Neural Nets - Backpropagation

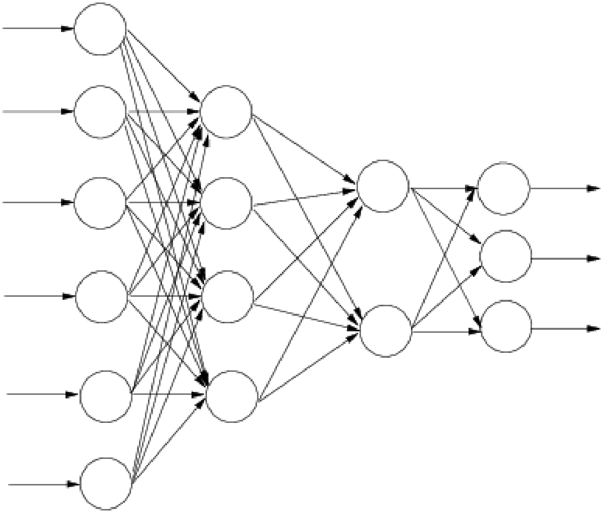
Aprendizagem Automática Avançada

Luís Correia

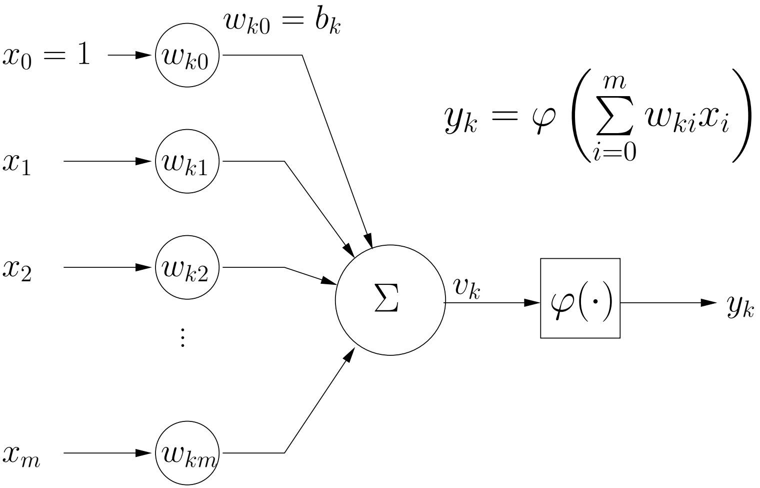
Ciências - ULisboa

MLP example

forward; complete connection



the additive model of artificial neuron



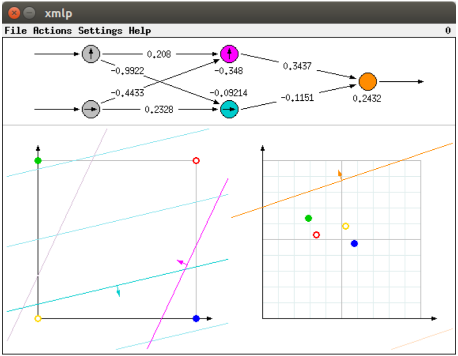
$$y_k = \varphi\left(\sum_{i=0}^{m} w_{ki} x_i\right)$$

MLP

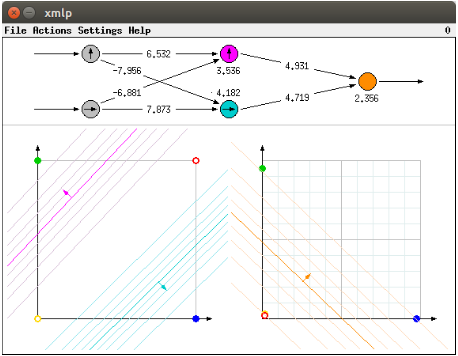
- **multi layer perceptron (MLP)**
- general designation of feedforward Artificial Neural Nets (ANN)
- nets with several (> 1) layers of neurons (without feedback)
- signals propagate layer by layer from input to output
- most common learning model:
  - **error backpropagation algorithm**, or
  - **backpropagation**, or
  - **backprop**
  - errors propagate layer by layer from output to input
- **backprop** is a generalisation of LMS

MLP learning

before learning



after learning



source: https://borgelt.net/mlpd.html

to solve in 1st TP

backprop in two words

two steps:

- **forward step** - input vector presented and output is computed
  - by forward propagation
  - with constant weights
- **backward step** - weights are adjusted from an error signal
  - error = difference between real output and desired output
  - error is propagated backwards adjusting weights layer by layer

backprop in MLP - activation function I/II

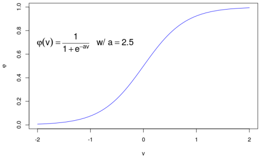
requirements

non linear and differentiable activation function

- typically sigmoid, such as the logistic function

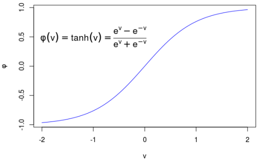
$$y_j = \frac{1}{1+\exp(-av_j)} \quad \text{with} \quad a > 0$$

y_j - neuron j output
v_j - induced local field
(weighted sum of all inputs)



- or the hyperbolic tangent function

$$y_j = a \tanh(bv_j) \quad \text{with} \quad a, b > 0$$

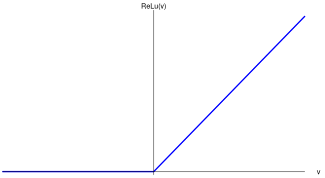


backprop in MLP - activation function II/II

relaxing the differentiable requirement (in a single point)

- introducing the rectified linear unit (ReLU) function

$$y_j = \begin{cases} 0, & \text{if } v_j < 0 \\ v_j, & \text{if } v_j \geq 0 \end{cases}$$



notes on backprop in MLP

- if activation function was linear, a MLP would reduce to a single neuron(!)
- hidden layers (neither input nor output layers)
    - allow to increase complexity of processing/classification
    - ... and also increase difficulty of analysis
- *backprop* is a computationally efficient learning algorithm

notation

to formalise backprop

n iteration
$\mathscr{E}(n)$ error energy $(1/2 \sum_j e_j^2(n))$
$e_j(n)$ error of neuron j
$y_j(n)$ real output
$d_j(n)$ desired output
$w_{ji}(n)$ synapse weight:

output of i to input of j

$\Delta w_{ji}(n)$ learning correction
$\varphi(\cdot)$ activation function

$v_j(n)$ induced local field $(\sum_i w_{ji} x_i)$
$x_{ji}(n)$ i-th input of neuron j
$o_k(n)$ k-th output of the network
η learning rate
L network depth; $l = 0, 1, \ldots, L$
layer
m_l number of neurons in layer l

m_0 input size; m_L output size;
usually $m_L = M$

learning goal in MLP

supposing a neuron j in the output layer

$$e_j(n) = d_j(n) - y_j(n)$$

error energy:

$$\mathscr{E}(n) = \frac{1}{2}\sum_{j \in C} e_j^2(n)$$

with C as the set of neurons in the output layer
designating N as the number of examples in the training set, the average error energy is:

$$\mathscr{E}_{av} = \frac{1}{N}\sum_{n=1}^{N} \mathscr{E}(n)$$

learning: minimise $\mathscr{E}_{av}$ by adjusting weights

backprop algorithm
introduction

- weights adjusted for each input vector as a function of the error
    - repeat for all examples of the training set: 1 epoch
- repeat for several epochs until stopping criteria
- the average individual weight change is an *estimate* of the change needed to minimise the cost function over the training set, $\mathscr{E}_{av}$

backprop detailed
part I/III

the induced local field of output neuron j is

$$v_j(n) = \sum_{i=0}^{m} w_{ji}(n) y_i(n)$$

its output:

$$y_j(n) = \varphi_j(v_j(n))$$

alike LMS, backprop applies a correction $\Delta w_{ji}(n)$ to the weights, proportional to the partial derivative $\partial \mathscr{E}(n) / \partial w_{ji}(n)$
applying the chain rule:

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = \frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \frac{\partial v_j(n)}{\partial w_{ji}(n)}$$

backprop detailed

part II/IIII

these hold:

$$\frac{\partial \mathscr{E}(n)}{\partial e_j(n)} = e_j(n) \qquad \frac{\partial y_j(n)}{\partial v_j(n)} = \varphi_j'(v_j(n))$$

$$\frac{\partial e_j(n)}{\partial y_j(n)} = -1 \qquad \frac{\partial v_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

replacing in the equation of $\partial \mathscr{E}(n)/\partial w_{ji}(n)$, we obtain:

$$\frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi_j'(v_j(n))y_i(n)$$

backprop detailed

part III/III

the weight correction is (delta rule):

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)}$$

or, by replacing the error derivative equation:

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

in which $\delta_j(n)$ is the local gradient defined by:

$$\begin{aligned} \delta_j(n) &= -\frac{\partial \mathscr{E}(n)}{\partial v_j(n)} = \frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= e_j(n) \varphi'_j(v_j(n)) \end{aligned}$$

backprop detailed

part III/III

the weight correction is (delta rule):

$$\Delta w_{ji}(n) = -\eta \frac{\partial \mathscr{E}(n)}{\partial w_{ji}(n)}$$

notice: correction is contrary to the gradient

or, by replacing the error derivative equation:

$$\Delta w_{ji}(n) = \eta \delta_j(n) y_i(n)$$

notice: $w_{ji}(n+1) = w_{ji}(n) + \Delta w_{ji}(n)$

in which $\delta_j(n)$ is the local gradient defined by:

$$\begin{aligned} \delta_j(n) &= -\frac{\partial \mathscr{E}(n)}{\partial v_j(n)} = \frac{\partial \mathscr{E}(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= e_j(n) \varphi_j'(v_j(n)) \end{aligned}$$

backprop comments

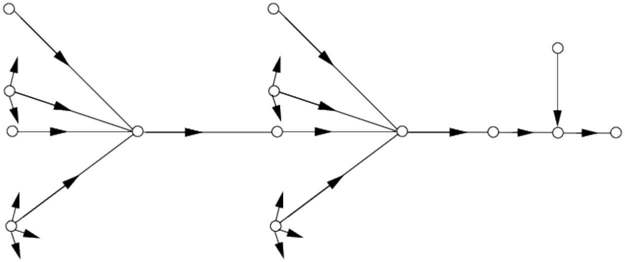
output layer

- local gradient provides sign and magnitude of corrections to do in synaptic weights
- corrections only depend on local error and activation function derivative
  - usually activation function is identical in all neurons of the network
- output error computation is immediate

backprop hidden layers

introduction

- error computation in hidden neuron is harder than in output layer
  - its influence in any single output neuron is shared with other hidden neurons
- to determine how to change weights of a hidden neuron according to its share of influence in the result is a credit-assignment problem



backprop hidden layers

local gradient calculus I/V

given

k an output neuron

j an hidden layer (just before output) neuron

the local gradient may be rewritten:

$$\begin{aligned}\delta_j(n) &= -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} \frac{\partial y_j(n)}{\partial v_j(n)} \\ &= -\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} \varphi_j'(v_j(n))\end{aligned}$$

backprop hidden layers

local gradient calculus II/V

since:

$$\mathscr{E}(n) = \frac{1}{2}\sum_{k\in C} e_k^2(n) \quad (k \text{ is an output neuron})$$

we obtain:

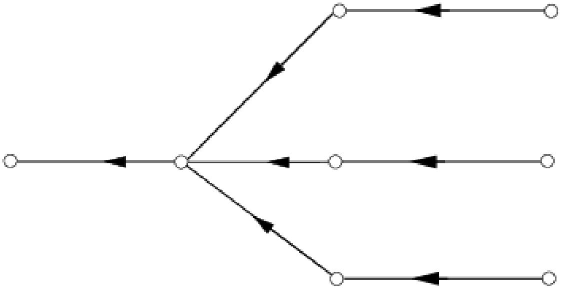
$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial y_j(n)}$$

notice indexes j and k

backprop hidden layers

hidden layer neuron error contributions

to obtain the error derivative of a hidden layer neuron
we need to take into account the contributions of all the output neurons to which it is connected



backprop hidden layers

local gradient calculus III/V

by the chain rule:

$$\frac{\partial \mathscr{E}(n)}{\partial y_j(n)} = \sum_k e_k \frac{\partial e_k(n)}{\partial v_k(n)} \frac{\partial v_k(n)}{\partial y_j(n)}$$

given that:

$$e_k(n) = d_k(n) - y_k(n) = d_k(n) - \varphi_k(v_k(n))$$

we obtain:

$$\frac{\partial e_k(n)}{\partial v_k(n)} = -\varphi_k'(v_k(n))$$

backprop hidden layers

local gradient calculus IV/V

noting that:

$$v_k(n) = \sum_{j=0}^{m} w_{kj}(n) y_j(n) \quad \text{with } m \text{ as \#inputs of neuron } k$$

the derivative in order to $y_j(n)$ becomes:

$$\frac{\partial v_k(n)}{\partial y_j(n)} = w_{kj}(n)$$

backprop hidden layers

local gradient calculus V/V

replacing in $\partial\mathscr{E}(n)/\partial y_j(n)$ we get:

$$\begin{aligned}\frac{\partial\mathscr{E}(n)}{\partial y_j(n)} &= -\sum_k e_k(n)\varphi_k'(v_k(n))w_{kj}(n)\\ &= -\sum_k \delta_k(n)w_{kj}(n)\end{aligned}$$

where:

$$\delta_k(n) = e_k(n)\varphi_k'(v_k(n))$$

finally, replacing this in the rewritten $\delta_j(n)$ equation:

$$\delta_j(n) = \varphi_j'(v_j(n))\sum_k \delta_k(n)w_{kj}(n)$$

backprop hidden layers

further hidden...

- the factor

$$\sum_k \delta_k(n) w_{kj}(n)$$

  can be considered as the error of a hidden layer neuron

- it is easy to see that this analysis can be recursively applied to neurons of previous hidden layers. . .

enters the activation function

logistic I/III

$$y_j(n) = \varphi_j(v_j(n)) = \frac{1}{1 + \exp(-av_j(n))}$$

with $a > 0$ and $-\infty < v_j(n) < \infty$
resulting in $0 \leq y_j \leq 1$
the derivative in order to $v_j(n)$ is:

$$\varphi_j'(v_j(n)) = \frac{a \exp(-av_j(n))}{[1 + \exp(-av_j(n))]^2}$$

since $y_j(n) = \varphi_j(v_j(n))$ the derivative can be written as

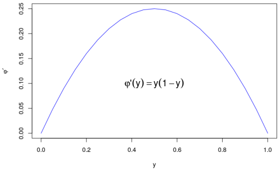
$$\varphi_j'(v_j(n)) = ay_j(n)[1 - y_j(n)]$$

enters the activation function

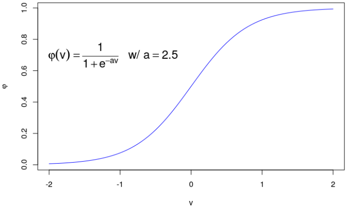
logistic II/III

analysing

$$\varphi_j'(v_j(n)) = a y_j(n)[1 - y_j(n)]$$



- maximum when $y_j(n) = 0, 5$
- minimum when $y_j(n) = 0$ or $y_j(n) = 1$
- since $\Delta w \propto \varphi_j'(v_j(n))$
  - max variation in the intermediate zone of the sigmoid
  - least variation with a *saturated* neuron



- this feature provides stability to the backprop learning algorithm

enters the activation function

logistic III/III - simplifications

for an output neuron, $y_k(n) = o_k(n)$, therefore the local gradient is:

$$\begin{aligned}\delta_k(n) &= \varphi_k'(v_k(n))e_k(n) \\ &= a o_k(n)[1 - o_k(n)][d_k(n) - o_k(n)]\end{aligned}$$

for a hidden layer neuron:

$$\begin{aligned}\delta_j(n) &= \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \\ &= a y_j(n)[1 - y_j(n)] \sum_k \delta_k(n) w_{kj}(n)\end{aligned}$$

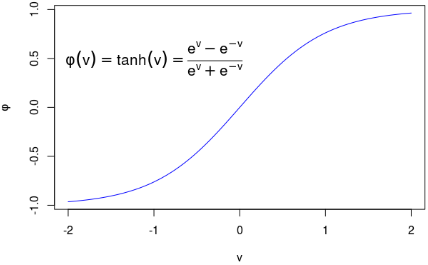
enters the activation function

hyperbolic tangent I/II

general form:

$$\varphi_j(v_j(n)) = a \tanh(bv_j(n))$$

- with $a, b > 0$ constants
- similar to the logistic function, with a different scale and a bias



its derivative is:

$$\begin{aligned}\varphi_j'(v_j(n)) &= ab\,\mathrm{sech}^2(bv_j(n)) \\ &= ab(1-\tanh^2(bv_j(n)) \\ &= \frac{b}{a}[a - y_j(n)][a + y_j(n)]\end{aligned}$$

enters the activation function

hyperbolic tangent II/II

for an output neuron:

$$
\begin{aligned}
\delta_k(n) &= e_k(n)\varphi_k'(v_k(n)) \\
&= \frac{b}{a}[d_k(n) - o_k(n)][a - o_k(n)][a + o_k(n)]
\end{aligned}
$$

for a hidden layer neuron:

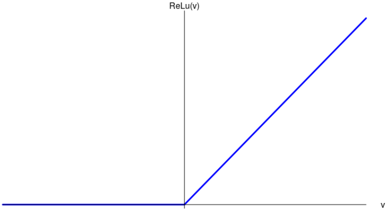
$$
\begin{aligned}
\delta_j(n) &= \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \\
&= \frac{b}{a}[a - y_j(n)][a + y_j(n)] \sum_k \delta_k(n) w_{kj}(n)
\end{aligned}
$$

enters the activation function

rectified linear unit (ReLU) I/II

general form:

$$\varphi_j(v_j(n)) = \begin{cases} 0, & \text{if } v_j(n) < 0 \\ v_j(n), & \text{if } v_j(n) \geq 0 \end{cases}$$



- very simple piecewise linear function

its derivative is:

$$\varphi_j'(v_j(n)) = \begin{cases} 0, & \text{if } v_j(n) < 0 \\ 1, & \text{if } v_j(n) \geq 0 \end{cases}$$

enters the activation function

ReLU II/II

for an output neuron:

$$\begin{aligned}\delta_k(n) &= e_k(n)\varphi_k'(v_k(n)) \\ &= \begin{cases} 0, & \text{if } v_k(n) < 0 \\ d_k(n) - o_k(n), & \text{if } v_k(n) \geq 0 \end{cases}\end{aligned}$$

for a hidden layer neuron:

$$\begin{aligned}\delta_j(n) &= \varphi_j'(v_j(n)) \sum_k \delta_k(n) w_{kj}(n) \\ &= \begin{cases} 0, & \text{if } v_j(n) < 0 \\ \sum_k \delta_k(n) w_{kj}(n), & \text{if } v_j(n) \geq 0 \end{cases}\end{aligned}$$

backpropagation happy ending

- with both logistic, hyperbolic tangent and ReLU:

backprop does not need to compute

- the activation function
- nor its derivative!

learning config I/IV

- η controls learning rate

$$
\begin{aligned}
\eta \quad << \quad & \text{smooth trajectory} \\
& \text{but slow learning} \\
\eta \quad >> \quad & \text{quick learning,} \\
& \text{but unstable (possible oscillations)}
\end{aligned}
$$

workaround with a momentum parameter α:

$$\boxed{\Delta w_{ji}(n) = \alpha \Delta w_{ji}(n-1) + \eta \delta_j(n) y_i(n)}$$

generalised delta rule - delta rule as a particular case when $\alpha = 0$

learning config II/IV

generalised delta rule alternative formulation:

$$\Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t)$$

but

$$\delta_j(t) y_i(t) = -\partial \mathscr{E}(t) / \partial w_{ji}(t)$$

therefore:

$$\Delta w_{ji}(n) = -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial \mathscr{E}(t)}{\partial w_{ji}(t)}$$

learning config III/IV

- $\Delta w_{ji}(n)$ is the sum of a series
  - **to converge** we need $0 \leq |\alpha| < 1$ (usually $\alpha > 0$)
- momentum tends to **accelerate descend** in a downward segment
  - $\partial \mathscr{E}(n)/\partial w_{ji}(n)$ with identical signs in consecutive iterations makes $\Delta w_{ji}(n)$ grow (i.e. $w_{ji}(n)$ adjustment grows)
- momentum tends to **stabilise trajectories** with signal changes
  - $\partial \mathscr{E}(n)/\partial w_{ji}(n)$ with opposed signs in successive iterations produces small $\Delta w_{ji}(n)$ (i.e. small $w_{ji}(n)$ adjustments)

learning config IV/IV

- momentum may contribute to better learning and to avoid local minima
- η may not be constant and
  we may have as many instances η_{ji} as synapses
- with $\eta_{ji} = 0$ there is no learning in that synapse

improved momentum

adaptive momentum estimation (ADAM)

computes exponentially decaying average of past gradients
similar to a heavy ball with friction instead just heavy (momentum)

$$\begin{aligned} m_t &= \beta_1 m_{t-1} + (1-\beta_1)\partial\mathscr{E}(t)/\partial w_{ji}(t) \\ v_t &= \beta_2 v_{t-1} + (1-\beta_2)(\partial\mathscr{E}(t)/\partial w_{ji}(t))^2 \end{aligned}$$

m_t: estimate of 1st moment (mean), corrected for bias: $\hat{m}_t = m_t/(1-\beta_1)$
v_t: estimate of second moment (variance), corrected: $\hat{v}_t = v_t/(1-\beta_2)$

$$\Delta w_{ji}(n) = -\frac{\eta}{\sqrt{\hat{v}_t}+\epsilon}\hat{m}_t$$

with suggested values of $\beta_1 = 0.9, \beta_2 = 0.999, \epsilon = 10^{-8}$

sequential training

- an epoch is the presentation of all the set of training examples

$$(\mathbf{x}(1), \mathbf{d}(1)), \ldots, (\mathbf{x}(N), \mathbf{d}(N))$$

1. first example $\mathbf{x}(1)$ presented at the input and
a learning cycle is performed:
forward signal propagation and given $\mathbf{d}(1)$, the error backpropagation
2. the process is repeated to complete the epoch $(\mathbf{x}(N), \mathbf{d}(N))$

- if error not low enough the epoch is repeated in a different (random) order of examples
  - randomisation order makes learning stochastic and avoids limit cycles (around local minima)

batch training

all weights are updated only once given epoch examples

- cost function for an epoch as the average square error

$$\mathscr{E}_{av} = \frac{1}{2N} \sum_{n=1} N \sum_{i \in C} e_j^2(n)$$

- adjustment of synaptic weights according to delta rule:

$$\Delta w_{ji} = -\eta \frac{\partial \mathscr{E}_{av}}{\partial w_{ji}} = -\frac{\eta}{N} \sum_{n=1}^{N} e_j(n) \frac{\partial e_j(n)}{\partial w_{ji}}$$

  with $\partial e_j(n) / \partial w_{ji}$ as before

- Δw_{ji} adjustment is done once incorporating all the individual errors of the epoch

sequential vs. batch

sequential training

- \+ simple implementation
- \+ being stochastic may avoid local minima
- \+ in general good results
- \- hard to model theoretically

batch training

- \+ easy to determine convergence conditions
- \+ easy to parallelise
- \- doesn't take advantage of redundant examples

stopping criteria

- *backprop has converged when the absolute variation of the average squared error per epoch is small enough*
  - "small" maybe from 1% downto 0,1%, or even 0,01%...

better alternative:

test generalisation capability and (early) stop when it peaks

backprop heuristics I/IX

1. *sequential vs. batch*
    - sequential simpler and more robust
2. *maximise the information content*
   of each example in the training set:
    - use an example producing a large error
    - use an example radically different from others

   possible problems:

   distorted example distribution
   learning of extreme cases may worsen backprop behaviour

backprop heuristics II/IX

3. *activation function*

   learning is faster with antisymmetric (odd) activation function:

   $$\varphi(-v) = -\varphi(v)$$

   logistic is not, but hyperbolic tangent $\varphi(v) = a\tanh(bv)$ yes!

   suggested parameter values:

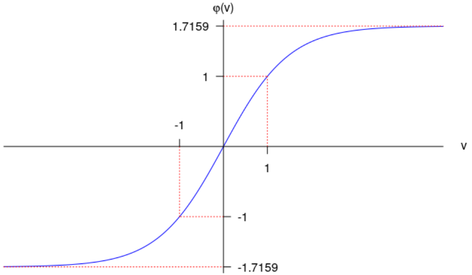
   $a = 1,7159 \qquad b = \frac{2}{3}$

which result in:

$\varphi(1) = 1$, $\varphi(-1) = -1$

$\varphi'(0) = ab = 1,1424 \simeq 1$

and max $\varphi''(v)$ at $v = 1$



backprop heuristics III/IX

4. *desired outputs (d_k)* should fall within and at some distance (ϵ) of the limiting values of φ
   for the case of the hyperbolic tangent
   in the positive value $+a$:
   $$d_k = a - \epsilon$$
   in the negative value $-a$:
   $$d_k = -a + \epsilon$$
   if $a = 1,7159$ then with $\epsilon = 0,7159$ we obtain d_j as ± 1
   well in the linear zone of φ

backprop heuristics IV/IX

5. *input normalisation*
    - each input should have **average null** or close to it
      supposing all inputs positive, all weights in the input layer would tend to vary together
    - input variables should **not be correlated**
      PCA may be useful for this
    - scale variables so that their **covariances** $(\sigma_{X_i X_j})^\dagger$ are **similar**
      $\Rightarrow$ learning speed of synaptic weights is similar

$^\dagger \sigma_{X_i X_j} = E[(X_i - E[X_i])(X_j - E[X_j])]$

backprop heuristics V/IX

6. *initialisation* initial weight values **not very large** (saturation) **nor very small** (saddle point in antisymmetric φ)

   assuming null average inputs and unit variance:

$$\mu_X = E[X_i] = 0 \quad \wedge \quad \sigma_X^2 = E[(X_i - \mu_{X_i})^2] = E[X_i^2] = 1 \quad \forall i$$

   an supposing non correlated inputs:

$$E[X_i X_j] = \begin{cases} 1 & \text{for} \quad i = j \\ 0 & \text{for} \quad i \neq j \end{cases}$$

backprop heuristics VI/IX

initialisation (cont.)

supposing initial random uniform synaptic weights with null average

$$\mu_w = E[w_{ji}] = 0 \quad \text{for all } (j, i)$$

and variance

$$\sigma_w^2 = E\left[(w_{ji} - \mu_w)^2\right] = E[w_{ji}^2]$$

the induced local field average is

$$\mu_v = E[v_j] = E\left[\sum_{i=1}^{m} w_{ji} x_i\right] = \sum_{i=1}^{m} E[w_{ji}] E[x_i] = 0$$

(assuming null bias $v_j = \sum_{i=1}^{m} w_{ji} x_i$)

backprop heuristics VII/IX

initialisation (cont.) and the induced local field variance is

$$\begin{aligned}
\sigma_v^2 &= E\left[(V_j - \mu_v)^2\right] = E[V_j^2] \\
&= E\left[\sum_{i=1}^{m}\sum_{k=1}^{m} w_{ji} w_{jk} x_i x_k\right] \\
&= \sum_{i=1}^{m}\sum_{k=1}^{m} E[w_{ji} w_{jk}] E[x_i x_k] \\
&= \sum_{i=1}^{m} E[w_{ji}^2] \\
&= m\sigma_w^2
\end{aligned}$$

backprop heuristics VIII/IX

for $\varphi = \tanh$
a good strategy for initialisation of the synaptic weights is

$$\sigma_v = 1$$

so that, with a and b as previously defined weights will fall along the linear zone of the sigmoid
and therefore

$$\sigma_w = m^{-1/2}$$

weight variance reciprocal of the number of synapses of a neuron

backprop heuristics IX/IX

7. *learning clues*
   - use information about the function to learn to accelerate backprop
     ex: invariance, symmetry properties
8. *learning rate*
   - for learning to occur at the same rhythm in all neurons, η should be smaller in the last layers (usually with higher local gradients)
     neurons with more inputs should have smaller η (inversely proportional to the square root of the nr. of synapses) - adjustment

backprop summary

1. *initialisation*
   if no information available, choose synaptic weights with random uniform distribution, null average and adequate variance to be in the linear zone of the sigmoid
2. *training*
   present an epoch while performing the learning steps for each example
3. *iterations*
   repeat epoch presentation with a different random order until stopping criterium

   adjust α and η (decreasing their values) with the number of iterations

pros & cons of backprop I/II

connectionism (local computation)

- \+ metaphor for biological networks
- \+ (potential) graceful degradation
- \+ easy to parallelise
- \- unrealistic in face of natural neurons

+ universal approximator of functions

+ computationally efficient (linear with W)

pros & cons of backprop II/II

\+ **robustness** - small perturbations only produce small estimate variations

\- **slow convergence**

- stochastic algorithm - local gradient may not point to the minimum of the error surface
- possible *overshoot* with high gradient in a single weight, or
- small adjustments in flat error surfaces

\- **local minima** - backprop can get stuck

\- **scaling** - time may grow exponentially with nr. inputs

- try to simplify connections (and avoid fully connected MLP in complex problems)

final comments to backprop

- advantages stem from:
  - local learning method
  - efficient in computing local error derivatives
- sequential (stochastic) mode vastly more used for simplicity

non-linear neurons

- each one establishes a separation hyperplane
- the combination of all hyperplanes is iteratively adjusted to separate example patterns minimising classification error on average

references

- Haykin, S. S. (2009). Neural networks and learning machines. Pearson Education.

heuristics to adjust η

to accelerate convergence

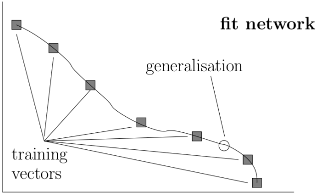
1. one η for each parameter
2. each η should be allowed to change in each iteration
3. when error derivative in order to one parameter maintains sign in consecutive iterations, increase respective η
4. when error derivative in order to one parameter alternates sign in consecutive iterations, decrease respective η

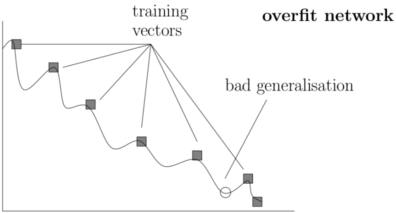
. . . meta-learning. . .

Back

generalisation I/III

an ANN generalises if, for an input vector not used in training the result is correct (or nearly so. . .)





generalisation II/III

3 main factors influence generalisation capability:

- size and representativity of the training set
- network architecture
- problem complexity

the last is not controllable!
about the others:

- define the architecture and try to obtain a good training set
- define the training set and try to obtain a good architecture

generalisation III/III

maintaining the architecture...
rule of thumb:

$$N = O\left(\frac{W}{\epsilon}\right)$$

where

N	size of the training set
W	total of free parameters
ϵ	admissible error

to determine the architecture we need to go deeper...

cross-validation I/II

- statistics technique very used in machine learning

data set divided into:

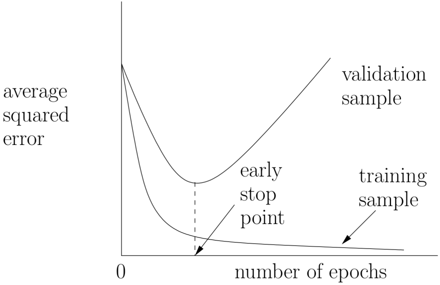
- *training sets*
  - training set
    - used to fit the model
  - validation set
    - used to estimate generalisation, or prediction error of the model to avoid overfit
- *test set*
  - used to assess the generalisation error of the obtained model

test set is not used for training!

cross-validation II/II

early stop method

1. perform a few epochs of training
2. with parameters fixed, present the validation set and measure the error for each of its examples
3. repeat from 1, until validation error increases



- rule of thumb examples partition:

  80% for training set + 20% for validation set
  (see also $k-$*fold cross validation*)

Back

function approximation I/IV

universal approximation theorem (UAP)

let $\varphi(\cdot)$ be a non constant, bounded and monotonous-increasing function. Let I_{m_0} be the m_0-dimensional unit hypercube $[0,1]^{m_0}$. The space of the continuous functions in I_{m_0} is denoted by $C(I_{m_0})$. Then, given any function $f \in C(I_{m_0})$ and $\epsilon > 0$, 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

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

as an approximation of function $f(\cdot)$; meaning,

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

for all $x_1, x_2, \ldots, x_{m_0}$ within the input space

function approximation II/IV

- the *universal approximation theorem* says that a single hidden layer is sufficient for a MLP to compute a ϵ approximation to a given training set $x_1, x_2, \ldots, x_{m_0}$ and the corresponding desired output $f(x_1, \ldots, x_{m_0})$

however,

- it does not guarantee optimality of training time, or generalisation

function approximation III/IV

an error risk bound of using a MLP with m_0 input nodes and m_1 hidden layer neurons is [Barron, 1992]:

$$R \leq O\left(\frac{C_f^2}{m_1}\right) + O\left(\frac{m_0 m_1}{N} \log N\right)$$

where $C_f \approx$ smoothness of function to learn f
expresses a tradeoff between

- *accuracy of best approximation* (1st term) - $\mathbf{m_1}$ **must be large** (see also UAP)
- *accuracy of empirical fit to the approximation* (2nd term) - **ratio** $\mathbf{m_1/N}$ **must be small** (for N constant m_1 should be small)

function approximation IV/IV

with *2 hidden layers* learning is more manageable:

- the first hidden layer extracts *local features* - partition of input and learning of local features to each one
- the second hidden layer extracts *global features* - learns global features combining outputs of neurons in the first hidden layer for a particular region of the output space

Back