Neural Networks and supervised learning algorithms

Maurizio Valle

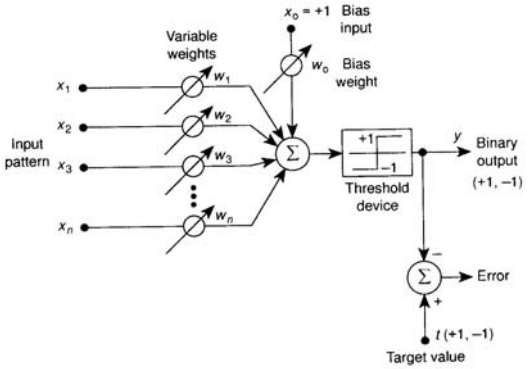
McCulloch and Pitts model [1943]

- **y = f(net), where**

$$net = \left(\sum_{i=1}^{N} X_i W_i \right) - \Theta$$

- **Unsupervised learning [Hebb, 1949]**

$$W_{j,i}(t+1) = W_{j,i}(t) + \eta X_i Y_j$$

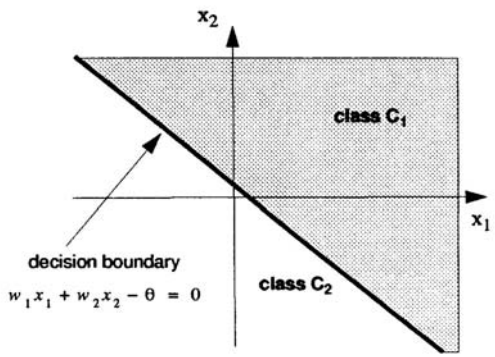


Perceptron convergence procedure [Rosenblatt, 1961]

Example

- **y > 0, if the input vector at iteration t**
  x(t) = (x_1, x_2) ∈ class C_1

- **y < 0, if the input vector at iteration t**
  x(t) = (x_1, x_2) ∈ class C_2



Perceptron convergence procedure [Rosenblatt, 1961]

- **initialise weights W_i (t=0) and Θ to small random values**
- **present new input vector x(t) (t is the iteration index)**
- **calculate the actual output y(t)**
- **adapt weights according to:**

$$W_i(t+1) = W_i(t) + \eta[d(t) - y(t)]X_i(t)$$

where 0 < η <1 and d(t) is the target value

- **go to step 2 and repeat for the next pattern**

Minsky and Papert [1969]

Perceptron can only create linear decision regions

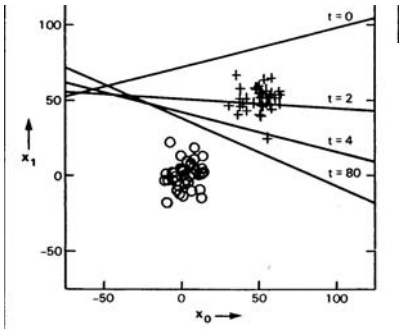
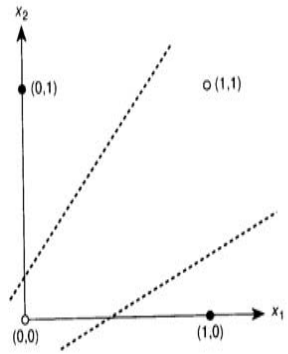


Figure 13. An example of the decision boundaries formed by the perceptron convergence procedure with two classes. Samples from class A are represented by circles and samples from class B by crosses. Lines represent decision boundaries after trials where errors occurred and weights were adapted.



Widrow-Hoff algorithm [1960] (delta or adaline or Widrow-Hoff or LMS rule)

The neuron transfer function f is made linear (ADAptive LINEar combiner, ADALINE) or replaced by a threshold-logic non-linearity

$$\varepsilon_p(t) = (d_p(t) - t_p(t))^2 = (d_p(t) - f\left(\sum_{i=1}^{N} X_i(t) W_i(t)\right))^2 \text{ pattern error index}$$

$$\varepsilon\ (t) = \sum_{p=1}^{M} \varepsilon_p(t) \text{ total error index}$$

- **Steepest descent type learning algorithm (on-line or by-pattern)**

$$\Delta W_i(t) = W_i(t+1) - W_i(t) = -\eta \frac{\partial \varepsilon_p(t)}{\partial W_i(t)}$$

Widrow-Hoff algorithm [1960] (delta or adaline or Widrow-Hoff or LMS rule)

$$\frac{\partial \varepsilon_p(t)}{\partial W_i(t)} = -\frac{1}{2}(d_p(t) - y_p(t))(X_{i,p}(t))\, f'$$

$$\Delta W_i(t) = \frac{1}{2}\eta(d_p(t) - y_p(t))(X_{i,p}(t))\, f'$$

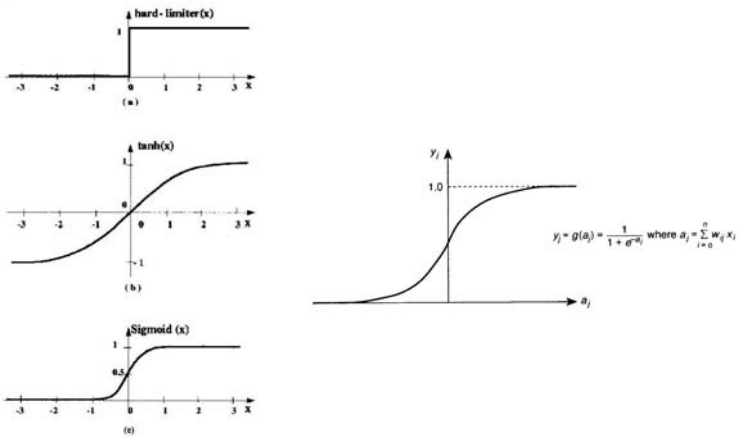
- **Steepest descent type learning algorithm (batch or by-epoch)**

$$\Delta W_i(t) = W_i(t+1) - W_i(t) = -\eta \frac{\partial \varepsilon\ (t)}{\partial W_i(t)}$$

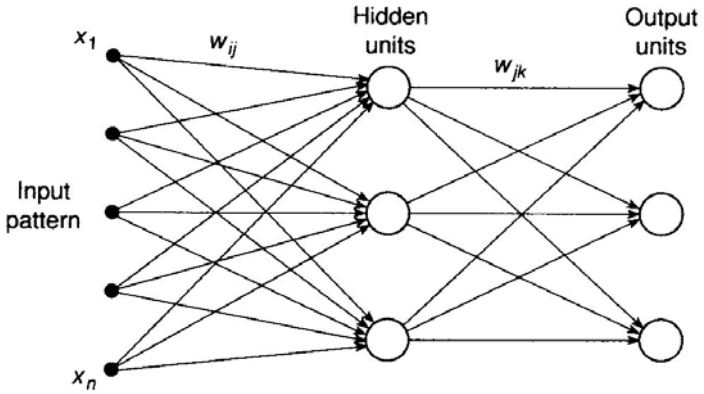
$$\frac{\partial \varepsilon\ (t)}{\partial W_i(t)} = \frac{\partial \sum_{p=1}^{M} \varepsilon_p(t)}{\partial W_i(t)} = \sum_{p=1}^{M} \frac{\partial \varepsilon_p(t)}{\partial W_i(t)} = -\frac{1}{2}\sum_{p=1}^{M} (d_p(t) - y_p(t))(X_{i,p}(t))\, f'$$

$$\Delta W_i(t) = \frac{1}{2}\eta \sum_{p=1}^{M} (d_p(t) - y_p(t))(X_{i,p}(t))\, f'$$

Neuron transfer functions



Multi-Layer Networks



Why Neural Networks?

- **learning from experience**
- **generalising from examples**
- **developing solutions faster and with less reliance on domain expertise**
- **computational efficiency**
- **non-linearity**

Identifying neural computing applications [Tarassenko 1999]

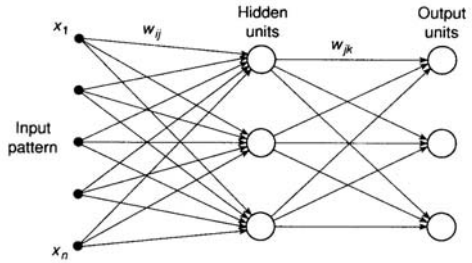
- **the solution of the problem cannot be explicitly described by an algorithm, a set of equations or a set of rules**
- **there is some evidence that an input-output mapping exists between a set of input variables x and corresponding output data y, such that y=f(x). The form of f however is not known.**
- **there should be a large amount of data available, i.e. many different examples with which to train the network.**

Typical business functions and neural computing application areas [Tarassenko 1999]

	Fault diagnosis	Condition monitoring	Forecasting	Signal/Image analysis	Pattern detection in databases	Industrial inspection	Fraud detection	Process modelling and control
Part manufacturing	✓	✓				✓		
Process manufacturing		✓					✓	✓
Retailing			✓		✓		✓	
Finance and Insurance			✓		✓		✓	
Engineering	✓	✓		✓				✓
Production control	✓		✓					✓
Service			✓		✓			
Treasury function			✓				✓	
Sales and Marketing			✓		✓			

Multi-Layer Networks

- **Richard and Lippmann (1991) showed that multi-layer neural networks estimate the posterior probability $P(C_k/x)$ directly (i.e. the probability of the class C_k given the input vector x). This holds if:**
  - a 1-out-of-K output coding (so that $t_k = 1$ if x belongs to C_k and 0 otherwise) is used
  - the weights are chosen so as to minimize a squared-error cost function.



Multi-Layer Networks [Lippmann 1987]

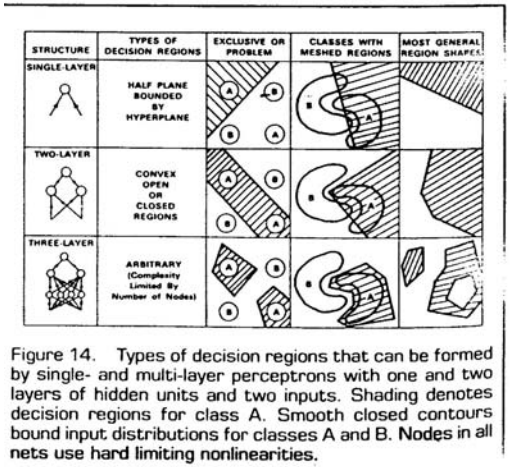


Figure 14. Types of decision regions that can be formed by single- and multi-layer perceptrons with one and two layers of hidden units and two inputs. Shading denotes decision regions for class A. Smooth closed contours bound input distributions for classes A and B. Nodes in all nets use hard limiting nonlinearities.

Some properties of MLP networks

- **If $x_k = \pm 1$ ($x_k = +1/0$) (components of the input vector) and f=sgn(x) (hard limiter) for the single output unit and f=tanh(x) (f=sigmoid(x)) for the hidden units, then only one hidden layer suffices to represent any Boolean function.**
- **To approximate a function F(x) to a given accuracy, at most two hidden layers, with arbitrary accuracy being obtainable given enough neurons per layer, are needed [Cybenko 1988].**
- **Only one hidden layer is enough to approximate any continuous function [Cybenko 1989][Hornik et al., 1989].**

How much hidden units?

- **The number of training examples (i.e. input vectors in the training set P) should be of the same order as the number of free parameters of the network. Given a I-J-K MLP network, the number of weights W is: W=(I+1)J+(J+1)K; then P=W.**
- **Baum and Haussler (1989):**

$$P=W/\varepsilon$$

where ε is an "accuracy parameter" (i.e. the fraction of patterns of the test set which are incorrectly classified). For a good generalization the accuracy level "should be" 90% corresponding to ε=0.1 then: 10W=P

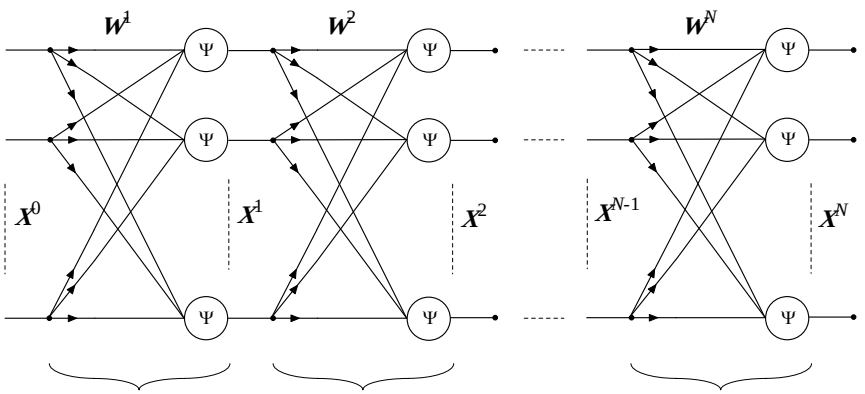
- **Widrow and Lear (1990) argued that:**

$$P >> W/K$$

In conclusion

- **A lower bound for the number of training patterns P is: W=P**
- **A realistic upper bound might be P =10W**

Standard architecture of an N-layer Multi Layer Perceptron network



The Back Propagation learning algorithm

- **feedforward computations**

$$a_j^l = \sum_{i=1}^{n^{l-1}} W_{j,i}^l X_i^{l-1}$$

$$X_j^l = \tanh\left(a_j^l\right)$$

$$l \in [1 \div N],\ j \in [1 \div n^l],\ i \in [1 \div n^{l-1}]$$

- **error index for the pth pattern**

$$\varepsilon_p(\mathbf{w}) = \frac{1}{2}\sum_{k=1}^{n^N}\left[\overline{X}_k^N - X_k^N\right]^2$$

The Back Propagation learning algorithm

- **error index**

$$\varepsilon\ (\mathbf{w}) = \sum_{p=1}^{N^p}\ \varepsilon_p(\mathbf{w}) = \frac{1}{2}\sum_{p=1}^{N^p}\ \sum_{k=1}^{n^N}\left[\overline{X}_k^N - X_k^N\right]^2$$

- **weight update rule**

$$\Delta W_{j,i}^l(t+1) = W_{j,i}^l(t+1) - W_{j,i}^l(t) = \Delta W_{j,i}^l(t) + \alpha \Delta W_{j,i}^l(t-1)$$

usually $0<\alpha<1$

The Back Propagation learning algorithm

- **batch (by-epoch)** $\Delta W_{i,j} = -\eta \dfrac{\partial \varepsilon(\mathbf{w})}{\partial w_{i,j}}$

$\Delta W_{k,\upsilon}^N = \eta \sum_{p=1}^{N^p} \delta_k^N X_\upsilon^{N-1}$ in the output layer

$\Delta W_{j,i}^l = \eta \sum_{p=1}^{N^p} \delta_j^l X_i^{l-1}$ in the hidden layers

- **on-line (by pattern)** $\Delta W_{i,j} = -\eta \dfrac{\partial \varepsilon_p(\mathbf{w})}{\partial w_{i,j}}$

a pattern is presented at the input and then all weights are updated before the next pattern is considered. If η is small enough, this clearly decreases the cost function at each step.

If the pattern are chosen in random order from the training set, it also makes the path through weight-space stochastic allowing wider exploration of the cost surface.

The Back Propagation learning algorithm

- **on-line BP**

$$\Delta W_{k,\upsilon}^{N} = -\eta \frac{\partial \varepsilon_p(\mathbf{w})}{\partial W_{k,\upsilon}^{N}} \qquad \text{in the output layer}$$

$$\Delta W_{j,i}^{l} = -\eta \frac{\partial \varepsilon_p(\mathbf{w})}{\partial W_{j,i}^{l}} \qquad \text{in the hidden layers}$$

$$k \in [1 \div n^{N}], \upsilon \in [1 \div n^{N-1}]$$

$$l \in [1 \div N-1],\ j \in [1 \div n^{l}], i \in [1 \div n^{l-1}]$$

The Back Propagation learning algorithm

- for the k^{th} neuron of the output layer the error term is:

$$\Delta W_{k,\upsilon}^{N} = \eta \delta_k^{N} \mathrm{X}_{\upsilon}^{N-1} \qquad \text{in the output layer}$$

$$\Delta W_{j,i}^{l} = \eta \delta_j^{l} \mathrm{X}_{i}^{l-1} \qquad \text{in the hidden layers}$$

$$k \in [1 \div n^{N}], \upsilon \in [1 \div n^{N-1}]$$

$$l \in [1 \div N-1],\ j \in [1 \div n^{l}], i \in [1 \div n^{l-1}]$$

- for the i^{th} neuron of the l^{th} hidden layer:

$$\delta_j^{l} = -\sum_{k=1}^{n^{l+1}} \frac{\partial \varepsilon_p(\mathbf{w})}{\partial X_k^{l+1}} \cdot \frac{\partial X_k^{l+1}}{\partial a_k^{l+1}} \cdot \frac{\partial a_k^{l+1}}{\partial X_j^{l+1}} \cdot \frac{\partial X_j^{l}}{\partial a_j^{l}} = \sum_{k=1}^{n^{l+1}} \delta_k^{l+1} W_{k,j}^{l+1} D_j^{l}$$

The Back Propagation learning algorithm

- where D_k^N and D_j^l are the derivatives of the neuron activation function with respect to a_k^N and a_j^l respectively

$$D_k^{N} = \frac{\partial X_k^{N}}{\partial a_k^{N}} = 1 - \tanh^2(a_k^{N}) = 1 - (X_k^{N})^2$$

$$D_j^{l} = \frac{\partial X_j^{l}}{\partial a_j^{l}} = 1 - \tanh^2(a_j^{l}) = 1 - (X_j^{l})^2$$

The Back Propagation learning algorithm

- **1. Initialize the synaptic weights to small random values.**
- **2. Present an input pattern (chosen randomly out of the training set) to the network and compute the neuron outputs**
- **3. Present the corresponding target output to the network and compute the error terms for all neurons**
- **4. Update the synaptic weight values**
- **5. Go back to step 2 until the error ε_p is acceptably low (i.e. the given termination condition/stopping criterion is satisfied).**

The stopping criterion

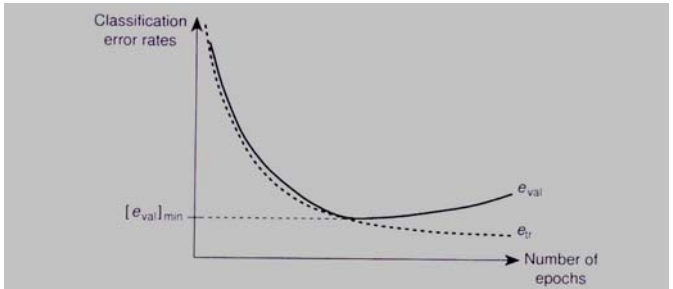


Fig. 7.5 *Plots of training and validation classification error rates during training.*

Learning rate and momentum term

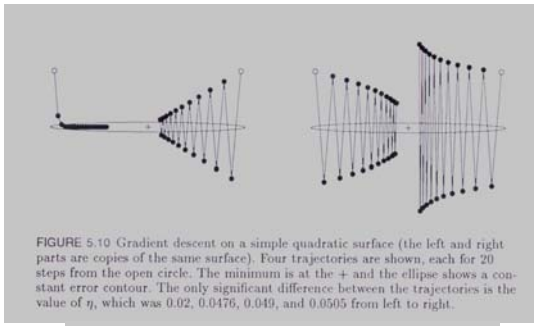


FIGURE 5.10 Gradient descent on a simple quadratic surface (the left and right parts are copies of the same surface). Four trajectories are shown, each for 20 steps from the open circle. The minimum is at the + and the ellipse shows a constant error contour. The only significant difference between the trajectories is the value of η, which was 0.02, 0.0476, 0.049, and 0.0505 from left to right.

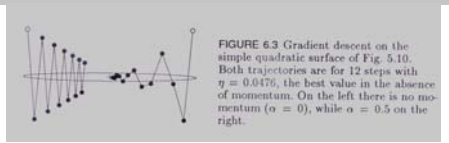


FIGURE 6.3 Gradient descent on the simple quadratic surface of Fig. 5.10. Both trajectories are for 12 steps with $\eta = 0.0476$, the best value in the absence of momentum. On the left there is no momentum ($\alpha = 0$), while $\alpha = 0.5$ on the right.

The Perturbation-based learning algorithm

Based on the approximation of the error gradient by a finite difference

- **Weight perturbation (Jabri and Flower, 1992)**

$$\frac{\partial \varepsilon_p(\mathbf{w})}{\partial w_{j,i}} \cong \frac{\varepsilon_p(w_{j,i} + p_{j,i}^{(n)}) - \varepsilon_p(w_{j,i})}{p_{j,i}^{(n)}}$$

$$\Delta w_{j,i} = -\eta \frac{{}_p\varepsilon(w_{j,i} + p_{j,i}^{(n)}) - \varepsilon_p(w_{j,i})}{p_{j,i}^{(n)}}$$

- **Stochastic error descent (Cauwenberghs, 1993)**

$\Delta \mathbf{w} = -\eta(\varepsilon_p(\mathbf{w}+\mathbf{p}^{(n)}) - \varepsilon_p(\mathbf{w}))\,\mathbf{p}^{*(n)}$

where n is the iteration index· $p^{(n)}$ is the perturbation matrix of elements $p_{j,i}$ which are spatially and temporally uncorrelated

$$E(p_{j,i}^n p_{k,l}^m) = \sigma^2 \delta_{(ji,kl)} \delta_{(n,m)}$$

The Perturbation-based learning algorithm

- **the algorithm performs gradient descent on average**
- **the error index always decreases provided that the $|p_{j,i}|^2$ is "small" and $\eta$is "small" and strictly positive**

For each pattern of the training set

{apply perturbation to all weights;

weight update;}

- **Stochastic error descent (Alspector et al., 1993, 1996)**

Every weight perturbation $p_{j,i}^{(n)}$ is equal in value and random only in sign:

$$p_{j,i}^{(n)} = pert_{j,i}^{(n)} step$$

$pert_{j,i}^{(n)}$ can assume the values +1 or –1 with equal probability.

$$\Delta w_{j,i} = -\eta \frac{\varepsilon_p(w_{j,i} + p_{j,i}^{(n)}) - \varepsilon_p(w_{j,i})}{p_{j,i}^{(n)}} = -\eta \frac{\Delta \varepsilon_p}{step} pert_{j,i}^{(n)} = -\eta \Big/ step \, \Delta \varepsilon_p \cdot pert_{j,i}^{(n)}$$

•

The Perturbation-based learning algorithm

$$\Delta w_{j,i} = -\eta' \cdot \Delta\varepsilon_p \cdot pert_{j,i}^{(n)} \qquad \eta' = \eta / step$$

To compute the synapse's weight $w_{j,i}$, we only need to compute $\Delta\varepsilon_p$ and to known $pert_{j,i}^{(n)}$.

for(each epoch)
{set each $pert_{j,i}^{(0)}$ at a random value;
for(each pattern of the training set)
{Choose a pattern in random way and put it in input to the network;
Feed-Forward phase;
Compute $\varepsilon_p(w_{j,i})$;
Weight Perturbation;
Feed-Forward phase;
Compute $\varepsilon_p(w_{j,i}+step \cdot pert_{j,i}^{(n)})$;
Compute $\Delta w_{j,i} = -\eta' \cdot \Delta\varepsilon_p \cdot pert_{j,i}^{(n)}$;
WeightUpdate;
}
}

The Perturbation-based learning algorithm

- **Fan-in perturbation (Flower and Jabri, 1993)**

For each pattern of the training set
{random selection of a neuron;
apply perturbation to all weights of the synapses that are connected in input of neuron j;
weight update;}

$\Delta w = -\eta(\varepsilon_p(w+\rho_j^{(n)}) - \varepsilon_p(w))\,\rho_j^{-1(n)}$

where $\rho_j^{(n)}$ is the perturbation matrix where only the elements $p_{j,i}$ corresponding to the connections in input to neuron j are non null and $\rho_j^{-1(n)}$ is the inverse matrix of elements $1/p_{j,i}$ (inverse of perturbation).

- **Fan-out perturbation (Flower and Jabri, 1996)**

For each pattern of training set
{random selection of a neuron;
apply perturbation to all weights of the synapses connected at the output of neuron j;
weight update;}

The Perturbation-based learning algorithm

$\Delta w = -\eta(\varepsilon_p(w+\xi_j^{(n)}) - \varepsilon_p(w))\,\xi_j^{-1(n)}$

where $\xi_j^{(n)}$ is the perturbation matrix where only the elements $p_{j,i}$ corresponding to the connections leaving the neuron j are non null and $\xi_j^{-1(n)}$ is the inverse matrix of elements $1/p_{j,i}$ (inverse of perturbation).

- **Fan-in-out perturbation (Flower and Jabri, 1996)**

For each pattern
{random selection of a neuron;
apply perturbation to all weights of the synapses feeding into and leaving the neuron j;
weight update;}

$\Delta w = -\eta(\varepsilon_p(w+\psi_j^{(n)}) - \varepsilon_p(w))\,\psi_j^{-1(n)}$ where $\psi_j^{(n)}$ is the perturbation matrix where only the elements $p_{j,i}$ corresponding to the connections that leave and feed the neuron j are non null and $\psi_j^{-1(n)}$ is the inverse matrix of elements $1/p_{j,i}$ (inverse of perturbation).

Adaptive parameters

Vogl's acceleration technique [Vogl 1988]

$$\Delta\eta = \begin{cases} +a \text{ if } & \Delta\varepsilon < 0 \\ -b\eta \text{ if } & \Delta\varepsilon > 0 \\ 0 \text{ otherwise} & \end{cases}$$

The learning rate cannot be considered unique for all the synapses of the network, but each synapse has its own learning rate, i.e. $\eta_{j,i}^{l}$

$$\Delta W_{j,i}^{l} = \eta_{j,i}^{l}\delta_j^{l} X_i^{l-1}$$
$$l \in [1 \div N],\ j \in [1 \div n^{l}],\ i \in [1 \div n^{l-1}]$$

Improvements in the learning convergence speed

The weight update rule becomes:

$$\Delta W^{l}_{j,i}(t+1) = -\eta_{j,i}\frac{\partial \varepsilon(\mathbf{w(t)})}{\partial W^{l}_{j,i}}$$

The local learning rate value can be locally managed according to the sign of the component of the gradient of the error function ($\frac{\partial \varepsilon_p(\mathbf{w})}{\partial W^{l}_{j,i}}$):

- if the sign is equal during two consecutive iterations, the corresponding learning rate is increased since a (local) minimum lies in such direction;
- if the sign changes during two consecutive iterations, the corresponding learning rate is decreased, since probably a (local) minimum is being skipped over.

Improvements in the learning convergence speed

$$if\ S^{l}_{j,i}(t) = S^{l}_{j,i}(t-1)$$

$$\eta^{l}_{j,i}(t+1) = \eta^{l}_{j,i}(t)\cdot\left[\frac{\eta^{\max}}{\eta^{l}_{j,i}(t)}\right]^{\gamma}$$

$$else$$

$$\eta^{l}_{j,i}(t+1) = \eta^{l}_{j,i}(t)\cdot\left[\frac{\eta^{\min}}{\eta^{l}_{j,i}(t)}\right]^{\gamma}$$