting input X, one neuron getting the output of the 2 first neurons

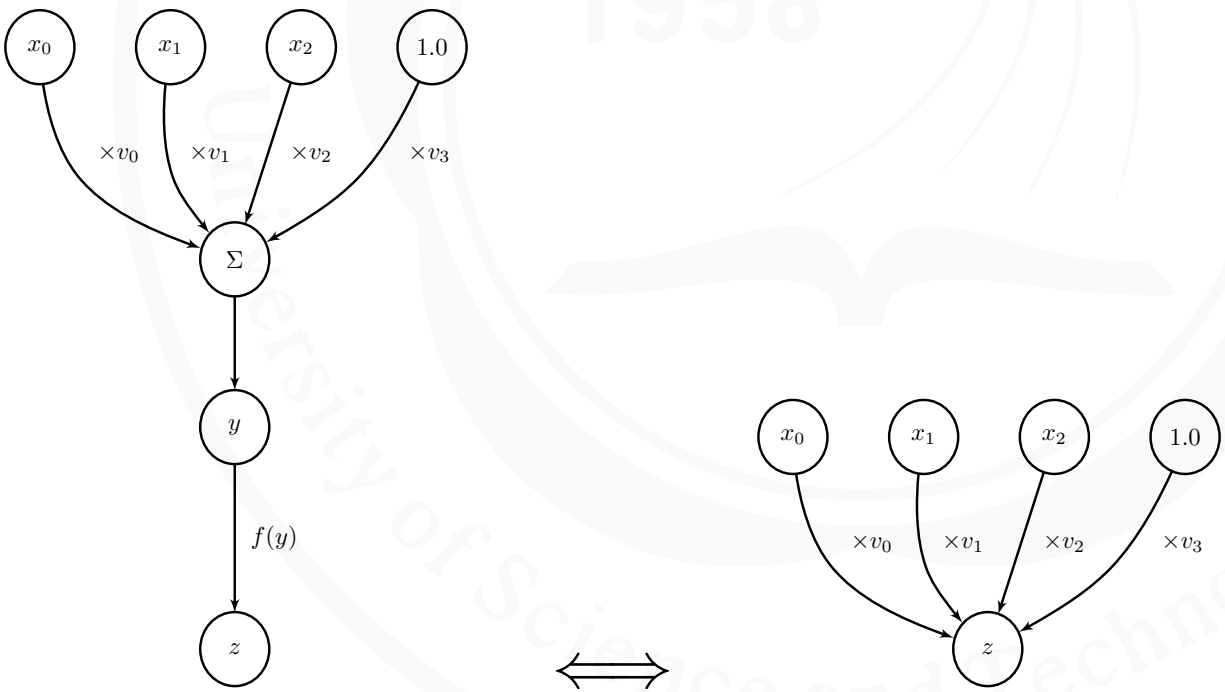


Same inputs x_i, so we can make it more compact

Combining neurons

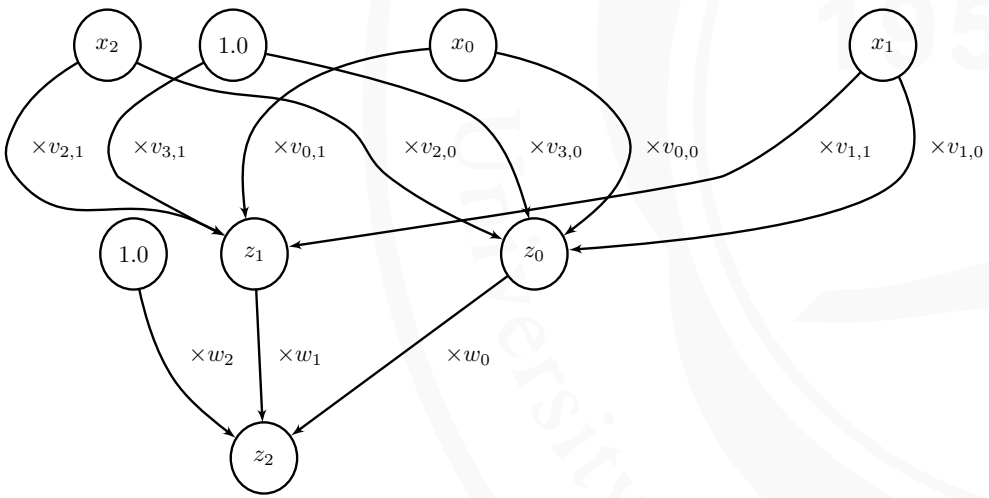
Let's make things a bit more compact

$$z = f\left(\sum_{i=0}^{n-1} v_i x_i + v_n\right)$$



Multi-Layer Perceptron

Our 2 combined hyperplanes construction is thus equivalent to this



$$z_0 = f\left(\sum_{i=0}^{n-1} v_{0,i} x_i + v_{0,n}\right)$$

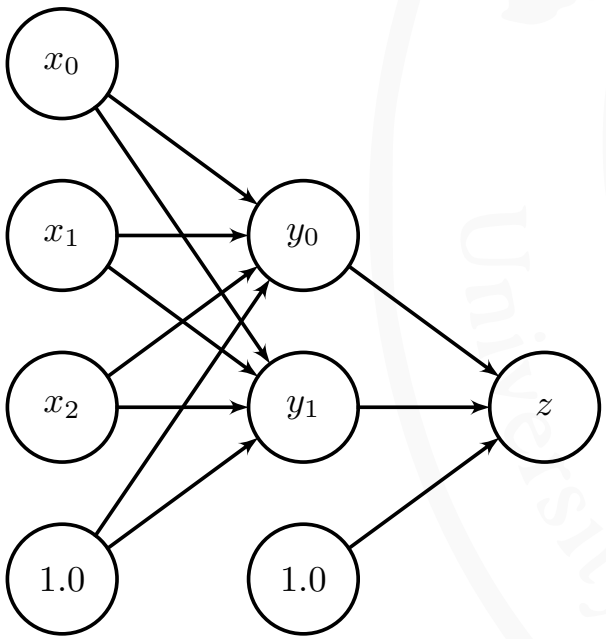
$$z_1 = f\left(\sum_{i=0}^{n-1} v_{1,i} x_i + v_{1,n}\right)$$

$$z_2 = f\left(\sum_{i=0}^{1} w_i z_i + w_2\right)$$

And this is called a *multi-layer perceptron* (*MLP*)

Multi-Layer Perceptron

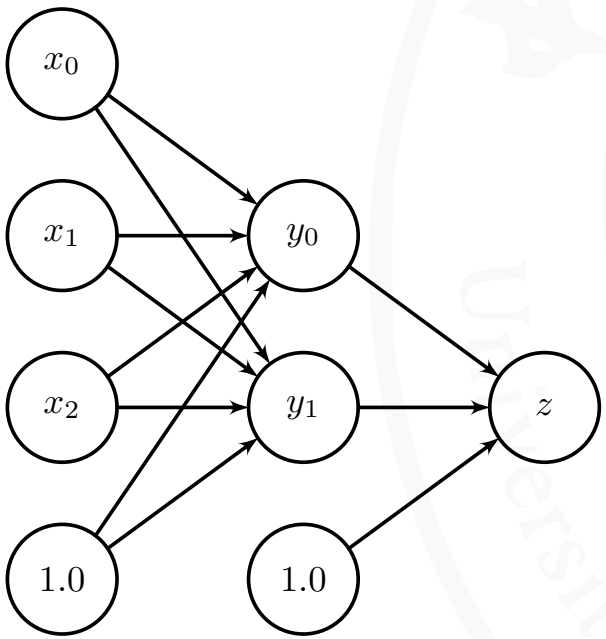
Let's re–arrange a bit the variables naming



- x_i, n components of input vector X
- y_i, outputs of 1st layer
- z, output of the perceptron
- $v_{j,i}$, connection weight from x_i to y_j
- w_i, connection weight from y_j to z

Multi-Layer Perceptron

We can compute the output z as following



$$y_b = f\left(\sum_{j=0}^{n-1} v_{b,i} x_i + v_{b,n}\right)$$

$$z = f\left(\sum_{i=0}^{h-1} w_i y_i + w_h\right)$$

h is the number of hyperplanes, or inner neurons, of our classifier

Multi-Layer Perceptron

A Multi-Layer Perceptron is made of c neurons, connected to an output neuron

- Each inner neuron acts as an independent hyperplanes
- The top neuron combines the independent hyperplanes
- We can now classify data by combining hyperplanes

Table of Contents

MLP learning

The idea $\Rightarrow$ reducing the error steps by steps, exactly like what we did for the *Delta rule*

1. pick a point at random from the dataset
2. computes the error for that point
3. alters the hyperplanes to reduce the error a bit
4. repeat ...

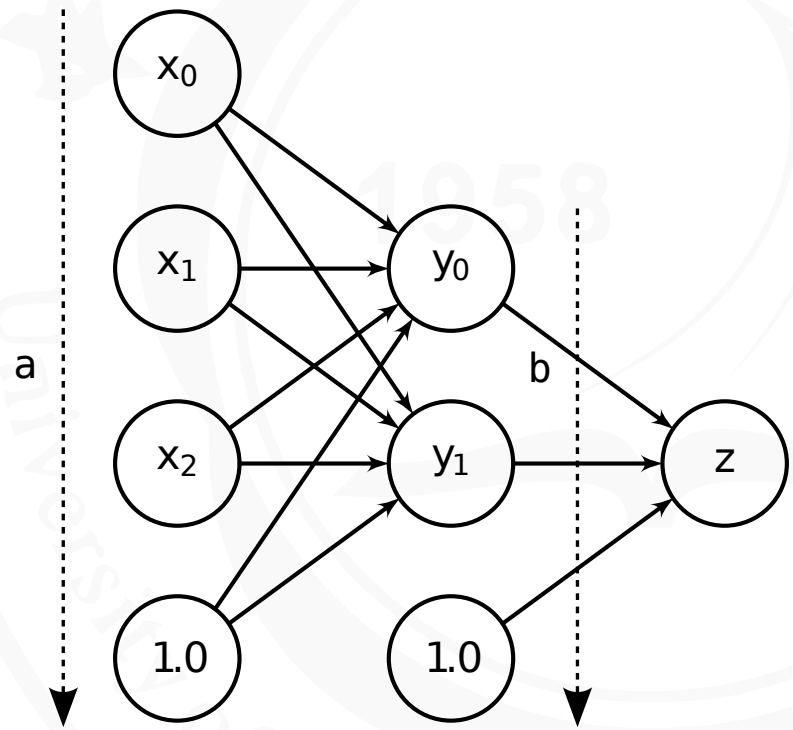
Output error

The output error e^{out} for a point X with class z^*

$$e^{\text{out}} = \frac{1}{2}(z - z^*)^2$$

Gradient descent

A little convention for the formulas which will follow



Gradient descent

We want to adjust the connection weights $v_{b,a}$ and w_b, to reduce the error e^{out}.

$$\frac{\partial e^{\text{out}}}{\partial v_{b,a}}$$

$$\frac{\partial e^{\text{out}}}{\partial w_b}$$

Like for the *Delta Rule*, we will use the 1st-derivatives to compute a correction for $v_{b,a}$ and w_b

Gradient descent

Let's start with $\frac{\partial e^{\text{out}}}{\partial w_b}$

$$\frac{\partial e^{\text{out}}}{\partial w_b} = (z - z^*)\frac{\partial z}{\partial w_b}$$

$$\frac{\partial e^{\text{out}}}{\partial w_b} = (z - z^*)y_b f'\left(\sum_{i=0}^{h} w_i y_i\right)$$

To keep the formulas short, we assume that $y_h = 1$ and $x_n = 1$

Gradient descent

Now, for the tough part $\frac{\partial e^{\text{out}}}{\partial v_{b,a}}$

$$\frac{\partial e^{\text{out}}}{\partial v_{b,a}} = (z - z^*)\frac{\partial z}{\partial v_{b,a}}$$

$$\frac{\partial z}{\partial v_{b,a}} = \frac{\partial \sum_{i=0}^{n} w_i y_i}{\partial v_{b,a}} f'\left(\sum_{i=0}^{h} w_i y_i\right)$$

$$\frac{\partial z}{\partial v_{b,a}} = w_b \frac{\partial y_b}{\partial v_{b,a}} f'\left(\sum_{i=0}^{h} w_i y_i\right)$$

$$\frac{\partial z}{\partial v_{b,a}} = w_b x_a f'\left(\sum_{i=0}^{n} v_{b,i} x_b\right) f'\left(\sum_{i=0}^{h} w_i y_i\right)$$

Gradient descent

We can now update $v_{b,a}$ and w_b

$$w_b(t+1) = w_b(t) + \eta \frac{\partial e^{\text{out}}}{\partial w_b}$$

$$v_{b,a}(t+1) = v_{b,a}(t) + \eta \frac{\partial e^{\text{out}}}{\partial v_{b,a}}$$

η is the *learning rate*, the strength of the correction we apply for an example point X

Table of Contents

Back–propagation algorithm

How to efficiently & elegantly compute the updates for $v_{b,a}$ and w_b ?

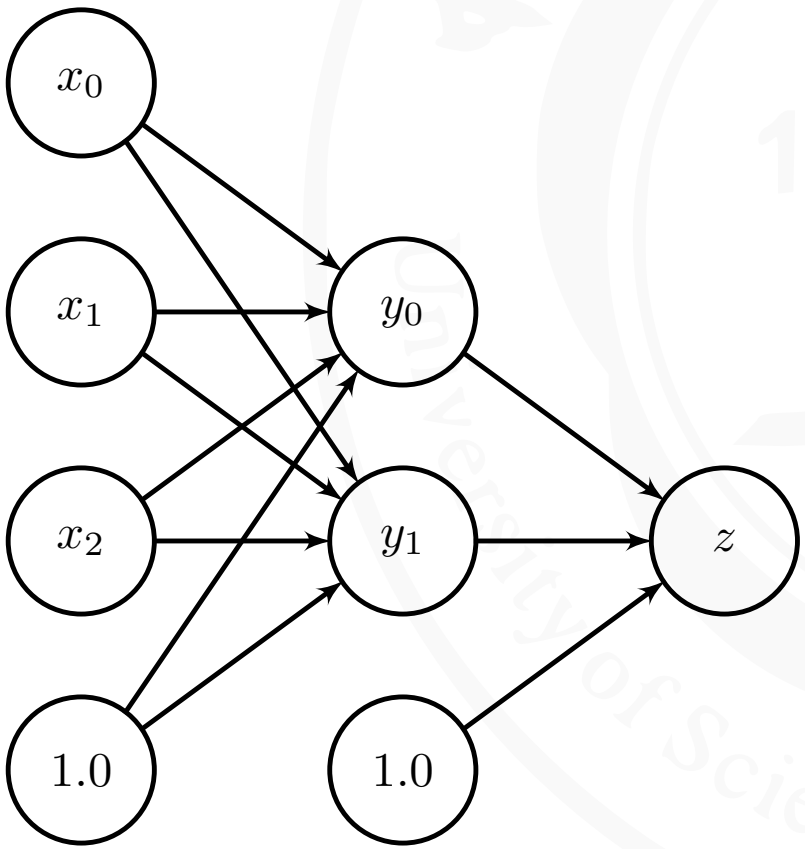
Back–propagation algorithm

The *back–propagation algorithm* is a two passes algorithm

1. the *forward pass*
2. the *backward pass*

Back–propagation algorithm

First, the *forward pass* computes, from the left to right layers



$$y_b = f\left(\sum_{i=0}^{n} v_{b,i} x_b\right)$$

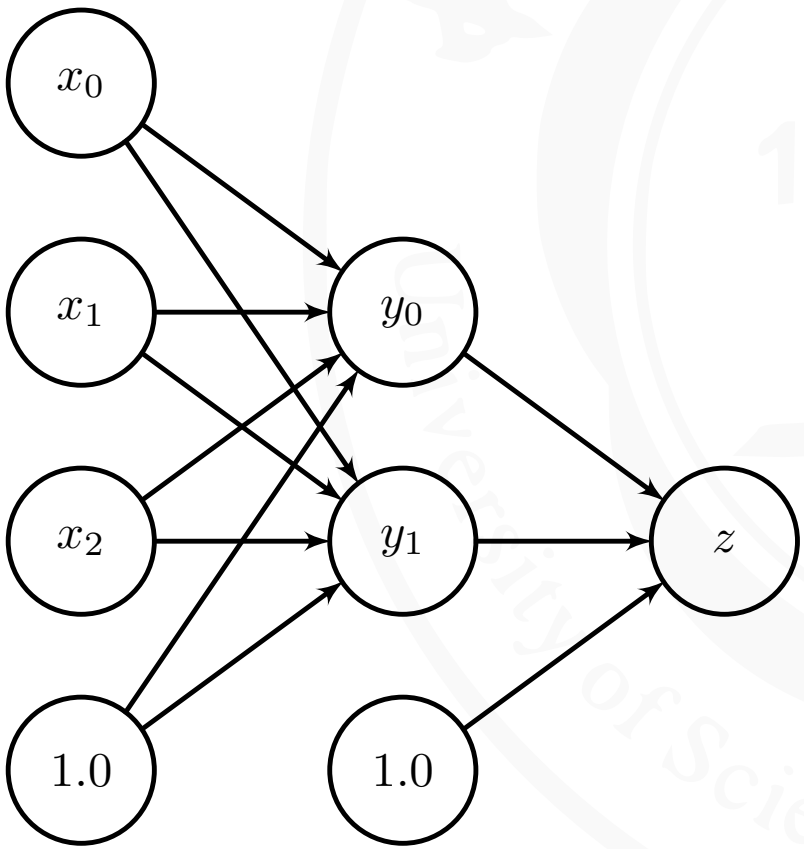
$$f'\left(\sum_{i=0}^{n} v_{b,i} x_b\right)$$

$$z = f\left(\sum_{i=0}^{c} w_i y_i\right)$$

$$f'\left(\sum_{i=0}^{c} w_i y_i\right)$$

Back–propagation algorithm

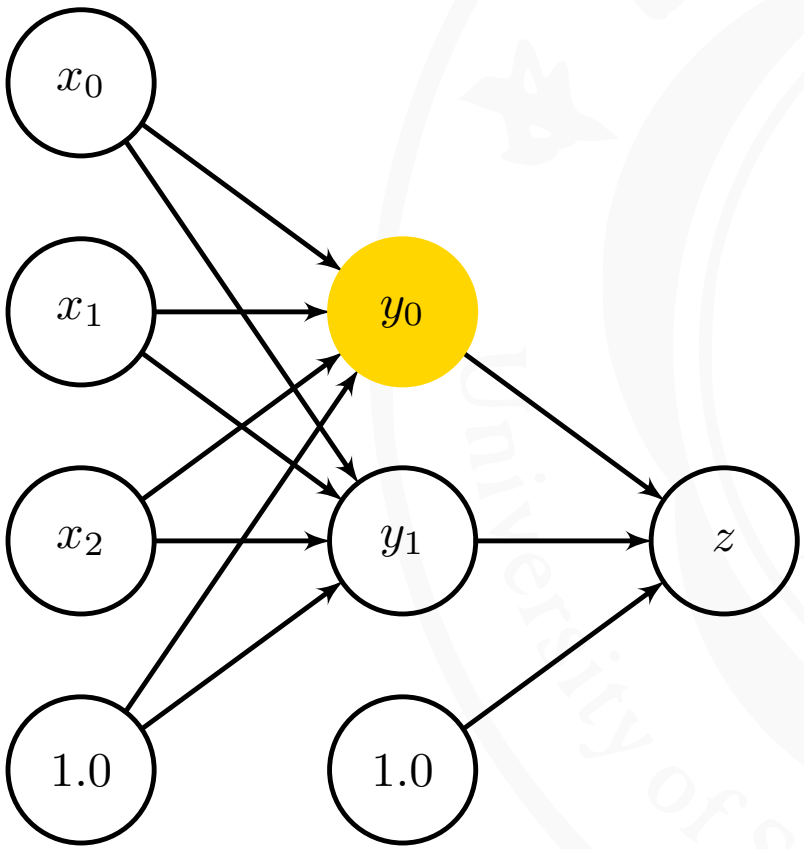
Second, the *backward pass* computes, from the right to left layers



$$\frac{\partial e^{\text{out}}}{\partial w_b}$$

$$\frac{\partial e^{\text{out}}}{\partial v_{b,a}}$$

Forward pass

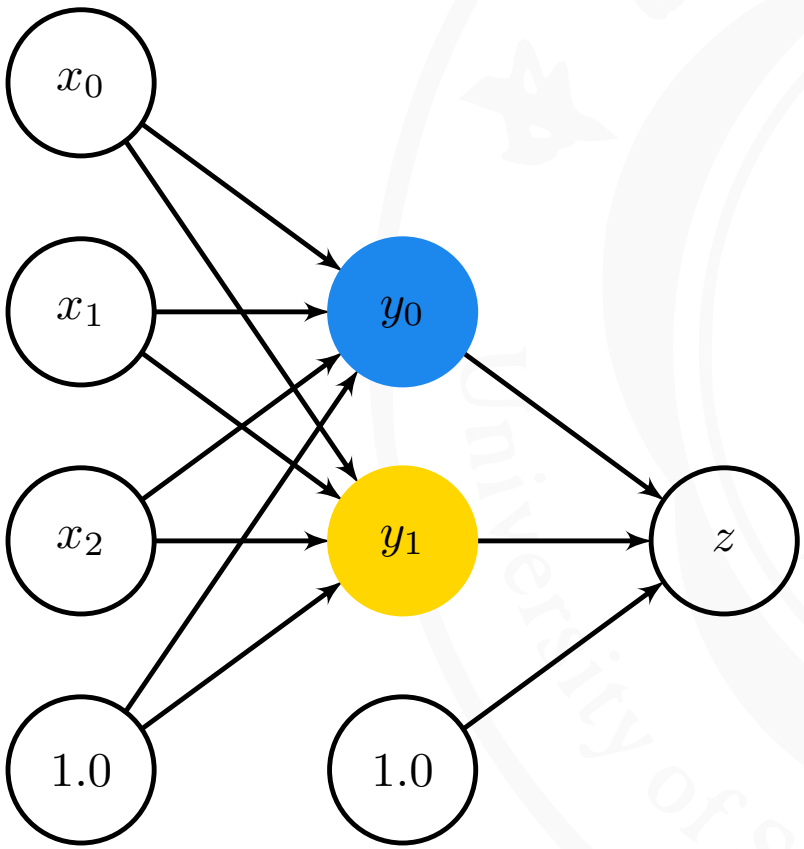


$$S = \sum_{i=0}^{3} v_{i,0} x_i$$

$$y_0 = f(S)$$

$$y_0' = f'(S)$$

Forward pass

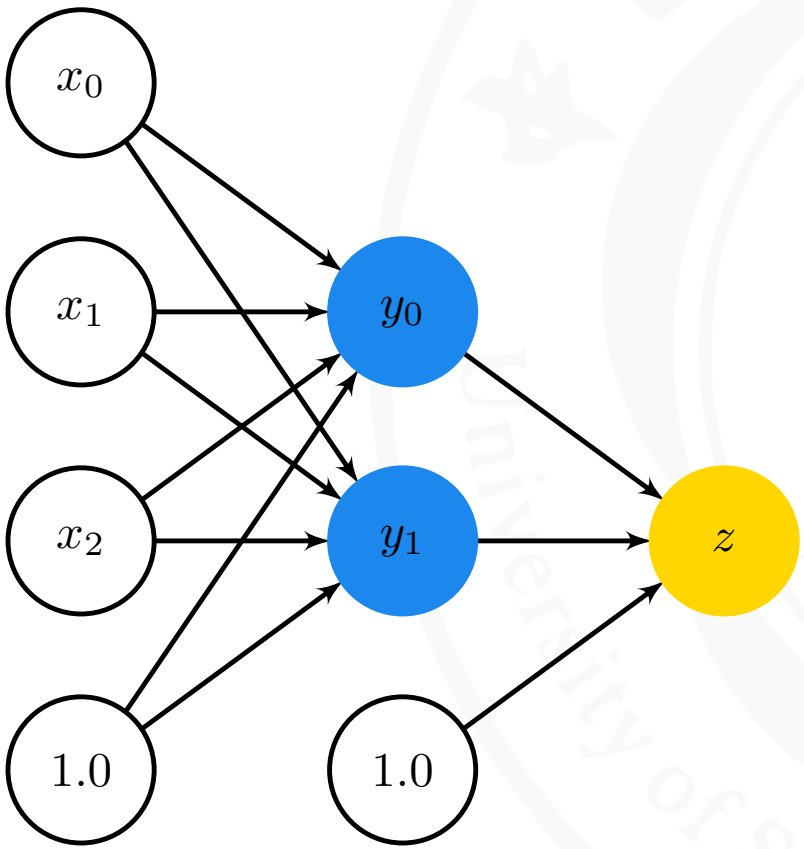


$$S = \sum_{i=0}^{3} v_{i,1} x_i$$

$$y_1 = f(S)$$

$$y_1' = f'(S)$$

Forward pass

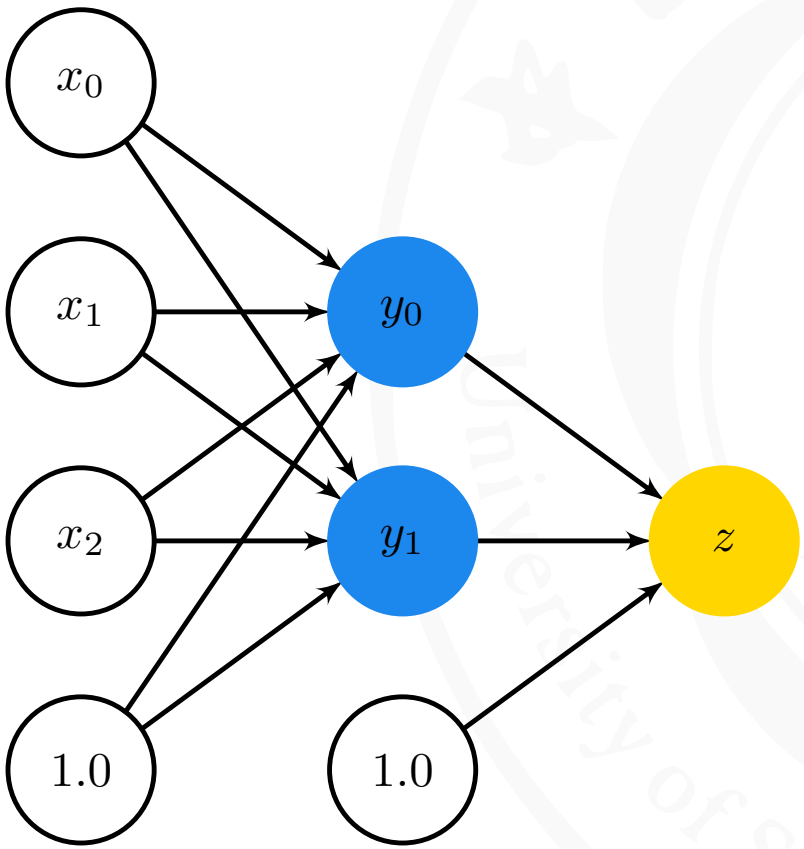


$$S = \sum_{i=0}^{2} w_i y_i$$

$$z = f(S)$$

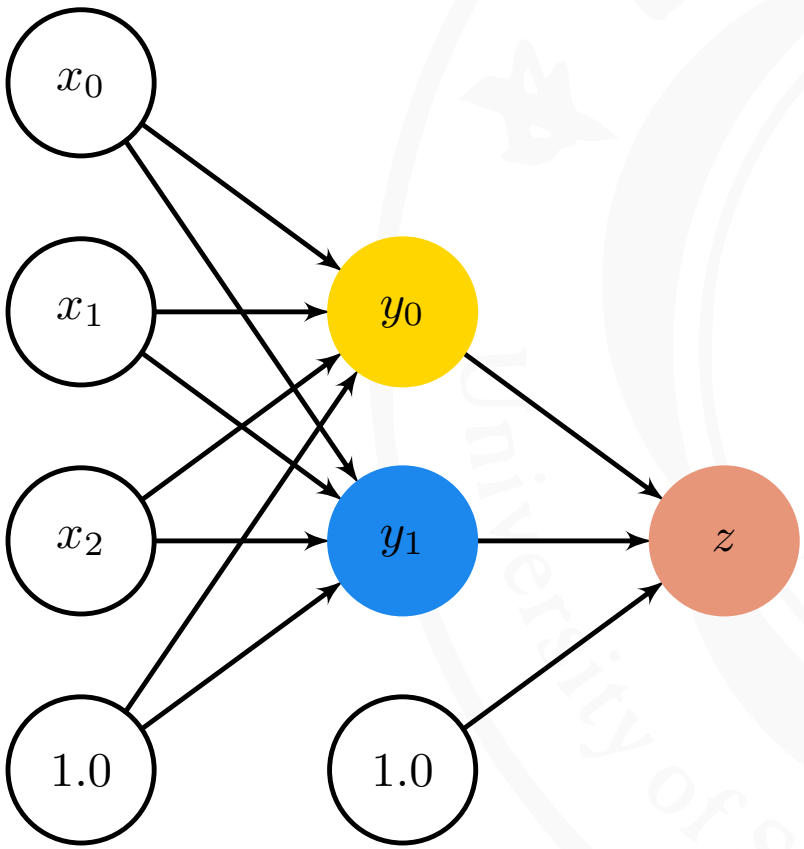
$$z' = f'(S)$$

Compute the error



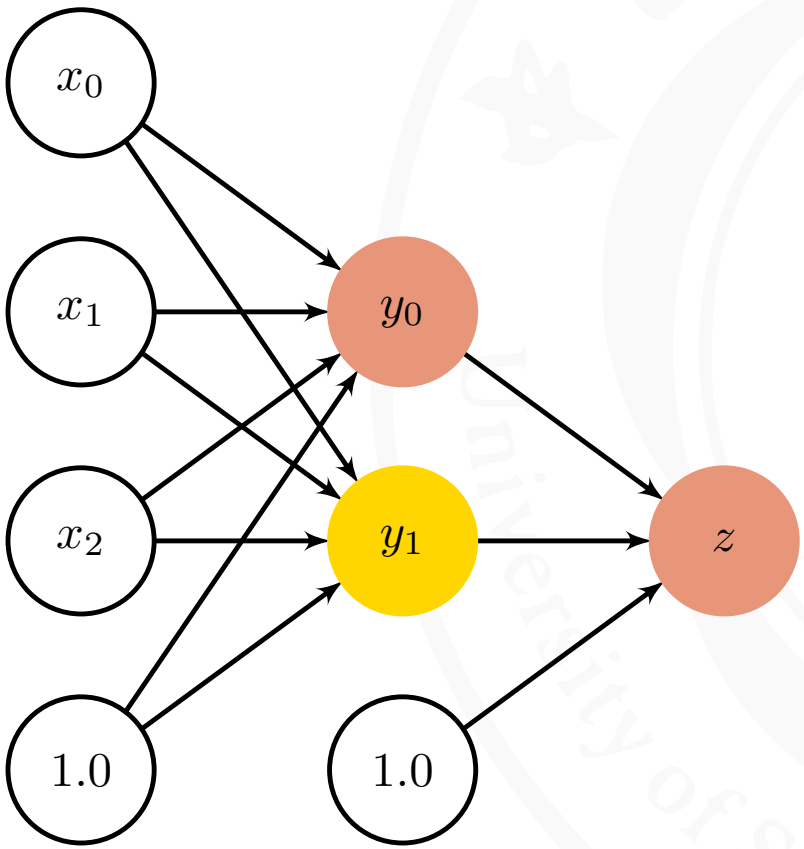
$$\rho^{out} = z'(z - z^*)$$

Retro-propagate the error



$$\rho_0^{in} = w_0 \rho^{\text{out}} y_0'$$

Retro-propagate the error



$$\rho_1^{in} = w_1 \rho^{\text{out}} y_1'$$

Retro-propagate the error

Adjusts the weights

$$\frac{\partial e^{\text{out}}}{\partial w_b} = y_b \rho^{\text{out}}$$

$$\frac{\partial e^{\text{out}}}{\partial v_{a,b}} = x_a \rho_b^{\text{in}}$$

$$w_b(t+1) = w_b(t) + \eta y_b \rho^{\text{out}}$$

$$v_{b,a}(t+1) = v_{b,a}(t) + \eta x_a \rho_b^{\text{in}}$$

Table of Contents

Regression

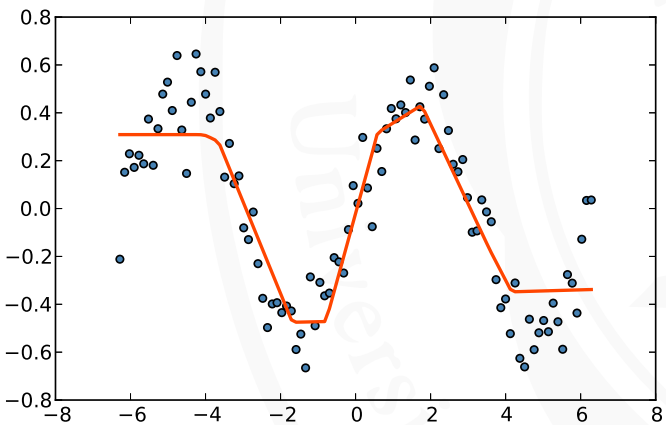
Retro-propagation algorithm can be used to do regression too

- A list of example points X
- A list of values t, one for each X
- Use back–propagation to reduce the error
- A very robust regression method !

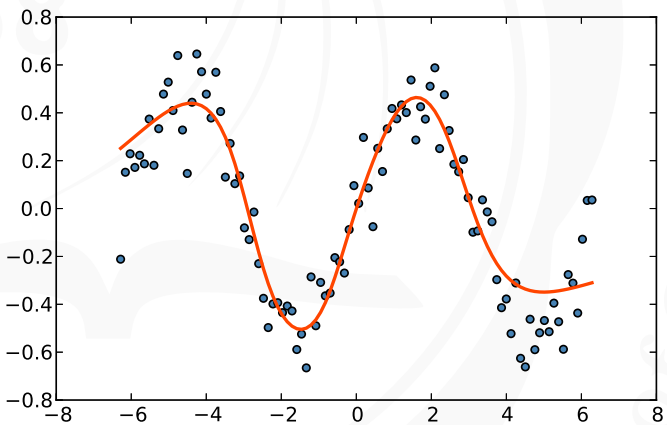
After learning, the MLP will approximate the function $X \Rightarrow z^*$

Transfer function

The transfer function you use matters a lot, if you do regression !



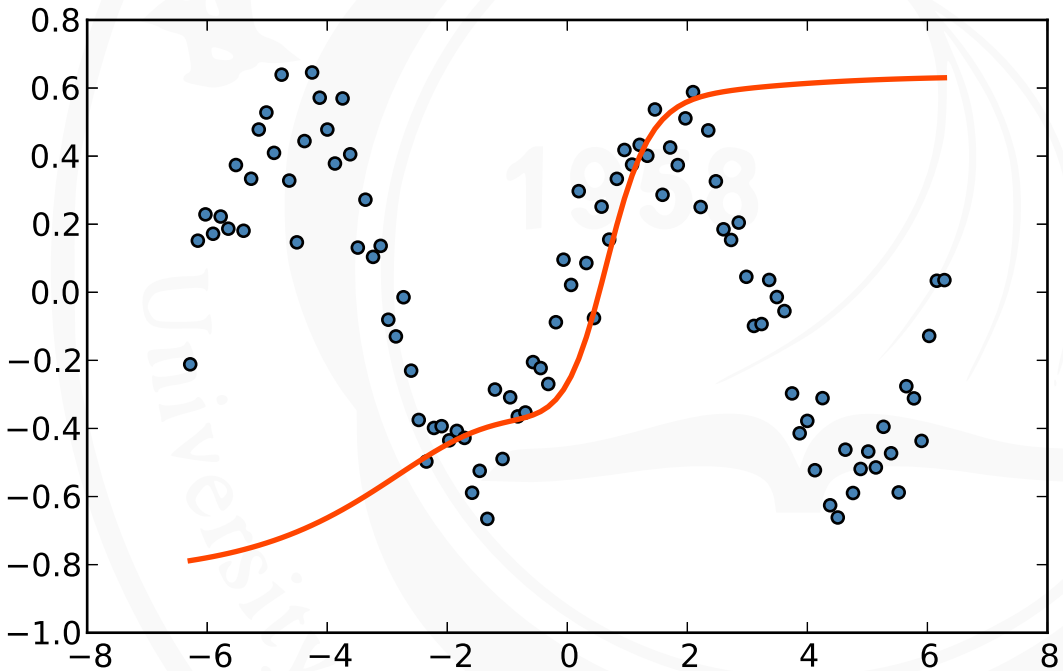
regression with linear step



regression with *tanh*

Regression

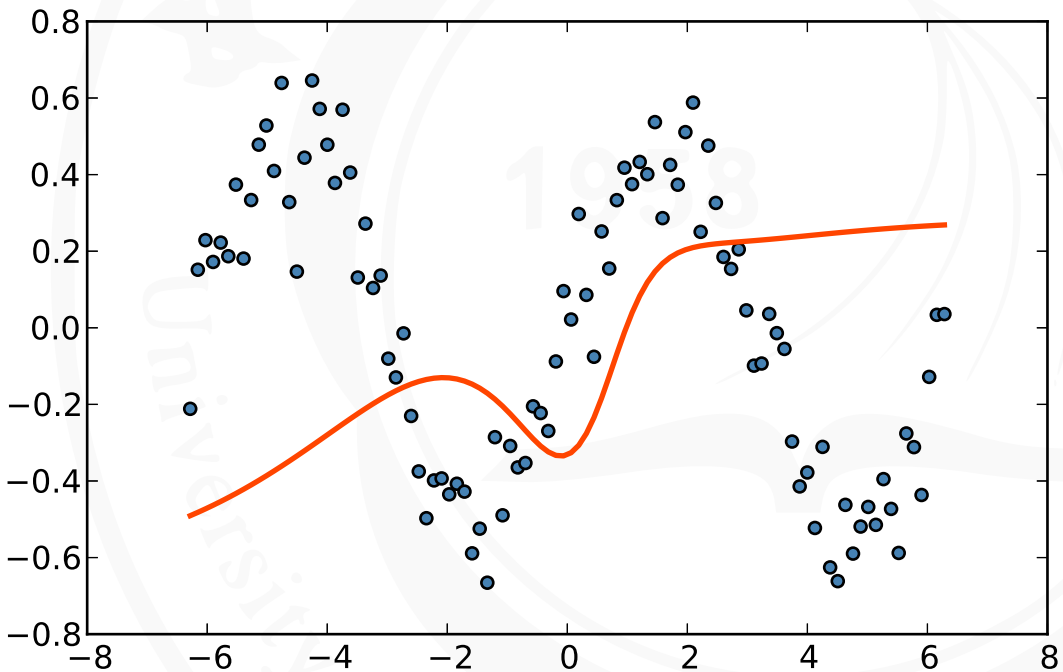
Regression in action : sinus function with noise



(careful) random initialization

Regression

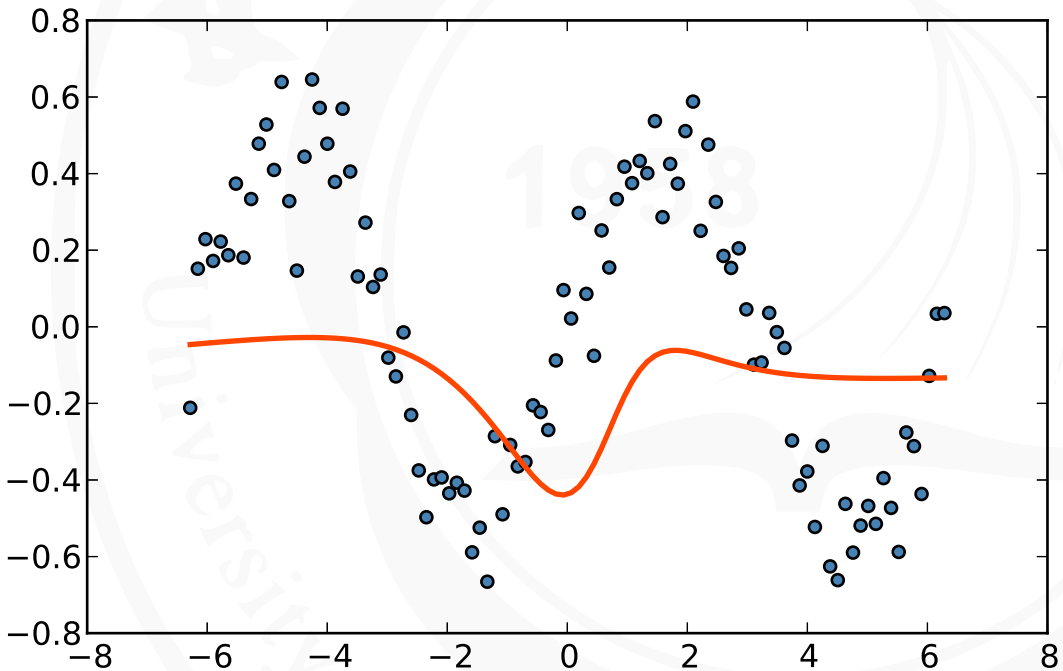
Regression in action : sinus function with noise



10 training epochs

Regression

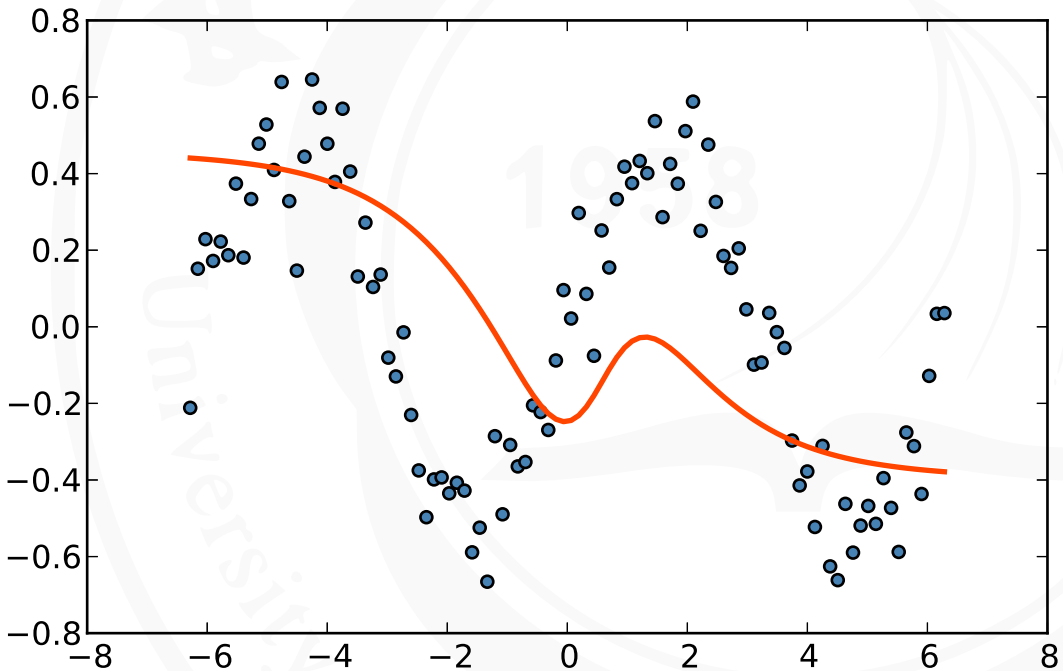
Regression in action : sinus function with noise



50 training epochs

Regression

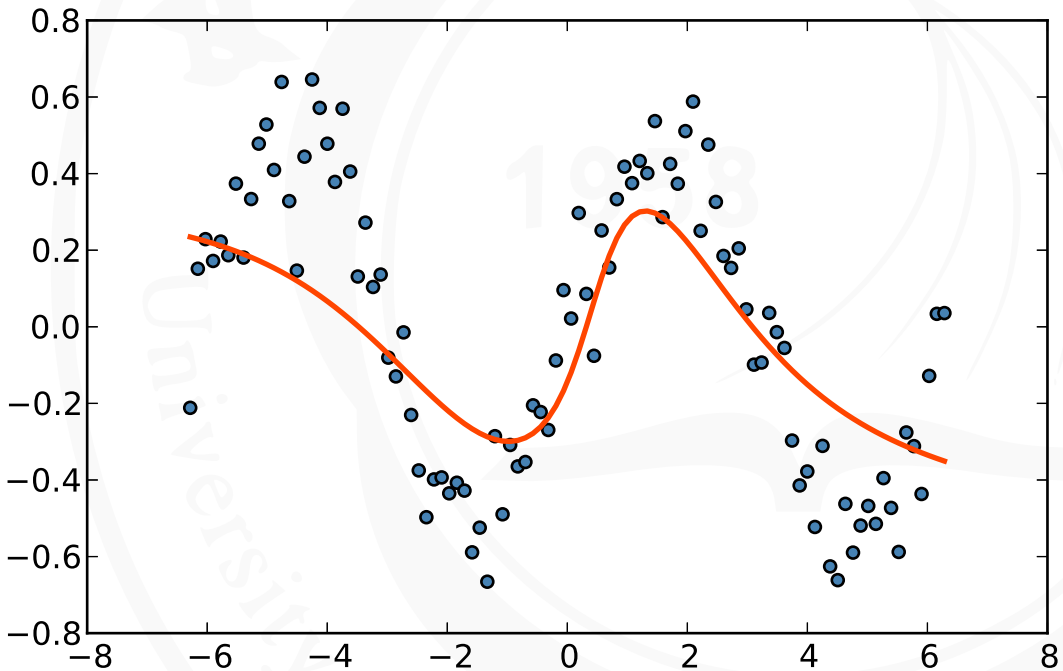
Regression in action : sinus function with noise



100 training epochs

Regression

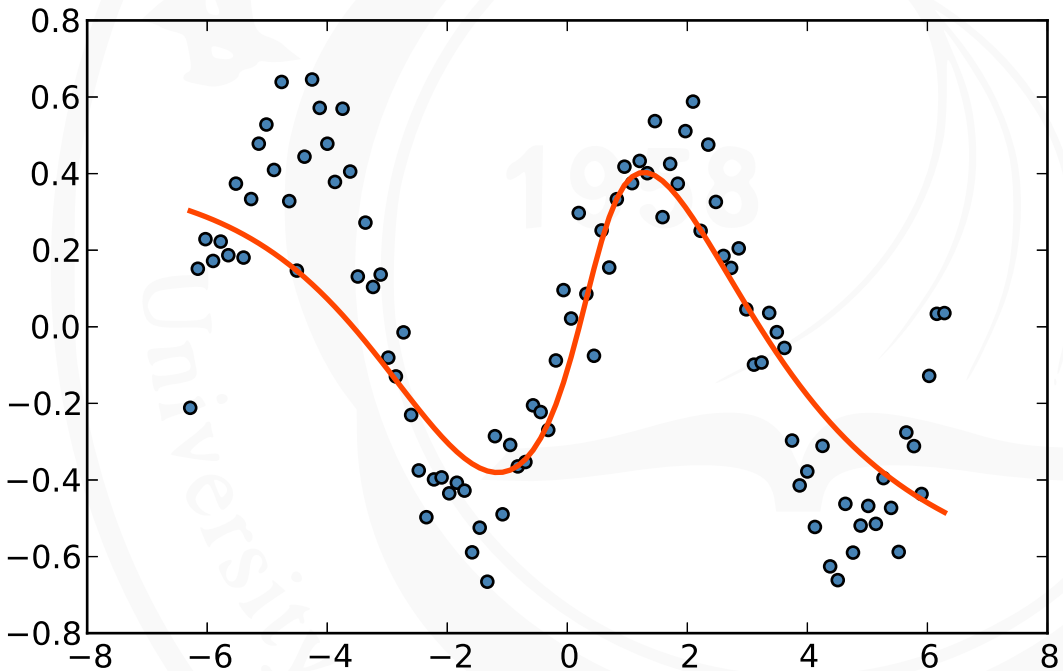
Regression in action : sinus function with noise



200 training epochs

Regression

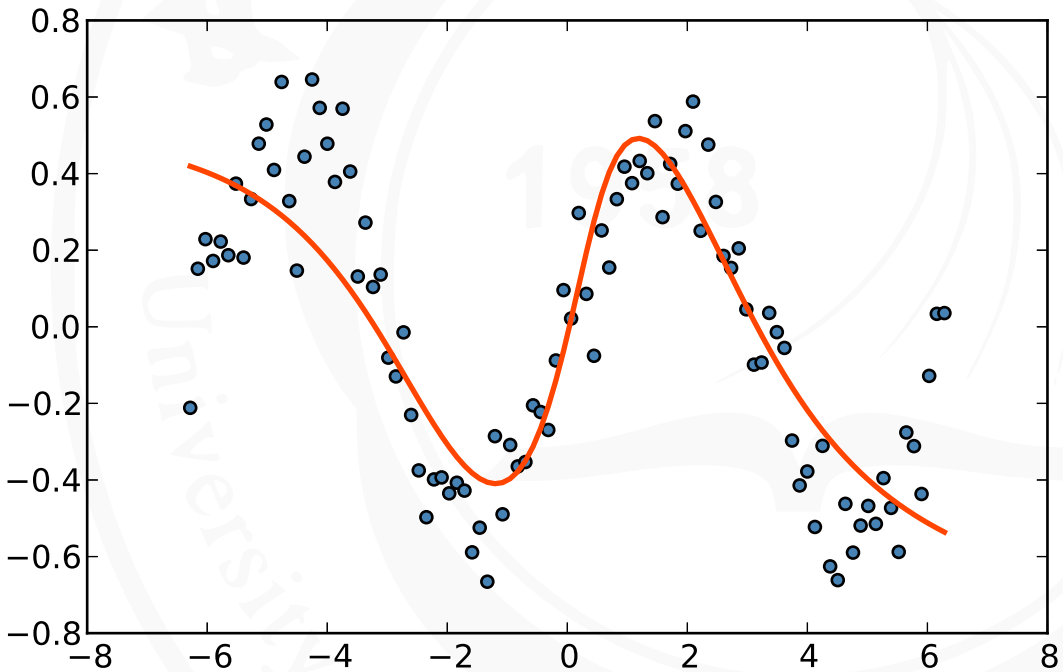
Regression in action : sinus function with noise



400 training epochs

Regression

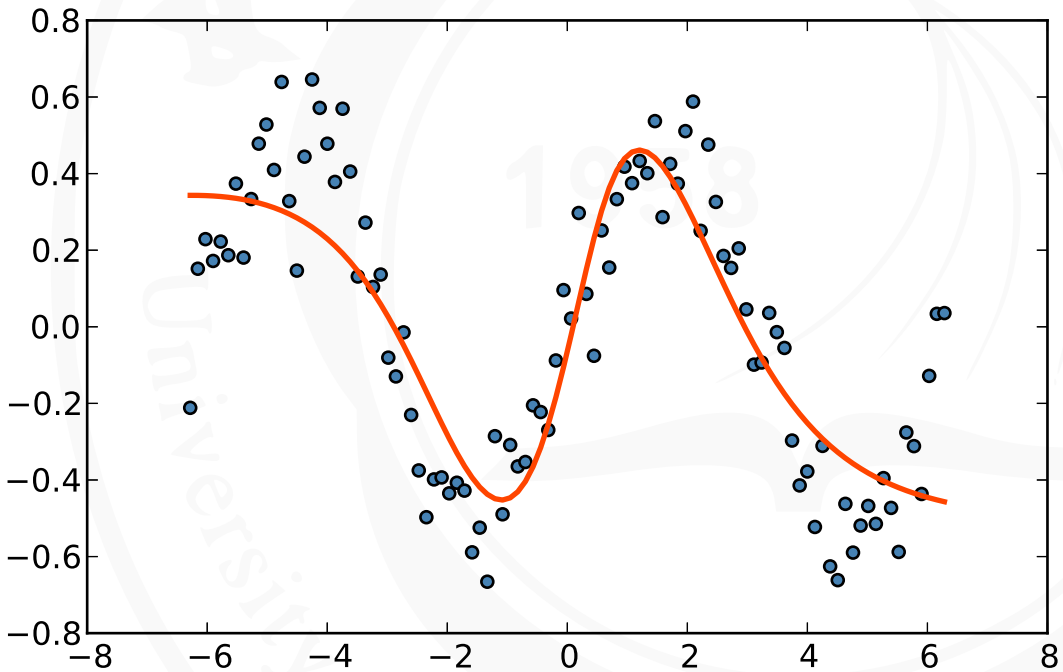
Regression in action : sinus function with noise



800 training epochs

Regression

Regression in action : sinus function with noise



1800 training epochs

Universality of the MLP

Cybenko theorem : an MLP can approximate any function !

Table of Contents

MLP learning

MLP can learn anything $\Rightarrow$ ultimate learning machine ?

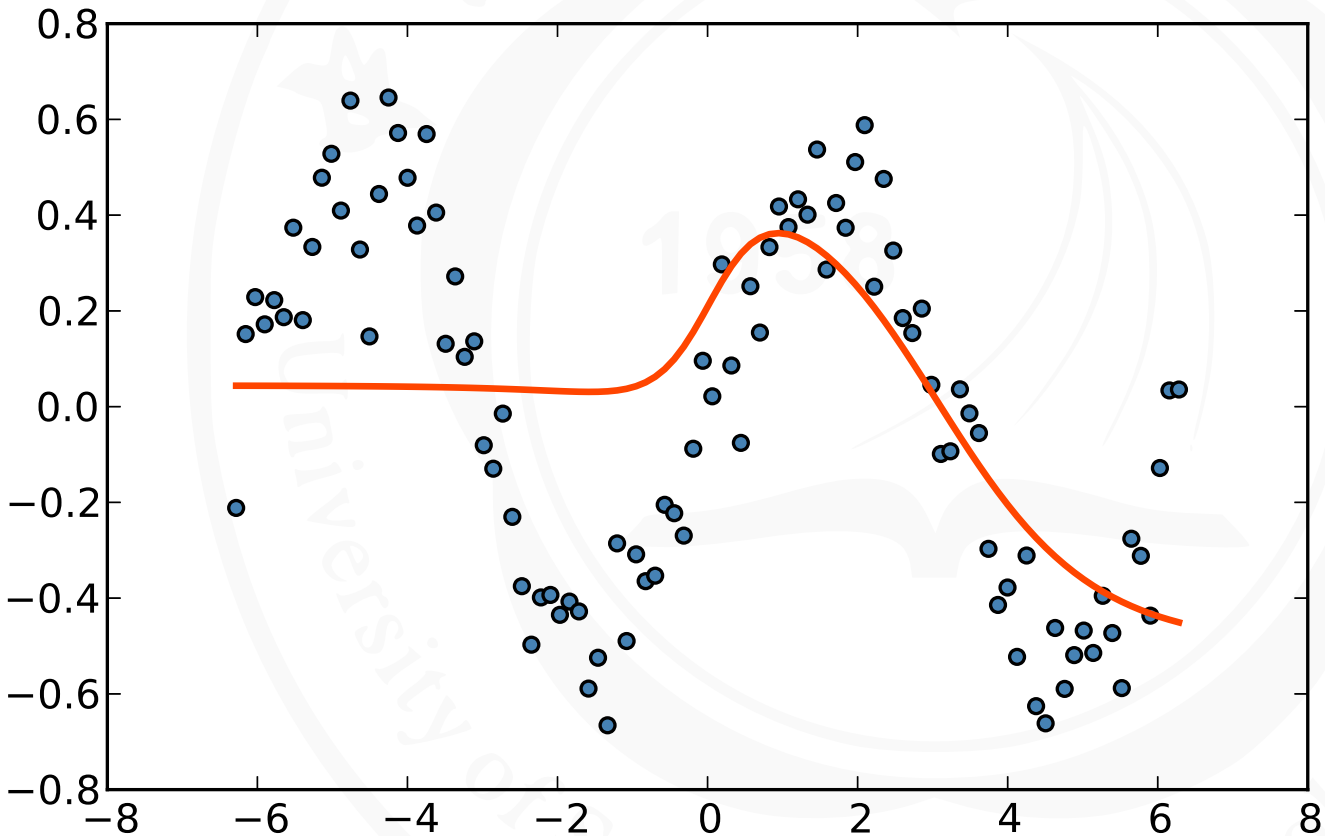
Data normalization

MLP won't work if you don't normalize your data

- The output of the MLP is in a limited range (say $[-1, 1]$)
- If the inputs are out of range, MLP loose information

Over-fitting

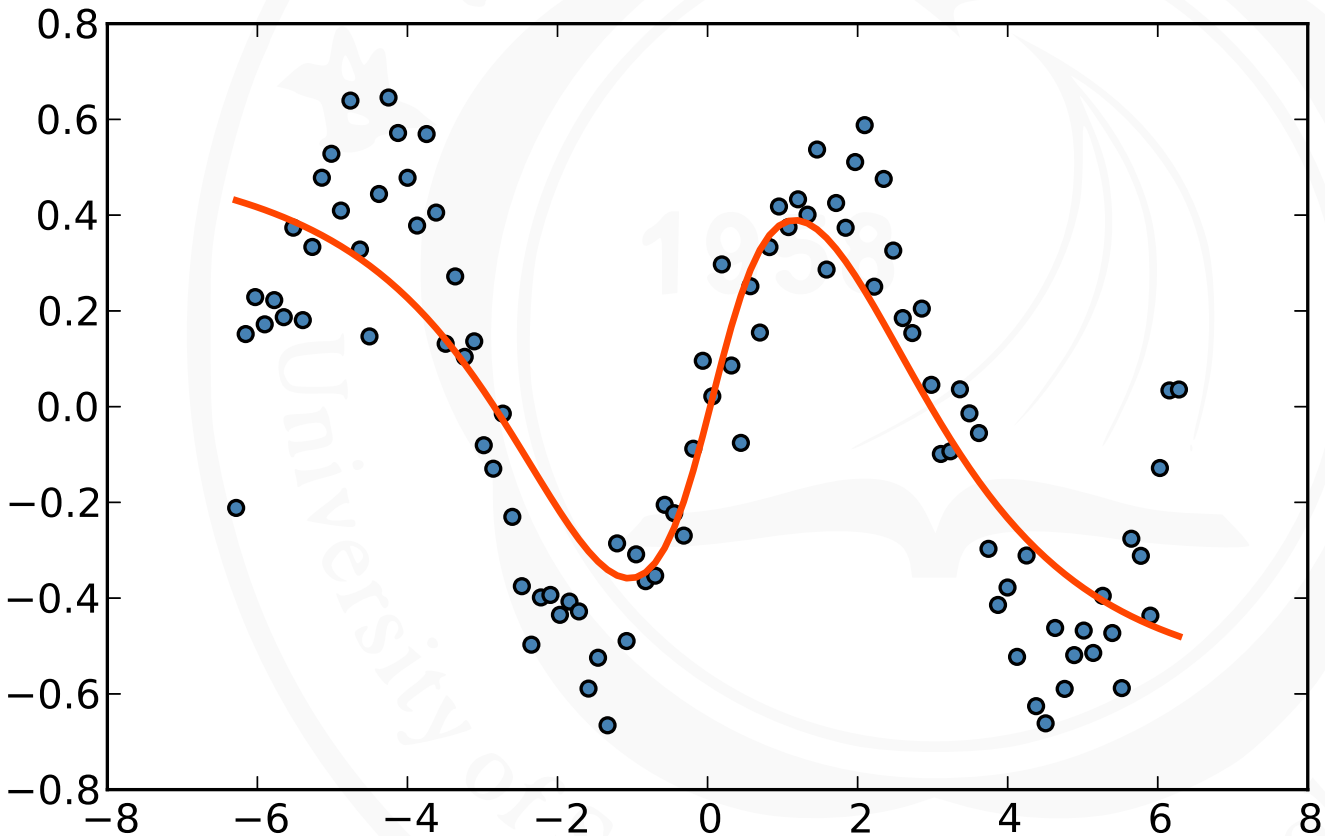
More neurons, better learning ?



2 neurons hidden layer

Over-fitting

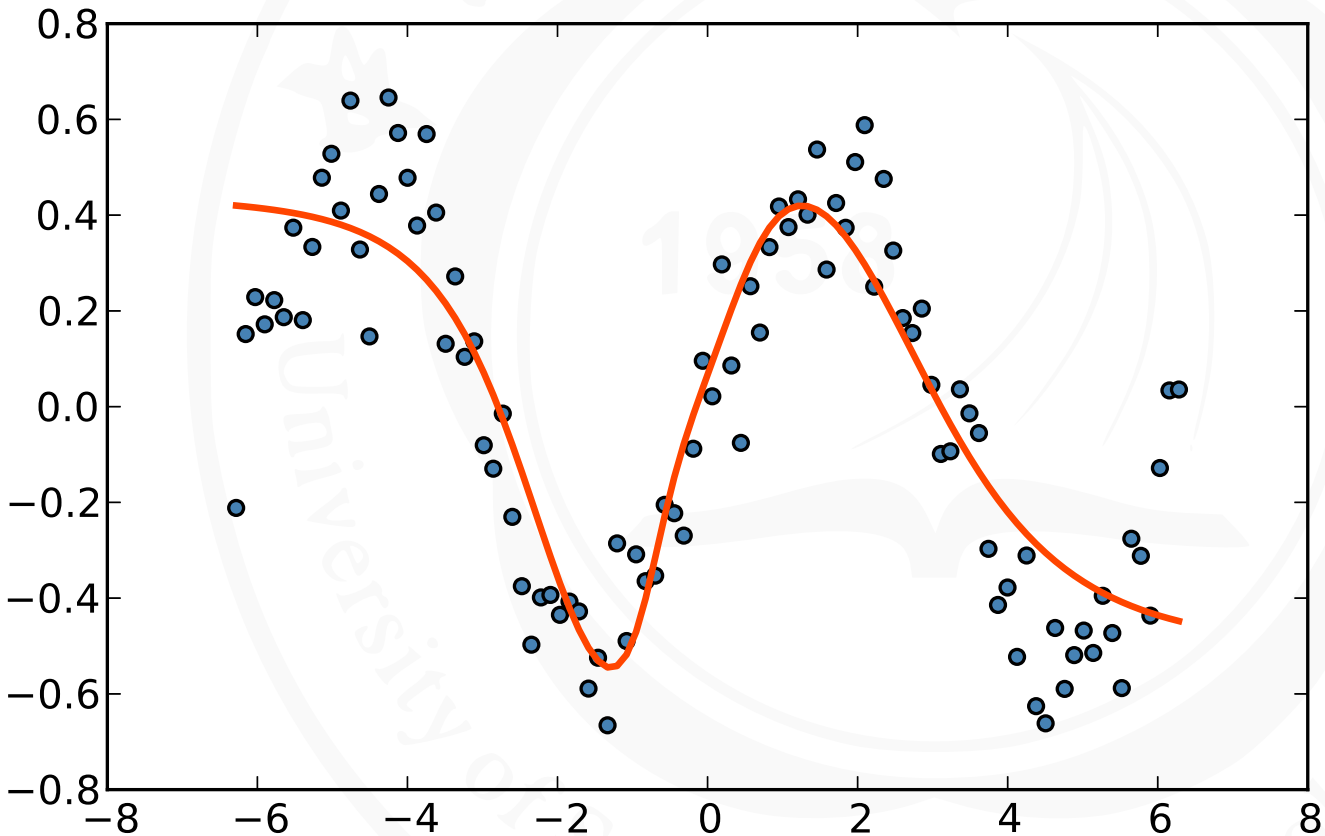
More neurons, better learning ?



4 neurons hidden layer

Over-fitting

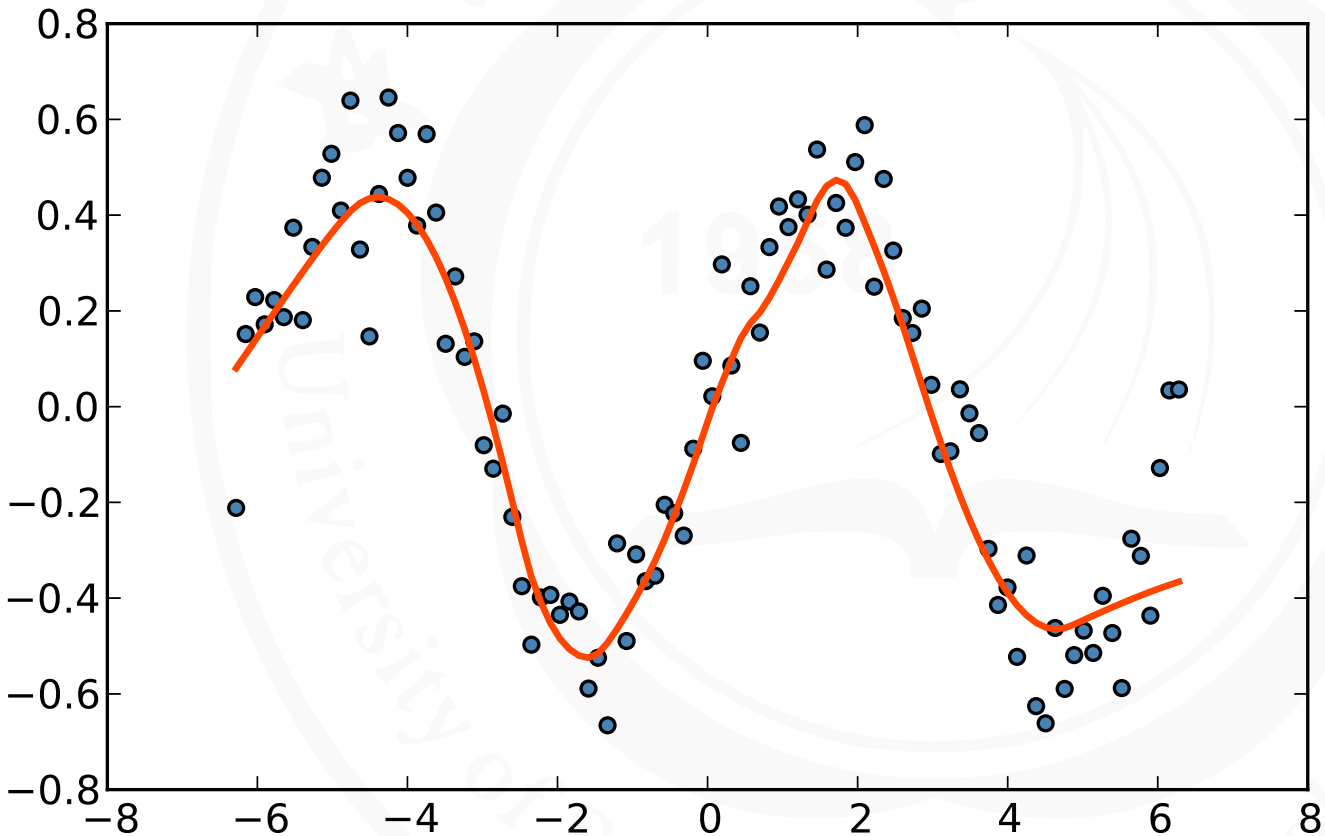
More neurons, better learning ?



6 neurons hidden layer

Over-fitting

More neurons, better learning ?



32 neurons hidden layer

Over-fitting

Past a number of a neurons

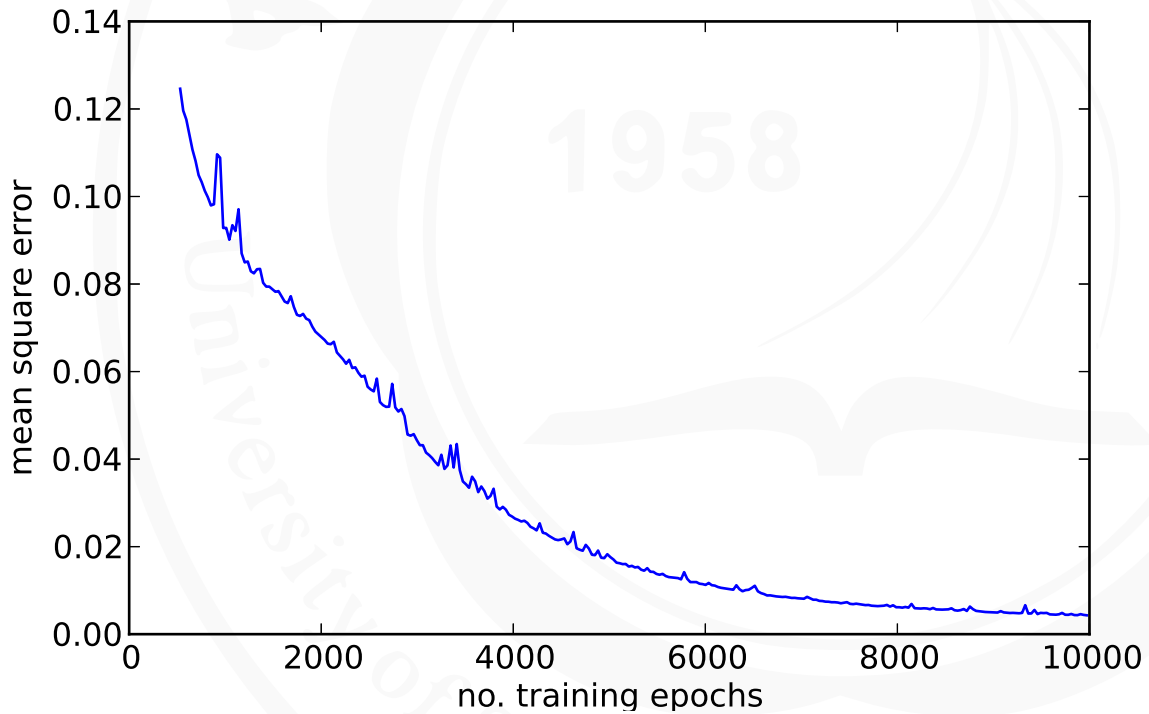
- Very little improvement of the error
- Mostly learning noise of the data
- *over-fitting*

Training error

Training error $\Rightarrow$ error for the training points

Training error

Training error converge to a minimum

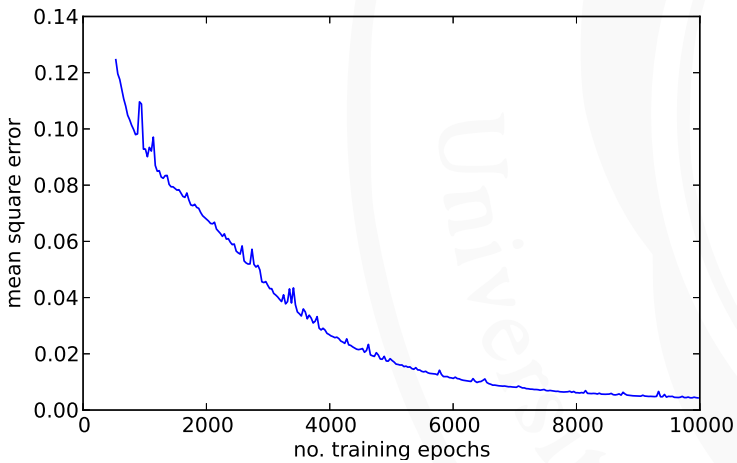


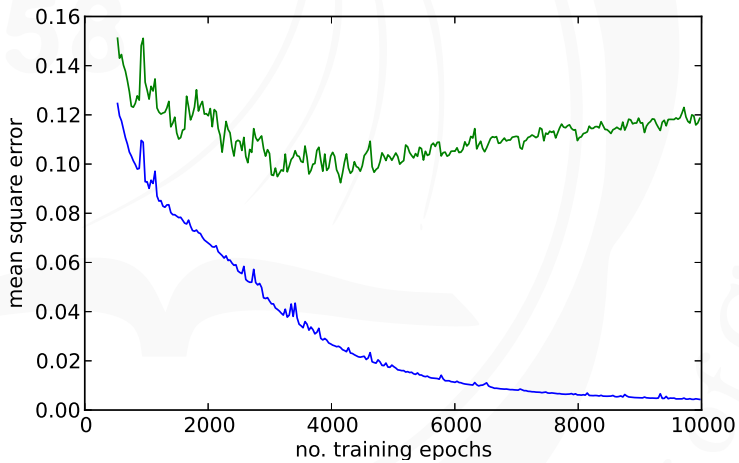
Testing error

Testing error $\Rightarrow$ error for points *not used* during training

Testing error

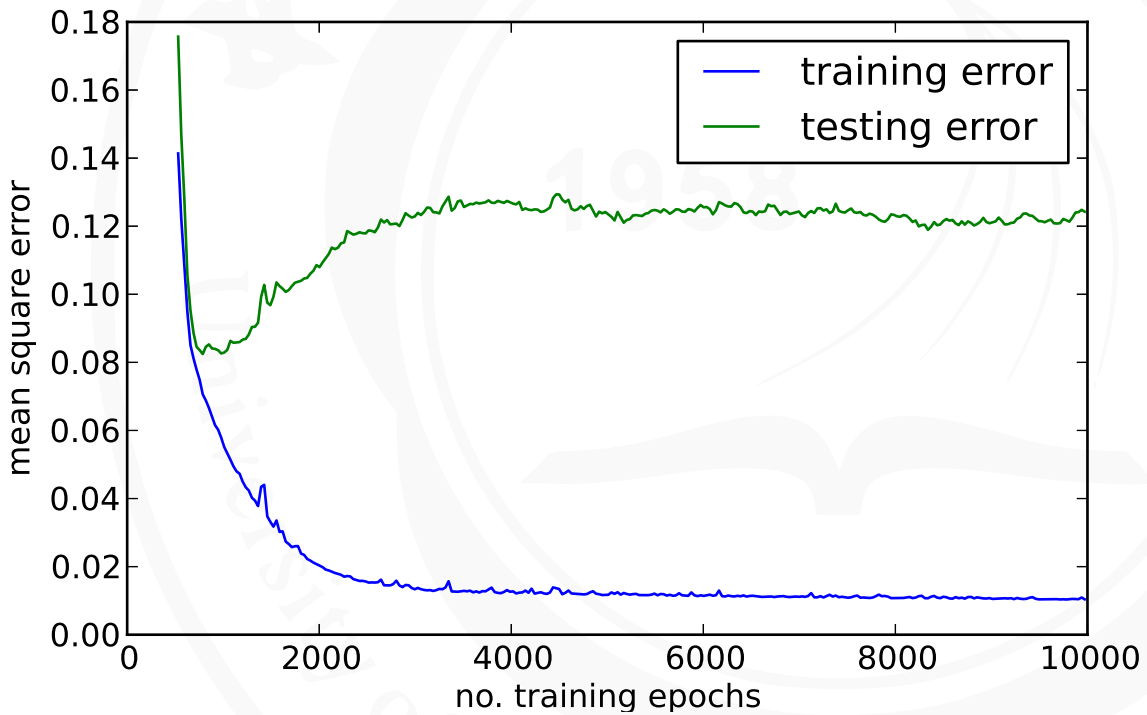
Testing error goes to a minimum . . . and raise up





Testing error

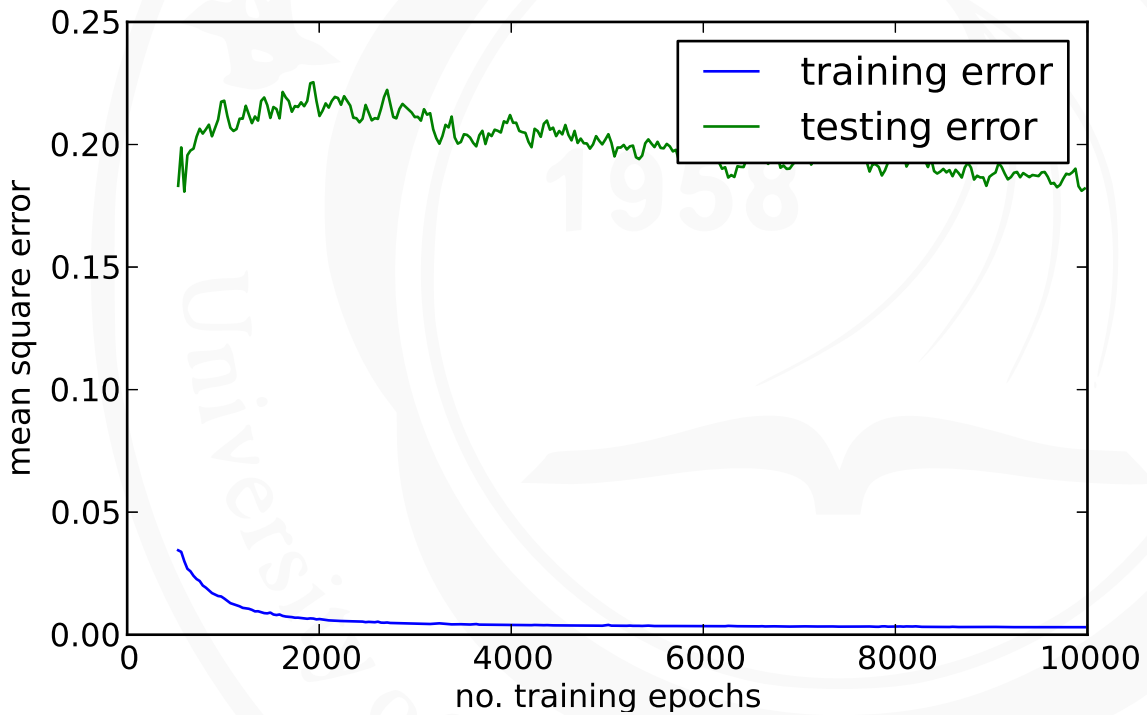
Testing error goes to a minimum . . . and raise up



10%/90%, 5 hidden neurons, $\eta = 10^{-2}$

Testing error

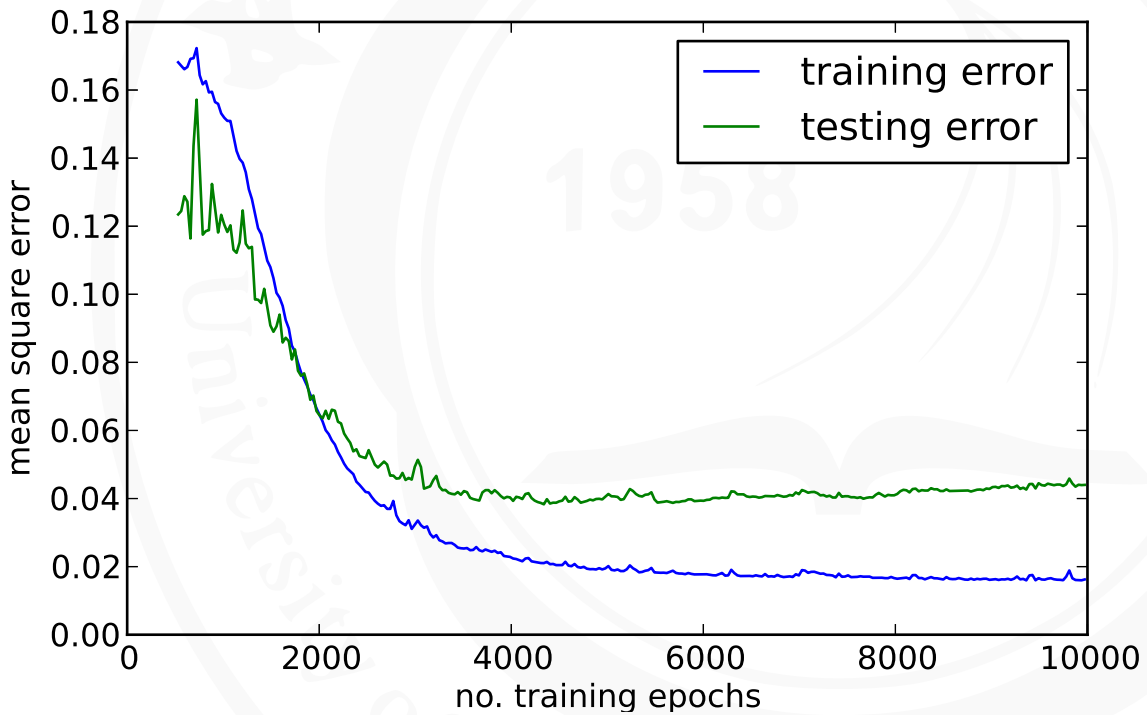
Testing error goes to a minimum . . . and raise up



10%/90%, 5 hidden neurons, $\eta = 10^{-2}$

Testing error

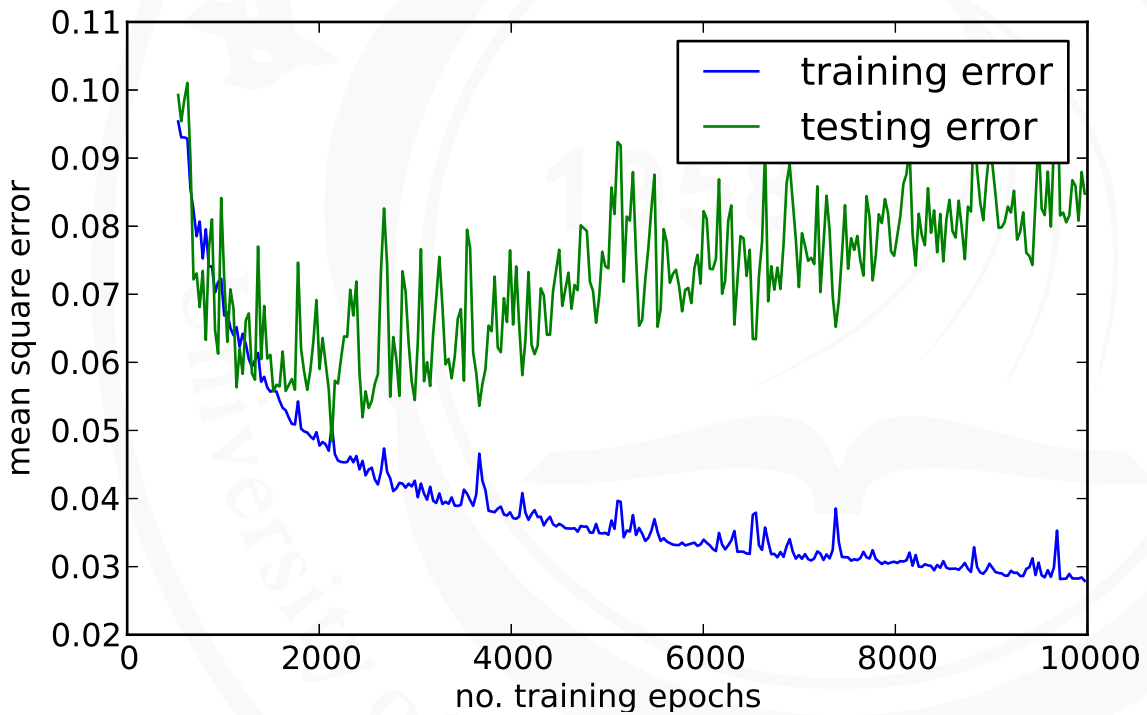
Testing error goes to a minimum . . . and raise up



10%/90%, 5 hidden neurons, $\eta = 10^{-2}$

Testing error

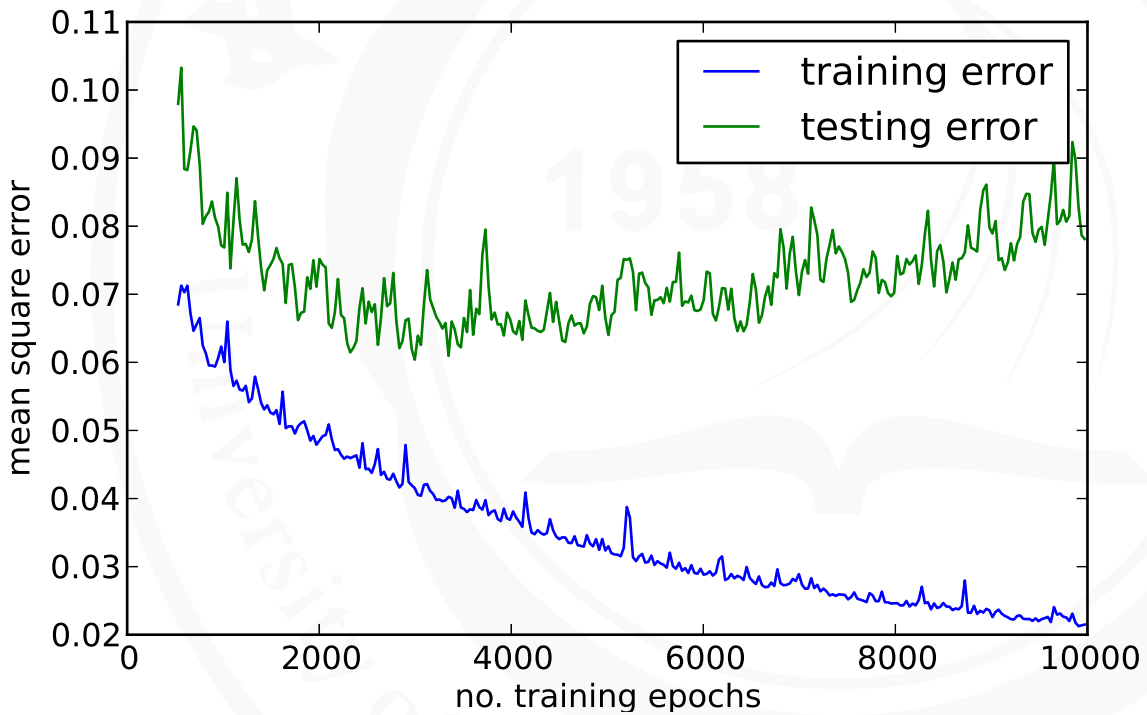
Testing error goes to a minimum . . . and raise up



10%/90%, 10 hidden neurons, $\eta = 10^{-2}$

Testing error

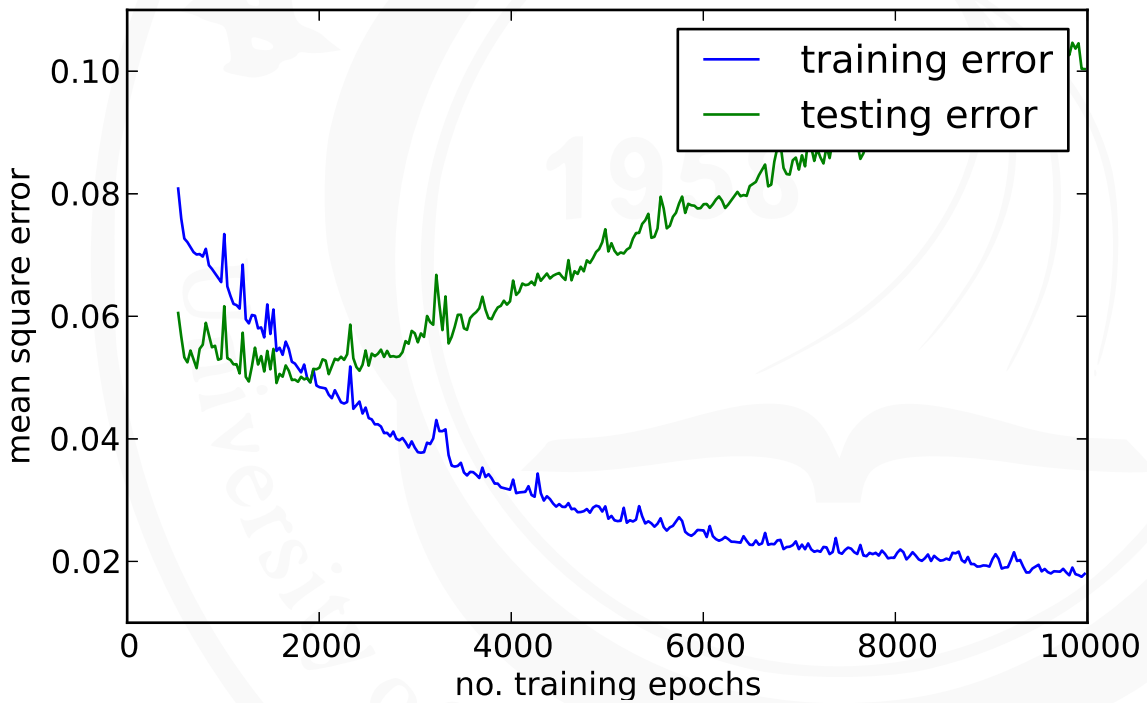
Testing error goes to a minimum . . . and raise up



10%/90%, 10 hidden neurons, $\eta = 10^{-2}$

Testing error

Testing error goes to a minimum . . . and raise up



10%/90%, 10 hidden neurons, $\eta = 10^{-2}$

MLP learning

When using any learning machine (MLP and others)

1. A training set and a test set
2. Observe what the algorithm is doing
3. Take theoretical properties with a grain of salt (and some lime too, taste better)