Neural networks

Doc. RNDr. Iveta Mrázová, CSc.

Department of Theoretical Computer Science and Mathematical Logic

Faculty of Mathematics and Physics

Charles University in Prague

Neural networks

– Multi-layered neural networks –

Doc. RNDr. Iveta Mrázová, CSc.

Department of Theoretical Computer Science and Mathematical Logic

Faculty of Mathematics and Physics

Charles University in Prague

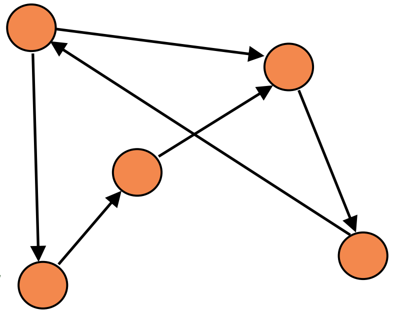
Multi-layered neural networks (1)

D: **A neural network** is a 6-tuple $M=(N,C,I,O,w,t)$, where:

- N is a finite non-empty set of neurons,
- $C \subseteq N \times N$ is a non-empty set of oriented inter-connections among neurons
- $I \subseteq N$ is a non-empty set of input neurons
- $O \subseteq N$ is a non-empty set of output neurons
- $w: C \rightarrow \mathbf{R}$ is a weight function
- $t: N \rightarrow \mathbf{R}$ is a threshold function

($\mathbf{R}$ is the set of all real numbers)

(N,C) is called the inter-connection graph of M



Multi-layered neural networks (2)

D: **A Back-Propagation network (BP-network)** ***B*** is a neural network with a directed acyclic inter-connection graph. Its set of neurons consists of a sequence of $l + 2$ pairwise disjunctive non-empty subsets called layers.

- The first layer called **the input layer** is the set of all input neurons of ***B***, these neurons have no predecesors in the inter-connection graph; their input value x equals their output value.
- The last layer called **the output layer** is the set of all output neurons of ***B***; these neurons are those having no successors in the inter-connection graph.
- All other neurons called hidden neurons are grouped in the remaining l **hidden layers**.

Back-propagation training algorithm (1)

Aim: find such a set of weights that ensure that for each input vector, the output vector producedd by the network is the same as (or sufficiently close to) the desired output vector

The actual or desired output values of the hidden neurons are not specified by the task.

- For a fixed, finite training set, the objective function represents the total error between the desired and actual outputs of all the output neurons in the BP-network taken for all the training patterns.

Back-propagation training algorithm (2)
The Error function

- corresponds to the difference between the actual and desired network output:

$$E = \frac{1}{2} \sum_{p} \sum_{j} \left(y_{j,p} - d_{j,p} \right)^2$$

desired output

patterns

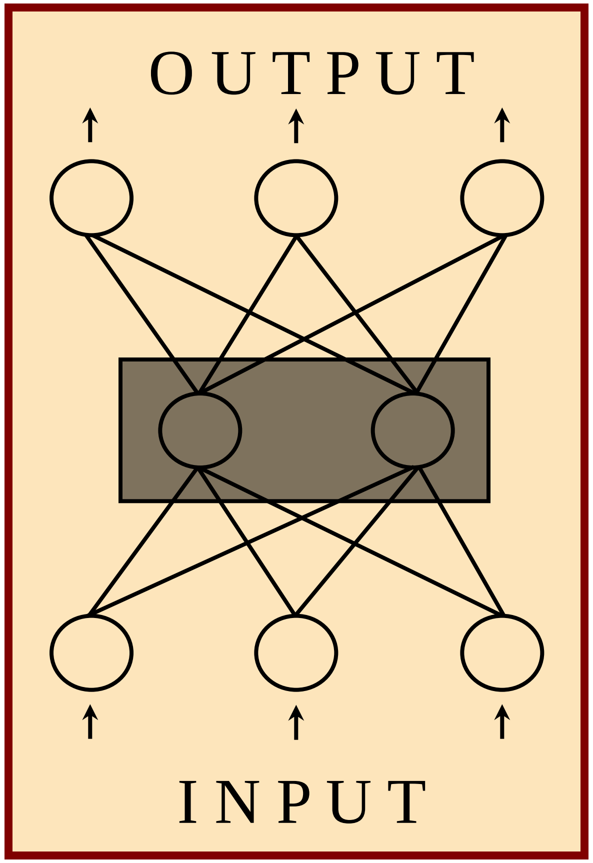
output neurons

actual output

- during training, this difference should be minimized on the given training set

⇨ **the back-propagation training algorithm**

Multi-layered neural networks
(BP-networks)



- produce the actual output for the presented input pattern
- compare the actual and desired outputs
- adjust the weights and thresholds
  - against the gradient of the error function
  - from the output layer towards the input layer

BP-networks: adjustment rules (1)

Synaptic weights are adjusted against the gradient:

$$w_{ij}(t+1) = w_{ij}(t) + \Delta_E w_{ij}(t)$$

$\Delta_E w_{ij}(t)$ .... the change of $\boldsymbol{w_{ij}}$ to minimize $\boldsymbol{E}$

error at network output

$$\Delta_E w_{ij} = -\frac{\partial E}{\partial w_{ij}} = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial \xi_j}\frac{\partial \xi_j}{\partial w_{ij}}$$

potential of the neuron j

actual output

connection weight

BP-networks: adjustment rules (2)

Weight adjustment in the output layer:

$$\Delta_E w_{ij} \cong -\frac{\partial E}{\partial w_{ij}} = -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial \xi_j}\frac{\partial \xi_j}{\partial w_{ij}} =$$

$$= -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial \xi_j}\frac{\partial}{\partial w_{ij}}\sum_{i'} w_{i'j}\, y_{i'} =$$

$$= -\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial \xi_j} y_i = -\frac{\partial E}{\partial y_j} f'(\xi_j)\, y_i =$$

$$= -(y_j - d_j)\, f'(\xi_j)\, y_i = \delta_j\, y_i$$

BP-networks: adjustment rules (3)

Weight adjustment in hidden layers:

$$\Delta_E w_{ij} \cong -\frac{\partial E}{\partial w_{ij}} = -\left(\sum_k \frac{\partial E}{\partial \xi_k}\frac{\partial \xi_k}{\partial y_j}\right)\frac{\partial y_j}{\partial \xi_j}\, y_i =$$

$$= -\left(\sum_k \frac{\partial E}{\partial \xi_k}\frac{\partial}{\partial y_j}\sum_{j'} w_{j'k}\, y_{j'}\right)\frac{\partial y_j}{\partial \xi_j}\, y_i =$$

$$= -\left(\sum_k \frac{\partial E}{\partial \xi_k} w_{jk}\right)\frac{\partial y_j}{\partial \xi_j}\, y_i =$$

$$= \left(\sum_k \delta_k w_{jk}\right) f'(\xi_j)\, y_i = \delta_j\, y_i$$

BP-networks: adjustment rules (4)

The derivative of the sigmoidal transfer function is:

$$f'(\xi_j) = \lambda\, y_j\, (1 - y_j)$$

Weight adjustment according to:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha \delta_j y_i + \alpha_m \left(w_{ij}(t) - w_{ij}(t-1)\right)$$

- where

$$\delta_j = \begin{cases} (d_j - y_j)\lambda\, y_j (1 - y_j) & \text{for an output neuron} \\ \left(\sum_k \delta_k w_{jk}\right)\lambda\, y_j (1 - y_j) & \text{for a hidden neuron} \end{cases}$$

Back-propagation training algorithm (1)

Step 1: Initialize the weights to small random values

Step 2: Present a new training pattern in the form of:

[input $\vec{x}$, desired output $\vec{d}$]

Step 3: Calculate actual output

in each layer, the activity of neurons is given by:

$$y_j = f(\xi_j) = \frac{1}{1+e^{-\lambda\xi_j}} \quad , \text{ where } \quad \xi_j = \sum_i y_i w_{ij}$$

The activities expressed in this way form then the input of the following layer.

Back-propagation training algorithm (2)

Step 4: Weight adjustment

The weight adjustment starts at the output layer and proceeds back towards the input layer according to:

$$w_{ij}(t+1) = w_{ij}(t) + \alpha\delta_j y_i + \alpha_m\left(w_{ij}(t) - w_{ij}(t-1)\right)$$

$$\delta_j = \begin{cases} (d_j - y_j)\,\lambda\, y_j(1-y_j) & \text{for an output neuron} \\ \left(\sum_k \delta_k w_{jk}\right)\lambda\, y_j(1-y_j) & \text{for a hidden neuron} \end{cases}$$

$w_{ij}(t)$ ……….. weight from neuron i to neuron j in time t

α , α_m ……….. learning rates, resp. moment ($0 \leq \alpha , \alpha_m \leq 1$)

ξ_j , resp. δ_j …... potential, resp. local error on neuron j

k …………….. index for the neurons from the layer above the neuron j

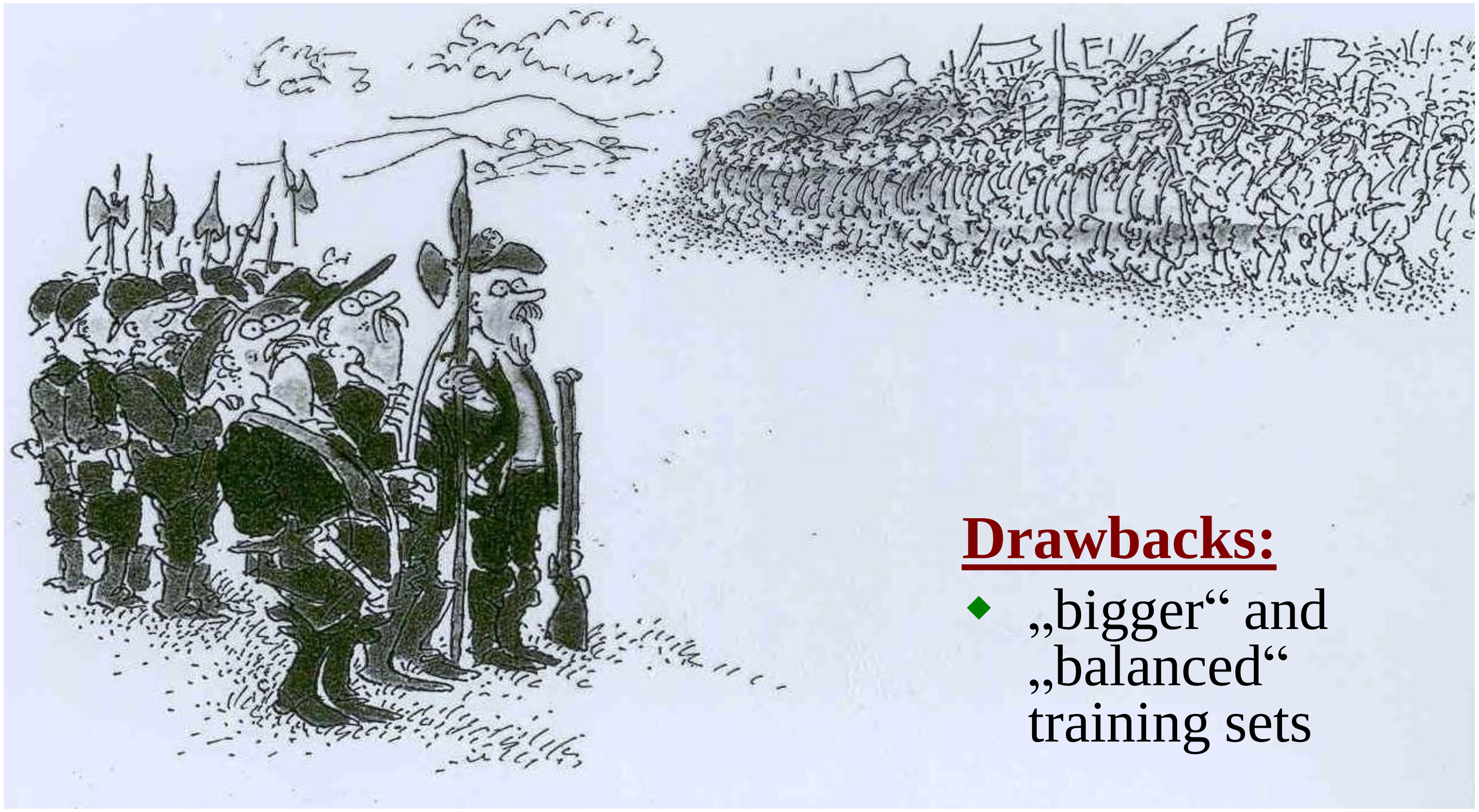
λ …………….. slope of the transfer function

Step 5: Repeat by going to Step 2

BP-networks: analysis of the model

- One of the most often used models
- Simple training algorithm
- Relatively good results
- **Drawbacks:**
  - Internal knowledge representation - „black box“
  - error function (knowledge of the desired outputs)
    - „bigger“ and „balanced“ training sets
    - assessment of network outputs during recall
  - the number of neurons and generalization capabilities
    - pruning and retraining

BP-networks: analysis of the model



Drawbacks:

- „bigger“ and „balanced“ training sets

Back-propagation training algorithm: speeding-up the training process (1)

- **The standard back-propagation training algorithm is rather slow**

  → a malicious selection of network parameters can make it even slower

- **The learning problem for artificial neural networks is NP-complete in the worst case**

  → computational complexity grows exponentially with the number of the variables

  → despite of that the standard back-propagation performs often better than many „fast learning algorithms“

  - especially when the task achieves a realistic level of complexity and the size of the training set goes beyond a critical threshold

Back-propagation training algorithm: speeding-up the training process (2)

Algorithms speeding-up the training process:

- **Keeping a fixed network topology**
- **Modular networks**
  - considerable improvement of network approximation abilities
- **Adjustment of both the parameters** (weights, thresholds, etc.) **and the network topology**

Back-propagation training algorithm: initial weight selection (1)

- **The weights should be uniformly distributed over the interval** ***$< -\alpha_m , \alpha_m >$***
- **Zero mean value**
  - leads to an expected zero value of the total input to each node in the network (potential)
- **The derivative of the sigmoidal transfer function is reached its maximum for zero** ***(~ 0.25)***
  - Larger values of the backpropagated errors
  - More significant weight updates when training starts

Back-propagation training algorithm: initial weight selection (2)

Problem:

- **Too small weights paralyze learning**
  - The error backpropagated from the output layer to hidden layers is too small
- **Too large weights lead to saturation of neurons and slow learning** (in flat zones of the error function)

→ **Learning then stops at a suboptimal local minimum**

× the right choice of initial weights can significantly reduce the risk of getting stuck in a local minimum

Back-propagation training algorithm: initial weight selection (3)

Reduce the danger of local minima:

~ inicialize the weights with small random values

Motivation:

- **Small weight values**
  - Large weight values impact saturation of hidden neurons (too active or too passive for all training patterns) → such neurons are incapable of further training (the derivative of the transfer function – sigmoid – is almost zero)
- **Random weight values**
  - The goal is to „break the symmetry“ → hidden neurons should specialize in the recognition of different featurers

Back-propagation training algorithm: initial weight selection (4)

IDEA:

- **The potential of a hidden neuron is given by:**

$$\xi = w_0 + w_1x_1 + \ldots + w_nx_n$$

x_i … the activity of the *i*-th neuron from the preceding layer

w_i …the weight from the *i*-th neuron from the preceding layer

- **Expected value of the potential for hidden neurons:**

$$E\{\xi_j\} = E\left\{\sum_{i=0}^{n} w_{ij} x_i\right\} = \sum_{i=0}^{n} E\{w_{ij}\} E\{x_i\} = 0$$

  - the weights are independent of the input patterns
  - the weights are random variables with zero mean

Back-propagation training algorithm: initial weight selection (5)

IDEA - continue:

- **The variance of the potential ξ is given by:**

$$\sigma_\xi^2 = E\left\{\left(\xi_j\right)^2\right\} - \underbrace{E^2\left\{\left(\xi_j\right)\right\}}_{=0} = E\left\{\left(\sum_{i=0}^{n} w_{ij}\; x_i\right)^2\right\} - 0 =$$

$$= \sum_{i,k=0}^{n} E\left\{\left(w_{ij} w_{kj}\; x_i\; x_k\right)\right\} =$$

mutual independence for all ***j***

$$= \sum_{i=0}^{n} E\left\{\left(w_{ij}\right)^2\right\} E\left\{\left(x_i\right)^2\right\}$$

Back-propagation training algorithm: initial weight selection (6)

IDEA - continue:

- **Further, we assume:** training patterns are normalized and from the interval *< 0 , 1 >* . Then:

$$E\left\{(x_i)^2\right\} = \int_0^1 x_i^2\, dx = \frac{x^3}{3}\Big|_0^1 = \frac{1}{3}$$

- Assumed that the weights of the hidden neurons are also random variables with a zero mean and uniformly distributed in the interval *[-a,a]*, then:

$$E\left\{(w_{ij})^2\right\} = \int_{-a}^{a} w_{ij}^2 \cdot \frac{1}{2a}\, dw_{ij} = \frac{w_{ij}^3}{6a}\Big|_{-a}^{a} = \frac{a^2}{3}$$

- *N* … number of weight leading to the considered neuron

Back-propagation training algorithm: initial weight selection (7)

IDEA - continue:

- **Standard deviation will thus correspond to:**

$$A = \sigma_{\xi} = \sqrt{N}\,\frac{a}{3} \qquad \left(\rightarrow\ a = A\frac{3}{\sqrt{N}} \right)$$

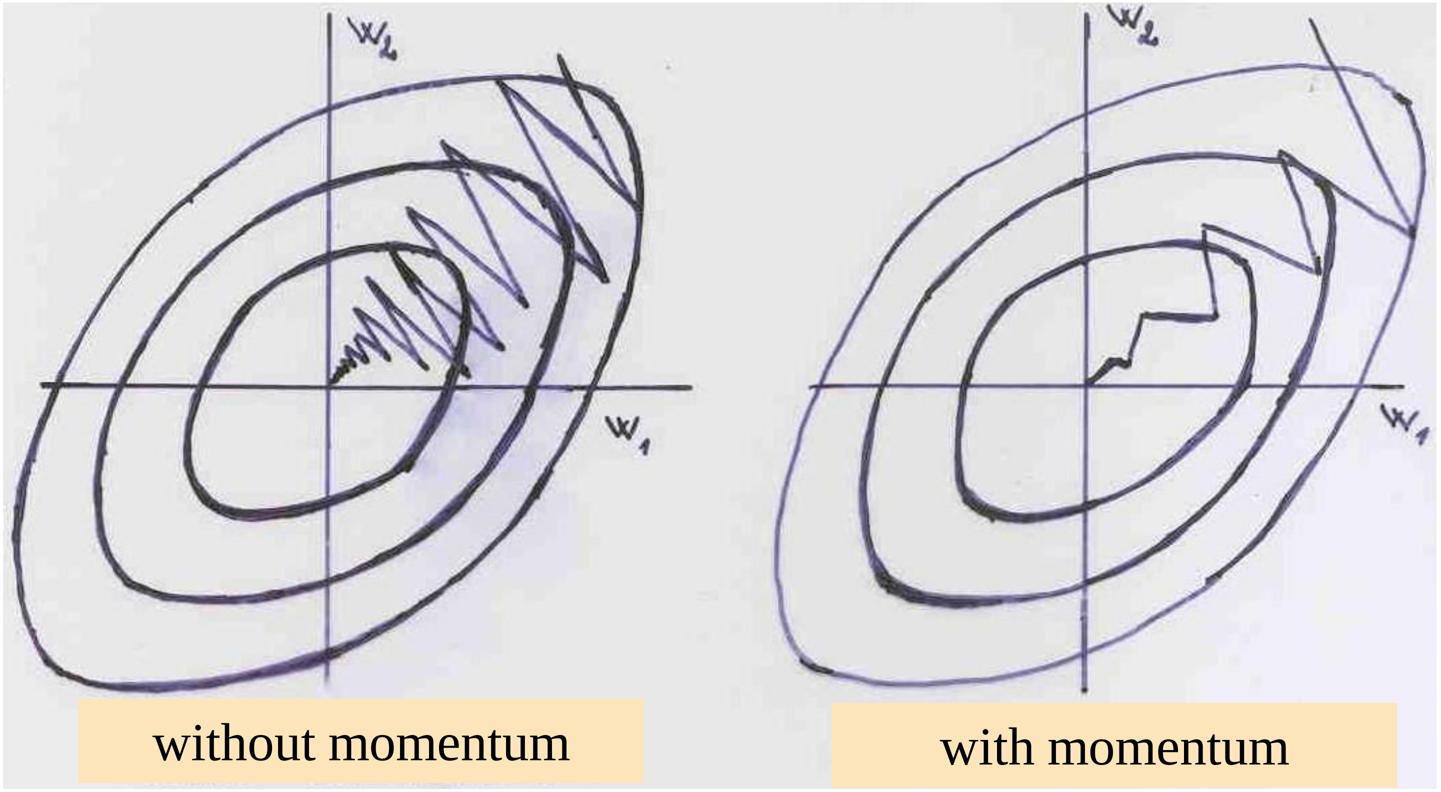
- **Neuron potential should be a random variable with the standard deviation** ***A*** (that is moreover independent of the number of weights leading to this neuron);
- **Select initial weights (roughly) from the interval:**

$$\left[-\frac{3}{\sqrt{N}} \cdot A, \frac{3}{\sqrt{N}} \cdot A \right]$$

- especially for ***A = 1*** large gradient (i.e., quick learning)

Back-propagation training algorithm with momentum (1)

Minimization of the error function with the gradient method



without momentum

with momentum

Back-propagation training algorithm with momentum (2)

- When the minimum of the error function for a given task lies in a „narrow valley,“ following the gradient direction can lead to wide oscillations of the search process
- **Solution:** **introduce a momentum term**
  - a weighted average of the current gradient and the previous correction direction is computed at each step

    → **Inertia ~ could help to avoid excessive oscillations in „narrow valleys of the error function“**

Back-propagation training algorithm with momentum (3)

- For a network with ***n*** different weights ***w_1, … , w_n*** the correction of ***w_k*** in time ***i + 1*** is given by:

$$\Delta w_k(i+1) = -\alpha \frac{\partial E}{\partial w_k(i)} + \alpha_m \Delta w_k(i) =$$

$$= -\alpha \frac{\partial E}{\partial w_k(i)} + \alpha_m \left(w_k(i) - w_k(i-1) \right)$$

where: ***α*** ….. learning rate

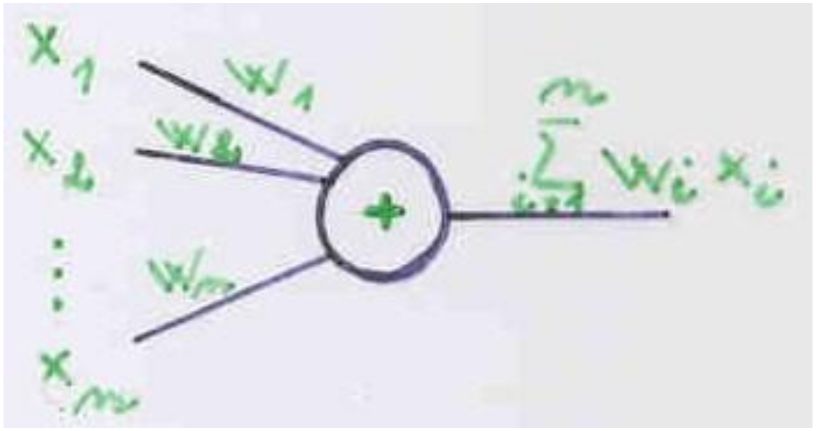
α_m … momentum rate

Back-propagation training algorithm with momentum (4)

- In order to accelerate convergence to the minimum of the error function:
  - Increase the learning rate up to an optimum value α , that still guarantees convergence of the training process
  - Introduction of the momentum rate allows to attenuate the oscillations that might occur during training
- Optimal values for α and α_m highly depend on the character of the respective learning task

Back-propagation training algorithm with momentum (5)

EXAMPLE: Linear transfer function, ***p*** patterns



X .... matrix $(p \times n)$; $\vec{d}$ ....vector

$$X = \begin{pmatrix} x_1^{(1)} & \dots & x_n^{(1)} \\ \vdots & \ddots & \vdots \\ x_1^{(p)} & \dots & x_n^{(p)} \end{pmatrix} \quad ; \quad \vec{d} = \begin{pmatrix} d^{(1)} \\ \vdots \\ d^{(p)} \end{pmatrix}$$

$\rightarrow$ MINIMIZATION OF ***E*** :

$$E = \left\| X\vec{w} - \vec{d} \right\|^2 = \left(X\vec{w} - \vec{d} \right)^T \left(X\vec{w} - \vec{d} \right) = \vec{w}^T \left(X^T X \right) \vec{w} - 2\vec{d}^T X\vec{w} + \vec{d}^T \vec{d}$$

Back-propagation training algorithm with momentum (6)

- E is a quadratic function $\longrightarrow$ the minimum can be found using gradient descent

  Interpretation: E has the form of a paraboloid in the n – dimensional space; its shape is determined by the eigenvalues of the correlation matrix $X^T X$

$\longrightarrow$ Gradient descent is most effective when the principal axes are all of the same length

$\longrightarrow$ When the axes are of very different sizes, the gradient direction can lead to oscillations

Back-propagation training algorithm with momentum (7)

- Excessive oscillations can be prevented by choosing a small value for α and a larger value for the momentum parameter α_m
  - × **too small values of α**
    - $\longrightarrow$ **the danger of local minima**
  - × **too big values of α**
    - $\longrightarrow$ **the danger of oscillations**

Back-propagation training algorithm with momentum (8)

In the nonlinear case, the gradient of the error function is almost zero in the regions far from local minima – possibility of oscillations

→ in such a case, larger learning rates could help

  → return back to „convex“ regions of the error function

Solution:

- **Adaptive learning rates**
- **Pre-processing of the training set**
  - decorrelation on input patterns (PCA, …)

Back-propagation training algorithm: strategies speeding-up the training process (1)

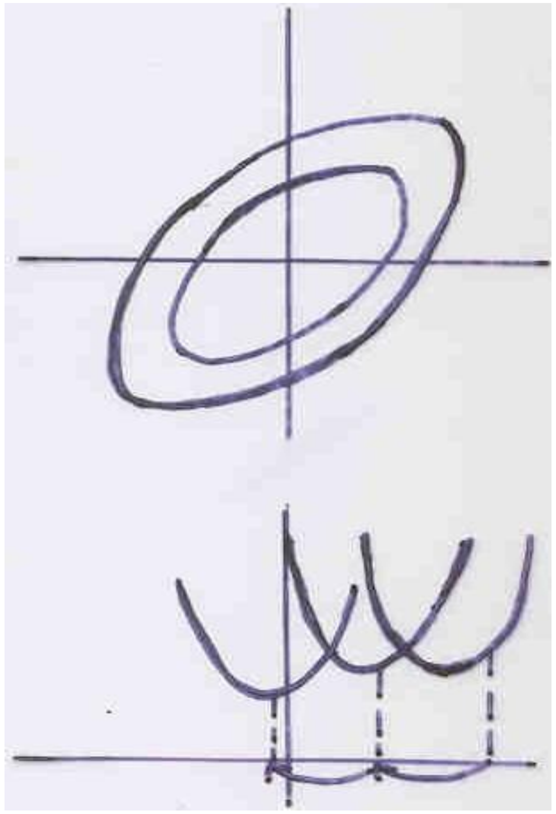
1. **Adaptive learning rates:**

   a local parametr $\boldsymbol{\alpha_i}$ for each weight $\boldsymbol{w_i}$

   weight adjustment: $$\Delta w_i = -\alpha_i \frac{\partial E}{\partial w_i}$$

- **Variants of the algorithm:**
  - Silva & Almeida
  - Delta-bar-delta
  - Super SAB

The algorithm of Silva & Almeida (1)



Assumed: the network has ***n*** weights

- Quadratic error function:

$$c_1^2 w_1^2 + c_2^2 w_2^2 + \ldots + c_n^2 w_n^2 + \sum_{i \neq j} d_{ij} w_i w_j + C$$

- The step in the ***i*** –th direction minimizes

$$c_i^2 w_i^2 + k_1 w_i + k_2$$

k_1*, *k_2 are constants, which depend on the values of the „frozen“ variables at the current iteration point (***c_i*** determine the curvature of the parabola)

The algorithm of Silva & Almeida (2)

The heuristic:

- ACCELERATE, if in two successive iterations, the sign of the partial derivative has not changed
- DECELERATE, if the sign changes

$\nabla_i E^{(k)}$ … partial derivative of the error function with respect to the weight w_i at the k –th iteration

$\alpha_i^{(0)}$ … initial learning rates $(i = 1, \ldots, n)$ initialized to a small positive value

The algorithm of Silva & Almeida (3)

- In the ***k*** –th iteration the value of the learning rate for the next step is recomputed for each weight by:

$$\alpha_i^{(k+1)} = \begin{cases} \alpha_i^{(k)} u \ , & \text{jestliže} \quad \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} > 0 \\ \alpha_i^{(k)} d \ , & \text{jestliže} \quad \nabla_i E^{(k)} \cdot \nabla_i E^{(k-1)} < 0 \end{cases}$$

- The constants ***u*** a ***d*** are set by hand with ***u > 1*** and ***d < 1***
- Weight adjustment: $\Delta^{(k)} w_i = -\alpha_i^{(k)} \nabla_i E^{(k)}$

The algorithm of Silva & Almeida (4)

Problems:

- The learning rates grow and decrease exponentially with regard to ***u*** and ***d***

→ **Problems can occur if too many acceleration steps are performed successively**

The algorithm Delta-bar-delta

- Acceleration is done with more caution than deceleration (especially from small initial weights)
- ***k*** –th iteration: $$\alpha_i^{(k+1)} = \begin{cases} \alpha_i^{(k)} + u \ , & \text{if} \ \ \nabla_i E^{(k)} \cdot \ \delta_i^{(k-1)} > 0 \\ \alpha_i^{(k)} \cdot d \ , & \text{if} \ \ \nabla_i E^{(k)} \cdot \ \delta_i^{(k-1)} < 0 \\ \alpha_i^{(k)} & \text{else} \end{cases}$$

  u, ***d*** … fixed pre-set constants

$$\delta_i^{(k)} = (1-\Phi)\nabla_i E^{(k)} + \ \Phi \ \delta_i^{(k-1)} \ , \quad \text{where } \ \Phi \text{ is a constant}$$

- Weight updates without momentum: $\Delta^{(k)} w_i \ = \ -\alpha_i^{(k)} \nabla_i E^{(k)}$

Algorithm Super SAB

- **Adaptive acceleration strategy** for the back-propagation training algorithm
  - Of order quicker that the original back-propagation algorithm
  - Relatively stable
  - Robust against the choice of initial parameters
- **Uses momentum:**
  - Accelerates convergence in flat areas of the weight space
  - In steep areas of the weight space, the momentum term curbs oscillations caused by changed signs of the gradient

Super SAB – the training algorithm (1)

α^+ ……... multiplicative constant to increase the learning rates
($\alpha^+ = 1.05$)

α^- ……..... multiplicative constant to decrease the learning rates
($\alpha^- = 2$)

α_{START} …. Initial value for the parameter $\alpha_{ij}, \forall i, j$ ($\alpha_{START} = 1.2$)

α_m ……..... momentum ($\alpha_m = 0.3$)

Step 1: set all α_{ij} to the initial value α_{START}

Step 2: perform Step (t) of back-propagation with momentum

Step 3: if the derivative (according to w_{ij}) didn´t change its sign,
increase the learning rates ($\forall w_{ij}$): $\alpha_{ij}(t+1) = \alpha^+ \cdot \alpha_{ij}(t)$

Super SAB – the training algorithm (2)

Step 4: if the derivative (according to w_{ij}) changed its sign:

- annul the previous weight change (that caused the change in the sign of the gradient): $\Delta w_{ij}(t+1) = -\Delta w_{ij}(t)$
- use smaller learning rates: $\alpha_{ij}(t+1) = \alpha_{ij}(t) \,/\, \alpha^{-}$
- and set: $\Delta w_{ij}(t+1) = 0$

(the change from the previus step will be thus not considered in the next training step)

Step 5: goto Step 2

Back-propagation training algorithm: strategies speeding-up the training process (2)

2. **Second-order algorithms:**

- Consider more information about the shape of the error function than gradient $\rightarrow$ curvature of the error function
- Second-order methods use a quadratic approximation of the error function ***E***

$\vec{w} = (w_1, \ldots, w_n)$ … weight vector of the network

$E(\vec{w})$ ………… error function

$\rightarrow$ The Taylor series approximating the error function ***E***:

$$E(\vec{w}+\vec{h}) \approx E(\vec{w}) + \nabla E(\vec{w})^T \vec{h} + \frac{1}{2} \vec{h}^T \nabla^2 E(\vec{w}) \vec{h}$$

Second-order algorithms (2)

$\nabla^2 E(\vec{w})$ ……. Hessian matrix ***(n × n)*** of second-order partial derivatives:

$$\nabla^2 E(\vec{w}) = \begin{pmatrix} \frac{\partial^2 E(\vec{w})}{\partial w_1^2} & \frac{\partial^2 E(\vec{w})}{\partial w_1 \partial w_2} & \cdots & \frac{\partial^2 E(\vec{w})}{\partial w_1 \partial w_n} \\ \frac{\partial^2 E(\vec{w})}{\partial w_2 \partial w_1} & \frac{\partial^2 E(\vec{w})}{\partial w_2^2} & \cdots & \frac{\partial^2 E(\vec{w})}{\partial w_2 \partial w_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 E(\vec{w})}{\partial w_n \partial w_1} & \frac{\partial^2 E(\vec{w})}{\partial w_n \partial w_2} & \cdots & \frac{\partial^2 E(\vec{w})}{\partial w_n^2} \end{pmatrix}$$

Second-order algorithms (3)

→ Gradient of the error function (by differentiating $\nabla E(\vec{w}+\vec{h})$):

$$\nabla E(\vec{w}+\vec{h})^T \approx \nabla E(\vec{w})^T + \vec{h}^T \nabla^2 E(\vec{w})$$

→ Gradient equal to zero (looking for the minimum of ***E***):

$$\vec{h} = -\left(\nabla^2 E(\vec{w})\right)^{-1} \nabla E(\vec{w})$$

==> **Newton´s methods:**

- **Work iteratively**
- **Weight adjustment in the *k* – th iteration according to:**

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \left(\nabla^2 E(\vec{w})\right)^{-1} \nabla E(\vec{w})$$

- **Quick convergence**

× **A problem might represent the inverse Hessian matrix**

Second-order algorithms (4)

Pseudo-Newton methods:

- Work with a „simplified form“ of the Hessian matrix
- Only the diagonal elements are computed: $\left(\frac{\partial^2 E(\vec{w})}{\partial w_i^2} \right)$
- The non-diagonal elements are all set to zero
- Weight adjustment according to:

$$w_i^{(k+1)} = w_i^{(k)} - \frac{\nabla_i E(\vec{w})}{\frac{\partial^2 E(\vec{w})}{\partial w_i^2}}$$

Second-order algorithms (5)

Pseudo-Newton methods:

- No matrix inversion necessary
- Limited computational effort involved in finding the required second partial derivatives
- Work well when the error function has a quadratic form, otherwise problems might occur since a small second-order partial derivative can lead to extremely large corrections

- **Variants of Newton´s method:**
  - Quickprop
  - Levenberg-Marquardt algrithm

The algorithm Quickprop (1)

- Takes into account also second-order information
  - × Only one-dimensional minimization steps are taken
  - → Information about the curvature of the error function in the update direction is obtained from the current and past partial derivative of the error function
- Independent optimization steps for each weight
- A quadratic one-dimensional approximation of the error function is used

The algorithm Quickprop (2)

- Weight adjustment in ***k***–th iteration according to:

$$\vec{w}_i^{(k+1)} = \vec{w}_i^{(k)} + \Delta^{(k)} w_i \quad , \text{ kde}$$

$$\Delta^{(k)} w_i = \Delta^{(k-1)} w_i \cdot \frac{\nabla_i E^{(k)}}{\nabla_i E^{(k-1)} - \nabla_i E^{(k)}}$$

<u>Assumed:</u> the error function has been computed at steps ***(k – 1)*** and ***k*** using the weight difference $\Delta^{(k-1)} w_i$ - obtained from a previous Quickprop or standard gradient descent step

The algorithm Quickprop (3)

- Weight adjustment rules can be written as:

$$\Delta^{(k)} w_i = - \frac{\nabla_i E^{(k)}}{\dfrac{\nabla_i E^{(k)} - \nabla_i E^{(k-1)}}{\Delta^{(k-1)} w_i}}$$

- The denominator is just a discrete approximation to the second-order derivative $\partial^2 E(\vec{w}) / \partial w_i^2$
- Quickprop ~ discrete pseudo-Newton method, that uses the so-called „ **SECANT STEP** “

Levenberg-Marquardt algorithm

- Quicker around the minimum of the error function
- A combination of a gradient and Newton´s method
  - Gradient only update rule

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \alpha \nabla E^{(k)}$$

  - The second order update rule

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \left(\nabla^2 E^{(k)} \right)^{-1} \nabla E^{(k)}$$

  - Levenberg – blend them together

$$\begin{aligned} \vec{w}^{(k+1)} &= \vec{w}^{(k)} - \left(\nabla^2 E^{(k)} + \lambda I \right)^{-1} \nabla E^{(k)} \\ &= \vec{w}^{(k)} - \left(H + \lambda I \right)^{-1} \nabla E^{(k)} \end{aligned}$$

Levenberg-Marquardt algorithm

- Hessian matrix can be approximated
  - for a single output

$$g_i = \frac{\partial e}{\partial w_i} = 2(y-d)\frac{\partial y}{\partial w_i}$$

$$\frac{\partial^2 e}{\partial w_i \partial w_j} = 2\left[\frac{\partial y}{\partial w_i}\frac{\partial y}{\partial w_j} + \cancel{(y-d)\frac{\partial^2 y}{\partial w_i \partial w_j}}\right]$$

  - Hence instead of H in

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \left(H + \lambda I\right)^{-1} \nabla E^{(k)}$$

it is used

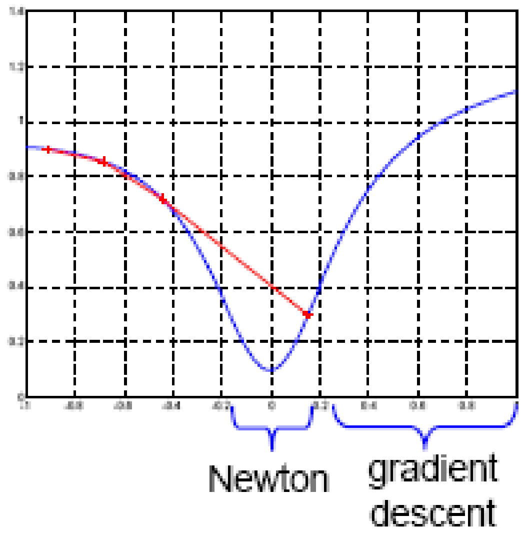
$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \left(J^{(k)\mathrm{T}} \cdot J^{(k)} + \lambda I\right)^{-1} \nabla E^{(k)}$$

where

$$J^{(k)} = \left(\frac{\partial g_1^{(k)}}{\partial w_1^{(k)}}, \ldots, \frac{\partial g_m^{(k)}}{\partial w_m^{(k)}}\right)^{\mathrm{T}}$$

Levenberg-Marquardt algorithm

- Away from the minimum, in regions of negative curvature, the Gauss-Newton approximation is not very good
- In such regions, a simple steepest-descent step is probably the best plan
- The Levenberg-Marquardt method is a mechanism for varying between steepest-descent and Gauss-Newton steps depending on how good the approximation $J^{\mathrm{T}}J$ is locally



Levenberg-Marquardt algorithm

$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \left(J^{(k)\mathbf{T}} \cdot J^{(k)} + \lambda I \right)^{-1} \nabla E^{(k)}$$

- When λ is small, the step approximates the second order Gauss-Newton method
- When λ is large, steepest-descent steps are taken.

Levenberg-Marquardt algorithm

1. Set λ = 0.001 (say)
2. Compute
$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \left(J^{(k)\mathbf{T}} \cdot J^{(k)} + \lambda I \right)^{-1} \nabla E^{(k)}$$
3. If error increases), increase λ (×10 say) and go to 2.
4. Otherwise, decrease λ (×0.1 say), update
$$\vec{w}^{(k+1)} = \vec{w}^{(k)} - \left(J^{(k)\mathbf{T}} \cdot J^{(k)} + \lambda I \right)^{-1} \nabla E^{(k)}$$

   and go to 2

Back-propagation training algorithm: speeding-up the training process (3)

3. **Relaxation methods** – weight perturbation:

- Discrete approximation to the gradient is made at each iteration by comparing the errors for the initial weights $E(\vec{w})$ and for the altered weights $\vec{w}'$ (a small perturbation $\boldsymbol{\beta}$ was added to the weight $\boldsymbol{w_i}$) – $E(\vec{w}')$
- Weight adjustment by: $\Delta w_i = - \alpha \dfrac{E(\vec{w}') - E(\vec{w})}{\beta}$
- This adjustment is repeated iteratively, randomly selecting the weight to be updated

Relaxation methods (2)

An alternative providing a faster convergence:

- Perturbation of the output of the ***i*** – th neuron $\boldsymbol{o_i}$ by $\boldsymbol{\Delta o_i}$
- The difference ***E - E´*** in the error function is computed
- If the difference is positive ***(> 0),*** the new error ***E´*** could be achieved with the output $\boldsymbol{o_i + \Delta o_i}$ for the ***i*** – th neuron
- In the case of the sigmoidal transfer function, the desired potential of the neuronu ***i*** can be determined as:

$$\sum_{k=1}^{m} w'_k \, x_k = s^{-1}\left(o_i + \Delta o_i \right)$$

$$\left(\text{for } y = s(\xi) = \frac{1}{1+\mathrm{e}^{-\xi}} \quad \text{is} \quad \xi = s^{-1}(y) = \ln \frac{y}{1-y} \right)$$

Relaxation methods (3)

- If the previous potential was $\sum_{k=1}^{m} w_k \; x_k$, then the new weights are given by:

$$w'_k = w_k \cdot \frac{s^{-1}\left(o_i + \Delta o_i \right)}{\sum_{k=1}^{m} w_k \; x_k}$$

- Weights are updapted in proportion to their size: $\frac{w'_k}{\xi'} = \frac{w_k}{\xi}$

(this can be avoided by means of stochastic factors or node perturbation can be alternated with weight perturbation)