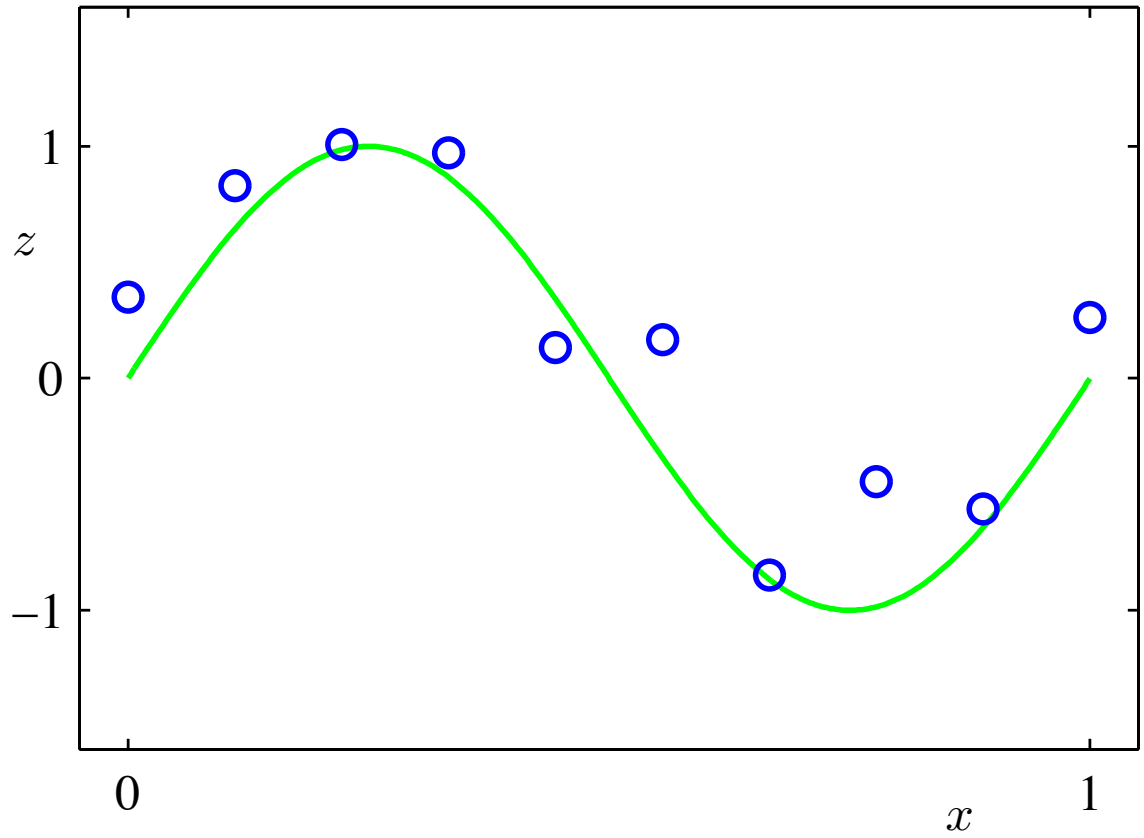
neural networks: introduction

Patrick van der Smagt

A noisy real-valued function

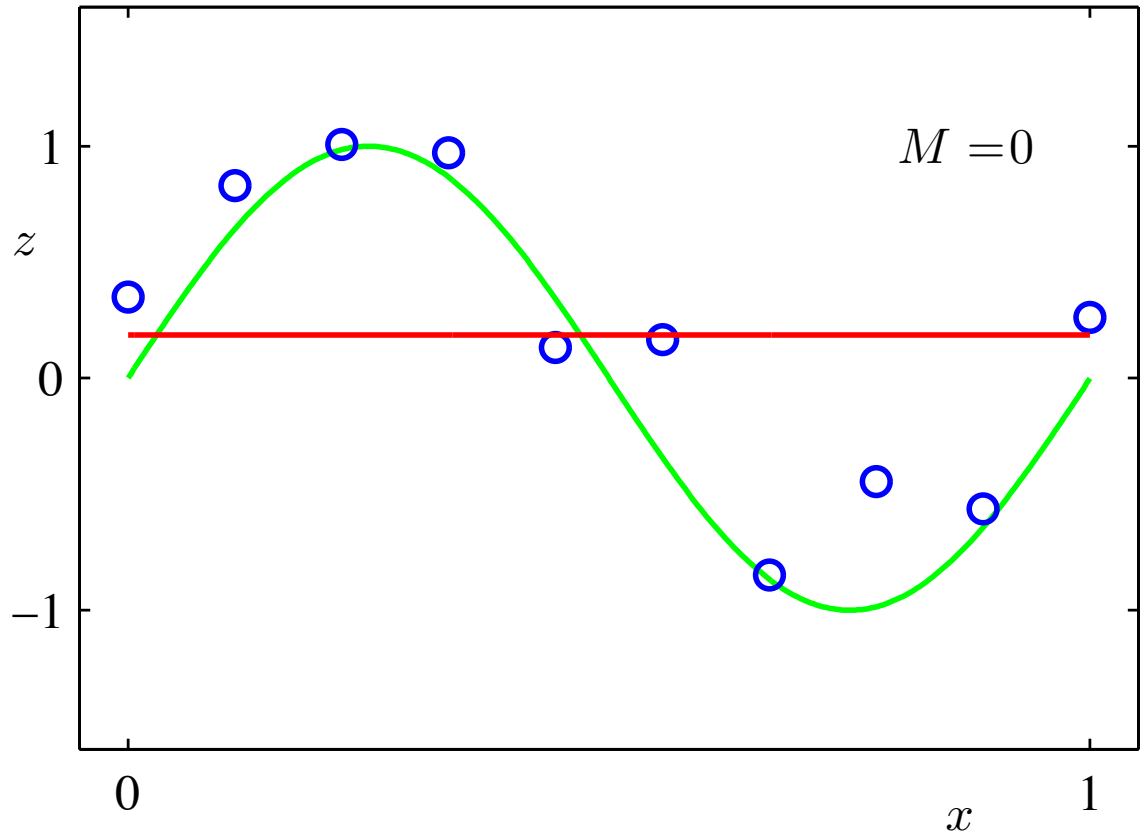


$$\text{inputs: } \boldsymbol{X} = (x_1, \ldots, x_N)^{\mathsf{T}} \tag{1}$$

$$\text{targets: } \boldsymbol{z} = (z_1, \ldots, z_N)^{\mathsf{T}}, \quad z_i = h(x_i) + \epsilon = \sin(2\pi x_i) + \epsilon \tag{2}$$

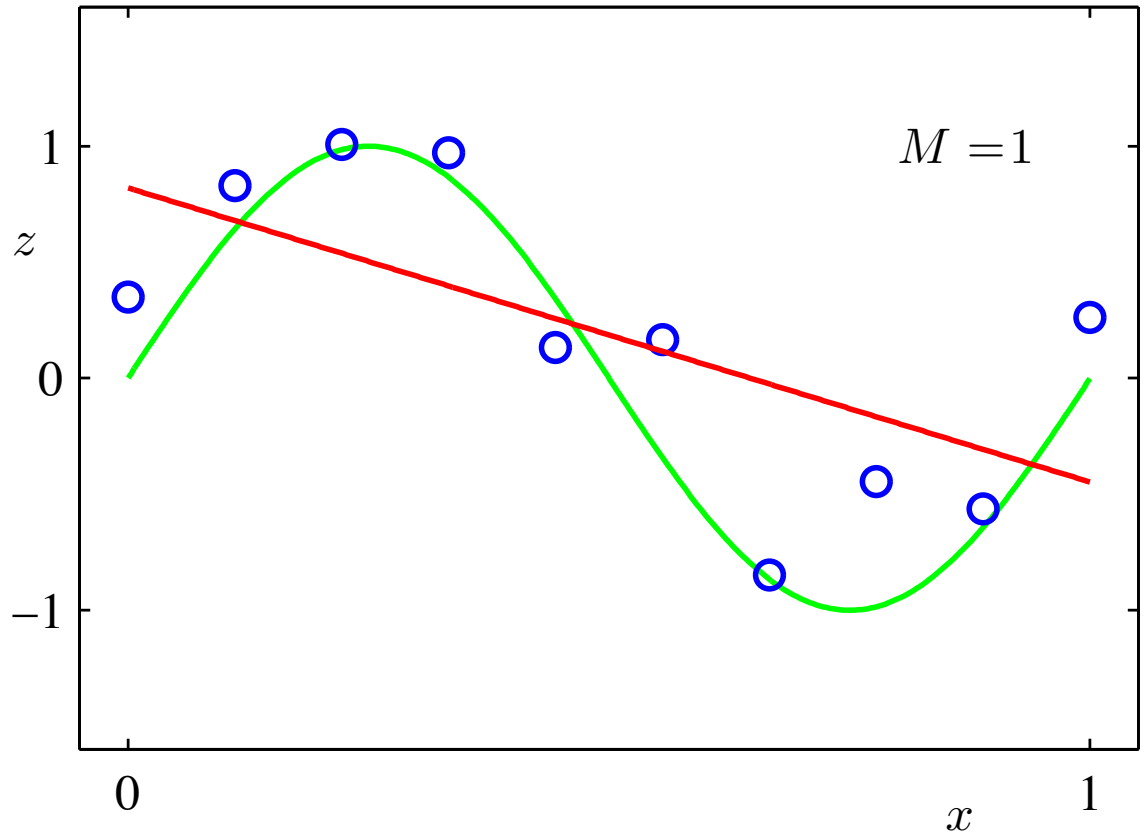
These figures are from C. Bishop: Pattern Recognition and Machine Learning

Model: 0th order polynomial



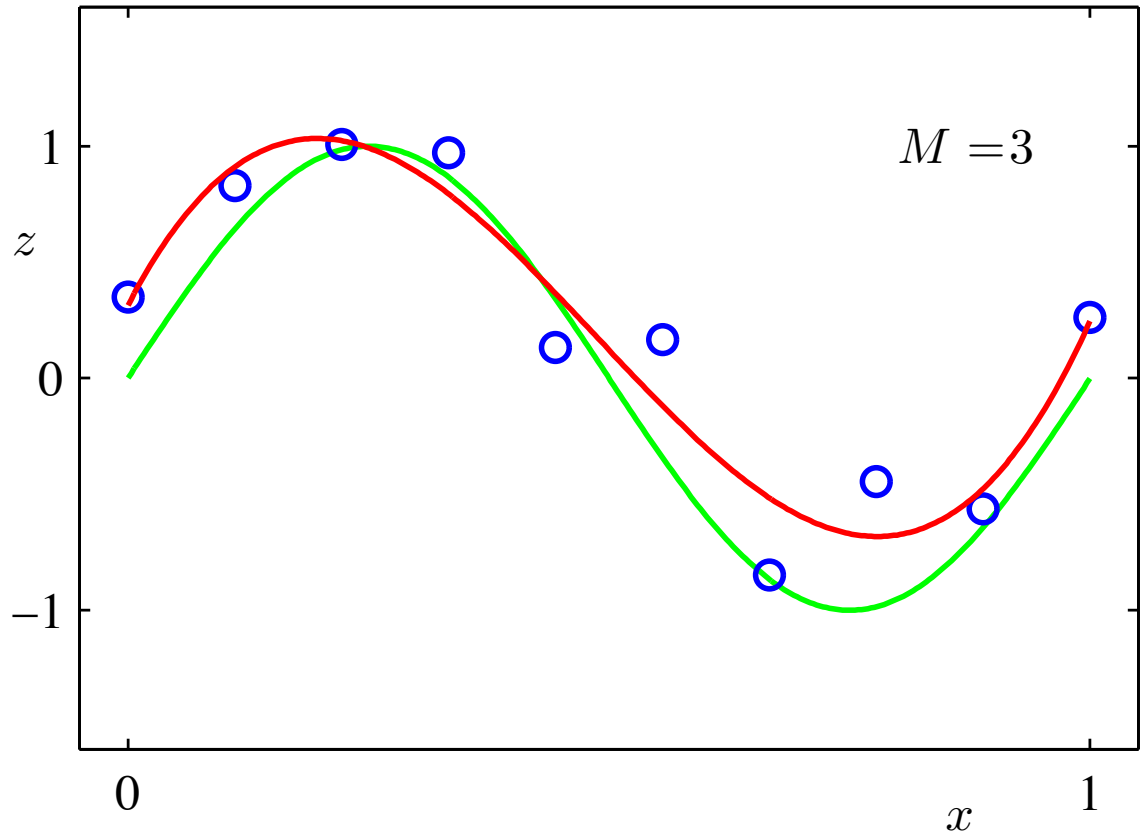
$$y(x, \boldsymbol{w}) = w_0$$

Model: 1st order polynomial



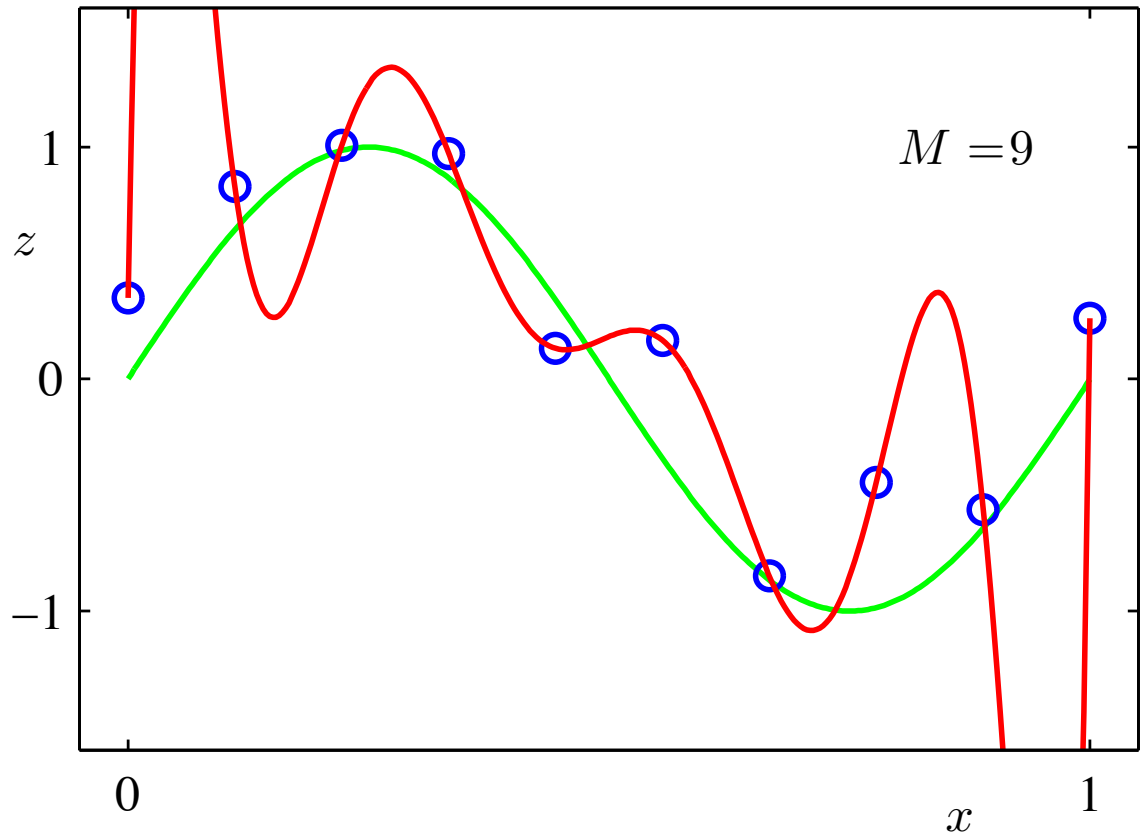
$$y(x, \boldsymbol{w}) = w_0 + w_1 x$$

Model: 3rd order polynomial



$$y(x, \boldsymbol{w}) = w_0 + w_1 x + w_2 x^2 + w_3 x^3$$

Model: 9th order polynomial



$$y(x, \boldsymbol{w}) = \sum_{j=0}^{M} w_j x^j$$

Problem Definition

We have input vectors $\boldsymbol{x}$ and associated output values z. We want to describe the underlying functional relation.

What about the following simple model?

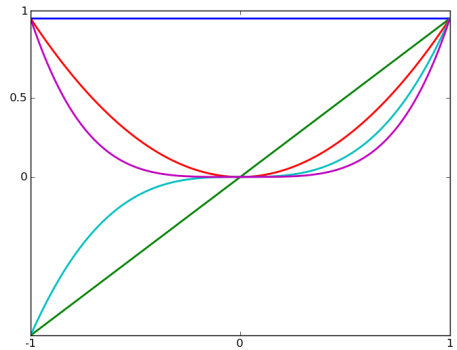
$$y(\boldsymbol{x}, \boldsymbol{w}) = w_0 + \sum_{j=1}^{M-1} w_j \phi_j(\boldsymbol{x}) = \boldsymbol{w}^{\mathsf{T}} \boldsymbol{\phi}(\boldsymbol{x}) \tag{3}$$

where

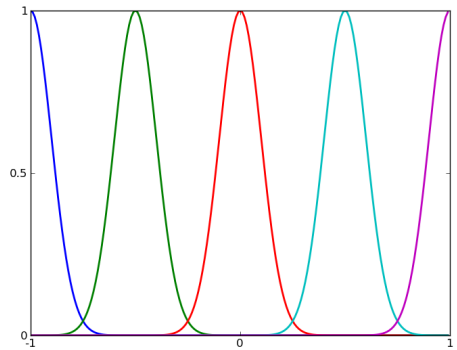
$\boldsymbol{\phi}$	basis function	—	many choices, can be nonlinear
w_0	bias	—	equivalent to defining $\phi_0 \equiv 1$

It is linear in $\boldsymbol{w}$! Nothing new if you know Taylor expansion, Fourier transform, wavelets...

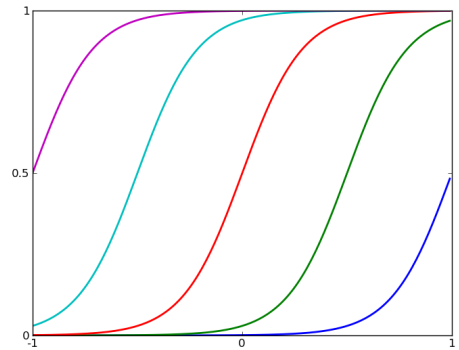
Typical Basis Functions



polynomials



Gaussians

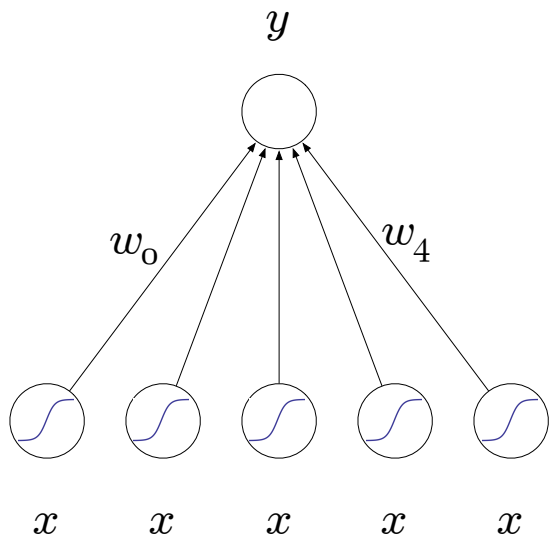


"sigmoids"
(=S-shaped curves)

towards nonlinear systems

How do we find optimal basis functions?

The above system could be graphically represented like this
(this is *not* a graphical model)

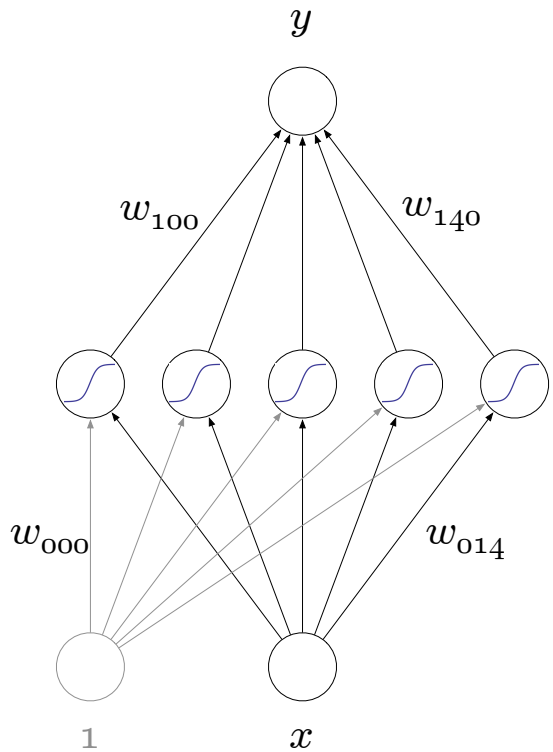


where the arrows represent weights and the circles the basis functions.

Why don't we let the system find the optimal basis functions?

the multi-layered perceptron = neural network

We can extend the system with an additional layer, and get

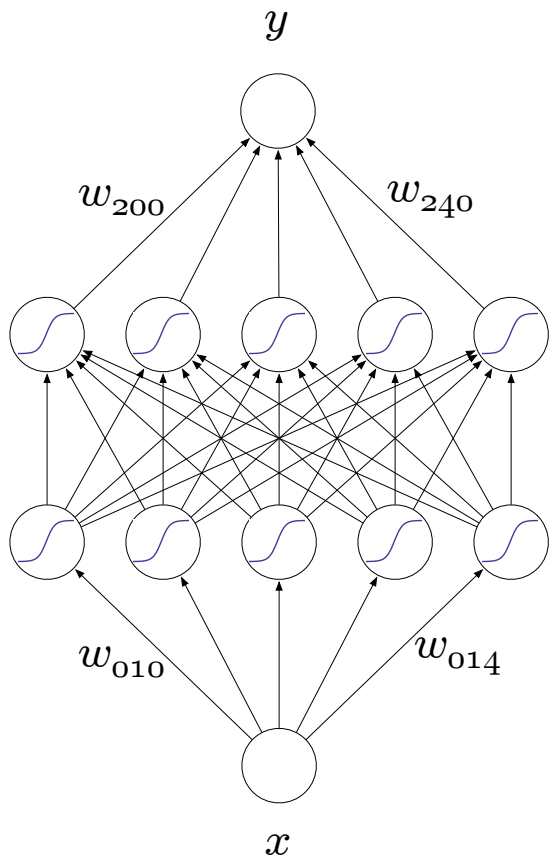


(for simplicity, the constant "1" is usually and from now on not depicted. But you always need it!)

We have generalised to $y(\boldsymbol{x}, \boldsymbol{w}_0, \boldsymbol{w}_1) = \boldsymbol{w}_1^\mathsf{T} \boldsymbol{\phi}(\boldsymbol{w}_0^T \boldsymbol{x})$

the deep neural network

We can continue adding more hidden layers



and get a deep neural network: $y(\boldsymbol{x}, \boldsymbol{w}) = \boldsymbol{w}_2^{\mathsf{T}} \boldsymbol{\phi}\big(\boldsymbol{w}_1^{\mathsf{T}} \boldsymbol{\phi}(\boldsymbol{w}_0^T \boldsymbol{x})\big)$.

how do we find W?

Remember that our data set consists of targets $\boldsymbol{z} = (z_1, z_2, \ldots, z_N)$ and corresponding input vectors $\boldsymbol{X} = (\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_N)$.

We measure random variable z as

$$z = y(\boldsymbol{x}, \boldsymbol{w}) + \boldsymbol{\epsilon} \qquad [\epsilon\text{: Gaussian, zero mean}] \tag{4}$$

Then the log likelihood is

$$\ln p(\boldsymbol{z} \mid \boldsymbol{X}, \boldsymbol{w}) \propto -\frac{1}{2} \sum_{n=1}^{N} \big(z_n - y(\boldsymbol{x}_n, \boldsymbol{w})\big)^2 \tag{5}$$

We call the negative log likelihood the loss $\mathcal{L}(\boldsymbol{w})$ aka $E(\boldsymbol{w})$.

how do we find W?

In the maximum-likelihood solution we therefore minimise

$$E = \sum_{n=1}^{N} \big(z_n - y(\boldsymbol{x}_n, \boldsymbol{w})\big)^2$$

There is one difference w.r.t. linear regression: $E(\boldsymbol{w})$ is no longer convex!

How can this be minimised? The minimum is located where its gradient is 0. So one typically minimises by using the gradient:

$$\boldsymbol{w}_{i+1} = \boldsymbol{w}_i - \alpha \nabla E$$

How do we compute the gradient ∇E? Back propagation does this.

one slide on back-propagation

A general rule to optimally find the weights $\boldsymbol{w}$ was not discovered until 1974 (Paul Werbos) or 1985 (LeCun) and 1986 (Rumelhart *et al.*): **back propagation**.

The idea: you need to compute the gradient $\partial E/\partial w_{ijk}$. To do so, compute the residual $y - z$ at the output, and propagate that back the the neurons in the layers below. From that you can then compute the gradient.

algorithm for backprop ("on-line" aka "stochastic" learning)

back-propagation algorithm:

initialise the weights
repeat
  for each training sample $(\boldsymbol{x}, \boldsymbol{z})$ **do**
  begin
    compute $\boldsymbol{o} = y(\boldsymbol{w}, \boldsymbol{x})$ (forward pass)
    calculate residual $\delta_{kj} = \boldsymbol{z} - \boldsymbol{o}$ at the output units
    for all k:
      propagate δ_{kj} back one layer by $\delta_{k-1,i} = \sum_j \delta_{kj} w_{k-1,i,j}$
    update the weights using $\partial E / \partial w_{kij} = \delta_{kj}\, \phi'(\cdot)\, x_i$
  end
  (this is called one epoch)
until stopping criterion satisfied