Matrix Techniques in Artificial Neural Networks

A thesis
submitted in partial fulfilment
of the requirements for the Degree
of
Master of Science in Mathematics
in the
University of Canterbury

by

Ryurick M. Hristev

University of Canterbury
2000

Matrix Techniques in ANN — *R. M. Hristev* — Edition 1.26a (15 June 2001)

This thesis was submited in fulfilment of the requirements for the Degree of Master of Science in Mathematics in the University of Canterbury in 2000.

The book version is called "Matrix ANN" and is released under the GNU GPL licence. For the latest version of this book please contact me at `R.Hristev@phys.canterbury.ac.nz`

Preface

About Artificial Neural Networks (ANN)

In recent years artificial neural networks (ANN) have emerged as a mature and viable framework with many applications in various areas. ANN are mostly applicable wherever some hard to define (exactly) patterns have to be dealt with. "Patterns" are taken here in the broadest sense, applications and models have been developed from speech recognition to (stock)market time series prediction with almost anything in between and new ones appear at a very fast pace. ANN

The ANN Technology

Artificial Neural Networks (ANN) offers improved performance in areas ([MHP90] pp. 1–9) which include:

- Pattern recognition (spatial, temporal and whenever patterns may be measured and thus quantified);
- Associative search (e.g. in database mining);
- Adaptive control;
- Scheduling and routing;
- Man-machine interfaces.

this representing also the main areas of application.

The advantages of ANN technology include:

- Adaptive learning (learning by example);
- Self-organization (ANN create their own internal representation of data);
- Fault-tolerance (data is distributed across several neuronal connections, often over the whole network);
- Modularity (an ANN may be built to perform a single task and then inserted into a larger system).

The ANN field of research contains several almost independent trees:

- The "perceptron/feedforward" tree: started with the perceptron developed by Rosenblatt (1958) continued with the development of adaline by Widrow and Hoff (1960) and the development of backpropagagation algorithm by Werbos (1974). Feedforward networks with backpropagation and related training algorithms are the most widely used in practice (and arguably the best understood and with the largest research base);

- The associative memory tree: best illustrated by the SOM networks first developed by Kohonen (1972). These networks may be used for unsupervised clustering and associative search;
- The adaptive resonance theory: developed mainly to better understand some features of natural brains (e.g. the ability to resume learning at any time).

Of course there are many other types of connectionist models and neuronal learning and new ones do appear at a rate which makes at least very difficult to follow all the new developments.

Remarks:

➥ ANN technology has also been combined with other (more or less) related methods like learning vector quantization, fuzzy logic and "classical" statistics.

Some Real–Life Applications

ANN have been implemented in many areas with various degree of success. A few examples of successful and well documented applications are:

- *WEBSOM*[1] — data mining / search engine application — "WEBSOM is a method for organizing miscellaneous text documents onto meaningful maps for exploration and search. WEBSOM is based on the SOM (Self-Organizing Map) algorithm that automatically organizes the documents onto a two-dimensional grid so that related documents appear close to each other."
- *Particle detection*[2] — pattern recognition — "The CDF experiment at the Tevatron proton-antiproton collider at Fermilab in the USA has had several calorimeter neural network triggers running for several of years now. . . . The H1 experiment at the HERA ep Collider at the DESY Laboratory in Hamburg, Ge., has developed a 1st level trigger that runs in 50ns in addition to the 2nd level trigger discussed below. It is based on the Neuroclassifier chip that was specifically designed for this experiment. This chip has 70 analog inputs, 6 hidden neurons with 5-bit digitally loaded weights and sigmoidal transfer functions, and 1 summed output (transfer function is off-chip to give the option of combining chips to increase the number of hidden units). . . . The H1 experiment at the HERA ep Collider at the DESY Laboratory in Hamburg, Ge., has implemented a level 2 trigger based on hardware neural networks."
- *Sharemarket prediction*[3] — finance application — "Today neural networks are the most common system for the prediction of share prices excluding traditional techniques."
- Oil prospecting[4] — geology application — "Shell Research trained neural networks to classify rock types as function of the wireline data. For different geological environments different neural networks were trained. The desired outputs were provided by expert geologists. The degree of consensus between the neural network and its trainer

[1] `http://websom.hut.fi/websom/`
[2] `http://www1.cern.ch/NeuralNets/nnwInHepExpt.html`
[3] `http://www.afin.freeserve.co.uk/`
[4] `http://www.mbfys.kun.nl/snn/siena/cases/shell.html`

were roughly equivalent to the degree of consensus among different geologists. The neural network has been incorporated in Shell's geological computing environment, in which the formation analyst can train and apply the neural network via a user-friendly graphical interface."

- El Nino prediction[5] — weather prediction — "The neural network used here is a simple feedforward neural network, consisting of an input layer for the 12 inputs, a hidden layer of 7 nonlinear units, and an output layer of 1 linear unit. An ensemble of 20 neural networks were trained, each with a different weight initialization. The final model output was the average of the outputs from the 20 members of the ensemble."

Future Trends

Many ANN require the ad-hoc setup of a small number of parameters by a "supervisor" (which may be a human) which is error prone and time consuming. Learning algorithms which do not require this (e.g. conjugate gradients) are superior and methods to avoid wherever possible the "hand" selection of these constants are highly desirable.

While the current practical applications of feedforward ANN do not use more than one or two hidden layers in our opinion the optimal number of layers is problem dependent and may be far greater than two (research indicates that the human brain may have 8 – 10 "layers").

A basis for the successful application of ANN technology is a good understanding of the methods and limitations. Generally the model built shall attempt (if and wherever possible) not only to *learn* it (considered successful when it may perform an good emulation/prediction of the phenomenon studied) but also to *understand* it, i.e. to be able to extract *rules* (an objective far more difficult to achieve). In either case a good integration with other techniques (e.g. through preprocessing) is crucial to achieve good performance.

About This Thesis

To be able to (correctly) apply the ANN technology it is not enough just to throw some data at it randomly and wait to see what happens next. At least some understanding of the underlying mechanism is required in order to make efficient use of it.

This thesis makes an attempt to cover some of the basic ANN development from the matrix methods perspective. ANN may be subject to aggressive optimizations using both linear algebra methods as well as programming techniques. What's more, these techniques may be applied across several unrelated ANN architectures, i.e. once the basic required matrix operations have been implemented they may be reused on several ANN algorithms. Many of the matrix techniques are novel in the ANN setting in the sense that they have not been seen in the literature to date.

The thesis is organized as follows:

- *The first part* covers some of the most widely used ANN architectures. New ones or variants appear at a fast rate so it is not possible to cover them all, but these are among the few ones with wide applications. Some theoretical results are left for the

[5] http://www.ocgy.ubc.ca/projects/clim.pred/neural/NINO34.html

second part where a more developed mathematical apparatus is introduced. Basically this part answers the question "How it works?" rather than "Why ?".

- *The second part* takes a deeper look at the fundamentals as well as establishing the most important theoretical results. It also describes some algorithmic optimizations/variants for ANN simulators which require a more advanced mathematical apparatus. This part attempts to answer to the questions "Why it works ?" and "How do I interpret the ANN outputs ?".

A section (chapter, sub–section, etc.) may have a corresponding special footnote carrying some bibliographic information relevant to that section. These footnotes are marked with the section number (e.g. 2.3.1 for sub-section numbered 2.3.1) or with a number and an ".*" for chapters (e.g. 3.* for the third chapter).

The programs used in this thesis were developed mostly under *Scilab*, available under a very generous licence (basically: free and with source code included) from: "`http://www-rocq.inria.fr/scilab/`". Scilab is very similar to Octave and Matlab. *Octave* is also released under the GNU licence, so it's free. Some Scilab ANN implementations and programs may be found in the source tree of this thesis under the directory "`Matrix_ANN_1/Scilab/ANN_Toolbox-0.22/`" (actual version numbers may differ). Read the "`README.install`" file for installation instructions.

The next section describes the notational system used in this thesis.

Mathematical Notations and Conventions

The following notational system will be used throughout the whole thesis. There will be two kind of notations: some which will be described here and usually will *not* be explained in the text again; the other ones will be local (to the chapter, section, etc.) and will appear also in the marginal notes *in the place where they are defined/used first*, marked with the symbol ❖, like the one appearing here.

❖ marginal note

So, when you encounter a symbol and you don't know what it is: first look in this section, if is not here follow the marginal notes *upstream* from the point where you encountered it until you find its definition (you should not go beyond the current chapter).

Proofs are typeset in a smaller (8 pt.) font size and refer to previous formulas when not explicitly specified. The reader may skip them, however following them will enhance the understanding of the topics discussed.

* * *

ANN involves heavy manipulations of vectors and matrices[6]. Some notations have been unified with other books in a effort to make reading easier, however many others had to be introduced to cope with the requirements of this thesis.

A vector will be also represented by a column matrix:

$$\mathbf{x} = \begin{pmatrix} x_1 \\ \vdots \\ x_N \end{pmatrix}$$

and in text will be often represented by its transpose $\mathbf{x}^{\mathrm{T}} = \begin{pmatrix} x_1 & \cdots & x_N \end{pmatrix}$ (for aesthetic reasons and readability). Also it will be represented by lowercase **bold** letters.

[6] For an introduction to matrix theory see [Ort87], for matrix algorithms see [GL96].

The Euclidean scalar product between two vectors may be represented by a product between the corresponding matrices:

$$\mathbf{x} \cdot \mathbf{y} := \sum_i x_i y_i =: \mathbf{x}^{\mathrm{T}} \mathbf{y}$$

Note that $\mathbf{x}^{\mathrm{T}}\mathbf{y}$ is a scalar and is also known as the *inner–product* while $\mathbf{x}\mathbf{y}^{\mathrm{T}}$ is a matrix and is known as the *outer–product* or *correlation matrix* (between $\mathbf{x}$ and $\mathbf{y}$).

inner–product
outer–product
correlation matrix

The other matrices will be represented by uppercase letters. The inverse of a square matrix will be denoted by $(\cdot)^{-1}$. The elements of matrices will be represented by the same letter but lowercase or by a notation of type $(A)_{ij}$ for element (i, j) of matrix A if necessary. Also the matrix of some elements a_{ij} may be noted as $\{a_{ij}\}$ (first index is the row, second represents column).

❖ $(\cdot)^{-1}$, $(\cdot)_{[\cdot]}$, $\{\cdot\}$

Scalars are represented by lowercase letters.

There is an important distinction between scalar and vectorial functions. When a scalar function is applied to a vector or matrix it means in fact that it will be applied to each element in turn and the result will be a vector, of the *same dimension* as its argument (the function is applied componentwise):

$$f(\mathbf{x}^{\mathrm{T}}) = \begin{pmatrix} f(x_1) & \cdots & f(x_N) \end{pmatrix} \quad , \quad f : \mathbb{R} \to \mathbb{R}$$

the application to a vector being just a convenient notation, while:

$$\mathbf{g}^{\mathrm{T}}(\mathbf{x}) = \begin{pmatrix} g_1(\mathbf{x}) & \cdots & g_K(\mathbf{x}) \end{pmatrix} \quad , \quad \mathbf{g} : \mathbb{R}^N \to \mathbb{R}^K$$

is a vectorial function and generally $K \neq N$. The result is a vector of different dimension (in general). Note that **bold** letters are used for vectorial functions. In particular matrix exponentiation is a *defined mathematical operation* for a square matrix A:

$$\mathbf{exp}(A) \equiv \mathbf{e}^A \equiv \sum_{i=1}^{\infty} \frac{A^i}{i!}$$

while $\exp(A) \equiv e^A$ is just the matrix whose i, j element is $\exp(a_{ij})$.

One operator which will be used is the ":" ("scissors" operator). The $A_{(i,:)}$ notation will represent row i of matrix A, while $A_{(:,j)}$ will stand for column j of the same matrix. $A_{(i:j,k:l)}$ stands for the submatrix of A, made from rows i to j and columns k to l, taken from A.

❖ :

Another operation used will be the *Hadamard product*, i.e. *the element-wise product between matrices (or vectors)*, denoted by $\odot$. The terms and result have to be of the same shape (number of rows and columns) and the elements of the result are the product of the *corresponding* elements of the input matrices:

Hadamard
❖ $\odot$

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{K1} & \cdots & a_{KN} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & \cdots & b_{1N} \\ \vdots & \ddots & \vdots \\ b_{K1} & \cdots & b_{KN} \end{pmatrix} \quad \Rightarrow \quad A \odot B := \begin{pmatrix} a_{11}b_{11} & \cdots & a_{1N}b_{1N} \\ \vdots & \ddots & \vdots \\ a_{K1}b_{K1} & \cdots & a_{KN}b_{KN} \end{pmatrix}$$

Note that the Hadamard product does not have the same priority as standard matrix product, e.g. $\mathbf{x}^{\mathrm{T}}\mathbf{y} \odot \mathbf{z}$ should be read as $\mathbf{x}^{\mathrm{T}} \cdot (\mathbf{y} \odot \mathbf{z})$ representing a scalar (and *not* $(\mathbf{x}^{\mathrm{T}} \cdot \mathbf{y}) \odot \mathbf{z}$ which does not make sense). Parentheses will be used wherever necessary to avoid confusion.

The *Kronecker product* of two matrices, denoted by $\otimes$, is defined as:

Kronecker
❖ $\otimes$

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1N} \\ \vdots & \ddots & \vdots \\ a_{K1} & \cdots & a_{KN} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & \cdots & b_{1M} \\ \vdots & \ddots & \vdots \\ b_{H1} & \cdots & b_{HM} \end{pmatrix} \quad \Rightarrow$$

$$A \otimes B := \begin{pmatrix} a_{11}B & \cdots & a_{1N}B \\ \vdots & \ddots & \vdots \\ a_{K1}B & \cdots & a_{KN}B \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} & \cdots & a_{1N}b_{1M} \\ \vdots & \ddots & \vdots \\ a_{K1}b_{H1} & \cdots & a_{KN}b_{HM} \end{pmatrix}$$

i.e. $A \otimes B$ will have dimensions $KH \times NM$.

None of the operators ":", "$\odot$" or "$\otimes$" are new but they have been defined here as they may be unfamiliar to some readers.

❖ $\circ$

Finally the collapsible tensorial multiplication $\circ$: for two tensors A and B the elements of $A \circ B$ are obtained by summation over all mutual indices, this results in a lower dimensional tensor, e.g. assume $A = \{a^i_k\}$, $A \otimes A^{\mathrm{T}} = B = \{b^{i\ell}_{kj}\}$, let $C = \{c^j_\ell\}$ then $D = B \circ C = \{d^i_k\}$ where $d^i_k = b^{i\ell}_{kj} c^j_\ell$ (summation over ℓ and j) and D has the same dimensions as A (matrix), $C^{\mathrm{T}} \circ B \circ C$ will be the scalar $c^k_i b^{i\ell}_{kj} c^j_\ell$ (summation over all indexes); summation indices will be specified wherever necessary to avoid confusion. *This operator is not used in the first part of this thesis.*

ANN	acronym for Artificial Neural Network(s).
$\odot$	Hadamard, element-wise, product of matrices (see above for the definition).
$\otimes$	Kronecker matrix product (see above for the definition).
$(\cdot)^{\odot n}$	a convenient notation for $\underbrace{(\cdot) \odot \cdots \odot (\cdot)}_{n} \equiv (\cdot)^{\odot n}$
:	matrix "scissors" operator (see above for the definition). This operator *takes precedence* over others, $W^{\mathrm{T}}_{(:,i)} = (W_{(:,i)})^{\mathrm{T}}$.
$(\cdot)^{\mathrm{T}}$	transpose of matrix $(\cdot)$.
$(\cdot)^{\mathrm{C}}$	complement of vector $(\cdot)$; it involves swapping $0 \leftrightarrow 1$ for binary vectors and $-1 \leftrightarrow +1$ for bipolar vectors; it also represents complementary sub-sets.
$\lvert \cdot \rvert$, $\mathrm{abs}(\cdot)$	determinant if applied to a square matrix, otherwise is the scalar's modulus; abs is the *element-wise* modulus of a matrix or vector.
$\lVert \cdot \rVert$	norm of a vector; the Euclidean norm $\lVert \mathbf{x} \rVert = \sqrt{\mathbf{x}^{\mathrm{T}}\mathbf{x}}$, unless otherwise specified.
$\langle \cdot \rangle$	mean value of a variable.
$[\cdot]$	used to specify a matrix made out of some submatrices; non–specified submatrices/elements are assumed to be zero.
$\mathcal{E}\{f \vert g\}$	expectation of event f (mean value), given event g. If f and g are functions then $\mathcal{E}\{f \vert g\}$ is a functional.
$\mathcal{V}\{f\}$	variance of f. If f is a function then $\mathcal{V}\{f\}$ is a functional.
$\mathrm{sign}(x)$	the sign function, defined as $\mathrm{sign}(x) = \begin{cases} 1 & \text{if } x>0 \\ 0 & \text{if } x=0 \\ -1 & \text{if } x<0 \end{cases}$. When applied to a matrix, it is an element–wise operation.

$\operatorname{Vec} A$, $\operatorname{Vec}_N^{-1} \mathbf{x}$	$\operatorname{Vec} A$ converts a matrix to a vector by stacking columns at the bottom of previous ones; $\operatorname{Vec} A^{\mathrm{T}}$ will transform matrix A to a vector using lexicographic convention on A's indices; for example if $A = \left(\begin{smallmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{smallmatrix}\right)$ then $(\operatorname{Vec} A^{\mathrm{T}})^{\mathrm{T}} = \begin{pmatrix} a_{11} & a_{12} & a_{21} & a_{22} \end{pmatrix}$. $\operatorname{Vec}_N^{-1} \mathbf{x}$ is the inverse operation applied to a vector, N represents the number of rows in the newly created matrix (the number of components in $\mathbf{x}$ has to be a multiple of N).
$\operatorname{Diag}(\mathbf{x})$	is the diagonal matrix whose (i,i) entry is x_i, off–diagonal elements are zero $\big((\operatorname{Diag}(\mathbf{x}))_{ij} = \delta_{ij} x_i\big)$.
$\operatorname{sum}_{(R,C)}(\cdot)$	sum of elements of a vector or matrix; $\operatorname{sum}(\mathbf{x}) := \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}$; $\operatorname{sum}_R(A) := A\widehat{\mathbf{1}}$ (row–wise sum); $\operatorname{sum}_C(A) := \widehat{\mathbf{1}}^{\mathrm{T}} A$ (column–wise sum); $\operatorname{sum}(A) := \widehat{\mathbf{1}}^{\mathrm{T}} A \, \widehat{\mathbf{1}}$.
$\widetilde{1}_{NK}$	a *matrix*, having *all* elements equal to 1; if N, K are not specified the dimensions will be *always* such that the mathematical operations in which it's involved are correct.
$\widehat{\mathbf{1}}_N$	a (column) *vector*, having *all* elements equal to 1; if N is not specified the dimension will be *always* such that the mathematical operations in which it's involved are correct.
$\widetilde{0}_{NK}$	a *matrix* having, *all* elements equal to 0; if N, K are not specified the dimensions will be *always* such that the mathematical operations in which it's involved are correct.
$\widehat{\mathbf{0}}_{(\cdot)}$	a (column) *vector*, having *all* elements equal to 0; if $(\cdot)$ is not specified its dimensions will be *always* such that the mathematical operations in which it's involved are correct.
$\mathbf{e}_{Nk}$	the one–of–k encoding vector, it has all elements 0 except one 1 in position k; if given, N represents the vector size; $\{\mathbf{e}_k\}$ system is orthonormal ($\mathbf{e}_k^{\mathrm{T}} \mathbf{e}_\ell = \delta_{k\ell}$).
$E_{NKk\ell}$, $E^*_{NKk\ell}$	a matrix with all elements 0 except one 1 in position (k,ℓ), i.e. $E_{k\ell} = \mathbf{e}_k \mathbf{e}_\ell^{\mathrm{T}}$; if given, NK represents the matrix size; $E^*_{(\cdot)k\ell}$ has 1 in both (k,ℓ) and (ℓ,k) positions.
I_N	the $N \times N$ identity matrix; if N is not specified the dimension will be *always* such that the mathematical operations in which it's involved are correct.
x_i	component i of input vector.
$\mathbf{x}$	the input vector: $\mathbf{x}^{\mathrm{T}} = \begin{pmatrix} x_1 & \cdots & x_N \end{pmatrix}$
y_k	component k of network output.
$\mathbf{y}$	the network output vector: $\mathbf{y}^{\mathrm{T}} = \begin{pmatrix} y_1 & \cdots & y_K \end{pmatrix}$
z_h	output of a hidden neuron h.
$\mathbf{z}$	the output vector of a hidden layer: $\mathbf{z}^{\mathrm{T}} = \begin{pmatrix} z_1 & \cdots & z_H \end{pmatrix}$
t_k	component k of target pattern.
$\mathbf{t}$	the target vector (desired output) corresponding to input $\mathbf{x}$.

$w_w, w_{ki}, \mathbf{w}_h$	w_w some weight; w_{ki} the weight associated with connection *to* neuron k, *from* neuron i; $\mathbf{w}_h$ the weight vector associated with input to neuron h.
$W, \mathbf{w}, W_{(\ell)}$	the weight matrix, e.g. $W = \begin{pmatrix} w_{11} & \cdots & w_{1N} \\ \vdots & \ddots & \vdots \\ w_{K1} & \cdots & w_{KN} \end{pmatrix}$, note that all weights associated with a particular neuron k ($k \in [1, K]$) are on the same row. Occasionally the weight set has to be treated as a *vector*, in this case it will be written as $\mathbf{w} = \operatorname{Vec} W^{\mathrm{T}}$; $W_{(\ell)}$ represents the weight matrix associated with layer ℓ (on multi–layer ANN). Also $W_{(t)}$ or $W_{(s)}$ will represent the weight matrix at step/time t or s. In these situations $\{w_w\}$ also gets a supplementary index.
a_h	total input to neuron h, the weighted sum of its inputs, i.e. $a_h = W_{(h,:)}\mathbf{z}$, for a neuron receiving input $\mathbf{z}$.
$\mathbf{a}$	the vector containing total inputs a_h for all neurons in a same layer, usually $\mathbf{a} = W\mathbf{z}_{\text{prev.}}$, where $\mathbf{z}_{\text{prev.}}$ is the output of the previous layer.
f	neuronal activation function; the neuron output is $f(a_h)$ and the output of the current layer is $\mathbf{z} = f(\mathbf{a})$.
f'	the derivative of activation function f.
E, E_p	the error function; E_p is the contribution to the error from training pattern p.
$\mathcal{C}_k$	class k.
$P(\cdot)$, $p(\cdot)$	probability and probability density.
$p(\text{event}_1 \vert \text{event}_2)$	probability density of event 1 given event 2 (conditional probability).
$\nabla_{(\cdot)}$	the gradient operator; if the subscript is specified then it's with respect to $(\cdot)$, otherwise it's with respect to weights W so will have same dimensions as W; $\nabla_{\mathbf{w}}$ is a vector.
δ_{ij}	the Kronecker symbol: $\delta_{ij} = \begin{cases} 1 & \text{if } i=j \\ 0 & \text{otherwise} \end{cases}$.
$\delta_D(x)$	the Dirac delta distribution.
λ_i	an eigenvalue of some matrix.
$\mathbf{u}_i$	an eigenvector of some matrix.

Note also that wherever possible the following symbols will be allocated:

i, j	indices for input vector components.
$k, \ell, (\ell)$	k, ℓ as indexes for output vector components, (ℓ) as layer number for multi–layer ANN;
h, g	indices for hidden neurons;
m, n	indices for models (mixtures, committee of networks, etc.);
t, s	indices for discrete time steps;

p, q	indices for training patterns;
w, v	index of $\mathbf{W}$, when the set of weights is viewed as a vector;
N	size of input vector;
K	size of output vector;
H	size of hidden vector, number of hidden neurons;
P	number of training patterns;
N_W	total number of weights;
M	number of models (in mixtures, committee of networks, etc.).

Finally, there are a few specially defined matrix operations:

$\overset{\mathrm{R}}{\oplus}$	$a \overset{\mathrm{R}}{\oplus} \mathbf{x} := \widehat{\mathbf{1}}a + \mathbf{x}$; $\mathbf{x}^{\mathrm{T}} \overset{\mathrm{R}}{\oplus} A := \widehat{\mathbf{1}}\mathbf{x}^{\mathrm{T}} + A$
$\overset{\mathrm{R}}{\ominus}$	$a \overset{\mathrm{R}}{\ominus} \mathbf{x} := \widehat{\mathbf{1}}a - \mathbf{x}$; $\mathbf{x}^{\mathrm{T}} \overset{\mathrm{R}}{\ominus} A := \widehat{\mathbf{1}}\mathbf{x}^{\mathrm{T}} - A$
$\overset{\mathrm{C}}{\oplus}$	$a \overset{\mathrm{C}}{\oplus} \mathbf{x}^{\mathrm{T}} := a\widehat{\mathbf{1}}^{T} + \mathbf{x}^{\mathrm{T}}$; $\mathbf{x} \overset{\mathrm{C}}{\oplus} A := \mathbf{x}\widehat{\mathbf{1}}^{\mathrm{T}} + A$
$\overset{\mathrm{C}}{\ominus}$	$a \overset{\mathrm{C}}{\ominus} \mathbf{x}^{\mathrm{T}} := a\widehat{\mathbf{1}}^{T} - \mathbf{x}^{\mathrm{T}}$; $\mathbf{x} \overset{\mathrm{C}}{\ominus} A := \mathbf{x}\widehat{\mathbf{1}}^{\mathrm{T}} - A$
$\overset{\mathrm{I}}{\oplus}$	$a \overset{\mathrm{I}}{\oplus} A := aI + A$ (A square matrix)
$\overset{\mathrm{I}}{\ominus}$	$a \overset{\mathrm{I}}{\ominus} A := aI - A$ (A square matrix)
$\overset{\mathrm{R}}{\odot}$	$\mathbf{x}^{\mathrm{T}} \overset{\mathrm{R}}{\odot} A := \widehat{\mathbf{1}}\mathbf{x}^{\mathrm{T}} \odot A$
$\overset{\mathrm{C}}{\odot}$	$\mathbf{x} \overset{\mathrm{C}}{\odot} A := \mathbf{x}\widehat{\mathbf{1}}^{\mathrm{T}} \odot A$
$\oslash$	Hadamard "division" — the inverse operation of $\odot$; e.g. if $A \odot B = C$ then $B = C \oslash A$, $(a_{ij} \neq 0)$.
$\mathcal{H}$	the matrix "if" (meta)operator; $\mathcal{H}\{A, B, \widehat{\alpha}, \widehat{\beta}, \widehat{\gamma}\}$ returns a matrix B' whose elements are: $b'_{ij} = \begin{cases} \widehat{\alpha}(b_{ij}) & \text{if } a_{ij}>0 \\ \widehat{\beta}(b_{ij}) & \text{if } a_{ij}=0 \\ \widehat{\gamma}(b_{ij}) & \text{if } a_{ij}<0 \end{cases}$. $\widehat{\alpha}$, $\widehat{\beta}$ and $\widehat{\gamma}$ are operators, e.g. "$= 1$" makes $b'_{ij} = 1$, "$=$" leaves b_{ij} unchanged.

Note that $\odot$, $\otimes$, $\overset{\mathrm{R}}{\odot}$, $\overset{\mathrm{C}}{\odot}$ and $\oslash$ have undefined priority with respect to the usual matrix multiplication and between one each other.

Acknowledgements

I would like to express my gratitude to Assoc. Prof. Rick Beatson (Mathematics and Statistics Department, University of Canterbury) for his invaluable advice while revising this manuscript.

Ryurick M. Hristev

Contents

ANN Architectures

CHAPTER 1

Basic Neuronal Dynamics

1.1 Simple Neurons and Networks

First attempts at building artificial neural networks (ANN) were motivated by the desire to create models of natural brains. Much later it was discovered that ANN are a very general statistical[1] framework for modelling posterior probabilities given a learning set of samples (the training data).

The basic building block of an ANN is the neuron. A neuron is a processing unit which has some (usually more than one) inputs and only one output. See Figure 1.1 on the following page. First each input x_i is weighted by a factor w_i and the whole sum of inputs is calculated $\sum\limits_i w_i x_i = \mathbf{w}^{\mathrm{T}}\mathbf{x} = a$. Then an activation function f is applied to the result a. The neuronal output is taken to be $f(a)$.

neuron

activation function

More general designs are possible, e.g. higher order ANNs where the total input to a neuron contains also higher order combinations of inputs, e.g. for 2-nd order ANN, to each neuron h corresponds a square symmetric matrix of weights W_h and its total input is of the form $\sum\limits_{ij} w_{ij} x_i x_j = \mathbf{x}^{\mathrm{T}} W \mathbf{x}$. However these higher order designs are seldom used in practice as they involve huge computational efforts without clear-cut benefits (except for some very special pre-processing techniques).

Generally the ANN are built by putting the neurons in layers and connecting the outputs of neurons from one layer to the inputs of the neurons in the next layer. See Figure 1.2 on the next page. The type of network depicted there is also named *feedforward* (a feedforward

feedforward ANN

[1.*] For more information see also [BB95]. This reference discusses some detailed theoretical models for true neurons.

[1] In the second part of this book it is explained in greater detail how (usually) the ANN output have a direct statistical significance.

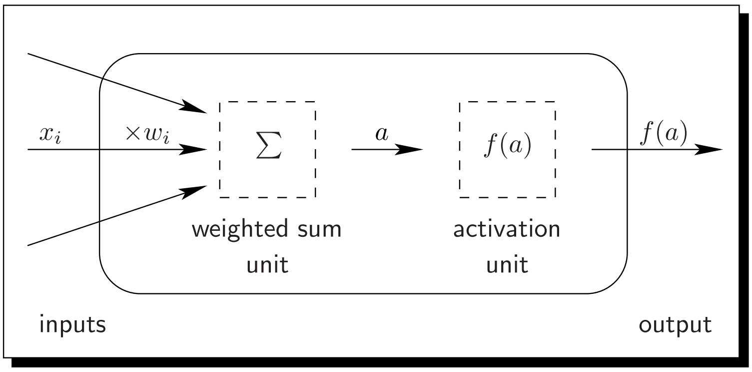

Figure 1.1: *The neuron.*

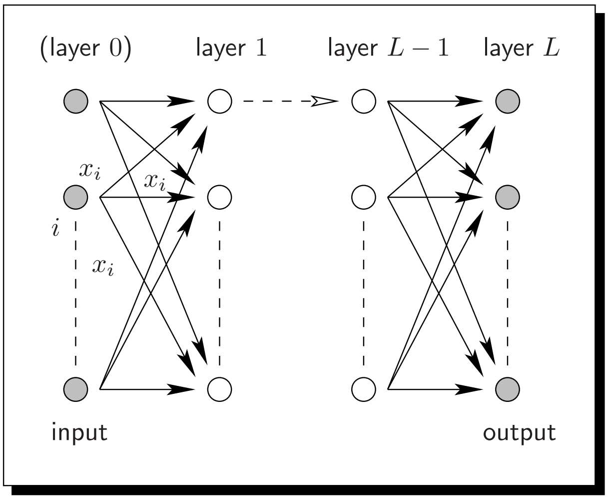

Figure 1.2: *The general layout of a feedforward ANN. Layer 0 distributes the input to the input layer 1. The output of the network is (generally) the output of the output layer L (last layer).*

network does not have feedback, i.e. no "loops"). Note that there is no processing on the layer 0, its role is just to distribute the inputs to the next layer (data processing really starts with layer 1), for this reason its representation will be omitted most of the time.

Variations are possible: the output of one neuron may go to the input of any neuron, including itself. If the outputs of a neuron from one layer go to the inputs of neurons from previous layers then the network is called *recurrent*, this providing feedback; lateral feedback is performed when the output of one neuron goes to the other neurons on the same layer or when neurons are indirectly influenced (indirect feedback) by their neighbors[2] (ANN topology wise).

recurrent ANN

The tunable parameters of an ANN are the weights W. They are found by different mathematical procedures[3] by using a given set of data, i.e. the training set. The procedure of

[2]This is used into the SOM/Kohonen architecture.
[3]Most usual is the gradient-descent method and derivatives.

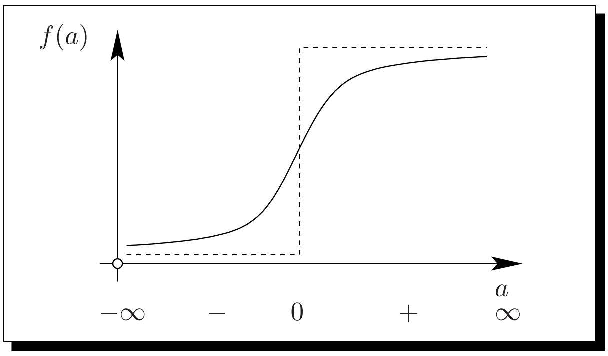

Figure 1.3: *Signal $f(a)$ as a bounded monotone-nondecreasing function of activation a. The dashed curve defines a threshold signal function.*

finding the weights is called *learning* or *training*. The data set is called the *learning* or *training set* and contains pairs of input vectors associated with the *desired* output vectors (targets): $\{\mathbf{x}_p, \mathbf{t}_p\}_{p \in [1,P]}$. Some ANN architectures do not need a learning set in order to set their weights, in this case the learning is said to be *unsupervised* (otherwise the learning is said to be *supervised*).

learning

Remarks:

- Usually the inputs are distributed to all neurons of the first layer, this one being called the *input layer*: layer 1 in Figure 1.2 on the facing page. Some networks may use an additional layer (layer 0 in Figure 1.2) each of whose neurons receive a corresponding single component of the total input and distribute it to all neurons of the input layer. This layer may be seen as a *sensory layer* and (usually) doesn't do any (real) processing. In some other architectures the input layer is also the sensory layer. Unless otherwise specified it will be omitted.

input layer

sensory layer

- The last layer is called the *output layer*. Usually what we are interested in is the outputs of the output layer.

output layer

- The layers between input and output are called *hidden layers*.

hidden layer

1.2 Neuronal Activation Functions

Neurons as functions

Neurons behave as functions. Neurons transform an unbounded input activation $a_{(t)}$ at a time t into a bounded output *signal* $f(a_{(t)})$. Usually the function f is sigmoidal having a graph as in Figure 1.3. f is called the *activation* or *signal function*.

activation function

The most popular activation function is the *logistic* signal function: $f(a) = \frac{1}{1+e^{-ca}}$ which is sigmoidal and strictly increasing in a for positive scaling constant $c > 0$. Strict monotonicity derives from the positivity of the activation derivative: $f' \equiv \frac{df}{da} = cf(1-f) > 0$.

❖ c

The threshold signal function, whose graph is the dashed line in figure 1.3 on the preceding page, illustrates a non–differentiable signal function. Loosely speaking the logistic signal function approaches the threshold function as $c \to +\infty$. Then f maps positive activations signals a to unity signals and negative activations to zero signals. A discontinuity occurs at the *zero* activation value (which equals the signal function's "threshold"). *Zero* activation values seldom occur in large neural networks[4].

signal velocity

The *signal velocity* df/dt measures the signal's rate of change with respect to time; it is the product between the change in the signal due to the activation and the change in the activation with time: $\frac{df}{dt} = \frac{df}{da}\frac{da}{dt} = f' \, \frac{da}{dt}$.

Common activation functions

The following activation functions are more often encountered in practice:

logistic

1. *Logistic:* $f(a) = \frac{1}{1+e^{-ca}}$, where $c > 0$, $c = \text{const.}$ is a positive constant. The activation derivative is $f' \equiv \frac{df}{da} = cf(1-f)$ and so f is monotone increasing ($f > 0$). This function is the most common one. Note that f' may be expressed in terms of f itself.

hyperbolic tangent

2. *Hyperbolic tangent:* $f(a) = \tanh(ca) = \frac{e^{ca}-e^{-ca}}{e^{ca}+e^{-ca}}$, where $c > 0$ is a positive constant. The activation derivative is $f' \equiv \frac{df}{da} = c(1-f^2) > 0$ and so f is monotone increasing ($f < 1$); and again the derivative f' may be expressed through f.

threshold

3. *Approximate threshold:* $f(a) = \begin{cases} 1 & \text{if } a \geqslant \frac{1}{c} \\ 0 & \text{if } a < 0 \\ ca & \text{otherwise } (x \in [0, 1/c]) \end{cases}$, where $c > 0$, $c = \text{const.}$ is a positive constant. The activation derivative is: $f'(a) \equiv \frac{df}{da} = \begin{cases} 0 & \text{if } a \in (-\infty, 0) \cup (1/c, \infty) \\ c & \text{if } a \in (0, 1/c) \end{cases}$. Note that it is not a true threshold function as it has a non-infinite slope between 0 and c.

exponential distribution

4. *Exponential–distribution:* $f(a) = \max(0, 1 - e^{-ca})$, where $c > 0$, $c = \text{const.}$ is a positive constant. The activation derivative is: $f'(a) \equiv \frac{df}{da} = ce^{-ca}$, and for $a > 0$, supra–threshold signals are monotone increasing as $f' > 0$ (if $a < 0$ then $f(a) = 0$ and $f' = 0$). Note: since the second derivative $f'' = -c^2 e^{-ca} < 0$ the exponential–distribution function is strictly concave down.

ratio polynomial

5. *Ratio–polynomial:* $f(a) = \max\left(0, \frac{a^n}{c+a^n}\right)$, for $n > 1$, where $a, c > 0$, $c = \text{const.}$ The activation derivative is $f' \equiv \frac{df}{da} = \frac{cna^{n-1}}{(c+a^n)^2}$ and for positive activation signals this activation function is strictly increasing.

pulse coded
❖ g

6. *Pulse–coded:* $f(t) = \int\limits_{-\infty}^{t} g(s) \, e^{s-t} \, ds$ where $g(s) \equiv \begin{cases} 1 & \text{if a pulse occurs at } s \\ 0 & \text{if no pulse at } s \end{cases}$. The pulse–coded function represents the exponentially weighted time average of sampled binary pulses and takes values within $[0, 1]$.

   Proof. If there are no signals then then $g(s) = 0$, $\forall s$ thus $f(t) = 0$ (trivial). If $g(s) = 1$, $\forall s$ then $f(t) = \int_{-\infty}^{t} e^{s-t} ds = e^{t-t} - \lim\limits_{s \to -\infty} e^{s-t} = 1$. When the number of arriving pulses increases then the "pulse count" can only increase so f is monotone nondecreasing. □

[4] Threshold activation functions were used in early developments of ANN, e.g. perceptrons, however because they were not differentiable they represented an obstacle in the development of ANNs till the sigmoidal functions were adopted and gradient descent techniques (for weight adaptation) were developed.

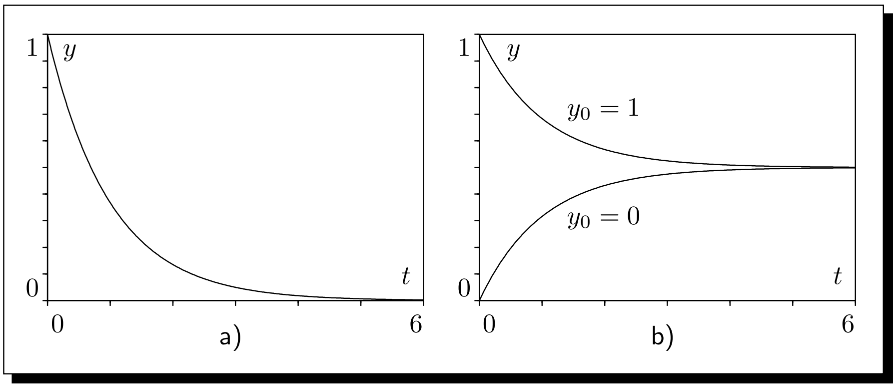

Figure 1.4: *a) Evolution in time of passive decay solution ($\alpha = 1$, $y_0 = 1$); b) Evolution in time of passive decay with resting potential for cases $y_0 = 0$ and $y_0 = 1$ ($\alpha = 1$, $\beta = 0.5$).*

The definition of this function is motivated by the fact that in biological neuronal systems the information seems to be carried by pulse trains rather than individual pulses. Train–pulse coded information can be decoded more reliably than shape–pulse coded information (arriving individual pulses can be somewhat corrupted in shape and still accurately decoded as present or absent).

1.3 Activation Dynamics

The previous section discussed models of neuronal response in which there is no delay or decay. In the current section we discuss models in which there is delay and/or decay and the neuronal outputs time–dependent behaviour is described by an equation of the form: $\frac{dy}{dt} = g(y, t)$ for some function g.

❖ y

First order passive decay model

The simplest model incorporating decay is to consider that the rate of decay of the output is proportional to the current output:, i.e. $\frac{dy}{dt} \propto -y$. This leads to the passive decay first–order differential equation, with initial condition $y_{(0)} = y_0$:

passive decay
❖ y_0

$$\frac{dy}{dt} = -\alpha y \quad \Rightarrow \quad y = y_0 e^{-\alpha t}$$

where $\alpha > 0$, $\alpha =$ const. is the passive decay rate. The output exponentially decreases to zero: $\lim_{t \to \infty} y_{(t)} = 0$. See figure 1.4–a.

❖ α

Proof. $\frac{dy}{dt} = -\alpha y \Rightarrow \frac{dy}{y} = -\alpha\, dt$, after integration $\ln y = -\alpha t +$ const. The initial condition gives const. $= \ln y_0$. □

Passive decay with resting potential

This is an enhanced version of the model above incorporating a possibly nonzero asymptotic

resting potential
❖ β

[1.3] See [Kos92] pp. 56–57.

neuronal output β/α, called the resting potential:

$$\frac{dy}{dt} = -\alpha y + \beta \quad \Rightarrow \quad y = y_0 e^{-\alpha t} + \frac{\beta}{\alpha}\left(1 - e^{-\alpha t}\right) \tag{1.1}$$

where $\alpha > 0$, $\alpha =$ const. The asymptotic neuronal output is $\lim\limits_{t\to\infty} y_{(t)} = \beta/\alpha$. See Figure 1.4–b.

Proof. The homogeneous equation $\frac{dy}{dt} + \alpha y = 0$ gives the solution $y = c_1 e^{-\alpha t}$. A particular solution to the nonhomogenous equation is of the form $y = c_1 e^{-\alpha t} + c_2$.

Substituting back into the differential equation gives $c_2 = \beta/\alpha$. The initial condition yields $c_1 + c_2 = y_0$ $\Rightarrow c_1 = x_0 - \frac{\beta}{\alpha}$. □

Remarks:

- In some ANN designs, β/α is replaced by an external input, considered stationary over the time required to reach the asymptotic solution.

1.4 New Matrix Operations

ANN involves heavy manipulations of large sets of numbers. It is most convenient to manipulate them as matrices or vectors (column matrices) and it is possible to express many ANN algorithms in matrix notation. Some of matrix operation that occur are common to several ANN algorithms and it makes sense to introduce new matrix operations for them in order to underline this.

The usage of matrix formulations in numerical simulations has the following advantages:

- splits the difficulty of implementations onto two levels: a lower one, involving matrix operations, and a higher one involving the ANNs themselves;
- leads to code reusage, with respect to matrix operations;
- makes implementation of new ANNs easier, once the basic matrix operations have been implemented;
- ANN algorithms, expressed through the new matrix operations, do *not* introduce any inefficiencies (e.g. unnecessary operations or waste of memory);
- makes heavy optimization of the basic matrix operations more desirable, as the resulting code is reusable; see [Mos97] and [McC97] for some ideas regarding optimizations; many systems have BLAS (Basic Linear Algebra Subprograms), a library for basic matrix operations, usually heavily optimized for the particular hardware is running on, see also the Strassen and related algorithms for fast matrix–matrix multiplication, there are also algorithms for matrix–vector multiplications when the matrix is structured (e.g. Toeplitz, circulant, etc.);

BLAS

- makes debugging of ANN implementations easier.

Definition 1.4.1. *We define the following additive operations where the matrices on the right have sizes such that the right hand side is well defined.*

❖ $\overset{R}{\oplus}$, $\overset{R}{\ominus}$, $\overset{C}{\oplus}$, $\overset{C}{\ominus}$

The addition/subtraction on rows/columns between a constant and a vector or a vector and a matrix is defined as follows:

a. *Addition/subtraction between a constant and a vector:*

$$a \overset{\mathrm{R}}{\oplus} \mathbf{x} := \widehat{\mathbf{1}}a + \mathbf{x} \qquad a \overset{\mathrm{C}}{\oplus} \mathbf{x}^{\mathrm{T}} := a\widehat{\mathbf{1}}^{\mathrm{T}} + \mathbf{x}^{\mathrm{T}}$$

$$a \overset{\mathrm{R}}{\ominus} \mathbf{x} := \widehat{\mathbf{1}}a - \mathbf{x} \qquad a \overset{\mathrm{C}}{\ominus} \mathbf{x}^{\mathrm{T}} := a\widehat{\mathbf{1}}^{\mathrm{T}} - \mathbf{x}^{\mathrm{T}}$$

b. *Addition/subtraction between a vector and a matrix:*

$$\mathbf{x}^{\mathrm{T}} \overset{\mathrm{R}}{\oplus} A := \widehat{\mathbf{1}}\mathbf{x}^{\mathrm{T}} + A \qquad \mathbf{x} \overset{\mathrm{C}}{\oplus} A := \mathbf{x}\widehat{\mathbf{1}}^{\mathrm{T}} + A$$

$$\mathbf{x}^{\mathrm{T}} \overset{\mathrm{R}}{\ominus} A := \widehat{\mathbf{1}}\mathbf{x}^{\mathrm{T}} - A \qquad \mathbf{x} \overset{\mathrm{C}}{\ominus} A := \mathbf{x}\widehat{\mathbf{1}}^{\mathrm{T}} - A$$

Remarks:

- ➥ These operations avoid an unnecessary expansion of a constant/vector to a vector/matrix, before doing an addition/subtraction.
- ➥ $\mathbf{x} \overset{\mathrm{R}}{\oplus} a$, $a \overset{\mathrm{R}}{\oplus} \mathbf{x}$, etc, are defined similarly.
- ➥ When the operation involves a constant then it represents in fact an addition/subtraction from all elements of the vector/matrix. In this situation $\overset{\mathrm{R}}{\oplus}$ is practically equivalent with $\overset{\mathrm{C}}{\oplus}$ and $\overset{\mathrm{R}}{\ominus}$ is equivalent with $\overset{\mathrm{C}}{\ominus}$ (and they could be replaced with something simpler, e.g. $\oplus$ and $\ominus$). However, it seems that not introducing separate operations keeps the formulas simpler and easier to follow.

Definition 1.4.2. *Addition/subtraction between a constant and the diagonal elements of a square matrix is defined as follows:* ❖ $\overset{\mathrm{I}}{\oplus}$, $\overset{\mathrm{I}}{\ominus}$

$$a \overset{\mathrm{I}}{\oplus} A := aI + A \qquad a \overset{\mathrm{I}}{\ominus} A := aI - A$$

Remarks:

- ➥ These operations avoid unnecessary operations on non–diagonal matrix elements.

Definition 1.4.3. *The Hadamard row/column product between a matrix and vector (row or column matrix) is defined as follows:* ❖ $\overset{\mathrm{R}}{\odot}$, $\overset{\mathrm{C}}{\odot}$

$$\mathbf{x}^{\mathrm{T}} \overset{\mathrm{R}}{\odot} A := \widehat{\mathbf{1}}\mathbf{x}^{\mathrm{T}} \odot A \qquad \textit{and} \qquad \mathbf{x} \overset{\mathrm{C}}{\odot} A := \mathbf{x}\widehat{\mathbf{1}}^{\mathrm{T}} \odot A$$

Remarks:

- ➥ These operations avoid expansion of vectors to matrices, before doing the Hadamard product.
- ➥ They seem to fill a gap between the operation of multiplication between a constant and a matrix and the Hadamard product.

❖ $\mathcal{H}$

Definition 1.4.4. *The (meta)operator $\mathcal{H}$ takes as arguments two matrices of the same size and three operators. Depending on the sign of elements of the first matrix, it applies one of the three operators to the corresponding elements of the second matrix. It returns the second matrix updated.*

Assuming that the two matrices are A and B, and the operators are $\widehat{\alpha}$, $\widehat{\beta}$ and $\widehat{\gamma}$ then $\mathcal{H}\{A, B, \widehat{\alpha}, \widehat{\beta}, \widehat{\gamma}\} = \widetilde{B}$, the elements of $\widetilde{B}$ being:

$$\widetilde{b}_{ij} = \begin{cases} \widehat{\alpha}(b_{ij}) \,, & \text{if } a_{ij} > 0 \,, \\ \widehat{\beta}(b_{ij}) \,, & \text{if } a_{ij} = 0 \,, \\ \widehat{\gamma}(b_{ij}) \,, & \text{if } a_{ij} < 0 \end{cases}$$

where a_{ij}, b_{ij} are the elements of A, respectively B.

Remarks:

- $\mathcal{H}$ may be replaced by some operations with the sign function when used in simulations. A straightforward implementation through "if" statements should be in general avoided (if possible) as it is slow. Note that for many algorithms two out of the three operators ($\widehat{\alpha}$, $\widehat{\beta}$ and $\widehat{\gamma}$) are equal, thus one bit is enough to store a flag for each corresponding a_{ij}.
- The $\widehat{\alpha}$, $\widehat{\beta}$ and $\widehat{\gamma}$ may be e.g.: "$= 1$" makes $b_{ij} = 1$, "$=$" leaves b_{ij} unchanged, "$c\cdot$" makes $\widetilde{b}_{ij} = c b_{ij}$, etc.

❖ $\oslash$

Definition 1.4.5. *Hadamard division is the inverse of Hadamard product. If $A \odot B = C$ then $A = C \oslash B$ and $B = C \oslash A$, (assuming $b_{ij} \neq 0$, respectively $a_{ij} \neq 0$).*

$(AB)^{\mathrm{T}} = B^{\mathrm{T}} A^{\mathrm{T}}$

Remarks:

- Note also that the matrix property $(AB)^{\mathrm{T}} = B^{\mathrm{T}} A^{\mathrm{T}}$ *is widely used.*

CHAPTER 2

The Backpropagation Network

The backpropagation network represents one of the most classical and widely used examples of ANN, being also one of the most simple in terms of overall design.

2.1 Network Architecture

The network consists of several layers of neurons. The first one (labelled layer 0 in Figure 2.1) distributes the inputs to first hidden layer 1. There is no processing in layer 0, it can be seen just as a *sensory* layer — each neuron receives just one component of the input (vector) $\mathbf{x}$ which gets distributed, unchanged, to all neurons from the input layer. The last layer gives the ANN output, from where the processed data, i.e. the $\mathbf{y}$ vector, is collected. The layers between sensory and output are called the hidden layers.

Remarks:

- From the above description, is becomes mandatory that layer 0 have the same number of neurons as the number of input components (dimension of input vector $\mathbf{x}$) and output layer have the same number of neurons as the desired output (target) has, i.e. the dimension of target $\mathbf{t}$ dictates the number of neurons on the output layer.
- The hidden layers may have any number of neurons, and in general this is not a critical issue. However, in some applications, their number may be chosen to be a particular value, in order to achieve a special effect.

The network is a straight feedforward network: each neuron receives as input the outputs of all neurons from the previous layer (except for the first sensory layer). See Figure 2.1 on the next page. feedforward ANN

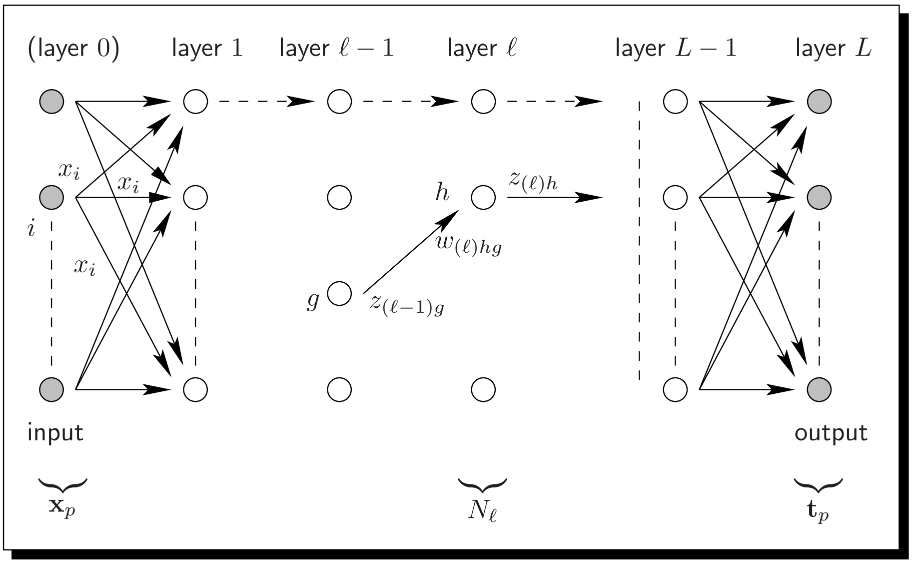

Figure 2.1: *The backpropagation network structure.*

❖ $z_{(\ell)h}$, $\mathbf{z}_{(\ell)}$, $\mathbf{z}_{(0)}$,
❖ $w_{(\ell)hg}$, $W_{(\ell)}$,
❖ N_ℓ, L

The following notations are used:

- $z_{(\ell)h}$ is the output of neuron h from layer ℓ.
  The output of layer ℓ is: $\mathbf{z}_{(\ell)}^{\mathrm{T}} = \begin{pmatrix} z_{(\ell)1} & \cdots & z_{(\ell)N_\ell} \end{pmatrix}$.
- $w_{(\ell)hg}$ is the weight applied to the output of neuron g from layer $\ell-1$ in calculating the input of neuron h from layer ℓ.

  Each layer has an associated matrix of weights $W_{(\ell)} = \begin{pmatrix} w_{(\ell)11} & \cdots & w_{(\ell)1N_{\ell-1}} \\ \vdots & \ddots & \vdots \\ w_{(\ell)N_\ell 1} & \cdots & w_{(\ell)N_\ell N_{\ell-1}} \end{pmatrix}$ and each neuron h, from layer ℓ has the associated weights $W_{(\ell)(h,:)}$.
- $z_{(0)i}$ is the i-th component of input vector. We adopt the notation $\mathbf{z}_{(0)} := \mathbf{x}_p$, a particular input pattern p.
- N_ℓ is the number of neurons in layer ℓ.
- L is the number of layers excluding the sensory layer. Thus the sensory layer is layer 0 and the output layer is layer L.

The learning set is $\{\mathbf{x}_p, \mathbf{t}_p\}_{p=\overline{1,P}}$.

2.2 Network Dynamics

2.2.1 Neuronal Output

The activation function used is usually the logistic function $f : \mathbb{R} \to (0,1]$:

$$f(a) = \frac{1}{1+\exp(-ca)} \quad ; \quad a \in \mathbb{R}\,,\ c > 0\,,\ c = \text{const.} \tag{2.1}$$

with:

$$\frac{df}{da} = \frac{c \exp(-ca)}{[1 + \exp(-ca)]^2} = cf(a)\,[1 - f(a)]\ . \tag{2.2}$$

However, the backpropagation algorithm is not restricted to this activation function.

2.2.2 Running Procedure

Each neuron computes as its output the weighted sum of its inputs to which it applies the activation function (see also Figure 2.1 on the facing page):

$$\mathbf{z}_{(\ell)} = f(W_{(\ell)}\mathbf{z}_{(\ell-1)}) \quad , \quad \ell = \overline{1, L} \tag{2.3}$$

where $\mathbf{z}_{(0)} \equiv \mathbf{x}$ and $\mathbf{a}_{(\ell)} = W_{(\ell)}\mathbf{z}_{(\ell-1)}$ is the total input to layer ℓ. Formula (2.3) must be applied in succession for each layer, starting from first hidden ($\ell = 1$) and going through all subsequent layers, until the last one (L) is reached. The network output is $\mathbf{y} \equiv \mathbf{z}_{(L)} = f(W_{(L)}\mathbf{z}_{(L-1)})$.

❖ $\mathbf{a}_{(\ell)}$, $\mathbf{z}_{(L)}$

2.2.3 Supervised Learning Using Delta Rule

The network learning process is supervised, i.e. the network receives, at training time, pairs of inputs and desired targets. *The learning involves adjusting weights so that error will be decreased* (approximately minimized). The function used to measure errors is usually the sum–of–squares defined, for one training pattern, as:

supervised learning

sum–of–squares

$$E_p(W) = \frac{1}{2}\,\|\mathbf{y}(\mathbf{x}_p) - \mathbf{t}_p\|^2 \tag{2.4}$$

where E_p is a function of the weights W, as $\mathbf{y} = \mathbf{y}(\mathbf{x}_p, W)$. If the number of training sets is large ($P \to \infty$), the dependence on $\{\mathbf{x}_p, \mathbf{t}_p\}$ should disappear. Note that other error functions can be used in the backpropagation algorithm.

✍ **Remarks:**

- ➥ Note that *all* components of input $\mathbf{x}$ will influence *any* component of output $\mathbf{y}$, thus $y_k = y_k(\mathbf{x})$.
- ➥ Considering all learning samples (the full training set) then the total *sum–of–squares error* $E(W)$ is defined as:

$$E(W) \equiv \sum_{p=1}^{P} E_p(W) = \frac{1}{2}\sum_{p=1}^{P} \|\mathbf{y}(\mathbf{x}_p) - \mathbf{t}_p\|^2 \tag{2.5}$$

The network weights are found (weights are adapted/changed) step by step. The error function $E, E_p : \mathbb{R}^{N_W} \to \mathbb{R}$ may be represented as a surface in the $\mathbb{R}^{N_W+1}$ space. The gradient vector $\nabla E = \left\{\frac{\partial E(W)}{\partial w_{(\ell)hg}}\right\}$ shows the direction of (local) maximum rate of increase and $\{w_{(\ell)hg}\}$ are to be changed in the *opposite* direction (i.e. in the direction $-\nabla E$). See also Figure 2.2 on the next page.

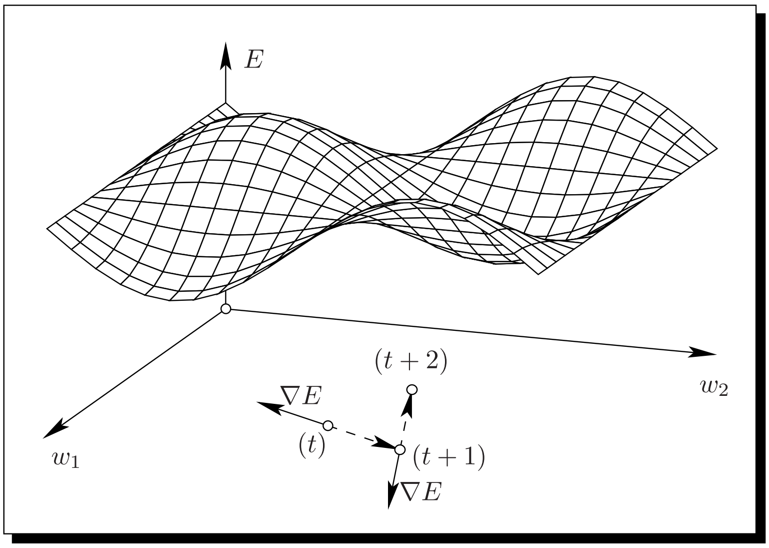

Figure 2.2: *$E(w)$ — Total square error as a function of the weights. ∇E shows the direction of (local) maximum rate of increase.*

In the discrete time approximation, at step $t+1$, given the weights at step t, the weights are adjusted according to:

$$W_{(t+1)} = W_{(t)} - \eta \nabla E \tag{2.6}$$

❖ η

where $\eta =$ const., $\eta > 0$ is called the learning constant and it is used for tuning the speed and quality of the learning process. Note that the error gradient may also be considered as a matrix or tensor, it has *the same dimensions* as W.

delta rule

(2.6) is called the *delta rule* and represents the basic learning in many ANN. More generally it is written as $\Delta W = W_{(t+1)} - W_{(t)} \propto -\nabla E$. Two possibilities with respect to delta rule usage are:

batch learning

- Calculate the error gradient with respect to the whole training set and then update the weights (and repeat) — this is called *batch* learning.

  Usually *the total error E represents a sum over all E_p contributions then the total error gradient ∇E will be also calculated by summation over all ∇E_p* (as the ∇ operator is linear); e.g. when the total error is specified by (2.5): $\nabla E = \sum_{p=1}^{P} \nabla E_p$.

on–line learning

- Update weights after calculating ∇E_p, for each training pattern in turn — this is called *on–line* learning.

Remarks:

➥ For on–line learning a shuffling of patterns is recommended between repeats. Also the on–line method allows for reduction in memory consumption (some patterns may be "put aside" until next usage) and enables a wider probing of weight space (when the next training pattern is chosen randomly from the learning set).

➥ Batch learning is not adequate when patterns represent a time series as the ordering information is lost.

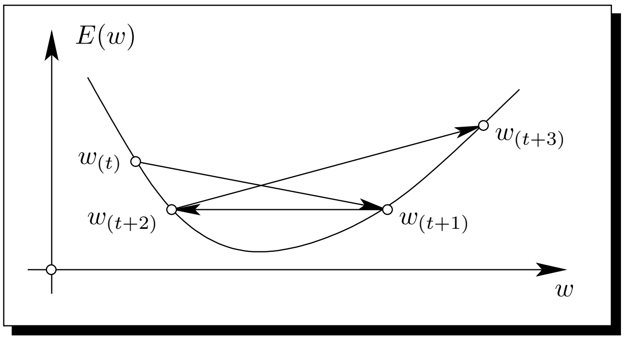

Figure 2.3: *Oscillations in learning process. The weights move around E minima without being able to reach it (arrows show the jumps made during learning).*

- The above method does not provide for a starting point. In practice, weights are initialized with small random values (usually in the interval $[-1, 1]$). The reason for randomness is to break a possible symmetry, while small values are chosen in order to avoid large (total) input into a neuron which could "saturate" it.
- If η is too small then the learning is slow and the learning process may stop at a *local* minima (of E), being unable to jump past a *local* maximum (it becomes trapped).
- If η is too large then the learning is fast but it may jump over *local* minima (of E) which may be deeper than the next one.
- Another point to consider is the problem of oscillations. When approaching error minima, a learning step may overshoot it, the next one may again overshoot it bringing back the weights to a similar point to previous one. The net result is that weights are changed to values around minima but never able to reach it. See figure 2.3. This problem is particularly likely to occur for deep and narrow minima because in this case ∇E is large (deep $\equiv$ steep) and subsequently ΔW is large (narrow $\equiv$ easy to overshoot).

The problem is to find the error gradient ∇E. A "standard" finite difference approach (i.e. $(\nabla E)_{(\ell)hg} \simeq \frac{\Delta E}{\Delta w_{(\ell)hg}}$ for some small $\Delta w_{(\ell)hg}$) would require an computational time of the order $\mathcal{O}(N_W^2)$, because each calculation of E requires $\mathcal{O}(N_W)$ operations and it has to be repeated for each $w_{(\ell)hg}$ in turn. The importance of the backpropagation algorithm resides in the fact that it reduces the operation count to compute ∇E to $\mathcal{O}(N_W)$, thus greatly improving the speed of learning.

Theorem 2.2.1. ***Backpropagation algorithm.*** *For each layer (except 0, input), an error gradient matrix may be build as follows:* $\nabla_{W_{(\ell)}} E_p := \begin{pmatrix} \frac{\partial E_p}{\partial w_{(\ell)11}} & \cdots & \frac{\partial E_p}{\partial w_{(\ell)1N_{\ell-1}}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E_p}{\partial w_{(\ell)N_\ell 1}} & \cdots & \frac{\partial E_p}{\partial w_{(\ell)N_\ell N_{\ell-1}}} \end{pmatrix}$, $1 \leqslant \ell \leqslant L$. *For each layer, except layer L, the error gradient vector, with respect to neuronal outputs, may be defined as:* $\nabla^{\mathrm{T}}_{\mathbf{z}_{(\ell)}} E_p := \begin{pmatrix} \frac{\partial E_p}{\partial z_{(\ell)1}} & \cdots & \frac{\partial E_p}{\partial z_{(\ell)N_\ell}} \end{pmatrix}$, $1 \leqslant \ell \leqslant L-1$.

backpropagation

❖ $\nabla_{W_{(\ell)}} E_p$

❖ $\nabla_{\mathbf{z}_{(\ell)}} E_p$

The error gradient with respect to network output $\mathbf{z}_{(L)}$ is considered to be known and dependent only on network outputs $\{\mathbf{z}_{(L)}(\mathbf{x}_p)\}$ and the set of targets $\{\mathbf{t}_p\}$. That is $\nabla_{\mathbf{z}_{(L)}} E_p$

is assumed known (it is dependent on definition of error and usually may be computed analytically).

Then, considering the error function E_p, the activation function f, and its total derivative f', the error gradient may be computed recursively according to formulas:

$$\nabla_{\mathbf{z}_{(\ell)}} E_p = W_{\ell+1}^{\mathrm{T}} \cdot \left[\nabla_{\mathbf{z}_{(\ell+1)}} E_p \odot f'(\mathbf{a}_{(\ell+1)})\right] \quad \textit{calculated for } \ell \textit{ from } L-1 \textit{ to } 1 \tag{2.7a}$$

$$\nabla_{W_{(\ell)}} E_p = \left[\nabla_{\mathbf{z}_{(\ell)}} E_p \odot f'(\mathbf{a}_{(\ell)})\right] \cdot \mathbf{z}_{(\ell-1)}^{\mathrm{T}} \quad \textit{for layers } \ell = 1 \textit{ to } \ell = L \tag{2.7b}$$

where $\mathbf{z}_0 \equiv \mathbf{x}$.

Proof. The error $E_p(W)$ is dependent on $w_{(\ell)hg}$ through the output of neuron $((\ell), h)$, i.e. $z_{(\ell)h}$ (the data flow is $\mathbf{x} \xrightarrow{w_{(\ell)hg}} \mathbf{z}_{(\ell)h} \to E_p$):

$$\frac{\partial E_p}{\partial w_{(\ell)hg}} = \frac{\partial E_p}{\partial z_{(\ell)h}} \frac{\partial z_{(\ell)h}}{\partial w_{(\ell)hg}} \quad \Rightarrow \quad \nabla_{W_{(\ell)}} E_p = \nabla_{\mathbf{z}_{(\ell)}} E_p \overset{\mathrm{C}}{\odot} \left\{ \frac{\partial z_{(\ell)h}}{\partial w_{(\ell)hg}} \right\}$$

and each derivative is computed separately.

a. Term $\frac{\partial z_{(\ell)h}}{\partial w_{(\ell)hg}}$ is:

$$\frac{\partial z_{(\ell)h}}{\partial w_{(\ell)hg}} = \frac{\partial}{\partial w_{(\ell)hg}} f((W_{(\ell)}\mathbf{z}_{(\ell-1)})_h) = f'(a_{(\ell)h}) \frac{\partial (W_{(\ell)}\mathbf{z}_{(\ell-1)})_h}{\partial w_{(\ell)hg}} = f'(a_{(\ell)h})\, z_{(\ell-1)g}$$

because weights are mutually independent $((W_{(\ell)}\mathbf{z}_{(\ell-1)})_h = W_{(\ell)(h,:)}\mathbf{z}_{(\ell-1)})$; and then:

$$\left\{ \frac{\partial z_{(\ell)h}}{\partial w_{(\ell)hg}} \right\} = f'(\mathbf{a}_{(\ell)})\, \mathbf{z}_{(\ell-1)}^{\mathrm{T}}$$

b. Term $\frac{\partial E_p}{\partial z_{(\ell)h}}$: neuron $((\ell)h)$ affects E_p through all following layers that are intermediate between layer (ℓ) and output (the influence being exercised through the interposed neurons, the data flow is $\mathbf{x} \to \mathbf{z}_{(\ell)} \to \mathbf{z}_{(\ell+1)} \to E_p$).

First affected is the next layer, layer $\ell+1$, through term $\frac{\partial z_{(\ell+1)g}}{\partial z_{(\ell)h}}$ (and then the dependency is carried on next successive layers) and, by similar means as for previous term:

$$\begin{aligned}
\frac{\partial E_p}{\partial z_{(\ell)h}} &= \sum_{g=1}^{N_{\ell+1}} \frac{\partial E_p}{\partial z_{(\ell+1)g}} \frac{\partial z_{(\ell+1)g}}{\partial z_{(\ell)h}} = \sum_{g=1}^{N_{\ell+1}} \frac{\partial E_p}{\partial z_{(\ell+1)g}} \frac{\partial}{\partial z_{(\ell)h}} f((W_{(\ell+1)}\mathbf{z}_{(\ell)})_g) \\
&= \sum_{g=1}^{N_{\ell+1}} \frac{\partial E_p}{\partial z_{(\ell+1)g}} f'(a_{(\ell+1)g}) \frac{\partial (W_{(\ell+1)}\mathbf{z}_{(\ell)})_g}{\partial z_{(\ell)h}} = \sum_{g=1}^{N_{\ell+1}} \frac{\partial E_p}{\partial z_{(\ell+1)g}} f'(a_{(\ell+1)g})\, w_{(\ell+1)gh} \\
&= (W_{(\ell+1)}^{\mathrm{T}}[\nabla_{\mathbf{z}_{(\ell+1)}} E_p \odot f'(\mathbf{a}_{(\ell+1)})])_h
\end{aligned}$$

which represents exactly the element h of column matrix $\nabla_{\mathbf{z}_{(\ell)}} E_p$ as built from (2.7a). The above formula applies iteratively from layer $L-1$ to 1; for layer L, $\nabla_{\mathbf{z}_{(L)}} E_p$ is assumed known.

Finally, the required derivative is: $\nabla_{W_{(\ell)}} E_p = \nabla_{\mathbf{z}_{(\ell)}} E_p \overset{\mathrm{C}}{\odot} [f'(\mathbf{a}_{(\ell)})\, \mathbf{z}_{\ell-1}^{\mathrm{T}}]$, i.e. exactly (2.7b). □

Proposition 2.2.1. *In the important special case of the logistic activation function and sum–of–squares error function the error gradient may be computed recursively according to formulas:*

$$\nabla_{\mathbf{z}_{(L)}} E_p = \mathbf{z}_{(L)}(\mathbf{x}_p) - \mathbf{t}_p \tag{2.8a}$$

$$\nabla_{\mathbf{z}_{(\ell)}} E_p = c W_{(\ell+1)}^{\mathrm{T}} \cdot \left[\nabla_{\mathbf{z}_{(\ell+1)}} E_p \odot \mathbf{z}_{(\ell+1)} \odot (1 \overset{\mathrm{R}}{\ominus} \mathbf{z}_{(\ell+1)}) \right] \quad \textit{for } \ell \textit{ from } L-1 \textit{ to } 1 \qquad (2.8b)$$

$$\nabla_{W_{(\ell)}} E_p = c \left[\nabla_{\mathbf{z}_{(\ell)}} E_p \odot \mathbf{z}_{(\ell)} \odot (1 \overset{\mathrm{R}}{\ominus} \mathbf{z}_{(\ell)}) \right] \cdot \mathbf{z}_{(\ell-1)}^{\mathrm{T}} \quad \textit{for } \ell \textit{ from } 1 \textit{ to } L \qquad (2.8c)$$

where $\mathbf{z}_0 \equiv \mathbf{x}$ *and* c *is the activation function constant.*

Proof. From the definition (2.4) of sum–of–squares: $\nabla_{\mathbf{z}_{(L)}} E_p = \mathbf{z}_{(L)}(\mathbf{x}_p) - \mathbf{t}_p$.

As $f(\mathbf{a}_\ell) = \mathbf{z}_\ell$, by using (2.2) in the main results (2.7a) and (2.7b) of theorem 2.2.1, the other two formulas are deducted immediately. □

2.2.4 Initialization and Stopping Criteria

Weights are initialized (in practice) with small random values and the adjusting process continues by iteration, using either batch or on–line learning.

The learning process may be stopped by one of the following methods:

① a fixed total number T of steps is chosen;

② the learning process continues until the adjusting quantity $\Delta w_{(\ell)hg} = w_{(\ell)hg(t+1)} - w_{(\ell)hg(t)}$ is under some specified magnitude, $\forall \ell$, $\forall h$, $\forall g$;

③ the learning stops when the total error stops decreasing on a *test set*, *not* used for learning.

Remarks:

➥ If the trained network performs well on the training set but has bad results on previously unseen patterns (i.e. it has poor generalization capabilities) then this is usually a sign of overtraining (assuming, of course, that the network is reasonably built and there are a sufficient number of training patterns). The network memorized the learning set rather than generalized out of it. In this situation it is recommended either to reduce the size of ANN or to use an early stop, when errors on the test set are at a minimum.

overtraining

early stop

2.3 The Algorithm

The algorithm is based on discrete time approximation, i.e. time is $t = 0, 1, 2, \ldots$.

The activation, error functions, and the stopping condition are presumed to be chosen (known) and fixed. The type of learning, i.e. batch or on–line, is also chosen here.

Network running procedure:

1. The sensory layer 0 is initialised: $\mathbf{z}_0 \equiv \mathbf{x}$.
2. For all layers ℓ from 1 to L, do: $\mathbf{z}_{(\ell)} = f(W_{(\ell)} \mathbf{z}_{(\ell-1)})$.
3. The network output is taken to be: $\mathbf{y} \equiv \mathbf{z}_{(L)}$.

Network learning procedure:

1. Initialize all $W_{(\ell)}$ weights with (small) random values and choose the learning constant (see the remarks, below, for possible values).
2. For all training sets $\{\mathbf{x}_p, \mathbf{t}_p\}$ (as long as the *stopping* condition is not met), do:
   (a) Run the network to find the network outputs $\mathbf{z}_{(\ell)}$ and derivatives $f'(\mathbf{a}_{(\ell)})$.

   Note that the algorithm requires the derivatives of activation functions for all neurons. For most activation functions this may be expressed in terms of activation itself as is the case for logistic, see (2.2). This approach may reduce the memory usage or increase speed (or both, in the case of logistic function).

   (b) Using $\{\mathbf{y}(\mathbf{x}_p), \mathbf{t}_p\}$, calculate $\nabla_{\mathbf{z}_{(L)}} E_p$, e.g. for sum–of–squares use (2.8a).

   (c) Compute the error gradient:
   - For output layer $\nabla_{W_{(\ell)}} E_p$ is calculated directly from (2.7b) (or from (2.8c) for sum–of–squares and logistic).
   - For all other layers ℓ, going *backwards* from $L-1$ to 1: calculate first $\nabla_{\mathbf{z}_{(\ell)}} E_p$ using (2.7a) (or (2.8b) for sum–of–squares and logistic) and then find $\nabla_{W_{(\ell)}} E_p$ using (2.7b) (or respectively (2.8c)).

   (d) Update the weights according to the *delta rule* (2.6).

   (e) Check the *stopping* condition and exit if it has been met.

Remarks:

➥ Trying to stop backpropagation of error when components are below some thresh-
❖ e old value e may also improve learning; e.g., in case of sum–of–squares, the $\nabla_{\mathbf{z}_{(L)}} E_p$ may be changed to: $\frac{\partial E_p}{\partial z_{(L)k}} = \begin{cases} z_{(L)k} - t_k & \text{if } |z_{(L)k} - t_k| > e \\ 0 & \text{otherwise} \end{cases}$:

$$(\nabla_{\mathbf{z}_{(L)}} E_p)_{\text{new}} = \mathcal{H}\{\text{abs}[\mathbf{z}_{(L)}(\mathbf{x}_p) - \mathbf{t}_p] \overset{\text{R}}{\ominus} e, (\nabla_{\mathbf{z}_{(\ell)}} E_p)_{\text{old}}, =, = 0, = 0\}$$

i.e. rounding towards 0 the elements of $\nabla_{\mathbf{z}_L} E_p$ smaller than e. Note that this technique will generate a pseudo-gradient of error, *different from the true one*, so it should be used with care.

➥ A classical way of approximating the gradient (of a smooth function) is the central difference:

$$\frac{\partial E_p}{\partial w_{(\ell)hg}} = \frac{E_p(w_{(\ell)hg} + \varepsilon) - E_p(w_{(\ell)hg} - \varepsilon)}{2\varepsilon} + \mathcal{O}(\varepsilon^2) \quad , \quad \text{for an } \varepsilon \gtrsim 0$$

and while too slow for direct usage, is an excellent tool for checking the correctness of algorithm implementation.

➥ There are not (yet) good theoretical methods of choosing the learning parameters η and e. The practical, hands-on, approach is still the best. Usual values for η are in the range $[0.1, 1]$ (but some networks may learn even faster with $\eta > 1$) and $[0, 0.1]$ for e.

➥ In accordance with neuron output function (2.1) the output of the neuron has values within $(0, 1)$. In practice, due to rounding errors, the range is in fact $[0, 1]$.

If the desired outputs have values within $[0, \infty)$ then the following transformation may be used: $g(y) = 1 - \exp(-\alpha y)$ ($\alpha > 0$, $\alpha =$ const.) which has the inverse: $g^{-1}(y) = \frac{1}{\alpha} \ln \frac{1}{1-y}$. Use g to transform targets to $(0, 1)$ and g^{-1} to trasform network output to targets.

The same procedure may be used for inputs and each time the desired input/output falls beyond neuron activation function range.

- Reaching the *absolute* minima of E is by no means guaranteed. First the training set is limited and the error minima with respect to the learning set will generally not coincide with the minima considering all *possible* patterns, but in most cases should be close enough for practical applications.

  On the other hand, the error surface has some symmetries, e.g. swapping two neurons from the same layer (or in fact their weights) will not affect the network performance, so the algorithm will not search through the whole weight space but rather a small area of it. This is also the reason why the starting point, given randomly, will not affect substantially the learning process.

- In digital simulation using batch backpropagation: usually the $\nabla_{\mathbf{z}_\ell} E_p$ vector is calculated using a matrix–vector multiplication. As the weights matrix does not change during current epoch and using the properties of block matrices it would be possible to calculate all $\nabla_{\mathbf{z}_\ell} E_p$ at once. To do so, stack all $\nabla_{\mathbf{z}_\ell} E_p$ (and separately the corresponding vectors from the right side of equations) in a matrix, as columns, thus transforming the equation into a matrix–matrix multiplication. Now it becomes possible to apply fast matrix-matrix multiplications algorithms (e.g. Strassen and related). The trade-off is an increase in memory consumption.

2.4 Bias

Some problems, while having an obvious solution, cannot be solved with the architecture described above[1]. To solve these, the neuronal activation (2.3) is changed to:

$$z_{(\ell)h} = f(w_{(\ell)h0} + (W_{(\ell)} \mathbf{z}_{(\ell-1)})_h) \tag{2.9}$$

and the new parameter $w_{(\ell)h0}$ introduced is called *bias*. bias ❖ $w_{(\ell)h0}$

As may be seen immediately, the change is equivalent to inserting a new neuron $z_{(\ell)0}$ — on all layers except output — whose activation is *always* 1. See Figure 2.4 on the next page.

The required changes in neuronal outputs and weight matrices are:

$$\widetilde{\mathbf{z}}^{\mathrm{T}}_{(\ell)} = \begin{pmatrix} 1 & z_{(\ell)1} & \cdots & z_{(\ell)N_\ell} \end{pmatrix} \quad \text{and} \quad \widetilde{W}_{(\ell)} = \begin{pmatrix} w_{(\ell)10} & w_{(\ell)11} & \cdots & w_{(\ell)1N_{\ell-1}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{(\ell)N_\ell 0} & w_{(\ell)N_\ell 1} & \cdots & w_{(\ell)N_\ell N_{\ell-1}} \end{pmatrix} \tag{2.10}$$

biases being added as a first column in $W_{(\ell)}$, and then the neuronal output is calculated as $\mathbf{z}_{(\ell)} = f(\mathbf{a}_{(\ell)}) = f(\widetilde{W}_{(\ell)} \, \widetilde{\mathbf{z}}_{(\ell-1)})$.

The error gradient matrix $\nabla_{\widetilde{W}_{(\ell)}} E_p$ associated with $\widetilde{W}_{(\ell)}$ is: ❖ $\nabla_{\widetilde{W}_{(\ell)}} E_p$

[1] E.g. the tight encoder described in section 2.6.2.

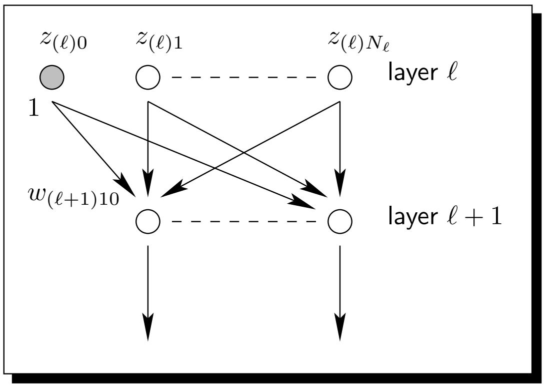

Figure 2.4: *Bias may be emulated with the help of an additional neuron $z_{(\ell)0}$ whose output is always 1 and is distributed to all neurons from next layer (exactly as for a "regular" neuron).*

$$\nabla_{\widetilde{W}_{(\ell)}} E_p = \begin{pmatrix} \frac{\partial E_p}{\partial w_{(\ell)10}} & \frac{\partial E_p}{\partial w_{(\ell)11}} & \cdots & \frac{\partial E_p}{\partial w_{(\ell)1N_{\ell-1}}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial E_p}{\partial w_{(\ell)N_\ell 0}} & \frac{\partial E_p}{\partial w_{(\ell)N_\ell 1}} & \cdots & \frac{\partial E_p}{\partial w_{(\ell)N_\ell N_{\ell-1}}} \end{pmatrix} \tag{2.11}$$

Following the changes from above, the backpropagation theorem becomes:

backpropagation

Theorem 2.4.1. ***Backpropagation with biases.*** *If the error gradient with respect to neuronal outputs $\nabla_{\mathbf{z}_{(L)}} E_p$ is known, and depends only on (actual) network outputs $\{\mathbf{z}_{(L)}(\mathbf{x}_p)\}$ and targets $\{\mathbf{t}_p\}$: $\nabla_{\mathbf{z}_{(L)}} E_p =$ known., then the error gradient (with respect to weights) may be calculated recursively according to formulas:*

$$\nabla_{\mathbf{z}_{(\ell)}} E_p = W_{(\ell+1)}^{\mathrm{T}} \cdot \left[\nabla_{\mathbf{z}_{(\ell+1)}} E_p \odot f'(\mathbf{a}_{(\ell+1)})\right] \quad \textit{calculated from } L-1 \textit{ to } 1 \tag{2.12a}$$

$$\nabla_{\widetilde{W}_{(\ell)}} E_p = \left[\nabla_{\mathbf{z}_{(\ell)}} E_p \odot f'(\mathbf{a}_{(\ell)})\right] \cdot \widetilde{\mathbf{z}}_{(\ell-1)}^{\mathrm{T}} \quad \textit{for layers } \ell \textit{ from } 1 \textit{ to } L \tag{2.12b}$$

where $\mathbf{z}_{(0)} \equiv \mathbf{x}$.

Proof. The proof is very similar to that of Theorem 2.2.1.

Equation (2.12a) results directly from (2.7a).

Considering the expression (2.10) of $\mathbf{z}_{(\ell-1)}$ and the form (2.11) of $\nabla_{\widetilde{W}_{(\ell)}} E_p$ then columns 2 to $N_{\ell-1}+1$ of $\nabla_{\widetilde{W}_{(\ell)}} E_p$ represents $\nabla_{W_{(\ell)}} E_p$ given by (2.7b):

$$\left\{\nabla_{\widetilde{W}_{(\ell)}} E_p\right\}_{(:,2:N_{\ell-1}+1)} = \nabla_{W_{(\ell)}} E_p = \left[\nabla_{\mathbf{z}_{(\ell)}} E_p \odot f'(\mathbf{a}_{(\ell)})\right] \cdot \mathbf{z}_{(\ell-1)}^{\mathrm{T}}$$

The only terms to be calculated remains $\left\{\nabla_{\widetilde{W}_{(\ell)}} E_p\right\}_{(:,1)}$, i.e. those of the first column, of the form $\frac{\partial E_p}{\partial w_{(\ell)h0}}$. But these terms may be written as (see proof of Theorem 2.2.1): $\frac{\partial E_p}{\partial w_{(\ell)h0}} = \frac{\partial E_p}{\partial z_{(\ell)h}} \frac{\partial z_{(\ell)h}}{\partial w_{(\ell)h0}}$ where $\frac{\partial E_p}{\partial z_{(\ell)h}}$ is term h of $\nabla_{\mathbf{z}_{(\ell)}} E_p$, already calculated, and from (2.9): $\frac{\partial z_{(\ell)h}}{\partial w_{(\ell)h0}} = f'(a_{(\ell)h}) \cdot 1 = f'(a_{(\ell)h})$, thus:

$$\left\{\nabla_{\widetilde{W}_{(\ell)}} E_p\right\}_{(:,1)} = \nabla_{\mathbf{z}_{(\ell)}} E_p \odot f'(\mathbf{a}_{(\ell)})$$

As $\widetilde{z}_{(\ell-1)1} \equiv 1$ and $\widetilde{\mathbf{z}}_{(\ell-1)(2:N_{\ell+1})} = \mathbf{z}_{(\ell)}$ (by construction), then formula (2.12b) follows. □

Proposition 2.4.1. *If using the logistic activation function and sum–of–squares error function then the error gradient with bias may be computed recursively using formulas:*

$$\nabla_{\mathbf{z}_{(L)}} E_p = \mathbf{z}_{(L)}(\mathbf{x}_p) - \mathbf{t}_p \tag{2.13a}$$

$$\nabla_{\mathbf{z}_{(\ell)}} E_p = c W^{\mathrm{T}}_{(\ell+1)} \cdot \left[\nabla_{\mathbf{z}_{(\ell+1)}} E_p \odot \mathbf{z}_{(\ell+1)} \odot (1 \overset{\mathrm{R}}{\ominus} \mathbf{z}_{(\ell+1)}) \right] \quad \text{for } \ell \text{ from } L-1 \text{ to } 1 \tag{2.13b}$$

$$\nabla_{\widetilde{W}_{(\ell)}} E_p = c \left[\nabla_{\mathbf{z}_{(\ell)}} E_p \odot \mathbf{z}_{(\ell)} \odot (1 \overset{\mathrm{R}}{\ominus} \mathbf{z}_{(\ell)}) \right] \cdot \widetilde{\mathbf{z}}^{\mathrm{T}}_{(\ell-1)} \quad \text{for } \ell \text{ from } 1 \text{ to } L \tag{2.13c}$$

where $\mathbf{z}_{(0)} \equiv \mathbf{x}$.

Proof. The argument is analogous to that of Proposition 2.2.1 but using Theorem 2.4.1 instead of Theorem 2.2.1. □

Remarks:

- The algorithm for a backpropagation ANN with biases is (mutatis mutandis) identical to the one described in Section 2.3.
- In practice, biases are usually initialized with 0.

2.5 Algorithm Enhancements

All algorithms presented here (from the simple momentum to the more sophisticated SuperSAB) may be seen as predictors of error surface features. Based on previous behaviour of error gradient, they try to predict the future and change the learning path accordingly.

Note that there are also more sophisticated algorithms to improve speed of learning (e.g. conjugate gradients) described in the second part of this book (because they require more advanced knowledge introduced later).

2.5.1 Momentum

The weight adaption described in standard ("vanilla") backpropagation (see section 2.2.3) is very sensitive to small perturbations. If a small "bump" appears in the error surface the algorithm is unable to jump over it and it will change direction.

This situation can often be avoided by taking into account the previous adaptations in the learning process (see also (2.6)):

$$\Delta W_{(t)} = W_{(t+1)} - W_{(t)} = -\,\eta \nabla E|_{W_{(t)}} + \mu \Delta W_{(t-1)} \tag{2.14}$$

the procedure being called *backpropagation with momentum*, and $\mu \in [0,1)$ is called the momentum (learning) parameter. momentum ❖ μ

The algorithm is very similar (mutatis mutandis) to the one described in Section 2.3. As the main memory consumption is given by the requirement to store the weight matrix

[2.5.1] See [BTW95] p. 50.

(especially true for large ANN), the momentum algorithm requires double the amount of standard backpropagation, to store ΔW for the next step.

Remarks:

- for on–line learning method, (2.14) changes to ($\nabla E \to \nabla E_p$):

$$\Delta W_{(t)} = W_{(t+1)} - W_{(t)} = -\,\eta \nabla E_p|_{W_{(t)}} + \mu \Delta W_{(t-1)}$$

- When choosing the momentum parameter the following results have to be considered:
  - if $\mu \geqslant 1$ then the contribution of each $\Delta w_{(\ell)hg}$ grows infinitely.
  - if $\mu \gtrsim 0$ then the momentum contribution is insignificant.

  so μ should be chosen somewhere in $[0.5, 1)$ (in practice, usually $\mu \simeq 0.9$).
- The momentum method assumes that the error gradient *slowly* decreases when approaching the absolute minimum. If this is not the case then the algorithm may jump over it.

flat spot elimination

- Another improvement over momentum is flat spot elimination. If the error surface is very flat then $\nabla E \simeq \widetilde{0}$ and subsequently $\Delta W \simeq \widetilde{0}$. This may lead to a very slow learning due to the increased number of training steps required. To avoid this problem, a change to the calculation of error gradient (2.7b) may be performed as follows:

$$\widetilde{\nabla}_{W_{(\ell)}} E_p = \left\{ \nabla_{\mathbf{z}_{(\ell)}} E_p \odot \left[f'(\mathbf{a}_{(\ell)}) \overset{\mathrm{R}}{\oplus} c_f \right] \right\} \cdot \mathbf{z}^{\mathrm{T}}_{(\ell-1)}$$

❖ c_f

  where $\widetilde{\nabla}_{W_{(\ell)}} E_p$ *is no more the real* $\nabla_{W_{(\ell)}} E_p$, but a pseudo-gradient. The constant c_f is called the flat spot elimination constant.

  Several points should be noted here:

  - The procedure of *adding* a term to f' instead of multiplying it means that $\widetilde{\nabla}_{W_{(\ell)}} E_p$ is more affected when f' is smaller — a desirable effect.
  - An error gradient term corresponding to a weight close to first hidden layer is smaller than a similar term for a weight near to output, because the effect of changing the weight gets attenuated when propagated through layers. So, another effect of c_f is to speedup weight adaptation in layers close to input, again a desirable effect.
  - The formulas (2.8c), (2.12b) and (2.13c) change the same way:

$$\widetilde{\nabla}_{W_{(\ell)}} E_p = c \left\{ \nabla_{\mathbf{z}_{(\ell)}} E_p \odot \left[\mathbf{z}_{(\ell)} \odot (1 \overset{\mathrm{R}}{\ominus} \mathbf{z}_{(\ell)}) \overset{\mathrm{R}}{\oplus} c_f \right] \right\} \cdot \mathbf{z}^{\mathrm{T}}_{(\ell-1)}$$

$$\widetilde{\nabla}_{\widetilde{W}_{(\ell)}} E_p = \left\{ \nabla_{\mathbf{z}_{(\ell)}} E_p \odot \left[f'(\mathbf{a}_{(\ell)}) \overset{\mathrm{R}}{\oplus} c_f \right] \right\} \cdot \widetilde{\mathbf{z}}^{\mathrm{T}}_{(\ell-1)}$$

$$\widetilde{\nabla}_{\widetilde{W}_{(\ell)}} E_p = c \left\{ \nabla_{\mathbf{z}_{(\ell)}} E_p \odot \left[\mathbf{z}_{(\ell)} \odot (1 \overset{\mathrm{R}}{\ominus} \mathbf{z}_{\ell}) \overset{\mathrm{R}}{\oplus} c_f \right] \right\} \cdot \widetilde{\mathbf{z}}^{\mathrm{T}}_{(\ell-1)}$$

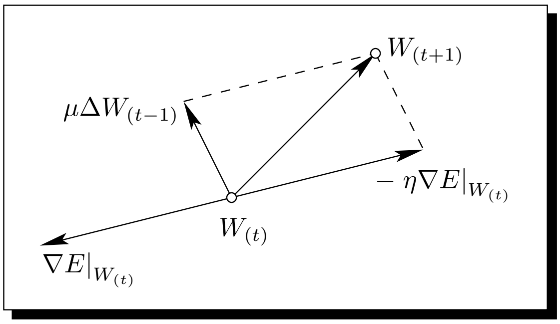

Figure 2.5: *Learning with momentum. A contributing term from the previous step is added.*

➥ In terms of physics: The set of weights W may be seen as a set of coordinates defining a point in the $\mathbb{R}^{N_W}$ space. During learning, this point is moved so as to reduce the error E (into error surface "valleys"). The momentum introduces an "inertia" proportional to μ such that, when changing direction under the influence of ∇E "force", it has a tendency to keep the old direction of movement and "overshoot" the point given by $-\eta\nabla E$. See figure 2.5.

The momentum method assumes that if the weights have been moved in some direction then this direction is good for the next steps and is kept as a trend: unwinding the weight adaptation over 2 steps (applying (2.14) twice, for $t-1$ and t) gives:

$$\Delta W_{(t)} = -\eta \left.\nabla E\right|_{W_{(t)}} - \mu\eta \left.\nabla E\right|_{W_{(t-1)}} + \mu^2 \Delta W_{(t-2)}$$

and it may be seen that the contributions of previous ΔW gradually disappear with the increase of power of μ (as $0 < \mu < 1$).

2.5.2 Quick Backpropagation

The main idea of quick backpropagation is to use momentum with a self–adaptive parameter calculated considering the hypotheses:

- error may be *locally* approximated by a special type of quadratic function of the weights, and the weights are assumed independent[2]:

$$E \simeq \widehat{\mathbf{1}}^{\mathrm{T}} \left(A + B \odot W + C \odot W^{\odot 2}\right) \widehat{\mathbf{1}} \quad , \quad A, B, C = \text{const.}$$

- in discrete–time approximation, local minima is attained at each step.

This leads to the weight update formula:

$$\Delta W_{(t+1)} = -\eta \nabla E|_{W_{(t)}} - \left[\nabla E|_{W_{(t)}} \oslash \left(\nabla E|_{W_{(t)}} - \nabla E|_{W_{(t-1)}}\right)\right] \odot \Delta W_{(t)}$$

[2.5.2] See [Has95] pp. 214–215 and [Bis95] pp. 270–271.

[2] The assumption of weights independence is equivalent to the Hessian being diagonal (see second part of this book for the definition of Hessian), i.e. off diagonal terms are considered zero; this is in most cases not true at a global level but as a local approximation may work in many cases.

Proof. Under the above hypotheses:

$$E = \hat{\mathbf{1}}^{\mathrm{T}} \left(A + B \odot W + C \odot W^{\odot 2} \right) \hat{\mathbf{1}} \quad \Rightarrow \quad \nabla E = B + 2C \odot W$$

$$\Rightarrow \quad \begin{cases} \nabla E|_{W_{(t)}} &= B + 2C \odot W_{(t)} \\ \nabla E|_{W_{(t-1)}} &= B + 2C \odot W_{(t-1)} \end{cases} \quad \Rightarrow \quad 2C = \left(\nabla E|_{W_{(t)}} - \nabla E|_{W_{(t-1)}} \right) \oslash \Delta W_{(t)}$$

Attaining minimum (as assumed) means:

$$\text{minimum} \quad \Rightarrow \quad \nabla E = \widetilde{0} \quad \Rightarrow \quad W = -B \oslash (2C)$$

From the error gradient at t:

$$B = \nabla E|_{W_{(t)}} - \left[\left(\nabla E|_{W_{(t)}} - \nabla E|_{W_{(t-1)}} \right) \oslash \Delta W_{(t)} \right] \odot W_{(t)}$$

and then the weights update formula is:

$$W_{(t+1)} = -B \oslash (2C) \quad \Rightarrow \quad \Delta W_{(t+1)} = -\left[\nabla E|_{W_{(t)}} \oslash \left(\nabla E|_{W_{(t)}} - \nabla E|_{W_{(t-1)}} \right) \right] \odot \Delta W_{(t)} \quad \square$$

Remarks:

➥ The algorithm assumes that the quadratic has a minimum at the stationary point. If it has a maximum or a saddle then the weights may be updated in the wrong direction.

➥ The algorithm starts at step $t = 1$ with $\Delta W_{(0)} = \widetilde{0}$.

➥ Into a region of nearly flat error surface it is necessary to limit the size of the weights update, otherwise the algorithm may jump over minima, especially if it is narrow; usually the relative weight increase is limited to a value μ found empirically.

❖ μ

➥ The "η" term is used to pull out weights from the assumed local minima.

➥ Typical values: $\eta \in [0.1, 0.3]$, maximum amount of weights change (relative to 1) $\mu \in [1.75, 2.25]$.

➥ The algorithm is applicable to general feedforward networks but error gradient is calculated fastest through Theorems 2.2.1 or 2.4.1 for layered feedforward ANN.

2.5.3 Adaptive Backpropagation

The main idea of this algorithm came from the following observations:

- If the slope of the error surface is gentle then a big learning parameter *could* be used to speedup learning over flat spot areas.
- If the slope of the error surface is steep then a small learning parameter *should* be used to avoid overshooting the error minima.
- In general the slopes are gentle in some directions and steep in others.

This algorithm is based on assigning individual learning rates for each weight $w_{(\ell)hg}$ based on previous behavior. This means that the learning constant η becomes a matrix of the same dimension as W.

[2.5.3] See [BTW95] p. 50.

The learning rate is increased if the gradient kept the same direction over last two steps (i.e. is likely to continue) and decreased otherwise:

$$\eta_{(\ell)hg(t)} = \begin{cases} c_i \eta_{(\ell)hg(t-1)} & \text{if } \Delta w_{(\ell)hg(t)}\, \Delta w_{(\ell)hg(t-1)} \geqslant 0 \\ c_d \eta_{(\ell)hg(t-1)} & \text{if } \Delta w_{(\ell)hg(t)}\, \Delta w_{(\ell)hg(t-1)} < 0 \end{cases} \quad \Leftrightarrow \tag{2.15}$$

$$\eta_{(t)} = \mathcal{H}\{\Delta W_{(t)} \odot \Delta W_{(t-1)}, \eta_{(t-1)}, c_i\cdot, c_i\cdot, c_d\cdot\}$$

where $c_i \geqslant 1$ and $c_d \in (0,1)$. The c_i parameter is called the adaptive increasing factor and the c_d parameter is called the adaptive decreasing factor. ❖ c_i, c_d

Remarks:

➥ If $c_i = 1$ and $c_d = 1$ then there is no adaptation.

In practice $c_i \in [1.1, 1.3]$ and $c_d \lesssim 1/c_i$ gives good results for a wide spectrum of applications.

➥ $\eta_{(0)}$ is initialized to a constant matrix $\eta_0 \widetilde{1}$ and $\Delta W_{(t-1)} = \widetilde{0}$. η is updated *after* each training session (considering the current $\Delta W_{(t)}$). For the rest, the same main algorithm as described in Section 2.3 applies. ❖ η_0

Note that, after initialization when $\eta(0) = \eta_0 \widetilde{1}$ and $\Delta W_{(0)} = \widetilde{0}$, the first step will lead automatically to the increase $\eta_{(1)} = c_i \eta_0 \widetilde{1}$, so η_0 should be chosen accordingly.

Also, this algorithm requires three times as much memory as the standard back-propagation: to store η and ΔW for next step, both being of the same size as W.

➥ By using just standard matrix operations, equation (2.15) may be written as:

$$\eta_{(t)} = \left\{(c_i - c_d)\ \text{sign}\left[\text{sign}(\Delta W_{(t)} \odot \Delta W_{(t-1)}) + \widetilde{1}\right] + c_d \widetilde{1}\right\} \odot \eta_{(t-1)} \tag{2.16}$$

Proof. The problem is to build a matrix containing 1's corresponding to each:

$$\Delta w_{(\ell)hg(t)} \Delta w_{(\ell)hg(t-1)} \geqslant 0$$

and 0's in rest. This matrix multiplied by c_i will be used to increase the corresponding $\eta_{(\ell)hg}$ elements. The complementary matrix will be used to decrease the matching $\eta_{(\ell)hg}$.

The $\text{sign}(\Delta W_{(t)} \odot \Delta W_{(t-1)})$ matrix has elements consisting only of 1, 0 and -1. By adding $\widetilde{1}$ and taking the sign again, all 1 and 0 elements are transformed to 1 while the -1 elements are transformed to 0. So, the desired matrix is:

$$\text{sign}\left[\text{sign}(\Delta W_{(t)} \odot \Delta W_{(t-1)}) + \widetilde{1}\right]$$

while its complement is:

$$\widetilde{1} - \text{sign}\left[\text{sign}(\Delta W_{(t)} \odot \Delta W_{(t-1)}) + \widetilde{1}\right]$$

Then the updating formula for η is:

$$\begin{aligned} \eta_{(t)} = & c_i \eta_{(t-1)} \odot \text{sign}\left[\text{sign}(\Delta W_{(t)} \odot \Delta W_{(t-1)}) + \widetilde{1}\right] \\ & + c_d \eta_{(t-1)} \odot \left\{\widetilde{1} - \text{sign}\left[\text{sign}(\Delta W_{(t)} \odot \Delta W_{(t-1)}) + \widetilde{1}\right]\right\} . \quad \square \end{aligned}$$

Note that $\text{sign}(\Delta W_{(t)} \odot \Delta W_{(t-1)}) = \text{sign}(\Delta W_{(t)}) \odot \text{sign}(\Delta W_{(t-1)})$. Here there is a tradeoff between one floating point multiplication followed by a sign versus two sign operations followed by an integer multiplication. Which method is faster will depend on the actual system used.

Due to the fact that the next change is not exactly in the direction of the error gradient (because each component of ∇E is multiplied with a different constant) this technique may cause problems. This may be avoided by testing the output after an adaptation has taken place. If there is an increase in output error then the adaptation should be rejected and the next step calculated with the classical method. Afterwards the adaptation process may be resumed.

2.5.4 SuperSAB

SuperSAB (Super Self-Adapting Backpropagation) is a combination of momentum and adaptive backpropagation algorithms.

The algorithm uses adaptive backpropagation for the $w_{(\ell)hg}$ terms which continue to move in the same direction and momentum for the others:

- If $\Delta w_{(\ell)hg(t)} \, \Delta w_{(\ell)hg(t-1)} \geqslant 0$ then:

$$\eta_{(\ell)hg(t)} = c_i \eta_{(\ell)hg(t-1)}$$

$$\Delta w_{(\ell)hg(t+1)} = -\eta_{(\ell)hg(t)} \left. \frac{\partial E_p}{\partial w_{(\ell)hg}} \right|_{W_{(t)}}$$

  the momentum being 0 because it's not necessary: the learning rate grows in geometric progression due to the adaptive algorithm.

- If $\Delta w_{(\ell)hg(t)} \, \Delta w_{(\ell)hg(t-1)} < 0$ then:

$$\eta_{(\ell)hg(t)} = c_d \eta_{(\ell)hg(t-1)}$$

$$\Delta w_{(\ell)hg(t+1)} = -\eta_{(\ell)hg} \left. \frac{\partial E_p}{\partial w_{(\ell)hg}} \right|_{W_{(t)}} - \mu \Delta w_{(\ell)hg(t)}$$

  Note the "$-$" sign in front of μ which is being used to cancel the previous "wrong" weight adaption (not to boost $\Delta w_{(\ell)hg}$ as in momentum method); the corresponding $\eta_{(\ell)hg}$ is decreased to get smaller steps.

In matrix terms:

$$\eta_{(t)} = \mathcal{H}\{\Delta W_{(t)} \odot \Delta W_{(t-1)}, \eta_{(t-1)}, c_i \cdot, c_i \cdot, c_d \cdot\}$$

$$\Delta W_{(t+1)} = -\eta_{(t)} \odot \nabla E_p - \mathcal{H}\{\Delta W_{(t)} \odot \Delta W_{(t-1)}, \Delta W_{(t)}, = 0, = 0, \mu \cdot\}$$

Remarks:

➥ While this algorithm uses the same main algorithm as described in section 2.3 (of course with the required changes) note that memory requirement is four times

2.5.4 See [BTW95] p. 51.

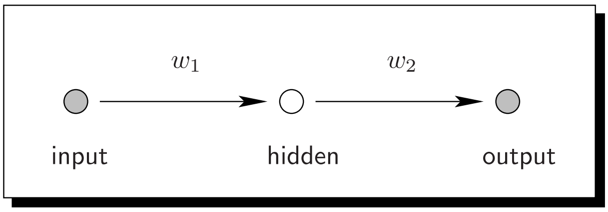

Figure 2.6: *The identity mapping network.*

higher than for standard backpropagation in order to store the supplementary η, $\Delta W_{(t)}$, and $\Delta W_{(t-1)}$.

➥ Using standard matrix operations, SuperSAB rules may be written as:

$$\eta_{(t)} = \left\{(c_i - c_d)\,\mathrm{sign}\left[\mathrm{sign}(\Delta W_{(t)} \odot \Delta W_{(t-1)}) + \widetilde{1}\right] + c_d \cdot \widetilde{1}\right\} \odot \eta_{(t-1)}$$

$$\begin{aligned}\Delta W_{(t+1)} = & -\eta_{(t)} \odot \nabla E_p \\ & -\mu \Delta W_{(t)} \odot \left\{\widetilde{1} - \mathrm{sign}\left[\mathrm{sign}(\Delta W_{(t)} \odot \Delta W_{(t-1)}) + \widetilde{1}\right]\right\}\end{aligned}$$

Proof. The first equation comes directly from (2.16).

For the second equation, the matrix $\widetilde{1} - \mathrm{sign}\left[\mathrm{sign}(\Delta W_{(t)} \odot \Delta W_{(t-1)}) + \widetilde{1}\right]$ contains as elements: 1 if $\Delta w_{(\ell)hg(t)} \Delta w_{(\ell)hg(t-1)} < 0$ and is zero otherwise, so momentum terms are added exactly to the $w_{(\ell)hg}$ requiring it, see the proof of (2.16). ☐

The matrix formulas introduced here lead to some supplementary operations, so they are not the most efficient when implemented directly. However they use just standard matrix operation which may be already available for the system targeted for implementation.

2.6 Examples

2.6.1 Identity Mapping Network

The network consists of one input and a hidden and an output neuron with 2 weights: w_1 and w_2. It uses the logistic activation and sum–of–squares error. See Figure 2.6.

This particular network, while of little practical use, has some interesting features:

- there are only 2 weights so it is possible to visualize exactly the error surface, see Figure 2.7 on the following page;
- the error surface has a segment of local minima, note that if the weights reach values corresponding to a local minimum of the error, standard backpropagation can't move forward as ∇E becomes zero there. That is the weights become "trapped".

The problem is to configure w_1 and w_2 such that the identity mapping is realized for binary input.

2.6.1 See [BTW95] pp. 48–49.

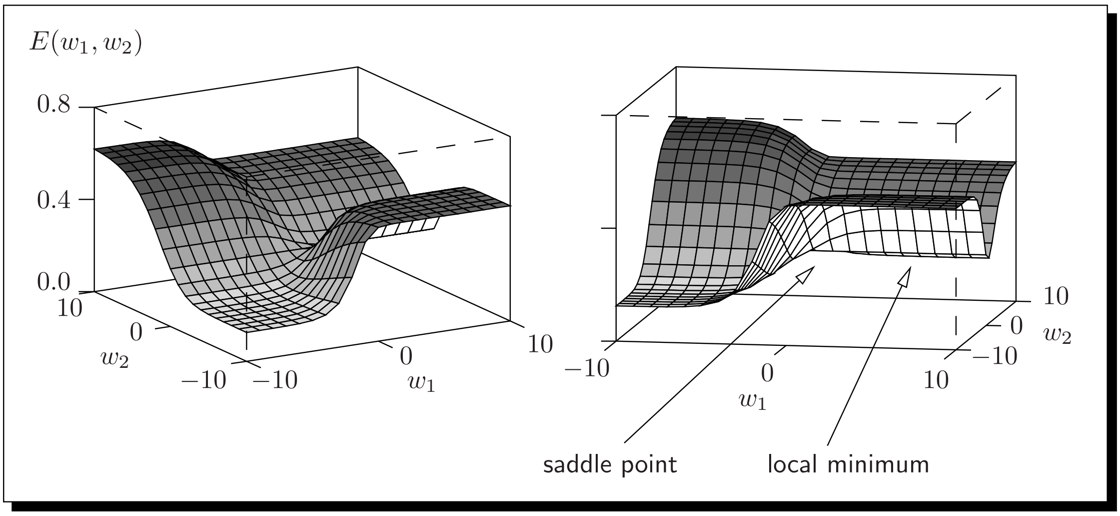

Figure 2.7: *The error surface for identity mapping network with logistic activation ($c = 1$) and sum–of–squares error.*

The input is $x \equiv z_{(0)}$ (by notation). The output of the hidden neuron $z_{(1)}$ is (see (2.3)):

$$z_{(1)} = \frac{1}{1 + \exp(-cw_1 z_{(0)})}$$

The network output is:

$$y(x) \equiv z_{(2)}(z_{(0)}) = \frac{1}{1 + \exp(-cw_2 z_{(1)})} = \frac{1}{1 + \exp\left(\frac{-cw_2}{1+\exp(-cw_1 z_{(0)})}\right)}$$

The identity mapping network tries to map its input to output, i.e. the training sets are ($P = 2$): $x_1 = 0$, $t_1 = 0$ and $x_2 = 1$, $t_2 = 1$. Then the sum–of–squares error (2.5) becomes:

$$\begin{aligned} E(w_1, w_2) &= \frac{1}{2} \sum_{p=1}^{2} \left[z_{(2)}(x_p) - t_p\right]^2 \\ &= \frac{1}{2} \left\{ \left[\frac{1}{1 + \exp\left(\frac{-cw_2}{2}\right)}\right]^2 + \left[\frac{1}{1 + \exp\left(\frac{-cw_2}{1+\exp(-cw_1)}\right)} - 1\right]^2 \right\} \end{aligned}$$

For $c = 1$ the error surface is shown in Figure 2.7. The surface has a local minimum and a saddle point which may “trap” the weights in the local minima region (see also the remarks at the end of Section 2.5.2).

2.6.2 The Encoder

This network is also an identity mapping ANN (targets are the same as inputs) with a single hidden layer which is smaller in size than the input/output. See figure 2.8.

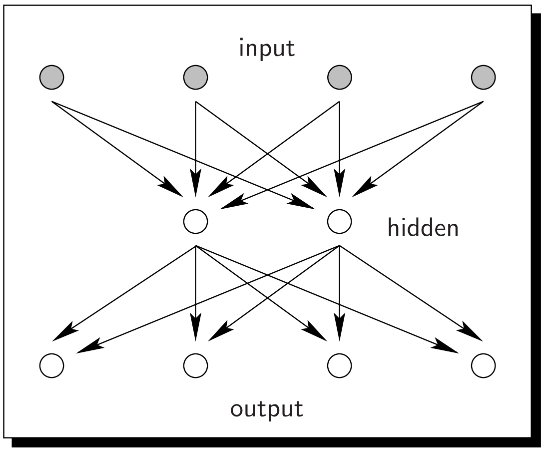

Figure 2.8: *The 4-2-4 encoder: 4 inputs, 2 hidden, 4 outputs.*

This model network shows the following[3]:

- the architecture of an backpropagation ANN may be important with respect to its purpose;
- the output of a hidden layer can sometimes have meaning,
- the importance of biases.

The input vectors and targets are ($P = 4$):

$$\mathbf{x}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} , \quad \mathbf{x}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} , \quad \mathbf{x}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} , \quad \mathbf{x}_4 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} , \quad \mathbf{t}_p \equiv \mathbf{x}_p , \quad p \in \{1, 2, 3, 4\}$$

The idea is that inputs have to be "squeezed" through the bottleneck represented by hidden layer, before being reproduced at output. The network seeks a way to encode the 4–component vectors onto a 2–component vector (output of the hidden layer). Obviously the ideal encoding will be a one–to–one correspondence between $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$ and:

$$\mathbf{z}_1 = \left(\begin{smallmatrix} 0 \\ 0 \end{smallmatrix} \right) , \quad \mathbf{z}_2 = \left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix} \right) , \quad \mathbf{z}_3 = \left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix} \right) , \quad \mathbf{z}_4 = \left(\begin{smallmatrix} 1 \\ 1 \end{smallmatrix} \right)$$

Note that one of $\mathbf{x}_p$ vectors will be encoded by $\mathbf{z}_1$. On a network without biases this means that the output layer will receive total input $\mathbf{a}_1 = \widehat{\mathbf{0}}$ and, if the logistic activation function is used (the same thing may happen for other activations as well), then the corresponding output will *always* be: $\mathbf{y}^{\mathrm{T}} = \begin{pmatrix} 0.5 & 0.5 & 0.5 & 0.5 \end{pmatrix}$ and this particular output will be weight–independent. One of the input vectors will never be learned by the encoder. In practice usually (but not always) the net will enter an oscillation trying to learn two vectors on one encoding, with the outcome that there will be two unlearnt vectors. When using biases this does not happen as the output layer will always receive something weight–dependent[4].

An ANN was trained[5] with logistic activation, sum–of–squares error and momentum with

[3] Also: larger networks of similar design could be used for compression.

[4] For a deeper analysis of bias significance see the second part of this book.

[5] The actual Scilab code is in the source tree of this book, in directory, e.g. `Matrix_ANN_1/Scilab/ANN_Toolbox-0.22/examples/ann/encoder_bm.sce`; note that actual version numbers may differ.

flat spot elimination, using the following parameters:

$$\eta = 2.5 \quad , \quad \mu = 0.9 \quad , \quad c_f = 0.25$$

An on–line method was used and weights were randomly initialized with values in range $[-1, 7]$. After 200 epochs (800 weight updates) the outputs of the hidden layer became:

$$\mathbf{z}_1 = \left(\begin{smallmatrix} 1 \\ 0 \end{smallmatrix}\right) \ , \quad \mathbf{z}_2 = \left(\begin{smallmatrix} 0 \\ 0 \end{smallmatrix}\right) \ , \quad \mathbf{z}_3 = \left(\begin{smallmatrix} 1 \\ 1 \end{smallmatrix}\right) \ , \quad \mathbf{z}_1 = \left(\begin{smallmatrix} 0 \\ 1 \end{smallmatrix}\right)$$

to double precision accuracy and the corresponding output:

$$\mathbf{y}_1 = \begin{pmatrix} 0.9975229 \\ 0.0047488 \\ 0.0015689 \\ 1.876 \cdot 10^{-8} \end{pmatrix} , \ \mathbf{y}_2 = \begin{pmatrix} 0.0000140 \\ 0.9929489 \\ 8.735 \cdot 10^{-9} \\ 0.0045821 \end{pmatrix} , \ \mathbf{y}_3 = \begin{pmatrix} 0.0020816 \\ 7.241 \cdot 10^{-11} \\ 0.997392 \\ 0.0010320 \end{pmatrix} , \ \mathbf{y}_4 = \begin{pmatrix} 7.227 \cdot 10^{-11} \\ 0.0000021 \\ 0.0021213 \\ 0.9960705 \end{pmatrix}$$

which are $\{\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4\}$ to within 1% accuracy.

Remarks:

- Encoders with $N_1 = \log_2 N_0$ are called *tight* encoders, those with $N_1 < \log_2 N_0$ are *supertight* those with $N_1 > \log_2 N_0$ are *loose*.
- It is possible to train a loose encoder for inputs of type $\mathbf{e}_i$ on an ANN without biases as the null vector doesn't have to be among the outputs of hidden neurons.

CHAPTER 3

The SOM / Kohonen Network

The Kohonen network is an example of an ANN with unsupervised learning.

3.1 Network Architecture

A SOM (Self Organizing Map, known also as Kohonen) network has *one* single layer of neurons: the output. The additional "sensory" layer is just distributing the inputs to all neurons, there is no data processing on it. The output layer has a lateral feedback interaction between its neurons (see also Section 3.3). The number of inputs into each neuron is N, the number of neurons is K. See Figure 3.1 on the following page. SOM

Remarks:

- Here the output layer has been considered to be unidimensional. Taking into account the "mapping" feature of the Kohonen networks the output layer may be sometimes considered as being multidimensional, and this may be more convenient for some particular applications.
- A multidimensional output layer may be trivially mapped to a unidimensional one and the discussion below will remain the same.

  E.g. a $K \times K$ bidimensional layer may be mapped to a unidimensional layer with K^2 neurons just by finding a function $f : K \times K \to K$ to do the relabeling/numbering of neurons. Such a function may be $f(k, \ell) = (k-1)K + \ell$ which maps first row of neurons $(1,1) \ldots (1,K)$ to first K unidimensional chunk and so on ($1 \leqslant k, \ell \leqslant K$). This type of mapping is called the *lexicographic convention*, or sometimes the *natural ordering*. lexicographic convention

[3.1] See [BTW95] pp. 83–89 and [Koh88] pp. 119–124.

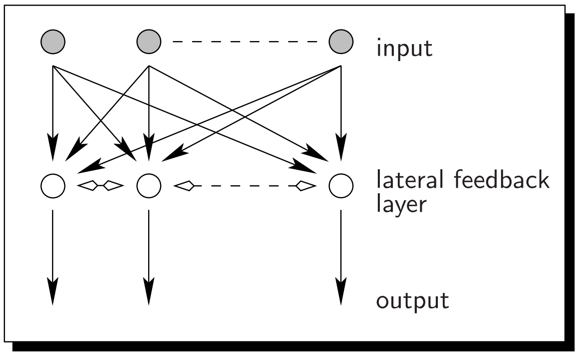

Figure 3.1: *The SOM/Kohonen network structure.*

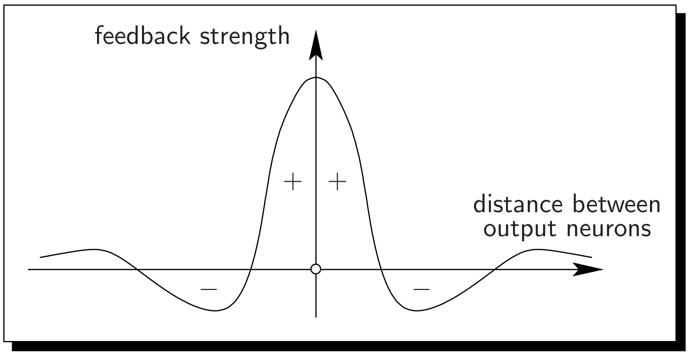

Figure 3.2: *A lateral feedback interaction function of the "mexican hat" type.*

The important thing to remember is that all output neurons receive all components of the input vector and a little bit of care is to be taken when establishing the neuronal neighborhood (see below).

In general, the lateral feedback interaction function is *indirect*, i.e. neurons do *not* receive the inputs of their neighbors, and of "mexican hat" type. See figure 3.2. The nearest neurons receive a positive feedback, the more distant ones receive negative feedback and the faraway ones are unaffected. Also the positive feedback is stronger than the negative one.

✍ **Remarks:**

interneuronal distances

- The distance between neuron neighbors in output layer is (obviously) a discrete one. On unidimensional networks, it may be defined as being 0 for the neuron itself (auto-feedback), 1 for the closest neighbors, 2 for the next ones, and so on. On multidimensional output layers there are several choices, the most commonly used one being the Euclidean distance.
- The feedback function determines the quantity by which the weights of neuron neighbors are updated during the learning process (as well as *which* weights are updated).

neuronal neighborhood

- The area affected by the lateral feedback is called the *neuronal neighborhood*.

3.2 Neuronal Learning

3.2.1 Unsupervised Learning with Indirect Lateral Feedback

Let $W = \{w_{ki}\}_{\substack{k \in \{1,\dots,K\} \\ i \in \{1,\dots,N\}}}$ be the weight matrix and $\mathbf{x}$ the input vector, so that the weighted input to the (output) layer is $\mathbf{a} = W\mathbf{x}$. Note that each row from W represents the weights associated with one neuron and may be seen as a vector $W_{(k,:)}$ of the *same size* and from the *same space* $\mathbb{R}^N$ as the input vector $\mathbf{x}$. $\mathbb{R}^N$ is called the *weights space*.

weights space

When an input vector $\mathbf{x}$ is presented to the network, the neuron having its associated weight vector $W_{(k,:)}$ closest to $\mathbf{x}$, i.e. one for which:

$$\|W^{\mathrm{T}}_{(k,:)} - \mathbf{x}\| = \min_{1 \leqslant \ell K} \|W^{\mathrm{T}}_{(\ell,:)} - \mathbf{x}\|$$

is declared "winner". All and only the neurons from winner's vicinity, including itself, will participate in the "learning" of $\mathbf{x}$. The rest remain unaffected.

The learning process consists of changing the weight vectors $W_{(k,:)}$ towards the input vector (positive feedback) There is also a "forgetting" process which tries to slow down the progress (negative feedback).

learning

The feedback is indirect because the neurons are affected by being in the winner's neuronal neighborhood, not by receiving directly its output.

indirect feedback

Considering a linear learning process — changes are restricted to occur only in the direction of a linear combination of $\mathbf{x}$ and $W_{(k,:)}$ for each neuron — then:

$$\frac{dW_{(k,:)}}{dt} = \varphi(\cdot)\, \mathbf{x}^{\mathrm{T}} - \gamma(\cdot)\, W_{(k,:)}$$

where φ and γ are scalar (possibly nonlinear) functions, φ representing the positive feedback, γ being the negative one. These two functions will have to be built in such a way as to affect only the neuronal neighborhood of winning neuron k, which varies in time as the input vector is a function of time $\mathbf{x} = \mathbf{x}_{(t)}$.

❖ φ, γ

There are many ways to build an adaptation model (differential equation for $W_{(k,:)}$). Some of the more simple ones which may be analyzed (at least to some extent) analytically are discussed in the following sections. To simplify further the discussion it will be considered (at this stage) that all neurons are equally adapted. Later we will discuss how to use a lateral feedback function and limit the weights adjusting to targeted neurons.

3.2.2 Trivial Equation

One of the most simple adaptation equations is a linear differential equation:

❖ α, β

$$\frac{dW_{(k,:)}}{dt} = \alpha \mathbf{x}^{\mathrm{T}} - \beta W_{(k,:)} \ , \quad \alpha, \beta = \text{const.} \ , \quad \beta > 0 \ , \quad 1 \leqslant k \leqslant K$$

[3.2] See [Koh88] pp. 92–98.

which in matrix form becomes:

$$\frac{dW}{dt} = \alpha \widehat{\mathbf{1}} \mathbf{x}^{\mathrm{T}} - \beta W \quad \Leftrightarrow \quad \frac{dW}{dt} = \alpha \mathbf{x}^{\mathrm{T}} \overset{\mathrm{R}}{\ominus} \beta W \tag{3.1}$$

and with initial condition $W_{(0)} \equiv W_0$ the solution is:

$$W_{(t)} = \left(\alpha \widehat{\mathbf{1}} \int\limits_0^t \mathbf{x}_{(s)}^{\mathrm{T}} e^{\beta s} ds + W_0 \right) e^{-\beta t} = \left(\alpha \int\limits_0^t \mathbf{x}_{(s)}^{\mathrm{T}} e^{\beta s} ds \overset{\mathrm{R}}{\oplus} W_0 \right) e^{-\beta t}$$

showing that, for $t \to \infty$, $W_{(k,:)}$ is the *exponentially* weighted average of $\mathbf{x}_{(t)}$ and does not produce any interesting effects.

Proof. The equation is solved by the method of variation of parameters. First, the homogeneous equation: $\frac{dW}{dt} + \beta W = \widetilde{0}$ has a solution of the form: $W_{(t)} = C e^{-\beta t}$, C being a matrix of constants.

The general solution for the non–homogeneous equation (3.1) is found by considering $C = C_{(t)}$:

$$\frac{dW}{dt} = \frac{dC}{dt} e^{-\beta t} - \beta C_{(t)} e^{-\beta t}$$

and, by substituting $W_{(t)} = C_{(t)} e^{-\beta t}$ back into (3.1):

$$\frac{dC}{dt} e^{-\beta t} - \beta C_{(t)} e^{-\beta t} = \alpha \widehat{\mathbf{1}} \mathbf{x}_{(t)}^{\mathrm{T}} - \beta C_{(t)} e^{-\beta t} \quad \Rightarrow$$

$$\Rightarrow \quad \frac{dC}{dt} = \alpha \widehat{\mathbf{1}} \mathbf{x}_{(t)}^{\mathrm{T}} e^{\beta t} \quad \Rightarrow \quad C(t) = \alpha \widehat{\mathbf{1}} \int\limits_0^t \mathbf{x}_{(s)}^{\mathrm{T}} e^{\beta s} ds + C' \quad (C' = \text{matrix of constants})$$

and, at $t = 0$, $C_{(0)} = C' \Rightarrow W_{(0)} = C' \Rightarrow C' = W_0$. □

3.2.3 Simple Equation

❖ α, β

The simple equation is defined as:

$$\frac{dW_{(k,:)}}{dt} = \alpha a_k \mathbf{x}^{\mathrm{T}} - \beta W_{(k,:)} \ , \quad \alpha, \beta = \text{const.} \ , \quad k = \overline{1, K} \tag{3.2}$$

and in matrix notation it becomes: $\frac{dW}{dt} = \alpha \mathbf{a} \mathbf{x}^{\mathrm{T}} - \beta W$. As $\mathbf{a} = W\mathbf{x}$ then:

$$\frac{dW}{dt} = W(\alpha \mathbf{x}\mathbf{x}^{\mathrm{T}} - \beta I) \quad \Leftrightarrow \quad \frac{dW}{dt} = W(\alpha \mathbf{x}\mathbf{x}^{\mathrm{T}} \overset{\mathrm{I}}{\ominus} \beta)$$

(in simulations, calculate $\sqrt{\alpha}\mathbf{x}$ first, then $\alpha \mathbf{x}\mathbf{x}^{\mathrm{T}}$, this reduces the number of multiplications, note that $\mathbf{x}\mathbf{x}^{\mathrm{T}}$ is symmetric).

In discrete–time approximation:

$$\frac{dW}{dt} \to \frac{\Delta W}{\Delta t} = \frac{W_{(t+1)} - W_{(t)}}{(t+1) - t} = W_{(t)}(\alpha \mathbf{x}_{(t)} \mathbf{x}_{(t)}^{\mathrm{T}} - \beta I) \quad \Rightarrow$$

$$W_{(t+1)} = W_{(t)}(\alpha \mathbf{x}_{(t)} \mathbf{x}_{(t)}^{\mathrm{T}} - \beta I + I)$$

with $W_{(0)} \equiv W_0$ as initial condition. The general solution is:

$$W_{(t)} = W_0 \prod_{s=0}^{t-1} \left[\alpha \mathbf{x}_{(s)} \mathbf{x}_{(s)}^{\mathrm{T}} - (\beta - 1) I \right] \tag{3.3}$$

Remarks:

➥ For most cases the solution (3.3) is either divergent or converges to zero, both cases unacceptable. However, for a relatively short time, the simple equation may approximate a more complicated, asymptotically stable, process.

➥ For t or α relatively small, such that the higher order terms $\mathcal{O}(\alpha^2)$ may be neglected, and considering $\beta = 0$, i.e. no "forgetting" effect) then from (3.3):

$$W_{(t)} \simeq W_0 \left(I + \alpha \sum_{s=0}^{t-1} \mathbf{x}_{(s)} \mathbf{x}_{(s)}^{\mathrm{T}} \right)$$

Orthogonal projection operator

A particular case of interest of (3.2) is when weights are moved away from $\mathbf{x}$ and there is no "forgetting" effect:

$$\frac{dW_{(k,:)}}{dt} = -\alpha a_k \mathbf{x} \;, \quad \alpha > 0 \;, \quad \alpha = \text{const.} \;, \quad k = \overline{1, K}$$

and by replacing $a_k = W_{(k,:)}\mathbf{x}$: $\frac{dW_{(k,:)}}{dt} = -\alpha W_{(k,:)} \mathbf{x}\mathbf{x}^{\mathrm{T}}$. In matrix notation:

$$\frac{dW}{dt} = -\alpha W \mathbf{x}\mathbf{x}^{\mathrm{T}} \quad \Leftrightarrow \quad \frac{dW}{dt} = -W(\alpha\mathbf{x})\mathbf{x}^{\mathrm{T}}$$

(in simulations, calculate $\sqrt{\alpha}\mathbf{x}$ first, then $\alpha\mathbf{x}\mathbf{x}^{\mathrm{T}}$, this reduces the number of multiplications, note that $\mathbf{x}\mathbf{x}^{\mathrm{T}}$ is symmetric).

For $\mathbf{x} = \text{const.}$ and $W_{(0)} \equiv W_0$, this equation has the solution:

$$W_{(t)} = W_0 \left(I - \frac{1 - e^{-\alpha\|\mathbf{x}\|^2 t}}{\|\mathbf{x}\|^2} \mathbf{x}\mathbf{x}^{\mathrm{T}} \right)$$

Proof. By checking directly the given solution:

$$\begin{aligned}
\frac{dW}{dt} &= \frac{W_0}{\|\mathbf{x}\|^2} (-\alpha\|\mathbf{x}\|^2) e^{-\alpha\|\mathbf{x}\|^2 t} \mathbf{x}\mathbf{x}^{\mathrm{T}} = -\alpha W_0 e^{-\alpha\|\mathbf{x}\|^2 t} \frac{\mathbf{x}\|\mathbf{x}\|^2\mathbf{x}^{\mathrm{T}}}{\|\mathbf{x}\|^2} = -\alpha W_0 \left(e^{-\alpha\|\mathbf{x}\|^2 t} \frac{\mathbf{x}\mathbf{x}^{\mathrm{T}}}{\|\mathbf{x}\|^2} \right) \mathbf{x}\mathbf{x}^{\mathrm{T}} \\
&= -\alpha W_0 \left(\frac{\mathbf{x}\mathbf{x}^{\mathrm{T}}}{\|\mathbf{x}\|^2} - \frac{\mathbf{x}\mathbf{x}^{\mathrm{T}}}{\|\mathbf{x}\|^2} + e^{-\alpha\|\mathbf{x}\|^2 t} \frac{\mathbf{x}\mathbf{x}^{\mathrm{T}}}{\|\mathbf{x}\|^2} \right) \mathbf{x}\mathbf{x}^{\mathrm{T}} = -\alpha W_0 \mathbf{x}\mathbf{x}^{\mathrm{T}} + \alpha W_0 \frac{1 - e^{-\alpha\|\mathbf{x}\|^2 t}}{\|\mathbf{x}\|^2} \mathbf{x}\mathbf{x}^{\mathrm{T}}\mathbf{x}\mathbf{x}^{\mathrm{T}} \\
&= -\alpha W_{(t)} \mathbf{x}\mathbf{x}^{\mathrm{T}} \quad \square
\end{aligned}$$

The weights converge to $W_{(\infty)} \equiv \lim_{t\to\infty} W_{(t)} = W_0 \left(I - \frac{\mathbf{x}\mathbf{x}^{\mathrm{T}}}{\|\mathbf{x}\|^2} \right)$. This means that, while the total input (to any neuron in the lateral feedback layer) converges to zero: $\lim_{t\to\infty} \mathbf{a} = W_{(\infty)}\mathbf{x} = W_0 \left(I - \frac{\mathbf{x}\mathbf{x}^{\mathrm{T}}}{\|\mathbf{x}\|^2} \right) \mathbf{x} = \widehat{\mathbf{0}}$, the weights do not have to converge to zero (except for the case discussed below).

Let us consider the situation where learning has stabilized (by maintaining $\mathbf{x}$ for a sufficiently long time) and weights have been "frozen". A new vector $\widetilde{\mathbf{x}}$ is presented at input; $\widetilde{\mathbf{x}}$ may be written as a combination of one component parallel to $\mathbf{x}$ and another one perpendicular: $\widetilde{\mathbf{x}} = \widetilde{\mathbf{x}}_{\|} + \widetilde{\mathbf{x}}_{\perp}$. As $\widetilde{\mathbf{x}}_{\|} \propto \mathbf{x}$ then $\widetilde{\mathbf{a}} = W\widetilde{\mathbf{x}} = W\widetilde{\mathbf{x}}_{\perp}$. Summarizing, the network will react only to the component perpendicular to all previously learned vectors: if the $\{\mathbf{x}_p\}_{p=\overline{1,P}}$ system of *linearly independent* vectors have been learnt then a new vector may be written as $\widetilde{\mathbf{x}} = \sum\limits_{p=1}^{P} \widetilde{\mathbf{x}}_{\|\mathbf{x}_p} + \widetilde{\mathbf{x}}_{\perp}$, where $\widetilde{\mathbf{x}}_{\perp} \perp \mathbf{x}_p$, $\forall p \in \{1, \ldots, P\}$, and $\widetilde{a} = W\widetilde{\mathbf{x}}_{\perp}$. The network works similarly to a orthogonal projection operator: it reacts only to inputs from the subspace orthogonal to the one spanned by $\{\mathbf{x}_p\}$, i.e. it has become *habituated* to the training set and will react only to the novel components of input. SOM's built on this learning model may be used as novelty detectors/filters.

habituation

novelty detector

How many *linearly independent* training vectors may be learnt ? First as $\mathbf{x}_p \in \mathbf{R}^N$ then $P \leqslant N$ (in order to preserve linear independence, learning patterns which are combination of others do not count, see above). For k–th neuron, after learning has stopped, $W_{(k,:)}\mathbf{x}_p = 0$ and there are P such linear equations in $\{w_{ki}\}$ unknowns. For $P = N$ there is one solution: $W_{(k,:)} = \widehat{\mathbf{0}}^{\mathrm{T}}$ and network output will be trivially zero for any input vector. So P should be chosen less than N, depending upon the dimension of subspace $\mathbb{R}^{N-P}$ the network should react to (or dimension P the network should be insensitive to).

3.2.4 Riccati Equation

❖ α, β

The Riccati equation is defined as:

$$\frac{dW_{(k,:)}}{dt} = \alpha\mathbf{x}^{\mathrm{T}} - \beta a_k W_{(k,:)} \ , \quad \alpha, \beta = \text{const.} \ , \quad \alpha, \beta > 0 \ , \quad 1 \leqslant k \leqslant K \tag{3.4}$$

and, after the replacement $a_k = W_{(k,:)}\mathbf{x} = \mathbf{x}^{\mathrm{T}} W_{(k,:)}^{\mathrm{T}}$, it becomes:

$$\frac{dW_{(k,:)}}{dt} = \mathbf{x}^{\mathrm{T}} \left(\alpha I - \beta W_{(k,:)}^{\mathrm{T}} \, W_{(k,:)} \right) \tag{3.5}$$

or in matrix notation:

$$\frac{dW}{dt} = \alpha \widehat{\mathbf{1}} \mathbf{x}^{\mathrm{T}} - \beta (W \mathbf{x} \widehat{\mathbf{1}}^{\mathrm{T}}) \odot W \quad \Leftrightarrow \quad \frac{dW}{dt} = \alpha \mathbf{x}^{\mathrm{T}} \overset{\mathrm{R}}{\ominus} (\beta W \mathbf{x}) \overset{\mathrm{C}}{\odot} W \tag{3.6}$$

Proof. The set of equations (3.4) may be written in one matrix formula as:

$$\frac{dW}{dt} = \alpha \widehat{\mathbf{1}} \mathbf{x}^{\mathrm{T}} - \beta (\mathbf{a} \widehat{\mathbf{1}}^{\mathrm{T}}) \odot W = \alpha \mathbf{x}^{\mathrm{T}} \overset{\mathrm{R}}{\ominus} \beta \mathbf{a} \overset{\mathrm{C}}{\odot} W \quad , \quad \mathbf{a} = W\mathbf{x} \quad \square$$

In general, the Riccati equation has no closed form solution.

Proposition 3.2.1. *For* $\mathbf{x} =$ *const.* $\neq \widehat{\mathbf{0}}$*, if the Riccati equation* (3.5) *has a solution, i.e.* $\lim\limits_{t \to \infty} W$ *exists, then it is of the form:*

$$\lim_{t \to \infty} W = \sqrt{\frac{\alpha}{\beta}} \ \widehat{\mathbf{1}} \ \frac{\mathbf{x}^{\mathrm{T}}}{\|\mathbf{x}\|}$$

Proof. Assume steady state ($dW/dt = \widetilde{0}$). From (3.5):

$$\mathbf{x}^{\mathrm{T}} \left(\alpha I - \beta W_{(k,:)}^{\mathrm{T}} \, W_{(k,:)} \right) = \widehat{\mathbf{0}}^{\mathrm{T}} \quad \Rightarrow \quad \alpha \mathbf{x}^{\mathrm{T}} = \beta \left(\mathbf{x}^{\mathrm{T}} W_{(k,:)}^{\mathrm{T}} \right) W_{(k,:)}$$

which implies $W_{(k,:)}^{\mathrm{T}}$ is parallel to $\mathbf{x}$.

Suppose $W_{(k,:)}^{\mathrm{T}} = \theta\mathbf{x}$. Then:

$$\alpha\mathbf{x}^{\mathrm{T}} = \beta\left(\mathbf{x}^{\mathrm{T}}W_{(k,:)}^{\mathrm{T}}\right)W_{(k,:)} = \beta(\mathbf{x}^{\mathrm{T}}\theta\mathbf{x})\theta\mathbf{x}^{\mathrm{T}} \quad \Rightarrow \quad \alpha\mathbf{x}^{\mathrm{T}} = \beta\theta^2(\mathbf{x}^{\mathrm{T}}\mathbf{x})\mathbf{x}^{\mathrm{T}}$$

$$\alpha = \beta\theta^2\|\mathbf{x}\|^2 \quad \Rightarrow \quad \theta^2 = \frac{\alpha}{\beta\|\mathbf{x}\|^2}$$

As $\alpha, \beta > 0$ then $\lim\limits_{t\to\infty} W_{(k,:)} = \sqrt{\frac{\alpha}{\beta}}\frac{\mathbf{x}^{\mathrm{T}}}{\|\mathbf{x}\|}$. □

3.2.5 More General Equations

Theorem 3.2.1. *Let $\alpha > 0$, $\mathbf{x} =$ const. $\neq \widehat{\mathbf{0}}$, $\mathbf{a} = W\mathbf{x}$ and γ a continous real valued function*[1]. ❖ α, γ

Then, if a learning model (process) of type:

$$\frac{dW_{(k,:)}}{dt} = \alpha\mathbf{x}^{\mathrm{T}} - \gamma(a_k)\,W_{(k,:)} \quad , \quad 1 \leqslant k \leqslant K \tag{3.7}$$

or, in matrix notation:

$$\frac{dW}{dt} = \alpha\widehat{\mathbf{1}}\mathbf{x}^{\mathrm{T}} - \left[\gamma(W\mathbf{x})\,\widehat{\mathbf{1}}^{\mathrm{T}}\right] \odot W \quad \Leftrightarrow \quad \frac{dW}{dt} = \alpha\mathbf{x}^{\mathrm{T}} \overset{\mathrm{R}}{\ominus} \gamma(W\mathbf{x}) \overset{\mathrm{C}}{\odot} W$$

has a finite steady state solution W as $t \to \infty$, then it must be of the form:

$$\lim_{t\to\infty} W \propto \widehat{\mathbf{1}}\,\mathbf{x}^{\mathrm{T}}$$

i.e. all $W_{(k,:)}$ become parallel to $\mathbf{x}^{\mathrm{T}}$ in $\mathbb{R}^N$.

Proof. Assume steady state ($dW/dt = \widetilde{0}$). From the continuity of γ, $\lim\limits_{t\to\infty} \gamma(a_k)$ exists. From (3.7):

$$\alpha\mathbf{x}^{\mathrm{T}} - \gamma(a_k)\,W_{(k,:)} = \widehat{\mathbf{0}}^{\mathrm{T}} \quad \Rightarrow \quad \alpha\mathbf{x}^{\mathrm{T}} = \gamma(a_k)\,W_{(k,:)}$$

which implies $W_{(k,:)}$ is parallel to $\mathbf{x}^{\mathrm{T}}$. □

Theorem 3.2.2. *Let $\alpha > 0$, $\mathbf{x} =$ const. $\neq \widehat{\mathbf{0}}$, $\mathbf{a} = W\mathbf{x}$ and γ be a continous real valued function with $\gamma(0) \neq 0$.* ❖ α, γ

Then, if a learning model (process) of type:

$$\frac{dW_{(k,:)}}{dt} = \alpha a_k\mathbf{x}^{\mathrm{T}} - \gamma(a_k)W_{(k,:)} \quad , \quad 1 \leqslant k \leqslant K \tag{3.8}$$

or, in matrix notation:

$$\frac{dW}{dt} = \alpha W\mathbf{x}\mathbf{x}^{\mathrm{T}} - \left[\gamma(W\mathbf{x})\,\widehat{\mathbf{1}}^{\mathrm{T}}\right] \odot W \quad \Leftrightarrow \quad \frac{dW}{dt} = \alpha W\mathbf{x}\mathbf{x}^{\mathrm{T}} - \gamma(W\mathbf{x}) \overset{\mathrm{C}}{\odot} W$$

has a nonzero finite steady state solution W for $t \to \infty$, it has to be of the form:

$$W \propto \widehat{\mathbf{1}}\mathbf{x}^{\mathrm{T}}$$

i.e. all $W_{(k,:)}$ become parallel to $\mathbf{x}^{\mathrm{T}}$ in $\mathbb{R}^N$.

[3.2.5] See [Koh88] pp. 98–101.

[1] Recall the convention $\gamma(\mathbf{a}) = \left(\gamma(a_1)\,, \quad \cdots\,, \quad \gamma(a_K)\right)^{\mathrm{T}}$

Proof. Assume steady state ($dW/dt = \widetilde{0}$). From the continuity of γ, $\lim_{t\to\infty} \gamma(a_k)$ exists. From (3.8):

$$\alpha a_k \mathbf{x}^{\mathrm{T}} - \gamma(a_k) W_{(k,:)} = \widehat{\mathbf{0}}^{\mathrm{T}} \quad \Rightarrow \quad \alpha a_k \mathbf{x}^{\mathrm{T}} = \gamma(a_k) W_{(k,:)}$$

which implies $W_{(k,:)}^{\mathrm{T}}$ is parallel to $\mathbf{x}$, since if a_k is zero, $\gamma(a_k)$ is not. ☐

❖ $\{\mathbf{x}\}^{\perp}$, s

Suppose that at some time $t = s$, $W_{(k,:)}$ becomes perpendicular to $\mathbf{x}$, i.e. $W_{(k,:)} \in \{\mathbf{x}\}^{\perp}$. Then $a_k = W_{(k,:)}\mathbf{x} = 0$ and from (3.8):

$$\left.\frac{dW_{(k,:)}}{dt}\right|_s = -\gamma(0) W_{(k,:)(s)} \tag{3.9}$$

where $W_{(k,:)(s)}$ is $W_{(k,:)}$ at time s. Then all future changes in $W_{(k,:)}$ will be contained in[2] $\{\mathbf{x}\}^{\perp}$ ($W_{(k,:)}$ becomes "trapped"). The solution to (3.9), for $t \geqslant s$, is:

$$W_{(k,:)(t)} = W_{(k,:)(s)} e^{-\gamma(0)t} \quad , \quad t \geqslant s$$

Recall from the hypotheses $\gamma(0) \neq 0$. At limit, for $\gamma(0) > 0$, $\lim_{t\to\infty} W_{(k,:)} = \widehat{\mathbf{0}}^{\mathrm{T}}$, while for $\gamma(0) < 0$, $W_{(k,:)}$ diverges. So a necessary condition for the existence of a finite steady state solution $\lim_{t\to\infty} W$ is either $\gamma(0) > 0$ or $W\mathbf{x} \neq \widehat{\mathbf{0}}$ at all times.

The solution $\lim_{t\to\infty} W_{(k,:)} = \widehat{\mathbf{0}}^{\mathrm{T}}$ involves an incomplete learning for neuron k so the condition $W\mathbf{x} \neq \widehat{\mathbf{0}}$ at all times appears necessary. However, in practice it may be neglected and this deserves a discussion.

The initial value for weights, $W_{(0)}$, should be chosen such that $W_{(0)}\mathbf{x} \neq \widehat{\mathbf{0}}$. But $W_{(k,:)}\mathbf{x} \neq 0$ means that $W_{(k,:)} \not\perp \mathbf{x}$, i.e. $W_{(k,:)}$ is *not* contained in $\{\mathbf{x}\}^{\perp}$ ($\{\mathbf{x}\}^{\perp}$ being a hyperplane through 0 perpendicular to $\mathbf{x}$). Even a random selection on $W_{(0)}$ would have a good chance to satisfy this condition, even more so as $\mathbf{x}$ are seldom known exactly in practice, being a stochastic input variable.

The conclusion is then that the condition $W\mathbf{x} \neq \widehat{\mathbf{0}}$, $\forall t$, may be neglected in practice, with the remark that there is a small probability that some $W_{(k,:)}$ weight vectors may become trapped in $\{\mathbf{x}\}^{\perp}$ and then learning will be incomplete.

3.2.6 Bernoulli Equation

❖ α, β

The Bernoulli equation is a particular case of (3.8) with $\gamma(a_k) = \beta a_k$:

$$\frac{dW_{(k,:)}}{dt} = \alpha a_k \mathbf{x}^{\mathrm{T}} - \beta a_k W_{(k,:)} \ , \quad \alpha, \beta > 0 \ , \quad \alpha, \beta = \text{const.} \ , \quad k \in \{1, \dots, K\}$$

and, after the replacement $a_k = W_{(k,:)}\mathbf{x}$, it becomes:

$$\frac{dW_{(k,:)}}{dt} = W_{(k,:)}(\alpha \mathbf{x}\mathbf{x}^{\mathrm{T}} - \beta \mathbf{x} W_{(k,:)}) \tag{3.10}$$

[2] $\mathbf{x}\mathbf{x}^{\mathrm{T}}$ is the covariance matrix of the input vector and has rank 1. Thus it will have only one non–zero eigenvalue and the corresponding eigenspace will consists of multiples of $\mathbf{x}$ with eigenvalue $\|\mathbf{x}\|^2$ (seen another way $\mathbf{x}\mathbf{x}^{\mathrm{T}}\mathbf{y} = \mathbf{x}(\mathbf{x}^{\mathrm{T}}\mathbf{y})$ is always parallel to $\mathbf{x}$ so $\mathrm{span}\{\mathbf{x}\}$ is the eigenspace corresponding to the non–zero eigenvalue). $\{\mathbf{x}\}^{\perp}$ is the $N-1$ dimensional space corresponding to the eigenvalue 0.

or in matrix notation:

$$\frac{dW}{dt} = \alpha W \mathbf{x}\mathbf{x}^T - \beta(W\mathbf{x}\widehat{\mathbf{1}}^{\mathrm{T}}) \odot W \quad \Leftrightarrow \quad \frac{dW}{dt} = \alpha W \mathbf{x}\mathbf{x}^T - \beta(W\mathbf{x}) \overset{\mathrm{C}}{\odot} W$$

(in simulations, calculate $\sqrt{\alpha}\mathbf{x}$ first, then $\alpha\mathbf{x}\mathbf{x}^{\mathrm{T}}$, this reduces the number of multiplications, note that $\mathbf{x}\mathbf{x}^{\mathrm{T}}$ is symmetric).

From Theorem 3.2.2, for $\mathbf{x} =$ const., the solution is in general $W \propto \widehat{\mathbf{1}}\mathbf{x}^{\mathrm{T}}$.

3.2.7 PCA Equation

The PCA equation is another particular case of (3.8) with $\gamma(a_k) = \beta a_k^2$: ❖ α, β

$$\frac{dW_{(k,:)}}{dt} = \alpha a_k \mathbf{x}^{\mathrm{T}} - \beta a_k^2 W_{(k,:)} \ , \quad \alpha, \beta > 0 \ , \quad \alpha, \beta = \text{const.} \ , \quad k = \overline{1, K}$$

and after the substitution $a_k = W_{(k,:)}\mathbf{x}$, it becomes:

$$\frac{dW_{(k,:)}}{dt} = W_{(k,:)}\mathbf{x}\mathbf{x}^{\mathrm{T}}(\alpha I - \beta W_{(k,:)}^{\mathrm{T}} W_{(k,:)}) \tag{3.11}$$

or in matrix notation:

$$\frac{dW}{dt} = \alpha W \mathbf{x}\mathbf{x}^{\mathrm{T}} - \beta[(W\mathbf{x})^{\odot 2}\widehat{\mathbf{1}}^{\mathrm{T}}] \odot W \quad \Leftrightarrow \quad \frac{dW}{dt} = \alpha W \mathbf{x}\mathbf{x}^{\mathrm{T}} - \beta(W\mathbf{x})^{\odot 2} \overset{\mathrm{C}}{\odot} W$$

(in simulations: see previous remarks with respect to computation of $\alpha\mathbf{x}\mathbf{x}^{\mathrm{T}}$).

From Theorem 3.2.2, for $\mathbf{x} =$ const., the solution is in general $W \propto \widehat{\mathbf{1}}\mathbf{x}^{\mathrm{T}}$. However, the norm of $W_{(k,:)}$ will converge to $\sqrt{\alpha/\beta}$, if $W_{(k,:)}\mathbf{x}\mathbf{x}^{\mathrm{T}}W_{(k,:)}^{\mathrm{T}} \neq 0$, at all times; the result being:

$$W = \sqrt{\frac{\alpha}{\beta}}\ \widehat{\mathbf{1}}\mathbf{x}^{\mathrm{T}}$$

Proof. $\frac{d\|W_{(k,:)}\|^2}{dt} = \frac{dW_{(k,:)}}{dt} W_{(k,:)}^{\mathrm{T}}$; from (3.11):

$$\frac{d\|W_{(k,:)}\|^2}{dt} = 2\ \frac{dW_{(k,:)}}{dt} W_{(k,:)}^{\mathrm{T}} = 2\ W_{(k,:)}\mathbf{x}\mathbf{x}^{\mathrm{T}}W_{(k,:)}^{\mathrm{T}}(\alpha - \beta\|W_{(k,:)}\|^2)$$

Existence of a solution implies $\lim\limits_{t\to\infty} \frac{d\|W_{(k,:)}\|^2}{dt} = 0$. As $W_{(k,:)}\mathbf{x}\mathbf{x}^{\mathrm{T}}W_{(k,:)}^{\mathrm{T}} \neq 0$ then $\lim\limits_{t\to\infty}(\alpha - \beta\|W_{(k,:)}\|^2) = 0 \Rightarrow \lim\limits_{t\to\infty}\|W_{(k,:)}\|^2 = \alpha/\beta$. □

3.3 Network Dynamics

3.3.1 Running Procedure

The process of running the network consists just in the application of activation f to the weighted input $\mathbf{a} = W\mathbf{x}$. The output vector is taken to be $f(W\mathbf{x})$.

Neither activation function or network output have been defined; they are irrelevant to the functionality of network itself (and also to many applications of SOM).

3.3.2 Learning Procedure

In Section 3.2 the neuronal neighborhood discussion was postponed. A very general way of accounting for lateral feedback is to change the learning equation $\frac{dW}{dt} = F(W, \mathbf{x}, t)$ (for some *matrix* function F) by multiplying the right side with a scalar function $\psi_\ell(k)$, modelling the vicinity of winner k behaviour (thus $\psi_\ell(k)$ will only be non–zero for neurons ℓ near the winner k and the graph of $\psi_{(\cdot)}(k)$ will typically be a "mexican hat" centered around neuron k):

❖ F, ψ_ℓ

$$\frac{dW_{(\ell,:)}}{dt} = \psi_\ell(k_{(t)})\, F_{(\ell,:)}$$

(note that the winner changes in time thus $k_{(t)}$). This function has to be non–zero only for affected neurons (winner's surroundings). For neurons ℓ' outside winner's influence $\psi_{\ell'}(k) = 0$ and thus $\frac{dW_{(\ell',:)}}{dt} = \widehat{\mathbf{0}}^{\mathrm{T}}$, i.e. these weights will not change for current input. The $\psi_\ell(k)$ functions are chosen such that:

$$\psi_\ell(k) \begin{cases} > 0 & \text{for } \ell \text{ relatively close to } k \\ \leqslant 0 & \text{for } \ell \text{ far, but not too much, from } k \\ = 0 & \text{for } \ell \text{ far away from } k \end{cases}$$

distances being measured network topology wise.

In matrix notation:

$$\frac{dW}{dt} = \boldsymbol{\psi}(k_{(t)})\, \widehat{\mathbf{1}}^{\mathrm{T}} \odot F \quad \Leftrightarrow \quad \frac{dW}{dt} = \boldsymbol{\psi}(k_{(t)}) \overset{\mathrm{C}}{\odot} F$$

(the $\boldsymbol{\psi}$ vector is K–dimensional).

Some examples of lateral feedback models are given below:

- Simple function:

$$\psi_\ell(k) = \begin{cases} h_+ & \text{for } \ell \in \{k - n_+, \dots, k, k + n_+\} \text{ (positive feedback)} \\ -h_- & \text{for } \ell \in \{k - n_+ - n_-, \dots, k - n_+ - 1\} \cup \\ & \qquad \{k + n_+ + 1, \dots, k + n_+ + n_-\} \\ & \text{(negative feedback)} \\ 0 & \text{otherwise} \end{cases}$$

❖ h_+, h_-

  where $h_\pm \in [0, 1]$, $h_\pm = \text{const.}$ defines the height of positive/negative feedback and $n_\pm \geqslant 1$, $n_\pm \in \mathbb{N}$ defines the neural neighborhood size. See Figure 3.3.

- Truncated Gaussian function:

$$\psi_\ell(k) = \begin{cases} h_+ e^{-d_{\ell k}^2} & \text{if } d_{\ell k} \leqslant d_{\mathsf{max}} \\ 0 & \text{otherwise} \end{cases}$$

  where $h_+ > 0$ is the constant defining the positive feedback height; there is no negative feedback. $d_{\ell k}$ is the "distance" between neurons ℓ and k (may be measured in different ways) and $d_{\mathsf{max}} > 0$ defines the neuronal neighborhood size. See Figure 3.4 on the facing page.

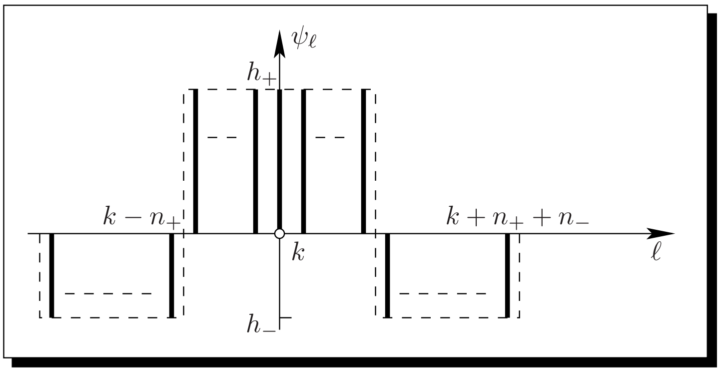

Figure 3.3: *The simple lateral feedback function.*

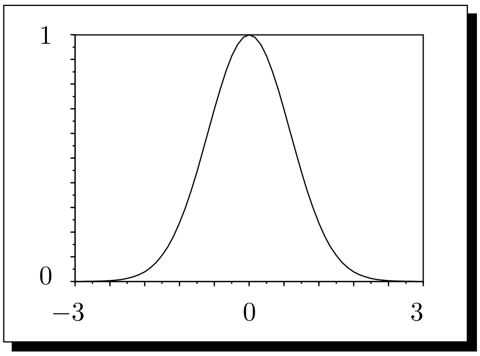

Figure 3.4: *The Gaussian lateral feedback function* $\psi_\ell(k) = e^{-d_{\ell k}^2}$.

As previously discussed, learning is unsupervised — there are *no targets*. First, a learning equation dW/dt and a lateral feedback model have to be selected (or built). Then, at time $t = 0$, the weights are initialized with (small) random values $W_{(0)} = W_0$. The weights $W_{(t)}$ (at time t) are updated using current input $\mathbf{x}_{(t)}$:

① For $\mathbf{x}_{(t)}$ find the winning neuron k for which:

$$\|W^{\mathrm{T}}_{(k,:)} - \mathbf{x}_{(t)}\| = \min_{\ell \in \{1,\dots,K\}} \|W^{\mathrm{T}}_{(\ell,:)} - \mathbf{x}_{(t)}\| \tag{3.12}$$

i.e. the one with associated weight vector $W_{(k,:)}$ closest to $\mathbf{x}$ (in weights space, both $\mathbf{x}$ and $W_{(\ell,:)}$ are from same space $\mathbb{R}^N$). The winner will be used to decide which weights have to be changed using the current $\mathbf{x}_{(t)}$.

② Update weights according to chosen model; see Section 3.2 for a selection of learning equations (dW/dt).

In discrete–time approximation and using the lateral feedback function $\boldsymbol{\psi}$, the adaptation equation $dW = \left(\frac{dW}{dt}\right) \cdot dt$ becomes:

$$\Delta W_{(t)} = \boldsymbol{\psi}(k_{(t)})\, \widehat{\mathbf{1}}^{\mathrm{T}} \odot \left(\frac{dW}{dt}\right)_{(t)} \quad \Leftrightarrow \quad \Delta W_{(t)} = \boldsymbol{\psi}(k_{(t)}) \overset{\mathrm{C}}{\odot} \left(\frac{dW}{dt}\right)_{(t)} \tag{3.13}$$

($dt \to \Delta t = t - (t-1) = 1$ and $dW_{(t)} \to \Delta W_{(t)} = W_{(t)} - W_{(t-1)}$).

All and only the neurons found in the winner's neighborhood participate to learning, i.e. will have their weights changed/adapted. All other weights remain unmodified at this stage; later a new input vector may replace the winner and thus the area of weight adaptation.

Remarks:

- $\|W_{(k,:)}^{\mathrm{T}} - \mathbf{x}\|$ is the distance between vectors $\mathbf{x}$ and $W_{(k,:)}$. This distance is user definable but the Euclidean distance is the most popular choice in practice.
- If the input vectors $\mathbf{x}$ and weight vectors $\{W_{(\ell,:)}\}_{\ell=\overline{1,K}}$ are normalized to the same value $\|\mathbf{x}\| = \|W_{(\ell,:)}\|$ (all $\mathbf{x}$ and $W_{(\ell,:)}$ are points on a hyper–sphere in $\mathbb{R}^N$) then winner k may be found through the dot product:

  $$W_{(k,:)}\mathbf{x} = \max_{\ell \in \{1,\ldots,K\}} W_{(\ell,:)}\mathbf{x}$$

  i.e. the winner's weight vector $W_{(k,:)}$ points in the *direction* closest to the one given by $\mathbf{x}$.

  This operation is faster, however it requires normalization of $\mathbf{x}$ and $W_{(\ell,:)}$ which is not always desirable in practice (also it may require a re–normalization of *adapted* weights and, for neighborhoods large relative to whole network, it may not be worth considering it).
- SOM As Kohonen algorithm "moves" the weights vectors towards input vectors, the Kohonen network tries to map the input vectors, i.e. the weights vectors will try to copy the topology of input vectors in the weight space. *This mapping occurs in weights space*. See Section 3.5.1 for an example. For this reason Kohonen networks are also called *self organizing maps* (SOM).
- Even if training is unsupervised, in fact a poor choice of parameters may lead to an incomplete learning (so, a full successful training may be seen as "supervised" at a "higher" level). See Section 3.5.1 and Figure 3.7 for an example of an incomplete learning.
- On large networks with small neighborhoods, using $\boldsymbol{\psi}$ directly may lead to inefficiencies as most Δw_{ki} will be zero. In this situation is probably better to use the ":" operator to "cut" a sub-matrix out of W, around winner k, containing the targeted neuronal vicinity. For example assuming that the neuronal neighborhood is of size ℓ on both sides of winner k (linear network topology) and the network does not "wraparound" then the sub-matrix affected by change is $W_{(\min(0,k-\ell):\max(k+\ell,K),:)}$ (the rest of W remains unaltered); a corresponding subvector of the $\boldsymbol{\psi}$ must also be used.

3.3.3 Initialization and Stopping Rules

As previously discussed, weights are initialized (in practice) with small random values (normalization is required if using the dot product to find the winner) and the adjusting process continues by iteration (in time–discrete steps).

Learning may be stopped by one of the following methods:

① choose a fixed number of steps T;

② continue learning until $|\Delta w_{ki(t)}| = |w_{ki(t)} - w_{ki(t-1)}| \leqslant \varepsilon$, where $\varepsilon \gtrsim 0$ is some specified threshold constant (weight adaptation becomes too small to go further with adaptation).

Another way to conclude learning is to multiply (3.13) by a stopping function $\tau(t)$ with the property $\lim_{t\to\infty} \tau(t) = 0$: ❖ τ

$$\Delta W_{(t)} = \tau(t)\, \boldsymbol{\psi}(k_{(t)})\, \widehat{\mathbf{1}}^{\mathrm{T}} \odot \left(\frac{dW}{dt}\right)_{(t)} \quad \Leftrightarrow \quad \Delta W_{(t)} = \tau(t)\, \boldsymbol{\psi}(k_{(t)}) \overset{\mathrm{C}}{\odot} \left(\frac{dW}{dt}\right)_{(t)} \tag{3.14}$$

This method has the benefit of ensuring termination of learning: $\lim_{t\to\infty} \Delta W = \widetilde{0}$, *for any model* (for divergent learning models it is necessary to select a $\tau(t)$ with a convergence faster than W's divergence). However, note that in practice it may be difficult to use the size of $\Delta W_{(t)}$ as the stopping criteria so a specification of a reasonable number of t steps instead may be easier to implement.

Some examples of stopping functions are:

- Geometrical progression: $\tau(t) = \tau_{\mathsf{init}} \tau_{\mathsf{ratio}}^t$, where $\tau_{\mathsf{init}}, \tau_{\mathsf{ratio}} \in (0,1)$. The τ_{init} and τ_{ratio} constants are the initial/ratio values.
- Exponential: $\tau(t) = \tau_{\mathsf{init}} e^{-g(t)}$, where $g(t) : \mathbb{N}^+ \to [0,\infty)$ is a *monotone increasing* function. Note that for $g(t) = ct$ ($c > 0$, $c =$ const.) this function is a geometric progression.

3.4 The Algorithm

The network running procedure represents just the calculation of $\mathbf{y} = f(W\mathbf{x})$, after a suitable activation has been chosen and the network trained.

From the discussion in previous sections, it becomes clear that there is a large number of possibilities for building a SOM network. An outline of the learning algorithm is given below:

1. For all neurons: initialize weights with (small) random values (normalize weights and inputs if using the dot product to find winners).
2. Choose a model — type of neuronal learning. See Section 3.2 for some examples of dW/dt equations.
3. Choose a model for the neuronal neighborhood — lateral feedback function. See Section 3.3.2 for some examples.
4. Choose a *stopping* condition. See Section 3.3.3 for some examples.
5. Knowing the learning model, the neuronal neighborhood function and the stopping condition, build the final procedure giving the weight adaptation algorithm, e.g. an equation of the form (3.14). Depending upon previous selections this step may not be reducible to one formula.
6. In discrete–time approximation repeat the following steps until the *stopping* condition is met:
   (a) Get the input vector $\mathbf{x}_{(t)}$.

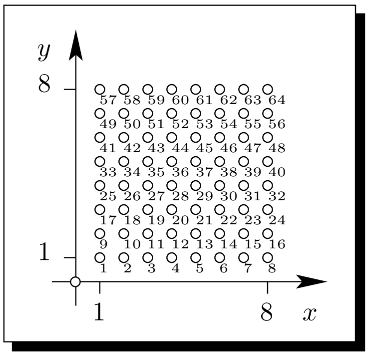

Figure 3.5: *The topological arrangement of a square mapping* 8×8 *network.*

(b) For all neurons ℓ in output layer, find the "winner" — a neuron k for which:

$$\|W^{\mathrm{T}}_{(k,:)} - \mathbf{x}_{(t)}\| = \min_{\ell \in \{1,\dots,K\}} \|W^{\mathrm{T}}_{(\ell,:)} - \mathbf{x}_{(t)}\|$$

(or if working with normalized vectors: $W_{(k,:)}\mathbf{x} = \max\limits_{\ell \in \{1,\dots,K\}} W_{(\ell,:)}\mathbf{x}$).

(c) Knowing the winner, change weights by using the adaptation algorithm built at step 5 (and re–normalize changed weights if working with dot product method).

3.5 Examples

3.5.1 Square Mapping

This example shows:

- how to build a adaptation formula;
- the mapping feature of Kohonen networks;
- how a poor choice of learning parameters may lead to incomplete learning.

Bidimensional Network

❖ L, K

Consider a bidimensional Kohonen network with 8×8 neurons arranged into a square pattern, $L = 8$ neurons per side, the total number of neurons being $K = L^2 = 64$. See Figure 3.5.

Weights are initialized with random values in the interval $[0, 1]$. At each time step t the network receives an bidimensional ($N = 2$) input vector $\mathbf{x}_{(t)}$, randomly selected from $[0, 1] \times [0, 1]$.

The winner is chosen according to (3.12), using the Euclidean distance:

$$\begin{aligned}\|\mathbf{x} - W^{\mathrm{T}}_{(k,:)}\|^2 &= (x_{1(t)} - w_{k1})^2 + (x_{2(t)} - w_{k2})^2 \\ &= \min_{\ell \in \{1,\dots,K\}} \left[(x_{1(t)} - w_{\ell 1})^2 + (x_{2(t)} - w_{\ell 2})^2\right]\end{aligned}$$

Remarks:

➥ A matrix form may be developed as follows:

$$B = (\widehat{\mathbf{1}}_K \mathbf{x}_{(t)}^{\mathrm{T}} - W)^{\odot 2} = (\mathbf{x}^{\mathrm{T}} \overset{\mathrm{R}}{\ominus} W)^{\odot 2} \quad , \quad \mathbf{b} = B_{(:,1)} + B_{(:,2)}$$

and the winning k is the one with the smallest corresponding b_k. This method leads to suboptimal memory consumptionb ut perhaps it may be considered for a non–standard system capable of fast matrix operations, e.g. massively parallel.

The learning model chosen is the trivial equation with $\alpha = 1$ and $\beta = 1$:

$$\frac{dW}{dt} = \widehat{\mathbf{1}}\mathbf{x}^{\mathrm{T}} - W \quad \Leftrightarrow \quad \frac{dW}{dt} = \mathbf{x}^{\mathrm{T}} \overset{\mathrm{R}}{\ominus} W \tag{3.15}$$

The lateral feedback function is chosen of the form:

$$\psi_\ell(k) = h_+ \exp\left[-\frac{d_{\ell k}^2}{(d_{\mathsf{init}} d_{\mathsf{rate}}^t)^2}\right] \tag{3.16}$$

where $d_{\ell k}$ is the Euclidian distance between neurons ℓ and k (see Figure 3.5) and $h_+ = 0.8$, $d_{\mathsf{init}} = 4$ and $d_{\mathsf{rate}} = 0.99995$. Using the Euclidean distance then $d_{\ell k}^2 = d_{(x)\ell k}^2 + d_{(y)\ell k}^2$ (an "x" part and a "y" one). But for the $K = L^2$ neurons there are only L values for $d_{(x)\ell k}^2$, respectively $d_{(y)\ell k}^2$, for a given k. Let:

❖ $d_{\ell k}$
❖ h_+, d_{init}, d_{rate}
❖ $\boldsymbol{\delta}$, $d_{(t)}$

$$\boldsymbol{\delta} \equiv \begin{pmatrix} 1, & \cdots, & L \end{pmatrix}^{\mathrm{T}} \quad \text{and} \quad d_{(0)}^2 = d_{\mathsf{init}}^2 \ , \ d_{(t)}^2 = d_{(t-1)}^2 d_{\mathsf{rate}}^2$$

and $k_{x(t)}, k_{y(t)} \in \{1, \ldots, L\}$ be the winner's coordinates:

❖ $k_{x(t)}$, $k_{y(t)}$

$$k_{x(t)} = \underset{L}{\mathrm{mod}}(k_{(t)} - 1) + 1 \quad \text{and} \quad k_{y(t)} = \frac{k_{(t)} - k_{x(t)}}{L} + 1$$

Then the L combinations of coordinates for "x" and "y" may be stored as two vectors:

❖ $\mathbf{d}_{(x)(t)}$, $\mathbf{d}_{(y)(t)}$

$$\mathbf{d}_{(x)(t)}^{\odot 2} = (\boldsymbol{\delta} - k_{x(t)} \widehat{\mathbf{1}}_L)^{\odot 2} = (\boldsymbol{\delta} \overset{\mathrm{R}}{\ominus} k_{x(t)})^{\odot 2}$$

$$\mathbf{d}_{(y)(t)}^{\odot 2} = (\boldsymbol{\delta} - k_{y(t)} \widehat{\mathbf{1}}_L)^{\odot 2} = (\boldsymbol{\delta} \overset{\mathrm{R}}{\ominus} k_{y(t)})^{\odot 2}$$

and the vectorial lateral feedback function is:

$$\boldsymbol{\psi}(k_{(t)}) = h_+ \exp\left[-\frac{\mathbf{d}_{(y)(t)}^{\odot 2}}{d_{(t)}^2}\right] \otimes \exp\left[-\frac{\mathbf{d}_{(x)(t)}^{\odot 2}}{d_{(t)}^2}\right] \tag{3.17}$$

(proved by checking all elements against (3.16)). Note the order of "$\otimes$" multiplication, all neurons from same row have same "y" distance to winner (respectively all neurons from same column have same "x" distance, see Figure 3.5).

The stopping function is chosen as the geometrical progression:

$$\tau(0) = \tau_{\mathsf{init}} \ , \quad \tau(t) = \tau(t-1)\, \tau_{\mathsf{rate}} \tag{3.18}$$

with $\tau_{\mathsf{init}} = 1$ and $\tau_{\mathsf{rate}} = 0.9999$.

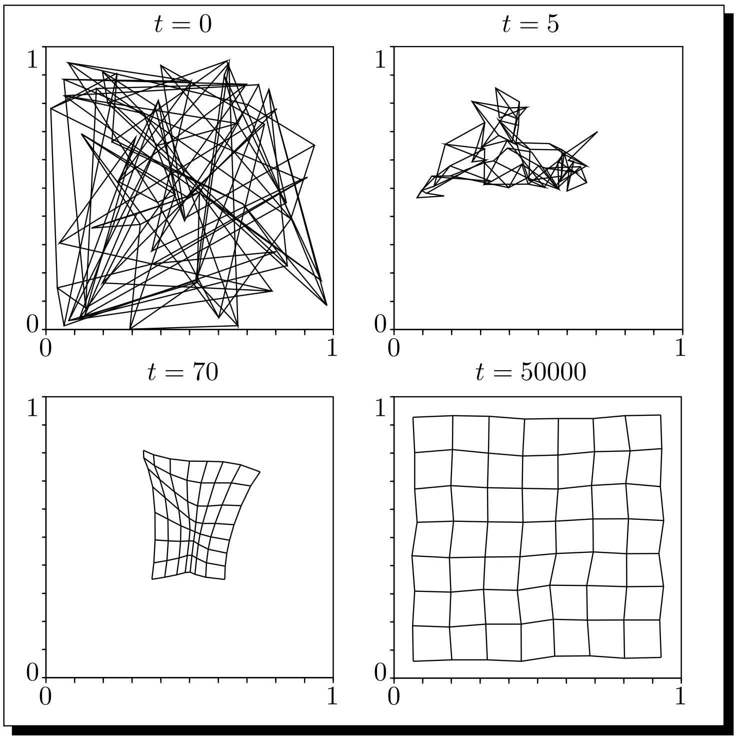

Figure 3.6: *Mapping of the $[0, 1] \times [0, 1]$ square by an 8×8 network in weights space. Lines are drawn between weights (represented as points at intersections) corresponding to neuronal neighbors (network topology wise, see Figure 3.5). The snapshots have been taken at various t.*

The general weights update formula is obtained by combining (3.15), (3.17) and (3.18) (as described in Section 3.4 and formula (3.14)):

$$W_{(t+1)} = W_{(t)} + \tau(t)\,\boldsymbol{\psi}(k_{(t)})\widehat{\mathbf{1}}_K^{\mathrm{T}} \odot (\widehat{\mathbf{1}}_K \mathbf{x}_{(t)}^{\mathrm{T}} - W_{(t)})$$

$$= W_{(t)} + \tau(t)\,\boldsymbol{\psi}(k_{(t)}) \overset{\mathrm{C}}{\odot} (\mathbf{x}_{(t)}^{\mathrm{T}} \overset{\mathrm{R}}{\ominus} W_{(t)})$$

The results of network learning[3] at various times are shown in Figure 3.6.

Even if the learning is unsupervised, in fact, a poor choice of learning parameters (e.g. h_+, etc.) may lead to an incomplete learning. See Figure 3.7 on the facing page: small values of feedback function at the beginning of learning results in the network being unable to "deploy" itself fast enough, leading to the appearance of "twist" in the mapping. The network was the same as the one used to generate Figure 3.6 including same inputs and same weights initial values. The only parameters changed were: $h_+ = 0.3$ and $d_{\mathsf{init}} = 2$.

[3]The Scilab source, dynamically showing the weight adaptation, may be found in "`Matrix_ANN_1/Scilab/som_w_map_anim.sci`" (actual version number may differ).

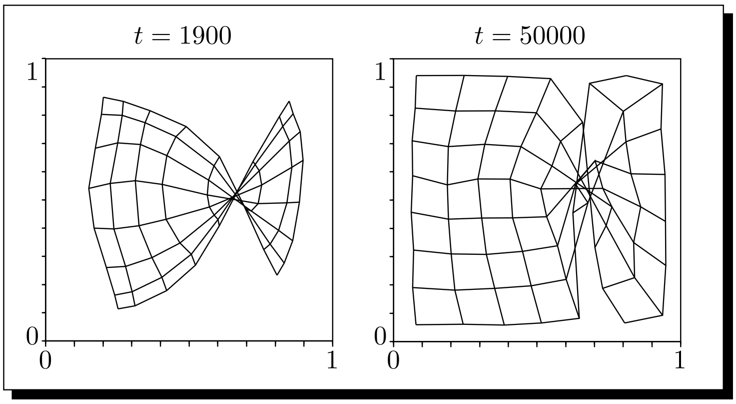

Figure 3.7: *Incomplete learning of the* $[0,1] \times [0,1]$ *square by an* 8×8 *network.*

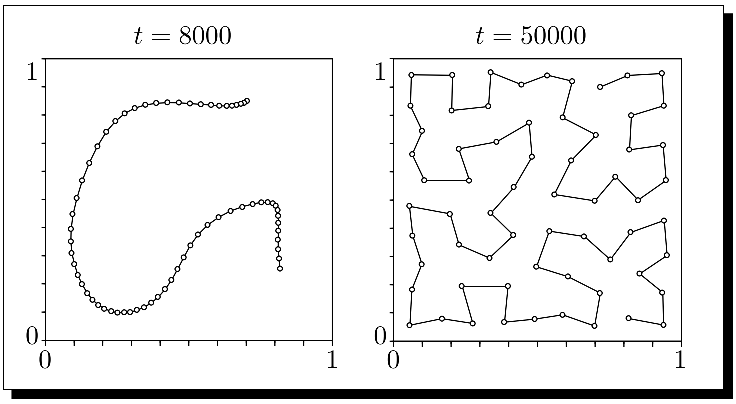

Figure 3.8: *Mapping of bidimensional input space by a unidimensional network. Lines are drawn between neuronal neighbors (network topology wise). Weight values are marked with small circles.*

Unidimensional network

An interesting aspect of SOM networks is the answer to question "What happens when a lower dimensionality network tries to map an higher dimensionality input space ?".

A network similar to previous example was used to map same input space (same inputs, number of neurons, weights initial values). The network topology was unidimensional and this is reflected in the new lateral feedback function (compare with (3.17)):

$$\boldsymbol{\delta} \equiv \begin{pmatrix} 1 & \cdots & K \end{pmatrix}^{\mathrm{T}} \quad \Rightarrow \quad \mathbf{d}_{(x)(t)}^{\odot 2} = (\boldsymbol{\delta} - k_{(t)} \widehat{\mathbf{1}}_K)^{\odot 2} = (\boldsymbol{\delta} \overset{\mathrm{R}}{\ominus} k)^{\odot 2}$$

$$\boldsymbol{\psi}(k_{(t)}) = h_+ exp \left[-\frac{\mathbf{d}_{(x)(t)}^{\odot 2}}{d_{(t)}^2} \right]$$

Other imposed changes were $d_{\mathsf{init}} = 15$, $d_{\mathsf{rate}} = 0.9999$.

The results are depicted in Figure 3.8. The network "folds", *in the weights space*, trying to cover the higher dimensions of input space.

3.5.2 Feature Map

This example shows the clustering capabilities of SOM networks.

Consider a collection of objects each of which is associated with a set of 6 features. The presence of a feature may be marked with "1" while it's absence with "0". Thus each object has attached a 6-dimensional binary vector describing the presence or absence of all features considered. On the basis of the feature set alone, all objects may be classified in $2^6 = 64$ categories. The purpose is to create a bidimensional map of these categories such that those with most common features present (or absent) are clustered together.

Let $N = 6$, $P = 64$ and $\mathbf{x}_p$ the set of binary vectors representing the binary representation of numbers $\{0, \ldots, 64\}$ (i.e. from $\begin{pmatrix} 0, & \cdots, & 0 \end{pmatrix}^{\mathrm{T}}$ to $\begin{pmatrix} 1, & \cdots & 1, \end{pmatrix}^{\mathrm{T}}$). The $\{\mathbf{x}_p\}$ will be used to represent the 64 categories.

Clustering is performed using the mapping feature of SOM networks. After training, different neurons will be winners for different $\mathbf{x}_p$ but similar $\mathbf{x}_p$ will cluster together the corresponding winning neurons.

A bidimensional network, similar to the one described in Section 3.5.1 was trained and built[4]. The differences were as follows:

- Network dimension was $L = 9$ (9×9 network). While the dimension could have been chosen as $L = 8$ (as $8^2 = 64$) however it would have been very tight and hard to train.
- The $\{\mathbf{x}_p\}$ set was used repeatedly over $T = 150$ epochs; a random shuffling was performed between epochs. Note that $\{\mathbf{x}_p\}$ represents the coordinates of a hypercube in $\mathbb{R}^6$ (with edge equal 1).
- Other changed parameters were: $\boldsymbol{\delta} = \begin{pmatrix} 1, & \cdots, & 9 \end{pmatrix}^{\mathrm{T}}$, $\tau_{\mathsf{rate}} = 0.9997$, $d_{\mathsf{rate}} = 0.997$.

After training finished, the network was calibrated, i.e. for each input $\mathbf{x}_p$ the corresponding winning neuron was identified. The result, network topology wise, is (non-winning neurons are marked with a "$\cdot$"):

$$
\text{decimal base: }
\begin{array}{ccccccccc}
35 & 33 & 1 & 5 & 37 & 45 & 44 & 10 & 26 \\
34 & 32 & \cdot & 4 & \cdot & 47 & 46 & 42 & 58 \\
2 & \cdot & 0 & 36 & 38 & 39 & \cdot & 62 & 56 \\
18 & \cdot & 16 & \cdot & 54 & 55 & 63 & \cdot & 60 \\
50 & 48 & \cdot & 20 & 52 & 53 & \cdot & \cdot & 61 \\
51 & 49 & \cdot & 24 & 17 & 21 & \cdot & 31 & 29 \\
59 & 57 & 25 & 27 & 19 & \cdot & 7 & 15 & 13 \\
43 & \cdot & \cdot & 11 & 3 & \cdot & 6 & 12 & 28 \\
41 & 40 & 8 & 9 & \cdot & 23 & 22 & 14 & 30
\end{array}
\qquad
\text{octal base: }
\begin{array}{ccccccccc}
43 & 41 & 01 & 05 & 45 & 55 & 54 & 12 & 32 \\
42 & 40 & \cdot & 04 & \cdot & 57 & 56 & 52 & 72 \\
02 & \cdot & 00 & 44 & 46 & 47 & \cdot & 76 & 70 \\
22 & \cdot & 20 & \cdot & 66 & 67 & 77 & \cdot & 74 \\
62 & 60 & \cdot & 24 & 64 & 65 & \cdot & \cdot & 75 \\
63 & 61 & \cdot & 30 & 21 & 25 & \cdot & 37 & 35 \\
73 & 71 & 31 & 33 & 23 & \cdot & 07 & 17 & 15 \\
53 & \cdot & \cdot & 13 & 03 & \cdot & 06 & 14 & 34 \\
51 & 50 & 10 & 11 & \cdot & 27 & 26 & 16 & 36
\end{array}
$$

and it represents the required map. To see how well the net performed, the number of different bits between neighbours was calculated; H represents differences between neighbours on horizontal direction, V on vertical, $D_{\searrow}$ on diagonal $\searrow$ and $D_{\nearrow}$ on diagonal $\nearrow$

[4] The actual Scilab script is in: "`Matrix_ANN_1/Scilab/feature_map.sci`" (actual version numbers may differ).

respectively. The results are:

$$H = \begin{array}{cccccccc} 1&1&1&1&1&1&3&1\\ 1&\cdot&\cdot&\cdot&\cdot&1&1&1\\ \cdot&\cdot&2&1&1&\cdot&\cdot&2\\ \cdot&\cdot&\cdot&\cdot&1&1&\cdot&\cdot\\ 1&\cdot&\cdot&1&1&\cdot&\cdot&\cdot\\ 1&\cdot&\cdot&2&1&\cdot&\cdot&1\\ 1&1&1&1&\cdot&\cdot&1&1\\ \cdot&\cdot&\cdot&1&\cdot&\cdot&2&1\\ 1&1&1&\cdot&\cdot&1&2&1 \end{array} \quad V = \begin{array}{ccccccccc} 1&1&\cdot&1&\cdot&1&1&1&1\\ 1&\cdot&\cdot&1&\cdot&1&\cdot&2&1\\ 1&\cdot&1&\cdot&1&1&\cdot&\cdot&1\\ 1&\cdot&\cdot&\cdot&1&1&\cdot&\cdot&1\\ 1&1&\cdot&2&3&1&\cdot&\cdot&1\\ 1&1&\cdot&2&1&\cdot&\cdot&1&1\\ 1&\cdot&\cdot&1&1&\cdot&1&2&2\\ 1&\cdot&\cdot&1&\cdot&\cdot&1&1&1 \end{array} \quad D_{\searrow} = \begin{array}{cccccccc} 2&\cdot&2&\cdot&2&2&2&2\\ \cdot&1&\cdot&2&\cdot&\cdot&1&2\\ \cdot&\cdot&\cdot&2&2&2&\cdot&1\\ 2&\cdot&1&\cdot&2&\cdot&\cdot&\cdot\\ 2&\cdot&\cdot&2&2&\cdot&\cdot&\cdot\\ 2&2&\cdot&3&\cdot&2&\cdot&2\\ \cdot&\cdot&2&2&\cdot&\cdot&3&3\\ 2&\cdot&\cdot&\cdot&2&\cdot&1&2 \end{array} \quad D_{\nearrow} = \begin{array}{cccccccc} 2&2&\cdot&2&\cdot&2&2&2\\ 2&\cdot&1&\cdot&2&2&\cdot&1\\ \cdot&\cdot&3&\cdot&2&\cdot&1&\cdot\\ \cdot&1&\cdot&2&2&2&\cdot&\cdot\\ 2&\cdot&\cdot&3&2&\cdot&\cdot&2\\ 2&\cdot&1&2&2&\cdot&2&2\\ 2&\cdot&\cdot&2&\cdot&\cdot&2&1\\ \cdot&\cdot&2&2&\cdot&2&3&2 \end{array}$$

The following comments are to be made:

- Most differences between horizontal and vertical neighbors equal 1 bit (see H and V), i.e. the clustering was achieved.
- Most differences between diagonal neighbors are 2 but this is to be expected. Consider a neuron somewhere in the middle of network and the smallest square neighborhood: it contains 4 neurons on horizontal/vertical positions and 4 neurons on diagonal positions. But for each pattern there are only 5 others which differ by one component. Considering the horizontal/vertical distance between neighbors as 1 unit long then the diagonal ones are at $\sqrt{2} \simeq 1.41$ units, i.e. further apart. It to be expected that the closest 4 positions are occupied by "one component different" vectors, this leaving just one for the rest of the 4 diagonal neighbors, i.e. at least 3 of them will be occupied by "2 components different" vectors even in an optimal arrangement (or remain unoccupied, but the map is quite "tight").
- There are several sub–optimally mapped "spots", e.g. around $\mathbf{x}_{17}$. Probably a different choice of parameters, models, etc., would achieve a better mapping.

CHAPTER 4

The BAM / Hopfield Memory

This network illustrates an (auto)associative memory. Unlike the classical von Neumann systems where there is no relation between memory address and its contents, in ANN part of the information is used to retrieve the rest associated with it[1].

4.1 Associative Memory

Consider P pairs of vectors $\{(\mathbf{x}_1, \mathbf{y}_1), \ldots (\mathbf{x}_P, \mathbf{y}_P)\}$, named *exemplars*, with $\mathbf{x}_p \in \mathbb{R}^N$ and $\mathbf{y}_p \in \mathbb{R}^K$, $N, K, P \in \mathbb{N}^+$.

exemplars

Definition 4.1.1. *The mapping $\mathcal{M} : \mathbb{R}^N \to \mathbb{R}^K$ is said to implement an heteroassociative memory if:*

heteroassociative memory

$$\mathcal{M}(\mathbf{x}_p) = \mathbf{y}_p \quad \forall p \in \{1, \ldots, P\}$$

$$\mathcal{M}(\mathbf{x}) = \mathbf{y}_p \quad \forall \mathbf{x} \text{ such that } \|\mathbf{x} - \mathbf{x}_p\| < \|\mathbf{x} - \mathbf{x}_q\| \quad \forall q \in \{1, \ldots, P\},\ q \neq p$$

Definition 4.1.2. *The mapping $\mathcal{M} : \mathbb{R}^N \to \mathbb{R}^K$ is said to implement an interpolative associative memory if:*

interpolative associative memory

$$\mathcal{M}(\mathbf{x}_p) = \mathbf{y}_p \quad \forall p \in \{1, \ldots, P\} \qquad \text{and}$$

$$\forall\, \mathbf{d} \neq \widehat{\mathbf{0}}\,, \exists\, \mathbf{e} \neq \widehat{\mathbf{0}}\,, \text{ such that } \mathcal{M}(\mathbf{x}_p + \mathbf{d}) = \mathbf{y}_p + \mathbf{e} \qquad (\mathbf{d} \in \mathbb{R}^N\,,\ \mathbf{e} \in \mathbb{R}^K)$$

i.e. if $\mathbf{x} \neq \mathbf{x}_p$, then $\mathbf{y} = \mathcal{M}(\mathbf{x}) \neq \mathbf{y}_p$, $\forall p \in \{1, \ldots, P\}$.

[1] An example regarding the usefulness of this type of memory is given in Section 4.5.1 (thesaurus).

[4.1] See [FS92] pp. 130–131.

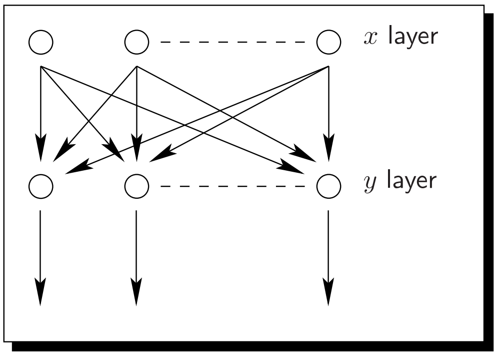

Figure 4.1: *The BAM network structure.*

An interpolative associative memory may be build from a set of orthonormal set of exemplars $\{\mathbf{x}_p\}$ and a linearly independent set of $\{\mathbf{y}_p\}$ $(P = N = K)$. The $\mathcal{M}$ function is then defined as:

$$\mathcal{M}(\mathbf{x}) = \left(\sum_{p=1}^{P} \mathbf{y}_p \mathbf{x}_p^{\mathrm{T}}\right) \mathbf{x} \tag{4.1}$$

Proof. Orthonormality of $\{\mathbf{x}_p\}$ means that $\mathbf{x}_p^{\mathrm{T}} \mathbf{x}_q = \delta_{pq}$.

From Equation 4.1: $\mathcal{M}(\mathbf{x}_q) = \left(\sum_{p=1}^{P} \mathbf{y}_p \mathbf{x}_p^{\mathrm{T}}\right) \mathbf{x}_q = \sum_{p=1}^{P} \mathbf{y}_p \mathbf{x}_p^{\mathrm{T}} \mathbf{x}_q = \sum_{p=1}^{P} \mathbf{y}_p \delta_{pq} = \mathbf{y}_q$.

For $\mathbf{x} = \mathbf{x}_q + \mathbf{d}$: $\mathcal{M}(\mathbf{x}) = \mathcal{M}(\mathbf{x}_q + \mathbf{d}) = \mathbf{y}_q + \mathbf{e}$ where $\mathbf{e} = \sum_{p=1}^{P} \mathbf{y}_p \mathbf{x}_p^{\mathrm{T}} \mathbf{d}$ (clearly $\mathcal{M}$ is linear with $\mathcal{M}(\mathbf{x}_q + \mathbf{d}) = \mathcal{M}(\mathbf{x}_q) + \mathcal{M}(\mathbf{d})$). If $\mathbf{d} \neq \widehat{\mathbf{0}}$ then not all $\mathbf{x}_p^{\mathrm{T}} \mathbf{d}$ are zero and therefore $\mathbf{e}$ is non–zero by the linear independence of the $\{\mathbf{y}_p\}$. □

autoassociative memory

Definition 4.1.3. *The mapping $\mathcal{M} : \mathbb{R}^K \to \mathbb{R}^K$ is said to implement an autoassociative memory if:*

$$\mathcal{M}(\mathbf{y}_p) = \mathbf{y}_p \quad \forall p \in \overline{1, P}$$
$$\mathcal{M}(\mathbf{y}) = \mathbf{y}_p \quad \forall \mathbf{y} \textit{ such that } \|\mathbf{y} - \mathbf{y}_p\| < \|\mathbf{y} - \mathbf{y}_q\| \quad \forall q \in \overline{1, P},\ q \neq p$$

4.2 BAM Architecture and Dynamics

4.2.1 The Architecture

BAM

The BAM (Bidirectional Associative Memory) implements a interpolative associative memory and consists of 2 layers of neurons fully interconnected.

The Figure 4.1 shows the net as $\mathcal{M}(\mathbf{x}) = \mathbf{y}$ but the input and output may swap places, i.e. the direction of connection arrows may be reversed and $\mathbf{y}$ play the role of input, using the same weight matrix (but transposed, see below), i.e. the network is reversible.

4.2 See [FS92] pp. 131–132.

4.2.2 BAM Dynamics

Running Procedure

The procedure is developed for vectors belonging to Hamming space $\mathbb{H}$. Due to the fact that most information may be encoded in binary form this is not a significant limitation and it does improve the reliability and speed of the net.

Bipolar vectors are used (with component values either $+1$ or -1). A transition to and from binary vectors (component values $\in \{0, 1\}$) can be easily performed using the following relation: $\mathbf{x} = 2\widetilde{\mathbf{x}} - \widehat{\mathbf{1}}$ where $\mathbf{x}$ is a bipolar (Hamming) vector and $\widetilde{\mathbf{x}}$ is a binary vector.

Remarks:

- BAM may also work for binary vectors but experimentally it was shown to be less accurate, see [Kos92], pp. 88–89. Intuitively: $(+1) + (-1) = 0$ while $1 + 0 = 1$, i.e. binary vectors are biased with respect to "$+$" operation.

The BAM functionality differs from the functionality of other architectures in that *weights are not adjusted* during a training process but calculated directly from the set of vectors to be stored $\{\mathbf{x}_p, \mathbf{y}_p\}_{p=\overline{1,P}}$. From a set of vectors $\{\mathbf{x}_p, \mathbf{y}_p\}$, the weight matrix is defined as:

$$W = \sum_{p=1}^{P} \omega_p \mathbf{y}_p \mathbf{x}_p^{\mathrm{T}} \quad , \quad \omega_p > 0 \ , \ \omega_p = \text{const.} \tag{4.2}$$

Both $\{\mathbf{x}_p\}$ and $\{\mathbf{y}_p\}$ sets have to be orthogonal as the network works also in reverse. ω_p is a weighting factor influencing the priority of storage for $\{\mathbf{x}_p, \mathbf{y}_p\}$ pair. Note that new pattern sets may be added to W at any time; if ω_p is chosen as an increasing sequence ($\omega_1 < \cdots < \omega_P$) then old patterns will gradually "fade" in favor of those more recently stored ("forgetting" effect). The same weight w_{ik} is utilized for $i \leftrightarrow k$ neuronal connection ($w_{ik} = w_{ki}$); this means that the total input to layer y is $\mathbf{a}_y = W\mathbf{x}$ while for layer x, $\mathbf{a}_x = W^{\mathrm{T}}\mathbf{y}$, i.e. for the connections from y to x layers the transpose of W is to be used. ❖ ω_p

BAM uses the threshold activation function and passive decay with resting potential as its model for activation dynamics:

$$\frac{d\mathbf{x}}{dt} = -\mathbf{x} + f(W^{\mathrm{T}}\mathbf{y}) \quad \text{and} \quad \frac{d\mathbf{y}}{dt} = -\mathbf{y} + f(W\mathbf{x})$$

(f being the threshold function).

In discrete–time approximation $dt \to \Delta t = 1$, $d\mathbf{x} \to \Delta\mathbf{x} = \mathbf{x}_{(t+1)} - \mathbf{x}_{(t)}$ and then the update equation $d\mathbf{x}/dt = \ldots$ becomes $\mathbf{x}_{(t+1)} = f(W^{\mathrm{T}}\mathbf{y}_{(t)})$; for the "$y$" equation the procedure is similar but the last available $\mathbf{x}$ (i.e. $\mathbf{x}_{(t+1)}$) will be used instead.

BAM works by propagating the information forward and back between x and y layers, until a stable state is reached and subsequently a pair $\{\mathbf{x}, \mathbf{y}\}$ is retrieved.

The procedure progresses as follows:

- At $t = 0$ the $\mathbf{x}_{(0)}$ is applied to the net and $\mathbf{y}$ is initialized to $\mathbf{y}_{(0)} = \operatorname{sign}(W\mathbf{x}_{(0)})$.

[4.2.2] See [FS92] pp. 132–136 and [Kos92] pp. 63–64, pp. 80–81, pp. 88–89 and pg. 92.

- The outputs of x and y layers are propagated back and forward, until a stable state is reached, according to formulas:

$$x_{i(t+1)} = f(W_{(:,i)}^{\mathrm{T}}\, \mathbf{y}_{(t)}) = \begin{cases} +1 & \text{if } W_{(:,i)}^{\mathrm{T}}\, \mathbf{y}_{(t)} > 0 \\ x_{i(t)} & \text{if } W_{(:,i)}^{\mathrm{T}}\, \mathbf{y}_{(t)} = 0 \\ -1 & \text{if } W_{(:,i)}^{\mathrm{T}}\, \mathbf{y}_{(t)} < 0 \end{cases} \quad , \quad i = \overline{1,N}$$

$$y_{k(t+1)} = f(W_{(k,:)}\, \mathbf{x}_{(t+1)}) = \begin{cases} +1 & \text{if } W_{(k,:)}\, \mathbf{x}_{(t+1)} > 0 \\ y_{k(t)} & \text{if } W_{(k,:)}\, \mathbf{x}_{(t+1)} = 0 \\ -1 & \text{if } W_{(k,:)}\, \mathbf{x}_{(t+1)} < 0 \end{cases} \quad , \quad k = \overline{1,K}$$

Note that the "0" part of threshold activation was modified and old values of $\mathbf{x}$ and $\mathbf{y}$ are kept instead of setting $x_{i(t+1)}$ (respectively $y_{k(t+1)}$) to zero. This case seldom appears in practice (on large networks).

In matrix notation:

$$\mathbf{x}_{(t+1)} = \mathcal{H}\{W^{\mathrm{T}}\mathbf{y}_{(t)}, \mathbf{x}_{(t)}, = +1, =, = -1\} \tag{4.3a}$$

$$\mathbf{y}_{(t+1)} = \mathcal{H}\{W\mathbf{x}_{(t+1)}, \mathbf{y}_{(t)}, = +1, =, = -1\} \tag{4.3b}$$

and stability is reached when there are no further changes, i.e. $\mathbf{x}_{(t+1)} = \mathbf{x}_{(t)}$ and $\mathbf{y}_{(t+1)} = \mathbf{y}_{(t)}$. Convergence of the process is ensured by Theorem 4.2.1.

Remarks:

➥ If working with binary vectors, replace "$= -1$" part, in (4.3), with "$= 0$" (a similar comment applies to (4.4)).

➥ An alternative formulation of (4.3) is:

$$\mathbf{x}_{(t+1)} = \mathrm{sign}(W^{\mathrm{T}}\mathbf{y}_{(t)}) + \{\widehat{\mathbf{1}} - \mathrm{abs}[\mathrm{sign}(W^{\mathrm{T}}\mathbf{y}_{(t)})]\} \odot \mathbf{x}_{(t)}$$

$$\mathbf{y}_{(t+1)} = \mathrm{sign}(W\mathbf{x}_{(t+1)}) + \{\widehat{\mathbf{1}} - \mathrm{abs}[\mathrm{sign}(W\mathbf{x}_{(t+1)})]\} \odot \mathbf{y}_{(t)}$$

Proof. Consider the proposed new form of (4.3a). $\mathrm{sign}(W^{\mathrm{T}}\mathbf{y}_{(t)})$ gives the correct (± 1) values of $\mathbf{x}_{(t+1)}$ for the changing components and is zero if $W_{(:,i)}^{\mathrm{T}}\mathbf{y} = 0$.

The vector $\widehat{\mathbf{1}} - \mathrm{abs}[\mathrm{sign}(W\mathbf{y}_{(t)})]$ has its elements equal to 1 only for those x_i components which have to remain unchanged and thus restores the values of $\mathbf{x}$ to the previous ones (only for those elements requiring it).

The proof for second formula is similar. ☐

➥ This network may be used to build an autoassociative memory by taking $\mathbf{x} \equiv \mathbf{y}$. The weight matrix becomes: $W = \sum\limits_{p=1}^{P} \omega_p \mathbf{y}_p \mathbf{y}_p^{\mathrm{T}}$.

When working in reverse $\mathbf{y}_{(0)}$ is applied to the net, $\mathbf{x}$ is initialized to $\mathbf{x}_{(0)} = \mathrm{sign}(W^{\mathrm{T}}\mathbf{y}_{(0)})$ and the formulas change to:

$$\mathbf{y}_{(t+1)} = \mathcal{H}\{W\mathbf{x}_{(t)}, \mathbf{y}_{(t)}, = +1, =, = -1\} \tag{4.4a}$$

$$\mathbf{x}_{(t+1)} = \mathcal{H}\{W^{\mathrm{T}}\mathbf{y}_{(t+1)}, \mathbf{x}_{(t)}, = +1, =, = -1\} \tag{4.4b}$$

Remarks:

➥ An alternative to (4.4) (proved the same way as above) is:

$$\mathbf{y}_{(t+1)} = \operatorname{sign}(W\mathbf{x}_{(t)}) + \{\widehat{\mathbf{1}} - \operatorname{abs}[\operatorname{sign}(W\mathbf{x}_{(t)})]\} \odot \mathbf{y}_{(t)}$$

$$\mathbf{x}_{(t+1)} = \operatorname{sign}(W^{\mathrm{T}}\mathbf{y}_{(t+1)}) + \{\widehat{\mathbf{1}} - \operatorname{abs}[\operatorname{sign}(W^{\mathrm{T}}\mathbf{y}_{(t+1)})]\} \odot \mathbf{x}_{(t)}$$

The Hamming or binary vectors are symmetric with respect to the information they carry: $\mathbf{x}$ contains the same amount of information as its complement $\mathbf{x}^{\mathrm{C}}$. The BAM network stores both automatically because only the direction of these vectors counts, not the orientation. When trying to retrieve a vector, if the initial $\mathbf{x}_{(0)}$ is closer to the complement $\mathbf{x}_p^{\mathrm{C}}$ of a stored $\mathbf{x}_p$ then the complement pair will be retrieved $\{\mathbf{x}^{\mathrm{C}}, \mathbf{y}^{\mathrm{C}}\}$ (because both *exemplars* and their complements are stored with equal precedence; the same discussion applies when working in reverse).

The theoretical upper limit (number of vectors to be stored) of BAM is limited by two factors:

- the $\{\mathbf{x}_p\}$ and $\{\mathbf{y}_p\}$ have to be orthogonal sets hence $P \leqslant \min(N, K)$;
- the $\mathbf{x}_p$ and $-\mathbf{x}_p$ carry the same amount of information (due to symmetry), similarly for $\mathbf{y}_p$.

Combining these conditions gives $P \leqslant \min(N, K)/2$. But, if the possibility to work with noisy data is sought, then the real capacity is much lower because of the crosstalk — a different vector from the desired one is retrieved (the input with noise may be closer to another stored vector). crosstalk

Remarks:

➥ Practice shows that the capacity of BAM may be greatly improved by using some real–valued threshold vectors $\mathbf{t}_x$ and $\mathbf{t}_y$. Equations (4.3) and (4.4) change to:

$$\begin{cases} \mathbf{x}_{(t+1)} = \mathcal{H}\{W^{\mathrm{T}}\mathbf{y}_{(t)} - \mathbf{t}_x, \mathbf{x}_{(t)}, = +1, =, = -1\} \\ \mathbf{y}_{(t+1)} = \mathcal{H}\{W\mathbf{x}_{(t+1)} - \mathbf{t}_y, \mathbf{y}_{(t)}, = +1, =, = -1\} \end{cases}$$

$$\begin{cases} \mathbf{y}_{(t+1)} = \mathcal{H}\{W\mathbf{x}_{(t)} - \mathbf{t}_y, \mathbf{y}_{(t)}, = +1, =, = -1\} \\ \mathbf{x}_{(t+1)} = \mathcal{H}\{W^{\mathrm{T}}\mathbf{y}_{(t+1)} - \mathbf{t}_x, \mathbf{x}_{(t)}, = +1, =, = -1\} \end{cases}$$

The rationale is that threshold vectors define hyperplanes (perpendicular to the threshold vectors) in the $[-1, 1]^{N \text{ or } K}$ spaces. The values of $x_{i(t+1)}$ are decided depending on relative position of $W^{\mathrm{T}}\mathbf{y}_{(t)}$ with respect to the hyperplane (same for $y_{k(t+1)}$). There are not yet good methods to find the best threshold vectors given the data.

➥ In practice the stored vectors are not always checked for orthogonality because for $P \ll \min(N, K)/2$ the sets of vectors in Hamming space are usually close to orthogonal. However, when BAM memory is gradually filled up, crosstalk may begin to appear (even in the absence of noise) when P is approaching $\min(N, K)/2$.

4.2.3 Energy Function

❖ E
BAM energy

Definition 4.2.1. *The BAM energy function*[2] *is defined as:*

$$E(\mathbf{x}, \mathbf{y}) = -\mathbf{y}^{\mathrm{T}} W \mathbf{x} \tag{4.5}$$

Theorem 4.2.1. *The BAM energy function has the following properties:*

1. *Any change in $\mathbf{x}$ or $\mathbf{y}$ (during a network run) results in an energy decrease, i.e. if $[\mathbf{x}_{(t+1)}, \mathbf{y}_{(t+1)}] \neq [\mathbf{x}_{(t+1)}, \mathbf{y}_{(t+1)}]$ then $E_{(t+1)}(\mathbf{x}_{(t+1)}, \mathbf{y}_{(t+1)}) < E_{(t)}(\mathbf{x}_{(t+1)}, \mathbf{y}_{(t+1)})$.*
2. *E is bounded below by $E_{\text{lower}} = -\widehat{\mathbf{1}}^{\mathrm{T}} \operatorname{abs}(W)\, \widehat{\mathbf{1}} = -\sum\limits_{k,i} |w_{ki}|$.*
3. *Associated with the network is a number $\varepsilon > 0$ such that any change in the energy $\Delta E = E_{(t+1)} - E_{(t)}$ is of magnitude at least $-\varepsilon$.*

Proof. **1.** Consider that just one component, the k–th, of vector $\mathbf{y}$, changes from t to $t+1$, i.e. $y_{k(t+1)} \neq y_{k(t)}$. Then from equation (4.5):

$$\begin{aligned}
\Delta E &= E_{(t+1)} - E_t = \\
&= \left(-y_{k(t+1)} W_{(k,:)} \mathbf{x} - \sum_{\ell=1, \ell \neq k}^{K} y_\ell W_{(\ell,:)} \mathbf{x} \right) - \left(-y_{k(t)} W_{(k,:)} \mathbf{x} - \sum_{\ell=1, \ell \neq k}^{K} y_\ell W_{(\ell,:)} \mathbf{x} \right) \\
&= [y_{k(t)} - y_{k(t+1)}] W_{(k,:)} \mathbf{x} = -\Delta y_k \, W_{(k,:)} \mathbf{x}
\end{aligned}$$

According to the updating procedure (see Section 4.2.2) and as it was assumed that y_k does change, then there are two cases:

- $y_{k(t)} = +1 \to y_{k(t+1)} = -1$ (or 0 for binary vectors); then $\Delta y_k < 0$ and $W_{(k,:)}\mathbf{x} < 0$ (according to the algorithm) so $\Delta E < 0$.
- $y_{k(t)} = -1$ (or 0 for binary vectors) $\to y_{k(t+1)} = +1$; and an analogous argument shows: $\Delta E < 0$.

If several y_k terms do change then: $\Delta E = E_{(t+1)} - E_{(t)} = -\sum\limits_{\text{changed } y_k} \Delta y_k W_{(k,:)} \mathbf{x} < 0$, which represents a sum of negative terms.

The same discussion holds for a change in $\mathbf{x}$.

2. The $\{y_k\}_{k \in \overline{1,K}}$ and $\{x_i\}_{i \in \overline{1,N}}$ have all values in $\{-1, +1\}$ (or $\{0, 1\}$ for binary vectors).

The lowest possible value for E is obtained when all the terms of the form $y_k w_{ki} x_i$ in the sum (4.5) are positive (and then $|y_k| = 1$ and $|x_i| = 1$):

$$E_{\text{lower}} = -\sum_{k,i} |y_k w_{ki} x_i| = -\sum_{k,i} |w_{ki}| = -\widehat{\mathbf{1}}^{\mathrm{T}} \operatorname{abs}(W)\, \widehat{\mathbf{1}}$$

3. By part **1** the energy function decreases, it doesn't increase, so $\Delta E < 0$. On the other hand the energy function is bounded below (according to the second part of the theorem) so it cannot decrease by an infinite amount: $\Delta E \neq -\infty$.

❖ U, $\widetilde{\mathbf{x}}$, $\widetilde{\mathbf{x}}$

Call the finite set of all possible pairs $(\mathbf{x}, \mathbf{y})$, U. Then the set of all possible steps in the energy :

$$\left\{ -\mathbf{y}^{\mathrm{T}} \mathbf{x} + \widetilde{\mathbf{y}}^{\mathrm{T}} \widetilde{\mathbf{x}} \ : \ (\mathbf{x}, \mathbf{y}) \text{ and } (\widetilde{\mathbf{x}}, \widetilde{\mathbf{y}}) \in U \right\}$$

is also finite. Hence there is a minimum distance between different energy levels. That is there is an $\varepsilon > 0$ so that if energy goes down it changes by at least $-\varepsilon$. □

[4.2.3] See [FS92] pp. 136–141 and [Kos92] pp. 73–79.

[2] This is the Liapunov function for BAM. All state changes, with respect to time ($\mathbf{x} = \mathbf{x}_{(t)}$ and $\mathbf{y} = \mathbf{y}_{(t)}$) involve a *decrease* in the value of the function.

The following two conclusions are to be drawn:

- there was no assumption about how W was built, the procedure described in Section 4.2.2 will work with any matrix;
- the procedure will work in both synchronous and asynchronous modes, i.e. any neuron may change its state (output) at any time.

Proposition 4.2.1. *If the input pattern* $\mathbf{x}_q$ *is exactly one of the* $\{\mathbf{x}_p\}$ *stored in* W*, as built from* (4.2)*, then the corresponding* $\mathbf{y}_q$ *is retrieved in one step.*

Proof. According to the procedure, the vector $\mathbf{y}_{(0)}$ is initialized with $\operatorname{sign}(W\mathbf{x}_q)$ But as $\mathbf{x}_p^{\mathrm{T}}\mathbf{x}_q = \delta_{pq}$ (orthogonality) then ($\omega_p > 0$):

$$\mathbf{y}_{(0)} = \operatorname{sign}(W\mathbf{x}_q) = \operatorname{sign}\left(\sum_{p=1}^{P} \omega_p \mathbf{y}_p \mathbf{x}_p^{\mathrm{T}} \mathbf{x}_q\right) = \operatorname{sign}\left(\sum_{p=1}^{P} \omega_p \mathbf{y}_p \delta_{pq}\right) = \operatorname{sign}(\mathbf{y}_q) = \mathbf{y}_q$$

(the last equality holding because $\mathbf{y}_q$ is bipolar or binary). □

Conclusions:

- The existence of the BAM energy function with the outlined properties ensures that the running process is *convergent* and for any input vector a solution is reached in *finite time*.
- The BAM procedure may be compared to a database retrieval; the time required for retrieval being $\mathcal{O}(1)$.

Remarks:

➥ Suppose that an input vector $\mathbf{x}$ which is slightly different from one of the stored vectors $\mathbf{x}_p$, (i.e. there is noise in data) is presented to the network. Then, after stabilization, the network returns a pair $\{\widetilde{\mathbf{x}}_p, \widetilde{\mathbf{y}}_p\}$. This pair may not be the expected $\{\mathbf{x}_p, \mathbf{y}_p\}$. Results may vary depending upon the amount of noise and the degree of memory saturation.

4.3 BAM Algorithm

Network initialization

The weight matrix is calculated directly[3] from the set to be stored: $W = \sum_{p=1}^{P} \omega_p \mathbf{y}_p \mathbf{x}_p^{\mathrm{T}}$.

Note that there is no learning process. Weights are directly initialized with their final values, but it is possible to add new patterns at any time. If a memory with fading process is sought then ω_p should be chosen such that $\omega_1 < \cdots < \omega_P$.

Network running forward

The network runs in discrete–time approximation. Given $\mathbf{x}_{(0)}$, calculate $\mathbf{y}_{(0)} = \operatorname{sign}(W\mathbf{x}_{(0)})$.

Propagate: Apply assignments (4.3a) and (4.3b) repeatedly. The network stabilizes when $\mathbf{x}_{(t+1)} = \mathbf{x}_{(t)}$ and $\mathbf{y}_{(t+1)} = \mathbf{y}_{(t)}$ (and at this stage $\{\mathbf{x}_{(t)}, \mathbf{y}_{(t)}\} = \{\mathbf{x}_p, \mathbf{y}_p\}$, for some p, with high probability).

[3]Note that this represents a sum of rank 1 matrices. On digital simulations: calculate first $\omega_p \mathbf{y}_p$ and then multiply by $\mathbf{x}_p$ to reduce the number of multiplications; if $N < K$ then calculate first $\omega_p \mathbf{x}_p^{\mathrm{T}}$.

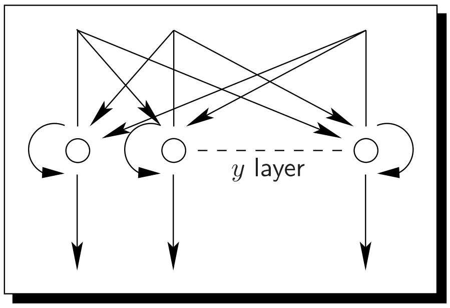

Figure 4.2: *The autoassociative memory architecture (compare to Figure 4.1 on page 52).*

Network running backwards

In the same discrete–time approximation, given $\mathbf{y}_{(0)}$, calculate $\mathbf{x}_{(0)} = \mathrm{sign}(W^{\mathrm{T}}\mathbf{y}_{(0)})$. Then propagate using equations (4.4) repeatedly. The network stabilizes when $\mathbf{x}_{(t+1)} = \mathbf{x}_{(t)}$ and $\mathbf{y}_{(t+1)} = \mathbf{y}_{(t)}$ (and at this stage, with high probability, $\{\mathbf{x}_{(t)}, \mathbf{y}_{(t)}\} = \{\mathbf{x}_p, \mathbf{y}_p\}$, for some p).

4.4 Hopfield Memory

4.4.1 Discrete Memory

❖ $\widetilde{\mathbf{y}}_p$

Consider an autoassociative memory. The weight matrix, built from a set of bipolar vectors $\{\widetilde{\mathbf{y}}_p\}$ is:

$$W = \sum_{p=1}^{P} \omega_p \widetilde{\mathbf{y}}_p \widetilde{\mathbf{y}}_p^{\mathrm{T}} \quad , \quad \omega_p > 0 \ , \ \omega_p = \text{const.}$$

which is *square* $(K \times K)$ and *symmetric* $(W = W^{\mathrm{T}})$.

An autoassociative memory is similar to a BAM with the remark that the 2 layers (x and y) are identical. In this case the 2 layers may be replaced with one fully interconnected layer, including a feedback for each neuron — see Figure 4.2: the output of each neuron is connected to the inputs of all neurons, including itself.

❖ $\mathbf{x}$

The discrete Hopfield memory is build from the autoassociative memory described above by replacing the autofeedback (feedback from a neuron to itself) by an external input signal $\mathbf{x}$ — see Figure 4.3 on the facing page.

The differences from autoassociative memory or BAM are as follows:

① The discrete Hopfield memory works with *binary* vectors rather than bipolar ones (see also Section 4.2.2), here and below the $\mathbf{y}$ *vectors are considered binary* and so are the input $\mathbf{x}$ vectors; note that dimension of $\mathbf{x}$ and $\mathbf{y}$ is the same K.

② The weight matrix is obtained from: $\sum\limits_{p=1}^{P} (2\mathbf{y}_p - \widehat{\mathbf{1}})(2\mathbf{y}_p - \widehat{\mathbf{1}})^{\mathrm{T}}$ by replacing the diagonal values with 0, i.e. auto–feedback is removed;it will be replaced by input $\mathbf{x}$, one

4.4.1 See [FS92] pp. 141–144, [Has95] pg. 396.

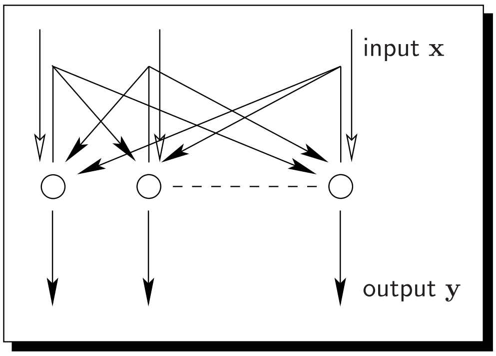

Figure 4.3: *The discrete Hopfield memory architecture.*

component per neuron.

③ A Hopfield network uses the same threshold activation function and passive decay with resting potential as BAM.

The algorithm is similar with the BAM one but the updating formula is: Hopfield algorithm

$$y_{k(t+1)} = \begin{cases} +1 & \text{if } W_{(k,:)}\mathbf{y}+x_k>t_k \\ y_{k(t)} & \text{if } W_{(k,:)}\mathbf{y}+x_k=t_k \\ 0 & \text{if } W_{(k,:)}\mathbf{y}+x_k<t_k \end{cases} \tag{4.6}$$

where $\mathbf{t}$ is called the threshold vector and is a fixed vector with positive components (note that $W_{(k,:)}\mathbf{y} = \sum\limits_{\ell \neq k} w_{k\ell} y_\ell$, because $w_{kk} = 0$ by construction). $\mathbf{x}$ plays the role of a bias vector. The $\mathbf{t}$ vector is kept constant for all network runs while $\mathbf{x}$ may change from one run to the next. ❖ $\mathbf{t}$

In matrix notation equation (4.6) becomes:

$$\mathbf{y}_{(t+1)} = \mathcal{H}\{W\mathbf{y}_{(t)} + \mathbf{x} - \mathbf{t}, \mathbf{y}_{(t)}, = +1, =, = 0\}$$

Remarks:

➥ Note that Hopfield network may also work with bipolar vectors the same way BAM may work with binary vectors; to do so replace the "$= 0$" part with "$= -1$" in the formula above.

➥ Another matrix formulation would be:

$$A_{(t)} = \operatorname{sign}(W\mathbf{y}_{(t)} + \mathbf{x} - \mathbf{t})$$

$$\mathbf{y}_{(t+1)} = \widehat{\mathbf{1}} + \operatorname{sign}(A_{(t)} - \widehat{\mathbf{1}}) + [\widehat{\mathbf{1}} - \operatorname{abs}(A_{(t)})] \odot \mathbf{y}_{(t)}$$

Proof. The elements of $A_{(t)}$ are from $\{+1, 0, -1\}$ so consider these three cases:

a. $\widehat{\mathbf{1}} + \operatorname{sign}(A_{(t)} - \widehat{\mathbf{1}})$ returns $\{1, 0, 0\}$ thus taking care of first and last cases;

b. $[\widehat{\mathbf{1}} - \operatorname{abs}(A_{(t)})]$ returns $\{0, 1, 0\}$ so the corresponding term inserts $y_{k(t)}$ in the required places. □

❖ E
Hopfield energy

Definition 4.4.1. *The discrete Hopfield memory energy function is defined as:*

$$E = -\frac{1}{2}\,\mathbf{y}^{\mathrm{T}} W \mathbf{y} - \mathbf{y}^{\mathrm{T}}(\mathbf{x} - \mathbf{t}) \tag{4.7}$$

Remarks:

- Comparing to the BAM energy the discrete Hopfield energy has a factor of $1/2$ because there is just one layer of neurons (in BAM both forward and backward passes contribute to the energy function).
- The $\mathbf{x} - \mathbf{t}$ term helps define the constraints Hopfield memory has to meet.

Theorem 4.4.1. *The discrete Hopfield energy function has the following properties:*

1. *Any change in $\mathbf{y}$ (during a network run) results in an energy decrease: $E_{(t+1)}(\mathbf{y}_{(t+1)}) < E_{(t)}(\mathbf{y}_{(t)})$.*
2. *E is bounded below by: $E_{\mathit{lower}} = -\frac{1}{2}\,\widehat{\mathbf{1}}^{\mathrm{T}}\,\mathrm{abs}(W)\,\widehat{\mathbf{1}} - K = -\frac{1}{2}\sum\limits_{k,\ell}|w_{k\ell}| - K$.*
3. *Associated with the network is a number $\varepsilon > 0$ such that any change in the energy $\Delta E = E_{(t+1)} - E_{(t)}$ is of magnitude at least $-\varepsilon$.*

Proof. The proof is similar to proof of Theorem 4.2.1.

1. Consider that, from t to $t+1$, just one component of vector y_k changes (both $\mathbf{x}$ and $\mathbf{t}$ are constants during a network run, $\mathbf{x}$ changes only between runs). Then from (4.7):

$$\begin{aligned}\Delta E = E_{(t+1)} - E_{(t)} &= 2[y_{k(t+1)} - y_{k(t)}]\left(-\frac{1}{2}\sum_{\ell=1}^{K} w_{k\ell}y_\ell\right) - [y_{k(t+1)} - y_{k(t)}](x_k - t_k) \\ &= [y_{k(t+1)} - y_{k(t)}](-W_{(k,:)}\mathbf{y} - x_k + t_k) = -\Delta y_k(W_{(k,:)}\mathbf{y} + x_k - t_k)\end{aligned}$$

because in the sum $\sum\limits_{k,\ell=1}^{K} y_k w_{k\ell} y_\ell$, y_k appears twice: once at the left and once at the right and $w_{k\ell} = w_{\ell k}$ (also $w_{kk} = 0$, by construction).

According to the updating procedure (4.6) and as it was assumed that y_k does change, then there are 2 cases:

- $y_k(t) = +1 \to y_k(t+1) = 0$ (or -1 for bipolar vectors); then $\Delta y_k < 0$ and $W_{(k,:)}\mathbf{y} + x_k - t_k < 0$ (according to the algorithm) so $\Delta E < 0$.
- $y_{k(t)} = 0$ (or -1 for bipolar vectors) $\to y_{k(t+1)} = +1$; analogous preceding case: $\Delta E < 0$.

If more than one term changes then $\Delta E = -\sum\limits_{\text{changed } y_k} \Delta y_k(W_{(k,:)}\mathbf{y} + x_k - t_k) < 0$, which represents a sum of negative terms.

2. The lowest possible value for E is obtained when $\mathbf{y} = \mathbf{x} = \widehat{\mathbf{1}}$, $\mathbf{t} = \widehat{\mathbf{0}}$ and all weights are positive (similar discussion for bipolar vectors but there weights may be also negative). Then negative terms are maximum and the positive term is minimum (see (4.7)):

$$E_{\text{lower}} = -\frac{1}{2}\sum_{k,\ell}|w_{k\ell}| - K = -\frac{1}{2}\,\widehat{\mathbf{1}}^{\mathrm{T}}\,\mathrm{abs}(W)\,\widehat{\mathbf{1}} - K$$

3. By part **1** the energy function decreases, it doesn't increase, so $\Delta E < 0$. On the other hand the energy function is bounded below (according to the second part of the theorem) so it cannot decrease by an infinite amount: $\Delta E \neq -\infty$.

❖ $U, \widetilde{\mathbf{x}}, \widetilde{\mathbf{x}}$

Call the finite set of all possible pairs $(\mathbf{x}, \mathbf{y})$, U. Then the set of all possible steps in the energy :

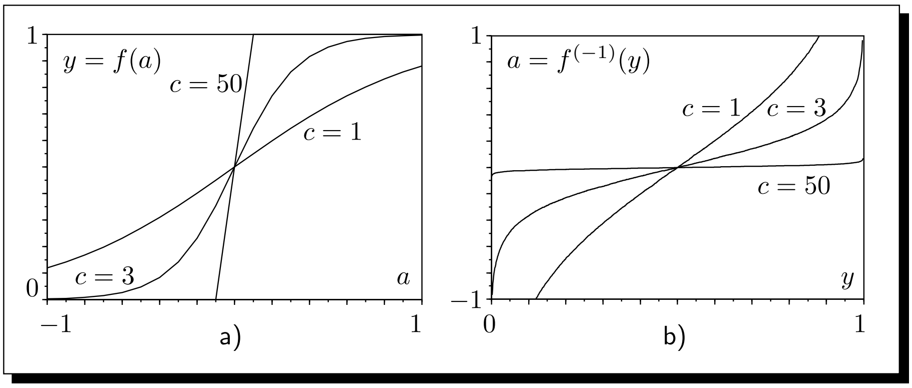

Figure 4.4: *a) The neuron activation function for continuous Hopfield memory (for various c values); b) Inverse of activation.*

$$\left\{ -\mathbf{y}^{\mathrm{T}}\mathbf{x} + \widetilde{\mathbf{y}}^{\mathrm{T}}\widetilde{\mathbf{x}} \ : \ (\mathbf{x}, \mathbf{y}) \text{ and } (\widetilde{\mathbf{x}}, \widetilde{\mathbf{y}}) \in U \right\}$$

is also finite. Hence there is a minimum distance between different energy levels. That is there is an $\varepsilon > 0$ so that if energy goes down it changes by at least $-\varepsilon$. □

This theorem has the following consequences:

- the existence of discrete Hopfield energy function with the outlined properties ensures that the running process is *convergent* and a solution is reached in *finite* time;
- analogous of these results hold when W is replaced by any symmetric matrix with zeros on the diagonal;
- the discrete Hopfield memory works in both synchronous and asynchronous modes, any neuron may change its state at any time.

4.4.2 Continuous Memory

The continuous Hopfield memory model is similar to the discrete one except for the activation function of the neuron which is of the form:

$$y = f(a) = \frac{1 + \tanh(ca)}{2} \quad , \quad c = \text{const.}\ , \ c > 0 \tag{4.8}$$

where c is called the *gain parameter*. See Figure 4.4–a. The inverse of activation is (see Figure 4.4–b): $a = f^{(-1)}(y) = \frac{1}{2c} \ln \frac{y}{1-y}$ (use $\tanh(ca) = \frac{e^{ca} - e^{-ca}}{e^{ca} + e^{-ca}}$ in the above formula). ❖ c gain parameter

The differential equation governing the evolution of continuous Hopfield memory is passive decay with resting potential, of the form: ❖ λ

$$\lambda \frac{da_k}{dt} = -t_k a_k + \sum_{\substack{\ell=1 \\ \ell \neq k}}^{K} w_{k\ell} y_\ell + x_k \quad , \quad \lambda = \text{const.}\ , \ \lambda > 0 \tag{4.9}$$

[4.4.2] See [FS92] pp. 144–148.

or in matrix notation (note that W has zeroes on its main diagonal, as in the discrete Hopfield case, i.e. $w_{kk} = 0$):

$$\lambda \frac{d\mathbf{a}}{dt} = -\mathbf{t} \odot \mathbf{a} + W\mathbf{y} + \mathbf{x} \tag{4.10}$$

In discrete–time approximation the updating procedure may be written as:

$$\mathbf{a}_{(t+1)} = \left(\widehat{\mathbf{1}} - \frac{1}{\lambda}\,\mathbf{t}\right) \odot \mathbf{a}_{(t)} + \frac{1}{\lambda}\left(W\mathbf{y}_{(t)} + \mathbf{x}\right) = \left(1 \overset{\mathrm{R}}{\ominus} \frac{1}{\lambda}\,\mathbf{t}\right) \odot \mathbf{a}_{(t)} + \frac{1}{\lambda}\left(W\mathbf{y}_{(t)} + \mathbf{x}\right) \tag{4.11a}$$

$$\mathbf{y}_{(t+1)} = \frac{\widehat{\mathbf{1}} + \tanh(c\mathbf{a}_{(t+1)})}{2} = \frac{1 \overset{\mathrm{R}}{\oplus} \tanh(c\mathbf{a}_{(t+1)})}{2} \tag{4.11b}$$

Proof. $d\mathbf{a} \to \Delta\mathbf{a} = \mathbf{a}_{(t+1)} - \mathbf{a}_{(t)}$ and $dt \to \Delta t = 1$; also $\mathbf{a} = \widehat{\mathbf{1}} \odot \mathbf{a}$. From (4.10):

$$\Delta\mathbf{a} = -\frac{1}{\lambda}\,\mathbf{t} \odot \mathbf{a}_{(t)} + \frac{1}{\lambda}\left(W\mathbf{y}_{(t)} + \mathbf{x}\right)$$

For second formula use (4.8) in vectorial form. □

❖ E, $\boldsymbol{\zeta}$
Hopfield energy

Definition 4.4.2. *The continuous Hopfield memory energy function is defined as:*

$$E = -\frac{1}{2}\,\mathbf{y}^{\mathrm{T}} W \mathbf{y} - \mathbf{y}^{\mathrm{T}}\mathbf{x} + \frac{1}{2c}\,\boldsymbol{\zeta}^{\mathrm{T}}\mathbf{t} \quad \textit{where} \quad \zeta_k = \int_0^{y_k} f^{(-1)}(\widetilde{y})\, d\widetilde{y}\ ,\ k \in \{1, \dots, K\} \tag{4.12}$$

Theorem 4.4.2. *The continuous Hopfield energy function has the following properties:*

1. *Any change in $\mathbf{y}$ (during a network run) result in an energy decrease: $\frac{dE}{dt} \leqslant 0$.*
2. *E is bounded below by: $E_{\textit{lower}} = -\frac{1}{2}\,\widehat{\mathbf{1}}^{\mathrm{T}} \operatorname{abs}(W)\,\widehat{\mathbf{1}} - K = -\frac{1}{2}\sum\limits_{k,\ell}^{K} |w_{k\ell}| - K$.*

Proof. **1.** First: $\int \ln x \, dx = x \ln x - x \Rightarrow \int \ln \frac{x}{1-x}\, dx = \ln x^x (1-x)^{1-x} + \text{const.}$ and $\lim\limits_{x \searrow 0} \ln x^x = \lim\limits_{x\searrow 0} \frac{\ln x}{\frac{1}{x}} \overset{\text{(L'Hospital)}}{=} \lim\limits_{x \searrow 0} -x = 0$. Then:

$$\zeta_k = \int_0^{y_k} \ln \frac{\widetilde{y}}{1-\widetilde{y}}\, d\widetilde{y} = \lim_{y_0 \searrow 0} \int_{y_0}^{y_k} \ln \frac{\widetilde{y}}{1-\widetilde{y}}\, d\widetilde{y} = \ln y_k^{y_k}(1-y_k)^{1-y_k} - \lim_{y_0 \searrow 0} \ln y_0^{y_0}(1-y_0)^{1-y_0}$$
$$= \ln y_k^{y_k}(1-y_k)^{1-y_k}$$
$$\frac{d\zeta_k}{dt} = \frac{d}{dy_k}\left[\ln y_k^{y_k}(1-y_k)^{1-y_k}\right]\frac{dy_k}{dt} = \left[\ln y_k - \ln(1-y_k)\right]\frac{dy_k}{dt} = 2cf^{(-1)}(y_k)\,\frac{dy_k}{dt}$$

then, from (4.12) and using (4.9) (also $\frac{da_k}{dt} = \frac{da_k}{dy_k}\frac{dy_k}{dt}$):

$$\frac{dE}{dt} = -\frac{d\mathbf{y}^{\mathrm{T}}}{dt}\left[W\mathbf{y} + \mathbf{x} - f^{(-1)}(\mathbf{y}) \odot \mathbf{t}\right] = -\lambda\,\frac{d\mathbf{y}^{\mathrm{T}}}{dt}\frac{d\mathbf{a}}{dt} = -\lambda \sum_{k=1}^{K} \frac{df^{(-1)}(y_k)}{dy_k}\left(\frac{dy_k}{dt}\right)^2 < 0$$

(because $\frac{df^{(-1)}(y_k)}{dy_k} = \frac{2c}{y_k(1-y_k)} > 0$ as $y_k \in (0,1)$), i.e. $\frac{dE}{dt}$ is always negative and E decreases in time.

2. The lowest possible value for E is obtained when $\mathbf{y} = \mathbf{x} = \widehat{\mathbf{1}}$, $\mathbf{t} = \widehat{\mathbf{0}}$ and all weights are positive. A similar discussion applies for bipolar vectors but there weights may be also negative). Then negative terms are maximum and the positive term is minimum (see (4.12)), assuming that all $w_{k\ell} > 0$:

$$E_{\text{lower}} = -\frac{1}{2}\sum_{k,\ell} |w_{k\ell}| - K = -\frac{1}{2}\,\widehat{\mathbf{1}}^{\mathrm{T}} \operatorname{abs}(W)\,\widehat{\mathbf{1}} - K \quad \square$$

Remarks:

- The existence of continuous Hopfield energy function with the outlined properties ensures that the running process is *convergent*. However there is no guarantee that this will happen in a finite amount of time.
- For $c \to +\infty$, the continuous Hopfield becomes identical to the discrete one and stable states are represented by binary vectors representing the corners of a hypercube (with side length "1"). Otherwise:
  - For $c \to 0$ there is only one stable state: $\mathbf{y} = \frac{1}{2}\widehat{\mathbf{1}}$ (because, from the activation function definition, in this case $y_k = 1/2$, $\forall k$, $\forall a_k$).
  - For $c \in (0, +\infty)$ the stable states are somewhere between the corners of the hypercube (having its center at $\frac{1}{2}\widehat{\mathbf{1}}$) and its center (as gain decreases from $+\infty$ to 0 the stable points moves from corners towards the center).

4.5 Examples

4.5.1 Thesaurus

This example shows the associative memory capabilities of BAM and the crosstalk occurrence.

A list of English words and one of their synonyms is required to be memorized in pairs such that when a word is given its counterpart may be retrieved:

```
constant    invariable
emotional   heartfelt
hereditary  ancestral
intolerant  dogmatic
miniature   diminutive
regretful   apologetic
squeamish   fastidious
```

The letters of the alphabet have been binary encoded on 5 bits each:

$$a \leftrightarrow \begin{pmatrix} 0 & 0 & 0 & 0 & 0 \end{pmatrix}^{\mathrm{T}}, \; b \leftrightarrow \begin{pmatrix} 0 & 0 & 0 & 0 & 1 \end{pmatrix}^{\mathrm{T}}, \text{ e.t.c.}$$

The longest word has 10 letters; all words were brought to the same 10 characters length, by padding with blanks (spaces were added at the end), encoded as $\begin{pmatrix} 1 & 1 & 1 & 1 & 0 \end{pmatrix}^{\mathrm{T}}$. This procedure encoded each word in 50 bit binary vectors which were transformed finally to bipolar vectors (by the procedure already outlined: $\mathbf{x}_{\text{bipolar}} = 2\mathbf{x}_{\text{binary}} - \widehat{\mathbf{1}}$). The left column of words were transformed to vectors $\{\mathbf{x}_p\}$ while the right column was transformed to $\{\mathbf{y}_p\}$. Finally the vectors were memorized with equal precedence ($\omega_p = 1$) and weight matrix was built as $W = \sum_{p=1}^{7} \mathbf{y}_p \mathbf{x}_p^{\mathrm{T}}$.

Testing the memory[4] in both direct and reverse run gave the following results:

[4]The actual Scilab code is in the source tree of this book, e.g. `Matrix_ANN_1/Scilab/BAM_thesaurus.sci`, note that actual version numbers may differ.

direct run		reverse run	
`constant`	`invariable`	`constant`	`invariable`
`emotional`	`heartfelt`	`emotional`	`heartfelt`
`hereditary`	`ancestral`	`hergdiraz_`	`ancestaal`
`intolerant`	`dogmatic`	`amtglmran_`	`dmceatic`
`miniature`	`diminutive`	`miniature`	`diminutive`
`regretful`	`apologetic`	`regretful`	`apologetic`
`squeamish`	`fastidious`	`squeamish`	`fastidious`

(the "_" character was encoded as binary $\begin{pmatrix} 1 & 1 & 0 & 1 & 0 \end{pmatrix}^{\mathrm{T}}$).

Note that crosstalk has occurred between $\{\mathbf{x}_3, \mathbf{y}_3\}$ and $\{\mathbf{x}_4, \mathbf{y}_4\}$ (between {`hereditary`, `ancestral`} and {`intolerant`, `dogmatic`}). Calculating the Hamming distance [5] among $\{\mathbf{x}_p\}$ and $\{\mathbf{y}_p\}$ gives:

	$\mathbf{x}_1$	$\mathbf{x}_2$	$\mathbf{x}_3$	$\mathbf{x}_4$	$\mathbf{x}_5$	$\mathbf{x}_6$	$\mathbf{x}_7$
$\mathbf{x}_1$	0	19	25	26	21	24	22
$\mathbf{x}_2$	19	0	26	23	22	15	21
$\mathbf{x}_3$	25	26	0	19	26	27	21
$\mathbf{x}_4$	26	23	19	0	25	30	26
$\mathbf{x}_5$	21	22	26	25	0	21	23
$\mathbf{x}_6$	24	15	27	30	21	0	22
$\mathbf{x}_7$	22	21	21	26	23	22	0

	$\mathbf{y}_1$	$\mathbf{y}_2$	$\mathbf{y}_3$	$\mathbf{y}_4$	$\mathbf{y}_5$	$\mathbf{y}_6$	$\mathbf{y}_7$
$\mathbf{y}_1$	0	23	18	25	24	24	31
$\mathbf{y}_2$	23	0	21	22	25	27	20
$\mathbf{y}_3$	18	21	0	15	26	24	27
$\mathbf{y}_4$	25	22	15	0	23	23	20
$\mathbf{y}_5$	24	25	26	23	0	26	27
$\mathbf{y}_6$	24	27	24	23	26	0	25
$\mathbf{y}_7$	31	20	27	20	27	25	0

and shows that $\{\mathbf{x}_3, \mathbf{y}_3\}$ and $\{\mathbf{x}_4, \mathbf{y}_4\}$ are the closest related pair ($\{\mathbf{x}_2, \mathbf{x}_6\}$ is also a closely related pair but for the corresponding $\{\mathbf{y}_2, \mathbf{y}_6\}$ the Hamming distance is greater).

4.5.2 Traveling Salesperson Problem

This example shows a practical problem of scheduling, e.g. as it arises in communication networks. A bidimensional Hopfield continuous memory is being used. It is also a classical example of an NP–problem but solved with the help of an ANN.

The problem: A traveling salesperson must visit a number of cities, each only once. Moving from one city to another has a cost e.g. the intercity distance associated. The cost/distance traveled must be minimized. The salesperson has to return to the starting point (close the loop).

The traveling salesperson problem is of NP (non-polynomial) type.

Proof. Assuming that there are K cities there will be $K!$ possible paths. For a given tour (closed loop) it doesn't matter which city is first (one division by K) nor does the direction matter (one division by 2). So the number of different of paths is $(K-1)!/2$ ($K \geqslant 3$ otherwise the problem is trivial).

Adding a new city to the previous set means that now there are $K!/2$ routes. That means an increase in the number of paths by a factor of:

$$\frac{K!/2}{(K-1)!/2} = K$$

so, for an arithmetic progression growth of problem size, the space of possible solutions grows *exponentially*. □

❖ C_k Let $C_1, \ldots, C_K$ be the cities involved. Associated with each of the K cities is a binary vector representing the order of visiting in the current tour, as follows: the first city visited is associated with $\begin{pmatrix} 1 & 0 & \ldots & 0 \end{pmatrix}^{\mathrm{T}}$, the second one with $\begin{pmatrix} 0 & 1 & 0 & \ldots & 0 \end{pmatrix}^{\mathrm{T}}$ and so on,

[5] Recall that the Hamming distance between two binary vectors is the number of different bits.
[4.5.2] See [FS92] pp. 148–156 and [FMTG93].

the last one to be visited being associated with the vector: $\begin{pmatrix}0 & \dots & 0 & 1\end{pmatrix}^{\mathrm{T}}$, i.e. each vector has one element "1", all others being "0".

A square matrix T may be built using the previously defined vectors as rows. Because the cities aren't necessarily visited in their listed order the matrix is not necessarily diagonal: ❖ T

$$T = \begin{array}{c c}
\begin{matrix} 1 & \cdots & i & \cdots & j & \cdots & K \end{matrix} & \\
\begin{pmatrix}
1 & \cdots & 0 & \cdots & 0 & \cdots & 0 \\
\vdots & & & & & & \vdots \\
0 & \cdots & 0 & \cdots & 1 & \cdots & 0 \\
\vdots & & & & & & \vdots \\
0 & \cdots & 1 & \cdots & 0 & \cdots & 0 \\
\vdots & & & & & & \vdots \\
0 & \cdots & 0 & \cdots & 0 & \cdots & 1
\end{pmatrix} &
\begin{matrix} C_1 \\ \vdots \\ C_\ell \\ \vdots \\ C_k \\ \vdots \\ C_K \end{matrix}
\end{array} \tag{4.13}$$

This matrix defines the tour: for each column $1 \leqslant i \leqslant K$ (step in tour) pick as city to be visited at the i–th step the one with the corresponding element equal to 1 (follow the row to the right). Note that T is a permutation matrix.

The idea is to build a bidimensional Hopfield memory such that its output is a matrix Y (not a vector) whose entries are to be interpreted in the same way as those of T and this will give the solution, as the position of the "1" in each row will give the visiting order number of the respective city.

In order to be an acceptable solution, the Y matrix should have the following properties:

① Each city must not be visited more that once ⇔ Each *row* of the matrix (4.13) should have no more that one "1", all others elements should be "0".

② Two cities cannot be visited at the same time (can't have the same order number) ⇔ Each *column* of the matrix (4.13) should have no more than one "1" all others elements should be "0".

③ All cities should be visited ⇔ Each *row or column* of the matrix (4.13) should have at least one "1".

④ The total distance/cost of the tour should be minimised. Let $d_{k\ell}$ be the distance/cost between cities C_k and C_ℓ, obviously $d_{kk} \equiv 0$. ❖ $d_{k\ell}$

Note that items ①, ② and ③ define the permutation matrix (as Y should be) and item ④ defines the problem's constraints.

As the network is bidimensional each index of the theoretical results established in previous sections will be split in two, this way all results will become immediately applicable. Each weight has 4 subscripts: $w_{\ell jki}$ is the weight *from* neuron at $\{\text{row } k, \text{column } i\}$ *to* neuron at $\{\text{row } \ell, \text{column } j\}$. See Figure 4.5 on the next page. Then the weight matrix may be written as a block matrix: ❖ $w_{\ell jki}$, $\widehat{W}$, $W_{\ell k}$

$$\widehat{W} = \begin{pmatrix} W_{11} & \cdots & W_{1K} \\ \vdots & \ddots & \vdots \\ W_{K1} & \cdots & W_{KK} \end{pmatrix} = \begin{pmatrix} w_{1111} & \cdots & w_{11KK} \\ \vdots & \ddots & \vdots \\ w_{11KK} & \cdots & w_{KKKK} \end{pmatrix} \text{ where } W_{\ell k} = \begin{pmatrix} w_{\ell 1k1} & \cdots & w_{\ell 1kK} \\ \vdots & \ddots & \vdots \\ w_{\ell Kk1} & \cdots & w_{\ell KkK} \end{pmatrix}$$

Note that each row of $\widehat{W}$ holds the weights associated with a particular neuron. The Y matrix is converted to/from a vector using the lexicographic convention (graphically: each row of Y is transposed and "glued" at the bottom of previous one):

$$Y \to \widehat{\mathbf{y}} = \begin{pmatrix} y_{11} & \cdots & y_{KK} \end{pmatrix}^{\mathrm{T}} \quad \Leftrightarrow \quad Y_{(k,:)} = \widehat{\mathbf{y}}^{\mathrm{T}}_{((k-1)K+1\,:\,kK)}$$

The weights cannot be built from a set of some $\{Y_p\}$ as these are not known — the idea is to *find* them — but they may be built considering the following reasoning:

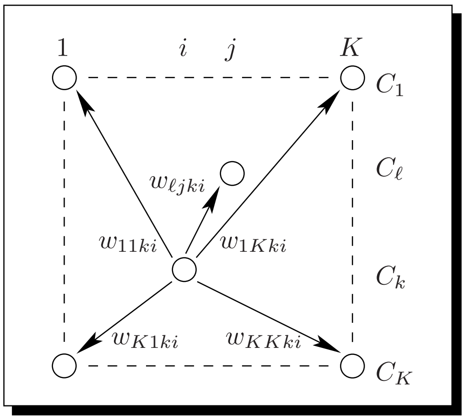

Figure 4.5: *The bidimensional Hopfield memory and its weights.*

① A city must appear only once in a tour: this means that one neuron on a row must inhibit all others on the same row such that in the end only one will have active output 1, all others will drop output to 0. Then the weight should have a term of the form:

❖ α, $w^{(1)}_{\ell jki}$

$$w^{(1)}_{\ell jki} = -\alpha \delta_{\ell k}(1 - \delta_{ji}) \; , \quad \alpha = \text{const.} \; , \; \alpha > 0$$

❖ $W^{(1)}_{\ell k}$, $\widehat{W}^{(1)}$

This means all $w^{(1)}_{\ell jki} = 0$ for neurons on different rows, $w^{(1)}_{\ell j \ell i} < 0$ for a given row ℓ if $j \neq i$ and $w^{(1)}_{\ell j \ell j} = 0$ for autofeedback. In matrix notation:

$$W^{(1)}_{\ell k} = -\alpha \delta_{\ell k}(\widetilde{1}_{KK} - I_K) \quad \Rightarrow \quad W^{(1)}_{\substack{\ell k \\ \ell \neq k}} = \widetilde{0}_{KK} \; , \; W^{(1)}_{kk} = \begin{pmatrix} 0 & -\alpha & \cdots & -\alpha \\ -\alpha & 0 & \cdots & -\alpha \\ \vdots & & & \vdots \\ -\alpha & \cdots & -\alpha & 0 \end{pmatrix}$$

$$\widehat{W}^{(1)} = -\alpha I_K \otimes (\widetilde{1}_{KK} - I_K)$$

Note that both $W^{(1)}_{\ell k}$ and $\widehat{W}^{(1)}$ are circulant matrices.

❖ β, $w^{(2)}_{\ell jki}$

② There must be no cities with the same order number in a tour: this means that one neuron on a column must inhibit all others on the same column such that in the end only one will have active output 1, all others will drop output to 0. Then the weight should have a term of the form:

$$w^{(2)}_{\ell jki} = -\beta(1 - \delta_{\ell k})\delta_{ji} \; , \quad \beta = \text{const.} \; , \; \beta > 0$$

❖ $W^{(2)}_{\ell k}$, $\widehat{W}^{(2)}$

This means all $w^{(2)}_{\ell jki} = 0$ for neurons on different columns, $w^{(2)}_{\ell jkj} < 0$ for a given column if $\ell \neq k$ and $w^{(2)}_{\ell j \ell j} = 0$ for autofeedback. In matrix notation:

$$W^{(2)}_{\ell k} = -\beta(1 - \delta_{\ell k}) I_K \quad \Rightarrow \quad W^{(1)}_{\substack{\ell k \\ \ell \neq k}} = \begin{pmatrix} -\beta & 0 & \cdots & 0 \\ 0 & & & \vdots \\ \vdots & & & 0 \\ 0 & \cdots & 0 & -\beta \end{pmatrix} \; , \; W^{(1)}_{kk} = \widetilde{0}_{KK}$$

$$\widehat{W}^{(2)} = -\beta(\widetilde{1}_{KK} - I_K) \otimes I_K$$

Note that both $W^{(2)}_{\ell k}$ and $\widehat{W}^{(2)}$ are circulant matrices.

③ Most of the neurons should have output 0: at end of network run only K out of K^2 neurons should have nonzero output; so a global inhibition may be used. Then the weight should have a term of the form:

❖ γ, $w^{(3)}_{\ell jki}$

$$w^{(3)}_{\ell jki} = -\gamma \ , \quad \gamma = \text{const.} \ , \ \gamma > 0$$

i.e. all neurons receive the same global inhibition $\propto \gamma$. In matrix notation:

❖ $W^{(3)}_{\ell k}$, $\widehat{W}^{(3)}$

$$W^{(3)}_{\ell k} = -\gamma \widetilde{1}_{KK} \ , \quad \widehat{W}^{(3)} = -\gamma \widetilde{1}_{K^2 K^2} = -\gamma \widetilde{1}_{KK} \otimes \widetilde{1}_{KK}$$

Obviously $W^{(3)}_{\ell k}$ and $\widehat{W}^{(3)}$ are circulant matrices.

④ The total distance/cost has to be minimized: neurons receive an inhibitory input proportional to the distance between cities represented by them. Only neurons on adjacent columns, representing cities which *may* come before or after the current city in the tour order (only one will actually be selected) should send this inhibition:

❖ η, $w^{(4)}_{\ell jki}$

$$w^{(4)}_{\ell jki} = -\eta d_{\ell k} (\delta_{\underset{K}{\mathrm{mod}}(j)+1,i} + \delta_{j,\underset{K}{\mathrm{mod}}(i)+1}) \ , \quad \eta = \text{const.} \ , \ \eta > 0$$

where the modulo function was used to take care of special cases: $\begin{cases} i=1 \text{ and } j=K \\ i=K \text{ and } j=1 \end{cases}$ (note that $K \geqslant 3$ so there are no further special cases). The term $\delta_{\underset{K}{\mathrm{mod}}(j)+1,i}$ takes care of the case when column j comes *before* i ($\delta_{\underset{K}{\mathrm{mod}}(j)+1,i} \neq 0$ for $i = j+1$, column K comes "before' column 1, inhibition from next city to visit) while $\delta_{j,\underset{K}{\mathrm{mod}}\, i+1}$ operate similar for the case when j comes *after* i ($\delta_{j,\underset{K}{\mathrm{mod}}(i)+1} \neq 0$ for $j = i+1$, column 1 come after column K, inhibition from previous city visited). In matrix notation:

❖ $W^{(4)}_{\ell k}$, $\widehat{W}^{(4)}$
❖ C, D

$$W^{(4)}_{\substack{\ell k \\ \ell \neq k}} = -\eta d_{\ell k} C \quad , \quad W^{(4)}_{kk} = \widetilde{0}_{KK} \quad \Rightarrow \quad \widehat{W}^{(4)} = -\eta D \otimes C$$

$$\text{where} \quad C = \begin{pmatrix} 0 & 1 & 0 & \cdots & 0 & 0 & 1 \\ 1 & 0 & 1 & & 0 & 0 & 0 \\ 0 & 1 & 0 & & 0 & 0 & 0 \\ \vdots & & & & & & \vdots \\ 0 & 0 & 0 & & 0 & 1 & 0 \\ 0 & 0 & 0 & & 1 & 0 & 1 \\ 1 & 0 & 0 & \cdots & 0 & 1 & 0 \end{pmatrix} \quad \text{and} \quad D = \begin{pmatrix} 0 & d_{12} & \cdots & d_{1K} \\ d_{21} & & & \vdots \\ \vdots & & & d_{K-1,K} \\ d_{K1} & \cdots & d_{K,K-1} & 0 \end{pmatrix}$$

Note that $W_{\ell k}$ and C are circulant matrices while D is symmetric and thus $\widehat{W}^{(4)}$ is a block circulant and symmetric matrix.

Combining all weights together gives $W_{\ell k} = W^{(1)}_{\ell k} + W^{(2)}_{\ell k} + W^{(3)}_{\ell k} + W^{(4)}_{\ell k}$ as a circulant matrix:

$$W_{\substack{\ell k \\ \ell \neq k}} = \begin{pmatrix} -\beta-\gamma & -\gamma-\eta d_{\ell k} & \cdots & -\gamma & -\gamma-\eta d_{\ell k} \\ -\gamma-\eta d_{\ell k} & -\beta-\gamma & & -\gamma & -\gamma \\ \vdots & & & & \vdots \\ -\gamma & -\gamma & & -\beta-\gamma & -\gamma-\eta d_{\ell k} \\ -\gamma-\eta d_{\ell k} & -\gamma & \cdots & -\gamma-\eta d_{\ell k} & -\beta-\gamma \end{pmatrix} , \ W_{kk} = \begin{pmatrix} -\gamma & -\gamma-\alpha & \cdots & -\gamma-\alpha \\ -\gamma-\alpha & & & \vdots \\ \vdots & & & -\gamma-\alpha \\ -\gamma-\alpha & \cdots & -\gamma-\alpha & -\gamma \end{pmatrix}$$

The weight matrix is obtained as $\widehat{W} = \widehat{W}^{(1)} + \widehat{W}^{(2)} + \widehat{W}^{(3)} + \widehat{W}^{(4)}$ and is a symmetric block circulant matrix ($\widehat{W}^{(4)}$ not fully circulant because of $d_{\ell k}$):

$$\widehat{W} = -\alpha I_K \otimes (\widetilde{1}_{KK} - I_K) - \beta(\widetilde{1}_{KK} - I_K) \otimes I_K - \gamma \widetilde{1}_{K^2 K^2} - \eta D \otimes C$$

❖ $\mathbf{x}$, $\mathbf{t}$

By taking $\mathbf{x} = \gamma K \widehat{\mathbf{1}}_{K^2}$(this particular value is explained in next section), $\mathbf{t} = \widehat{\mathbf{0}}_{K^2}$, gain parameter $c = 1$ and $\lambda = 1$, the updating formula (4.11) becomes:

$$\mathbf{a}_{(t+1)} = \mathbf{a}_{(t)} + \widehat{W}\widehat{\mathbf{y}} + \gamma K \widehat{\mathbf{1}} = \mathbf{a}_{(t)} + \widehat{W}\widehat{\mathbf{y}} \overset{\mathrm{R}}{\oplus} \gamma K$$

$$\mathbf{y}_{(t+1)} = \frac{\widehat{\mathbf{1}} + \tanh(\mathbf{a}_{(t+1)})}{2} = \frac{1 \overset{\mathrm{R}}{\oplus} \tanh(\mathbf{a}_{(t+1)})}{2}$$

and the α, β, γ and η constants are used to tune the process. $\mathbf{a}$ is initialized to zero $\mathbf{a}_{(0)} = \widehat{\mathbf{0}}$ and $\mathbf{y}_{(0)}$ is initialized to a small random value (in order to break any possible symmetry).

Remarks:

- In digital simulations, if used for large K, note that $\widehat{W}$ is a block circulant and specialized algorithms for fast matrix–vector multiplication do exist.
- Paths with low cost, vertices with many paths or with small path average cost constitute attractors and thus are favoured over others, thus they are "obstacles" in finding the global minima.

Discussion: energy function significance

Considering $\mathbf{x} = \gamma K \widehat{\mathbf{1}}$ and $\mathbf{t} = \widehat{\mathbf{0}}$ then the energy (4.12) becomes $E = -\frac{1}{2}\,\widehat{\mathbf{y}}^{\mathrm{T}}\widehat{W}\widehat{\mathbf{y}} - \gamma K \mathbf{y}^{\mathrm{T}}\widehat{\mathbf{1}}$. The previously defined weights are:

$$w_{\ell jki} = -\alpha\delta_{\ell k}(1-\delta_{ji}) - \beta(1-\delta_{\ell k})\delta_{ji} - \gamma - \eta d_{\ell k}(\delta_{\underset{K}{\mathrm{mod}}(j)+1,i} + \delta_{j,\underset{K}{\mathrm{mod}}(i)+1})$$

note that the fourth term has nonzero elements only for $i = j+1$, respectively $i = j-1$ (where the special cases have been already discussed).

❖ $E^{(1)}$, $E^{(2)}$
❖ $E^{(3)}$, $E^{(4)}$

The energy $E = -\frac{1}{2}\sum\limits_{\ell jki} w_{\ell jki} y_{\ell j} y_{ki} - \gamma K \sum\limits_{\ell j} y_{\ell j}$ may be explicitly written as a sum of four terms plus a constant:

$$\begin{aligned}
E = {} & \frac{\alpha}{2}\sum_{\ell=1}^{K}\sum_{\substack{j,i=1\\ j\neq i}}^{K} y_{\ell j}y_{\ell i} + \frac{\beta}{2}\sum_{\substack{\ell,k=1\\ \ell\neq k}}^{K}\sum_{j=1}^{K} y_{\ell j}y_{kj} + \frac{\gamma}{2}\sum_{\ell,j=1}^{K}\sum_{k,i=1}^{K} y_{\ell j}y_{ki} \\
& + \frac{\eta}{2}\sum_{\substack{\ell,k=1\\ \ell\neq k}}^{K}\sum_{j=1}^{K} d_{\ell k}(y_{\ell j}y_{k,\underset{K}{\mathrm{mod}}(j)+1} + y_{\ell,\underset{K}{\mathrm{mod}}(j)+1}y_{kj}) - \gamma K \sum_{\ell=1}^{K}\sum_{j=1}^{K} y_{\ell j} \\
= {} & \frac{\alpha}{2}\sum_{k=1}^{K}\sum_{\substack{j,i=1\\ j\neq i}}^{K} y_{\ell j}y_{\ell i} + \frac{\beta}{2}\sum_{\substack{\ell,k=1\\ \ell\neq k}}^{K}\sum_{j=1}^{K} y_{\ell j}y_{kj} + \frac{\gamma}{2}\left(\sum_{\ell,j=1}^{K} y_{\ell j} - K\right)^2 \\
& + \frac{\eta}{2}\sum_{\substack{\ell,k=1\\ \ell\neq k}}^{K}\sum_{j=1}^{K} d_{\ell k}(y_{\ell j}y_{k,\underset{K}{\mathrm{mod}}(j)+1} + y_{\ell,\underset{K}{\mathrm{mod}}(j)+1}y_{kj}) - \frac{\gamma K^2}{2} \\
= {} & E^{(1)} + E^{(2)} + E^{(3)} + E^{(4)} - \frac{\gamma K^2}{2}
\end{aligned}$$

According to Theorem 4.4.2, during a network run, the energy decreases and reaches a minima. This may be interpreted as follows:

① Energy minimum will favor states that have each city only once in the tour:

$$E^{(1)} = \frac{\alpha}{2} \sum_{\ell=1}^{K} \sum_{\substack{j,i=1 \\ j \neq i}}^{K} y_{\ell j} y_{\ell i}$$

reaches the global minima $E^{(1)} = 0$ if and only if each city appears only once in the tour such that the products $y_{\ell j} y_{\ell i}$ are either of type $1 \cdot 0$ or $0 \cdot 0$, i.e. there is only one 1 in each row of matrix (4.13).

The $1/2$ factor means that the terms $y_{\ell j} y_{\ell i} = y_{\ell i} y_{\ell j}$ will be added only once, not twice.

② Energy minimum will favor states that have each position of the tour occupied by only one city, i.e. if a city is the ℓ-th to be visited then any other city can't be in the same ℓ-th position in the tour:

$$E^{(2)} = \frac{\beta}{2} \sum_{j=1}^{K} \sum_{\substack{\ell,k=1 \\ \ell \neq k}}^{K} y_{\ell j} y_{kj}$$

reaches the global minima $E^{(2)} = 0$ if and only if each city has different order number in the tour such that the products $y_{\ell j} y_{kj}$ are either of type $1 \cdot 0$ or $0 \cdot 0$, i.e. there is only one 1 in each column of the matrix (4.13).

The $1/2$ factor means that the terms $y_{\ell j} y_{kj} = y_{kj} y_{\ell j}$ will be added only once, not twice.

③ Energy minimum will favor states that have all cities in the tour:

$$E^{(3)} = \frac{\gamma}{2} \left(\sum_{\ell,j=1}^{K} y_{\ell j} - K \right)^2$$

reaches minimum $E^{(3)} = 0$ if all cities are represented in the tour, i.e. $\sum\limits_{\ell,j=1}^{K} y_{\ell j} = K$ (there are K and only K "ones" in the whole matrix (4.13)). The fact that *if* a city is present, it appears once and only once was taken care of in previous terms.

The squaring shows that the *modulus* of the difference is important, otherwise energy may decrease for an increase of the number of missed cities, i.e. either $\sum\limits_{\ell,k=1}^{K} y_{\ell j} \lessgtr K$ is bad.

④ Energy minimum will favor states with minimum distance/cost of the tour:

$$E^{(4)} = \frac{\gamma}{2} \sum_{\substack{\ell,k=1 \\ \ell \neq k}}^{K} \sum_{j=1}^{K} d_{\ell k} (y_{\ell j} y_{k,\underset{K}{\mathrm{mod}}(j)+1} + y_{\ell,\underset{K}{\mathrm{mod}}(j)+1} y_{kj})$$

First term: if $y_{\ell j} \simeq 0$ then no distance will be added; if $y_{\ell j} \simeq 1$ then 2 cases arise:

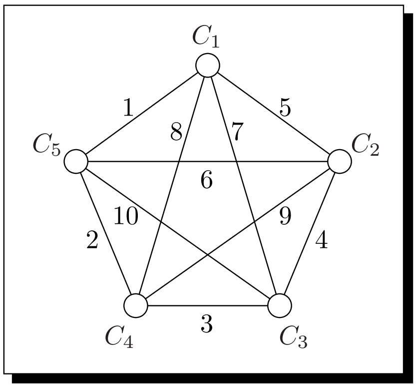

Figure 4.6: *City arrangement and distances for travelling salesperson numerical example.*

(a) $y_{k,\operatorname{mod}_K(j)+1} \simeq 1$ that means that city C_k is the next one in the tour and the distance $d_{\ell k}$ will be added;

(b) $y_{k,\operatorname{mod}_K(j)+1} \simeq 0$ that means that the city C_k is not the next one on the tour and then the corresponding distance will not be added.

Similar discussion for y_{kj}.

The $1/2$ means that the distances $d_{\ell k} = d_{k\ell}$ will be added only once, not twice. From previous terms there should be only one digit "1" on each row so a distance $d_{\ell k}$ should appear only once (the factor $1/2$ was already considered). Overall, $E^{(4)}$ should stabilize to a value proportional to the cost of tour, this cost should be minimum (or more generally a local minima as reaching the global minimum is not guaranteed).

⑤ The term $-\gamma K^2/2$ is just an additive constant, used to create the square in $E^{(3)}$.

Numerical example

A setup of 5 cities was built and costs were assigned for "travelling"; see Figure 4.6. The distances and constants were set to:

$$D = \begin{pmatrix} 0 & 5 & 7 & 8 & 1 \\ 5 & 0 & 4 & 9 & 6 \\ 7 & 4 & 0 & 3 & 10 \\ 8 & 9 & 3 & 0 & 2 \\ 1 & 6 & 10 & 2 & 0 \end{pmatrix} , \qquad \begin{array}{lll} \alpha = 0.2 & , & \gamma = 0.2 \\ \beta = 0.2 & , & \eta = 0.01 \end{array}$$

Note that the "cheapest" path is C_1–C_2–C_3–C_4–C_5–C_1 (cost $= 15$) while the most "expensive is C_1–C_4–C_2–C_5–C_3–C_1 (cost $= 40$).

After initializing $\mathbf{y}_{(0)}$ with small values in range $[0, 0.01]$ the network was run[6] for 80 epochs with the following result:

$$Y = \begin{pmatrix} 0.5942664 & 0.2529832 & 0.0010231 & 0.0006491 & 0.0078743 \\ 0.0155616 & 0.5121468 & 0.0344982 & 0.0041798 & 0.0018511 \\ 0.0034565 & 0.0066986 & 0.7138115 & 0.0030159 & 0.0280554 \\ 0.0002931 & 0.0001799 & 0.0007551 & 0.9886174 & 0.0200765 \\ 0.2026421 & 0.0083686 & 0.0316909 & 0.0004419 & 0.5972267 \end{pmatrix} \xrightarrow{t\to\infty} \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}$$

[6] The actual Scilab code is in the source tree of this book, e.g. `Matrix_ANN_1/Scilab/BAM_travel_sp.sci`, note that actual version numbers may differ.

which shows that the network successfully found the optimal path. Note that in Y there are two relatively high values: Y_{15} and Y_{12} — this is due to the attractor created by path C_1–C_5 with smaller cost; C_5 is also an attractor as paths emerging from it have smaller cost on average.

Remarks:

- One way to check the correctness of implementation is as follows: set the cost to zero ($\eta = 0$) and keep all other parameters unchanged (including initial $\widehat{\mathbf{y}}_{(0)}$). Proceeding in this manner we obtain:

$$\lim_{t\to\infty} Y_{(t)} = \begin{pmatrix} 0&0&0&0&1 \\ 1&0&0&0&0 \\ 0&0&1&0&0 \\ 0&0&0&1&0 \\ 0&1&0&0&0 \end{pmatrix}$$

  i.e. path C_1–C_2–C_5–C_3–C_4–C_1 with an (above) average cost as expected from a random initial input (cost $= 32$); different $\widehat{\mathbf{y}}_{(0)}$ may set the network to a different final state.

CHAPTER 5

The Counterpropagation Network

The counterpropagation network (CPN) is an example of an ANN interconnectivity. From some subnetworks, a new one is created, to form a reversible, heteroassociative memory[1]. CPN

5.1 CPN Architecture

Consider a set of vector pairs $\{\mathbf{x}_p, \mathbf{y}_p\}$ $(x_{ip}, y_{jp} \geqslant 0)$ which may be classified into several classes $\{\mathcal{C}_h\}_{h=\overline{1,H}}$. The CPN associates an input $\mathbf{x}$ vector with a $\langle \mathbf{y} \rangle_h \in \mathcal{C}_h$ for which the corresponding $\langle \mathbf{x} \rangle_h$ is closest to input $\mathbf{x}$. $\langle \mathbf{x} \rangle_h$ and $\langle \mathbf{y} \rangle_h$ are the "averages" over those $\mathbf{x}_p$, respectively $\mathbf{y}_p$ which are from *the same class* $\mathcal{C}_h$.

CPN may also work in reverse, a $\mathbf{y}$ being input and $\langle \mathbf{x} \rangle_h$ being retrieved.

The CPN architecture consists of 5 layers on 3 levels. The input level contains x and y layers; the middle level contains the hidden layer and the output level contains the x' and y' layers. See Figure 5.1 on the next page. Note that each neuron on x layer, input level, receives the whole $\mathbf{x}$ (and similar for y layer) and also there are *direct* links between input and output levels.

Considering a trained network, an $\mathbf{x}$ vector is applied, $\mathbf{y}$ is set to $\widehat{\mathbf{0}}$ at input level and the corresponding vector $\langle \mathbf{y} \rangle_h$ is retrieved. When running in reverse a $\mathbf{y}$ vector is applied, $\mathbf{x} = \widehat{\mathbf{0}}$ at input level and the corresponding $\langle \mathbf{x} \rangle_h$ is retrieved. Both cases are identical, only the first will be discussed in detail.

This functionality is achieved as follows:

- The first level normalizes the input vector; the size of this level is $N + K$.

[1]See "The BAM/Hopfield Memory" chapter for the definition.

[5.1]See [FS92] pp. 213–234.

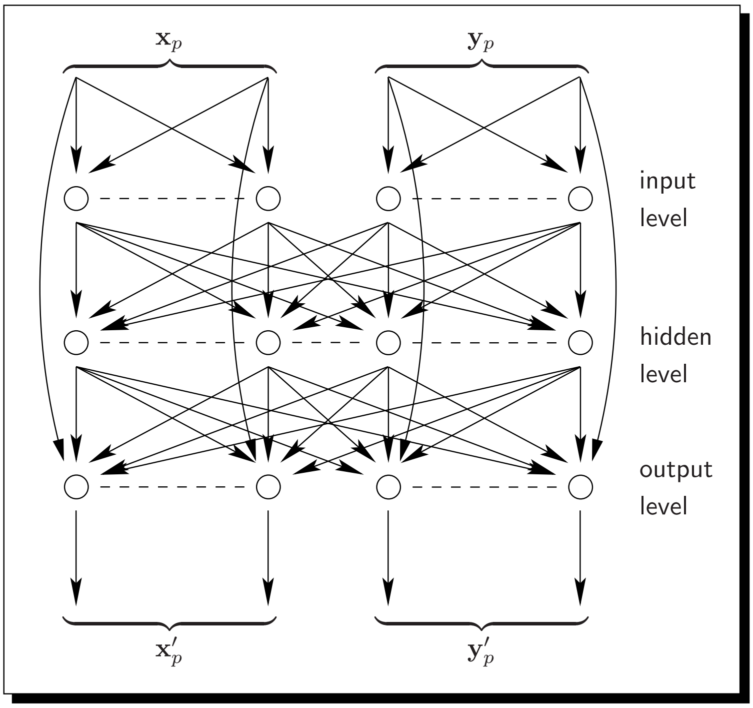

Figure 5.1: *The CPN architecture.*

one–of–k encoding

- The second level (hidden layer) does a classification of input vector, outputting an one–of–h (usually known as one–of–k but here the k index will be reserved for $\mathbf{y}$ components) encoded classification, i.e. the outputs of all hidden neurons are zero with the exception of one: and the number/label of its corresponding neuron identifies the input vector as belonging to a class (as being closest to some particular, "representative", previously stored, vector); the size of this layer is H.
- Based on the classification performed on hidden layer, the output layer actually retrieve a "representative" vector; the size of this level is $N + K$, the same as the input level.

All three subnetworks are quasi–independent and training at one level is performed only *after* the training at the previous level has been finished.

5.2 CPN Dynamics

5.2.1 Input Layer

❖ $\mathbf{z}_{(x)}$

Consider the x input layer[2]. Let $\mathbf{z}_{(x)}$ be the output vector of the input x layer. See Figure 5.2 on the facing page.

The input layer has to achieve a normalization of input; this may be done if the neuronal

[5.2] See [FS92] pp. 213–234.

[2] An identical discussion goes for y layer, as previously specified.

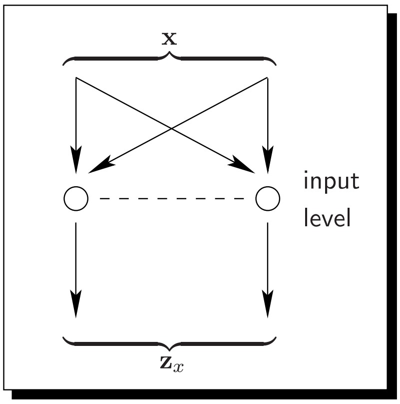

Figure 5.2: *The input layer.*

activity on the input layer is defined as follows:

- each neuron receives a positive excitation proportional to it's corresponding input αx_i, $\alpha = \text{const.}$, $\alpha > 0$; ❖ α
- each neuron receives a negative excitation from all neurons on the same layer, including itself, equal to: $-z_{(x)i} \sum_{j=1}^{N} x_j = -z_{(x)i} \widehat{\mathbf{1}}^{\mathrm{T}} \mathbf{x}$;
- the input vector $\mathbf{x}$ is applied at time $t = 0$ and removed ($\mathbf{x}$ becomes $\widehat{\mathbf{0}}$) at time $t = t'$;
- in the absence of input, the neuronal output $z_{(x)i}$ decreases to zero following an exponential, defined by $\beta = \text{const.}$, $\beta > 0$, i.e. $z_{(x)i} \propto e^{-\beta t}$. ❖ β

Then the neuronal behaviour may be summarized as:

$$\frac{dz_{(x)i}}{dt} = -\beta z_{(x)i} + \alpha x_i - z_{(x)i} \widehat{\mathbf{1}}^{\mathrm{T}} \mathbf{x} \quad \text{for } t \in [0, t')$$
$$\frac{dz_{(x)i}}{dt} = -\beta z_{(x)i} \quad \text{for } t \in [t', \infty)$$

(note that these equations are versions of passive decay with resting potential). In matrix notation:

$$\frac{d\mathbf{z}_{(x)}}{dt} = -\beta \mathbf{z}_{(x)} + \alpha \mathbf{x} - \mathbf{z}_{(x)} \widehat{\mathbf{1}}^{\mathrm{T}} \mathbf{x} \quad \text{for } t \in [0, t') \tag{5.1a}$$

$$\frac{d\mathbf{z}_{(x)}}{dt} = -\beta \mathbf{z}_x \quad \text{for } t \in [t', \infty) \tag{5.1b}$$

The boundary conditions are $\mathbf{z}_{(x)(0)} = \widehat{\mathbf{0}}$ (starts from zero, no previously applied signal) and $\lim\limits_{t \to \infty} \mathbf{z}_{(x)(t)} = \widehat{\mathbf{0}}$ (returns to zero after the signal have been removed). For continuity purposes the condition $\lim\limits_{t \nearrow t'} \mathbf{z}_{(x)(t)} = \lim\limits_{t \searrow t'} \mathbf{z}_{(x)(t)}$ should be imposed. With these limit conditions, the solutions to (5.1a) and (5.1b) are:

$$\mathbf{z}_{(x)} = \frac{\alpha \mathbf{x}}{\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}} \left\{ 1 - \exp\left[- \left(\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x} \right) t \right] \right\} \quad \text{for } t \in [0, t') \tag{5.2a}$$

$$\mathbf{z}_{(x)} = \frac{\alpha \mathbf{x}}{\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}} \left\{ 1 - \exp\left[- \left(\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x} \right) t' \right] \right\} e^{-\beta(t-t')} \quad \text{for } t \in [t', \infty) \tag{5.2b}$$

Proof. **1.** From (5.1a), for $t \in [0, t')$:

$$\frac{d\mathbf{z}_{(x)}}{dt} + \left(\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x} \right) \mathbf{z}_{(x)} = \alpha \mathbf{x} \quad \Rightarrow \quad \frac{dz_{(x)i}}{dt} + \left(\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x} \right) z_{(x)i} = \alpha x_i$$

First a solution for the homogeneous equation is to be found:

$$\frac{dz_{(x)i}}{dt} + \left(\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x} \right) z_{(x)i} = 0 \quad \Rightarrow \quad z_{(x)i} = z_{(x)i0} \exp\left[- \left(\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x} \right) t \right]$$

❖ $z_{(x)0}$

(by variable separation and integration) where $z_{(x)i0}$ is the constant of integration.

A general solution to the non–homogeneous equation may be found by using an integrating factor considering $\mathbf{z}_{(x)0} = \mathbf{z}_{(x)0}(t)$. Then from (5.1a) ($x_i =$ const.):

$$\frac{d\mathbf{z}_{(x)0}}{dt} \exp[-(\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x})t] = \alpha \mathbf{x}$$

$$\Rightarrow \quad \mathbf{z}_{(x)0} = \int \alpha \mathbf{x} \exp[(\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x})t] \, dt = \frac{\alpha \mathbf{x}}{\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}} \exp[(\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x})t] + \text{const.}$$

and then, substituting back into homogeneous equation solution, and using the first boundary condition gives const. $= -\frac{\alpha \mathbf{x}}{\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}}$ and thus (5.2a).

2. From equation (5.1b), by separating variables and integrating, for $t \in [t', \infty)$:

$$\mathbf{z}_{(x)} = \mathbf{z}_{(x)0} e^{-\beta(t-t')} \quad \mathbf{z}_{(x)0} = \text{const.}$$

Then, from the continuity condition and (5.2a), the $z_{(x)0}$ is:

$$\mathbf{z}_{(x)0} = \frac{\alpha \mathbf{x}}{\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}} \{1 - \exp[-(\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x})t']\} \quad \square$$

The output of a neuron from the input layer as a function of time is shown in Figure 5.3 on the next page. The maximum value attainable on output is $\mathbf{z}_{(x)\text{max}} = \frac{\alpha \mathbf{x}}{\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}}$, this value occuring in the limit as $t' \to \infty$.

Remarks:

- In practice, due to the exponential nature of the output, close values to $\mathbf{z}_{(x)\text{max}}$ are obtained for t' relatively small, see again Figure 5.3 on the facing page, about 98% of the maximum value was attained at $t = t' = 4$.
- Even if the input vector $\mathbf{x}$ is big ($x_i \to \infty$) the output is bounded but parallel with the input:

$$\mathbf{z}_{(x)\text{max}} = \frac{\alpha \mathbf{x}}{\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}} = \boldsymbol{\zeta}_{(x)} \frac{\alpha \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}}{\beta + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}} \propto \boldsymbol{\zeta}_{(x)} \quad \text{where} \quad \boldsymbol{\zeta}_{(x)} = \frac{\mathbf{x}}{\widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}} \propto \mathbf{x}$$

❖ $\boldsymbol{\zeta}_{(x)}$ reflectance pattern

$\boldsymbol{\zeta}_{(x)}$ being called *the reflectance pattern* and is "normalized" in the sense that it sums to unity: $\widehat{\mathbf{1}}^{\mathrm{T}}\boldsymbol{\zeta}_{(x)} = 1$.

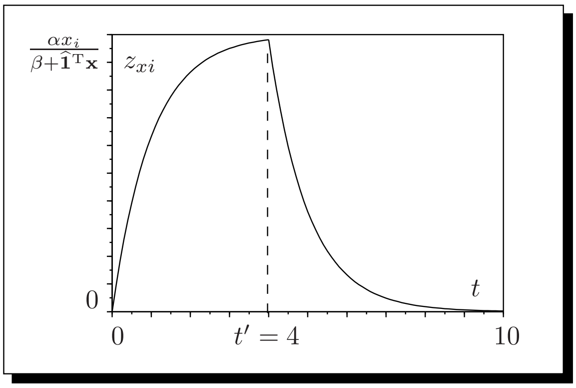

Figure 5.3: *The output of a neuron from the input layer as a function of time ($\beta = 1$, $z_{(x)i\mathsf{max}} = 1$).*

5.2.2 Hidden Layer

The Instar

The neurons from the hidden layer are called *instars*.

The input vector is $\mathbf{z} = \{z_i\}_{i=\overline{1,N+K}}$, here $\mathbf{z}$ will contain the outputs from both x and y layers. Let be H the dimension (number of neurons) and $\mathbf{z}_{(h)}$ the output vector of hidden layer. The fact that H equals the number of classes is not a coincidence, later it will be shown that there has to be *at least* one hidden neuron for each class. ❖ $\mathbf{z}$, H, $\mathbf{z}_{(h)}$

Let $\{w_{hi}\}_{\substack{h=\overline{1,H}\\ i=\overline{1,N+K}}}$ be the weight matrix (by which $\mathbf{z}$ enters the hidden layer) such that the total input to hidden neuron h is $W_{(h,:)}\mathbf{z}$.

The equations governing the behavior of hidden neuron h are defined in a similar way as those of input layer ($\mathbf{z}$ is applied from $t = 0$ to $t = t'$):

$$\frac{d\mathbf{z}_{(h)}}{dt} = -a\mathbf{z}_{(h)} + bW\mathbf{z} \quad \text{for } t \in [0, t') \tag{5.3a}$$

$$\frac{d\mathbf{z}_{(h)}}{dt} = -a\mathbf{z}_{(h)} \quad \text{for } t \in [t', \infty) \tag{5.3b}$$

where $a, b =$ const., $a, b > 0$, and boundary conditions are $\mathbf{z}_{(h)(0)} = \widehat{\mathbf{0}}$, $\lim\limits_{t\to\infty} \mathbf{z}_{(h)(t)} = \widehat{\mathbf{0}}$ and, for continuity purposes $\lim\limits_{t \nearrow t'} \mathbf{z}_{(h)(t)} = \lim\limits_{t \searrow t'} \mathbf{z}_{(h)(t)}$. ❖ a, b

The weight matrix is defined to change as well (network is learning) according to the following equation:

$$\frac{dW_{(h,:)}}{dt} = \begin{cases} -c\left[W_{(h,:)} - \mathbf{z}^{\mathrm{T}}\right] & \text{if } \mathbf{z} \neq \widehat{\mathbf{0}} \\ \widehat{\mathbf{0}}^{\mathrm{T}} & \text{if } \mathbf{z} = \widehat{\mathbf{0}} \end{cases} \tag{5.4}$$

where $c =$ const., $c > 0$ and boundary condition is $W_{(h,:)(0)} = \widehat{\mathbf{0}}^{\mathrm{T}}$. The reason to define weights change as such will be detailed below; note also that only the weights related to one hidden neuron will adapt for a given input (this will also be explained later). ❖ c

Assuming that *the weight matrix is changing much slower that the neuron output* then $W_{(h,:)}\mathbf{z} \simeq$ const. and the solutions to (5.3) are:

$$\mathbf{z}_{(h)} = \frac{b}{a} W\mathbf{z} \left(1 - e^{-at}\right) \qquad \text{for } t \in [0, t') \tag{5.5a}$$

$$\mathbf{z}_{(h)} = \frac{b}{a} W\mathbf{z} \left(1 - e^{-at'}\right) e^{-a(t-t')} \quad \text{for } t \in [t', \infty) \tag{5.5b}$$

Proof. This is proven in a similar way as for the input layer, see Section 5.2.1, proof of equations (5.2a) and (5.2b). It may also be checked directly. □

The output of hidden neuron is similar to the output of input layer, see also Figure 5.3 on the page before, but the maximal possible value is $\mathbf{z}_{(h)\mathrm{max}} = \frac{b}{a} W\mathbf{z}$, attained in the limit $t' \to \infty$.

Assuming that an input $\mathbf{z}$ is applied and kept sufficiently long then the solution[3] to (5.4) is:

$$W_{(h,:)} = \mathbf{z}^{\mathrm{T}}(1 - e^{-ct})$$

i.e. $W_{(h,:)}$ moves towards $\mathbf{z}^{\mathrm{T}}$. If $\mathbf{z}$ is kept applied sufficiently long then $W_{(h,:)} \xrightarrow{t\to\infty} \mathbf{z}^{\mathrm{T}}$. This was one reason of building (5.4) in this manner.

Remarks:

- In absence of input ($\mathbf{z} = \widehat{\mathbf{0}}$), if learning would continue then: $\frac{dW_{(h,:)}}{dt} = -cW_{(h,:)}$ $\Rightarrow W_{(h,:)} = \mathbf{c}^{\mathrm{T}} e^{-ct} \xrightarrow{t\to\infty} \widehat{\mathbf{0}}^{\mathrm{T}}$ ($\mathbf{c}$ being a constant vector).

Consider a set of input vectors $\{\mathbf{z}_p\}_{p=\overline{1,P}}$ applied as follows: $\mathbf{z}_1$ between $t = 0$ and $t = t_1$, …, $\mathbf{z}_P$ between $t = t_{P-1}$ and $t = t_P$; then:

$$\begin{aligned}
W_{(h,:)(t_1)} &= \mathbf{z}_1 \left(1 - e^{-ct_1}\right) \\
W_{(h,:)(t_2)} &= \mathbf{z}_2 \left(1 - e^{-c(t_2-t_1)}\right) + W_{(h,:)(t_1)} \quad , \quad \dots \\
W_{(h,:)(t_P)} &= \mathbf{z}_P \left(1 - e^{-c(t_P-t_{P-1})}\right) + W_{(h,:)(t_{P-1})} = \sum_{p=1}^{P} \mathbf{z}_p \left(1 - e^{-c(t_p-t_{p-1})}\right)
\end{aligned} \tag{5.6}$$

This shows that the final weight vector $W_{(h,:)}$ represents a linear combination of all input vectors $\{\mathbf{z}_p\}$. Because the coefficients $\left(1 - e^{-c(t_p-t_{p-1})}\right)$ are all positive then the final direction of $W_{(h,:)}$ will be an "average" direction pointed by $\{\mathbf{z}_p\}$ and this is exactly the purpose of defining (5.4) as it was. See Figure 5.4 on the facing page.

Remarks:

- It is also possible to give each $\mathbf{z}_p$ a time–slice dt and when finishing with $\mathbf{z}_P$ to start over with $\mathbf{z}_1$ until some (external) stopping conditions are met.

The trained instar is able to classify the direction of input vectors (see (5.5a)):

$$z_{(h)h} \propto W_{(h,:)}\mathbf{z} = \|W_{(h,:)}\| \, \|\mathbf{z}\| \cos(\widehat{W_{(h,:)}, \mathbf{z}}) \propto \cos(\widehat{W_{(h,:)}, \mathbf{z}})$$

[3]The differential equation (5.4) is very similar to previous encountered equations. It may be proven also by direct replacement.

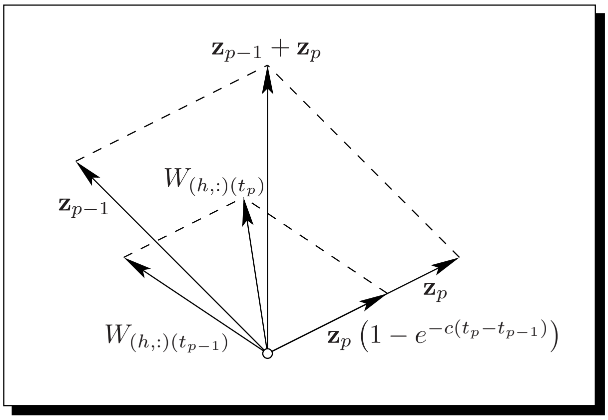

Figure 5.4: *The directions taken by weight vectors $W_{(h,:)}$, relative to input, in hidden layer.*

The Competitive Network

The hidden layer of CPN is made out of instars interconnected such that each inhibits all others and eventually there is only one "winner" (the instars compete one against each other). The purpose of the hidden layer is to classify the normalized input vector $\mathbf{z}$ (which is proportional to input). Its output is of the one–of–h encoding type (known in general as one–of–k, but h will be used here), i.e. all neurons have output zero except one neuron h, the "winner" which has output one. Note that it is assumed that classes do *not* overlap, i.e. an input vector may belong to one class only.

The property of instars that their associated weight vector moves towards an average of input will be preserved, but a feedback function is to be added, to ensure the required output. Let $f = f(\mathbf{z}_{(h)})$ be the feedback function of the instars, i.e. the value added at the neuron input. See Figure 5.5 on the next page ❖ f

Then the instar equations (5.3a) and (5.3b) are redefined as:

$$\frac{dz_{(h)h}}{dt} = -az_{(h)h} + b[f(z_{(h)h}) + W_{(h,:)}\mathbf{z}] - z_{(h)h}\sum_{g=1}^{H}[f(z_{(h)g}) + W_{(g,:)}\mathbf{z}] \;, \qquad \text{for } t \in [0, t') \;, \tag{5.7a}$$

$$\frac{dz_{(h)h}}{dt} = -az_{(h)h} \;, \qquad \text{for } t \in [t', \infty) \;, \tag{5.7b}$$

where $a, b = \text{const.}$, $a, b > 0$. The second term represents the positive feedback while the third term is the negative feedback. In matrix notation: ❖ a, b

$$\frac{d\mathbf{z}_{(h)}}{dt} = -a\mathbf{z}_{(h)} + b[f(\mathbf{z}_{(h)}) + W\mathbf{z}] - \mathbf{z}_{(h)}\widehat{\mathbf{1}}^{\mathrm{T}}[f(\mathbf{z}_{(h)}) + W\mathbf{z}] \;, \qquad \text{for } t \in [0, t') \;,$$

$$\frac{d\mathbf{z}_{(h)}}{dt} = -a\mathbf{z}_{(h)} \;, \qquad \text{for } t \in [t', \infty) \;.$$

The feedback function f has to be selected such that the hidden layer performs as a compet- feedback function

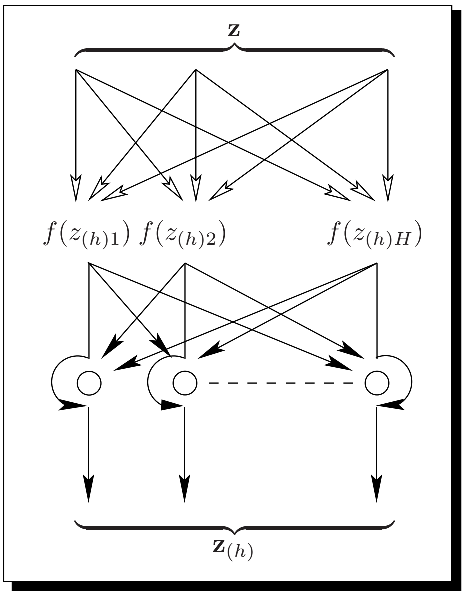

Figure 5.5: *The competitive hidden layer, neurons receive both input* $\mathbf{z}$ *and feedback from the other neurons on same hidden layer.*

itive layer, i.e. at equilibrium all neurons will have the output zero except one, the "winner" which will have the output "1". This type of behaviour is also known as "winner-takes-all" [4].

❖ n For a feedback function of type $f(\mathbf{z}_{(h)}) = \mathbf{z}_{(h)}^{\odot n}$ where $n > 1$, equations (5.7) define a competitive layer.

❖ $\widetilde{\mathbf{z}}_{(h)}$, $z_{(h)\text{tot}}$ *Proof.* First a *change of variable* is performed as follows: $z_{(h)\text{tot}} \equiv \widehat{\mathbf{1}}^{\mathrm{T}} \mathbf{z}_{(h)}$ and $\widetilde{\mathbf{z}}_{(h)} \equiv \frac{\mathbf{z}_{(h)}}{z_{(h)\text{tot}}}$; then $\mathbf{z}_{(h)} = \widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}}$ and $\frac{d\mathbf{z}_{(h)}}{dt} = \frac{d\widetilde{\mathbf{z}}_{(h)}}{dt} z_{(h)\text{tot}} + \widetilde{\mathbf{z}}_{(h)} \frac{dz_{(h)\text{tot}}}{dt}$.

By making the sum over h in (5.7a) (and using $f(z_{(h)g}) = f(\widetilde{z}_{(h)g} z_{(h)\text{tot}})$):

$$\frac{dz_{(h)\text{tot}}}{dt} = -a z_{(h)\text{tot}} + (b - z_{(h)\text{tot}}) \widehat{\mathbf{1}}^{\mathrm{T}} [f(\widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}}) + W\mathbf{z}] \; , \tag{5.8}$$

and substituting $\mathbf{z}_{(h)} = \widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}}$ in (5.7a):

$$\frac{d\mathbf{z}_{(h)}}{dt} = -a \widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}} + b[f(\widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}}) + W\mathbf{z}] - \widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}} \widehat{\mathbf{1}}^{\mathrm{T}} [f(\widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}}) + W\mathbf{z}] \; . \tag{5.9}$$

As established: $\frac{d\widetilde{\mathbf{z}}_{(h)}}{dt} z_{(h)\text{tot}} = \frac{d\mathbf{z}_{(h)}}{dt} - \widetilde{\mathbf{z}}_{(h)} \frac{dz_{(h)\text{tot}}}{dt}$, by using (5.8) and (5.9):

$$\frac{d\widetilde{\mathbf{z}}_{(h)}}{dt} z_{(h)\text{tot}} = b[f(\widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}}) + W\mathbf{z}] - b\widetilde{\mathbf{z}}_{(h)} \widehat{\mathbf{1}}^{\mathrm{T}} [f(\widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}}) + W\mathbf{z}] \tag{5.10}$$

The following cases, with regard to the feedback function, may be discussed:

- $n = 1$, identity function: $f(\widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}}) = \widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}}$. Then, by using $\widehat{\mathbf{1}}^{\mathrm{T}} \widetilde{\mathbf{z}}_{(h)} = 1$, the above equation becomes:

$$\frac{d\widetilde{\mathbf{z}}_{(h)}}{dt} z_{(h)\text{tot}} = bW\mathbf{z} - b\widetilde{\mathbf{z}}_{(h)} \widehat{\mathbf{1}}^{\mathrm{T}} W\mathbf{z}$$

[4] There is a strong resemblance with SOM networks but here the feedback is direct.

The stable value, obtained by setting $\frac{d\widetilde{\mathbf{z}}_{(h)}}{dt} = \widehat{\mathbf{0}}$, is $\widetilde{\mathbf{z}} = \frac{W\mathbf{z}}{\widehat{\mathbf{1}}^{\mathrm{T}} W\mathbf{z}}$.

- $n = 2$, square function: $f(\widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}}) = (\widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}})^{\odot 2}$. Again, as $\widehat{\mathbf{1}}^{\mathrm{T}} \widetilde{\mathbf{z}} = 1$, (5.10) may be rewritten, in components, as:

$$\begin{aligned}\frac{d\widetilde{z}_{(h)h}}{dt} z_{(h)\text{tot}} &= bf(\widetilde{z}_{(h)h} z_{(h)\text{tot}})\, \widehat{\mathbf{1}}^{\mathrm{T}} \widetilde{\mathbf{z}} - b\widetilde{z}_{(h)h} \widehat{\mathbf{1}}^{\mathrm{T}} f(\widetilde{\mathbf{z}}_{(h)} z_{(h)\text{tot}}) + bW_{(h,:)}\, \mathbf{z} - b\widetilde{z}_{(h)h} \widehat{\mathbf{1}}^{\mathrm{T}} W\mathbf{z} \\ &= b\widetilde{z}_{(h)h} z_{(h)\text{tot}} \sum_{g=1}^{H} \widetilde{z}_{(h)g} \left[\frac{f(\widetilde{z}_{(h)h} z_{(h)\text{tot}})}{\widetilde{z}_{(h)h} z_{(h)\text{tot}}} - \frac{f(\widetilde{z}_{(h)g} z_{(h)\text{tot}})}{\widetilde{z}_{(h)g} z_{(h)\text{tot}}} \right] + bW_{(h,:)}\, \mathbf{z} - b\widetilde{z}_{(h)h} \widehat{\mathbf{1}}^{\mathrm{T}} W\mathbf{z}\end{aligned}$$

  and then, considering the expression of f, the term: $\frac{f(\widetilde{z}_{(h)h} z_{(h)\text{tot}})}{\widetilde{z}_{(h)h} z_{(h)\text{tot}}} - \frac{f(\widetilde{z}_{(h)g} z_{(h)\text{tot}})}{\widetilde{z}_{(h)g} z_{(h)\text{tot}}}$ reduces to $z_{(h)\text{tot}}(\widetilde{z}_{(h)h} - \widetilde{z}_{(h)g})$, representing an amplification for $\widetilde{z}_{(h)h} > \widetilde{z}_{(h)g}$ while for $\widetilde{z}_{(h)h} < \widetilde{z}_{(h)g}$ is an inhibition. Term $bW_{(h,:)}\mathbf{z}$ is a constant with respect to $\widetilde{\mathbf{z}}_{(h)}$; term $-b\widetilde{\mathbf{z}}_{(h)} \widehat{\mathbf{1}}^{\mathrm{T}} W\mathbf{z}$ represents an inhibition.

  So, the differential equation describes the behaviour of a network where all neurons inhibit all others with smaller output than theirs and are inhibited by neurons with bigger output. The gap between neurons with high output and those with low output gets amplified. In this case *the layer acts like an "winner-takes-all"* network, eventually only one neuron, that one with the largest $\widetilde{z}_{(h)h}$ will have a non zero output.

The general case for $n > 1$ is similar to the case $n = 2$. □

Finally, only the winning neuron, let h be that one, is to be affected by the learning process. This neuron will represent the class $\mathcal{C}_h$ to which the input vector belongs; its associated weight vector $W_{(h,:)}$ has to be moved towards the average "representative" $\{\langle \mathbf{x} \rangle_h, \langle \mathbf{y} \rangle_h\}$ (combined as to form one vector), all other neurons should remain untouched (weights unchanged).

Two points to note here:

- It becomes obvious that there should be at least one hidden neuron for each class.
- Several neurons may represent the same class (but there will be only one winner at a time). This is particularly necessary if classes are represented by unconnected domains in $\mathbb{R}^{N+K}$ (because for a single neuron representative, the moving weight vector, towards the average input for the class, may become stuck somewhere in the middle and represent another class, see also Figure 5.6 on page 85) or for cases with deep entangling between various domains corresponding to different classes, i.e. "inter-knotted" non–convex domains. Note that any connected domain in $\mathbb{R}^{N+K}$ may be approximated arbitrary well by a sufficiently large number of convex sub–domains (e.g. hyper–cubes) and each sub–domain may be assigned to a hidden neuron.

5.2.3 Output Layer

The neurons on the output level are called outstars. The output level contains 2 layers: x' of dimension N and y' of dimension K — same as for input layers. For both the discussion goes the same way. The weight matrix describing the connection strengths between hidden and output layer is W'. ❖ W'

The purpose of the output level is to retrieve a pair $\{\langle \mathbf{x} \rangle_h, \langle \mathbf{y} \rangle_h\}$, where $\langle \mathbf{x} \rangle_h$ is closest to input $\mathbf{x}$, from an "average" over previously learned training vectors *from the same class*. The x' layer is discussed below, the y' part is, mutatis mutandis, identical. Note that the output level receives input from hidden layer but also the input $\{\mathbf{x}, \mathbf{y}\}$.

The differential equations governing the behavior of outstars are defined as (passive decay with resting potential):

$$\frac{dx'_i}{dt} = -\beta' x'_i + \alpha' x_i + \gamma' W'_{(i,:)} \mathbf{z}_{(h)} \;, \quad \text{for } t \in [0, t') \;,$$

$$\frac{dx'_i}{dt} = -\beta' x'_i + \gamma' W'_{(i,:)} \mathbf{z}_{(h)} \;, \qquad \text{for } t \in [t', \infty) \;,$$

❖ α', β', γ'

where $\alpha', \beta', \gamma' = \text{const.}$, $\alpha', \beta', \gamma' > 0$. In matrix notation:

$$\frac{d\mathbf{x}'}{dt} = -\beta' \mathbf{x}' + \alpha' \mathbf{x} + \gamma' W'_{(1:N,:)} \mathbf{z}_{(h)} \;, \quad \text{for } t \in [0, t') \;, \tag{5.11a}$$

$$\frac{d\mathbf{x}'}{dt} = -\beta' \mathbf{x}' + \gamma' W'_{(1:N,:)} \mathbf{z}_{(h)} \;, \qquad \text{for } t \in [t', \infty) \;, \tag{5.11b}$$

with boundary condition $\mathbf{x}'_{(0)} = \widehat{\mathbf{0}}$.

The weight matrix is changing (network is learning) by construction, according to the following equation:

$$\frac{dW'_{(1:N,h)}}{dt} = \begin{cases} -c' \left[W'_{(1:N,h)} - \mathbf{x} \right] & \text{if } \mathbf{z}_{(h)} \neq \widehat{\mathbf{0}} \\ \widehat{\mathbf{0}} & \text{if } \mathbf{z}_{(h)} = \widehat{\mathbf{0}} \end{cases} \tag{5.12}$$

❖ c'

where $c' = \text{const.}$, $c' > 0$, with the boundary condition $W'_{(1:N,h)(0)} = \widehat{\mathbf{0}}$ (the part for $\mathbf{z}_{(h)} = \widehat{\mathbf{0}}$ being defined in order to avoid weight "decaying", see also the discussion on hidden layer learning). Note that for a particular input vector there is just one winner on the hidden layer and thus just one column of matrix W' gets changed, all others remain the same (i.e. just the connections coming from the hidden winner to output layer get updated).

It is assumed that *the weight matrix is changing much slower than the neuron output.* Considering that the hidden layer is also much faster than output (only the asymptotic behaviour is of interest here), i.e. $W_{(i,:)} \mathbf{z}_{(h)} \simeq \text{const.}$ then the solution to (5.11a) is:

$$\mathbf{x}' = \left[\frac{\alpha'}{\beta'} \mathbf{x} + \frac{\gamma'}{\beta'} W'_{(1:N,:)} \mathbf{z}_{(h)} \right] (1 - e^{-\beta' t})$$

Proof. $\alpha' \mathbf{x} + \gamma' W'_{(1:N,:)} \mathbf{z}_{(h)} = \text{const.}$ and then the solution is built in the same way as for previous differential equations by starting to search for the solution for homogeneous equation. See also the proof of equations (5.2). The boundary condition is used to find the integration constant. □

The solution to weights update formula (5.12) is:

$$W'_{(1:N,h)} = \mathbf{x}(1 - e^{-c' t}) \tag{5.13}$$

Proof. The proof proceeds in the same way as for W. The result may also be proved by direct replacement. □

The asymptotic behaviour for $t \to \infty$ of $\mathbf{x}'$ and $W'_{(1:N,h)}$ are (from the solutions found):

$$\lim_{t \to \infty} \mathbf{x}' = \frac{\alpha'}{\beta'} \mathbf{x} + \frac{\gamma'}{\beta'} W'_{(1:N,:)} \mathbf{z}_{(h)} \quad \text{and} \quad \lim_{t \to \infty} W'_{(1:N,h)} = \mathbf{x} \tag{5.14}$$

After a training with a set of $\{\mathbf{x}_p, \mathbf{y}_p\}$ vectors, *from the same class* h, the weights will be: $W'_{(1:N,h)} \propto \langle \mathbf{x} \rangle_h$ respectively $W'_{(N+1:N+K,h)} \propto \langle \mathbf{y} \rangle_h$

Proof. The method is very similar to the discussion for (5.6). □

At runtime $\mathbf{x} \neq \widehat{\mathbf{0}}$ and $\mathbf{y} = \widehat{\mathbf{0}}$ to retrieve an $\mathbf{y}'$ (or vice-versa to retrieve an $\mathbf{x}'$). Then:

$$\lim_{t\to\infty} \mathbf{y}' = \frac{\alpha'}{\beta'}\mathbf{y} + \frac{\gamma'}{\beta'} W_{(N+1:N+K,:)}\mathbf{z}_{(h)} = \frac{\gamma'}{\beta'} W_{(N+1:N+K,:)}\mathbf{z}_{(h)} \tag{5.15}$$

The $\mathbf{z}_{(h)}$ represents a one–of–h encoding, this means that all its elements are 0 except for one $z_{(h)h} = 1$ (the winner) and then $W_{(N+1:N+K,:)}\mathbf{z}_H = W_{(N+1:N+K,h)}$, i.e. $\mathbf{z}_{(h)}$ selects the column h out of W' (for the corresponding winner) and:

$$\mathbf{y}' \propto W'_{(N+1:N+K,h)} \propto \langle \mathbf{y} \rangle_h$$

Remarks:

- Combining (5.15) and the $W'_{N+1:K,h}$ equivalent of (5.13) gives: $\lim_{t\to\infty} \mathbf{y}' = \frac{\gamma'}{\beta'} \langle \mathbf{y} \rangle_h$, which shows that in order to retrieve a $\mathbf{y}'$ of the same magnitude as $\langle \mathbf{y} \rangle_h$ it is necessary to choose the constants such that $\gamma'/\beta' \simeq 1$.

5.3 The Algorithm

The procedure uses the discrete–time approximation ($dt \to \Delta t = 1$, $dW \to \Delta W = W_{(t+1)} - W_{(t)}$), using the asymptotic values of the solutions to differential equations describing CPN. It is also considered that $\gamma'/\beta' = 1$.

Running procedure

1. The $\mathbf{x}$ vector is assumed to be known and the corresponding $\langle \mathbf{y} \rangle_h$ is to be retrieved. For the reverse run, when $\mathbf{y}$ is known and $\langle \mathbf{x} \rangle_k$ is to be retrieved, the algorithm is similar, just exchange $\mathbf{x}$ and $\mathbf{y}$.

   Make the $\mathbf{y}$ null ($\widehat{\mathbf{0}}$) at input and compute the normalized input vector $\mathbf{z}$:

   $$\mathbf{z}_{(1:N)} = \frac{\mathbf{x}}{\sqrt{\mathbf{x}^{\mathrm{T}}\mathbf{x}}} \quad \text{and} \quad \mathbf{z}_{(N+1:N+K)} = \widehat{\mathbf{0}}$$

2. Find a winning neuron in the hidden layer, an h for which: $W_{(h,:)}\mathbf{z} = \max_g W_{(g,:)}\mathbf{z}$.

3. Find the $\mathbf{y}$ vector in the W' matrix:

   $$\mathbf{y} = W'_{(N+1:N+K,h)}$$

Learning procedure

1. For all $\{\mathbf{x}_p, \mathbf{y}_p\}$ training sets, normalize the input vector, i.e. compute $\mathbf{z}_p$:

   $$\mathbf{z}_{(1:N)p} = \frac{\mathbf{x}_p}{\sqrt{\mathbf{x}_p^{\mathrm{T}}\mathbf{x}_p + \mathbf{y}_p^{\mathrm{T}}\mathbf{y}_p}} \quad \text{and} \quad \mathbf{z}_{(N+1:N+K)p} = \frac{\mathbf{y}_p}{\sqrt{\mathbf{x}_p^{\mathrm{T}}\mathbf{x}_p + \mathbf{y}_p^{\mathrm{T}}\mathbf{y}_p}}$$

[5.3] See [FS92] pp. 235–239.

Note that the input layer does just a normalisation of the input vectors. No further training is required.

2. Initialize weights on hidden layer. For all neurons h in the hidden layer ($h = \overline{1, H}$), select a representative input vector $\mathbf{z}_q$ for class $\mathcal{C}_h$ and then $W_{(h,:)} = \mathbf{z}_q^{\mathrm{T}}$; this way the weight vectors become automatically normalized.

   Note that in extreme cases there may be just one vector available for training for each class. In this case that vector becomes the "representative".

3. Train the hidden layer. For all normalized training vectors $\mathbf{z}_p$ find a winning neuron in the hidden layer, an h for which $W_{(h,:)}\mathbf{z}_p = \max\limits_{g} W_{(g,:)}\mathbf{z}_p$ (as normalization has been performed at previous steps such that the scalar product may be used to find the winner).

   Update the winner's weights, use (5.4) in discrete–time approximation:

$$W_{(h,:)(t+1)} = W_{(h,:)(t)} + c[\mathbf{z}_p^{\mathrm{T}} - W_{(h,:)(t)}] \ .$$

   The training of the hidden layer has to be completed before moving forward to the output layer. It is possible to reuse the training set and repeat learning over several epochs.

4. Initialize the weights on the output layer. As for the hidden layer, select a representative input vector pair $\{\mathbf{x}_q, \mathbf{y}_q\}$ for each class $\mathcal{C}_h$.

$$W'_{(1:N,h)} = \mathbf{x}_q \quad \text{and} \quad W'_{(N+1:N+K,h)} = \mathbf{y}_q \ .$$

   Another possibility would be to make an average over several $\{\mathbf{x}_p, \mathbf{y}_p\}$ belonging to the *same* class.

5. Train the output layer. For all training vectors $\mathbf{z}_p$ find a winning neuron on hidden layer, an h for which $W_{(h,:)}\mathbf{z}_p = \max\limits_{g} W_{(g,:)}\mathbf{z}_p$.

   Update the winner's output weights, use (5.12) in discrete–time approximation:

$$\begin{aligned} W'_{(1:N,h)(t+1)} &= W'_{(1:N,h)(t)} + c'[\mathbf{x}_p - W'_{(1:N,h)(t)}] \\ W'_{(N+1:N+K,h)(t+1)} &= W'_{(N+1:N+K,h)(t)} + c'[\mathbf{y}_p - W'_{(N+1:N+K,h)(t)}] \end{aligned}$$

   It is possible to reuse the training set and repeat learning over several epochs.

Remarks:

➥ From the learning procedure it becomes clear that the CPN is in fact a system composed of several semi–independent subnetworks:

- the input level which normalizes input,
- the hidden layer of "winner-takes-all" type and
- the output level which generates the actual required output.

Each level is independent and the training of the next layer starts only *after* the learning in the preceding layer have been done.

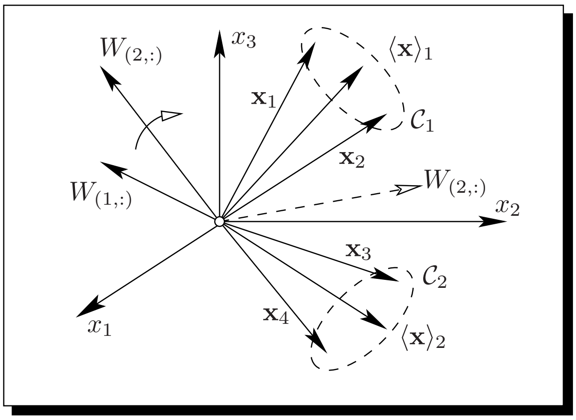

Figure 5.6: *The "stuck vector" situation: $W_{(1,:)}$ points to a far direction and the corresponding hidden neuron never wins; $W_{(2,:)}$ moves to its final position (showed with a dashed line) between $\langle \mathbf{x} \rangle_1$ and $\langle \mathbf{x} \rangle_2$ and becomes representative for both classes.*

- Usually the CPN is used to classify an input vector $\mathbf{x}$ as belonging to a class represented by $\langle \mathbf{x} \rangle_h$. A set of input vectors $\{\mathbf{x}_p\}$ will be used to train the network such that it will have the output $\{\langle \mathbf{x} \rangle_h, \langle \mathbf{y} \rangle_h\}$ if $\mathbf{x} \in \mathcal{C}_h$ (see also Figure 5.6).
- An unfortunate choice of weight vectors for the hidden layer $W_{(h,:)}$ may lead to the "stuck-vector" situation when one neuron from the hidden layer may never win. See Figure 5.6: the vector $W_{(2,:)}$ will move towards $\mathbf{x}_{1,2,3,4}$ during learning and will become representative for both classes $\mathcal{C}_1$ and $\mathcal{C}_2$, the corresponding hidden neuron 2 will be a winner for both classes.

  The "stuck vector" situation may be avoided by two means: one is to initialize each weight by a vector belonging to the class for which the corresponding hidden neuron will win — the most representative if possible, e.g. by averaging over the training set. The other is to attach to the neuron an "overloading device": if the neuron wins too often, e.g. during training wins more that the number of training vectors from the class it is supposed to represent, then it will be shutdown allowing other neurons to win and their corresponding weights to be adapted.
- The hidden layer should have *at least* as many neurons as the number of classes to be recognized. At least one neuron is needed to win the "competition" for the class it represents. If classes form unconnected domains in the input space $\mathbf{z} \in \mathbb{R}^{N+K}$ then one neuron at least is necessary for each *connected* domain. Otherwise the "stuck vector" situation is likely to appear. It is also recommended to have several hidden neurons for a class if is not convex (even if it is connected).
- For the output layer the critical point is to select an adequate learning constant c': the learning constant can be chosen small $0 \lesssim c' \ll 1$ at the beginning and increased later when $\mathbf{x}_p - W'_{(1:N,h)(t)}$ decreases (respectively $\mathbf{y}_p - W'_{(N+1:N+K,h)(t)}$ decreases).
- Obviously the hidden layer may be replaced by any other system able to perform a one–of–h encoding, e.g. a SOM network.

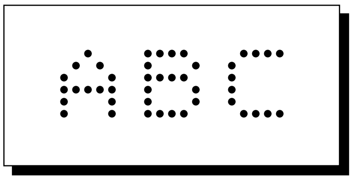

Figure 5.7: *The representative set for letters A, B, C.*

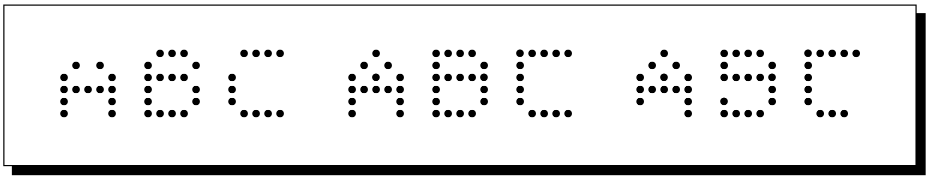

Figure 5.8: *The training set for letters A, B, C.*

5.4 Examples

5.4.1 Letter classification

Given a set of letters as 5×6 binary images, the network aims to correctly associate the ASCII code to the image even if there are missing parts or noise in the image.

The letters are uppercase A, B, C so there are 3 classes and correspondingly 3 neurons on the hidden layer. The representative letters are depicted in Figure 5.7.

The representative $\mathbf{x}$ vectors are created by reading the graphical representation of the characters on rows; a "dot" gets an 1, its absence gets an 0 (30–dimensional vectors):

$$\mathbf{x}_{\text{rep.A}} = \begin{array}{l} (\,0\,0\,1\,0\,0\;, \\ \;\;0\,1\,0\,1\,0\;, \\ \;\;1\,0\,0\,0\,1\;, \\ \;\;1\,1\,1\,1\,1\;, \\ \;\;1\,0\,0\,0\,1\;, \\ \;\;1\,0\,0\,0\,1\;)^{\mathrm{T}} \end{array} , \quad \mathbf{x}_{\text{rep.B}} = \begin{array}{l} (\,1\,1\,1\,1\,0\;, \\ \;\;1\,0\,0\,0\,1\;, \\ \;\;1\,1\,1\,1\,0\;, \\ \;\;1\,0\,0\,0\,1\;, \\ \;\;1\,0\,0\,0\,1\;, \\ \;\;1\,1\,1\,1\,0\;)^{T} \end{array} , \quad \mathbf{x}_{\text{rep.C}} = \begin{array}{l} (\,0\,1\,1\,1\,1\;, \\ \;\;1\,0\,0\,0\,0\;, \\ \;\;1\,0\,0\,0\,0\;, \\ \;\;1\,0\,0\,0\,0\;, \\ \;\;1\,0\,0\,0\,0\;, \\ \;\;0\,1\,1\,1\,1\;)^{\mathrm{T}} \end{array}$$

The ASCII codes are 65 for A, 66 for B and 67 for C. These ASCII codes are used to generate 8–dimensional binary vectors $\mathbf{y}$ corresponding to each letter.

$$\mathbf{y}_A = \left(0\,1\,0\,0\,0\,0\,0\,1\right)^{\mathrm{T}} , \quad \mathbf{y}_B = \left(0\,1\,0\,0\,0\,0\,1\,0\right)^{\mathrm{T}} , \quad \mathbf{y}_C = \left(0\,1\,0\,0\,0\,0\,1\,1\right)^{\mathrm{T}}$$

The training letters are depicted in Figure 5.8 — the first set contains a "dot" less (information missing), the second one contains a supplementary "dot" (noise added) while the third set contains both.

Training[5] was done just once for the training set (one epoch because the weights have been initialized to good representatives) with the constants: $c = 0.1$ and $c' = 0.1$.

[5] The actual Scilab code is in the source tree of this book, in directory, e.g. `Matrix_ANN_1/Scilab/CPN_alpha.sci`, note that actual version numbers may differ.

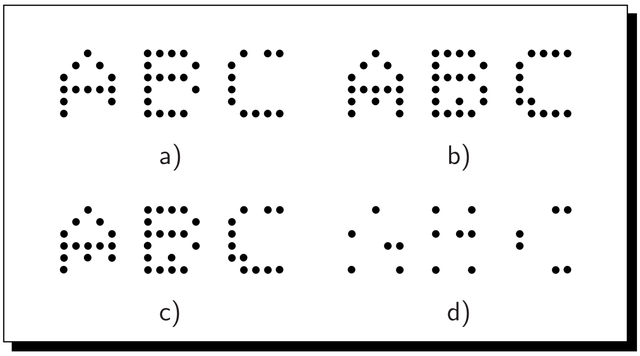

Figure 5.9: *The test set for letters A, B, C.*

The test letters are depicted in Figure 5.9 — the first 3 sets are similar, but not identical, to the training sets and they where not used in training. The system is able to recognize correctly even the fourth set (labeled d) from which a large amount of information is missing.

Remarks:

➥ The conversion of training and test sets to binary $\mathbf{x}$ vectors is done into a similar ways as for the representative set.

➥ If a large part of the information is missing then the system may miss–classify the letters due to the fact that there are "dots" in common positions (especially for letters B and C).

CHAPTER 6

Adaptive Resonance Theory

The ART networks are unsupervised classifiers, composed of several sub–systems and able to resume learning at a later stage *without* restarting from scratch. ART

6.1 ART1 Architecture

The ART1 network is made from 2 main layers of neurons: F_1 and F_2, a gain control neuron and a reset neuron. See Figure 6.1 on the next page. The ART1 network works only with binary vectors. ART1 ❖ F_1, F_2

The 2/3 rule: The neurons from F_1 layer receive inputs from 3 sources: input, gain control and F_2. The neurons from F_2 layer also receive inputs from 3 sources: F_1, gain control and reset unit. Both layers F_1 and F_2 are built such that they become active if and only if 2 out of 3 input sources are active (an input vector is active when it has at least one non–zero component). 2/3 rule

Propagation of signals through ART1 network is performed as follows:

① The input vector $\mathbf{x}$ is distributed to F_1 layer, gain control and reset neurons; each component of $\mathbf{x}$ is distributed to a different F_1 neuron — F_1 has the same dimension N as the input $\mathbf{x}$.

② The F_1 output is sent as inhibition to the reset neuron. The design of the network is such that the inhibitory signal from F_1 cancels the input $\mathbf{x}$ and the reset neuron remains inactive.

③ The gain control neuron sends a nonspecific excitation to F_1 (an identical signal to all neurons from F_1).

[6.1] For the whole ART1 topic see [CG87b], see [FS92] pp. 293–298.

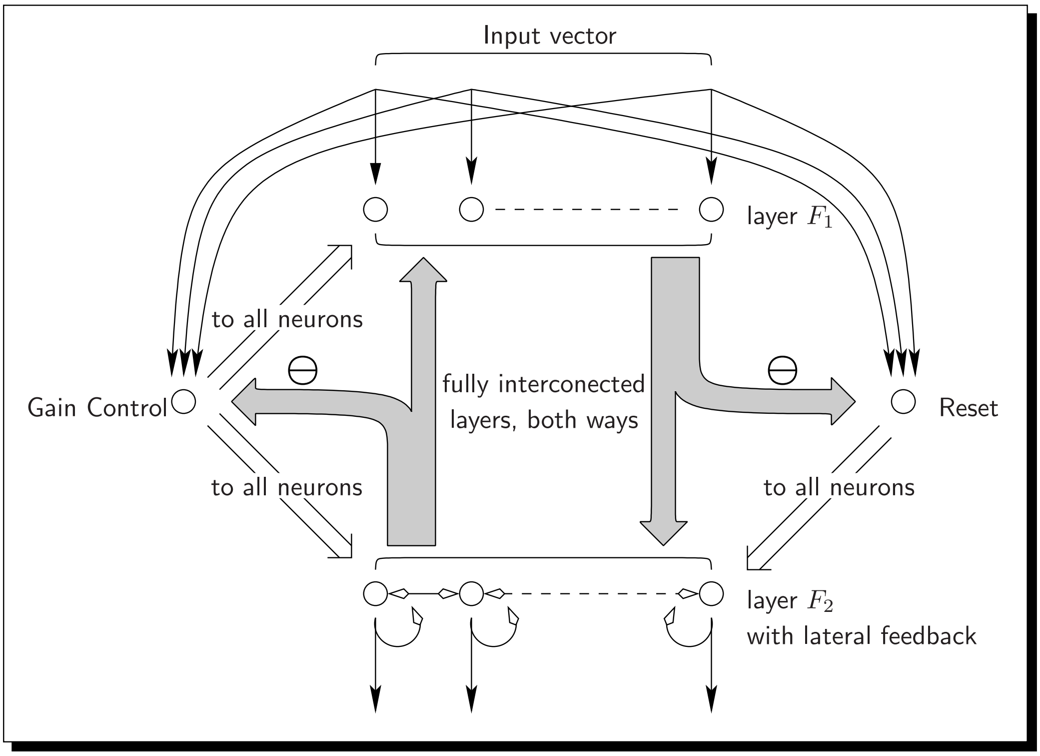

Figure 6.1: *ART1 network architecture; inhibitory input is marked with $\ominus$ symbol.*

competitive layer

④ F_2 receives the output of F_1 (all neurons between F_1 and F_2 are fully interconnected). The F_2 layer is of competitive type: only one neuron should trigger for a given pattern. The size of F_2 is K (defining the whole network classification capacity, i.e. the total number of different classes to learn now *and in the future*).

⑤ The F_2 output is sent back as excitation to F_1 and as inhibition to the gain control neuron. The design of the network is such that if the gain control neuron receives an inhibition from F_2 it ceases activity.

⑥ Then F_1 receives signals from F_2 and input (the gain control has been deactivated). The output of F_1 layer changes such that it isn't anymore identical to the first one, because the overall input had changed: the gain control ceases activity and (instead) the layer F_2 sends its output to F_1. Also there is the *2/3 rule* which has to be considered: only those F_1 neurons who receive input from both input and F_2 will trigger. Because the output of F_1 had changed, the reset neuron becomes active.

⑦ The reset neuron sends a reset signal to the active neuron from F_2 which forces it to become inactive for a *long* period of time, i.e. it does not participate in the next network pattern matching cycle(s) as long as input $\mathbf{x}$ does not change. The inactive neurons are not affected.

⑧ The output of F_2 disappears due to the reset action and the whole cycle is repeated until a match is found, i.e. the output of F_2 causes F_1 to output a pattern which will not trigger a reset because it is identical or very similar to the input, or no match was found and a learning process begins in F_2.

The action of the reset neuron (see previous step) ensures that a neuron already used in the "past" will not be used again for the current pattern matching, i.e. as long as current input vector does not change.

The neurons who have learned in the past are called *committed* (they have been committed to a class of inputs). committed neurons

The network design is such that:

- all committed neurons have a chance to win before any uncommitted ones (to see if the input belongs to the "their" class);
- when a previously uncommitted neuron is used to learn a new input, the reset neuron will not trigger (uncommitted neurons do not change the output of F_1 if they are winners).

Remarks:

➥ In complex systems an ART network may be just a link into a bigger chain. The F_2 layer may receive signals from some other networks/layers. This will make F_2 send a signal to F_1 and, consequently F_1 may receive a signal from F_2 *before* receiving the input signal.

A premature signal from F_2 layer usually means an *expectation*. If the gain control system and consequently the 2/3 rule would not have been in place then the *expectation* from F_2 would have triggered an action *in absence of the input signal*. expectation

In the presence of the gain control neuron F_2 can't trigger a process by itself. However it can precondition the F_1 layer such that when the input arrives the process of pattern matching will start at a position closer to the final state and the process takes less time.

6.2 ART1 Dynamics

For both layers F_1 and F_2, the neuronal dynamics are described by the "passive decay with resting potential" equation:

$$\frac{da}{dt} = -a + (1 - \alpha a) \times \text{excitatory input} - (\beta + \gamma a) \times \text{inhibitory input} \tag{6.1}$$

where $\alpha, \beta, \gamma = \text{const.}$, $\alpha, \beta, \gamma > 0$ ❖ α, β, γ

6.2.1 The F_1 layer

A $F_1 : i$ neuron receives input x_i, an input from F_2 and *excitatory* input from the gain control neuron. The *inhibitory* input is set to 1. See (6.1).

Let $\mathbf{a}_{(2)}$ be the activation (total input) of F_2 layer, $f_2(\mathbf{a}_{(2)})$ output vector of F_2 (f_2 being the activation function on F_2) and $W_{(1)}$ the matrix of weights associated with connections ❖ $\mathbf{a}_{(2)}$, f_2 ❖ $W_{(1)}$, $w_{(1)ik}$

[6.2] See [FS92] pp. 298–310.

$F_2 \to F_1$ — it will be shown later that these weights have to be $w_{(1)ik} \in [0, 1]$. The total input received by an $F_1 : i$ neuron, from F_2, is $W_{(1)(i,:)} f_2(\mathbf{a}_{(2)})$, but as F_2 is competitive, there is only one $F_2 : k$ neuron with non–zero output (the "winner") and then each $F_1 : i$ neuron receives, as input from F_2, the value w_{ik} (the winner's output is 1, ART1 works with binary vectors).

❖ g

The gain control neuron is set such that if its input is $\mathbf{x} \neq \widehat{\mathbf{0}}$ and F_2 output is $f_2(\mathbf{a}_{(2)}) = \widehat{\mathbf{0}}$ then its output is 1, otherwise it is 0:

$$g = \begin{cases} 1 , & \text{if } \mathbf{x} \neq \widehat{\mathbf{0}} \text{ and } f_2(\mathbf{a}_{(2)}) = \widehat{\mathbf{0}} , \\ 0 , & \text{otherwise.} \end{cases}$$

❖ $\mathbf{a}_{(1)}$

Finally the dynamic equation for the F_1 layer becomes (from (6.1), $\mathbf{a}_{(1)}$ is total input on F_1):

$$\frac{d\mathbf{a}_{(1)}}{dt} = -\mathbf{a}_{(1)} + \left(\widehat{\mathbf{1}} - \alpha_1 \mathbf{a}_{(1)}\right) \odot \left[\mathbf{x} + \eta_1 W_{(1)} f_2(\mathbf{a}_{(2)}) + \beta_1 g \widehat{\mathbf{1}}\right] - \left(\beta_1 \widehat{\mathbf{1}} + \gamma_1 \mathbf{a}_{(1)}\right) \quad (6.2)$$

❖ α_1, β_1, γ_1, η_1

where the constants α, β, γ and η have been given the subscript 1 to denote that they are for F_1 layer. Here $\eta_1 > 0$ is a constant controlling the amplitude of $W_{(1)}$ weights, it should be chosen such that all weights $w_{(1)ik} \in [0, 1]$, it will be shown later why and how. Note that β_1 represents also the weight of connection from gain control to F_1.

In order to allow for the *2/3 rule* to work, there are some conditions to be imposed over constants and weights. To see them, the following cases are discussed:

① Input is inactive ($\mathbf{x} = \widehat{\mathbf{0}}$) and F_2 is inactive ($f_2(\mathbf{a}_{(2)}) = \widehat{\mathbf{0}}$). Then $g = 0$ and (6.2) becomes:

$$\frac{d\mathbf{a}_{(1)}}{dt} = -\mathbf{a}_{(1)} - \left(\beta_1 \widehat{\mathbf{1}} + \gamma_1 \mathbf{a}_{(1)}\right) .$$

At equilibrium $\frac{d\mathbf{a}_{(1)}}{dt} = \widehat{\mathbf{0}}$ and $\mathbf{a}_{(1)} = -\frac{\beta_1 \widehat{\mathbf{1}}}{1+\gamma_1}$, i.e. *inactive F_1 neurons have negative activation* ($a_{(1)i} < 0$).

② Input is active ($\mathbf{x} \neq \widehat{\mathbf{0}}$) but F_2 is still inactive ($f_2(\mathbf{a}_{(2)}) = \widehat{\mathbf{0}}$) — there was no time for the signal to travel from F_1 to F_2 and back (and deactivating the gain control neuron on the way back). The gain control is activated: $g = 1$ and (6.2) becomes:

$$\frac{d\mathbf{a}_{(1)}}{dt} = -\mathbf{a}_{(1)} + \left(\widehat{\mathbf{1}} - \alpha_1 \mathbf{a}_{(1)}\right) \odot \left(\mathbf{x} + \beta_1 \widehat{\mathbf{1}}\right) - \left(\beta_1 \widehat{\mathbf{1}} + \gamma_1 \mathbf{a}_{(1)}\right) .$$

At equilibrium $\frac{d\mathbf{a}_{(1)}}{dt} = \widehat{\mathbf{0}}$ and:

$$\mathbf{a}_{(1)} = \mathbf{x} \oslash \left[(1 + \alpha_1 \beta_1 + \gamma_1)\widehat{\mathbf{1}} + \alpha_1 \mathbf{x}\right] , \quad (6.3)$$

i.e. neurons which received non–zero input ($x_i \neq 0$) have positive activation ($a_{(1)i} > 0$) and the neurons which received a zero input have their activation *raised* to zero.

③ Input is active ($\mathbf{x} \neq \widehat{\mathbf{0}}$) and F_2 is also active ($f_2(\mathbf{a}') \neq \widehat{\mathbf{0}}$). Then the gain control is deactivated ($g = 0$) and (6.2) becomes:

$$\frac{d\mathbf{a}_{(1)}}{dt} = -\mathbf{a}_{(1)} + \left(\widehat{\mathbf{1}} - \alpha_1 \mathbf{a}_{(1)}\right) \odot \left[\mathbf{x} + \eta_1 W_{(1)} f_2(\mathbf{a}_{(2)})\right] - \left(\beta_1 \widehat{\mathbf{1}} + \gamma_1 \mathbf{a}_{(1)}\right) .$$

At equilibrium $\frac{d\mathbf{a}_{(1)}}{dt} = \widehat{\mathbf{0}}$ and:

$$\mathbf{a}_{(1)} = \left[\mathbf{x} + \eta_1 W_{(1)} f_2(\mathbf{a}_{(2)}) - \beta_1 \widehat{\mathbf{1}}\right] \oslash \left[(1+\gamma_1)\widehat{\mathbf{1}} + \alpha_1 \left(\mathbf{x} + \eta_1 W_{(1)} f_2(\mathbf{a}_{(2)})\right)\right] \quad (6.4)$$

The following cases may be discussed here:

(a) Input is maximum: $x_i = 1$ and input from F_2 is minimum: $\mathbf{a}_{(2)} \to \widehat{\mathbf{0}}$. Because the gain control unit has been deactivated and the activity of the F_2 layer is dropping to zero then (according to the *2/3 rule*) the neuron has to switch to inactive state and consequently a_i has to switch to a negative value. From (6.4):

$$\lim_{\mathbf{a}_{(2)} \to \widehat{\mathbf{0}}} a_{(1)i} = \lim_{\mathbf{a}_{(2)} \to \widehat{\mathbf{0}}} \left[\frac{x_i + \eta_1 W_{(1)(i,:)} f_2(\mathbf{a}_{(2)}) - \beta_1}{1 + \gamma_1 + \alpha_1 (x_i + \eta_1 W_{(1)(i,:)} f_2(\mathbf{a}_{(2)}))}\right] = \frac{1 - \beta_1}{1 + \gamma_1 + \alpha_1} < 0$$

$$\Rightarrow \quad \beta_1 > 1 \quad (6.5)$$

(all constants α_1, β_1, γ_1, η_1 are positive by construction).

(b) Input is maximum: $x_i = 1$ and input from F_2 is non–zero. F_2 layer is of a contrast enhancement type ("winner takes all") and it has only (maximum) one winner, let k be that one ($f_2(a_{(2)k}) = 1$), i.e. $W_{(1)(i,:)}\, f_2(\mathbf{a}_{(2)}) = w_{(1)ik}$. Then according to the *2/3 rule* the neuron is active and the activation value should be $a_{(1)i} > 0$ and (6.4) gives (as above):

$$1 + \eta_1 w_{(1)ik} - \beta_1 > 0 \quad \Rightarrow \quad w_{(1)ik} > \frac{\beta_1 - 1}{\eta_1} \; . \quad (6.6)$$

There is a discontinuity between this condition and the preceding one: from (6.6), if $w_{ik} \to 0$ then $\beta_1 - 1 < 0$ which seems to be in contradiction with the previous (6.5) condition. Consequently this condition will be imposed on $W_{(1)}$ weights and *not* on constants β_1 and η_1.

(c) Input is maximum $x_i = 1$ and input from F_2 is maximum, i.e. $w_{(1)ik} = 1$ (because $w_{(1)ik} \in [0, 1]$, also see above, k is the F_2 winning neuron). Then (6.4) gives:

$$1 + \eta_1 - \beta_1 > 0 \quad \Rightarrow \quad \beta_1 < \eta_1 + 1 \quad (6.7)$$

(d) Input is minimum $\mathbf{x} \to \widehat{\mathbf{0}}$ and input from F_2 is maximum. Similarly to the previous case above, from (6.4):

$$\eta_1 - \beta_1 < 0 \quad \Rightarrow \quad \eta_1 < \beta_1 \quad (6.8)$$

(because of the *2/3 rule* at the limit the F_1 neuron has to switch to negative state and subsequently has a negative activation).

(e) Input is minimum ($\mathbf{x} \to \widehat{\mathbf{0}}$) and input from F_2 is also minimum ($\mathbf{a}_{(2)} \to \widehat{\mathbf{0}}$). Similar to the above cases the F_1 neuron turns to inactive state, so it will have a negative activation and (6.4) (on similar premises as above) gives:

$$\lim_{\substack{\mathbf{x} \to \widehat{\mathbf{0}} \\ \mathbf{a}_{(2)} \to \widehat{\mathbf{0}}}} a_i < 0 \quad \Rightarrow \quad -\beta_1 < 0$$

which is satisfied automatically as $\beta_1 > 0$.

Combining all the requirements (6.5), (6.7) and (6.8) in one gives:

$$\max(1, \eta_1) < \beta_1 < \eta_1 + 1 \tag{6.9}$$

which represents one of the conditions to be put on F_1 constants such that the *2/3 rule* will operate ((6.6) will be imposed on the weights).

❖ f_1

The output value for the $F_1 : i$ neuron is obtained by applying the threshold activation:

$$f_1(a_{(1)i}) = \begin{cases} 1 , & \text{if } a_{(1)i} > 0 , \\ 0 , & \text{if } a_{(1)i} \leqslant 0 . \end{cases} \tag{6.10}$$

Note that when an input has been applied and *before* receiving any input from F_2, the output of F_1 is $f_1(\mathbf{a}_{(1)}) = \mathbf{x}$ (see (6.3), this property will be used in next sections).

6.2.2 The F_2 layer

① Initially the network is started at a "zero" state. There are no signals traveling internally and no input ($\mathbf{x} = \widehat{\mathbf{0}}$). So, the output of the gain control unit is 0. Even if the F_2 layer receives a direct input from the outside environment (another network, etc.) the *2/3 rule* stops F_2 from sending any output.

② Once an input has arrived on F_1 the output of the gain control neuron switches to 1, because the output of F_2 is still $\widehat{\mathbf{0}}$.

Now the output of F_2 is allowed. There are two cases:

expectation

(a) There is already an input from outside, then the F_2 layer will output immediately *without waiting for the input from* F_1 – see the remarks about *expectation* in Section 6.1.

(b) If there is no external input then the F_2 layer has to wait for the output of F_1 before being able to send an output.

The conclusion is that the gain control is used just to turn on/off the right of F_2 to send an output. Because the output of the gain control neuron (i.e. 1) is sent uniformly to all neurons it doesn't play any other active role and can be left out of the equations describing the behavior of F_2 units. So in fact the equation describing the activation of the F_2 neuron is of the form:

$$\begin{array}{l} F_2 \text{ output} \\ \text{equation} \end{array} : \begin{cases} \frac{da_{(2)k}}{dt} = -\, a_{(2)k} + (1 - \alpha_2 a_{(2)k}) \times \text{excitatory input} & \text{if } g = 1 , \\ \qquad\qquad -\, (\beta_2 + \gamma_2 a_{(2)k}) \times \text{inhibitory input} & \\ a_{(2)k} = 0 , & \text{othewise.} \end{cases}$$

❖ $\mathbf{a}_{(2)}$, α_2, β_2, γ_2

where $a_{(2)k}$ is the neuron activation (total input); $\alpha_2, \beta_2, \gamma_2 =$ const. and $\alpha_2, \beta_2, \gamma_2 > 0$, (see (6.1)). The $g = 1$ case is analyzed below.

❖ h

The F_2 layer is of competitive type: only one neuron "wins". In order to achieve this, a feedback function $h(\mathbf{a}_{(2)})$ is required. The feedback is tuned so as to be positive on autofeedback (on the neuron itself) and negative on other neurons.

❖ $W_{(2)}$

The neuron $F_2 : k$ receives an *excitatory* input from F_1: $W_{(2)(k,:)} f_1(\mathbf{a}_{(1)})$, where $W_{(2)}$ is the weight matrix of connections $F_1 \to F_2$, and from itself: $h(a_{(2)k})$:

$$\text{excitatory input} = W_{(2)(k,:)} f_1(\mathbf{a}_{(1)}) + h(a_{(2)k})$$

The same $F_2 : k$ neuron receives a *direct inhibitory* input from all other $F_2 : \ell$ neurons: $\sum\limits_{\ell \neq k} h(a_{(2)\ell})$ and an *indirect inhibitory* input: $\sum\limits_{\ell \neq k} W_{(2)(\ell,:)} f_1(\mathbf{a}_{(1)})$:

$$\text{inhibitory input} = \sum_{\ell=1,\ell\neq k}^{K} h(a_{(2)\ell}) \quad + \sum_{\ell=1,\ell\neq k}^{K} W_{(2)(\ell,:)} f_1(\mathbf{a}_{(1)})$$

The second term represents the indirect inhibition (feedback) due to the fact that other neurons will have an positive output (because of their input), while the first one is due to *direct* inter–connections (lateral feedback) between neurons in F_2 layer.

Eventually, from (6.1):

$$\begin{aligned}\frac{da_{(2)k}}{dt} = & - a_{(2)k} + (1 - \alpha_2 a_{(2)k})[\eta_2 W_{(2)(k,:)} f_1(\mathbf{a}_{(1)}) + h(a_{(2)k})] \\ & - (\beta_2 + \gamma_2 a_{(2)k}) \sum_{\ell=1,\ell\neq k}^{K} \left[h(a_{(2)\ell}) + W_{(2)(\ell,:)} f_1(\mathbf{a}_{(1)})\right]\end{aligned}$$

where $\alpha_2, \beta_2, \gamma_2, \eta_2 =$ const. Let $\beta_2 = 0$, $\gamma_2 = \alpha_2$ and $\eta_2 = 1$, then: ❖ α_2, β_2, γ_2, η_2

$$\begin{aligned}\frac{da_{(2)k}}{dt} = & - a_{(2)k} + h(a_{(2)k}) + W_{(2)(k,:)} f_1(\mathbf{a}_{(1)}) \\ & - \alpha_2 a_{(2)k} \widehat{\mathbf{1}}^{\mathrm{T}} \left[h(\mathbf{a}_{(2)}) + W_{(2)} f_1(\mathbf{a}_{(1)})\right] .\end{aligned} \tag{6.11}$$

or in matrix notation:

$$\frac{d\mathbf{a}_{(2)}}{dt} = -\mathbf{a}_{(2)} + h(\mathbf{a}_{(2)}) + W_{(2)} f_1(\mathbf{a}_{(1)}) - \alpha_2 \mathbf{a}_{(2)} \widehat{\mathbf{1}}^{\mathrm{T}} \left[h(\mathbf{a}_{(2)}) + W_{(2)} f_1(\mathbf{a}_{(1)})\right] . \tag{6.12}$$

For an feedback function of the form $h(\mathbf{a}_{(2)}) = \mathbf{a}_{(2)}^{\odot n}$, where $n > 1$, the above equation defines a contrast–enhancement/competitive layer where one or more neurons maximize their output while all others minimize it. ❖ n

Discussion. First the following *change of variable* is performed: $a_{(2)\text{tot}} = \widehat{\mathbf{1}}^{\mathrm{T}} \mathbf{a}_{(2)}$ and $\widetilde{\mathbf{a}}_{(2)} = \frac{\mathbf{a}_{(2)}}{a_{(2)\text{tot}}}$; then $\mathbf{a}_{(2)} = \widetilde{\mathbf{a}}_{(2)} a_{(2)\text{tot}}$ and $\frac{d\mathbf{a}_{(2)}}{dt} = \frac{d\widetilde{\mathbf{a}}_{(2)}}{dt} a_{(2)\text{tot}} + \widetilde{\mathbf{a}}_{(2)} \frac{da_{(2)\text{tot}}}{dt}$ ❖ $\widetilde{\mathbf{a}}_{(2)}$, $a_{(2)\text{tot}}$

By making a sum over k in (6.11) and using the change of variable just defined:

$$\begin{aligned}\frac{da_{(2)\text{tot}}}{dt} = & - a_{(2)\text{tot}} + \widetilde{\mathbf{1}}^{\mathrm{T}} h(\widetilde{\mathbf{a}}_{(2)} a_{(2)\text{tot}}) + \widetilde{\mathbf{1}}^{\mathrm{T}} W_{(2)} f_1(\mathbf{a}_{(1)}) \\ & - \alpha_2 a_{(2)\text{tot}} \widetilde{\mathbf{1}}^{\mathrm{T}} \left[h(\widetilde{\mathbf{a}}_{(2)} a_{(2)\text{tot}}) + W_{(2)} f_1(\mathbf{a}_{(1)})\right] .\end{aligned} \tag{6.13}$$

As established: $\frac{d\widetilde{\mathbf{a}}_{(2)}}{dt} a_{(2)\text{tot}} = \frac{d\mathbf{a}_{(2)}}{dt} - \widetilde{\mathbf{a}}_{(2)} \frac{da_{(2)\text{tot}}}{dt}$ and from (6.12) and (6.13):

$$\frac{d\widetilde{\mathbf{a}}_{(2)}}{dt} a_{(2)\text{tot}} = h(\widetilde{\mathbf{a}}_{(2)} a_{(2)\text{tot}}) + W_{(2)} f_1(\mathbf{a}_{(1)}) - \widetilde{\mathbf{a}}_{(2)} \widetilde{\mathbf{1}}^{\mathrm{T}} \left[h(\widetilde{\mathbf{a}}_{(2)} a_{(2)\text{tot}}) + W_{(2)} f_1(\mathbf{a}_{(1)})\right] .$$

The following cases with respect to feedback function may be discussed:

- $n = 1$, identity function: $h(\widetilde{\mathbf{a}}_{(2)} a_{(2)\text{tot}}) = \widetilde{\mathbf{a}}_{(2)} a_{(2)\text{tot}}$. Then, by using $\widehat{\mathbf{1}}^{\mathrm{T}} \widetilde{\mathbf{a}}_{(2)} = 1$, the above equation becomes:

  $$\frac{d\widetilde{\mathbf{a}}_{(2)}}{dt} a_{(2)\text{tot}} = W_{(2)} f_1(\mathbf{a}_{(1)}) - \widetilde{\mathbf{a}}_{(2)} \widetilde{\mathbf{1}}^{\mathrm{T}} W_{(2)} f_1(\mathbf{a}_{(1)}) .$$

  The stable value, obtained by setting $\frac{d\widetilde{\mathbf{a}}_{(2)}}{dt} a_{(2)\text{tot}} = \widehat{\mathbf{0}}$, is $\widetilde{\mathbf{a}}_{(2)} = \frac{W_{(2)} f_1(\mathbf{a}_{(1)})}{\widetilde{\mathbf{1}}^{\mathrm{T}} W_{(2)} f_1(\mathbf{a}_{(1)})}$.

- $n = 2$, square function: $h(\widetilde{\mathbf{a}}_{(2)} a_{(2)\mathrm{tot}}) = (\widetilde{\mathbf{a}}_{(2)} a_{(2)\mathrm{tot}})^{\odot 2}$. Then, by using again $\widehat{\mathbf{1}}^{\mathrm{T}} \widetilde{\mathbf{a}}_{(2)} = 1$ and writing the equations in a component form:

$$\frac{d\widetilde{a}_{(2)k}}{dt} a_{(2)\mathrm{tot}} = h(\widetilde{a}_{(2)k} a_{(2)\mathrm{tot}}) + W_{(2)(k,:)} f_1(\mathbf{a}_{(1)}) - \widetilde{a}_{(2)k} \widetilde{\mathbf{1}}^{\mathrm{T}} [h(\widetilde{\mathbf{a}}_{(2)} a_{(2)\mathrm{tot}}) + W_{(2)} f_1(\mathbf{a}_{(1)})]$$
$$= \widetilde{a}_{(2)k} a_{(2)\mathrm{tot}} \sum_{\ell=1}^{K} \widetilde{a}_{(2)\ell} \left[\frac{h(\widetilde{a}_{(2)k} a_{(2)\mathrm{tot}})}{\widetilde{a}_{(2)k} a_{(2)\mathrm{tot}}} - \frac{h(\widetilde{a}_{(2)\ell} a_{(2)\mathrm{tot}})}{\widetilde{a}_{(2)\ell} a_{(2)\mathrm{tot}}} \right]$$
$$+ W_{(2)(k,:)} f_1(\mathbf{a}_{(1)}) - \widetilde{a}_{(2)k} \widetilde{\mathbf{1}}^{\mathrm{T}} W_{(2)} f_1(\mathbf{a}_{(1)}) \ ,$$

and then, considering the expression of h, the term $\left[\frac{h(\widetilde{a}_{(2)k} a_{(2)\mathrm{tot}})}{\widetilde{a}_{(2)k} a_{(2)\mathrm{tot}}} - \frac{h(\widetilde{a}_{(2)\ell} a_{(2)\mathrm{tot}})}{\widetilde{a}_{(2)\ell} a_{(2)\mathrm{tot}}} \right]$ reduces to $a_{(2)\mathrm{tot}}(\widetilde{a}_{(2)k} - \widetilde{a}_{(2)\ell})$, representing an amplification for $\widetilde{a}_{(2)k} > \widetilde{a}_{(2)\ell}$, while for $\widetilde{a}_{(2)k} < \widetilde{a}_{(2)\ell}$ it is an inhibition. The term $+W_{(2)(k,:)} f_1(\mathbf{a}_{(1)})$ is a constant with respect to $\widetilde{\mathbf{a}}_{(2)}$; term $-\widetilde{a}_{(2)k} \widetilde{\mathbf{1}}^{\mathrm{T}} W_{(2)} f_1(\mathbf{a}_{(1)})$ represents an inhibition.

The differential equation describes the behaviour of a network where all neurons inhibit all others with smaller output and are inhibited by neurons with larger output. The gap between neurons with large respectively small output increases, eventually only one/few neuron(s) will have non–zero output (the "winner(s)"). The layer is of contrast–enhancement/competitive type.

The general case $n > 1$ is similar to case $n = 2$.

❖ f_2

The winning F_2 neuron sends a value of 1 to F_1 layer, all others send 0. Let $f_2(\mathbf{a}_{(2)})$ be the activation function of F_2 neurons:

$$f_2(a_{(2)k}) = \begin{cases} 1 & \text{if } a_{(2)k} = \max\limits_{\ell=\overline{1,K}} a_{(2)\ell} \\ 0 & \text{otherwise} \end{cases} \tag{6.14}$$

Remarks:

➥ It seems at first sight that the F_2 neuron has *two* outputs: one sent to the F_2 layer via h function and one sent to the F_1 layer via f_2 function. This runs counter to the definition of a neuron: it should have just *one* output. However this contradiction may be overcome if the neuron is replaced with an ensemble of three neurons: the main one which calculates the activation $a_{(2)k}$ and sends the result (it has the identity function as activation) to two others which receive its output (they have one input with weight 1), apply the h respectively f_2 functions and send their output wherever is required. See Figure 6.2 on the facing page.

➥ Note that after stabilisation on F_2 (i.e. for $t \to \infty$) $f_2(\mathbf{a}_{(2)}) = \mathbf{a}_{(2)}$.

6.2.3 Learning on F_1

The differential equations describing the F_1 learning process, (i.e. $W_{(1)}$ weights adaptation) are defined as:

$$\frac{dw_{(1)ik}}{dt} = \left[-w_{(1)ik} + f_1(a_{(1)i}) \right] f_2(a_{(2)k}) \ . \tag{6.15}$$

Note that there is only one "winner", let k be that one, in F_2 for which $f_2(a_{(2)k}) \neq 0$. For all others $f_2(a_{(2)\ell}) = 0$, i.e. only the weights related to the winning F_2 neuron are adapted on F_1 layer, all others remain unchanged for a given input (only column k of $W_{(1)}$ may change).

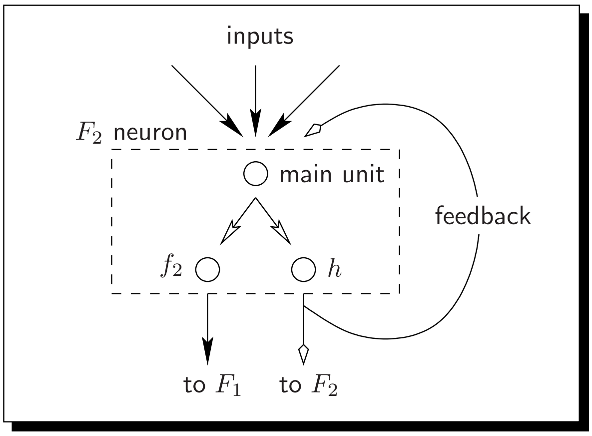

Figure 6.2: *The F_2 neuronal structure.*

Because of the definition of the f_1 and f_2 functions (see (6.10) and (6.14)) the following cases may be considered:

① $F_2 : k$ neuron winner $\left(f_2(a_{(2)k}) = 1\right)$ and $F_1{:}i$ neuron active $\left(f_1(a_{(1)i}) = 1\right)$, then:

$$\frac{dw_{(1)ik}}{dt} = -w_{(1)ik} + 1 \quad \text{thus} \quad w_{(1)ik} = 1 - e^{-t} ,$$

where the condition that the weight approaches 1 as $t \to \infty$ has been used to determine the constant[1].

② $F_2 : k$ neuron winner $\left(f_2(a_{(2)k}) = 1\right)$ and $F_1{:}i$ neuron non-active $\left(f_1(a_{(1)i}) = 0\right)$, then:

$$\frac{dw_{(1)ik}}{dt} = -w_{(1)ik} \quad \Rightarrow \quad w_{(1)ik} = w_{(1)ik(0)} e^{-t} ,$$

where $w_{(1)ik(0)}$ is the initial value at $t = 0$. The weight asymptotically decreases to 0 for $t \to \infty$.

③ $F_2 : k$ neuron *non*–winner $\left(f_2(a_{(2)k}) = 0\right)$, then:

$$\frac{dw_{(1)ik}}{dt} = 0 \quad \Rightarrow \quad w_{(1)ik} = \text{const.}$$

the weights do not change.

A supplementary condition for $W_{(1)}$ weights was previously found in (6.6):

$$w_{(1)ik} > \frac{\beta_1 - 1}{\eta_1} , \tag{6.16}$$

i.e. all weights have to be initialized to a value greater than $\frac{\beta_1 - 1}{\eta_1}$. Otherwise the $F_1 : i$ neuron is kept into an inactive state and the weights decrease to 0 (or do not change).

From the first case discussed above, the maximal $w_{(1)ik}$ value is 1; from the second case, the minimal $w_{(1)ik}$ value is 0; from (6.16) and (6.9) weights are initialized with positive

[1] The general solution is $w_{(1)ik} = C(1 - e^{-t})$, where $C =$ const. This solution is found by searching first for the solution of the homogeneous equation and then by using the "variation of parameters" method.

values smaller than 1; the conclusion is that $W_{(1)}$ weights always have values within the interval $[0, 1]$.

Asymptotic Learning

If the $F_2 : k$ and $F_1 : i$ neurons are both active then the weight $w_{ik} \to 1$, otherwise it decays towards 0 or remains unchanged. A fast way to achieve the learning is to set the weights to their asymptotic values as soon is possible, i.e. knowing the neuronal activities set:

$$w_{(1)ik} = \begin{cases} 1 & \text{if } i, k \text{ neurons are active} \\ \text{no change} & \text{if } k \text{ neuron is non–active} \\ 0 & \text{otherwise} \end{cases} \tag{6.17}$$

or in matrix notation:

$$W_{(1)(:,k)\text{new}} = \mathbf{x} \tag{6.18}$$

and all other weights are left unchanged.

Proof. Only column k of $W_{(1)}$ is to be changed (weights related to the F_2 winner). $f_1(\mathbf{a}_{(1)})$ returns 1 for active $F_1 : i$ neurons, 0 otherwise, see (6.10), then $f_1(\mathbf{a}_1) = \mathbf{x}$. ☐

6.2.4 Learning on F_2

The differential equations describing the F_2 learning process (i.e. $W_{(2)}$ weights adaptation) are defined as:

$$\frac{dw_{(2)ki}}{dt} = c\left[d(1 - w_{(2)ki})f_1(a_{(1)i}) - w_{(2)ki} \sum_{j=1, j\neq i}^{N} f_1(a_{(1)j})\right] f_2(a_{(2)k}) \ ,$$

❖ c, d

where $c, d =$ const. and, because $\sum\limits_{j=1,j\neq i}^{N} f_1(a_{(1)j}) = \sum\limits_{j=1}^{N} f_1(a_{(1)j}) - f_1(a_{(1)i})$, the equations may be rewritten as:

$$\frac{dw_{(2)ki}}{dt} = c\left[d(1 - w_{(2)ki})f_1(a_{(1)i}) - w_{(2)ki}\left(\widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)}) - f_1(a_{(1)i})\right)\right] f_2(a_{(2)k}) \ .$$

For all neurons $F_2 : \ell$, except the winner k, $f_2(a_{(2)\ell}) = 0$, so only the winner's weights are adapted, all others remain unchanged for a given input (only row k of $W_{(2)}$ may change).

Analogous to $W_{(1)}$ weights (see also (6.10) and (6.14)), the following cases are discussed:

① $F_2 : k$ neuron winner ($f_2(a_{(2)k}) = 1$) and $F_1 : i$ neuron active ($f_1(a_{(1)i}) = 1$) then:

$$\frac{dw_{(2)ki}}{dt} = c\left[d - w_{(2)ki}\left(d - 1 + \widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)})\right)\right] \ .$$

With the initial condition $w_{(2)ki(0)} = 0$ at $t = 0$, the solution (same as for $W_{(1)}$) is:

$$w_{(2)ki} = \frac{d}{d - 1 + \widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)})} \left\{1 - \exp\left[-c\left(d - 1 + \widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)})\right) t\right]\right\} \ .$$

The weight asymptotically approaches $\frac{d}{d-1+\widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)})}$ for $t \to \infty$.

In extreme cases it may be possible that $\widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)}) = 0$, then the condition $d > 1$ has to be imposed, otherwise the argument of $\exp$ may become positive and weights may grow indefinitely.

② $F_2 : k$ neuron winner $(f_2(a_{(2)k}) = 1)$ and $F_1 : i$ neuron non–active $(f_1(a_{(1)i}) = 0)$, then:

$$\frac{dw_{(2)ki}}{dt} = -cw_{(2)ki}\widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)}) \quad \Rightarrow \quad w_{(2)ki} = w_{(2)ki(0)} \exp\left(-ct\widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)})\right) ,$$

where $w_{(2)ki(0)}$ is the initial value at $t = 0$. The weight asymptotically decreases to 0 as $t \to \infty$.

③ $F_2 : k$ neuron *non*–winner $(f_2(a_{(2)k}) = 0)$, then:

$$\frac{dw_{(2)ki}}{dt} = 0 \quad \Rightarrow \quad w_{(2)ki} = \text{const.}$$

i.e. the weights associated with any non–winner remain unchanged.

Asymptotic learning

If the $F_2 : k$ and $F_1 : i$ neurons are both active then the weight $w_{(2)ki} \to \frac{d}{d-1+\widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)})}$, otherwise it decays towards 0 or remains unchanged. A fast way to achieve the learning is to set the weights to their asymptotic values as soon is possible, i.e. knowing the neuronal activities set:

$$w_{(2)ki} = \begin{cases} \frac{d}{d-1+\widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)})} & \text{if } k, i \text{ neurons are active} \\ \text{no change} & \text{if } k \text{ non-active} \\ 0 & \text{otherwise} \end{cases} \tag{6.19}$$

or, in matrix notation:

$$W_{(2)(k,:)\text{new}} = \frac{d\mathbf{x}^{\mathrm{T}}}{d - 1 + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}} \tag{6.20}$$

and all other weights are left unchanged.

Proof. Only row k of $W_{(2)}$ is to be changed (weights related to F_2 winner). $f_1(\mathbf{a}_{(1)})$ returns 1 for active neurons, 0 otherwise, see (6.10), then $f_1(\mathbf{a}_{(1)}) = \mathbf{x}$. □

6.2.5 Subpatterns and F_2 Weights Initialization

Subpatterns

As the input patterns are binary, all patterns are subpatterns of the unit vector $\widehat{\mathbf{1}}$. Also it is possible that a pattern $\mathbf{x}_1$ may be a subpattern of another $\mathbf{x}_2$: $\mathbf{x}_1 \subset \mathbf{x}_2$, i.e. either $x_{i1} = x_{i2}$ or $x_{i1} = 0$.

❖ $\mathbf{x}_1$, $\mathbf{x}_2$

Proposition 6.2.1. *Suppose $\mathbf{x}_1 \subset \mathbf{x}_2$. The ART1 network ensures that the proper F_2 neuron will win, when a previously learned pattern is presented to the network. The winning neurons are different for two inputs $\mathbf{x}_1$ and $\mathbf{x}_2$ if they are from different classes, i.e. they have been learned previously by different neurons.*

Proof. When first presented with an input vector the F_1 layer outputs the same vector and distributes it to F_2 layer, see (6.3) and (6.10) (the change in F_1 activation pattern later is used just to reset the F_2 layer). So at this stage $f_1(\mathbf{a}_{(1)}) = \mathbf{x}$.

❖ k_1, k_2

Assuming that $F_2 : k_1$, respectively $F_2 : k_2$ neurons have learned $\mathbf{x}_1$, respectively $\mathbf{x}_2$, according to (6.20) their weights should be:

$$W_{(2)(k_1,:)} = \frac{d\mathbf{x}_1^{\mathrm{T}}}{d - 1 + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_1} \quad \text{and} \quad W_{(2)(k_2,:)} = \frac{d\mathbf{x}_2^{\mathrm{T}}}{d - 1 + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_2}$$

1. $\mathbf{x}_1$ is presented as input. Total input to k_1 neuron is (output of F_1 *is* $\mathbf{x}_1$):

$$a_{(2)k_1} = W_{(2)(k_1,:)}\mathbf{x}_1 = \frac{d\mathbf{x}_1^{\mathrm{T}}\mathbf{x}_1}{d - 1 + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_1}$$

while the total input to k_2 neuron is:

$$a_{(2)k_2} = W_{(2)(k_2,:)}\mathbf{x}_1 = \frac{d\mathbf{x}_2^{\mathrm{T}}\mathbf{x}_1}{d - 1 + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_2}$$

Because $\mathbf{x}_1 \subset \mathbf{x}_2$ (thus $x_{i1} = 1 \Rightarrow x_{i2} = 1$) then $\mathbf{x}_2^{\mathrm{T}}\mathbf{x}_1 = \mathbf{x}_1^{\mathrm{T}}\mathbf{x}_1$ but $\widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_2 > \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_1$ and then $a_{(2)k_2} < a_{(2)k_1}$ and k_1 wins as it should.

2. $\mathbf{x}_2$ is presented as input. The total input to k_1 neuron is:

$$a_{(2)k_1} = W_{(2)(k_1,:)}\mathbf{x}_2 = \frac{d\mathbf{x}_1^{\mathrm{T}}\mathbf{x}_2}{d - 1 + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_1}$$

while the total input to k_2 neuron is:

$$a_{(2)k_2} = W_{(2)(k_2,:)}\mathbf{x}_2 = \frac{d\mathbf{x}_2^{\mathrm{T}}\mathbf{x}_2}{d - 1 + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_2}$$

As $\mathbf{x}_1 \subset \mathbf{x}_2$ and they are binary vectors then $\mathbf{x}_1^{\mathrm{T}}\mathbf{x}_2 = \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_1$, $\mathbf{x}_2^{\mathrm{T}}\mathbf{x}_2 = \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_2$ and $\widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_1 < \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_2$, then:

$$a_{(2)k_1} = \frac{d}{\frac{d-1}{\widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_1} + 1} < a_{(2)k_2} = \frac{d}{\frac{d-1}{\widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}_2} + 1}$$

and neuron k_2 wins as it should. □

Uncommitted neurons and F_2 weights initialization

Proposition 6.2.2. *If the weights $W_{(2)}$ are initialized such that:*

$$w_{(2)\ell i} \in \left(0, \frac{d}{d - 1 + N}\right) \tag{6.21}$$

then uncommitted neurons do not win over already used ones when a previously learned pattern is presented to the network.

Proof. Assume that a vector $\mathbf{x}$ has been previously learned by a $F_2 : k$ neuron, then $W_{(2)(k,:)} = \frac{d\mathbf{x}^{\mathrm{T}}}{d-1+\widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}}$ (see (6.20), $f_1(\mathbf{a}_{(1)}^{\mathrm{T}}) = \mathbf{x}$ during learning). When $\mathbf{x}$ is presented to the network the k neuron should win over uncommitted neurons, i.e. $a_{(2)k} > a_{(2)\ell}$:

$$a_{(2)k} = W_{(2)(k,:)}\mathbf{x} = \frac{d\mathbf{x}^{\mathrm{T}}\mathbf{x}}{d - 1 + \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}} \leqslant \frac{d\mathbf{x}^{\mathrm{T}}\mathbf{x}}{d - 1 + N} = \frac{d\widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{x}}{d - 1 + N} > W_{(2)(\ell,:)}\mathbf{x}$$

since the weights satisfy (6.21). □

Weights should not be initialized to 0, because this may give 0 output. Also the values by which they are initialized should be random such that when a new class of inputs are presented at input and none of previous committed neurons wins then only one of the uncommitted neurons wins (but if several unused neurons win then any may be committed to the new pattern).

6.2.6 The Reset Neuron, Noise and Learning

The reset neuron

The purpose of reset neuron is to detect mismatches between input $\mathbf{x}$ and output of F_1 layer:

- at the start, when an input is present, the output of F_1 is identical to the input and the reset unit should not activate;
- also, it should not activate if the difference between input and F_1 output is below some specified value (differences between input and stored pattern appear due to noise, missing data or small differences between vectors belonging to same class).

All inputs are of equal importance so they receive the same weight; the same happens with F_1 outputs but they came as inhibitory input to the reset unit. See Figure 6.1 on page 90.

Let w_I be the weight(s) for inputs and $-w_{F_1}$ the weight(s) for F_1 connections, $w_I, w_{F_1} > 0$. The total input to reset unit is a_R: ❖ w_I, w_{F_1}, a_R

$$a_R = w_I \widehat{\mathbf{1}}^{\mathrm{T}} \mathbf{x} - w_{F_1} \widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)})$$

The reset unit activates if the input a_R is positive:

$$a_R > 0 \quad \Leftrightarrow \quad w_I \widehat{\mathbf{1}}^{\mathrm{T}} \mathbf{x} - w_{F_1} \widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)}) > 0 \quad \Rightarrow \quad \frac{\widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)})}{\widehat{\mathbf{1}}^{\mathrm{T}} \mathbf{x}} < \frac{w_I}{w_{F_1}} = \rho$$

where ρ is called the *vigilance parameter*. For $\frac{\widehat{\mathbf{1}}^{\mathrm{T}} f_1(\mathbf{a}_{(1)})}{\widehat{\mathbf{1}}^{\mathrm{T}} \mathbf{x}} \geqslant \rho$ the reset unit does not trigger. Because at the beginning (before F_2 activates) $f_1(\mathbf{a}_{(1)}) = \mathbf{x}$ then the vigilance parameter should be $\rho \leqslant 1$, i.e. $w_I \leqslant w_{F_1}$ (otherwise it will always trigger). vigilance parameter ❖ ρ

Noise in data

The vigilance parameter ρ specifies the *relative* error value $1 - \rho$ (of $f_1(\mathbf{a}_{(1)})$ with respect to $\mathbf{x}$) which will trigger a reset.

For the same amount of absolute noise (same number of spurious ones in the input vector) the ratio between noise and information varies depending upon the number of ones in the input vector, e.g. assuming that the stored vector is the smallest one: $\mathbf{x}^{\mathrm{T}} = (0 \ \ldots \ 1 \ \ldots \ 0)$ (just one "1") then for an input with noise the ratio between noise and data is at least $1:1$; for an stored vector having two ones the minimal noise:data ratio drops to half ($0.5:1$) and so on.

New pattern learning

If the reset neuron is activated then it inhibits the winning F_2 neuron for a sufficiently long time such that all committed F_2 neurons have a chance to win (and see if the input pattern is "theirs") or a new uncommitted neuron is set to learn a new class of inputs.

If none of the already used neurons was able to establish a "resonance" between F_1 and F_2 then an unused (so far) $F_2 : k$ neuron wins. The activities of F_1 neurons are:

$$a_{(1)i} = \frac{x_i + \eta_1 W_{(1)(i,:)} f_2(\mathbf{a}_{(2)}) - \beta_1}{1 + \gamma_1 + \alpha_1 [x_i + \eta_1 W_{(1)(i,:)} f_2(\mathbf{a}_{(2)})]} = \frac{x_i + \eta_1 w_{(1)ik} - \beta_1}{1 + \gamma_1 + \alpha_1 (x_i + \eta_1 w_{(1)ik})}$$

(see (6.4), $f_2(\mathbf{a}_{(2)})$ is 1 just for winner k and zero in rest and then $W_{(1)(i,:)} f_2(\mathbf{a}_{(2)}) = w_{(1)ik}$). For newly committed F_2 neurons the weights (from F_2 to F_1) are initialized to a value $w_{(1)ik} > \frac{\beta_1 - 1}{\eta_1}$ (see (6.16)) and then:

$$x_i + \eta_1 w_{(1)ik} - \beta_1 > x_i - 1$$

which means that for $x_i = 1$ the activity a_i is positive and $f_1(a_i) = 1$ while for $x_i = 0$, because of the 2/3 rule, the activity is negative and $f_1(a_i) = 0$.

Conclusion: when a new F_2 neuron is committed to learn the input, the output of the F_1 layer is identical to the input, the reset neuron does not activate, and the learning of the new pattern begins.

Remarks:

- The F_2 layer should has enough neurons for all classes to be learnt, otherwise an overloading of neurons, and consequently instabilities, may occur.
- Learning of new patterns may be stopped and resumed at any time by allowing or denying the weight adaptation.

Missing data

Assume that an input vector, which is similar to a stored/learned one, is presented to the network. Consider the case where some data is missing, i.e. some components of input are 0 in places where the stored one has 1's. Assuming that it is "far" enough from other stored vectors, only the designated F_2 neuron will win — that one which previously learned the complete pattern (there is no reset).

Assuming that a reset does not occur, the vector sent by the F_1 layer to F_2 will have less 1-s than the stored pattern (after one transmission cycle between $F_{1,2}$ layers). The corresponding weights (see (6.17): i non-active because of 2/3 rule, k active) are set to 0 and eventually the F_2 winner learns the new input, assuming that learning was allowed (i.e. weights were free to adapt).

If the original, complete, input vector is applied again the original F_2 neuron may learn again the same class of input vectors or otherwise a new F_2 neuron will be committed to learn (either an uncommitted one or one of the committed neurons).

This kind of behavior may lead to a continuous change in the class of vectors represented by the F_2 neurons, if learning is always allowed. On the other hand this is also the mechanism which allows unsupervised classification.

Fast access to learned patterns

Theorem 6.2.1. *If the input is identical to a stored (previously learnt) pattern $\mathbf{x}$ then the ART1 network stabilizes in one cycle, i.e. the input has direct access to the F_2 neuron which has learned it.*

Proof. When a previously learned $\mathbf{x}$ is presented to the network, Proposition 6.2.2 ensures that no uncommitted neuron wins while Proposition 6.2.1 ensures that, from those committed ones, only the proper neuron wins.

So, an $\mathbf{x}$ is presented to the network, at first cycle $f_1(\mathbf{a}_{(1)}) = \mathbf{x}$ and then the proper $F_2 : k$ winner gets selected.

The new activation of F_1 layer is calculated from (6.4). In order to calculate the new $f_1(\mathbf{a}_{(1)})$ only the sign of $\mathbf{a}_{(1)}$ is necessary:

$$\operatorname{sign}(\mathbf{a}_{(1)}) = \operatorname{sign}[(1+\eta_1)\mathbf{x} - \beta_1 \widehat{\mathbf{1}}]$$

(as the proper winner has been selected then $W_{(1)} f_2(\mathbf{a}_{(2)}) = \mathbf{x}$, see (6.18); the denominator of (6.4) is always positive as being a sum of positive terms).

Calculating the new $f_1(\mathbf{a}_{(1)})$ according to (6.10) gives $f_1(\mathbf{a}_{(1)}) = \mathbf{x}$, i.e. the F_1 output does *not* change and the network has stabilized after one propagation $F_1 \to F_2$. □

6.3 The ART1 Algorithm

Initialization

The size of F_1 layer: N is determined by the dimension of the input vectors.

1. The dimension of the F_2 layer K is based on the desired number of classes to be learned *now and later*. Note also that in special cases some classes may need to be divided into "subclasses" with different assigned F_2 winning neurons. This is particularly necessary when classes are represented by non–connected domains in $\{0,1\}^N$ space.
2. Select the constants: β_1 and $\eta_1 > 0$ such that $\max(1,\eta_1) < \beta_1 < \eta_1 + 1$. Select $d > 1$ and $\rho \in (0,1]$.
3. Initialize the $W_{(1)}$ weights with random values within the range $\left(\frac{\beta_1 - 1}{\eta_1}, 1\right)$.
4. Initialize $W_{(2)}$ weights with random values within the range $\left(0, \frac{d}{d-1+N}\right)$.

Network running and learning

The algorithm uses the asymptotic learning method.

1. Apply an input vector $\mathbf{x}$, the output of F_1 is $\mathbf{x}$. Calculate the activities of F_2 neurons and find the winner. The neuron with the biggest input from F_1 wins (and all others will have zero output). The winner is the $F_2 : k$ neuron for which:

$$W_{(2)(k,:)}\mathbf{x} = \max_{\ell} W_{(2)(\ell,:)}\mathbf{x}$$

2. Calculate the new activities of F_1 neurons caused by inputs from F_2. The F_2 output is a vector which has all its components 0 except for the k–th component corresponding to the winner. Multiplying $W_{(1)}$ by such a vector means that column k gets selected: $W_{(1)} f_2(\mathbf{a}_{(2)}) = W_{(1)(:,k)}$. In order to calculate the new F_1 output only the sign of $\mathbf{a}_{(1)}$ is necessary. Using (6.4) (the denominator is always positive):

$$\operatorname{sign}(\mathbf{a}_{(1)}) = \operatorname{sign}[\mathbf{x} + \eta_1 W_{(1)(:,k)} - \beta_1 \widehat{\mathbf{1}}] = \operatorname{sign}[\mathbf{x} + \eta_1 W_{(1)(:,k)} \overset{\mathrm{C}}{\ominus} \beta_1]$$

and the new F_1 output is calculated from (6.10):

$$f_1(a_{(1)i})_{\text{new}} = \begin{cases} 1 & \text{if } a_{(1)i} > 0 \\ 0 & \text{if } a_{(1)i} \leqslant 0 \end{cases}.$$

Remarks:

- $f_1(\mathbf{a}_{(1)})$ may also be calculated through:

$$f_1(\mathbf{a}_1) = \widehat{\mathbf{1}} - \operatorname{sign}[\widehat{\mathbf{1}} - \operatorname{sign}(\mathbf{a}_{(1)})]$$

Proof. $\operatorname{sign}(\mathbf{a}_{(1)})$ may return the values $\{-1, 0, +1\}$; the formula is proved by considering all three cases in turn, see (6.10). □

3. Calculate the "degree of match" between input and the new output of the F_1 layer

$$\text{degree of match} = \frac{\operatorname{sum}[f_1(\mathbf{a}_{(1)})_{\text{new}}]}{\operatorname{sum}(\mathbf{x})} \; .$$

4. Compare the "degree of match" computed previously with the vigilance parameter ρ.

   If the vigilance parameter is bigger than the "degree of match" then:

   (a) mark the current $F_2{:}k$ neuron as inactive for the rest of the cycle, i.e. while working with the same input $\mathbf{x}$;

   (b) *restart* the procedure with the same input vector.

   Otherwise *continue*, the input vector was positively identified (assigned, if the winning neuron is a previously unused one) as being of class k.

5. Update the weights (if learning is enabled; it has to be always enabled for new classes). See (6.17) and (6.19).

$$W_{(1)(:,k)\text{new}} = \mathbf{x} \quad \text{and} \quad W_{(2)(k,:)\text{new}} = \frac{d\,\mathbf{x}^{\mathrm{T}}}{d - 1 + \operatorname{sum}(\mathbf{x})}$$

   (note that only the weights related to the winning F_2 neuron are updated).

6. The information returned by the network is the classification of the input vector given by the winning F_2 neuron (in the one–of–k encoding scheme).

6.4 ART2 Architecture

ART2

Unlike ART1, the ART2 network is designed to work with analog *positive* inputs. There is a broad similarity with ART1 architecture: there is an F_1 layer which sends its output to an F_2 layer and a reset *layer*. The F_2 layer is of "winner-takes-all" type and the reset has the same role as in ART1. However the F_1 layer is made out of 6 sublayers labeled w, s, u, v, p and q. See Figure 6.3 on the facing page. Each of the sublayers has the same number of neurons as the number of components in the input vector and they have one–to–one connections (including input to w).

❖ w, s, u, v, p, q

① The input vector is sent to the w sublayer.

② The output of the w sublayer is sent to s sublayer.

③ The output of the s sublayer is sent to the v sublayer.

④ The output of the v sublayer is sent to the u sublayer.

[6.4] For the whole ART2 topic see [CG87a], see [FS92] pp. 316–318.

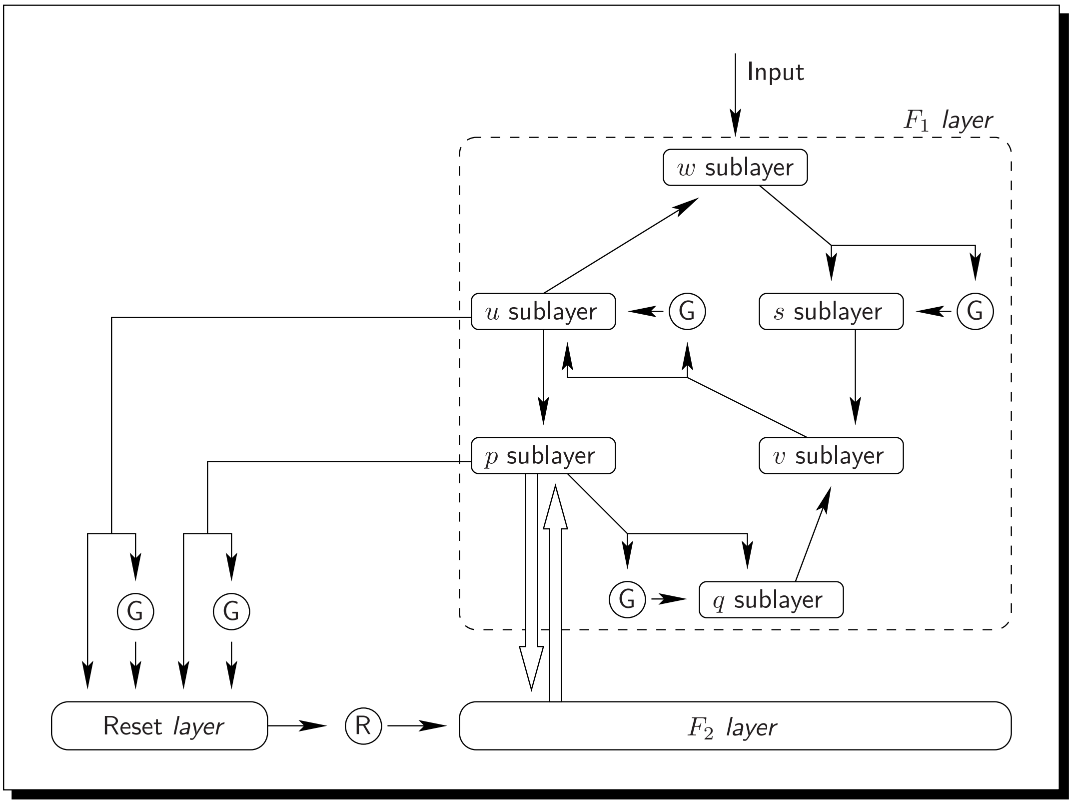

Figure 6.3: *The ART2 network architecture. Thin arrows represent neuron–to–neuron connections between (sub)layers; thick arrows represent full inter–layer connections (from all neurons to all neurons). The "G" are gain–control neurons which send an inhibitory signal; the "R" is the reset neuron.*

⑤ The output of the u sublayer is sent to the p sublayer, to the reset layer r and *back* to the w sublayer.

⑥ The output of the p sublayer is sent to the q sublayer and to the reset layer. The output of the p sublayer represents also the output of the F_1 layer and is sent to F_2.

Remarks:

- Between sublayers there is a one–to–one neuronal connection (neuron i from one layer to the corresponding neuron i from the other layer)
- All (sub)layers receive input from 2 sources, a supplementary gain–control neuron has been added where necessary such that the layers will comply with the 2/3 rule.
- The gain–control neurons have the role of normalizing the output of the corresponding layers (see also (6.22) equations); note that all layers have 2 sources of input either from 2 layers or from a layer and a gain–control. The gain–control neuron receives input from all neurons from the corresponding sublayer and sends

the sum of its input as *inhibitory* input (see also (6.23) and table 6.1), while the other layers sent an *excitatory* input.

6.5 ART2 Dynamics

6.5.1 The F_1 layer

The differential equations governing the F_1 sublayers behavior are:

$$\frac{d\mathbf{w}}{dt} = -\mathbf{w} + \mathbf{x} + a\mathbf{u} \qquad \frac{d\mathbf{v}}{dt} = -\mathbf{v} + h(\mathbf{s}) + bh(\mathbf{q})$$

$$\frac{d\mathbf{s}}{dt} = -\varepsilon\mathbf{s} + \mathbf{w} - \mathbf{s}\|\mathbf{w}\| \qquad \frac{d\mathbf{p}}{dt} = -\mathbf{p} + \mathbf{u} + W_{(1)} f_2(\mathbf{a}_{(2)})$$

$$\frac{d\mathbf{u}}{dt} = -\varepsilon\mathbf{u} + \mathbf{v} - \mathbf{u}\|\mathbf{v}\| \qquad \frac{d\mathbf{q}}{dt} = -\varepsilon\mathbf{q} + \mathbf{p} - \mathbf{q}\|\mathbf{p}\|$$

❖ a, b, ε
❖ $W_{(1)}$, f_2, $\mathbf{a}_{(2)}$
❖ h, α

where $a, b, \varepsilon =$ const., $W_{(1)}$ is the matrix of weights for connections $F_2 \to F_1$; f_2 is the activation function of F_2 while $\mathbf{a}_{(2)}$ is the total input to F_2; The h function determines the contrast enhancement which takes place inside the F_1 layer, possible definitions would be:

- continuously differentiable: $h(x) = \begin{cases} x & \text{if } x \geqslant \alpha \\ \frac{2\alpha x^2}{x^2+\alpha^2} & \text{if } x \in [0,\alpha) \end{cases}$
- piecewise linear: $h(x) = \begin{cases} x & \text{if } x \geqslant \alpha \\ 0 & \text{if } x \in [0,\alpha) \end{cases}$

where $\alpha \in (0,1)$, $\alpha =$ const..

Remarks:

➥ The piecewise linear function may be written as:

$$h(\mathbf{x}) = \mathbf{x} \odot \operatorname{sign}[\operatorname{sign}(\mathbf{x} - \alpha\widehat{\mathbf{1}}) + \widehat{\mathbf{1}}]$$

The equilibrium values (obtained by setting the derivatives to zero) are:

$$\begin{aligned} \mathbf{w} &= \mathbf{x} + a\mathbf{u} & \mathbf{v} &= h(\mathbf{s}) + bh(\mathbf{q}) \\ \mathbf{s} &= \frac{\mathbf{w}}{\|\mathbf{w}\| + \varepsilon} & \mathbf{p} &= \mathbf{u} + W_{(1)} f_2(\mathbf{a}_{(2)}) \\ \mathbf{u} &= \frac{\mathbf{v}}{\|\mathbf{v}\| + \varepsilon} & \mathbf{q} &= \frac{\mathbf{p}}{\|\mathbf{p}\| + \varepsilon} \end{aligned} \tag{6.22}$$

[6.5] See [FS92] pp. 318–324 and [CG87a].

Layer	Neuron output	c_1	c_2	Excitatory input	Inhibitory input
w	w_i	1	1	$x_i + au_i$	0
s	s_i	ε	1	w_i	$\\|\mathbf{w}\\|$
u	u_i	ε	1	v_i	$\\|\mathbf{v}\\|$
v	v_i	1	1	$h(s_i) + bh(q_i)$	0
p	p_i	1	1	$u_i + W_{(1)(i,:)} f_2(\mathbf{a}_{(2)})$	0
q	q_i	ε	1	p_i	$\\|\mathbf{p}\\|$
r	r_i	ε	1	$u_i + cp_i$	$\\|\mathbf{u}\\| + c\\|\mathbf{p}\\|$

Table 6.1: *The parameters for the general, ART2 differential equation* (6.23).

These results may be described by the means of one single equation with different parameters for different sublayers — see table 6.1:

$$\frac{d(\text{neuron output})}{dt} = - c_1 \text{neuron output} + \text{excitatory input} \qquad (6.23)$$
$$- c_2 \text{neuron output} \times \text{inhibitory input}$$

Remarks:

- An equation of the form of (6.23) is also applicable to the reset layer. The appropriate parameters are listed in Table 6.1: layer r ($c = \text{const.}$).
- The purpose of the constant ε is to limit the output of s, q, u and r (sub)layers when their input is 0 and consequently ε should be chosen $\varepsilon \gtrsim 0$ and may be neglected when real data is presented to the network.

6.5.2 The F_2 Layer

The F_2 layer of an ART2 network is identical in functionality to the F_2 layer of an ART1 network. The total input into neuron k is $a_{(2)k} = W_{(2)(k,:)}\mathbf{p}$, where $W_{(2)}$ is the matrix of weights for connections $F_1 \to F_2$ (output of F_1 is $\mathbf{p}$). The output of F_2 is:

$$f_2(a_{(2)k}) = \begin{cases} d & \text{if } a_{(2)k} = \max\limits_{\ell} a_{(2)\ell} \\ 0 & \text{otherwise} \end{cases} \qquad (6.24)$$

where $d = \text{const.}$, $d \in (0, 1)$. and then the output of the p sublayer becomes (see (6.22)): ❖ d

$$\mathbf{p} = \begin{cases} \mathbf{u} & \text{if } F_2 \text{ is inactive} \\ \mathbf{u} + dW_{(1)(:,k)} & \text{for } k \text{ neuron winner on } F_2 \end{cases} \qquad (6.25)$$

6.5.3 The Reset Layer

The differential equation defining the reset layer behaviour is of the form (6.23) with the parameters defined in Table 6.1:

$$\frac{d\mathbf{r}}{dt} = -\varepsilon\mathbf{r} + \mathbf{u} + c\mathbf{p} - \mathbf{r}(\|\mathbf{u}\| + c\|\mathbf{p}\|)$$

with the inhibitory input given by the 2 gain–control neurons. The equilibrium value is obtained from $\frac{d\mathbf{r}}{dt} = \widehat{\mathbf{0}}$:

$$\mathbf{r} = \frac{\mathbf{u} + c\mathbf{p}}{\varepsilon + \|\mathbf{u}\| + c\|\mathbf{p}\|} \simeq \frac{\mathbf{u} + c\mathbf{p}}{\|\mathbf{u}\| + c\|\mathbf{p}\|} \tag{6.26}$$

(see also the remarks regarding the value and role of ε).

❖ ρ

By definition — considering ρ the vigilance parameter — the reset occurs when:

$$\|\mathbf{r}\| < \rho \; . \tag{6.27}$$

The reset should not activate before an output from the F_2 layer has arrived. From the equations (6.22), if $f_2(\mathbf{a}_{(2)}) = \widehat{\mathbf{0}}$ then $\mathbf{p} = \mathbf{u}$ and then (6.26) gives $\|\mathbf{r}\| = 1$. This means that ρ should be chosen such that $\rho \in (0,1)$; values of ρ near 1 give a high sensitivity to mismatches.

6.5.4 Learning and Initialization

The differential equations governing the weights adaptation are defined as:

$$\frac{dw_{(1)ik}}{dt} = f_2(a_{(2)k})(p_i - w_{(1)ik}) \qquad \text{and} \qquad \frac{dw_{(2)ki}}{dt} = f_2(a_{(2)k})(p_i - w_{(2)ki})$$

and, considering (6.24) and (6.25):

$$\frac{dW_{(1)(:,k)}}{dt} = \begin{cases} d(\mathbf{u} + dW_{(1)(:,k)} - W_{(1)(:,k)}) & \text{for } k \text{ winner on } F_2 \\ \widehat{\mathbf{0}} & \text{otherwise} \end{cases}$$

$$\frac{dW_{(2)(k,:)}}{dt} = \begin{cases} d(\mathbf{u}^{\mathrm{T}} + dW^{\mathrm{T}}_{(1)(:,k)} - W_{(2)(k,:)}) & \text{for } k \text{ winner on } F_2 \\ \widehat{\mathbf{0}}^{\mathrm{T}} & \text{otherwise .} \end{cases}$$

This is also why the $d \in (0,1)$ condition is necessary, without it weights may grow indefinitely.

Proof. Consider just one $w_{(1)ik}$; its differential equation is: $\frac{dw_{(1)ik}}{dt} = -(d - d^2)w_{(1)ik} + du_i$. If the coefficient $d - d^2$ is negative then the weight will grow indefinitely as there will be no inhibition. □

Asymptotic learning

Only the weights related to the winning F_2 neuron are updated, all others remain unchanged. The equilibrium values are obtained by setting the derivatives to zero. Assuming that k is the winner then $\mathbf{u} + dW_{(1)(:,k)} - W_{(1)(:,k)} = \widehat{\mathbf{0}}$ and $\mathbf{u}^{\mathrm{T}} + dW^{\mathrm{T}}_{(1)(:,k)} - W_{(2)(k,:)} = \widehat{\mathbf{0}}^{\mathrm{T}}$, so:

$$W_{(1)(:,k)} = \frac{\mathbf{u}}{1-d} \qquad \text{and} \qquad W_{(2)(k,:)} = \frac{\mathbf{u}^{\mathrm{T}}}{1-d} = W^{\mathrm{T}}_{(1)(:,k)} \tag{6.28}$$

Eventually the $W_{(2)}$ weight matrix becomes the transpose of W matrix — when all the F_2 neurons have been used to learn new data.

New pattern learning, initialization of $W_{(1)}$ weights

The reset neuron should not activate when a learning process takes place.

Using the fact that $\mathbf{u} \simeq \frac{\mathbf{v}}{\|\mathbf{v}\|}$ $(\varepsilon \simeq 0)$ and then $\|\mathbf{u}\| \simeq 1$, from (6.26):

$$\|\mathbf{r}\| = \sqrt{\mathbf{r}^{\mathrm{T}}\mathbf{r}} \simeq \frac{\sqrt{1 + 2c\|\mathbf{p}\| \cos(\widehat{\mathbf{u},\mathbf{p}}) + c^2\|\mathbf{p}\|^2}}{1 + c\|\mathbf{p}\|} \tag{6.29}$$

where $\widehat{\mathbf{u},\mathbf{p}}$ is the angle between $\mathbf{u}$ and $\mathbf{p}$ vectors; if $\mathbf{p}$ is parallel to $\mathbf{u}$ then $\|\mathbf{r}\| \simeq 1$ and a reset does not occur as $\rho < 1$ and the reset condition (6.27) is not met.

If the $W_{(1)}$ weights are initialized to $\widetilde{0}$ then the input from F_2, *at the beginning of the learning process*, is zero and $\mathbf{p} = \mathbf{u}$ (see (6.25)) such that the reset does not occur.

During the learning process the weight vector $W_{(1)(:,k)}$, associated with connection from F_2 winner to F_1, becomes parallel to $\mathbf{u}$ (see (6.28)) and then, from (6.25), $\mathbf{p}$ moves (during the learning process) towards becoming parallel with $\mathbf{u}$ and again a reset does not occur.

Conclusion: The $W_{(1)}$ weights have to be initialized to $\widetilde{0}$.

Initialization of $W_{(2)}$ weights

Assume that an $F_2 : k$ neuron has learned an input vector and, after some time, that input is presented again to the network. The same k neuron should win again, and not one of the as yet uncommitted F_2 neurons. This means that the output of the k neuron, i.e. $a_{(2)k} = W_{(2)(k,:)}\mathbf{p}$ should be bigger than any $a_{(2)\ell} = W_{(2)(\ell,:)}\mathbf{p}$ for all unused $F_2 : \ell$ neurons[2]:

$$W_{(2)(k,:)}\mathbf{p} > W_{(2)(\ell,:)}\mathbf{p} \quad \Rightarrow \quad \|W_{(2)(k,:)}\| \, \|\mathbf{p}\| > \|W_{(2)(\ell,:)}\| \, \|\mathbf{p}\| \cos(\widehat{\mathbf{p}, W_{(2)(\ell,:)}})$$

because $\mathbf{p} \parallel \mathbf{u} \parallel W_{(2)(k,:)}$, see (6.25) and (6.28) (the learning process — weight adaptation — has been previously done for the k neuron).

The worst possible case is when $W_{(2)(\ell,:)} \parallel \mathbf{p}$ so that $\cos(\widehat{\mathbf{p}, W_{(2)(\ell,:)}}) = 1$. To ensure that no other neuron but k wins, the condition:

$$\|W_{(2)(\ell,:)}\| < \|W_{(2)(k,:)}\| = \frac{1}{1-d} = \left\| \frac{\widehat{\mathbf{1}}^{\mathrm{T}}}{\sqrt{K}\,(1-d)} \right\| \tag{6.30}$$

has to be imposed for unused $F_2 : \ell$ neurons (for a committed neuron $W_{(2)(k,:)} = \frac{\mathbf{u}^{\mathrm{T}}}{1-d}$ and $\|\mathbf{u}\| = 1$ as $\mathbf{u} \simeq \frac{\mathbf{v}}{\|\mathbf{v}\|}$, $\varepsilon \simeq 0$).

To maximize the unused neurons input $a_{(2)\ell}$ such that the network will be more sensitive to new patterns the weights of (unused) neurons have to be initialized with maximum values allowed by the condition (6.30), i.e. $w_{(2)\ell i} \lesssim \frac{1}{\sqrt{K}\,(1-d)}$.

Conclusion: The $W_{(2)}$ weights have to be initialized with: $w_{(2)ki} \lesssim \frac{1}{\sqrt{K}\,(1-d)}$.

[2]Note that ART2 works only with *positive* vectors so the symmetric case does not apply.

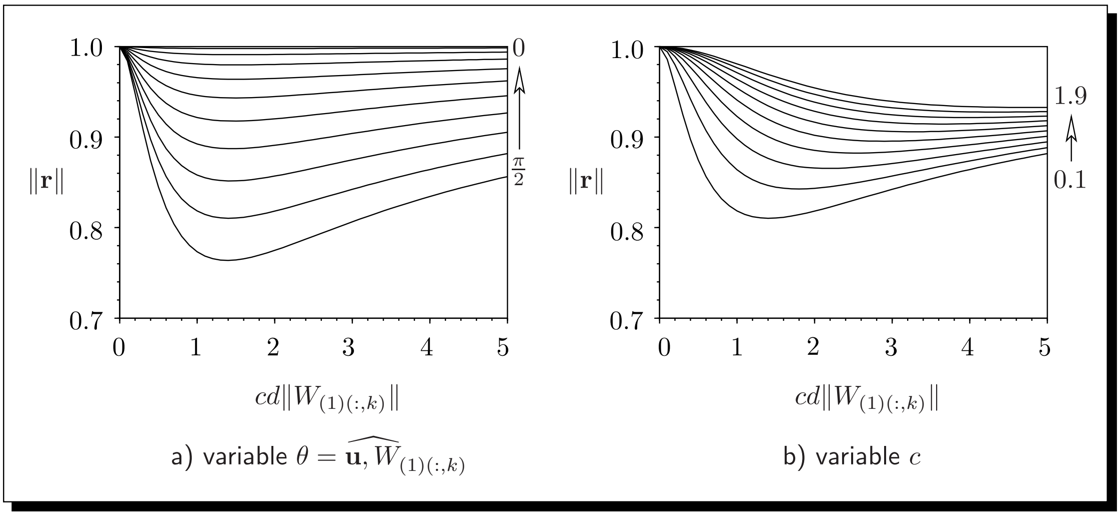

Figure 6.4: $\|\mathbf{r}\|$ *as function of* $cd\|W_{(1)(:,k)}\|$. *Figure a) shows dependency for various angles* $\widehat{\mathbf{u}, W_{(1)(:,k)}}$, *from* $\pi/2$ *to* 0 *in* $\pi/20$ *steps and* $c = 0.2$ *constant. Figure b) shows dependency for various* c *values, from* 0.1 *to* 1.9 *in* 0.2 *steps, and* $\widehat{\mathbf{u}, W_{(1)(:,k)}} = \pi/2 - \pi/20$ *constant.*

Constrains on the constants

As $\mathbf{p} = \mathbf{u} + dW_{(1)(:,k)}$ and $\|\mathbf{u}\| = 1$ then:

$$\mathbf{u}^{\mathrm{T}}\mathbf{p} = \|\mathbf{p}\| \cos(\widehat{\mathbf{u}, \mathbf{p}}) = 1 + d\|W_{(1)(:,k)}\| \cos(\widehat{\mathbf{u}, W_{(1)(:,k)}})$$

$$\|\mathbf{p}\| = \sqrt{\mathbf{p}^{\mathrm{T}}\mathbf{p}} = \sqrt{1 + 2d\|W_{(1)(:,k)}\| \cos(\widehat{\mathbf{u}, W_{(1)(:,k)}}) + d^2\|W_{(1)(:,k)}\|^2}$$

(k being the F_2 winner). Substituting $\|\mathbf{p}\| \cos(\widehat{\mathbf{u}, \mathbf{p}})$ and $\|\mathbf{p}\|$ into (6.29) gives:

$$\|\mathbf{r}\| = \frac{\sqrt{(1+c)^2 + 2(1+c) \cdot cd\|W_{(1)(:,k)}\| \cdot \cos(\widehat{\mathbf{u}, W_{(1)(:,k)}}) + [cd\|W_{(1)(:,k)}\|]^2}}{1 + \sqrt{c^2 + 2c \cdot cd\|W_{(1)(:,k)}\| + [cd\|W_{(1)(:,k)}\|]^2}}$$

Figure 6.4 shows $\|\mathbf{r}\|$ as function of $cd\|W_{(1)(:,k)}\|$, for various fixed values of $\theta = \widehat{\mathbf{u}, W_{(1)(:,k)}}$ and c — note that $\|\mathbf{r}\|$ *decreases for* $cd\|W_{(1)(:,k)}\| < 1$.

Discussion:

- The learning process should increase the mismatch sensitivity between the F_1 pattern sent to F_2 and the classification received from F_2, i.e. at the end of the learning process, the network should be able to discern better between different classes of input. This means that, while the network learns a new input class and $W_{(1)(:,k)}$ increases from initial value $\widehat{\mathbf{0}}$ to the final value when $\|W_{(1)(:,k)}\| = \frac{1}{1-d}$, the $\|\mathbf{r}\|$ value (defining the network sensitivity) has to decrease (such that reset condition (6.27)

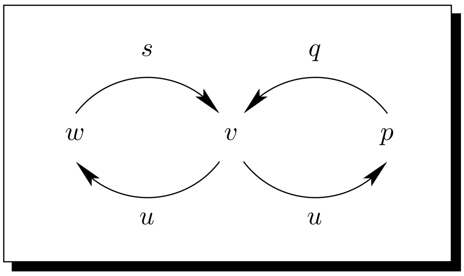

Figure 6.5: *The F_1 dynamics: data communication between sublayers.*

becomes easier to meet). In order to achieve this the following condition is imposed:

$$\frac{cd}{1-d} \leqslant 1$$

(at the end of the learning $\|W_{(1)(:,k)}\| = \frac{1}{1-d}$).

Or, in other words, when there is a perfect fit $\|\mathbf{r}\|$ reaches its maximal value 1; when presented with a slightly different input vector, the same F_2 neuron should win and adapt to the new value. During this process the value of $\|\mathbf{r}\|$ will first *decrease* before increasing back to 1. When decreasing it may happen that $\|\mathbf{r}\| < \rho$ and a reset occurs. This means that the input vector does not belong to the class represented by the current F_2 winner.

- For $\frac{cd}{1-d} \lesssim 1$ the network is more sensitive than for $\frac{cd}{1-d} \ll 1$ — the $\|\mathbf{r}\|$ value will drop more for the same input vector (slightly different from the stored/learned one). See Figure 6.4–a.
- For $c \ll 1$ the network is more sensitive than for $c \lesssim 1$ — same reasoning as above. See Figure 6.4–b.
- What happens in fact in the F_1 layer may be explained now:
  - s, u and q just normalize w, v and p outputs before sending them further.
  - There are connections between p and v (via q) and w and v (via s) and also back from v to w and p (via u). See Figure 6.5. Obviously v layer acts as a mediator between input $\mathbf{x}$ received via w and the output of p activated by F_2. During this negotiation $\mathbf{u}$ and $\mathbf{v}$ (as $\mathbf{u}$ is the normalization of $\mathbf{v}$) move away from $W_{(1)(:,k)}$ ($\|\mathbf{r}\|$ drops). If it moves too far then a reset occurs ($\|\mathbf{r}\|$ becomes smaller than ρ) and the process starts over with another $F_2 : \ell$ neuron and a new $W_{(1)(:,\ell)}$. Note that $\mathbf{u}$ represents eventually a normalized combination (filtered through h) of $\mathbf{x}$ and $\mathbf{p}$ (see (6.22)).

6.6 The ART2 Algorithm

Initialization

The dimensions of w, s, u, v, p, q and r layers equals N (dimension of input vector). The norm used is Euclidean: $\|\mathbf{x}\| = \sqrt{\mathbf{x}^{\mathrm{T}}\mathbf{x}}$.

1. Select network learning constants such that:

$$a, b > 0 \quad , \quad \rho \in (0,1) \quad ,$$

$$d \in (0,1) \quad , \quad c \in (0,1) \quad , \quad \frac{cd}{1-d} \leqslant 1 \quad ,$$

and K, the size of F_2 (similar to ART1 algorithm).

2. Choose a contrast enhancement function h, e.g. the piecewise linear.

3. Initialize the weights:

$$W_{(1)} = \widetilde{0} \quad \text{and} \quad W_{(2)} \text{ such that } w_{(2)ki} \lesssim \frac{1}{\sqrt{K}\,(1-d)}$$

$W_{(2)}$ to be initialized with random values such that the above condition is met.

Network running and learning

1. Take an input vector $\mathbf{x}$.

2. Iterate the following steps till stabilisation (i.e. there are no further output changes):

$$\mathbf{w} = \begin{cases} \mathbf{x} & \text{first iteration} \\ \mathbf{x} + a\mathbf{u} & \text{next iterations} \end{cases} \quad \rightarrow \quad \mathbf{s} = \frac{\mathbf{w}}{\|\mathbf{w}\|} \quad \rightarrow$$

$$\rightarrow \quad \mathbf{v} = \begin{cases} h(\mathbf{s}) & \text{first iteration} \\ h(\mathbf{s}) + bh(\mathbf{q}) & \text{next iterations} \end{cases} \quad \rightarrow \quad \mathbf{u} = \frac{\mathbf{v}}{\|\mathbf{v}\|} \quad \rightarrow$$

$$\rightarrow \quad \mathbf{p} = \begin{cases} \mathbf{u} & \text{first iteration} \\ \mathbf{u} + dW_{(1)(:,k)} & \text{next iterations} \end{cases} \quad \rightarrow \quad \mathbf{q} = \frac{\mathbf{p}}{\|\mathbf{p}\|}$$

where k is the F_2 winner (note that at first iteration $\mathbf{u} = \mathbf{q} = \widehat{\mathbf{0}}$ and the F_2 layer is inactive).

The F_2 winner is found by calculating first the activation: $\mathbf{a}_{(2)} = W_{(2)}\mathbf{p}$; the winner k is found from: $a_{(2)k} = \max\limits_{\ell} a_{(2)\ell}$ (F_2 is of contrast enhancement type).

Note that usually two iterations are enough and the winner does not change here (during current pattern matching and before a reset has occurred).

3. Calculate the output of r layer:

$$\mathbf{r} = \frac{\mathbf{u} + c\mathbf{p}}{\|\mathbf{u}\| + c\|\mathbf{p}\|}$$

If there is a reset, i.e. $\|\mathbf{r}\| < \rho$ then the F_2 winner is made inactive for the current input vector and go back to step number 2, i.e. start over with same input $\mathbf{x}$.

If there is no reset then the resonance was found and go to next step.

4. If learning is allowed update the weights:

$$W_{(1)(:,k)} = W^{\mathrm{T}}_{(2)(k,:)} = \frac{\mathbf{u}}{1-d}$$

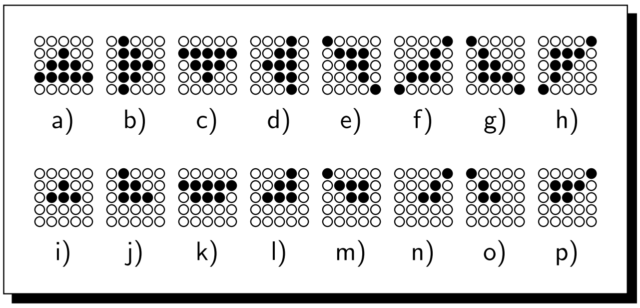

Figure 6.6: *The patterns used in ART1 rotation detection; a–h were used for training, i–p for testing (missing data).*

5. The information returned by the network is the classification of the input vector given by the winning F_2 neuron in one–of–k encoding.

6.7 Examples

6.7.1 Rotation Detection Using ART1

The problem is to detect the rotation position of an arrowhead shape as depicted in Figure 6.6.

An ART1 network was built[3] and trained. F_1 has 25 neurons while F_2 was built with 8 neurons. The parameters used were: $\beta_1 = 1.01$, $\eta_1 = 0.02$, $d = 25$ and $\rho = 0.99$ (ρ was chosen close to 1 in order to assure "narrow" classes).

The network was first trained with patterns a–d and tested with patterns i–l, note that data is missing from these (last two rows), see Figure 6.6. At this stage only 4 F_2 neurons have been used. The test were successful with pattern (i) being identified with (a), etc.

At a later stage training was resumed and additional patterns e–h were learned. Testing was performed using patterns m–p. The test were successful with pattern (e) being identified with (m), etc. This used up all neurons from F_2 and learning of further classes is no longer possible.

Note that in order to use this network it is necessary to calibrate it, i.e. to identify the F_2 winner for each class of inputs.

6.7.2 Signal Recognition Using ART2

The problem is to recognize a set of signals as depicted in Figure 6.7 on the next page, all with the same period $T = 1$ ($t \in [0, T]$): ❖ T

- sinusoidal: $x(t) = 0.4 \sin(2\pi t) + 0.5$;
- pulse: $x(t) = \begin{cases} 0.9 & \text{for } t \in [0, T/2) \\ 0.1 & \text{for } t \in [T/2, T) \end{cases}$;

[3]The actual Scilab code is in the source tree of this book, in directory, i.e. `Matrix_ANN_1/Scilab/ART1_rot_detect.sci`, note that actual version numbers may differ.

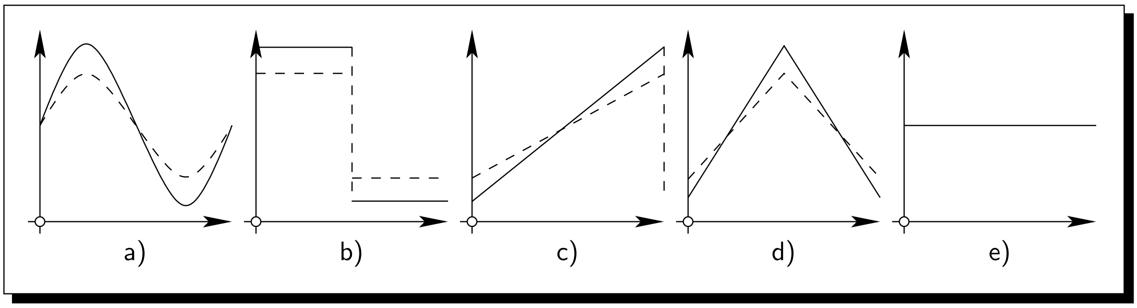

Figure 6.7: *The waveforms $x(t)$ used in the signal recognition example. The dotted lines show the test signals, which are the training signals centered and reduced in amplitude by 30%.*

- "saw dent": $x(t) = 0.8t + 0.1$;
- triangular: $x(t) = \begin{cases} 1.6t + 0.1 & \text{for } t \in [0, T/2) \\ -1.6t + 1.7 & \text{for } t \in [T/2, T) \end{cases}$;
- constant: $x(t) = 0.5$;

An ART2 network was built[4] and trained as follows:

- the signals were sampled at 20 equally spaced points, from 0 to 0.95, so input size of F_1 was $N = 20$;
- the size of F_2 was chosen at $K = 8$ (it is necessary to choose $K \geqslant 5$ in order to recognize each different signal as a separate class).
- the constants were: $a = 0.2$, $b = 0.2$, $c = 0.001$, $d = 0.999$ and $\rho = 0.99$ (a high sensitivity was sought);
- the chosen contrast enhancement function was the piecewise linear with $\alpha = 0.1$.

After training, the network was able to recognize the test set depicted in Figure 6.7 (dotted line) which has a 30% reduction in amplitude, calculated according to:

$$x_{\text{test}}(t) = 0.7[x(t) - 0.5] + 0.5$$

[4] The actual Scilab code is in the source tree of this book, in directory, e.g. `Matrix_ANN_1/Scilab/ART2_signal.sci`, note that actual version numbers may differ.

Basic Principles

CHAPTER 7

General Feedforward Networks

Feedforward ANN represents a large category including the perceptron, all backpropagation networks and RBF (Radial Basis Function) networks. They are used in classification and regression problems. This chapter describe the most general properties of feedforward ANN.

7.1 The Tensorial Notation

The notational system used so far will be changed to adopt a more general tensorial approach:

- A vector will be considered a 1–covariant tensor, e.g. $\mathbf{x} = \{x_i\}$.
- A matrix will be considered a 1–covariant and 1–contravariant tensor $W = \{w_k^i\}$, note that w_k^i will represent weight from neuron i to neuron k.
- The summation over the free index will be automatically assumed for the product of two tensors, e.g. $\mathbf{y} = W\mathbf{x}$ is a 1–covariant tensor whose components are $y_k = w_k^i x_i$ (summation over the "dummy" index i *appearing both as a subscript and as a superscript*). Note that a product of this form entails a reduction of W's dimensionality.
- The transpose of a tensor means swapping super and subscripts, e.g. if $H = \{h_{kj}^{i\ell}\}$ then $H^{\mathrm{T}} = \{h'^{kj}_{i\ell}\}$ where $h'^{kj}_{i\ell} = h_{kj}^{i\ell}$ (the $'$ warns that the indices have significance top $\leftrightarrow$ bottom reversed).
- The Kronecker matrix product transforms to the more general Kronecker tensorial product: $C = A \otimes B$ has the elements: $c_{k_1,\dots,k_m,\ell_1,\dots,\ell_n}^{i_1,\dots,i_p,j_1,\dots,j_q} = a_{k_1,\dots,k_m}^{i_1,\dots,i_p} b_{\ell_1,\dots,\ell_n}^{j_1,\dots,j_q}$.

7.*For a general treatment of feedforward ANN, from a statistical point of view see [Bis95].

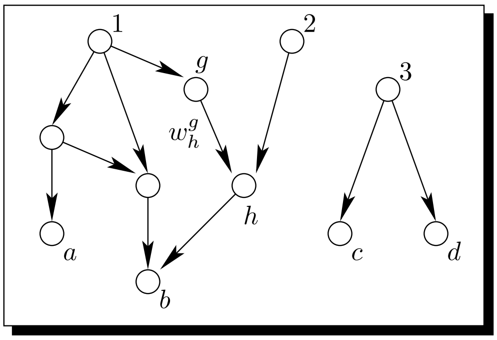

Figure 7.1: *The general feedforward network architecture. Vertices 1, 2 and 3 represent inputs while vertices a, b, c and d represent outputs. Each edge (g, h) has an associated weight w_h^g.*

- The Hadamard product keeps its significance, both tensors have to be of the same type and size; similarly for Hadamard division.

7.2 Architecture and Description

The general feedforward ANN may be represented as a directed graph without cycles, see Figure 7.1.

- vertices without parents (roots) represent inputs;
- vertices without children represent outputs;
- each edge represents an interneuronal connection and has an associated weight: w_h^g for connection from vertex g to h.

Remarks:

➥ Note that the graph is not necessarily connected but in this case inputs and outputs of the unconnected subgraphs should be treated separately as they are independent. While improbable, it may happen that after a network has been trained, some weights become zero such that the corresponding edges may be removed and the initial graph splits into two unconnected subgraphs. However this will usually show a poor choice of the set of inputs.

The network operates as follows:

- each vertex, except roots, calculates the sum of outputs of their parents, weighted by the w_{hg} parameters: $a_h = w_h^g z_g$ (the summation being performed over all parents g of h). Note that in discrete–time approximation the neuron waits for all its inputs to stabilize first, i.e. it will wait till all outputs of its parents are calculated;
- the output of each vertex is calculated by applying its activation function f_h to the weighted sum of inputs: $z_h = f_h(a_h)$. Note that each vertex may have a different activation function.

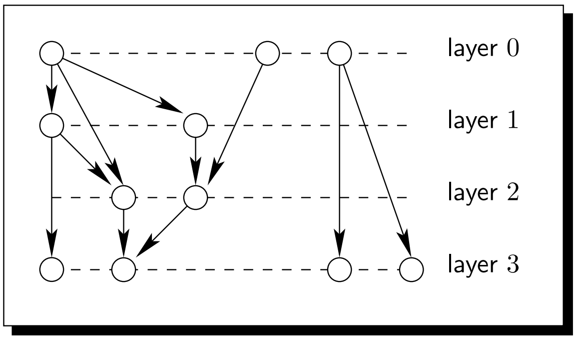

Figure 7.2: *The intermediate step in transformation of a general feedforward to a layered network (see also Figure 7.2).*

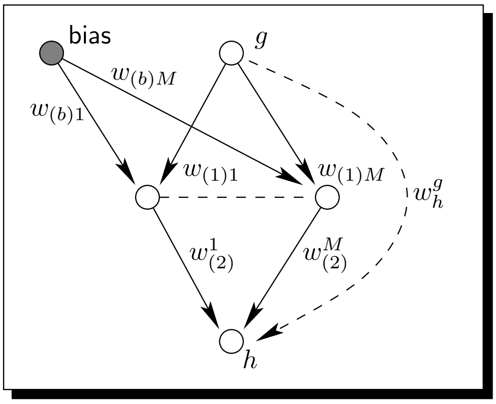

Figure 7.3: *Replacing an edge skipping across layers by an identity mapping subnetwork, bias being used.*

Most general feedforward ANNs used in practice may be approximated arbitrary well by a layered network, where each pair of consecutive layers is fully interconnected and there are no edges skipping over a layer. In particular this is the case if the threshold activation or the logistic activation are used.

Proof. This assertion will be proved by a constructive method:

1. arrange all inputs (root vertices) on layer 0;
2. arrange on layer 1 all vertices who have edges coming only from inputs;
3. arrange on layer 2 all vertices which have edges coming only from layers 0 and 1, continue the same procedure with subsequent layers;
4. move all outputs (vertices without children) on the highest ranked layer.

At this stage the network looks like the one depicted in Figure 7.2.

Now the edges going across several layers remain to be eliminated. They may be replaced with identity mapping networks as follows:

- edges skipping just one layer may be replaced by a (sub)network as depicted in Figure 7.3.

  The activation function is taken to be similar to the one used by other neurons (in backpropagation all neurons use the same activation function); the problem is to find suitable weights so as to obtain at the input of neuron h the same old value $w_h^g z_g$ (there is no summation over g here). In general

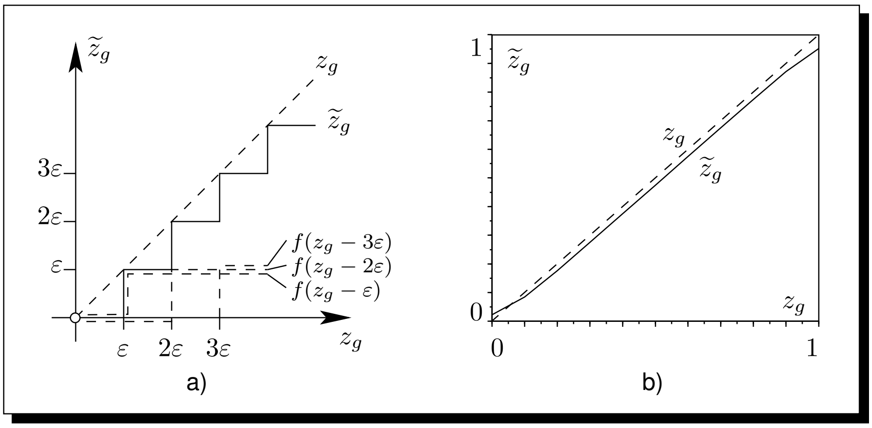

Figure 7.4: *a) approximation of z_g using an identity mapping network with threshold activation functions; b) approximation of z_g using sigmoidal activation ($c = 1$, $\varepsilon = 0.05$).*

the weights should be tuned so as to obtain:

$$w_h^g z_g \simeq \sum_m w_{(2)}^m f(w_{(1)m} z_g + w_{(b)m})$$

❖ $w_{(1)m}$, $w_{(2)}^m$
❖ $w_{(b)m}$

where $\{w_{(1)m}\}$ represents the weights from g to newly introduced neurons, $\{w_{(2)}^m\}$ are the weights from the new neurons to h and $\{w_{(b)m}\}$ are the weights related to bias, see Figure 7.3 on the preceding page.

For particular cases of activation functions the $\{w_{(1)m}\}$, $\{w_{(2)}^m\}$ and $\{w_{(b)m}\}$ weights may be found with relative ease:

❖ M, $\widetilde{z}_g$

a For example considering the threshold activation and $z_g \in [0, 1]$ then it may be approximated to ε precision by setting the total number of neurons to $M = \lceil 1/\varepsilon \rceil$ and the weights as $w_{(1)m} = 1$, $w_{(2)}^m = w_h^g \varepsilon$ and $w_{(b)m} = -m\varepsilon$. Then the approximation function obtained is: $\widetilde{z}_g = \sum\limits_{m=1}^{M} \varepsilon f(z_g - m\varepsilon) = \sum\limits_{m=1}^{M} \varepsilon (z_g - m\varepsilon)_+^0$, see Figure 7.4–a, thus neuron h will receive $w_h^g \widetilde{z}_g$. The process here may be viewed as approximating a linear function by a piecewise constant spline.

b If using logistic activation functions: $f(a) = 1/(1 + e^{-ca})$ then the approximation obtained is smooth. The parameters are set up as: $w_{(1)m} = 1/\varepsilon$, $w_{(2)}^m = w_h^g \varepsilon$, $w_{(b)m} = -m$ and $M = \lceil 1/\varepsilon \rceil$; the resulting function: $\widetilde{z}_g = \sum\limits_{m=1}^{M} \frac{\varepsilon}{1+\exp[-c(\frac{z_g}{\varepsilon} - m)]}$ is depicted in Figure 7.4–b. The ε parameter is correlated to c: for small c the ε should be chosen also small such that c/ε is sufficiently large (M will increase as well). The process here may be viewed as approximation of the linear function $g(x) = x$ by a quasi–interpolant consisting of a sum of shifts of $f(\cdot/\varepsilon)$.

A proof can be constructed by estimating the error between this approximation and the corresponding piecewise constant approximation for the threshold activation. The algebraic details will be omitted but the graph of Figure 7.5 on the next page gives geometric insight into why the procedure can be expected to work.

- For edges skipping several layers: repeat the above procedure. It will be shown later that networks with several hidden layers may approximate better and more easily (using less neurons) more complicated functions[1].

[1] See Section 7.4.

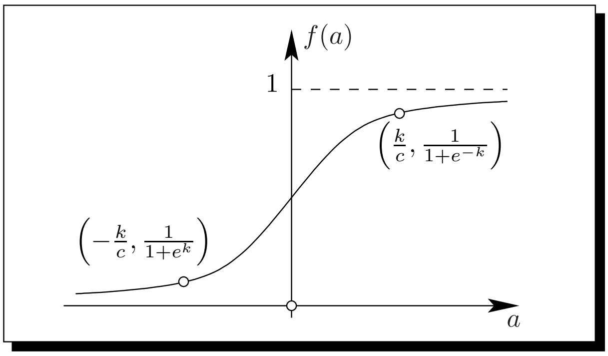

Figure 7.5: *Logistic activation function.*

Finally the procedure is completed by adding all missing edges between two adjacent layers and setting their weights to zero. ☐

Remarks:

- By following the procedure described here the general feedforward network is transformed to one with regularities; this allows it to be analyzed more efficiently at a global level.
- During learning the weights on last layer, associated to identity mappings (e.g. $w^m_{(2)}$) should be allowed to change in order to account for the former weight they've replaced (e.g. w^g_h). All other weights should be kept constant (e.g. by restoring to old values after the global change took place) including the added "0" weights. learning
- In general, a well designed and well trained network should discover by itself how to bring a particular z_g value across several layers if needed. In practice however in some cases training may be eased if connections are established directly, e.g. direct connections between input and output help establishing linearities in the model to be built.
- Because bias is important all networks will be considered as having bias simulated as a supplementary neuron on each layer except output; then $z_{(\ell)0} = 1$ for all layers ℓ including input and except output; in the weight matrix biases will be on column *zero* position (so column numbering starts at 0 while row numbering starts at 1). bias
- A more general proof of the above approximation process could perhaps be built using Kolmogorov's theorem regarding the approximation of functions of several variables by the superposition of a number of functions of one variable (e.g. see [Lor66]). However the theorem doesn't give any hint of how to find the functions and parameters. Kolmogorov's theorem
- Because there is no processing on input vertices then they will be omitted from further representations.

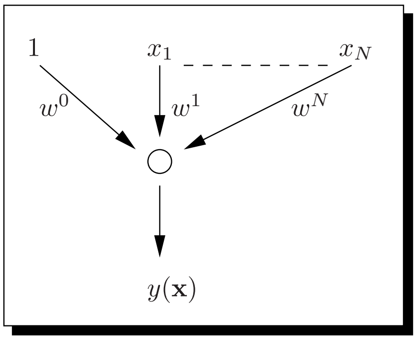

Figure 7.6: *The perceptron.*

7.3 Classification Problems

Perceptron networks are theoretically important in the understanding of the basic functionality of more general feedforward ANN because many networks contain as a limiting case the perceptron ANN (e.g. the logistic activation function converges to the threshold activation function used by perceptron for $c \to \infty$).

7.3.1 The Perceptron, Bias and Linear Separability

The perceptron is defined as a neuron using the threshold activation function, see Figure 7.6.

❖ $\widetilde{\mathbf{x}}$

Let $W = \begin{pmatrix} w^0 & w^1 & \cdots & w^N \end{pmatrix}$ be the matrix of weights (just one row in this case) and let $\widetilde{\mathbf{x}} = \begin{pmatrix} 1 & x_1 & \cdots & x_N \end{pmatrix}^{\mathrm{T}}$. The total input to neuron y is $a = W\widetilde{\mathbf{x}}$ and $y = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{if } a < 0 \end{cases}$ (the case $a = 0$ may be assigned to either of the other two occurrences).

linear separability

Definition 7.3.1. *Consider a set of points defined by vectors $\{\mathbf{x}^p\} \subset \mathbb{R}^N$, classified in a set of finite classes $\{\mathcal{C}_k\}$. The set $\{\mathbf{x}^p\}$ is called linearly separable if the points $\mathbf{x}^p$ may be separated by a set of hyperplanes in $\mathbb{R}^N$ into subdomains such that the interior of each domain will contain only points belonging to one class (p is the index of input vector from the set).*

Proposition 7.3.1. *A single perceptron is able to perform a simple classification of inputs into two classes if and only if the input vectors $\{\mathbf{x}^p\}$ are separable by a single hyperplane in $\mathbb{R}^N$ (see Figure 7.7–b).*

Proof. The input to the perceptron is $a = W\widetilde{\mathbf{x}}$. The decision boundary (where output may switch $0 \leftrightarrow 1$) is defined by $a = W\widetilde{\mathbf{x}} = w_0 + \sum_{i=1}^{N} w_i x_i = 0$, an equation describing a hyperplane in $\mathbb{R}^N$, see Figure 7.7–a.

For any point $\mathbf{x}^p$, not contained in the hyperplane, either $a < 0$ or $a > 0$ thus either $y = 0$ or $y = 1$. When an input vector will be presented to the network zero output will mean that it is on one side of the hyperplane while output 1 will signify that input is on the other side of the hyperplane, see Figure 7.7–b. The points from hyperplane may be assigned to either class (depending upon how the threshold function was defined with respect to $a = 0$ case). □

[7.3.1] See [Has95] pp. 60–65 and [Rip96] pp. 119–120.

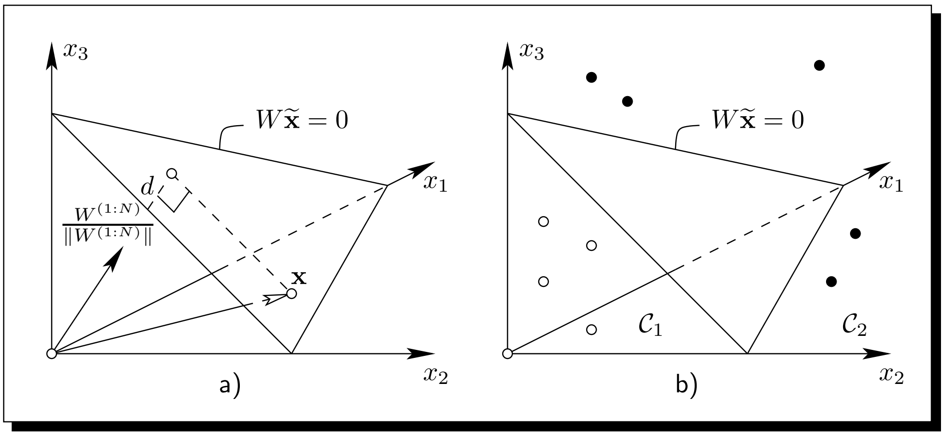

Figure 7.7: *a) The perceptron hyperplane decision boundary in $\mathbb{R}^3$ and the weight vector; b) the perceptron is able to separate two classes $\mathcal{C}_1$ and $\mathcal{C}_2$ by a hyperplane.*

Bias significance and importance

The $W^{(1:N)}$ represents a vector perpendicular on the hyperplane, see Figure 7.7.

Proof. Let $\mathbf{x}^1$ and $\mathbf{x}^2$ be distinct points included in the hyperplane then $W\widetilde{\mathbf{x}}^1 = W\widetilde{\mathbf{x}}^2 = 0$ and $W^{(1:N)}(\mathbf{x}_1 - \mathbf{x}_2) = 0$, i.e. $W^{(1:N)}$ is perpendicular to any vector contained in the hyperplane. ☐

The distance d between the hyperplane and the origin is given by the inner–product between a unit vector perpendicular to hyperplane $\frac{W^{(1:N)}}{\|W^{(1:N)}\|}$ and any point $\mathbf{x}$ on the hyperplane ($W\widetilde{\mathbf{x}} = 0$): ❖ d

$$d = \frac{W^{(1:N)}}{\|W^{(1:N)}\|}\mathbf{x} \quad , \quad W\widetilde{\mathbf{x}} = W^{(1:N)}\mathbf{x} + w^0 = 0 \quad \Rightarrow \quad d = -\frac{w^0}{\|W^{(1:N)}\|}$$

This shows the importance and significance of bias: if bias w^0 is zero or nonexistent then the separating hyperplane has to pass through origin and the number of classification problems linearly separable by a hyperplane passing through the origin is much smaller that the number of general linear separable classification problems.

Perceptron capacity

Consider a perceptron with N inputs which has to classify P vectors. All input vectors belong to one of two classes, i.e. either $\mathcal{C}_1$ or $\mathcal{C}_2$ and the perceptron output indicates to which class the input belongs (e.g. $y(\mathbf{x}) = 0$ for $\mathbf{x} \in \mathcal{C}_1$ and $y(\mathbf{x}) = 1$ for $\mathbf{x} \in \mathcal{C}_2$). The input vectors are points in $\mathbb{R}^N$ space and by Proposition 7.3.1 the perceptron may learn only those cases where the inputs are linearly separable by a single hyperplane. As the number of linearly separable cases is limited so is the learning capacity/memory of a single perceptron.

Definition 7.3.2. *A set $X = \{\mathbf{x}_1, \dots, \mathbf{x}_P\}$ of P points from the space $\mathbb{R}^N$ is in general position if one of the following conditions are met:* general position

- *$P > N$ and no subset of $N+1$ points lies in an $N-1$ dimensional hyperplane (or affine subspace); or*

- *$P \leqslant N$ and no $P-2$ dimensional hyperplane (or affine subspace) contains the set.*

Note that from the definition above:

- Any set of points in general position is a set of distinct points.
- Any subset of a set of points in general position is also in general position.

❖ X, $\mathbf{z}_p$

Proof. Consider the case $P \leqslant N$ then the set $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_P\}$ lies in a $P-1$ dimensional affine subspace. However from the definition it does not lie in any $P-2$ dimensional affine subspace. Thus associating a point $\mathbf{z}_p$ in the $P-1$ dimensional affine space with $\mathbf{x}_p$, and choosing coordinates there so that $\mathbf{z}_1 = \widehat{\mathbf{0}}$, the simplex with vertices $\{\mathbf{z}_1, \ldots, \mathbf{z}_P\}$ has non-zero $P-1$ dimensional volume. Hence each of its faces has non-zero $P-2$ dimensional volume. That is each subset of $P-1$ points from X is in general position.

The argument for the case $P > N$ is similar. □

❖ X, H, δ

It may appear that there are possible differences between separability, allowing points to fall on a separating hyperplane, and a stricter definition requiring that none of the points lie on the hyperplane. However if the collection of points to be separated, X, is finite, and the separation is to be done with a single hyperplane then the definitions coincide. For suppose without loss of generality, one can separate X using a hyperplane H containing some points in H, and classifying such points as being in $\mathcal{C}_1$. Then the finiteness of X means there is a non-zero minimum distance δ from H to the nearest point in X not in H. Hence we can move H the distance $\delta/2$ in the normal direction and achieve strict separation.

❖ $\mathcal{F}(P,N)$

Consider P fixed points in $\mathbb{R}^N$, in *general position*. Each of these points may belong to class $\mathcal{C}_1$ or $\mathcal{C}_2$, the total number of different cases is 2^P (as each point may be in either class, independently of the others). From the 2^P cases some are linearly separable by a single hyperplane and some are not, let $\mathcal{F}(P,N)$ be the number of linearly separable cases (it will turn out that $\mathcal{F}(P,N)$ is independent of the actual positions of the points). Then the probability of linear separability is:

$$\text{Probability of linear separability} = \frac{\mathcal{F}(P,N)}{2^P}$$

Proposition 7.3.2. *The number of linearly separable cases is given by*[2]*:*

$$\mathcal{F}(P,N) = 2\sum_{i=0}^{N} \binom{P-1}{i} \tag{7.1}$$

❖ X

Proof. Let X be a set of P points in general position. Recall from the discussion above that if separability is possible then it can be done with a hyperplane containing no points of X.

❖ $\mathbf{a}$, b, $\mathbf{y}$

Study first the case: $P \leqslant N+1$. Consider a set of points $\{\mathbf{x}_p\}_{p\in\overline{1,P}}$ each point of which belongs either to a class $\mathcal{C}_1$ or $\mathcal{C}_2$. We seek a hyperplane with parameters $\mathbf{a} \in \mathbb{R}^N$ and $b \in \mathbb{R}$ such that $\begin{cases} \mathbf{x}_p^{\mathrm{T}}\mathbf{a} + b = +1 & \text{if } \mathbf{x}_p \in \mathcal{C}_1 \\ \mathbf{x}_p^{\mathrm{T}}\mathbf{a} + b = -1 & \text{if } \mathbf{x}_p \in \mathcal{C}_2 \end{cases}$. Let $\mathbf{y} \in \mathbb{R}^P$ be a vector such that $\begin{cases} y_p = +1 & \text{if } \mathbf{x}_p \in \mathcal{C}_1 \\ y_p = -1 & \text{if } \mathbf{x}_p \in \mathcal{C}_2 \end{cases}$. Then to find the required hyperplane is equivalent to solving the system of equations:

$$\begin{bmatrix} \mathbf{x}_1^{\mathrm{T}} & 1 \\ \vdots & \vdots \\ \mathbf{x}_P^{T} & 1 \end{bmatrix} \begin{bmatrix} \mathbf{a} \\ b \end{bmatrix} = \mathbf{y}\ , \tag{7.2}$$

[2] For $x \in \mathbb{R}$, $n \in \mathbb{Z}^+$, $\binom{x}{n} \equiv \begin{cases} 1\ , & n = 0 \\ \frac{x(x-1)\cdots(x-n+1)}{n!}\ , & n > 0 \end{cases}$.

for $\mathbf{a}$ and b. The assumption that the points are in general position implies that the matrix:

$$\begin{bmatrix} \mathbf{x}_2^{\mathrm{T}} - \mathbf{x}_1^{\mathrm{T}} & 0 \\ \vdots & \vdots \\ \mathbf{x}_P^{\mathrm{T}} - \mathbf{x}_1^{\mathrm{T}} & 0 \end{bmatrix}$$

has full rank. Hence the matrix of equation (7.2) has rank P. In particular the column rank is P and thus the system (7.2) always has at least one solution.

Hence there is always a way to separate the two classes with a single hyperplane (there will indeed be infinitely many separating hyperplanes) and:

$$\mathcal{F}(P, N) = 2^P \quad \text{for} \quad P \leqslant N + 1 \; .$$

Consider now the case $P \geqslant N + 1$. This case will be proven by induction.

Induction basis: The previous analysis provides the formula for $P = N + 1$.

Induction step: assume the result (7.1) is true for some $P \geqslant N + 1$. We seek a recursive formula for $\mathcal{F}(P + 1, N)$. Consider a set X of $P + 1$ points in general position. Suppose that the first P points are linearly separable (relative to the classes being considered). That is we have an case from $\mathcal{F}(P, N)$ and a new point is being added. There are two cases to be considered:

- The new point lies on one side only of any separating hyperplane for the P point set. Then the new set is separable only if the new point is on the "correct" side, i.e. the appropriate one for its class.
- If the above it not true then the first P points can be linearly separated by a hyperplane $\mathcal{A}$ that passes through the $(P + 1)$–st point. Then, as discussed above, a small perturbation of the hyperplane $\mathcal{A}$ will correctly classify all $P + 1$ points. ❖ $\mathcal{A}$

  To calculate how many cases fall in this category consider again only the first P points and the hyperplane $\mathcal{A}$. If these P points are projected into a hyperplane $\mathcal{B} \perp \mathcal{A}$ then these projection points contained in $\mathcal{B}$ are linearly separated by the $N - 2$ dimensional hyperline[3] defined by the intersection of $\mathcal{A}$ and $\mathcal{B}$. This means that the number of possibilities for the first P points in this situation is $\mathcal{F}(P, N - 1)$. ❖ $\mathcal{B}$

For any occurrence of the first case only one assignment of class to the $(P + 1)$–st point gives a linearly separable set of $P + 1$ points. However in the second case any assignment of class to the $(P + 1)$–st point gives a linearly separable set of $P + 1$ points. Hence:

$$\mathcal{F}(P + 1, N) = 1 \times \text{number of occurrences of first case} + 2 \times \text{number of occurrences of second case}$$

Since the total number of occurrences is $\mathcal{F}(P, N)$ and the number of occurrences of the second case is $\mathcal{F}(P, N - 1)$ then the number of occurrences of the first case is $\mathcal{F}(P, N) - \mathcal{F}(P, N - 1)$. We see that:

$$\mathcal{F}(P + 1, N) = [\mathcal{F}(P, N) - \mathcal{F}(P, N - 1)] + 2\mathcal{F}(P, N - 1) = \mathcal{F}(P, N) + \mathcal{F}(P, N - 1) \qquad (7.3)$$

From the induction hypothesis: $\mathcal{F}(P, N - 1) = 2 \sum\limits_{i=0}^{N-1} \binom{P-1}{i} = 2 \sum\limits_{i=1}^{N} \binom{P-1}{i-1}$ and then, using also (7.3) and (7.1), the expression for $\mathcal{F}(P + 1, N)$ becomes:

$$\mathcal{F}(P + 1, N) = \mathcal{F}(P, N) + \mathcal{F}(P, N - 1) = 2 + 2\sum_{i=1}^{N} \left[\binom{P-1}{i} + \binom{P-1}{i-1} \right] = 2\sum_{i=0}^{N} \binom{P}{i}$$

i.e. is of the same form as (7.1) (the property $\binom{P-1}{i} + \binom{P-1}{i-1} = \binom{P}{i}$ was used here). □

The probability of linear separability is then:

$$P_{\substack{\text{linear}\\\text{separability}}} = \begin{cases} 1 \, , & P \leqslant N + 1 \\ \frac{1}{2^{P-1}} \sum\limits_{i=0}^{N} \binom{P-1}{i} \, , & P \geqslant N + 1 \end{cases} \Rightarrow P_{\substack{\text{linear}\\\text{separability}}} \begin{cases} > 0.5 & \text{for } P < 2(N + 1) \\ = 0.5 & \text{for } P = 2(N + 1) \\ < 0.5 & \text{for } P > 2(N + 1) \end{cases}$$

[3]The dimension of hyperline is $N - 2$ by the "rank + nullity" theorem.

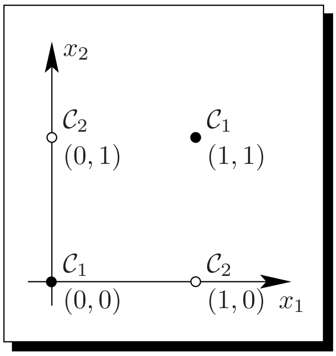

Figure 7.8: *The XOR problem. The vectors marked with black circles are from one class; the vectors marked with white circles are from the other class. $\mathcal{C}_1$ and $\mathcal{C}_2$ are not linearly separable.*

that is the memory capacity of a single perceptron is around $2(N+1)$.

❖ S_m

Proof. Let $S_m \equiv \sum\limits_{i=0}^{m} \binom{P-1}{i}$ (for some m). Consider first the case $P = 2(N+1)$:

$$2^{P-1} = (1+1)^{P-1} = S_{P-1} = 2S_{\frac{P}{2}-1}$$

because P is even, S_{P-1} contains P terms and $\binom{P-1}{i} = \binom{P-1}{P-1-i}$. Then $S_{\frac{P}{2}-1} = 2^{P-2}$ and $P_{\substack{\text{linear}\\\text{separability}}} = \frac{S_N}{2^{P-1}} = \frac{S_{\frac{P}{2}-1}}{2^{P-1}} = \frac{1}{2}$.

For $P < 2(N+1)$ it follows that $S_N = S_{\frac{P}{2}-1} +$ additional terms thus $P_{\substack{\text{linear}\\\text{separability}}} > 0.5$; the case $P > 2(N+1)$ being similar. □

Remarks:

- As the points (vectors) from the same class are usually (to some extent) correlated then the "practical" memory capacity of a single perceptron is much higher than the above reasoning suggests.
- For $P = 4$ and $N = 2$ the total number of cases is $2^4 = 16$ out of which 14 are linearly separable. One of the two cases which may not be linearly separated is depicted in Figure 7.8, the other one is its mirror image. This is the *exclusive-or* (XOR), in the bidimensional space. The vectors $(0,0)$ and $(1,1)$ are from one class (0 xor $0 = 0$, 1 xor $1 = 0$); while the vectors $(0,1)$ and $(1,0)$ are from the other (1 xor $0 = 1$, 0 xor $1 = 1$).

7.3.2 Two–layered Perceptron and Convex Decision Domains

Consider a network with one hidden layer and a single output, made out of perceptrons, see Figure 7.9 on the facing page.

Proposition 7.3.3. *A 2–layered perceptron network may perform a classification for arbitrary convex polygonal decision boundary, given enough hidden neurons.*

7.3.2 See [Has95] pp. 41–43.

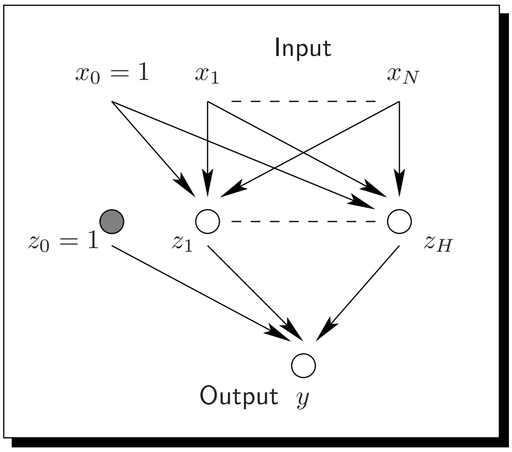

Figure 7.9: *The 2–layer perceptron network, biases are represented by* $x_0 = 1$ *and* $z_0 = 1$.

Proof. Recall that a perceptron network involves the threshold activation function.

A single layer perceptron neural network (with one output) has a decision boundary which is a hyperplane. Let the hidden layer represent the hyperplanes (see Proposition 7.3.1).

Then the output layer may perform an logical AND between the outputs of the hidden layer to decide if the vector is inside the decision region or not — each output neuron representing a class. See Figure 7.10 on the next page.

The functionality of AND operation may be achieved as follows:

- Set all $W_{(1)}$ weights from input to hidden layer such that the output of a hidden neuron will be 1 if the point is on the correct side of the hyperplane and zero otherwise; then the output of hidden layer is $\widehat{\mathbf{1}}$ if input $\mathbf{x}$ belongs to the class to be represented.
- Set up the $W_{(2)}$ weights from hidden layer to output as: $w^h_{(2)} = 1$ for $h \neq 0$ and bias as $w^0_{(2)} = -H+\varepsilon$, where $1 > \varepsilon \gtrsim 0$; then the total input to y neuron is $a_y = \widehat{\mathbf{1}}^{\mathrm{T}}\mathbf{z} - H + \varepsilon$ and $a_y > 0$ $(y = 1)$ if and only if $\mathbf{z} = \widehat{\mathbf{1}}$. □

More generally: a 2–layered perceptron network may perform a one–of–k encoding (approximate) classification, provided that there are enough hidden neurons, *classes do not overlap* (i.e. an input vector $\mathbf{x}$ may belong to one class only) and decision areas are convex (*a convex area may be approximated arbitrarily well by a polygon*).

Proof. Set up the hidden layer to have enough perceptrons to represent the necessary hyperplanes, *for each class separately*, e.g. if each class k requires H hyperplanes then the required number of hidden neurons would be KH.

Establish connections between all hidden neurons to represent hyperplanes from the same class to the output neuron responsible for that class (there are no connections between hidden neurons representing *different* classes). Set up the weights as discussed in proof of Proposition 7.3.3. □

7.3.3 Three–layered Perceptron and Arbitrary Decision Boundaries

Proposition 7.3.4. *A three–layered perceptron network with two hidden layers and one output may perform a classification for arbitrary decision boundaries, given enough hidden neurons.*

[7.3.3] See [Has95] pp. 35–54.

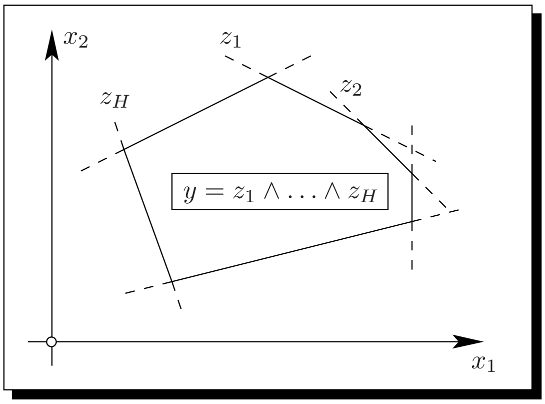

Figure 7.10: *In a 2–layer perceptron network the hidden layer may approximate the hyperplane decision boundaries and then the output may perform a logical AND to establish if the input vector is within the decision region. Thus any polygonal convex decision area may be represented by a two layer network.*

Proof. Remark: Here we mean an approximate classification with the misclassified region being arbitrary small.

Note: A bounded measurable set A can be approximated arbitrary well by a finite union of disjoint hypercubes B_i $(i \in \{1, \ldots, N\})$, i.e. if $\triangle$ is the symmetric difference (all points in A or $\bigcup_{i=1}^{N} B_i$ but not both) then the volume $V(A \triangle \bigcup_{i=1}^{N} B_i) < \varepsilon$.

❖ X

The network has to decide if an input is included in some bounded measurable subset $X \in \mathbb{R}^N$ from where all vectors belong to the same class.

Noting the remark at the start of the proof X is approximated by a finite union of disjoint hypercubes.

The neural network is built as follows:

- The *first hidden layer* contains a group of $2N$ neurons for each hypercube (2 hyperplanes for each pair of faces);
- The *second hidden layer* contains H neurons, each receiving the input from the corresponding group of $2N$ neurons from the first hidden layer;
- The output layer receives inputs from all H neurons from the second hidden layer.

see Figure 7.11 on the facing page.

By the same means as described in previous section, second layer perceptrons may decide if the input $\mathbf{x}$ is within the hypercube represented by them. As hypercubes do not overlap then the output of second hidden layer is either of the form $\mathbf{z}_{(2)} = \mathbf{e}_h$ or $\mathbf{z}_{(2)} = \widehat{\mathbf{0}}$. Then the output neuron may detect if input was in any hypercube if the $W_{(2)}$ weights (from second hidden layer to output) are set as follows: $w^h_{(2)} = 1$ for $h \neq 0$ and $w^0_{(2)} = -1 + \varepsilon$ for bias, where $1 > \varepsilon \gtrsim 0$. Thus input to y neuron is $a_y > 0$ for $\mathbf{z}_{(2)} \neq \widehat{\mathbf{0}}$ and $a_y < 0$ otherwise and y output is obtained as required after applying the threshold activation. □

More generally: a 3–layered perceptron network may perform a one–of–k encoding classification, provided that there are enough hidden neurons and *classes do not overlap* (the proof is identical to the one discussed in previous section).

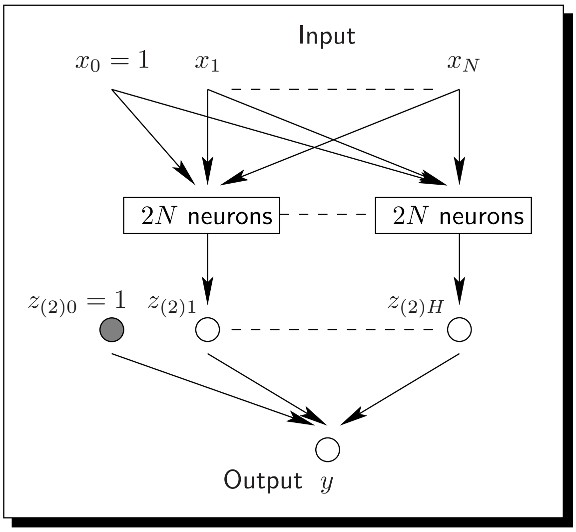

Figure 7.11: *The 3–layer perceptron network for classification of arbitrary decision boundary.*

7.4 Regression Problems

Lemma 7.4.1. *Let $X \subset \mathbb{R}^N$ be a closed bounded set and $f : X \to \mathbb{R}$ be continuous. Then f may be approximated arbitrary well by a piecewise constant spline s on X* ❖ X, f, s

Proof. A closed bounded subset of $\mathbb{R}^N$ is compact. A function continuous on a compact set is uniformly continuous there. Hence given any $\varepsilon > 0$ there exists $\delta > 0$ such that $\mathbf{x}, \mathbf{y} \in X$ and $\|\mathbf{x} - \mathbf{y}\|_\infty \leqslant \delta$ together imply $|f(x) - f(y)| < \varepsilon$. ❖ ε, δ

For each N–tuple of integers $\mathbf{m} = \begin{pmatrix} m_1 & \dots & m_N \end{pmatrix}$ define the half open hypercube ❖ $\mathbf{m}$, $\mathcal{C}_\mathbf{m}$

$$\mathcal{C}_\mathbf{m} = \{\mathbf{x} : m_i\delta \leqslant x_i < (m_i + 1)\delta \,,\; 1 \leqslant i \leqslant N\}$$

These cubes tile $\mathbb{R}^N$. Then wherever $X \cap \mathcal{C}_\mathbf{m} \neq \emptyset$ choose an $\mathbf{x}_\mathbf{m} \in X \cap \mathcal{C}_\mathbf{m}$. Define ❖ s, $\Psi_{\mathcal{C}_\mathbf{m}}$

$$s(\cdot) = \sum_\mathbf{m} f(\mathbf{x}_\mathbf{m})\, \Psi_{\mathcal{C}_\mathbf{m}}(\cdot)$$

where $\Psi_{\mathcal{C}_\mathbf{m}}(\cdot)$ is the characteristic function $\Psi_{\mathcal{C}_\mathbf{m}}(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{x} \in \mathcal{C}_\mathbf{m} \\ 0 & \text{if } \mathbf{x} \notin \mathcal{C}_\mathbf{m} \end{cases}$. This defines a piecewise constant spline s. Now given $\mathbf{x} \in X$ choose $\mathbf{k}$ so that $\mathbf{x} \in \mathcal{C}_\mathbf{k}$. Then

$$|f(\mathbf{x}) - s(\mathbf{x})| = \left| f(\mathbf{x}) - \sum_\mathbf{m} f(\mathbf{x}_\mathbf{m})\, \Psi_{\mathcal{C}_\mathbf{m}}(\mathbf{x}) \right| = \left| f(\mathbf{x}) - f(\mathbf{x}_\mathbf{k})\, \Psi_{\mathcal{C}_\mathbf{k}}(\mathbf{x}) \right| = |f(\mathbf{x}) - f(\mathbf{x}_\mathbf{k})| < \varepsilon$$

since $\|\mathbf{x} - \mathbf{x}_\mathbf{k}\|_\infty \leqslant \delta$. □

Proposition 7.4.1. *A two layer neuronal network, with a hidden perceptron layer and an identity activation on output, can approximate arbitrary well any continuous function $y : X \to Y$, where X is a closed bounded set in $\mathbb{R}^N$ and Y is a closed bounded region in $\mathbb{R}$, provided that there are enough hidden neurons.*

Proof. y is a continuous function on a closed bounded set so y is uniformly continuous. According to Lemma 7.4.1 y may be approximated arbitrary well by a piecewise constant spline s.

[7.4] See [Has95] pp. 46–50, 296–301 and [Bis95] pp. 128–130.

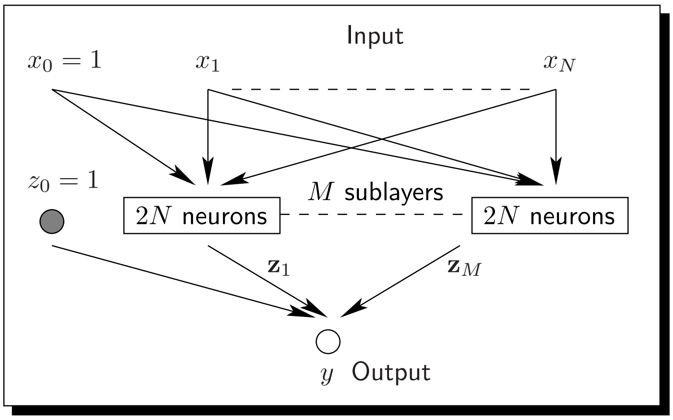

Figure 7.12: *The 2–layer network for arbitrary function approximation.*

The architecture of the neural network is very similar to the one used in the proof of Proposition 7.3.3:

❖ M, m, $\mathcal{C}_m$

- The X domain is approximated by a set of M hypercubes $\mathcal{C}_m$, $1 \leqslant m \leqslant M$.
- The hidden layer is composed out of M sublayers with $2N$ neurons each (i.e. the total number of neurons on the hidden layer is $M \times 2N$). See Figure 7.12.
- Each $\mathcal{C}_m$ hypercube is defined by $2N$ hyperplanes. Let these hyperplanes be defined by $x_{n(m)} =$ const and $x'_{n(m)} =$ const, where $x_{n(m)}$ and $x'_{n(m)}$ are ordered, i.e. $x_{n(m)} < x'_{n(m)}$, $\begin{cases} x_{n(m)} < x_{n(m')} \\ x'_{n(m)} < x'_{n(m')} \end{cases}$ if $m < m'$, $\forall n$ and $x'_{n(m)} < x_{n(m+1)}$, $\forall m \in \{1, \dots, M\}$ (i.e. $\mathcal{C}_m$ is closed to the "left" and open to the "right"). We will consider $x_{n(m)} =$ const. plane as part of $\mathcal{C}_m$ but not the plane $x'_{n(m)} =$ const.
- The way X was tiled by $\mathcal{C}_m$ means that an input $\mathbf{x}$ may belong to only one $\mathcal{C}_k$ hypercube. Then by a procedure similar to the one used in Proposition 7.3.3 we can arrange things such that only the corresponding hidden sublayer has nonzero output:

  $$\mathbf{x} \in \mathcal{C}_k \;\Rightarrow\; \mathbf{z}_k = \widehat{\mathbf{1}}_{2N} \quad ; \quad \mathbf{z}_m = \widehat{\mathbf{0}}_{2N} \text{ for } m \neq k$$

  See Figure 7.12.
- We want to approximate $y(\mathbf{x})$ by a piecewise spline $s(\mathbf{x})$. This can be performed if the weights from the corresponding k "winning" sublayer are set to the value $\frac{s(\mathbf{x})}{2N}$, we use the identity function on output neuron and the weight associated with bias is set to zero. □

More generally: a 2–layered network network may approximate arbitrary well a continuous vectorial function $\mathbf{y} : \mathbb{R}^N \to \mathbb{R}^K$ provided that there are enough hidden neurons.

Proof. Each component of vectorial function may be represented by a different output neuron with the corresponding weights set up accordingly (see also proof of Proposition 7.4.1). □

Remarks:

➥ It is possible to build a three–layer network, using logistic activation which will give a localized output, provided that there are enough hidden neurons.

Let $W_{(1)}$ be the weights from input to first hidden layer. The output $z_{(1)h}$ of a neuron from first hidden layer is: $z_{(1)h} = \frac{1}{1+\exp(-cW_{(1)h}^{(:)}\mathbf{x})}$, see Figure 7.13–a.

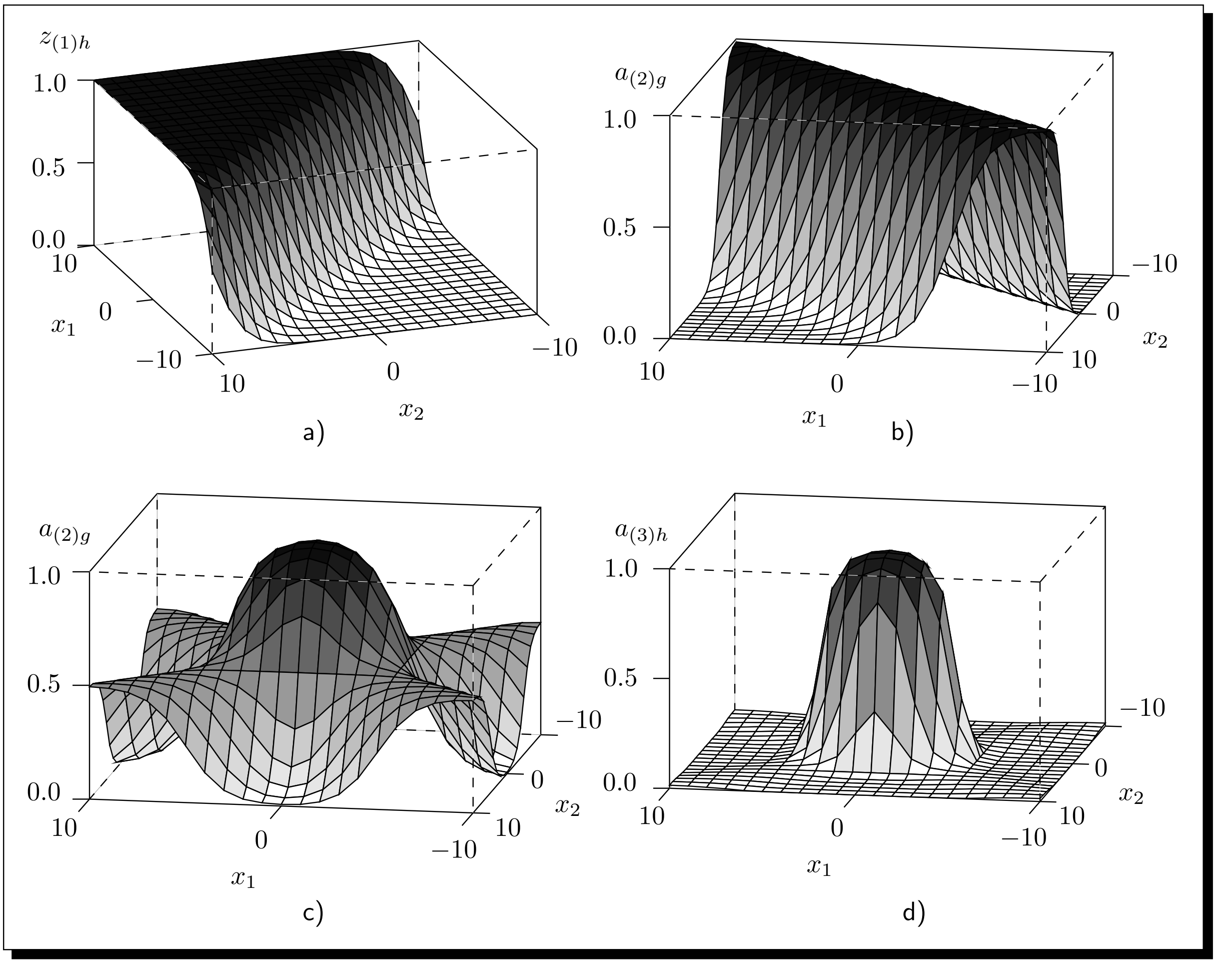

Figure 7.13: *Two dimensional input space. Figure a) shows the output of a neuron from first hidden layer as:* $z_{(1)h} = \frac{1}{1+e^{-x_1-x_2}}$. *Figure b) shows the linear combination* $a_{(2)g} = \frac{1}{1+e^{-x_1-x_2-5}} - \frac{1}{1+e^{-x_1-x_2+5}}$ *as they could be received from first hidden neurons with different biases (i.e.* ± 5*). Figure c) shows a linear combination of 4 logistic functions* $a_{(2)g} = \frac{0.5}{1+e^{-x_1-x_2-5}} - \frac{0.5}{1+e^{-x_1-x_2+5}} + \frac{0.5}{1+e^{-x_1+x_2-5}} - \frac{0.5}{1+e^{-x_1+x_2+5}}$ *(*$W_{(2)}$ *weights are* ± 0.5*). Figure d) shows the output after applying the logistic function again* $a_{(3)h} = \frac{1}{1+e^{-14a_{(2)g}+11.2}}$*, only the central maximum remains, this represents the localized output of the second hidden layer.*

By making linear combinations when entering the second neuronal layer, it is possible to get an activation $a_{(2)g}$ with an absolute maximum, see Figure 7.13–b and 7.13–c.

By applying the logistic function on the second layer, i.e. when exiting the second layer, it is possible to get just a localized output, i.e. just the maximum of the linear combination. See Figure 7.13–d. So for any given input $\mathbf{x}$ *only one neuron from second hidden layer will have a significant output*, i.e. second layer acts as a competitive layer.

The third layer will combine the weights $W_{(3)}$, corresponding to connection from second hidden layer neuron to output, to perform the required approximation, e.g. if the output of the second layer is of the form $\mathbf{z}_{(2)} = \mathbf{e}_g$ and the activation function is identity then the network output will be $\mathbf{y} = W_{(3)g}^{(:)\ \mathrm{T}}$, where g is the winner on second hidden layer.

➥ From the above remark it follows that a competitive layer may be replaced by a two–layer subnetwork and a two–layer (sub)network may be replaced by a competitive layer when such functionality is required.

7.5 Learning

supervised learning

Learning refers to the process of weights adaptation in order to achieve the desired goal. The learning process is usually supervised meaning that the training set contains both inputs $\{\mathbf{x}^p\}$ and their associated targets $\{\mathbf{t}^p\}$.

error

In order to assess quantitatively the learning process it is necessary to have a measure showing the distance between actual achievement $\{\mathbf{y}(\mathbf{x}^p)\}$ (i.e. actual network output given the training inputs) and targets $\{\mathbf{t}^p\}$ — this measure is called the *error* and denoted by E. For a given network and training set, network outputs are dependent only on weights hence error is a function of the weights $E = E(W)$.

The purpose of learning is to achieve generalization, i.e. the trained network should give good results for previously unseen inputs. Then different training sets should lead (roughly) to the same result. This means that each training pair $(\mathbf{x}^p, \mathbf{t}^p)$ should have an equal contribution to error and then the error may be written as $E = \sum\limits_p E_p$, where E_p is the contribution to error from $(\mathbf{x}^p, \mathbf{t}^p)$.

7.5.1 Perceptron Learning

The *perceptron* (or *adaline*[4]) represents a neuron with a threshold activation function (see also Section 7.3.1), usually chosen as being essentially odd:

$$f(a) = \begin{cases} +1 & \text{for } a \geqslant 0 \\ -1 & \text{for } a < 0 \end{cases}$$

[7.5.1] See [Has95] pp. 57–65 and [Bis95] pp. 98–105.

[4] From ADAptive LINear Element.

The Error Function

The neuronal output is $y(\mathbf{x}) = f(W\mathbf{x})$ where (in this particular case) W *is represented by a one–row matrix and* W^{T} *may be considered as belonging to the same space as input vectors:* $W^{\mathrm{T}}, \mathbf{x} \in \mathbb{R}^N$. Because the output of the single neuron is either $+1$ or -1 then the perceptron may classify at most two classes: if $W\mathbf{x} \geqslant 0$ then the output is $+1$ and $\mathbf{x} \in \mathcal{C}_1$, else $W\mathbf{x} < 0$, the output is -1 and $\mathbf{x} \in \mathcal{C}_2$. ❖ W^{T}

We want W such that for a *correct* classification, $tW\mathbf{x} > 0$, $\forall \mathbf{x}$ correctly classified (t being the target value given the input vector $\mathbf{x}$) and for a *misclassified* input vector either $W\mathbf{x} > 0$ while $t = -1$ or vice-versa, i.e. $tW\mathbf{x} < 0$, $\forall \mathbf{x}$ misclassified. Thus a good choice for the error function will be: ❖ t

$$E(W) = - \sum_{\mathbf{x}^p \in \mathfrak{M}} t^p W \mathbf{x}^p \tag{7.4}$$

where $\mathfrak{M}$ is the set of all misclassified $\mathbf{x}^p$. ❖ $\mathfrak{M}$

Remarks:

➥ From the discussion in Section 7.3.1: for a point $\mathbf{x}_0$ within the separation hyperplane it follows that $W\mathbf{x}_0 = \widehat{\mathbf{0}}$ so W^{T} is perpendicular on the hyperplane and $W\mathbf{x}$ is proportional to the distance from $\mathbf{x}$ to the hyperplane representing the decision boundary.

The process of minimizing the function (7.4) is equivalent to shifting the decision boundary such that the total distance between misclassified vectors and the hyperplane becomes minimum. During the shifting process the set of misclassified patterns changes as some previously misclassified vectors becomes correctly classified and vice-versa.

The Learning Algorithm

The error gradient is:

$$\frac{\partial E_p}{\partial w_i} = \begin{cases} -t^p x_i^p & \text{if } \mathbf{x}^p \text{ is misclassified} \\ 0 & \text{if } \mathbf{x}^p \text{ is correctly classified} \end{cases}$$

and then $\nabla E_p = -t\mathbf{x}^p$ if $\mathbf{x}^p$ is misclassified or $\nabla E_p = \widehat{\mathbf{0}}$ otherwise; note that ∇E_p is from the same $\mathbb{R}^N$ space as $\mathbf{x}$ and W^{T}.

As the error gradient shows the direction of error increase then, in order to decrease E, it is necessary to move W^{T} in opposite direction. The learning rule is:

$$\Delta W^{\mathrm{T}} = W^{\mathrm{T}}_{(t+1)} - W^{\mathrm{T}}_{(t)} = -\eta \nabla E = \begin{cases} \eta t^p \mathbf{x}^p & \text{if } \mathbf{x}^p \text{ is misclassified} \\ \widehat{\mathbf{0}} & \text{if } \mathbf{x}^p \text{ is correctly classified} \end{cases} \tag{7.5}$$

i.e. all training vectors are tested: if the $\mathbf{x}^p$ is *correctly classified* then W is left *unchanged*, otherwise it is "adapted" and the process is repeated until the error is minimized (in general not all vectors are classified correctly at the end of learning). See Figure 7.14 on the next page. Equation (7.5) is also known as the *delta rule* (see also Section 7.5.2). η is a *positive* parameter governing the speed and quality of learning. The algorithm is started by choosing $W_{(0)}$ randomly. ❖ η, $W_{(0)}$

The contribution made by the *same* misclassified $\mathbf{x}$ to the error function (7.4) decreases after an updating step (7.5).

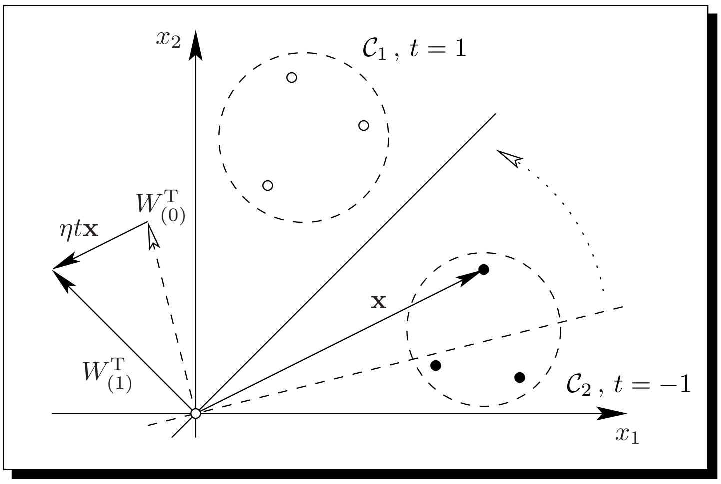

Figure 7.14: *The learning process for a perceptron in a bidimensional space. White circles represents patterns from one class while black ones represents the other class. Initially the parameter is $W_{(0)}$ and the pattern shown by $\mathbf{x}$ is misclassified. Then $W_{(1)}^{\mathrm{T}} = W_{(0)}^{\mathrm{T}} - \eta \mathbf{x}$; η was chosen $1/3$ and note that $t = -1$. The decision boundary is always perpendicular to W^{T} vector (see Section 7.5.2) and during learning moves following the dotted arrow. The other case of a misclassified $\mathbf{x} \in \mathcal{C}_1$ is similar.*

Proof. The contribution to E, from (7.4), after one learning step using (7.5), are:

$$-tW_{(t+1)}\mathbf{x} = -tW_{(t)}\mathbf{x} - \eta t^2 \|\mathbf{x}\|^2 < -tW_{(t)}\mathbf{x}$$

As $\eta > 0$ then $\eta t^2 \|\mathbf{x}\|^2 > 0$ and thus the contribution of $\mathbf{x}$ to E decreases (it was assumed that $\mathbf{x}$ is misclassified both at t and $t + 1$). □

Proposition 7.5.1. ***Convergence of learning.*** *If the classification problem is linearly separable then the learning process defined by* (7.5) *stabilizes after a finite number of steps, i.e. the algorithm is convergent and all training vectors become correctly classified.*

Proof. **1.** Consider first the case where the separating hyperplane may pass through the origin.

❖ W_{opt}

As the problem is linearly separable there does exists a W_{opt} solution such that:

$$t^p W_{\text{opt}} \mathbf{x}^p > 0 \quad , \quad \forall p = \overline{1, P}$$

i.e. all patterns are correctly classified.

❖ $W_{(0)}$, η

Without any loss of generality it is assumed that $W_{(0)}^{\mathrm{T}} = \widehat{\mathbf{0}}$ and $\eta = 1$. Then for misclassified $\mathbf{x}^q$ vectors at step t (see (7.5)):

$$W_{(t+1)}^{\mathrm{T}} = W_{(t)}^{\mathrm{T}} + \sum_{\mathbf{x}^q \in \mathfrak{M}_{(t)}} t^q \mathbf{x}^q$$

❖ $\mathfrak{M}_{(t)}$, $\tau_{(t)p}$

where $\mathfrak{M}_{(t)}$ is the set of all misclassified vectors after the step t update. Consider that each vector has been misclassified $\tau_{(t)p}$ times during the t discrete time steps (including $t = 0$ and the last t step). Then the weight vector may be written as:

$$W_{(t+1)}^{\mathrm{T}} = \sum_{p=1}^{P} \tau_{(t)p} t^p \mathbf{x}^p$$

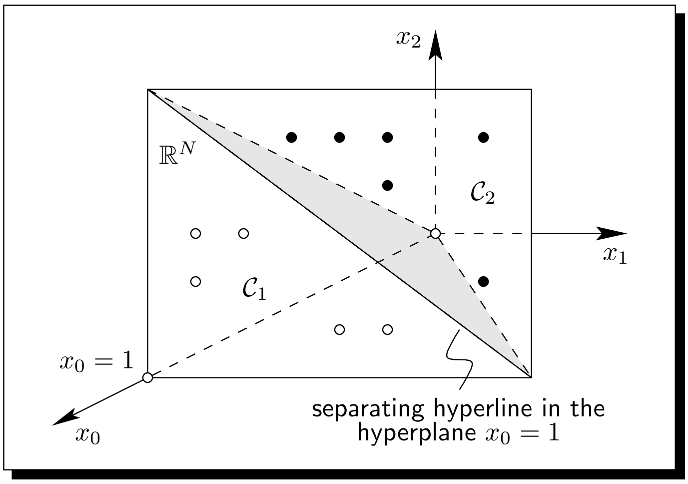

Figure 7.15: *If the separating plane does not go through the origin then the perceptron needs to use bias in order to be able to classify correctly (here $N = 2$ and $\mathcal{C}_1$ and $\mathcal{C}_2$ cannot be separated in $\mathbb{R}^2$ by a line going through origin, the hatched surface represents the separating plane in $\mathbb{R}^3$, all pattern vectors are contained in the plane defined by $x_0 = 1$).*

Note that while W changes, the decision boundary changes and a training pattern vector may move from being correctly classified to being misclassified and back; the training set may be used several times in any order.

By multiplying with W_{opt} to the left:

$$W_{\mathsf{opt}} W_{(t+1)}^{\mathrm{T}} = \sum_{p=1}^{P} \tau_{(t)p} t^p W_{\mathsf{opt}} \mathbf{x}^p \geqslant \left(\sum_{p=1}^{P} \tau_{(t)p} \right) \min_p t^p W_{\mathsf{opt}} \mathbf{x}^p$$

such that this product is limited from below by a function linear in $\tau = \sum\limits_{p=1}^{P} \tau_{(t)p}$ — and thus $W_{(t+1)}$ is ❖ τ *limited from below* as W_{opt} is constant. On the other hand:

$$\|W_{(t+1)}\|^2 = \|W_{(t)}\|^2 + \sum_{\mathbf{x}^q \in \mathfrak{M}_{(t)}} (t^q)^2 \|\mathbf{x}^q\|^2 + 2W_{(t)} \sum_{\mathbf{x}^q \in \mathfrak{M}_{(t)}} t^q \mathbf{x}^q \leqslant \|W_{(t)}\|^2 + \sum_{\mathbf{x}^q \in \mathfrak{M}_{(t)}} \|\mathbf{x}^q\|^2$$

as $t^q = \pm 1$ and for a misclassified vector $t^q W_{(t)} \mathbf{x}^q < 0$. Therefore:

$$\Delta \|W_{(t+1)}\|^2 = \|W_{(t+1)}\|^2 - \|W_{(t)}\|^2 \leqslant \sum_{\mathbf{x}^q \in \mathfrak{M}_{(t)}} \max_p \|\mathbf{x}^p\|^2 \xrightarrow{\text{sum over } t} \|W_{(t+1)}\|^2 \leqslant \tau \max_p \|\mathbf{x}^p\|^2$$

i.e. $\|W_{(t+1)}\|$ is *limited from above* by a function linear in $\sqrt{\tau}$.

Considering both limitations (below by τ and above by $\sqrt{\tau}$) it follows that no matter how large t is, i.e. no matter how many update steps are taken, τ has to be limited (because τ from below grows faster than $\sqrt{\tau}$ from above, during training) and hence at some stage τ becomes stationary and thus (because W_{opt} was presumed to exists) all training vectors become correctly classified.

2. Consider now the case where the separating hyperplane cannot contain the origin. In this case bias must be used. See Figure 7.15.

Bias may be inserted as follows: consider $\mathbf{x} \to \widetilde{\mathbf{x}} \in \mathbb{R}^{N+1}$ where $\widetilde{x}_0 = 1 =$ const and $\widetilde{x}_i = x_i$ for $\forall i = \overline{1, N}$ and also a similar transformation for weights $W \to \widetilde{W} \in \mathbb{R}^{N+1}$ where $\widetilde{w}^0$ holds the bias.

All the results previously established will hold. The difference is that while the separating hyperplane has to pass through the origin of $\mathbb{R}^{N+1}$ space it doesn't necessarily have to pass through the origin of $\mathbb{R}^N$ space (containing the $\mathbf{x}$ vectors). See Figure 7.15 on the page before. ☐

Remarks:

- ➥ The conventional wisdom is that the learning algorithm is good at generalization as long as the training set is sufficiently general.

7.5.2 Gradient Descent — Delta Rule

The basic idea of gradient descent is to move weights in the direction of error steepest descent (shown by $-\nabla E$). In discrete–time approximation this leads directly to the *delta rule*:

$$\Delta W_{(t)} = W_{(t+1)} - W_{(t)} = -\eta \nabla E|_{W_{(t)}} \tag{7.6}$$

learning rate
batch learning

where η is a parameter governing the speed of learning, called the *learning rate/constant* and controlling the step size (distance between $W_{(t+1)}$ and $W_{(t)}$ points, see also Figure 7.16). This type of learning is called a *batch method* (it uses all training vectors at once every time the gradient is calculated). At step $t = 0$ the weights are initialized with the value $W_{(0)}$ usually randomly selected to contain small values (randomly: to break any possible symmetry and small values: to avoid a premature saturation of neuronal output). Note that weights are *not* moved towards error minima but in the direction of error local steepest descent, see Figure 7.16. This can lead to a slow and zigzag progress toward the minimum, see the right hand side of Figure 7.17. As E is additive with respect to training set then:

$$E = \sum_{p=1}^{P} E_p \quad \Rightarrow \quad \nabla E = \sum_{p=1}^{P} \nabla E_p$$

Alternatively the same method (as above) may be used but with one vector at a time:

$$\Delta W_{(t)} = W_{(t+1)} - W_{(t)} = -\eta \nabla E_p|_{W_{(t)}}$$

on–line learning
epoch

where p denotes a vector from the training set, e.g. the training vectors may be numbered in some way and then considered in the order $p = t$ (first vector taken at first step, ..., and so on). This type of learning is called *on–line method* (it uses just one training vector at a time, it is also known as *incremental* or *sequential*). The training set may be used repeatedly, one pass over all vectors in the training set is called an *epoch*, vectors may be shuffled around between epochs.

Remarks:

- ➥ Batch and multiple epochs training may not be used when patterns form a time–series because the time sequencing information is lost.

A geometrical interpretation of gradient descent method can be given by representing the error E as a surface, see Figure 7.16 on the facing page (E is interpreted as a function mapping the weights space $\mathbb{R}^{N_W}$ to $\mathbb{R}$).

[7.5.2] See [Has95] pp. 199–206.

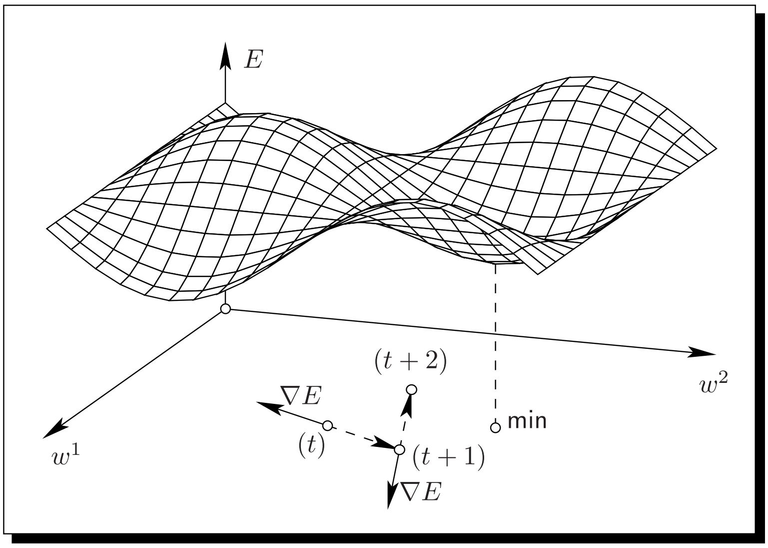

Figure 7.16: *The error surface for a bidimensional weights space. The steps t, $t+1$ and $t+2$ are also shown. The weights are moved in the direction of (local) error steepest descent (and* ***not*** *towards the minimum).*

Note that gradient descent method is not guaranteed to find the global or *absolute* error minima. However in practice the local minima found is usually good enough.

Remarks:

- The choice of η may be critical to the learning process. A large value may lead to over–shooting of the minima, (especially if it's narrow and steep in terms of error surface), and/or oscillations between 2 areas (points) in weights space (see also Figure 7.17 on the next page). A small value will lead to long learning time (large number of steps required).
- It is also possible to use a mixed learning, i.e. to divide the training set into subsets and use each subset in batch learning. This technique is especially useful if the training algorithm is intrinsically of batch type.

7.5.3 Gradient Descent — Momentum

The basic idea of momentum technique is to keep previous weight change directions as a trend, gradually fading away (i.e. it is assumed that previous changes were good and should continue for more than one learning step). Momentum usually leads to an increase in the speed of learning.

Momentum is achieved by adding a supplementary term to (7.6):

$$\Delta W_{(t)} = -\eta \nabla E|_{W_{(t)}} + \mu \Delta W_{(t-1)} \tag{7.7}$$

The μ parameter is called the *momentum*. See Figure 7.17 on the following page. ❖ μ

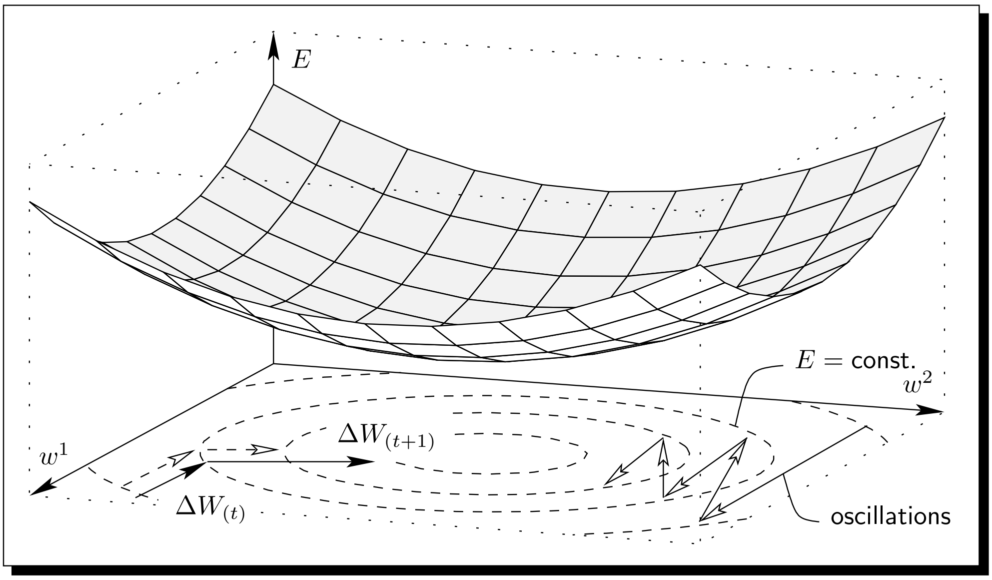

Figure 7.17: *Learning with momentum generally increases speed (dotted arrows ΔW show learning without momentum) but if there are oscillations (showed on right side of figure) then the benefits of momentum are lost (momentum terms tend to cancel between two consecutive steps).*

(7.7) may be rewritten into a continuous form as follows:

$$W_{(t+1)} - W_{(t)} = -\eta \nabla E|_{W_{(t)}} + \mu \left[W_{(t)} - W_{(t-1)}\right]$$

$$\begin{aligned} W_{(t+1)} - W_{(t)} &= -\eta \nabla E|_{W_{(t)}} + \mu \left[W_{(t)} - W_{(t-1)}\right] \\ &\quad - \mu \left[W_{(t+1)} - W_{(t)}\right] + \mu \left[W_{(t+1)} - W_{(t)}\right] \end{aligned}$$

$$\Delta W_{(t)} = -\eta \nabla E|_{W_{(t)}} - \mu \left[\Delta W_{(t)} - \Delta W_{(t-1)}\right] + \mu \Delta W_{(t)}$$

$$(1-\mu)\Delta W_{(t)} = -\eta \nabla E|_{W_{(t)}} - \mu \Delta^2 W_{(t-1)} \tag{7.8}$$

❖ ΔW, $\Delta^2 W$

where $\Delta W_{(t)} = W_{(t+1)} - W_{(t)}$ and $\Delta^2 W_{(t)} = \Delta W_{(t+1)} - \Delta W_{(t)}$.

❖ τ

In converting (7.8) to its limiting form the terms have to be of the same order. Let τ be equal to the unit of time (introduced for dimensional correctness), then (7.8) becomes in the limit:

$$(1-\mu) dW \, \frac{dt}{\tau} = -\eta \nabla E \, \frac{dt^2}{\tau^2} - \mu \, d^2 W$$

❖ m, ν

which gives finally the differential equation:

$$m \, \frac{d^2 W}{dt^2} + \nu \frac{dW}{dt} = -\nabla E \,, \quad \text{where} \quad m = \frac{\mu \tau^2}{\eta} \,, \; \nu = \frac{(1-\mu)\tau}{\eta} \tag{7.9}$$

Remarks:

➥ (7.9) represents the equation of movement of a particle into the weights space W, having "mass" m, subject to friction (viscosity) proportional with the speed, defined by ν and into a conservative force field E.

W represents the position, $\frac{dW}{dt}$ represents the speed, $\frac{d^2W}{dt^2}$ is the acceleration, finally E and $-\nabla E$ are the potential, respectively the force of the conservative field.

The momentum should be chosen $\mu \in (0,1)$ otherwise weight changes may accumulate and grow to infinity — see the constant gradient case below.

Discussion. Consider $\Delta W_{(t)} = -\eta \nabla E|_{W_{(t)}} + \mu \Delta W_{(t-1)}$ and $\Delta W_{(t-1)} = -\eta \nabla E|_{W_{(t-1)}} + \mu \Delta W_{(t-2)}$ then:

$$\Delta W_{(t)} = -\eta \nabla E|_{W_{(t)}} - \mu\eta \nabla E|_{W_{(t-1)}} + \mu^2 \Delta W_{(t-2)}$$

Previous weight adaptation tend to accumulate at a geometrical progression with rate μ.

To understand the effect given by the momentum two cases may be analyzed:

- The gradient is constant $\nabla E =$ const. Then, by applying iteratively (7.7):

$$\Delta W = -\eta \nabla E \, (1 + \mu + \mu^2 + \ldots) \simeq -\frac{\eta}{1-\mu} \, \nabla E$$

  (because $\mu \in (0,1)$ and the geometric series converges to $1/(1-\mu)$), i.e. the learning rate effectively increases from η to $\frac{\eta}{(1-\mu)}$.

- In a region with high curvature where the gradient changes direction almost completely (opposite direction), generating weights oscillations, the effects of momentum will tend to cancel from one oscillation to the next, see Figure 7.17 on the preceding page.

In discrete–time approximation the momentum algorithm starts (at $t = 1$) with $\Delta W_{(0)} = \widetilde{0}$.

7.6 Error Criteria/Functions

7.6.1 Sum–of–Squares Error

The sum–of–squares error (SSE) function is the most widely used, it is defined as: SSE

$$E = \frac{1}{2} \sum_{p=1}^{P} \|\mathbf{y}(\mathbf{x}^p) - \mathbf{t}^p\|^2 \quad \text{or} \quad E = \frac{1}{2P} \sum_{p=1}^{P} \|\mathbf{y}(\mathbf{x}^p) - \mathbf{t}^p\|^2$$

(the second form has the advantage of not growing to large values for $P \to \infty$).

Remarks:

➥ Consider the classification problem and let P_k be the number of $\mathbf{x}^p \in \mathcal{C}_k$ and $P_{\mathsf{true}}(\mathcal{C}_k)$ the *true* distribution for $\mathcal{C}_k$. If the *observed* $P(\mathcal{C}_k) = P_k/P$ is far off $P_{\mathsf{true}}(\mathcal{C}_k)$ then SSE will lead to wrong (biased) results (this happens if some classes are over/under represented in the training set). To correct the situation it is possible to built an *weighted* SSE: weighted SSE

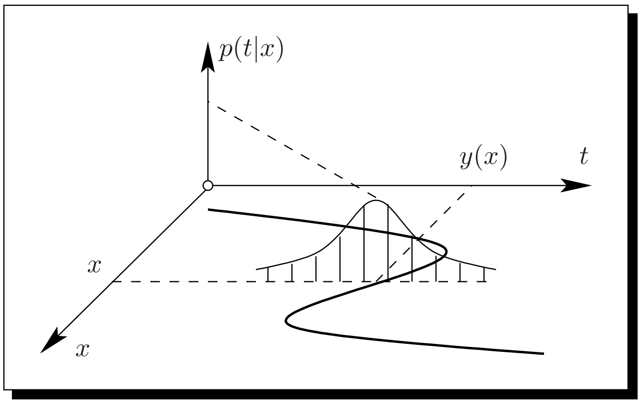

Figure 7.18: *The network output when using SSE. Unidimensional input and output space. Note that $\langle t|x\rangle$ doesn't necessary coincide with the maximum of $p(t|x)$.*

$$E = \frac{1}{2}\sum_{p=1}^{P} \kappa_p \|\mathbf{y}(\mathbf{x}^p) - \mathbf{t}^p\|^2 \quad \text{where} \quad \kappa_p = \frac{P_{\text{true}}(\mathcal{C}_k)}{P(\mathcal{C}_k)} \text{ for } \mathbf{x}^p \in \mathcal{C}_k$$

The $P_{\text{true}}(\mathcal{C}_k)$ are usually not known, then they may be assumed to be the probabilities related to some *test* patterns (patterns which are run through the net but not used to train it) or they may be estimated by other means.

Proposition 7.6.1. [5] *Assuming that:*

- *the training set is sufficiently large: $P \to \infty$;*
- *the number N_W of weights is sufficiently large;*
- *the absolute minimum of SSE was found;*

❖ $\langle \mathbf{t}|\mathbf{x}\rangle$

then the ANN output with SSE function represents the average of target conditioned on input:

$$\mathbf{y}(\mathbf{x})|_{W_{opt}} = \langle \mathbf{t}|\mathbf{x}\rangle := \int\limits_{\mathbb{R}^K} \mathbf{t}\, p(\mathbf{t}|\mathbf{x})\, d\mathbf{t} \tag{7.10}$$

❖ W_{opt}

where W_{opt} denotes the optimal weights, after the learning process has been finished (note that network output is weights dependent). See Figure 7.18.

Proof. Using the scaled version of SSE:

$$E = \lim_{P\to\infty} \frac{1}{2P}\sum_{p=1}^{P} \|\mathbf{y}(\mathbf{x}^p) - \mathbf{t}^p\|^2 = \frac{1}{2}\iint\limits_{\mathbb{R}^K,\mathbb{R}^N} \|\mathbf{y}(\mathbf{x}) - \mathbf{t}\|^2 p(\mathbf{t},\mathbf{x})\, d\mathbf{t}\, d\mathbf{x} \tag{7.11}$$

$$= \frac{1}{2}\iint\limits_{\mathbb{R}^K,\mathbb{R}^N} \|\mathbf{y}(\mathbf{x}) - \mathbf{t}\|^2 p(\mathbf{t}|\mathbf{x})\, p(\mathbf{x})\, d\mathbf{t}\, d\mathbf{x}$$

❖ $\langle \|\mathbf{t}\|^2|\mathbf{x}\rangle$

The following conditional averages are defined:

$$\langle \mathbf{t}|\mathbf{x}\rangle \equiv \int\limits_{\mathbb{R}^K} \mathbf{t}\, p(\mathbf{t}|\mathbf{x})\, d\mathbf{t} \quad \text{and} \quad \langle \|\mathbf{t}\|^2|\mathbf{x}\rangle \equiv \int\limits_{\mathbb{R}^K} \|\mathbf{t}\|^2 p(\mathbf{t}|\mathbf{x})\, d\mathbf{t} \tag{7.12}$$

[5]See [Bis95] pp. 201–206, the proof and formulas given here are *vectorial*.

Then, by using the above definitions, it's possible to write:

$$\begin{aligned}\|\mathbf{y}(\mathbf{x})-\mathbf{t}\|^2 &= [\mathbf{y}(\mathbf{x})-\mathbf{t}]^{\mathrm{T}}[\mathbf{y}(\mathbf{x})-\mathbf{t}] = \|\mathbf{y}(\mathbf{x})-\langle\mathbf{t}|\mathbf{x}\rangle+\langle\mathbf{t}|\mathbf{x}\rangle-\mathbf{t}\|^2 \\ &= \|\mathbf{y}(\mathbf{x})-\langle\mathbf{t}|\mathbf{x}\rangle\|^2+2[\mathbf{y}(\mathbf{x})-\langle\mathbf{t}|\mathbf{x}\rangle]^{\mathrm{T}}(\langle\mathbf{t}|\mathbf{x}\rangle-\mathbf{t})+\|\langle\mathbf{t}|\mathbf{x}\rangle-\mathbf{t}\|^2 \qquad (7.13)\end{aligned}$$

and by substituting into the expression for E obtain:

$$E=\frac{1}{2}\iint\limits_{\mathbb{R}^K,\mathbb{R}^N}\left\{\|\mathbf{y}(\mathbf{x})-\langle\mathbf{t}|\mathbf{x}\rangle\|^2+2[\mathbf{y}(\mathbf{x})-\langle\mathbf{t}|\mathbf{x}\rangle]^{\mathrm{T}}(\langle\mathbf{t}|\mathbf{x}\rangle-\mathbf{t})+\|\langle\mathbf{t}|\mathbf{x}\rangle-\mathbf{t}\|^2\right\}p(\mathbf{t}|\mathbf{x})\,p(\mathbf{x})\,d\mathbf{t}\,d\mathbf{x}$$

Then:

- the first term is independent of $p(\mathbf{t}|\mathbf{x})$ and $\int\limits_{\mathbb{R}^K} p(\mathbf{t}|\mathbf{x})\,d\mathbf{t}=1$ (normalization);
- the middle term cancels because: $\langle\mathbf{t}|\mathbf{x}\rangle-\mathbf{t}\to\langle\mathbf{t}|\mathbf{x}\rangle-\langle\mathbf{t}|\mathbf{x}\rangle=\widehat{\mathbf{0}}$;
- the third term transforms to:

$$\|\langle\mathbf{t}|\mathbf{x}\rangle-\mathbf{t}\|^2=\langle\mathbf{t}^{\mathrm{T}}|\mathbf{x}\rangle\langle\mathbf{t}|\mathbf{x}\rangle-\langle\mathbf{t}^{\mathrm{T}}|\mathbf{x}\rangle\mathbf{t}-\mathbf{t}^{\mathrm{T}}\langle\mathbf{t}|\mathbf{x}\rangle+\mathbf{t}^{\mathrm{T}}\mathbf{t}\to\langle\|\mathbf{t}\|^2|\mathbf{x}\rangle-\|\langle\mathbf{t}|\mathbf{x}\rangle\|^2 \qquad (7.14)$$

$$\Rightarrow\quad E=\frac{1}{2}\int\limits_{\mathbb{R}^N}\|\mathbf{y}(\mathbf{x})-\langle\mathbf{t}|\mathbf{x}\rangle\|^2p(\mathbf{x})\,d\mathbf{x}+\frac{1}{2}\int\limits_{\mathbb{R}^N}(\langle\|\mathbf{t}\|^2|\mathbf{x}\rangle-\|\langle\mathbf{t}|\mathbf{x}\rangle\|^2)\,p(\mathbf{x})\,d\mathbf{x} \qquad (7.15)$$

In the expression (7.15) of the error function, the second term is weights–independent. The error becomes minimum (with respect to W) when the integrand in the first term is zero, i.e. $\mathbf{y}(\mathbf{x})=\langle\mathbf{t}|\mathbf{x}\rangle$. □

Remarks:

➥ The above result does not make any assumption over the network architecture or even the existence of a neuronal model at all. It holds for any model which attempts to minimize SSE.

➥ Another proof of Proposition 7.6.1 may be performed through the error gradient with respect to network outputs. A necessary condition for a minima is:

$$\nabla_{\mathbf{y}}E=\iint\limits_{\mathbb{R}^K,\mathbb{R}^N}[\mathbf{y}(\mathbf{x})-\mathbf{t}]\,p(\mathbf{t}|\mathbf{x})\,p(\mathbf{x})\,d\mathbf{t}\,d\mathbf{x}$$

$$=\int\limits_{\mathbb{R}^N}[\mathbf{y}(\mathbf{x})-\langle\mathbf{t}|\mathbf{x}\rangle]\,p(\mathbf{x})\,d\mathbf{x}=\widehat{\mathbf{0}}$$

and the integral may be zero only if the integrand is zero.

Assuming that target components are statistically independent $\left(p(\mathbf{t}|\mathbf{x})=\prod\limits_{k=1}^{K}p(t_k|\mathbf{x})\right)$ and normalized then the absolute SSE minima represents the sum of target components variance:

$$E_{\mathsf{min}}=\sum_{k=1}^{K}\mathcal{V}\{t_k|\mathbf{x}\} \qquad (7.16)$$

Proof. From (7.13), (7.14) and (7.15) (at minima first term cancels):

$$\begin{aligned}E_{\mathsf{min}}&=\frac{1}{2}\iint\limits_{\mathbb{R}^K,\mathbb{R}^N}\|\mathbf{t}-\langle\mathbf{t}|\mathbf{x}\rangle\|^2p(\mathbf{t}|\mathbf{x})p(\mathbf{x})\,\mathbf{t}\,d\mathbf{x}\\ &=\sum_{k=1}^{K}\int\limits_{\mathbb{R}^N}\left[\int\limits_{\mathbb{R}}(t_k-\langle t_k|\mathbf{x}\rangle)^2p(t_k|\mathbf{x})\,dt_k\right]\left[\prod_{\ell\neq k}\int\limits_{\mathbb{R}}p(t_\ell|\mathbf{x})\,dt_\ell\right]d\mathbf{x}\end{aligned}$$

$$= \sum_{k=1}^{K} \iint_{\mathbb{R},\mathbb{R}^N} (t_k - \langle t_k|\mathbf{x}\rangle)^2 p(t_k|\mathbf{x})\, dt_k\, d\mathbf{x} \quad \square$$

Remarks:

- If a good feature extraction has been performed in preliminary processing then it will decrease redundancy and enhance the statistical independence of target components.
- The SSE based network is unable to discern between different distributions $p(\mathbf{t}|\mathbf{x})$ with same means (7.10) and sum of variances (7.16), see also Section 7.8.

7.6.2 Minkowski Error

The Minkowski error[6] is defined as:

$$E = \frac{1}{r} \sum_{p=1}^{P} \widehat{\mathbf{1}}^{\mathrm{T}} |\mathbf{y}(\mathbf{x}^p) - \mathbf{t}^p|^{\odot r}$$

Remarks:

- Obviously, for $r = 2$ it reduces to SSE; for $r = 1$ the corresponding error is called *city–block metric*. More generally the distance $\left(\widehat{\mathbf{1}}^{\mathrm{T}} |\mathbf{x}_1 - \mathbf{x}_2|^{\odot r}\right)^{1/r}$ is called the ℓ_r norm.

ℓ_r norm

- The use of values $1 \leqslant r < 2$ allows one to reduce the weighting of a large deviation (a highly noisy input $\mathbf{x}^p$ will lead to a $\mathbf{y}(\mathbf{x}^p)$ far from desired target $\mathbf{t}^p$ and thus a large additive term in E).

7.7 Jacobian and Hessian

Jacobian
❖ J

The Jacobian is defined as:

$$J = (\nabla_{\mathbf{x}} \mathbf{y}^{\mathrm{T}})^{\mathrm{T}} = \left\{ \frac{\partial y_k}{\partial x_i} \right\}_{\substack{k \in \overline{1,K} \\ i \in \overline{1,N}}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \cdots & \frac{\partial y_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_K}{\partial x_1} & \cdots & \frac{\partial y_K}{\partial x_N} \end{pmatrix}$$

and it represents a measure of the local network output sensitivity to a change in the inputs, i.e. for small perturbation in input vector the perturbation in the output vector is:

$$\Delta \mathbf{y} \simeq J \Delta \mathbf{x} \ .$$

Or more precisely:

$$\Delta \mathbf{y} = J \Delta \mathbf{x} + o\left(\|\Delta \mathbf{x}\|\right) \quad \text{as} \quad \|\Delta \mathbf{x}\| \to 0$$

❖ o

where the "o" Landau's little o notation. It may be also used to estimate network output

[6] See [Has95] pp. 82–84.

uncertainties $\Delta\mathbf{y}$, *given input uncertainties* $\Delta\mathbf{x}$, but it has to be calculated for each input in turn (note that there are also network intrinsic uncertainties calculated through the Hessian, see below).

Note that the Jacobian has NK elements and thus its computation requires at least $\mathcal{O}(NK)$ computational time.

Hessian
❖ H

The Hessian is defined as:

$$H = \nabla \otimes \nabla^{\mathrm{T}} E \quad \text{where} \quad h_{kj}^{i\ell} = \left\{ \frac{\partial^2 E}{\partial w_k^i \partial w_\ell^j} \right\}$$

note that the usage of Kronecker tensorial product fits both vector and matrix forms of ∇ thus being more general; however for one neuron only W is a row matrix and so is ∇_W.

The Hessian is additive with respect to E_p:

$$E = \sum_{p=1}^{P} E_p \quad \Rightarrow \quad H = \sum_{p=1}^{P} H_p = \sum_{p=1}^{P} \nabla \otimes \nabla^{\mathrm{T}} E_p$$

The Hessian is a symmetric in the sense that $h_{kj}^{i\ell} = h_{jk}^{\ell i}$ *if the second partial derivatives of* E *are continuous or if* E *is a two times continuously differentiable function of weights* (e.g. this is not the case for threshold activation). Unless otherwise specified it will be assumed that the Hessian is symmetric (because usually it is).

The Hessian is used to:

- analyze $E(W)$ in several algorithms;
- assess weights importance for pruning purposes[7];
- find network output uncertainties.

Note that the Hessian has N_W^2 elements and thus its computation requires at least $\mathcal{O}(N_W^2)$ computational time (for symmetric Hessian there are $\frac{N_W(N_W+1)}{2}$ different elements).

7.8 The Statistical Approach

The statistical theory plays a central role in the understanding of feedforward networks theory. The link to statistical theory is done generally through error criteria considered as the negative logarithm of likelihood of training set:

$$\mathcal{L} = \prod_{p=1}^{P} p(\mathbf{x}^p, \mathbf{t}^p) = \prod_{p=1}^{P} p(\mathbf{t}^p|\mathbf{x}^p)\, p(\mathbf{x}^p)$$

$$\widetilde{E} = -\ln \mathcal{L} = -\sum_{p=1}^{P} \ln p(\mathbf{t}^p|\mathbf{x}^p) - \sum_{p=1}^{P} \ln p(\mathbf{x}^p) \;\rightarrow\; E = -\sum_{p=1}^{P} \ln p(\mathbf{t}^p|\mathbf{x}^p) \tag{7.17}$$

[7]As in the "optimal brain damage" and "optimal brain surgeon" weight optimization techniques.

Minimizing $\widetilde{E}$ with respect to the weights is equivalent to minimization of E; the terms $p(\mathbf{x}^p)$ being dropped because they don't depend on network W parameters. Error minimization becomes equivalent to likelihood maximisation; the above expression gives a supplementary argument for error additivity with respect to training sets.

Sum–of–squares error

If the targets $\{\mathbf{t}^p\}$ have a deterministic component plus Gaussian distributed noise then maximizing the likelihood, and the SSE criteria, lead to the same choice of parameters[8] W.

Proof. Consider a Gaussian distribution of targets as composed from a deterministic function $\mathbf{g}(\mathbf{x})$ and some Gaussian noise ε:

$$\mathbf{t} = \mathbf{g}(\mathbf{x}) + \varepsilon \quad \text{where} \quad p(\varepsilon) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\|\varepsilon\|^2}{2\sigma^2}\right)$$

Then $\varepsilon = \mathbf{t} - \mathbf{g}(\mathbf{x})$, $\mathbf{g}(\mathbf{x}) = \mathbf{y}(\mathbf{x})|_{W_{\text{opt}}}$ because it's the model represented by the neural network (W_{opt} being optimal network parameters which minimize SSE absolutely), and $p(\varepsilon) = p(\mathbf{t}|\mathbf{x})$ (as $\mathbf{g}(\mathbf{x})$ is purely deterministic):

$$p(\mathbf{t}|\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\|\mathbf{y}(\mathbf{x})|_{W_{\text{opt}}} - \mathbf{t}\|^2}{2\sigma^2}\right)$$

By using the above expression in (7.17), the error function becomes:

$$E_{\text{min}} = \frac{1}{2\sigma^2} \sum_{p=1}^{P} \|\mathbf{y}(\mathbf{x}^p)|_{W_{\text{opt}}} - \mathbf{t}^p\|^2 + P \ln \sigma + \frac{P}{2} \ln 2\pi$$

and minimizing this expression is equivalent to minimizing SSE, as the only W–dependent term is: $\sum\limits_{p=1}^{P} \|\mathbf{y}(\mathbf{x}^p) - \mathbf{t}^p\|^2$. □

[8] See [Bis95] pp. 195–196, the proof and formulas here are *vectorial* and do not require the statistical independence of target components.

CHAPTER 8

Layered Feedforward Networks

This chapter describes various results, algorithms and techniques for computations on *layered* feedforward networks. Here "layered" refers to the regularities in architecture, e.g. there is only one type of activation function per layer. Many algorithms described here may also apply to general feedforward networks (especially once the error gradient has been calculated). The same notation as developed in "The Backpropagation Networks" and "General Feedforward Networks" chapter will be used here (with few exceptions, see below).

8.1 Architecture and Description

It will be assumed that all neurons from same layer have same activation function f_ℓ, generally continuous, bounded and differentiable. Full connections are assumed between adjacent layers, there are no connections skipping layers, each layer may have any number of neurons. The computations described here may generally be easily extended to the case when there are full known connections across non–adjacent layers. All neurons are assumed to have bias, simulated by a first neuron with fixed output on each layer, including input, excluding output (i.e. $z_0 =$ const., usually 1). The biases are stored in the first column, numbered 0, of (each) weight matrix. See Figure 8.1 on the next page.

❖ f_ℓ

Each layer calculates its activation $\mathbf{a}_{(\ell)} = W_{(\ell)}\mathbf{z}_{(\ell-1)}$ then applies its activation function to calculate output $\mathbf{z}_{(\ell)(1:N_\ell)} = f_\ell(\mathbf{a}_{(\ell)})$ and $z_{(\ell)0} = 1$ for bias.

8.2 Learning Algorithms

8.2.1 Backpropagation

Theorem 8.2.1. *If the error gradient with respect to neuronal outputs* $\nabla_{\mathbf{z}_{(L)}} E_p$ *is known,*

backpropagation

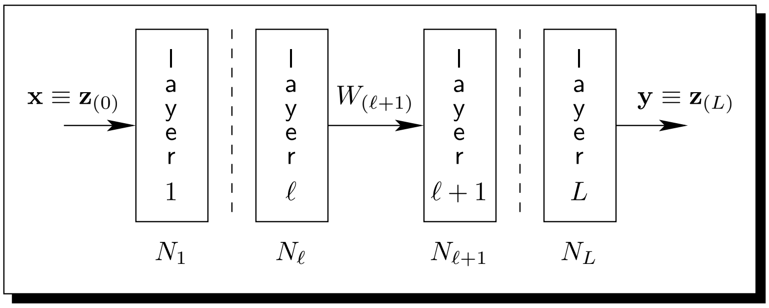

Figure 8.1: *The layered feedforward network architecture.*

and depends only on (actual) network outputs $\{\mathbf{z}_{(L)}(\mathbf{x}^p)\}$ *and targets* $\{\mathbf{t}^p\}$*:* $\nabla_{\mathbf{z}_{(L)}} E_p =$ *known., then the error gradient (with respect to weights) may be calculated recursively, from* $L-1$ *to* 1*, according to formulas:*

$$\nabla_{\mathbf{z}_{(\ell)(1:N_\ell)}} E_p = W_{(\ell+1)}^{(1:N_\ell)\mathrm{T}} \left[\nabla_{\mathbf{z}_{(\ell+1)(1:N_{\ell+1})}} E_p \odot f'_{\ell+1}(\mathbf{a}_{(\ell+1)}) \right] , \quad 1 \leqslant \ell \leqslant L-1 \tag{8.1a}$$

$$\nabla_{W_{(\ell)}} E_p = \left[\nabla_{\mathbf{z}_{(\ell)(1:N_\ell)}} E_p \odot f'_\ell(\mathbf{a}_{(\ell)}) \right] \cdot \mathbf{z}_{(\ell-1)}^{\mathrm{T}} \quad , \quad 1 \leqslant \ell \leqslant L \tag{8.1b}$$

where $\mathbf{z}_{(0)} \equiv \mathbf{x}$*.*

Proof. The error $E_p(W)$ is dependent on $w_{(\ell)h}^g$ through the output of neuron $((\ell), h)$, i.e. $z_{(\ell)h}$ (the data flow is $\mathbf{x} \to w_{(\ell)h}^g \to \mathbf{z}_{(\ell)h} \to E_p$, *note that* $h \neq 0$):

$$\frac{\partial E_p}{\partial w_{(\ell)h}^g} = \frac{\partial E_p}{\partial z_{(\ell)h}} \frac{\partial z_{(\ell)h}}{\partial w_{(\ell)h}^g} \quad \Rightarrow \quad \nabla_{W_{(\ell)}} E_p = \nabla_{\mathbf{z}_{(\ell)(1:N_\ell)}} E_p \overset{\mathrm{C}}{\odot} \left\{ \frac{\partial z_{(\ell)h}}{\partial w_{(\ell)h}^g} \right\}$$

and each derivative is computed separately.

a. Term $\frac{\partial z_{(\ell)h}}{\partial w_{(\ell)h}^g}$ is:

$$\frac{\partial z_{(\ell)h}}{\partial w_{(\ell)h}^g} = \frac{\partial}{\partial w_{(\ell)h}^g} f_\ell((W_{(\ell)}\mathbf{z}_{(\ell-1)})_h) = f'_\ell(a_{(\ell)h}) \frac{\partial (W_{(\ell)}\mathbf{z}_{(\ell-1)})_h}{\partial w_{(\ell)h}^g} = f'_\ell(a_{(\ell)h})\, z_{(\ell-1)g}$$

because weights *from the same layer* are mutually independent $((W_{(\ell)}\mathbf{z}_{(\ell-1)})_h = W_{(\ell)(h)}^{(:)}\mathbf{z}_{(\ell-1)})$; and then:

$$\left\{ \frac{\partial z_{(\ell)h}}{\partial w_{(\ell)h}^g} \right\} = f'_\ell(\mathbf{a}_{(\ell)})\, \mathbf{z}_{(\ell-1)}^{\mathrm{T}}$$

b. Term $\frac{\partial E_p}{\partial z_{(\ell)h}}$: neuron $((\ell)h)$ affects E_p through all following layers that are intermediate between layer (ℓ) and output (the influence being exercised through the interposed neurons, the data flow is $\mathbf{x} \to \mathbf{z}_{(\ell)} \to \mathbf{z}_{(\ell+1)} \to E_p$).

The first affected is the next, $\ell+1$ layer, through term $\frac{\partial z_{(\ell+1)g}}{\partial z_{(\ell)h}}$ (and then the dependency is carried on through successive layers) and, by similar means as for previous term (*note that neuron* $((\ell+1), 0)$ *does not receive any connections as it represents bias*, also the derivative with respect to $z_{(\ell)0}$ doesn't make sense as $z_{(\ell)0} =$ const.):

$$\frac{\partial E_p}{\partial z_{(\ell)h}} = \sum_{g=1}^{N_{\ell+1}} \frac{\partial E_p}{\partial z_{(\ell+1)g}} \frac{\partial z_{(\ell+1)g}}{\partial z_{(\ell)h}} = \sum_{g=1}^{N_{\ell+1}} \frac{\partial E_p}{\partial z_{(\ell+1)g}} \frac{\partial}{\partial z_{(\ell)h}} f_{\ell+1}((W_{(\ell+1)}\mathbf{z}_{(\ell)})_g)$$

$$= \sum_{g=1}^{N_{\ell+1}} \frac{\partial E_p}{\partial z_{(\ell+1)g}} f'_{\ell+1}(a_{(\ell+1)g}) \frac{\partial (W_{(\ell+1)} \mathbf{z}_{(\ell)})_g}{\partial z_{(\ell)h}} = \sum_{g=1}^{N_{\ell+1}} \frac{\partial E_p}{\partial z_{(\ell+1)g}} f'_{\ell+1}(a_{(\ell+1)g})\, w^h_{(\ell+1)g}$$

$$= \left(W_{(\ell+1)}^{(1:N_\ell)\mathrm{T}} [\nabla_{\mathbf{z}_{(\ell+1)(1:N_{\ell+1})}} E_p \odot f'_{\ell+1}(\mathbf{a}_{(\ell+1)})] \right)_h$$

which represents exactly the element h of vector $\nabla_{\mathbf{z}_{(\ell)(1:N_\ell)}} E_p$ as built from (8.1a). The above formula applies iteratively from layer $L-1$ to 1; for layer L, $\nabla_{\mathbf{z}_{(L)}} E_p$ is assumed known.

Finally, the sought derivative is: $\nabla_{W_{(\ell)}} E_p = \nabla_{\mathbf{z}_{(\ell)(1:N_\ell)}} E_p \overset{\mathrm{C}}{\odot} [f'_\ell(\mathbf{a}_{(\ell)})\, \mathbf{z}^{\mathrm{T}}_{\ell-1}]$. ☐

Remarks:

- ➥ The particular case of sum–of–squares error and logistic activation together with other improvements has been discussed already in the chapter "The Backpropagation Network".
- ➥ Note that $z_{(L)0}$ does *not* exist.

8.2.2 Conjugate Gradients

The main idea of conjugate gradients is to approximate the true error locally by a quadratic form:

$$E = E_0 + G^{\mathrm{T}} \circ W + \frac{1}{2} W^{\mathrm{T}} \circ H \circ W \tag{8.2}$$

where $G = \{g^i_k\}$ has the same size as W and $H = \{h^{i\ell}_{kj}\}$ has the same size and structure as $W \otimes W^{\mathrm{T}}$, in particular is symmetric (i.e. $h^{i\ell}_{kj} = h^{kj}_{\ell i}$); then $G^{\mathrm{T}} \circ W = {g'}^k_i w^i_k$ and $W^{\mathrm{T}} \circ H \circ W = {w'}^k_i h^{i\ell}_{kj} w^j_\ell$ (where the summation convention have been used with "pairs" of upper and lower indices). ❖ G, H

Remarks:

- ➥ (8.2) is a quadratic Taylor polynomial approximation to E (E is a gradient arranged as a matrix and H a Hessian arranged as a 4-dimensional tensor).
- ➥ The tensorial product of type $G^{\mathrm{T}} \circ W$ has a very simple matrix representation:

$$G^{\mathrm{T}} \circ W = {g'}^k_i w^i_k = \sum_{k,i} g^i_k w^i_k = \widehat{\mathbf{1}}^{\mathrm{T}} (G \odot W) \widehat{\mathbf{1}}$$

  another representation being $\mathrm{Vec}(G^{\mathrm{T}})^{\mathrm{T}}\, \mathrm{Vec}(W^{\mathrm{T}})$ (the reason to use $\mathrm{Vec}((\cdot)^{\mathrm{T}})$ is because matrices are usually represented as $(\cdot)_{ki}$ then the rearranged vector follows the lexicographic convention over (k, i)).

The minimum of E is achieved at the point $W_{\text{opt.}}$ where the error gradient is zero (H is symmetric): ❖ $W_{\text{opt.}}$

$$\nabla E = G + H \circ W \quad , \quad \nabla E|_{W_{\text{opt.}}} = \widetilde{0} \quad \Rightarrow \quad G + H \circ W_{\text{opt.}} = \widetilde{0} \tag{8.3}$$

Definition 8.2.1. *A set of nonzero directions* $\{D_{(t)}\}$ *(of type* d^i_k, $D_{(t)} \neq \widetilde{0}$*) is called* conjugate directions ❖ $D_{(t)}$

[8.2.2] See [GL96] pp. 520–540, [Rip96] pp. 342–345 and [Bis95] pp. 274–290, note that here error gradient and conjugate directions are matrices and the Hessian is a tensor.

H–conjugate if:

$$D_{(t)}^{\mathrm{T}} \circ H \circ D_{(s)} = 0 \quad \textit{for} \quad t \neq s \; .$$

Note that $D_{(t)}$ will be considered as being from the same space as W ($D_{(t)}$ has same size and layout as W) and H from the same space as $D_{(t)} \otimes D_{(t)}^{\mathrm{T}}$.

Proposition 8.2.1. *If H is symmetric positive definite then any set $\{D_{(t)}\}$ of H–conjugate directions is linearly independent.*

Proof. Suppose that one of the directions can be written as a linear combination of the other ones:

$$D_{(s)} = \sum_{i \neq s} \alpha_i D_{(i)} \quad , \quad \alpha_i = \text{const.}$$

According to Definition 8.2.1, for $\forall t \neq s$:

$$0 = D_{(s)}^{\mathrm{T}} \circ H \circ D_{(t)} = \left(\sum_{i \neq s} \alpha_i D_{(i)}^{\mathrm{T}} \right) \circ H \circ D_{(t)} = \alpha_t D_{(t)}^{\mathrm{T}} \circ H \circ D_{(t)}$$

Hence $\alpha_t = 0$, $\forall t \neq s$ and thus $D_{(s)} = \widetilde{0}$ which contradicts Definition 8.2.1. □

Proposition 8.2.1 ensures the existence of N_W linearly independent directions. This means that any tensor from $\mathbb{R}^{N_W}$ may be represented as a linear combination of $\{D_{(t)}\}$, in particular, considering $W_{(1)}$ as initial weights then:

$$W_{\text{opt.}} - W_{(1)} = \sum_{t=1}^{N_W} \alpha_t D_{(t)} \tag{8.4}$$

i.e. the search for error minima at $W_{\text{opt.}}$ may be performed as a series of steps along the directions $D_{(t)}$:

$$W_{(t+1)} = W_{(t)} + \alpha_t D_{(t)} \tag{8.5}$$

where $W_{\text{opt.}} = W_{(N_W+1)}$. The α_t steps are:

$$\alpha_t = - \frac{D_{(t)}^{\mathrm{T}} \circ (G + H \circ W_{(1)})}{D_{(t)}^{\mathrm{T}} \circ H \circ D_{(t)}} \tag{8.6}$$

Proof. Multiply (8.4) to the left by $D_{(s)}^{\mathrm{T}} \circ H \circ (\cdot)$, use the conjugate direction property and (8.3):

$$\begin{aligned} D_{(s)}^{\mathrm{T}} \circ H \circ (W_{\text{opt.}} - W_{(1)}) &= -D_{(s)}^{\mathrm{T}} \circ G - D_{(s)}^{\mathrm{T}} \circ H \circ W_{(1)} \\ &= \sum_{t=1}^{N_W} \alpha_t D_{(s)}^{\mathrm{T}} \circ H \circ D_{(t)} = \alpha_s D_{(s)}^{\mathrm{T}} \circ H \circ D_{(s)} \quad \square \end{aligned}$$

Another form for α_t coefficients is:

$$\alpha_t = - \frac{D_{(t)}^{\mathrm{T}} \circ \nabla E|_{W_{(t)}}}{D_{(t)}^{\mathrm{T}} \circ H \circ D_{(t)}} \tag{8.7}$$

Proof. $W_{(t)} = W_{(1)} + \sum\limits_{s=1}^{t-1} \alpha_s D_{(s)}$; multiplying by $D_{(t)}^{\mathrm{T}} \circ H \circ (\cdot)$ to the left and using the conjugate direction property it gives $D_{(t)}^{\mathrm{T}} \circ H \circ W_{(t)} = D_{(t)}^{\mathrm{T}} \circ H \circ W_{(1)}$. From (8.3): $D_{(t)}^{\mathrm{T}} \circ \nabla E|_{W_{(t)}} = D_{(t)}^{\mathrm{T}} \circ (G + H \circ W_{(t)}) = D_{(t)}^{\mathrm{T}} \circ (G + H \circ W_{(1)})$ and use this result in (8.6). □

Proposition 8.2.2. *If weights are updated according to procedure* (8.5) *then the error gradient at step* t *is orthogonal on all previous conjugate directions:*

$$D^{\mathrm{T}}_{(s)} \circ \nabla E|_{W_{(t)}} = 0 \quad , \quad \forall s,t \text{ such that } s < t \leqslant N_W \tag{8.8}$$

(equivalently $\mathrm{Vec}(D^{\mathrm{T}}_{(s)})^{\mathrm{T}} \, \mathrm{Vec}(\nabla^{\mathrm{T}} E|_{W_{(t)}}) = 0$*, i.e. the corresponding* Vec *vectors are orthogonal).*

Proof. From (8.3) and (8.5):

$$\nabla E|_{W_{(t+1)}} - \nabla E|_{W_{(t)}} = H \circ \left(W_{(t+1)} - W_{(t)}\right) = \alpha_t H \circ D_{(t)} \tag{8.9}$$

and, by multiplying on the left with $D^{\mathrm{T}}_{(t)} \circ (\cdot)$ and replacing α_t from (8.7), it follows that ($\{D_{(s)}\}$ are conjugate directions):

$$D^{\mathrm{T}}_{(t)} \circ \nabla E|_{W_{(t+1)}} = 0 \; . \tag{8.10}$$

On the other hand, by multiplying (8.9) with $D^{\mathrm{T}}_{(s)} \circ (\cdot)$ to the left (and using the conjugate directions property):

$$D^{\mathrm{T}}_{(s)} \circ \left(\nabla E|_{W_{(t+1)}} - \nabla E|_{W_{(t)}}\right) = \alpha_t D^{\mathrm{T}}_{(s)} \circ H \circ D_{(t)} = 0 \quad , \quad \forall s < t \leqslant N_W$$

Keeping s "fixed", this equation is written for all instances from $s+1$ to some $t+1$ and then a summation is done over all the equations obtained, which gives:

$$D^{\mathrm{T}}_{(s)} \circ \left(\nabla E|_{W_{(t+1)}} - \nabla E|_{W_{(s+1)}}\right) = 0 \quad , \quad \forall s < t \leqslant N_W$$

and, by using (8.10), finally:

$$D^{\mathrm{T}}_{(s)} \circ \nabla E|_{W_{(t+1)}} = 0 \quad , \quad \forall s < t \leqslant N_W$$

which combined with (8.10) (i.e. for $s = t$) proves the desired result. □

A set of conjugate directions $\{D_{(t)}\}$ may be built as follows:

1. The first direction is chosen as:

$$D_{(1)} = -\nabla E|_{W_{(1)}}$$

2. The following directions are built incrementally as:

$$D_{(t+1)} = -\nabla E|_{W_{(t+1)}} + \beta_t D_{(t)} \tag{8.11}$$

   where β_t are coefficients to be found such that the newly built $D_{(t+1)}$ is conjugate with the previous $D_{(t)}$, i.e. $D^{\mathrm{T}}_{(t+1)} \circ H \circ D_{(t)} = 0$ (note that $D_{(t+1)}$ is the direction from $W_{(t+1)}$, known at $t+1$, to next $W_{(t+2)}$). By multiplying (8.11) with $(\cdot) \circ H \circ D_{(t)}$ on the right: $\left(-\nabla E|_{W_{(t+1)}} + \beta_t D_{(t)}\right)^{\mathrm{T}} \circ H \circ D_{(t)} = 0$: ❖ β_t

$$\beta_t = \frac{\nabla^{\mathrm{T}} E|_{W_{(t+1)}} \circ H \circ D_{(t)}}{D^{\mathrm{T}}_{(t)} \circ H \circ D_{(t)}} \tag{8.12}$$

   (note that β_t is used to generate $D_{(t+1)}$ from $W_{(t+1)}$ to $W_{(t+2)}$, i.e. $W_{(t+1)}$ point is known at this stage).

Proposition 8.2.3. *By using the above method for building the set of directions, the error gradient at step t is orthogonal on all previous ones:*

$$\nabla^{\mathrm{T}} E|_{W_{(s)}} \circ \nabla E|_{W_{(t)}} = 0 \quad , \quad \forall t, s \text{ such that } s < t \leqslant N_W \tag{8.13}$$

(equivalent $\mathrm{Vec}(\nabla^{\mathrm{T}} E|_{W_{(s)}})^{\mathrm{T}} \, \mathrm{Vec}(\nabla^{\mathrm{T}} E|_{W_{(t)}}) = 0$*, i.e. the* Vec *vectors are orthogonal).*

Proof. Obviously by the way of building ((8.11) and (8.12)), each direction represents a linear combination of all previously gradients, of the form:

$$D_{(s)} = -\nabla E|_{W_{(s)}} + \sum_{n=1}^{s-1} \gamma_n \nabla E|_{W_{(n)}} \tag{8.14}$$

❖ γ_n

where γ_n are the coefficients of the linear combination (their exact value is not relevant to the proof).

Transposing (8.14) and multiplying on the right with $(\cdot) \circ \nabla E|_{W_{(t)}}$ and using the result established in (8.8):

$$\nabla^{\mathrm{T}} E|_{W_{(s)}} \circ \nabla E|_{W_{(t)}} = \sum_{n=1}^{s-1} \gamma_n \nabla^{\mathrm{T}} E|_{W_{(n)}} \circ \nabla E|_{W_{(t)}} \quad , \quad \forall s, t \text{ such that } s < t \leqslant N_W \; .$$

For $s = 1$ the error gradient equals direction $-D_{(1)}$ and, by using (8.8), the result (8.13) holds. For $s = 2$:

$$\nabla^{\mathrm{T}} E|_{W_{(2)}} \circ \nabla E|_{W_{(t)}} = \sum_{n=1}^{s-1} \gamma_n \nabla^{\mathrm{T}} E|_{W_{(n)}} \circ \nabla E|_{W_{(t)}} = \gamma_1 \nabla^{\mathrm{T}} E|_{W_{(1)}} \circ \nabla E|_{W_{(t)}} = 0$$

and so on, as long as $s < t$, and thus the (8.13) is true. ☐

Using the conjugate gradient directions, at each step, $\mathrm{Vec}(\Delta W_{(t)}^{\mathrm{T}})$ is perpendicular on all previous ones and the minima is reached necessarly in N_W steps. See Figure 8.2 on the facing page. *Note that the N_W steps limit applies only for a quadratic error, for nonquadratic functions the conjugate gradients method may require more steps.*

Proposition 8.2.4. *The set of directions $\{D_{(t)}\}$ built according to* (8.11) *and* (8.12) *are mutually H–conjugate.*

Proof. The result will be shown by induction. Induction basis: by construction $D_{(2)}^{\mathrm{T}} \circ H \circ D_{(1)} = 0$, i.e. these directions are conjugate. Induction step: assume that for some $t > 1$:

$$D_{(t)}^{\mathrm{T}} \circ H \circ D_{(s)} = 0 \quad , \quad \forall s \text{ such that } s < t \leqslant N_W$$

is true and it has to be shown that it holds for $t + 1$ as well (assuming $t + 1 \leqslant N_W$, of course).

Now consider $D_{(t+1)}$. Using the induction hypothesis and (8.11):

$$\begin{aligned} D_{(t+1)}^{\mathrm{T}} \circ H \circ D_{(s)} &= -\nabla^{\mathrm{T}} E|_{W_{(t+1)}} \circ H \circ D_{(s)} + \beta_i D_{(t)}^{\mathrm{T}} \circ H \circ D_{(s)} \\ &= -\nabla^{\mathrm{T}} E|_{W_{(t+1)}} \circ H \circ D_{(s)} \end{aligned} \tag{8.15}$$

$\forall s$ such that $s < t \leqslant N_W$.

From (8.9), $\nabla E|_{W_{(s+1)}} - \nabla E|_{W_{(s)}} = \alpha_s H \circ D_{(s)}$. By multiplying this equation with $\nabla^{\mathrm{T}} E|_{W_{(t+1)}} \circ (\cdot)$ to the left, and using (8.13), i.e. $\nabla^{\mathrm{T}} E|_{W_{(t+1)}} \circ \nabla E|_{W_{(s)}} = 0$ for $\forall s < t + 1 < N_W$, then:

$$\begin{aligned} &\nabla^{\mathrm{T}} E|_{W_{(t+1)}} \circ \left(\nabla E|_{W_{(s+1)}} - \nabla E|_{W_{(s)}} \right) \\ &\qquad = \nabla^{\mathrm{T}} E|_{W_{(t+1)}} \circ \nabla E|_{W_{(s+1)}} - \nabla^{\mathrm{T}} E|_{W_{(t+1)}} \circ \nabla E|_{W_{(s)}} = \alpha_s \nabla E|_{W_{(t+1)}} \circ H \circ D_{(s)} = 0 \end{aligned}$$

and by inserting this result into (8.15) then:

$$D_{(t+1)}^{\mathrm{T}} \circ H \circ D_{(s)} = 0 \quad , \quad \forall t, s \text{ such that } s < t \leqslant N_W$$

and this result is extensible from $s < t \leqslant N_W$ to $s < t + 1 \leqslant N_W$ because of the way $D_{(t+1)}$ is build, i.e. $D_{(t+1)} \circ H \circ D_{(t)} = 0$ by design. ☐

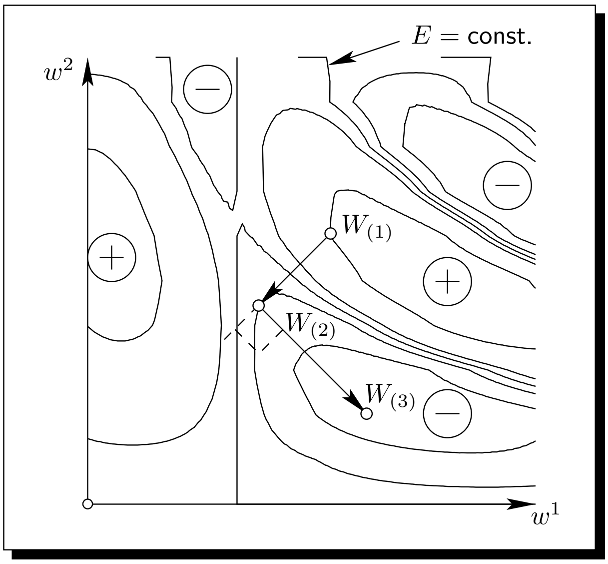

Figure 8.2: *Conjugate gradient learning in a $N_W = 2$ dimensional spaces (figure shows the contour plot of some error function, $\oplus$ are maxima, $\ominus$ are minima). $W_{(2)} - W_{(1)} \propto D_{(1)}$ and $W_{(3)} - W_{(2)} \propto D_{(2)}$, $W_{opt.} = W_{(3)}$. (note that the N_W steps limit applies only for a quadratic error).*

The Algorithm

The above discussion gives the general method for (fast) finding the minima of E given a quadratic error. However in general, error being a non–quadratic function, the Hessian is variable and then it has to be calculated at each $W_{(t)}$ point which results in a very computationally intensive process. Fortunately it is possible to express the α_t and β_t coefficients without explicit calculation of Hessian. Also while in practice the error function is not quadratic the conjugate gradient algorithm still gives a good way of finding the error minimum point.

There are several ways to express the β_t coefficients (note that β_t is used to build $D_{(t+1)}$ from $W_{(t+1)}$ to $W_{(t+2)}$, i.e. $W_{(t+1)}$ and $D_{(t)}$ are known at this stage):

- *The Hestenes–Stiefel formula*. By substituting (8.9) into (8.12): Hestenes–Stiefel

$$\beta_t = \frac{\nabla^{\mathrm{T}} E|_{W_{(t+1)}} \circ \left(\nabla E|_{W_{(t+1)}} - \nabla E|_{W_{(t)}} \right)}{D^{\mathrm{T}}_{(t)} \circ \left(\nabla E|_{W_{(t+1)}} - \nabla E|_{W_{(t)}} \right)} \tag{8.16}$$

- *The Polak–Ribiere formula*. From (8.11) and (8.8) and by making a multiplication to the right: Polak–Ribiere

$$D_{(t)} = -\nabla E|_{W_{(t)}} + \beta_{(t-1)} D_{(t-1)} \quad ; \quad D^{\mathrm{T}}_{(t-1)} \circ \nabla E|_{W_{(t)}} = 0 \quad \Rightarrow$$

$$\begin{aligned} D^{\mathrm{T}}_{(t)} \circ \nabla E|_{W_{(t)}} &= -\nabla^{\mathrm{T}} E|_{W_{(t)}} \circ \nabla E|_{W_{(t)}} + \beta_t D^{\mathrm{T}}_{(t-1)} \circ \nabla E|_{W_{(t)}} \\ &= -\nabla^{\mathrm{T}} E|_{W_{(t)}} \circ \nabla E|_{W_{(t)}} = -\|\nabla E|_{W_{(t)}}\|^2 \end{aligned}$$

(because $\nabla^{\mathrm{T}} E|_{W_{(t)}} \circ \nabla E|_{W_{(t)}} = \widehat{\mathbf{1}}^{\mathrm{T}} \left(\nabla E|_{W_{(t)}}\right)^{\odot 2} \widehat{\mathbf{1}} = \|\nabla E|_{W_{(t)}}\|^2$, where $\|\cdot\|$ is the Frobenius/Euclidean matrix norm). By using this result, together with (8.8) again, into (8.16), finally:

$$\beta_t = \frac{\nabla^{\mathrm{T}} E|_{W_{(t+1)}} \circ \left(\nabla E|_{W_{(t+1)}} - \nabla E|_{W_{(t)}}\right)}{\|\nabla E|_{W_{(t)}}\|^2} \tag{8.17}$$

Fletcher–Reeves

- *The Fletcher–Reeves formula*. From (8.17) and using (8.13):

$$\beta_t = \frac{\|\nabla E|_{W_{(t+1)}}\|^2}{\|\nabla E|_{W_{(t)}}\|^2} \tag{8.18}$$

Remarks:

➥ In theory, i.e. for a quadratic error function, (8.16), (8.17) and (8.18) are equivalent. In practice, because the error function is seldom quadratic, they may gives different results. According to [Bis95] (pg. 281) the Polak–Ribiere formula gives the best result in general.

The α_t coefficients can't be expressed without using the Hessian but fortunately they may be replaced by a procedure of finding E minima along $D_{(t)}$ directions.

Proof. Consider a quadratic error as a function of α_t:

$$E(W_{(t)} + \alpha_t D_{(t)}) = E_0 + G^{\mathrm{T}} \circ (W_{(t)} + \alpha_t D_{(t)}) + \frac{1}{2}\left(W_{(t)} + \alpha_t D_{(t)}\right)^{\mathrm{T}} \circ H \circ (W_{(t)} + \alpha_t D_{(t)})$$

The minimum of the error along the line through $W_{(t)}$ in the direction $D_{(t)}$ occurs at the stationary point:

$$\frac{\partial E}{\partial \alpha_t} = 0 \quad \Rightarrow \quad G^{\mathrm{T}} \circ D_{(t)} + \left(W_{(t)} + \alpha_t D_{(t)}\right)^{\mathrm{T}} \circ H \circ D_{(t)} = 0$$

As $\nabla E = G + H \circ W$, then:

$$\alpha_t = -\frac{\nabla^{\mathrm{T}} E|_{W_{(t)}} \circ D_{(t)}}{D_{(t)}^{\mathrm{T}} \circ H \circ D_{(t)}}$$

this formula being identical to (8.7) ($\nabla^{\mathrm{T}} E|_{W_{(t)}} \circ D_{(t)} = D_{(t)}^{\mathrm{T}} \circ \nabla E|_{W_{(t)}}$). This shows that the procedure of finding α_t coefficients may be replaced with any procedure for finding the error minima along a line in direction $D_{(t)}$. ☐

There is also a possibility to calculate directly the product $D_{(t)} \circ H$ (which appears in the expression of both α_t and β_t) through an efficient finite difference approach, see Section 8.4.5.

The general algorithm is:

1. Select an starting point (in the weight space) given by $W_{(1)}$.
2. Calculate $\nabla E|_{W_{(1)}}$ and make: $D_{(1)} = -\nabla E|_{W_{(1)}}$ (e.g. the error gradient may be calculated through the backpropagation algorithm).
3. For $t = 1, \ldots,$ (max. value):
   (a) Find the minimum of $E(W_{(t)} + \alpha_t D_{(t)})$ with respect to α_t and move to the next point $W_{(t+1)} = W_t + \alpha_{t(\text{min.})} D_{(t)}$.

   (b) Evaluate the stopping condition. This may be a requirement that the error drop under a specified value, a maximum number of steps, etc. Exit if stopping condition is met.

(c) Calculate $\nabla E|_{W_{(t+1)}}$ and then β_t, using one of the (8.16), (8.17) or (8.18), alternatively use the procedure developed in Section 8.4.5. Note that for two equally sized matrices A and B the "$\circ$" multiplication is: $A^{\mathrm{T}} \circ B = \mathrm{sum}(A \odot B)$. Finally calculate the new $D_{(t+1)}$ direction from (8.11).

8.3 Jacobian Computation

Theorem 8.3.1. *The Jacobian of a layered feedforward network may be calculated through a backpropagation procedure, according to formulas:*

$$(\nabla_{\mathbf{a}_{(L)}} \mathbf{z}^{\mathrm{T}}_{(L)})^{\mathrm{T}} = \mathrm{Diag}(f'_L(\mathbf{a}_{(L)}))$$

$$(\nabla_{\mathbf{a}_{(\ell)}} \mathbf{z}^{\mathrm{T}}_{(L)})^{\mathrm{T}} = (\nabla_{\mathbf{a}_{(\ell+1)}} \mathbf{z}^{\mathrm{T}}_{(L)})^{\mathrm{T}} W^{(1:N_\ell)}_{(\ell+1)} \,\mathrm{Diag}(f'_\ell(\mathbf{a}_{(\ell)})) = (\nabla_{\mathbf{a}_{(\ell+1)}} \mathbf{z}^{\mathrm{T}}_{(L)})^{\mathrm{T}} W^{(1:N_\ell)}_{(\ell+1)} \overset{\mathrm{R}}{\odot} f'_\ell(\mathbf{a}^{\mathrm{T}}_{(\ell)})$$

$$J = (\nabla_{\mathbf{z}_{(0)}} \mathbf{z}^{\mathrm{T}}_{(L)})^{\mathrm{T}} = (\nabla_{\mathbf{a}_{(1)}} \mathbf{z}^{\mathrm{T}}_{(L)})^{\mathrm{T}} W^{(1:N_1)}_{(1)}$$

Proof. For the output layer ($z'^k_{(L)} = f'_L(a'^k_{(L)})$):

$$\nabla_{\mathbf{a}_{(L)}} \mathbf{z}^{\mathrm{T}}_{(L)} = \left\{ \frac{\partial z'^k_{(L)}}{\partial a_{(L)\ell}} \right\} = \left\{ f'_L(a'^k_{(L)})\, \delta^k_\ell \right\} = \mathrm{Diag}(f'_L(\mathbf{a}_{(L)}))$$

i.e. all partial derivatives are 0 except $\frac{\partial z'^k_{(L)}}{\partial a_{(L)k}}$ and the matrix $\nabla_{\mathbf{a}_{(L)}} \mathbf{z}^{\mathrm{T}}_{(L)}$ is diagonal because there is no lateral interaction (on any layer, including output, $f'_{(\cdot)}$ is the derivative of activation $f_{(\cdot)}$).

For an intermediate layer the dependence of $\mathbf{z}_{(L)}$ on $\mathbf{a}_{(\ell)}$ is carried out through the next $\ell+1$ layer ($k, h, g \neq 0$, $\mathbf{a}_{(\ell)}$ represent total input to real neurons, not "bias"):

$$\nabla_{\mathbf{a}_{(\ell)}} \mathbf{z}^{\mathrm{T}}_{(L)} = \left\{ \frac{\partial z'^k_{(L)}}{\partial a_{(\ell)h}} \right\} = \left\{ \frac{\partial z'^k_{(L)}}{\partial a_{(\ell+1)g}} \right\} \left\{ \frac{\partial a'^g_{(\ell+1)}}{\partial a_{(\ell)h}} \right\} = \nabla_{\mathbf{a}_{(\ell)}} \mathbf{a}^{\mathrm{T}}_{(\ell+1)} \cdot \nabla_{\mathbf{a}_{(\ell+1)}} \mathbf{z}^{\mathrm{T}}_{(L)} \tag{8.19}$$

(where $\{a'^g_{(\ell+1)}\} = \mathbf{a}^{\mathrm{T}}_{(\ell+1)}$, as established for "$'$" notation). As $\mathbf{a}_{(\ell+1)} = W_{(\ell+1)} \mathbf{z}_{(\ell)}$, consider the bias (it will disappear shortly) as $z_{(\ell)0} \equiv f_\ell(a_{(\ell)0}) \equiv 1 =$ const. ($a_{(\ell)0}$ does not really exists, it is introduced artificially here) then $\mathbf{a}_{(\ell+1)} = W_{(\ell+1)} f_\ell(\mathbf{a}_{(\ell)})$ and $\{a'^g_{(\ell+1)}\} = \{w'^g_{(\ell+1)f}\}\{f_\ell(a'^f_{(\ell)})\}$:

$$\nabla_{\mathbf{a}_{(\ell)}} \mathbf{a}^{\mathrm{T}}_{(\ell+1)} = \left\{ \frac{\partial a'^g_{(\ell+1)}}{\partial a_{(\ell)h}} \right\} = \left\{ \frac{\partial \{w'^g_{(\ell+1)f}\}\{f_\ell(a'^f_{(\ell)})\}}{\partial a_{(\ell)h}} \right\}$$

$$\to \{w'^g_{(\ell+1)f}\} \left\{ f'_\ell(a'^f_{(\ell)})\, \delta^f_h \right\} = \mathrm{Diag}(f'_\ell(\mathbf{a}_{(\ell)}))\, W^{(1:N_\ell)\mathrm{T}}_{(\ell+1)}$$

(the bias disappears because it is constant and thus has a zero derivative). Substituting back into (8.19) gives:

$$\nabla_{\mathbf{a}_{(\ell)}} \mathbf{z}^{\mathrm{T}}_{(L)} = \mathrm{Diag}(f'_\ell(\mathbf{a}_{(\ell)}))\, W^{(1:N_\ell)\mathrm{T}}_{(\ell+1)} \nabla_{\mathbf{a}_{(\ell+1)}} \mathbf{z}^{\mathrm{T}}_{(L)}$$

For the input layer consider $\mathbf{a}_{(0)} = \mathbf{z}_{(0)(1:N_0)}$ and a layer "0" with N_0 neurons (aside from bias) each receiving one component of $\mathbf{z}_{(0)(1:N_0)}$ and all having the identity activation function $f_0(\mathbf{z}_{(0)(1:N_0)}) = \mathbf{z}_{(0)(1:N_0)}$, then $f'_0(\mathbf{z}_{(0)(1:N_0)}) = \widehat{\mathbf{1}}$ and apply the same formula as for intermediate layers ($\nabla_{\mathbf{a}_0} \to \nabla_{\mathbf{z}_{(0)}}$). □

[8.3] See [Bis95] pp. 148–150, the treatment here is *tensorial* for *layered* feedforward ANN.

Remarks:

- The algorithm described in Theorem 8.3.1 requires only one propagation forward (to compute $f'_\ell(\mathbf{a}_{(\ell)})$) and one backward (to calculate the Jacobian itself); compare this with the "naive" approach of calculating each element as:

$$j^i_k = \frac{\partial z_{(L)k}}{\partial {z'}^i_{(0)}} \simeq \frac{z_{(L)k}(\mathbf{z}_{(0)} + \varepsilon \mathbf{e}_i) - z_{(L)k}(\mathbf{z}_{(0)} - \varepsilon \mathbf{e}_i)}{2\varepsilon} \quad , \quad \varepsilon \gtrsim 0$$

which requires $2N_0N_L$ forward propagations; however the "naive" algorithm is still good to check the correctness of a particular system implementation.

- Note that for many activations f it is possible to calculate the derivative through activation itself (e.g. for the logistic activation $f' = cf(1-f)$).
- The forward propagation required by this algorithm may be performed in parallel with the forward propagation of the input itself when using the trained network.
- Note the multiplication properties: $A\,\mathrm{Diag}(\mathbf{b}) = A \overset{\mathrm{R}}{\odot} \mathbf{b}^{\mathrm{T}}$ and $\mathrm{Diag}(\mathbf{b})A = \mathbf{b} \overset{\mathrm{C}}{\odot} A$.

8.4 Hessian Computation

8.4.1 Diagonal Approximation

This method assumes that *all mixed second order derivatives are zero*. That is $\frac{\partial^2 E}{\partial w^i_k \partial w^j_\ell} = 0$ for $k \neq \ell$ or $i \neq j$; the Hessian reduces to $\nabla^{\odot 2} E$ and may be easily calculated through a backpropagation procedure, from $\ell = L-1$ to $\ell = 1$, according to formulas:

$$\nabla_{\mathbf{a}_{(\ell)}} \mathbf{a}^{\mathrm{T}}_{(\ell+1)} = \mathrm{Diag}(f'_\ell(\mathbf{a}_{(\ell)})) W^{(1:N_\ell)\mathrm{T}}_{(\ell+1)} = f'_\ell(\mathbf{a}_{(\ell)}) \overset{\mathrm{C}}{\odot} W^{(1:N_\ell)\mathrm{T}}_{(\ell+1)} \tag{8.20a}$$

$$\nabla^{\odot 2}_{\mathbf{a}_{(\ell)}} \mathbf{a}^{\mathrm{T}}_{(\ell+1)} = \mathrm{Diag}(f''_\ell(\mathbf{a}_{(\ell)})) W^{(1:N_\ell)\mathrm{T}}_{(\ell+1)} = f''_\ell(\mathbf{a}_{(\ell)}) \overset{\mathrm{C}}{\odot} W^{(1:N_\ell)\mathrm{T}}_{(\ell+1)} \tag{8.20b}$$

$$\nabla_{\mathbf{a}_{(\ell)}} E = \nabla_{\mathbf{a}_{(\ell)}} \mathbf{a}^{\mathrm{T}}_{(\ell+1)} \nabla_{\mathbf{a}_{(\ell+1)}} E \tag{8.20c}$$

$$\nabla^{\odot 2}_{\mathbf{a}_{(\ell)}} E = \left[\nabla_{\mathbf{a}_{(\ell)}} \mathbf{a}^{\mathrm{T}}_{(\ell+1)}\right]^{\odot 2} \nabla^{\odot 2}_{\mathbf{a}_{(\ell+1)}} E + \nabla^{\odot 2}_{\mathbf{a}_{(\ell)}} \mathbf{a}^{\mathrm{T}}_{(\ell+1)} \nabla_{\mathbf{a}_{(\ell+1)}} E \tag{8.20d}$$

$$\Rightarrow \quad \nabla^{\odot 2}_{W_{(\ell)}} E = \nabla^{\odot 2}_{\mathbf{a}_{(\ell)}} E \cdot \mathbf{z}^{\odot 2\,\mathrm{T}}_{(\ell-1)} \tag{8.20e}$$

and from the input layer consider $\ell = 0$ and:

$$\nabla_{\mathbf{a}_{(0)}} \mathbf{a}^{\mathrm{T}}_{(1)} \equiv W^{(1:N_0)\mathrm{T}}_{(1)} \quad \text{and} \quad \nabla^{\odot 2}_{\mathbf{a}_{(0)}} \mathbf{a}^{\mathrm{T}}_{(1)} \equiv \widetilde{0} \tag{8.20f}$$

Proof. As $\mathbf{a}_{(\ell)} = W_{(\ell)} \mathbf{z}_{(\ell-1)}$ and $\mathbf{z}_{(\ell-1)}$ is independent of $W_{(\ell)}$ then:

$$\frac{\partial}{\partial w^g_{(\ell)h}} = \frac{\partial a_{(\ell)h}}{\partial w^g_{(\ell)h}} \frac{\partial}{\partial a_{(\ell)h}} = z_{(\ell-1)g} \frac{\partial}{\partial a_{(\ell)h}} \quad \Rightarrow$$

$$\frac{\partial^2}{\partial (w^g_{(\ell)h})^2} = (z_{(\ell-1)g})^2 \frac{\partial^2}{\partial (a_{(\ell)h})^2} \quad \Rightarrow \quad \nabla^{\odot 2}_{W_{(\ell)}} E = \nabla^{\odot 2}_{\mathbf{a}_{(\ell)}} E \cdot \mathbf{z}^{\odot 2\,\mathrm{T}}_{(\ell-1)}$$

[8.4] See [Bis95] pp. 150–160, the treatment here is *tensorial* for *layered* feedforward ANN and allow for *different activation functions on each layer*.

This gives formula (8.20e).

The error depends on all intermediate layers between ℓ and output, the first intermediate layer being layer $\ell+1$. Then (neglecting the off–diagonal terms on second derivatives):

$$\frac{\partial E}{\partial a_{(\ell)h}} = \left\{\frac{\partial E}{\partial a_{(\ell+1)g}}\right\}\left\{\frac{\partial a'^{g}_{(\ell+1)}}{\partial a_{(\ell)h}}\right\}$$

$$\frac{\partial^2 E}{\partial (a_{(\ell)h})^2} = \left\{\frac{\partial^2 E}{\partial a_{(\ell+1)g}\partial a_{(\ell+1)f}}\right\}\left\{\frac{\partial a'^{f}_{(\ell+1)}}{\partial a_{(\ell)h}}\right\}\left\{\frac{\partial a'^{g}_{(\ell+1)}}{\partial a_{(\ell)h}}\right\} + \left\{\frac{\partial E}{\partial a_{(\ell+1)g}}\right\}\left\{\frac{\partial^2 a'^{g}_{(\ell+1)}}{\partial (a_{(\ell)h})^2}\right\}$$

$$\simeq \left\{\frac{\partial^2 E}{\partial (a_{(\ell+1)g})^2}\right\}\left\{\left[\frac{\partial a'^{g}_{(\ell+1)}}{\partial a_{(\ell)h}}\right]^{\odot 2}\right\} + \left\{\frac{\partial E}{\partial a_{(\ell+1)g}}\right\}\left\{\frac{\partial^2 a'^{g}_{(\ell+1)}}{\partial (a_{(\ell)h})^2}\right\}$$

This gives formulas (8.20c) and (8.20d).

The activation derivatives are calculated from: $a'^{g}_{(\ell+1)} = \left\{w'^{g}_{(\ell+1)f}\right\}\left\{f_\ell(a'^{f}_{(\ell)})\right\}$ and consider the same bias trick as performed in proof of Theorem 8.3.1; then (index $f \neq 0$):

$$\frac{\partial a'^{g}_{(\ell+1)}}{\partial a_{(\ell)h}} = \left\{w'^{g}_{(\ell+1)f}\right\}\left\{f'_\ell(a'^{f}_{(\ell)})\,\delta^f_h\right\} \quad \Rightarrow \quad \frac{\partial^2 a'^{g}_{(\ell+1)}}{\partial (a_{(\ell)h})^2} = \left\{w'^{g}_{(\ell+1)f}\right\}\left\{f''_\ell(a'^{f}_{(\ell)})\,\delta^f_h\right\}$$

(f''_ℓ being the second derivative of f_ℓ). This gives formulas (8.20a) and (8.20b). For $\ell+1=1$ consider $\mathbf{a}_{(0)} \equiv \mathbf{z}_{(0)(1:N_0)}$ and a layer "0" with N_0 neurons (aside from bias), each receiving one component of $\mathbf{z}_{(0)}$ and all having the identity activation $f_0(\mathbf{z}_{(0)(1:N_0)}) = \mathbf{z}_{(0)(1:N_0)}$; then $f'_0(\mathbf{z}_{(0)(1:N_0)}) = \widehat{\mathbf{1}}$ and $f''_0(\mathbf{z}_{(0)(1:N_0)}) = \widehat{\mathbf{0}}$, this gives (8.20f). □

8.4.2 Outer–Product Approximation

This method is designed for sum–of–squares error and *trained* networks. The Hessian may be calculated through a backpropagation procedure:

$$H = \sum_{p=1}^{P} H_p \quad \text{where} \quad H_p = \left\{\frac{\partial^2 E_p}{\partial w^i_k \partial w^j_\ell}\right\} = \sum_{n=1}^{N_L} \nabla z_{(L)n}(\mathbf{z}^p_{(0)}) \otimes \nabla z_{(L)n}(\mathbf{z}^p_{(0)}) \tag{8.21a}$$

$$\nabla_{\mathbf{z}_{(L)}} z_{(L)n} = \mathbf{e}_n \tag{8.21b}$$

$$\nabla_{\mathbf{z}_{(\ell)(1:N_\ell)}} z_{(L)n} = W^{(1:N_{\ell+1})\mathrm{T}}_{(\ell+1)}\left[\nabla_{\mathbf{z}_{(\ell+1)(1:N_{\ell+1})}} z_{(L)n} \odot f'_{\ell+1}(\mathbf{a}_{(\ell+1)})\right] \tag{8.21c}$$

$$\nabla_{W_{(\ell)}} z_{(L)n} = \left[\nabla_{\mathbf{z}_{(\ell)(1:N_\ell)}} z_{(L)n} \odot f'_\ell(\mathbf{a}_{(\ell)})\right] \cdot \mathbf{z}^{\mathrm{T}}_{(\ell-1)} \tag{8.21d}$$

Proof. Considering the SSE: $E_p = \frac{1}{2}\|\mathbf{z}_{(L)}(\mathbf{z}^p_{(0)}) - \mathbf{t}^p\|^2$ and some two weights w^i_k and w^j_ℓ *from any layer*; then: ❖ w^i_k, w^j_ℓ

$$\frac{\partial^2 E_p}{\partial w^i_k \partial w^j_\ell} = \left\{\frac{\partial z'^{n}_{(L)}(\mathbf{z}^p_{(0)})}{\partial w^i_k}\right\}\left\{\frac{\partial z_{(L)n}(\mathbf{z}^p_{(0)})}{\partial w^j_\ell}\right\} + \{z'^{n}_{(L)}(\mathbf{z}^p_{(0)}) - t'^{n}_{p}\}\left\{\frac{\partial^2 z_{(L)n}(\mathbf{z}^p_{(0)})}{\partial w^i_k \partial w^j_\ell}\right\} \tag{8.22}$$

Assuming the *trained* network then $z_{(L)n}(\mathbf{z}^p_{(0)}) - t^p_n \approx 0$ and thus:

$$H_p = \left\{\frac{\partial^2 E_p}{\partial w^i_k \partial w^j_\ell}\right\} \simeq \sum_{n=1}^{N_L} \nabla z_{(L)n}(\mathbf{z}^p_{(0)}) \otimes \nabla z_{(L)n}(\mathbf{z}^p_{(0)})$$

8.4.2 This method is known as "outer–product approximation" but the outer–product is the vector variant of the more general Kroneker product, used here instead.

The $\nabla z_{(L)n}(\mathbf{z}^p_{(0)})$ are calculated through a backpropagation procedure (see also the proof of Theorem 8.2.1). $z_{(L)n}$ is dependent on $w^g_{(\ell)h}$ through the output of neuron $((\ell), h)$, i.e. $z_{(\ell)h}$ (the data flow is $\mathbf{z}_{(0)} \to w^g_{(\ell)h} \to \mathbf{z}_{(\ell)h} \to z_{(L)n}$, *note that* $n, h \neq 0$):

$$\frac{\partial z_{(L)n}}{\partial w^g_{(\ell)h}} = \frac{\partial z_{(L)n}}{\partial z_{(\ell)h}} \frac{\partial z_{(\ell)h}}{\partial w^g_{(\ell)h}} \quad \Rightarrow \quad \nabla_{W_{(\ell)}} z_{(L)n} = \nabla_{\mathbf{z}_{(\ell)(1:N_\ell)}} z_{(L)n} \overset{\mathrm{C}}{\odot} \left\{ \frac{\partial z_{(\ell)h}}{\partial w^g_{(\ell)h}} \right\}$$

and each derivative is computed separately.

a. Term $\frac{\partial z_{(\ell)h}}{\partial w^g_{(\ell)h}}$ is:

$$\frac{\partial z_{(\ell)h}}{\partial w^g_{(\ell)h}} = \frac{\partial}{\partial w^g_{(\ell)h}} f_\ell((W_{(\ell)}\mathbf{z}_{(\ell-1)})_h) = f'_\ell(a_{(\ell)h}) \frac{\partial (W_{(\ell)}\mathbf{z}_{(\ell-1)})_h}{\partial w^g_{(\ell)h}} = f'_\ell(a_{(\ell)h})\, z_{(\ell-1)g}$$

because weights *from the same layer* are mutually independent ($(W_{(\ell)}\mathbf{z}_{(\ell-1)})_h = W^{(:)}_{(\ell)(h)}\mathbf{z}_{(\ell-1)}$); and then:

$$\left\{ \frac{\partial z_{(\ell)h}}{\partial w^g_{(\ell)h}} \right\} = f'_\ell(\mathbf{a}_{(\ell)})\, \mathbf{z}^{\mathrm{T}}_{(\ell-1)}$$

b. Term $\frac{\partial z_{(L)n}}{\partial z_{(\ell)h}}$: neuron $((\ell)h)$ affects $z_{(L)n}$ through all following layers that are intermediate between layer (ℓ) and output (the influence being exercised through the interposed neurons, the data flow is $\mathbf{z}_{(0)} \to \mathbf{z}_{(\ell)} \to \mathbf{z}_{(\ell+1)} \to z_{(L)n}$).

First affected is the next layer, layer $\ell+1$, through term $\frac{\partial z_{(\ell+1)g}}{\partial z_{(\ell)h}}$ (and then the dependency is carried on through next successive layers) and, by similar means as for previous term (*note that neuron* $((\ell+1), 0)$ *does not receive any connections as it represents bias*):

$$\begin{aligned}
\frac{\partial z_{(L)n}}{\partial z_{(\ell)h}} &= \sum_{g=1}^{N_{\ell+1}} \frac{\partial z_{(L)n}}{\partial z_{(\ell+1)g}} \frac{\partial z_{(\ell+1)g}}{\partial z_{(\ell)h}} = \sum_{g=1}^{N_{\ell+1}} \frac{\partial z_{(L)n}}{\partial z_{(\ell+1)g}} \frac{\partial}{\partial z_{(\ell)h}} f_{\ell+1}((W_{(\ell+1)}\mathbf{z}_{(\ell)})_g) \\
&= \sum_{g=1}^{N_{\ell+1}} \frac{\partial z_{(L)n}}{\partial z_{(\ell+1)g}} f'_{\ell+1}(a_{(\ell+1)g}) \frac{\partial (W_{(\ell+1)}\mathbf{z}_{(\ell)})_g}{\partial z_{(\ell)h}} = \sum_{g=1}^{N_{\ell+1}} \frac{\partial z_{(L)n}}{\partial z_{(\ell+1)g}} f'_{\ell+1}(a_{(\ell+1)g})\, w^h_{(\ell+1)g} \\
&= \left(W^{(1:N_{\ell+1})\mathrm{T}}_{(\ell+1)} \left[\nabla_{\mathbf{z}_{(\ell+1)(1:N_{\ell+1})}} z_{(L)n} \odot f'_{\ell+1}(\mathbf{a}_{(\ell+1)}) \right] \right)_h
\end{aligned}$$

which represents exactly the element h of vector $\nabla_{\mathbf{z}_{(\ell)(1:N_\ell)}} z_{(L)n}$ as built from (8.21c).

The above formula applies iteratively from layer $L-1$ to 1; for layer L, $\nabla_{\mathbf{z}_{(L)}} z_{(L)n} = \mathbf{e}_n$. □

Note that as $P \to \infty$ then $\mathbf{z}_{(L)} \to \langle \mathbf{t}|\mathbf{x} \rangle$ and second term in (8.22) cancels[1], i.e. the Hessian calculated through the outer–product approximation gradually approaches true Hessian for $P \to \infty$.

8.4.3 Finite Difference Approximation

The "brute force attack" in Hessian computation is to apply small perturbations to weights and calculate the error; then for *off–diagonal terms*:

$$\begin{aligned}
\frac{\partial^2 E}{\partial w^i_k \partial w^j_\ell} \simeq \frac{1}{4\varepsilon^2} & \left[E(W + \varepsilon E_{ki}, W + \varepsilon E_{\ell j}) - E(W - \varepsilon E_{ki}, W + \varepsilon E_{\ell j}) \right. \\
& \left. - E(W + \varepsilon E_{ki}, W - \varepsilon E_{\ell j}) + E(W - \varepsilon E_{ki}, W - \varepsilon E_{\ell j}) \right]
\end{aligned}$$

[1] See chapter "General Feedforward Networks".

(i.e. only w_k^i and w_ℓ^j are "perturbed"), where $0 \lesssim \varepsilon \ll 1$. This central difference approximation has accuracy $\mathcal{O}(\varepsilon^2)$. For diagonal terms the formula above has to be changed to $(0 \lesssim \varepsilon \ll 1)$:

$$\frac{\partial^2 E}{\partial {w_k^i}^2} \simeq \frac{1}{\varepsilon^2}\left[E(W+\varepsilon E_{ki}) - 2E(W) + E(W-\varepsilon E_{ki})\right]$$

(only w_k^i is "perturbed").

This "brute force" method requires $\mathcal{O}(4N_W)$ computing time for each Hessian element (one pass for each of four E terms), i.e. $\mathcal{O}(4N_W^3)$ computing time for the whole tensor. However it is a good way to check the correctness of other algorithm implementations (also in this respect the accuracy is of no concern).

Another approach is to use the gradient ∇E:

$$\frac{\partial^2 E}{\partial w_k^i \partial w_\ell^j} \simeq \frac{1}{2\varepsilon}\left[\left.\frac{\partial E}{\partial w_k^i}\right|_{w_\ell^j+\varepsilon} - \left.\frac{\partial E}{\partial w_k^i}\right|_{w_\ell^j-\varepsilon}\right]$$

This approach requires $2N_W$ gradient calculations who may be performed efficiently through the backpropagation method; thus the total time requirement is $\mathcal{O}(2N_W^2)$.

8.4.4 Exact Calculation

This method calculates the exact Hessian through a backpropagation procedure (note that parts of these calculations may have been already performed when used in conjunction with other algorithms, e.g. backpropagation). The Hessian will be calculated considering the following:

- The $\nabla \otimes \nabla^{\mathrm{T}} E$ is made out parts of the form $\nabla_{W_{(k)}} \otimes \nabla^{\mathrm{T}}_{W_{(\ell)}} E$ (where k and ℓ are some two layers) each to be calculated separately[2].
- $\nabla_{W_{(k)}} \otimes \nabla^{\mathrm{T}}_{W_{(\ell)}} E = \sum\limits_{p=1}^{P} \nabla_{W_{(k)}} \otimes \nabla^{\mathrm{T}}_{W_{(\ell)}} E_p$ and each $\nabla_{W_{(k)}} \otimes \nabla^{\mathrm{T}}_{W_{(\ell)}} E_p$ will be computed separately (and its contribution added to $\nabla_{W_{(k)}} \otimes \nabla^{\mathrm{T}}_{W_{(\ell)}} E$).

It is assumed that $\nabla_{\mathbf{z}_{(L)}} E_p$ and $\nabla_{\mathbf{z}_{(L)}} \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p$ are known (given the expression for the error is usually easy to find an analytical formula, in terms of $\{\mathbf{z}^p_{(L)}\}$). For the algorithm below, it is assumed that k, ℓ are fixed and all neuronal activities refer to the same input/output set p. It will be also considered that the Hessian is "symmetric", i.e. $h_{kj}^{i\ell} = h_{i\ell}^{kj}$; consequently only half of the full Hessian will be calculated and stored[3], specifically for $k \leqslant \ell$ ($k, \ell \in \overline{1, L}$, note that $\mathbf{z}_{(0)}$ is the input).

1. Do a forward propagation through the network and retain the following values for later calculation: $\mathbf{z}_{(k-1)}$, $\mathbf{z}_{(\ell-1)}$, all $f'_k(\mathbf{a}_{(k)})$ to $f'_L(\mathbf{a}_{(L)})$ and all $f''_\ell(\mathbf{a}_{(\ell)})$ to $f''_L(\mathbf{a}_{(L)})$ (note that in some cases f' and f'' may be calculated faster through f).

[2] The specific layout of $\nabla \otimes \nabla^{\mathrm{T}} E$ within the memory of the digital system used is irrelevant to the algorithm.

[3] The analogy here is with vectors and matrices, if W was a vector then $\nabla \otimes \nabla^{\mathrm{T}} E$ would be a symmetric matrix.

2. Calculate and retain $\nabla_{\mathbf{a}_{(m)}} \mathbf{a}^{\mathrm{T}}_{(m+1)}$ for all $m = \overline{k, L-1}$, as follows:

$$\nabla_{\mathbf{a}_{(m)}} \mathbf{a}^{\mathrm{T}}_{(m+1)} = f'_m(\mathbf{a}_{(m)}) \overset{\mathrm{C}}{\odot} W^{(1:N_m)\mathrm{T}}_{(m+1)}$$

3. Calculate and retain $\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(m)}$ for all occurrences $m = \overline{\ell, L}$ and, if $k < \ell - 1$ then $\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(\ell-1)}$ is also required. These terms are calculated recursively from $m = k$ to $m = L$ as follows (but only terms $m = \overline{\ell, L}$ are to be retained for later use):
   - if $k = \ell$ then $\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(k)} = I$;
   - if $k = \ell - 1$ then $\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(k+1)}$ was previously calculated;
   - if $k < \ell - 1$ then calculate recursively using:

   $$\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(m)} = \nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(m-1)} \cdot \left(\nabla_{\mathbf{a}_{(m-1)}} \mathbf{a}^{\mathrm{T}}_{(m)} \right)$$

   (go with m from k to L in order to have necessary terms previously computed, note that here $m \geqslant k+2$ as the previous two terms have been already calculated).

4. Calculate and retain $\nabla_{\mathbf{a}_{(k)}} \otimes \mathbf{z}_{(\ell-1)}$ as follows: if $k > \ell - 1$ (i.e. $k = \ell$) then: $\nabla_{\mathbf{a}_{(k)}} \otimes \mathbf{z}_{(\ell-1)} = \widehat{\mathbf{0}}_{N_k \cdot N_{\ell-1}}$; otherwise:

$$\begin{cases} \left\{ \nabla^{\mathrm{T}}_{\mathbf{a}_{(k)}} \mathbf{z}_{(\ell-1)} \right\}_{(1:N_{\ell-1})} = \mathrm{Diag}\left(f'_{\ell-1}(\mathbf{a}_{(\ell-1)}) \right) & \text{if } k = \ell - 1 \\ \left\{ \nabla^{\mathrm{T}}_{\mathbf{a}_{(k)}} \mathbf{z}_{(\ell-1)} \right\}_{(1:N_{\ell-1})} = f'_{\ell-1}(\mathbf{a}_{(\ell-1)}) \overset{\mathrm{C}}{\odot} \left[\left(\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(\ell-1)} \right)^{\mathrm{T}} \right] & \text{if } k < \ell - 1 \end{cases}$$

$$\text{and} \quad \left\{ \nabla^{\mathrm{T}}_{\mathbf{a}_{(k)}} \mathbf{z}_{(\ell-1)} \right\}_0 = \widehat{\mathbf{0}}^{\mathrm{T}}_{N_k}$$

$$\models\Rightarrow \qquad \nabla_{\mathbf{a}_{(k)}} \otimes \mathbf{z}_{(\ell-1)} = \mathrm{Vec}(\nabla^{\mathrm{T}}_{\mathbf{a}_{(k)}} \mathbf{z}_{(\ell-1)})$$

5. Calculate and retain $\nabla^{\mathrm{T}}_{\mathbf{a}_{(m)}} E_p$, for $m = \overline{\ell, L}$, as follows:
   - for output layer: $\nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p = \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p \odot f'_L(\mathbf{a}^{\mathrm{T}}_{(L)})$.
   - recursively for $m = \overline{L-1, \ell}$: $\nabla^{\mathrm{T}}_{\mathbf{a}_{(m)}} E_p = \nabla^{\mathrm{T}}_{\mathbf{a}_{(m+1)}} E_p \cdot \left[\left(\nabla_{\mathbf{a}_{(m)}} \mathbf{a}^{\mathrm{T}}_{(m+1)} \right)^{\mathrm{T}} \right]$ (go backwards in order to have previous terms already calculated).

6. Calculate $\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell)}} E_p$ as follows:
   - first for output layer:

   $$\nabla_{\mathbf{a}_{(L)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p = f'_L(\mathbf{a}_L) \overset{\mathrm{C}}{\odot} \left(\nabla_{\mathbf{z}_{(L)}} \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p \right) + \mathrm{Diag}(\nabla_{\mathbf{z}_{(L)}} E_p \odot f''_L(\mathbf{a}_{(L)}))$$

   - then, if $k < L$, calculate $\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p$ as:

   $$\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p = \nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(k+1)} \cdots \nabla_{\mathbf{a}_{(L-1)}} \mathbf{a}^{\mathrm{T}}_{(L)} \cdot \nabla_{\mathbf{a}_{(L)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p$$

- and finally, if $\ell < L$, calculate $\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell)}} E_p$ recursively as:

$$\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(m)}} E_p = \left[f''_m(\mathbf{a}^{\mathrm{T}}_{(m)}) \odot \left(\nabla^{\mathrm{T}}_{\mathbf{a}_{(m+1)}} E_p \cdot W^{(1:N_m)}_{(m+1)} \right) \right] \overset{\mathrm{R}}{\odot} \left(\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(m)} \right) + f'_m(\mathbf{a}^{\mathrm{T}}_{(m)}) \overset{\mathrm{R}}{\odot} \left[\left(\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(m+1)}} E_p \right) W^{(1:N_m)}_{(m+1)} \right]$$

for $m = \overline{L-1, \ell}$.

7. Finally calculate $\nabla_{W_{(k)}} \otimes \nabla^{\mathrm{T}}_{W_{(\ell)}} E_p$ as:

$$\nabla_{W_{(k)}} \otimes \nabla^{\mathrm{T}}_{W_{(\ell)}} E_p = \mathbf{z}^{\mathrm{T}}_{(k-1)} \otimes \left[\left(\nabla_{\mathbf{a}_{(k)}} \otimes \mathbf{z}_{(\ell-1)} \right) \otimes \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell)}} E_p + \mathbf{z}_{(\ell-1)} \otimes \left(\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell)}} E_p \right) \right] \quad (8.23)$$

Proof. First note that when multiplying a column by a row vector to the right, the Kroneker matrix product, Kroneker tensorial product and outer product yield the same result; also, for any two vectors $\mathbf{a}$ and $\mathbf{b}$, it is true that: $\mathbf{a}^{\mathrm{T}} \otimes \mathbf{b} = \left(\mathbf{a}\mathbf{b}^{\mathrm{T}}\right)^{\mathrm{T}}$.

The error gradient is:

$$\frac{\partial E_p}{\partial w^i_{(\ell)k}} = \frac{\partial E_p}{\partial a_{(\ell)k}} \frac{\partial a_{(\ell)k}}{\partial w^i_{(\ell)k}} = \frac{\partial E_p}{\partial a_{(\ell)k}} z_{(\ell-1)i} \quad \Rightarrow$$

$$\nabla_{W_{(\ell)}} E_p = \nabla_{\mathbf{a}_{(\ell)}} E_p \cdot \mathbf{z}^{\mathrm{T}}_{(\ell-1)} = \mathbf{z}^{\mathrm{T}}_{(\ell-1)} \otimes \nabla_{\mathbf{a}_{(\ell)}} E_p \quad \Rightarrow$$

$$\nabla^{\mathrm{T}}_{W_{(\ell)}} E_p = \mathbf{z}_{(\ell-1)} \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell)}} E_p$$

By the same reasoning:

$$\nabla_{W_{(k)}} \otimes (\cdot) = \left[\mathbf{z}^{\mathrm{T}}_{(k-1)} \otimes \nabla_{\mathbf{a}_{(k)}} \right] \otimes (\cdot) \quad \Rightarrow$$

$$\nabla_{W_{(k)}} \otimes \nabla^{\mathrm{T}}_{W_{(\ell)}} E_p = \mathbf{z}^{\mathrm{T}}_{(k-1)} \otimes \nabla_{\mathbf{a}_{(k)}} \otimes \left[\mathbf{z}_{(\ell-1)} \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell)}} E_p \right]$$
$$= \mathbf{z}^{\mathrm{T}}_{(k-1)} \otimes \left[\left(\nabla_{\mathbf{a}_{(k)}} \otimes \mathbf{z}_{(\ell-1)} \right) \otimes \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell)}} E_p + \mathbf{z}_{(\ell-1)} \otimes \left(\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell)}} E_p \right) \right]$$

(the last equality being correct only because there is no "interference" between the two "dimensions" (horizontal and vertical), i.e. between "pure" covariant and contravariant tensors. This gives formula (8.23), the various terms appearing here are to be calculated.

1. The terms of the form $\mathbf{z}_{(m)}$, as well as $\mathbf{a}_{(m)}$, $f'_m(\mathbf{a}_{(m)})$ and $f''_m(\mathbf{a}_{(m)})$ required as intermediate steps, are easily found directly through a forward propagation (in many cases $f'_m(\mathbf{a}_{(m)})$ and $f''_m(\mathbf{a}_{(m)})$ may be calculated faster by using the $f_m(\mathbf{a}_{(m)})$ values).

2. Terms $\nabla_{\mathbf{a}_{(n)}} \mathbf{a}^{\mathrm{T}}_{(m)}$ will be required as intermediate steps:

- if $n > m$ then there is no path *from* a neuron in layer n *to* a neuron in layer m, i.e. $\mathbf{a}_{(m)}$ is independent of $\mathbf{a}_{(n)}$ (as it comes "before" in the order of data flow) and thus: $\frac{\partial a^i_{(m)}}{\partial a_{(n)k}} = 0 \Leftrightarrow \nabla_{\mathbf{a}_{(n)}} \mathbf{a}^{\mathrm{T}}_{(m)} = \widetilde{0}$;
- if $n = m$ then obviously: $\frac{\partial a^i_{(n)}}{\partial a_{(n)k}} = \delta^i_k \Leftrightarrow \nabla_{\mathbf{a}_{(n)}} \mathbf{a}^{\mathrm{T}}_{(n)} = I$;
- if $n < m$ then $\nabla_{\mathbf{a}_{(n)}} \mathbf{a}^{\mathrm{T}}_{(m)}$ may be calculated recursively:

$$\left\{ \frac{\partial {a'}^i_{(m)}}{\partial a_{(n)k}} \right\} = \left\{ \frac{\partial {a'}^i_{(m)}}{\partial a_{(m-1)h}} \right\} \left\{ \frac{\partial {a'}^h_{(m-1)}}{\partial a_{(n)k}} \right\} \quad \Leftrightarrow$$

$$\nabla_{\mathbf{a}_{(n)}} \mathbf{a}^{\mathrm{T}}_{(m)} = \nabla_{\mathbf{a}_{(n)}} \mathbf{a}^{\mathrm{T}}_{(m-1)} \cdot \nabla_{\mathbf{a}_{(m-1)}} \mathbf{a}^{\mathrm{T}}_{(m)}$$

As $\mathbf{a}_{(m)} = W_{(m)} f_{m-1}(\mathbf{a}_{(m-1)})$ (consider here $f_{m-1}(a_{(m-1)0}) \equiv$ const. for bias, it will disappear shortly) then $\left\{ {a'}^i_{(m)} \right\} = \left\{ f_{m-1}({a'}^h_{(m-1)}) \right\} \left\{ {w'}^i_{(m)h} \right\}$ and:

$$\left\{ \frac{\partial {a'}^i_{(m)}}{\partial a_{(m-1)h}} \right\} = \left\{ f'_{m-1}({a'}^g_{(m-1)})\, \delta^g_h \right\} \left\{ {w'}^i_{(m)g} \right\}$$

The bias term disappears because the corresponding activation derivative is zero and finally:

$$\nabla_{\mathbf{a}_{(m-1)}} \mathbf{a}^{\mathrm{T}}_{(m)} = \mathrm{Diag}(f'_{m-1}(\mathbf{a}_{(m-1)})) W^{(1:N_{m-1})\mathrm{T}}_{(m)} = f'_{m-1}(\mathbf{a}^{\mathrm{T}}_{(m-1)}) \overset{\mathrm{C}}{\odot} W^{(1:N_{m-1})\mathrm{T}}_{(m)}$$

3. Term $\nabla_{\mathbf{a}_{(k)}} \otimes \mathbf{z}_{(\ell-1)} = \left\{ \frac{\partial z_{(\ell-1)i}}{\partial a_{(k)k}} \right\}_{ki}$ (this represents a vector with $N_k \cdot (N_{\ell-1} + 1)$ components, including the bias related ones; components are arranged in lexicographic order, k being the first index and i the second one).

As $z_{(\ell-1)i} = \begin{cases} 1 & \text{for } i = 0 \text{ (bias)} \\ f_{\ell-1}(a_{(\ell-1)i}) & \text{otherwise} \end{cases}$:

$$\frac{\partial z_{(\ell-1)i}}{\partial a_{(k)k}} = f'_{\ell-1}(a_{(\ell-1)i}) \frac{\partial a_{(\ell-1)i}}{\partial a_{(k)k}} \quad \Rightarrow$$

$$\begin{aligned} \nabla_{\mathbf{a}_{(k)}} \otimes \mathbf{z}_{(\ell-1)(1:N_{\ell-1})} &= \left(\widehat{\mathbf{1}}_{N_k} \otimes f'_{\ell-1}(\mathbf{a}_{(\ell-1)}) \right) \odot \left(\nabla_{\mathbf{a}_{(k)}} \otimes \mathbf{a}_{(\ell-1)} \right) \\ &= \mathrm{Vec}\left[f'_{\ell-1}(\mathbf{a}_{(\ell-1)}) \overset{\mathrm{C}}{\odot} \left(\nabla^{\mathrm{T}}_{\mathbf{a}_{(k)}} \otimes \mathbf{a}_{(\ell-1)} \right) \right] \end{aligned}$$

$$\text{and} \quad \nabla_{\mathbf{a}_{(k)}} z_{(\ell-1)0} = \widehat{\mathbf{0}}$$

There are three cases to be discussed:

- $k > \ell - 1$ (i.e. $k = \ell$) then there is no path from k to $\ell - 1$, $\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(\ell-1)} = \widetilde{0}$, thus:
$$\nabla_{\mathbf{a}_{(k)}} \otimes \mathbf{z}_{(\ell-1)} = \widehat{\mathbf{0}}$$
- $k = \ell - 1$ then $\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(\ell-1)} = \{\delta^i_k\}$ and:
$$\nabla_{\mathbf{a}_{(k)}} \otimes \mathbf{z}_{(\ell-1)(1:N_{\ell-1})} \equiv \nabla_{\mathbf{a}_{(\ell-1)}} \otimes \mathbf{z}_{(\ell-1)(1:N_{\ell-1})} = \mathrm{Vec}\left[\mathrm{Diag}\left(f'_{\ell-1}(\mathbf{a}_{(\ell-1)}) \right) \right]$$
- $k < \ell - 1$ then $\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(\ell-1)}$ is to be calculated recursively as previously outlined.

4. Terms $\nabla^{\mathrm{T}}_{\mathbf{a}_{(m)}} E_p = \left\{ \frac{\partial E_p}{\partial {a'}^h_{(m)}} \right\}$. The error depends on all intermediate layers between layer m and output, first intermediate layer being layer $m + 1$; then (consider first $m < L$):

$$\left\{ \frac{\partial E_p}{\partial {a'}^h_{(m)}} \right\} = \left\{ \frac{\partial E_p}{\partial {a'}^g_{(m+1)}} \right\} \left\{ \frac{\partial a_{(m+1)g}}{\partial {a'}^h_{(m)}} \right\} \;\Rightarrow\; \nabla^{\mathrm{T}}_{\mathbf{a}_{(m)}} E_p = \nabla^{\mathrm{T}}_{\mathbf{a}_{(m+1)}} E_p \cdot \nabla^{\mathrm{T}}_{\mathbf{a}_{(m)}} \mathbf{a}_{(m+1)}$$

For output layer the $\nabla_{\mathbf{z}_{(L)}} E_p$ is assumed known (usually may be easily calculated directly, note that $z_{(L)0}$ does *not* exists):

$$\left\{ \frac{\partial E_p}{\partial {a'}^h_{(L)}} \right\} = \left\{ \frac{\partial E_p}{\partial {z'}^g_{(L)}} \right\} \left\{ \frac{\partial z_{(L)g}}{\partial {a'}^h_{(L)}} \right\} = \left\{ \frac{\partial E_p}{\partial {z'}^g_{(L)}} \right\} \left\{ f'_L(a_{(L)g})\, \delta^h_g \right\} \quad \Rightarrow$$

$$\nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p = \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p \cdot \mathrm{Diag}(f'_L(\mathbf{a}_{(L)})) = \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p \odot f'_L(\mathbf{a}^{\mathrm{T}}_{(L)})$$

5. Term $\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell)}} E_p$. Taking a backpropagation approach, considering just one component $\frac{\partial}{\partial a_{(k)k}} \frac{\partial E_p}{\partial {a'}^h_{(\ell)}}$, the neuron h affects error through all subsequent neurons to which it sends connection ($h \neq 0$):

$$\frac{\partial E_p}{\partial {a'}^h_{(\ell)}} = f'_\ell(a_{(\ell)h}) \sum_{g=1}^{N_{\ell+1}} w^h_{(\ell+1)g} \frac{\partial E_p}{\partial a_{(\ell+1)g}} \quad \Rightarrow \quad \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell)}} E_p = f'_\ell(\mathbf{a}^{\mathrm{T}}_{(\ell)}) \odot \left(\nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell+1)}} E_p \cdot W^{(1:N_\ell)}_{(\ell+1)} \right)$$

$$\begin{aligned}
&\nabla_{\mathbf{a}_{(k)}} \left[f'_\ell(\mathbf{a}^{\mathrm{T}}_{(\ell)}) \odot \left(\nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell+1)}} E_p \cdot W^{(1:N_\ell)}_{(\ell+1)} \right) \right] \\
&= \left[\nabla_{\mathbf{a}_{(k)}} f'_\ell(\mathbf{a}^{\mathrm{T}}_{(\ell)}) \right] \odot \left[\widehat{\mathbf{1}}_{N_k} \left(\nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell+1)}} E_p \cdot W^{(1:N_\ell)}_{(\ell+1)} \right) \right] \\
&\quad + \left(\widehat{\mathbf{1}}_{N_k} f'_\ell(\mathbf{a}^{\mathrm{T}}_{(\ell)}) \right) \odot \left[\nabla_{\mathbf{a}_{(k)}} \left(\nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell+1)}} E_p \cdot W^{(1:N_\ell)}_{(\ell+1)} \right) \right]
\end{aligned}$$

The first square parenthesis from first term is:

$$\nabla_{\mathbf{a}_{(k)}} f'_\ell(\mathbf{a}^{\mathrm{T}}_{(\ell)}) = \left(\widehat{\mathbf{1}}_{N_k} f''_\ell(\mathbf{a}^{\mathrm{T}}_{(\ell)}) \right) \odot \left(\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(\ell)} \right)$$

and then the whole first term may be rewritten as:

$$\left[f''_\ell(\mathbf{a}^{\mathrm{T}}_{(\ell)}) \odot \left(\nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell+1)}} E_p \, W^{(1:N_\ell)}_{(\ell+1)} \right) \right] \overset{\mathrm{R}}{\odot} \left(\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(\ell)} \right)$$

The second term may be rewritten as:

$$f'_\ell(\mathbf{a}^{\mathrm{T}}_{(\ell)}) \overset{\mathrm{R}}{\odot} \left[\left(\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell+1)}} E_p \right) W^{(1:N_\ell)}_{(\ell+1)} \right]$$

Finally an recursive formula is obtained:

$$\begin{aligned}
\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell)}} E_p = &\left[f''_\ell(\mathbf{a}^{\mathrm{T}}_{(\ell)}) \odot \left(\nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell+1)}} E_p \, W^{(1:N_\ell)}_{(\ell+1)} \right) \right] \overset{\mathrm{R}}{\odot} \left(\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(\ell)} \right) \\
&+ f'_\ell(\mathbf{a}^{\mathrm{T}}_{(\ell)}) \overset{\mathrm{R}}{\odot} \left[\left(\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell+1)}} E_p \right) W^{(1:N_\ell)}_{(\ell+1)} \right]
\end{aligned}$$

Discussion:

- These formulas will hold provided that $k \leqslant \ell$ (as assumed in this algorithm), i.e. $W_{(k)}$ weights do not appear explicitly in $\nabla_{\mathbf{a}_{(\ell+1)}} E_p \cdot W^{(1:N_\ell)}_{(\ell+1)}$ expression because $\nabla_{\mathbf{a}_{(k)}}(\cdot)$ is used to calculate $\nabla_{W_{(k)}}(\cdot)$. Otherwise (i.e. for the case $k > \ell$), assuming Hessian continuous, calculate $\nabla_{\mathbf{a}_{(\ell)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(k)}} E_p$ and then $\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(\ell)}} E_p \equiv \left(\nabla_{\mathbf{a}_{(\ell)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(k)}} E_p \right)^{\mathrm{T}}$.
- The method of calculating $\nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(\ell)}$ has been previously discussed (at stage **2** within this proof).

In order to apply the above formula it is necessary to provide a starting point for the recurrence. This requires the explicit calculation of the following terms:

- Term $\nabla_{\mathbf{a}_{(L)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p$. As $\nabla_{\mathbf{a}_{(L)}} E_p$ has been calculated previously then:

  $$\nabla_{\mathbf{a}_{(L)}} \left(\nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p \right) = \left(\nabla_{\mathbf{a}_{(L)}} \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p \right) \overset{\mathrm{R}}{\odot} f'_L(\mathbf{a}^{\mathrm{T}}_{(L)}) + \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p \overset{\mathrm{R}}{\odot} \left(\nabla_{\mathbf{a}_{(L)}} f'_L(\mathbf{a}^{\mathrm{T}}_{(L)}) \right)$$

  Processing further into the first term gives:

  $$\begin{aligned}
  \nabla_{\mathbf{a}_{(L)}} \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p &= \left(\nabla_{\mathbf{a}_{(L)}} \mathbf{z}^{\mathrm{T}}_{(L)} \right) \left(\nabla_{\mathbf{z}_{(L)}} \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p \right) = \mathrm{Diag}(f'_L(\mathbf{a}_{(L)})) \left(\nabla_{\mathbf{z}_{(L)}} \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p \right) \\
  &= f'_L(\mathbf{a}_{(L)}) \overset{\mathrm{C}}{\odot} \left(\nabla_{\mathbf{z}_{(L)}} \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p \right)
  \end{aligned}$$

  while in the second:

  $$\nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p \overset{\mathrm{R}}{\odot} \left(\nabla_{\mathbf{a}_{(L)}} f'_L(\mathbf{a}^{\mathrm{T}}_{(L)}) \right) = \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p \overset{\mathrm{R}}{\odot} \mathrm{Diag}(f''_L(\mathbf{a}_{(L)})) = \mathrm{Diag}(\nabla_{\mathbf{z}_{(L)}} E_p \odot f''_L(\mathbf{a}_{(L)}))$$

  and thus finally:

  $$\nabla_{\mathbf{a}_{(L)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p = f'_L(\mathbf{a}_{(L)}) \overset{\mathrm{C}}{\odot} \left(\nabla_{\mathbf{z}_{(L)}} \nabla^{\mathrm{T}}_{\mathbf{z}_{(L)}} E_p \right) + \mathrm{Diag}(\nabla_{\mathbf{z}_{(L)}} E_p \odot f''_L(\mathbf{a}_{(L)}))$$

- Term $\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p$ for $k \neq L$ is calculated through a recursion (similar to $\nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p$ computation):

  $$\nabla_{\mathbf{a}_{(k)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p = \nabla_{\mathbf{a}_{(k)}} \mathbf{a}^{\mathrm{T}}_{(k+1)} \nabla_{\mathbf{a}_{(k+1)}} \nabla^{\mathrm{T}}_{\mathbf{a}_{(L)}} E_p \quad \square$$

Remarks:

- The overall speed may be improved if some terms, for different p, are calculated at once by transforming a matrix–vector multiplication to a matrix-matrix operation performed faster overall through Strassen or related algorithms for fast matrix–matrix multiplication; however this represents also a trade-off in terms of memory (RAM) usage.

8.4.5 Multiplication With Hessian

❖ A

Some algorithms (e.g. conjugate gradients) do not require directly the Hessian but rather a term of the form $A^{\mathrm{T}} \circ H$ where A is a constant tensor with the same layout as W. This term may be calculated through a finite difference approach:

$$A^{\mathrm{T}} \circ H \simeq \frac{1}{2\varepsilon} \left[\nabla^{\mathrm{T}} E|_{W+\varepsilon A} - \nabla^{\mathrm{T}} E|_{W-\varepsilon A} \right] \tag{8.24}$$

❖ $\mathbf{a}$, $\mathbf{w}$, $\widetilde{H}$

Proof. Consider a transformation to vectors and matrices: $A \to \mathbf{a} = \mathrm{Vec}(A^{\mathrm{T}})$, $W \to \mathbf{w} = \mathrm{Vec}(W^{\mathrm{T}})$ and $\widetilde{H} = \nabla_{\mathbf{w}} \nabla_{\mathbf{w}}^{\mathrm{T}} E$. Then:

$$\mathbf{a}^{\mathrm{T}} \widetilde{H} = \frac{1}{2\varepsilon} \left[\nabla_{\mathbf{w}}^{\mathrm{T}} E|_{\mathbf{w}+\varepsilon\mathbf{a}} - \nabla_{\mathbf{w}}^{\mathrm{T}} E|_{\mathbf{w}-\varepsilon\mathbf{a}} \right] + \mathcal{O}(\varepsilon^2) \quad \square$$

Equation (8.24) involves only two supplementary gradient computations (instead of a Hessian computation) which may be efficiently performed through the backpropagation procedure. Alternatively it may be possible to approximate $A^{\mathrm{T}} \circ H$ by just one supplemental gradient computation as:

$$A^{\mathrm{T}} \circ H \simeq \frac{1}{\varepsilon} \left[\nabla^{\mathrm{T}} E|_{W+\varepsilon A} - \nabla^{\mathrm{T}} E|_{W} \right] \; .$$

8.4.6 Avoiding Inverse Hessian Computation

Some algorithms require (at intermediate levels) the manipulation of inverse Hessian. The calculation of a matrix inverse is a very computationally intensive process, fortunately in most cases it may be avoided, leading to solutions with better numerical stability.

❖ $\mathbf{w}$, $\widetilde{H}$

First the weight matrix is transformed to a vector and thus the Hessian becomes a symmetric matrix:

$$\mathbf{w} = \mathrm{Vec}\, W \quad \text{and} \quad \widetilde{H} = \nabla_{\mathbf{w}} \nabla_{\mathbf{w}}^{\mathrm{T}} E$$

Two possible cases are analyzed below (others may be similar):

- $\widetilde{H}(\mathbf{a} - \mathbf{x}) = \mathbf{b}$ where $\mathbf{a}$ and $\mathbf{b}$ are known and $\mathbf{x}$ is required. This kind of equation appears in Newton's method. The mathematical solution is $\mathbf{x} = \mathbf{a} - \widetilde{H}^{-1}\mathbf{b}$ and involves the inverse Hessian. Computationally it is more effective to solve the required equation through a factorization of $\widetilde{H}$. There are two possibilities:

Cholesky factorization ❖ $\widetilde{H}_G$

  - $\widetilde{H}$ is positive definite then as $\widetilde{H}$ is symmetric there exists a Cholesky factorization: $\widetilde{H} = \widetilde{H}_G \widetilde{H}_G^{\mathrm{T}}$ where $\widetilde{H}_G$ is a lower triangular matrix of the same size as $\widetilde{H}$ known

8.4.5 See [Bis95] pp. 150–160.
8.4.6 See [GL96] pg. 121, 143.

as *Cholesky factor* (the decomposition is unique). The problem is reduced to solving two triangular systems of linear equations:

$$\widetilde{H}_G \mathbf{z} = \mathbf{b} \quad \text{and} \quad \widetilde{H}_G^{\mathrm{T}} \mathbf{y} = \mathbf{z} \quad \Rightarrow \quad \mathbf{x} = \mathbf{a} - \mathbf{y}$$

  – $\widetilde{H}$ is *not* positive definite; in this situation it is possible to generate a LU factorization in the form $P\widetilde{H} = LU$, where P is a permutation matrix, L is a lower–triangular matrix while U is a upper–triangular one. The problem is reduced to solving two triangular systems of linear equations as above. The LU factorization is not as fast and storage efficient as the Cholesky factorization but still faster that the computation of the inverse. LU factorization

- $\mathbf{a}^{\mathrm{T}} \widetilde{H}^{-1} \mathbf{b} = x$, where $\mathbf{a}$ and $\mathbf{b}$ are known and x is to be found. This kind of equation appears in the optimal brain surgeon technique of optimization. This may be solved as follows: first perform a LU factorization of $\widetilde{H}$ then solve the following triangular systems: $L\mathbf{y} = P\mathbf{b}$ and then $U\mathbf{z} = \mathbf{y}$; finally $x = \mathbf{a}^{\mathrm{T}}\mathbf{z}$. Note that $\widetilde{H}$ is symmetric, if it is also positive definite then it is faster and more storage efficient to perform a Cholesky factorization instead.

8.5 Hessian Based Learning Algorithms

8.5.1 Newton's Method

In order to better understand the basic idea of this method the weight matrix will be converted to a vector, using lexicographic convention: ❖ $\mathbf{w}$

$$W = \{w_k^i\} \quad \rightarrow \quad \mathbf{w}^{\mathrm{T}} = \begin{pmatrix} w_1^1 & w_1^2 & \cdots & w_2^1 & w_2^2 & \cdots \end{pmatrix}$$

and a similar transformation is performed on Hessian in order to build a *symmetric* matrix: ❖ $\widetilde{H}$

$$\{h_{kj}^{i\ell}\} \quad \rightarrow \quad \widetilde{H} = \begin{pmatrix} h_{11}^{11} & h_{12}^{11} & h_{13}^{11} & \cdots \\ h_{11}^{21} & h_{12}^{21} & h_{13}^{21} & \\ h_{11}^{31} & h_{12}^{31} & h_{13}^{31} & \\ \vdots & & & \end{pmatrix}$$

(note the way of ordering: $\{k, i\}$ vertically and $\{\ell, j\}$ horizontally; $h_{kj}^{i\ell} = h_{\ell i}^{jk}$).

A Taylor series development of error around some point W_0 leads to:

$$\begin{aligned} E(W) =& E(W_0) + (\mathbf{w} - \mathbf{w}_0)^{\mathrm{T}} \nabla_{\mathbf{w}} E|_{W_0} \\ &+ \frac{1}{2} (\mathbf{w} - \mathbf{w}_0)^{\mathrm{T}} \widetilde{H} (\mathbf{w} - \mathbf{w}_0) + \mathcal{O}(\|\mathbf{w} - \mathbf{w}_0\|^3) \end{aligned} \tag{8.25}$$

Newton's method is based on the assumption that error may be approximated by a quadratic around some (local) minima W_0. In this case $\nabla E|_{W_0} = \widetilde{0}$ and higher-order terms from (8.25) are neglected:

$$E(W) = E(W_0) + \frac{1}{2} (\mathbf{w} - \mathbf{w}_0)^{\mathrm{T}} \widetilde{H} (\mathbf{w} - \mathbf{w}_0)$$

[8.5.1] See [GL96] pp.142–143, [Has95] pp. 215–216 and [Bis95] pp. 285–287, the algorithm described here avoids explicit computation of inverse Hessian by using linear algebra factorization techniques.

From the equation above: $\nabla_{\mathbf{w}} E = \widetilde{H}(\mathbf{w} - \mathbf{w}_0)$ This leads to the following learning equation:

$$\mathbf{w}_0 = \mathbf{w} - \widetilde{H}^{-1} \nabla_{\mathbf{w}} E \tag{8.26}$$

If $\widetilde{H}$ is symmetric positive definite, then so is $\widetilde{H}^{-1}$ and the $\widetilde{H}^{-1} \nabla_{\mathbf{w}} E$ direction, called the *Newton direction* is a descent direction for the error (unless we are at a stationary point already).

Proof. Assume a move from W along $\widetilde{H}^{-1} \nabla_{\mathbf{w}} E$ such that the new current weights become $\mathbf{w} - \alpha \widetilde{H}^{-1} \mathbf{w}$, where $\alpha > 0$ is a constant. Then ($\widetilde{H}$ symmetric $\Rightarrow \left(\widetilde{H}^{-1}\right)^{\mathrm{T}} = \widetilde{H}^{-1}$):

$$\frac{\partial E(\mathbf{w} - \alpha \widetilde{H}^{-1} \nabla_{\mathbf{w}} E)}{\partial \alpha} = -(\widetilde{H}^{-1} \nabla_{\mathbf{w}} E)^{\mathrm{T}} \nabla_{\mathbf{w}} E = -\nabla_{\mathbf{w}}^{\mathrm{T}} E \cdot \widetilde{H}^{-1} \cdot \nabla_{\mathbf{w}} E$$

$\widetilde{H}^{-1}$ positive definite implies $\mathbf{a}^{\mathrm{T}} \widetilde{H}^{-1} \mathbf{a} > 0$, $\forall \mathbf{a} \neq \mathbf{0}$, thus $\partial E / \partial \alpha < 0$. □

Remarks:

➥ The direct approach of calculating H and then $\widetilde{H}^{-1}$ would be computationally very intensive, fortunately calculation of $\widetilde{H}^{-1}$ may be replaced with a LU factorization of Hessian (which may be calculated in parallel with ∇E).

LU factorization

Finding W_0 from (8.26) is equivalent to find $\Delta W = W - W_0$ from:

$$\widetilde{H} \Delta \mathbf{w} = \nabla_{\mathbf{w}} E \tag{8.27}$$

(as W is the known current point). There are two possibilities:

Cholesky factorization
❖ $\widetilde{H}_G$, $\Delta\widetilde{\mathbf{w}}$

1. $\widetilde{H}$ is positive definite — then as $\widetilde{H}$ is symmetric there exists a Cholesky factorization: $\widetilde{H} = \widetilde{H}_G \widetilde{H}_G^{\mathrm{T}}$ where $\widetilde{H}_G$ is a lower triangular matrix of the same size as $\widetilde{H}$ known as *Cholesky factor* (the decomposition is unique). This reduces (8.27) to solving two triangular systems (unknowns are $\Delta\widetilde{\mathbf{w}}$, $\Delta\mathbf{w}$):

$$\widetilde{H}_G \Delta\widetilde{\mathbf{w}} = \nabla_{\mathbf{w}} E \quad \text{and} \quad \widetilde{H}_G^{\mathrm{T}} \Delta\mathbf{w} = \Delta\widetilde{\mathbf{w}} \tag{8.28}$$

2. $\widetilde{H}$ is *not* positive definite then transform $\widetilde{H} \to \widetilde{H} + \alpha I$ with the constant α suitably chosen as to make the new $\widetilde{H} + \alpha I$ positive definite. Then apply the same procedure as above.

Remarks:

➥ When performing the $\widetilde{H} \to \widetilde{H} + \alpha I$ transformation:

- for $\alpha \gtrsim 0 \Rightarrow \widetilde{H} + \alpha I \simeq \widetilde{H}$ and the resulting direction is close to Newton's one;
- for $\alpha \gg 0 \Rightarrow \widetilde{H} + \alpha I \simeq \alpha I \Rightarrow \widetilde{H}^{-1} \simeq I/\alpha$ and the resulting direction is close to ∇E (the same as used by the "standard" but slow steepest descent algorithm).

➥ The Cholesky factorization requires $\mathcal{O}(N_W^3/3)$ operations[4] compared with the inversion of the Hessian which requires $\mathcal{O}(N_W^3)$; the solving of linear triangular systems require $\mathcal{O}(N_W^2)$ operations each.

[4] Or flops, here they are "new" flops, as defined in [GL96] pp. 18, i.e. a floating point multiplication or division, together with one floating point addition or substraction.

8.5.2 Levenberg–Marquardt Algorithm

This algorithm is specifically designed for "sum–of–squares" error. Let ε_p be the error given by the p-th training pattern vector and $\boldsymbol{\varepsilon}^{\mathrm{T}} = \begin{pmatrix} \varepsilon_1 & \dots & \varepsilon_P \end{pmatrix}$. Then the error may be written as: ❖ ε_p, $\boldsymbol{\varepsilon}$

$$E = \frac{1}{2} \sum_{p=1}^{P} (\varepsilon_p)^2 = \frac{1}{2} \|\boldsymbol{\varepsilon}\|^2 \tag{8.29}$$

The weight matrix will be converted to a vector, e.g. using lexicographic convention: ❖ $\mathbf{w}$

$$W = \{w_k^i\} \quad \to \quad \mathbf{w}^{\mathrm{T}} = \begin{pmatrix} w_1^1 & w_1^2 & \cdots & w_2^1 & w_2^2 & \cdots \end{pmatrix}$$

The following matrix is defined:

$$\nabla_{\mathbf{w}}^{\mathrm{T}} \otimes \boldsymbol{\varepsilon} = \begin{pmatrix} \frac{\partial \varepsilon_1}{\partial w_1^1} & \frac{\partial \varepsilon_1}{\partial w_1^2} & \cdots \\ \frac{\partial \varepsilon_2}{\partial w_1^1} & \frac{\partial \varepsilon_2}{\partial w_1^2} & \\ \vdots & & \ddots \end{pmatrix}$$

which may be efficiently computed through a backpropagation procedure.

Considering a small variation in W weights from step t to $t+1$, the error vector $\boldsymbol{\varepsilon}$ may be developed in a Taylor series to the first order:

$$\boldsymbol{\varepsilon}_{(t+1)} \simeq \boldsymbol{\varepsilon}_{(t)} + \left(\nabla_{\mathbf{w}}^{\mathrm{T}} \otimes \boldsymbol{\varepsilon}\right) \cdot (\mathbf{w}_{(t+1)} - \mathbf{w}_{(t)})$$

Let $\Delta\mathbf{w} = \mathbf{w}_{(t+1)} - \mathbf{w}_{(t)}$ and the error function at step $t+1$ is: ❖ $\Delta\mathbf{w}$

$$E_{(t+1)} = \frac{1}{2} \|\boldsymbol{\varepsilon}_{(t)} + \nabla_{\mathbf{w}}^{T} \otimes \boldsymbol{\varepsilon} \cdot \Delta\mathbf{w}\|^2 \tag{8.30}$$

Minimizing (8.30) with respect to $W_{(t+1)}$ means:

$$\nabla_{\mathbf{w}} E|_{W_{(t+1)}} = \nabla_{\mathbf{w}} \boldsymbol{\varepsilon}^{\mathrm{T}} \left[\boldsymbol{\varepsilon}_{(t)} + \nabla_{\mathbf{w}}^{T} \otimes \boldsymbol{\varepsilon} \cdot \Delta\mathbf{w}\right] = \widehat{\mathbf{0}} \quad \Rightarrow \quad \boldsymbol{\varepsilon}_{(t)} + \nabla_{\mathbf{w}}^{T} \otimes \boldsymbol{\varepsilon} \cdot \Delta\mathbf{w} = \widehat{\mathbf{0}}$$

so the problem is to solve the linear system:

$$\nabla_{\mathbf{w}}^{T} \otimes \boldsymbol{\varepsilon} \cdot \Delta\mathbf{w} = -\boldsymbol{\varepsilon}_{(t)} \tag{8.31}$$

and find $\Delta\mathbf{w}$ (i.e. $\mathbf{w}_{(t+1)}$), knowing $\boldsymbol{\varepsilon}_{(t)}$ and $\nabla_{\mathbf{w}}^{T} \otimes \boldsymbol{\varepsilon}$.

In general $P \gg N_W$ and (8.31) will represent an overdetermined system with no solution. This happens because both inputs and targets may contain an amount of noise and reducing error to zero is not necessarily desirable or possible. The alternative of selecting a larger network so as to have $P \leqslant N_W$ may lead to a system who will learn the training set well but with poor generalization capabilities, i.e. the system will act as a memory not as a predictor.

The solution to (8.31) is usually found through a least squares technique. This means that a $\Delta\mathbf{w}$ is sought such that: least squares

$$\nabla_{\mathbf{w}}^{T} \otimes \boldsymbol{\varepsilon} \cdot \Delta\mathbf{w} + \boldsymbol{\varepsilon}_{(t)} = \mathbf{r} \quad \text{and} \quad \|\mathbf{r}\|^2 = \min$$

where $\mathbf{r}$ is a residual. ❖ $\mathbf{r}$

[8.5.2] See [Bis95] pp. 290–292, the treatment here is *tensorial* and using linear algebra factorization techniques.

Assuming that $\nabla^T_{\mathbf{w}} \otimes \varepsilon$ has (full) rank N_W then two of the possible approaches are:

- *The method of normal equations.* Multiplying (8.31) to the left by $\nabla_{\mathbf{w}} \varepsilon^{\mathrm{T}}$ gives:

$$\nabla_{\mathbf{w}} \varepsilon^{\mathrm{T}} \cdot (\nabla^T_{\mathbf{w}} \otimes \varepsilon) \cdot \Delta \mathbf{w} = -\nabla_{\mathbf{w}} \varepsilon^{\mathrm{T}} \cdot \varepsilon_{(t)}$$

normal equations

called the normal equations. After a Cholesky factorization of $\nabla_{\mathbf{w}} \varepsilon^{\mathrm{T}} \cdot (\nabla^T_{\mathbf{w}} \otimes \varepsilon)$, the systems of linear equations are solved in a similar fashion as in (8.28).

Proof. Consider the function $\varphi(\Delta \mathbf{w}) = \frac{1}{2} \|\nabla^T_{\mathbf{w}} \otimes \varepsilon \cdot \Delta \mathbf{w} + \varepsilon_{(t)}\|^2$. Then $\nabla_{\mathbf{w}} \varphi(\Delta \mathbf{w}) = \nabla_{\mathbf{w}} \varepsilon^{\mathrm{T}} \left(\nabla^T_{\mathbf{w}} \otimes \varepsilon \cdot \Delta \mathbf{w} + \varepsilon_{(t)}\right)$ and finding $\mathbf{r}$ such that $\|\mathbf{r}\|^2$ is minimal is equivalent to finding $\Delta \mathbf{w}$ such that $\nabla_{\mathbf{w}} \varphi(\Delta \mathbf{w}) = \widehat{\mathbf{0}}$. ☐

QR factorization

- *The method of QR factorization.* The $P \times N_W$ matrix $\nabla^{\mathrm{T}}_{\mathbf{w}} \otimes \varepsilon$ may be decomposed in two matrices Q and R such that: Q is an orthogonal $P \times P$ matrix, R is an $P \times N_W$ upper–triangular matrix and $Q^{\mathrm{T}} \cdot \nabla^{\mathrm{T}}_{\mathbf{w}} \otimes \varepsilon = R$ (note that $R_{(N_W+1:P)} = \widetilde{0}$).

  The least squares solution to (8.31) is found by solving the triangular system:

$$R_{(1:N_W)} \Delta \mathbf{w} = -\left(Q^{\mathrm{T}} \varepsilon\right)_{(1:N_W)}$$

Proof. As Q is orthogonal, R is upper-triangular and $Q^{\mathrm{T}} \cdot \nabla^T_{\mathbf{w}} \otimes \varepsilon = R$ then:

$$\|\nabla^T_{\mathbf{w}} \otimes \varepsilon \cdot \Delta \mathbf{w} + \varepsilon_{(t)}\|^2 = \|Q^{\mathrm{T}} \cdot (\nabla^T_{\mathbf{w}} \otimes \varepsilon) \cdot \Delta \mathbf{w} + Q^{\mathrm{T}} \varepsilon_{(t)}\|^2 = \|R \Delta \mathbf{w} + Q^{\mathrm{T}} \varepsilon_{(t)}\|$$

Since R is upper–triangular $P \times N_W$ with rank N_W the choice of $\Delta \mathbf{w}$ does not change components $N_W \ldots P$ of $R \Delta \mathbf{w} + Q^{\mathrm{T}} \varepsilon_{(t)}$. Therefore:

$$\|\nabla^T_{\mathbf{w}} \otimes \varepsilon \cdot \Delta \mathbf{w} + \varepsilon_{(t)}\|^2 = \left\| R_{(1:N_W)} \Delta \mathbf{w} + \left(Q^{\mathrm{T}} \varepsilon\right)_{(1:N_W)} \right\|^2 + \left\| \left(Q^{\mathrm{T}} \varepsilon\right)_{(N_W+1:P)} \right\|^2$$

and the minimum of this quantity over choices of $\Delta \mathbf{w}$ occurs when $\Delta \mathbf{w}$ is chosen to the first norm on the right, i.e. $R_{(1:N_W)} \Delta \mathbf{w} = -\left(Q^{\mathrm{T}} \varepsilon\right)_{(1:N_W)}$. ☐

The Levenberg–Marquardt algorithm basically involves implicitly the inverse Hessian in its main update formula.

Proof. An approximate solution to (8.31), considering the system as being *compatible* and *overdetermined*, is obtained by multiplication to the left, first by $\nabla_{\mathbf{w}} \varepsilon^{\mathrm{T}}$ and then by $\left(\nabla_{\mathbf{w}} \varepsilon^{\mathrm{T}} \cdot \nabla^{\mathrm{T}}_{\mathbf{w}} \otimes \varepsilon\right)^{-1}$ which gives:

$$\mathbf{w}_{(t+1)} = \mathbf{w}_{(t)} - \left(\nabla_{\mathbf{w}} \varepsilon^{\mathrm{T}} \cdot \nabla^{\mathrm{T}}_{\mathbf{w}} \otimes \varepsilon\right)^{-1} \nabla_{\mathbf{w}} \varepsilon^{\mathrm{T}} \cdot \varepsilon \tag{8.32}$$

which represents the core of Levenberg-Marquardt weights update formula.

From (8.29) the Hessian may be written as a *matrix* $\{h^v_w\}$, of the form:

$$h^v_w = \frac{\partial^2 E}{\partial w_w \partial w'^v} = \sum_{p=1}^{P} \left(\frac{\partial \varepsilon_p}{\partial w_w} \frac{\partial \varepsilon_p}{\partial w'^v} + \varepsilon_p \frac{\partial^2 \varepsilon_p}{\partial w_w \partial w'^v} \right)$$

and by neglecting the second order derivatives the Hessian may be approximated as:

$$\{h^v_w\} \simeq \nabla_{\mathbf{w}} \varepsilon^{\mathrm{T}} \cdot \nabla^{\mathrm{T}}_{\mathbf{w}} \otimes \varepsilon$$

i.e. the equation (8.32) essentially involves the inverse Hessian. However this is done through the computation of the error gradient with respect to weights. ☐

Remarks:

➥ The Levenberg–Marquardt algorithm is of *model trust region* type. The model — the linearized approximate error function — is "trusted" just around the current point W.

➥ Because of the above remark and as the algorithm may give a relatively large $\Delta\mathbf{w}$, in a practical implementation, steps have to be taken to limit the change in weights.

Bibliography

[BB95] *James M. Bower and David Beeman*. **The Book of Genesis**. Springer–Verlag, New York, 1995. ISBN 0–387–94019–7.

[Bis95] *Cristopher M. Bishop*. **Neural Networks for Pattern Recognition**. Oxford University Press, New York, 1995. ISBN 0–19–853864–2.

[BTW95] *P.J. Braspenning, F. Thuijsman, and A.J.M.M. Weijters*, Eds. **Artificial Neural Networks**. Springer–Verlag, Berlin, 1995. ISBN 3–540–59488–4.

[CG87a] *G. A. Carpenter and S. Grossberg*. **Art2: self–organization of stable category recognition codes for analog input patterns**. *Applied Optics*, 26(23):4919–4930, December 1987. ISSN 0003–6935.

[CG87b] *G. A. Carpenter and S. Grossberg*. **A massively parallel architecture for a self–organizing neural pattern recognition machine**. *Computer Vision, Graphics and Image Processing*, 37:54–115, 1987.

[FMTG93] *T. Fritsch, M. Mittler, and P. Tran-Gia*. **Artificial neural net applications in telecommunication systems**. *Neural Computing & Applications*, (1):124–146, 1993.

[FS92] *J. A. Freeman and D. M. Skapura*. **Neural Networks, Algorithms, Applications and Programming Techniques**. Addison–Wesley, New York, 2nd edition, 1992. ISBN 0–201–51376–5.

[GL96] *Gene H. Golub and Charles S. Van Loan*. **Matrix Computations**. Johns Hopkins University Press, Baltimore, 3rd edition, 1996. ISBN 0–8018–5414–8.

[Has95] *Mohamad H. Hassoun*. **Fundamentals of Artificial Neural Networks**. MIT Press, Cambridge, Massachusetts, 1995. ISBN 0–262–08239–X.

[Koh88] *Teuvo Kohonen*. **Self–Organization and Associative Memory**. Springer–Verlag, New York, 2nd edition, 1988. ISBN 0–201–51376–5.

[Kos92] *Bart Kosko*. **Neural Networks and Fuzzy Systems**. Prentice–Hall, London, 1992. ISBN 0–13–612334–1.

[Lor66] *G. G. Lorentz*. **Approximation of Functions**. Holt, Rinehart and Winston, New York, 1966.

[McC97] *Martin McCarthy*. **What is multi–threading?** *Linux Journal*, –(34):31–40, February 1997.

[MHP90] *Alianna J. Maren, Craig T. Harston, and Robert M. Pap*. **Handbook of Neural Computing Applications**. Academic Press, San Diego, California, 1990. ISBN 0–12–471260–6.

[Mos97] *David Mosberger*. **Linux and the alpha: How to make your application fly, part 2.** *Linux Journal*, –(43):68–75, November 1997.

[Ort87] *James M. Ortega*. **Matrix Theory**. Plenum Press, New York, 1987. ISBN 0–306–42433–9.

[Rip96] *Brian D. Ripley*. **Pattern Recognition and Neural Networks**. Cambridge University Press, New York, 1996. ISBN 0–521–46086–7.

Index