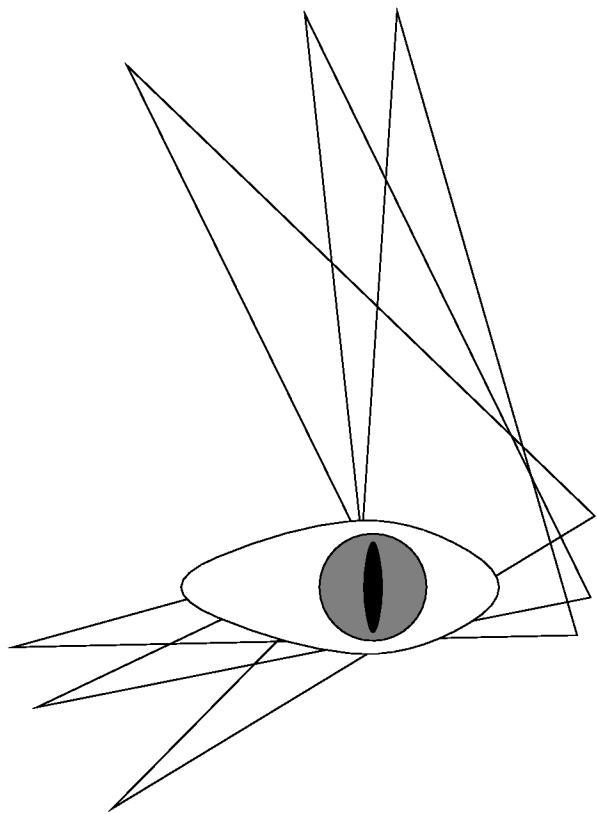


**R. M. Hristev**



**The ANN Book**

---

---

---

**The ANN Book — R. M. Hristev — Edition 1**  
(supercede the draft (edition 0) named “Artificial Neural Networks”)

Copyright © 1998 by R. M. Hristev

This book is released under the *GNU Public License, ver. 2* (the copyleft license). Basically  
“...the *GNU General Public License* is intended to guarantee your freedom to share and  
change free software—to make sure the software is free for all its users...”. It also means  
that, as is free of charge, there is **no warranty**.

---

# Preface

---

## ► About This Book

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.

However, to be able to (correctly) apply this 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.

**Please note that this book is released in electronic format ( $\text{\LaTeX}$ ) and under the GNU Public Licence (ver. 2) which allow for free copying and redistribution as long as you do not restrict the same rights for others. See the licence terms included in file "LICENCE.txt".** A freeware version of  $\text{\LaTeX}$  is available for almost any type of computer/OS combination and is practically guaranteed to produce the same high quality typesetting output. On Internet the URL where you can find the latest edition/version of this book is "<ftp://ftp.funet.fi/pub/sci/neural/books/>". Note that you may find two files there: one being this book in Postscript format and the other containing the source files, the source files contain the  $\text{\LaTeX}$  files as well as some additional programs — e.g. a program showing an animated learning process into a Kohonen network. The programs used in this book 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.

This book make an attempt to cover some of the basic ANN development: some theories, principles and ANN architectures which have found a way into the mainstream.

*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. This part would be of use as an introduction and for those who have to implement them but do not have to worry about their applications (e.g. programmers required to implement a particular ANN engine for some applications — but note that some important algorithmic improvements are explained in the second part).

*Second part* takes a deeper insight 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 math apparatus. It is important for those who want to develop applications using ANN. As ANN have been revealed to be statistical by their nature it requires some basic knowledge of statistical methods. An appendix containing a small introduction to statistics (very bare but essential for those who did not studied statistics) have been developed.

*Third part* is reserved to topics which are very recent developments and usually *open-ended*.

For each section (chapter, sub-section, e.t.c.) there is a special footnote designed in particular to refer some bibliographic information. 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). To avoid an ugly appearance they are hidden from the section’s title. Follow them for further references.

The appendix(es) contains also some information which have not been deemed appropriate for the main text (e.g. some useful mathematical results).

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

## ► Mathematical Notations and Conventions

❖ marginal note

The following notational system will be used (hopefully into a consistent manner) trough the whole book. There will be two kind of notations: one which will be described here and most of the time will *not* be explained in the text again; the other ones will be local (to the chapter, section, e.t.c.) 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.

So, when you encounter a symbol 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 till you find it and there should be 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 its skills in mathematical methods used in this field.

Do not worry about (fully) understanding all notations defined here, right now. Return here when the text will send you (automatically) back.

\* \* \*

ANN involves heavy manipulations of vectors and matrices. A vector will be often 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 transposed  $\mathbf{x}^T = (x_1 \quad \cdots \quad x_N)$  (for aesthetic reasons and readability). Also it will be represented by lowercase **bold** letters.

A scalar product between two vectors may be represented by a product between the corre-

spondent matrices:

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

The other matrices will be represented by uppercase letters. A inverse of a matrix will be marked by  $(\cdot)^{-1}$ .

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 is applied to each element in turn, i.e.

$$f(\mathbf{x}) = \begin{pmatrix} f(x_1) \\ \vdots \\ f(x_N) \end{pmatrix}, \quad f : \mathbb{R} \rightarrow \mathbb{R}$$

is a scalar function and application to a vector is just a convenient notation, while:

$$\mathbf{g}(\mathbf{x}) = \begin{pmatrix} g_1(\mathbf{x}) \\ \vdots \\ g_K(\mathbf{x}) \end{pmatrix}, \quad \mathbf{g} : \mathbb{R}^N \rightarrow \mathbb{R}^K$$

is a vectorial function and generally  $K \neq N$ . Note that bold letters are used for vectorial functions.

One operator which will be used is the ":". The  $A(i,:)$  notation will represent row  $i$  of matrix  $A$ , while  $A(:,j)$  will stand for column  $j$  of the same matrix.  $\diamond :$

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

$$A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \\ a_{k1} & \cdots & a_{kn} \end{pmatrix}, \quad B = \begin{pmatrix} b_{11} & \cdots & b_{1n} \\ \vdots & \ddots & \\ b_{k1} & \cdots & b_{kn} \end{pmatrix} \Rightarrow$$

$$C = A \odot B = \begin{pmatrix} a_{11}b_{11} & \cdots & a_{1n}b_{1n} \\ \vdots & \ddots & \\ a_{k1}b_{k1} & \cdots & a_{kn}b_{kn} \end{pmatrix}$$

ANN	acronym for Artificial Neural Network(s).
$\odot$	Hadamard product of matrices (see above for definition).
$(\cdot)^{\odot n}$	a convenient notation for $\underbrace{(\cdot) \odot \cdots \odot (\cdot)}_n \equiv (\cdot)^{\odot n}$
:	matrix “scissors” operator: $A(i:j, k:\ell)$ selects a submatrix from matrix $A$ , made from rows $i$ to $j$ and columns $k$ to $\ell$ . $A(i,:)$ represents row $i$ while $A(:,k)$ represents column $k$ .
$(\cdot)^T$	transposed of matrix $(\cdot)$ .

$(\cdot)^C$	complement of vector $(\cdot)$ . it involves swapping $0 \leftrightarrow 1$ for binary vectors and $-1 \leftrightarrow +1$ for bipolar vectors.
$ \cdot $	module of $(\cdot)$ (absolute value), when applied to a matrix is an <i>element-wise</i> operation (by contrast to the norm operation).
$\ \cdot\ $	norm of a vector or matrix — the actual definition may differ depending of the metric used.
$\langle \cdot \rangle$	mean value of a variable.
$\mathcal{E}\{f g\}$	expectation of event $f$ (mean value), given event $g$ . As $f$ and $g$ are usually functions then $\mathcal{E}\{f g\}$ is a functional.
$\mathcal{V}\{f\}$	variance of $f$ . As $f$ is usually a function then $\mathcal{V}\{f\}$ is a functional.
$\text{sign}(x)$	the sign function, defined as $\text{sign}(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x = 0. \\ -1 & \text{if } x < 0 \end{cases}$ In case of a matrix, it applies to each element individually.
$\tilde{\mathbf{1}}$	a <b>matrix</b> having <i>all</i> elements equal to 1, its dimensions will be <i>always</i> such that the mathematical operations in which is involved are correct.
$\hat{\mathbf{1}}$	a (column) <b>vector</b> having <i>all</i> elements equal to 1, its dimensions will be <i>always</i> such that the mathematical operations in which is involved are correct.
$\tilde{\mathbf{0}}$	a <b>matrix</b> having <i>all</i> elements equal to 0, its dimensions will be <i>always</i> such that the mathematical operations in which is involved are correct.
$\hat{\mathbf{0}}$	a (column) <b>vector</b> having <i>all</i> elements equal to 0, its dimensions will be <i>always</i> such that the mathematical operations in which is involved are correct.
$I$	the unit square matrix, assumed always to have the correct dimensions for the operations in which is involved.
$x_i$	component $i$ of the input vector.
$\mathbf{x}$	the input vector: $\mathbf{x}^T = (x_1 \ \dots \ x_N)$
$y_j$	output of the output neuron $j$ .
$\mathbf{y}$	the output vector of output layer: $\mathbf{y}^T = (y_1 \ \dots \ y_K)$
$z_k$	output of a hidden neuron $k$ .
$\mathbf{z}$	the output vector of a hidden layer: $\mathbf{z}^T = (z_1 \ \dots \ z_M)$
$t_k$	component $k$ of the target pattern.
$\mathbf{t}$	the target vector — desired output corresponding to input $\mathbf{x}$ .
$w_i, w_{ji}$	$w_i$ the weight associated with $i$ -th input of a neuron; $w_{ji}$ the weight associated with connection to neuron $j$ , from neuron $i$ .

$W$	the weight matrix $W = \begin{pmatrix} w_{11} & \cdots & w_{1N} \\ \vdots & \ddots & \vdots \\ w_{K1} & \cdots & w_{KN} \end{pmatrix}$ , note that all weights associated with a particular neuron $j$ ( $j = \overline{1, K}$ ) are on the same row.
$a_j$	total input to a neuron $j$ , the weighted sum of its inputs, e.g. $a_j = \sum_{\text{all } i} w_{ji}x_i$ — for a neuron receiving input $\mathbf{x}$ , $w_{ji}$ being the weights.
$\mathbf{a}$	the vector containing total inputs $a_j$ for all neurons in a same layer, usually $\mathbf{a} = W\mathbf{z}_{\text{prev.}}$ , where $\mathbf{z}_{\text{prev.}}$ is the output of previous layer.
$f$	activation function of the neuron; the neuron output is $f(a_j)$ and the output of current layer is $\mathbf{z} = f(\mathbf{a})$ .
$f'$	the derivative of the activation function $f$ .
$E$	the error function.
$\mathcal{C}_k$	class $k$ .
$P(\mathcal{C}_k)$	prior probability of a pattern $\mathbf{x}$ to belong to class $k$ .
$P(X_\ell)$	distribution probability of a pattern $\mathbf{x}$ to be in pattern subspace $X_\ell$ .
$P(\mathcal{C}_k, X_\ell)$	join probability of a pattern $\mathbf{x}$ to belong to class $k$ and pattern subspace $X_\ell$ .
$P(X_\ell   \mathcal{C}_k)$	class-conditional probability of a pattern $\mathbf{x}$ to belong to class $k$ to be in pattern subspace $X_\ell$ .
$P(\mathcal{C}_k   X_\ell)$	posterior probability of a pattern $\mathbf{x}$ to belong to class $k$ when is from subspace $X_\ell$ .
$p$	probability density.

Note also that wherever possible will try to reserve index  $i$  for input components,  $j$  for hidden neurons,  $k$  for output neurons and  $p$  for training patterns with  $P$  as the total number of (learning) patterns.

Ryurick M. Hristev



# Contents

---

<b>Preface</b>	i
About This Book . . . . .	i
Mathematical Notations and Conventions . . . . .	ii
<b>I ANN Architectures</b>	<b>1</b>
<b>1 Basic Neuronal Dynamics</b>	<b>3</b>
1.1 Simple Neurons and Networks . . . . .	3
1.2 Neurons as Functions . . . . .	5
1.3 Common Signal Functions . . . . .	6
<b>2 The Backpropagation Network</b>	<b>9</b>
2.1 Network Structure . . . . .	9
2.2 Network Dynamics . . . . .	10
2.2.1 Neuron Output Function . . . . .	10
2.2.2 Network Running Function . . . . .	11
2.2.3 Network Learning Function . . . . .	11
2.2.4 Initialization and Stop . . . . .	15
2.3 The Algorithm . . . . .	16
2.4 Bias . . . . .	18
2.5 Algorithm Enhancements . . . . .	20
2.5.1 Momentum . . . . .	20
2.5.2 Adaptive Backpropagation . . . . .	21
2.5.3 SuperSAB . . . . .	23
2.6 Applications . . . . .	24
2.6.1 Identity Mapping Network . . . . .	24
2.6.2 The Encoder . . . . .	26

<b>3 The SOM/Kohonen Network</b>	<b>29</b>
3.1 Network Structure . . . . .	29
3.2 Types of Neuronal Learning . . . . .	31
3.2.1 The Learning Process . . . . .	31
3.2.2 The Trivial Equation . . . . .	32
3.2.3 The Simple Equation . . . . .	32
3.2.4 The Riccati Equation . . . . .	33
3.2.5 More General Equations . . . . .	35
3.3 Network Dynamics . . . . .	39
3.3.1 Network Running Function . . . . .	39
3.3.2 Network learning function . . . . .	39
3.3.3 Initialization and Stop condition . . . . .	40
3.3.4 Remarks . . . . .	40
3.4 The algorithm . . . . .	40
3.5 Applications . . . . .	41
3.5.1 The Trivial Model with Forgetting Function . . . . .	41
3.5.2 Square mapping . . . . .	44
<b>4 The BAM/Hopfield Memory</b>	<b>47</b>
4.1 Associative Memory . . . . .	47
4.2 The BAM Architecture . . . . .	48
4.3 BAM Dynamics . . . . .	49
4.3.1 Network Running . . . . .	49
4.3.2 The BAM Energy Function . . . . .	51
4.4 The BAM Algorithm . . . . .	53
4.5 The Hopfield Memory . . . . .	54
4.5.1 The Discrete Hopfield Memory . . . . .	54
4.5.2 The Continuous Hopfield Memory . . . . .	57
4.6 Applications . . . . .	60
4.6.1 The Traveling Salesperson Problem . . . . .	60
<b>5 The Counterpropagation Network</b>	<b>67</b>
5.1 The CPN Architecture . . . . .	67
5.1.1 The Input Layer . . . . .	68
5.1.2 The Hidden Layer . . . . .	71
5.1.3 The Output Layer . . . . .	76

5.2 CPN Dynamics . . . . .	78
5.2.1 Network Running . . . . .	78
5.2.2 Network Learning . . . . .	79
5.3 The Algorithm . . . . .	81
5.4 Applications . . . . .	83
5.4.1 Letter classification . . . . .	83
<b>6 Adaptive Resonance Theory (ART)</b>	<b>85</b>
6.1 The ART1 Architecture . . . . .	85
6.2 ART1 Dynamics . . . . .	87
6.2.1 The $F_1$ layer . . . . .	87
6.2.2 The $F_2$ layer . . . . .	90
6.2.3 Learning on $F_1$ : The $W$ weights . . . . .	93
6.2.4 Learning on $F_2$ : The $W'$ weights . . . . .	94
6.2.5 Subpatterns . . . . .	96
6.2.6 The Reset Unit . . . . .	97
6.3 The ART1 Algorithm . . . . .	99
6.4 The ART2 Architecture . . . . .	100
6.5 ART2 Dynamics . . . . .	102
6.5.1 The $F_1$ layer . . . . .	102
6.5.2 The $F_2$ Layer . . . . .	103
6.5.3 The Reset Layer . . . . .	103
6.5.4 Learning and Initialization . . . . .	104
6.6 The ART2 Algorithm . . . . .	107
<b>II Basic Principles</b>	<b>109</b>
<b>7 Pattern Recognition</b>	<b>111</b>
7.1 Patterns: The Statistical Approach . . . . .	111
7.1.1 Patterns and Classification . . . . .	111
7.1.2 Feature Extraction . . . . .	113
7.1.3 Model Complexity . . . . .	114
7.1.4 Classification: Making Decisions and Minimizing Risk . . . . .	116
7.2 Likelihood Function . . . . .	120
7.2.1 The Discriminant Functions . . . . .	120
7.2.2 Likelihood Function and Maximum Likelihood Procedure . . . . .	121
7.3 Statistical Models . . . . .	125

<b>8 Single Layer Neural Networks</b>	<b>129</b>
8.1 Linear Separability . . . . .	129
8.1.1 Discriminant Functions . . . . .	129
8.1.2 Neuronal Memory Capacity . . . . .	132
8.1.3 Logistic discrimination . . . . .	134
8.1.4 Binary pattern vectors . . . . .	136
8.1.5 Generalized linear discriminants . . . . .	137
8.2 The Least Squares Technique . . . . .	137
8.2.1 The Error Function . . . . .	137
8.2.2 The Pseudo-inverse solution . . . . .	139
8.2.3 The Gradient Descent Solution . . . . .	142
8.3 The Perceptron . . . . .	144
8.3.1 The Error Function . . . . .	144
8.3.2 The Learning Procedure . . . . .	145
8.3.3 Convergence of Learning . . . . .	145
8.4 Fisher Linear Discriminant . . . . .	147
8.4.1 Two Classes Case . . . . .	147
8.4.2 Connections With The Least Squares Technique . . . . .	149
8.4.3 Multiple Classes Case . . . . .	151
<b>9 Multi Layer Neural Networks</b>	<b>153</b>
9.1 Feed-Forward Networks . . . . .	153
9.2 Threshold Neurons . . . . .	154
9.2.1 Binary Vectors . . . . .	155
9.2.2 Continuous Vectors . . . . .	155
9.3 Sigmoidal Neurons . . . . .	157
9.3.1 Three Layer Networks . . . . .	158
9.3.2 Two Layer Networks . . . . .	158
9.4 Weight-Space Symmetry . . . . .	160
9.5 Higher-Order Neuronal Networks . . . . .	161
9.6 Backpropagation Algorithm . . . . .	161
9.6.1 Error Backpropagation . . . . .	161
9.6.2 Application: Sigmoidal Neurons and Sum-of-squares Error . . . . .	163
9.7 Jacobian Matrix . . . . .	164
9.8 Hessian Tensor . . . . .	166
9.8.1 Diagonal Approximation . . . . .	166

9.8.2 Outer Product Approximation . . . . .	167
9.8.3 Inverse Hessian . . . . .	167
9.8.4 Finite Differences . . . . .	168
9.8.5 Exact Hessian . . . . .	169
9.8.6 Multiplication with Hessian . . . . .	170
<b>10 Radial Basis Function Networks</b>	<b>173</b>
10.1 Exact Interpolation . . . . .	173
10.2 Radial Basis Function Networks . . . . .	174
10.3 Relation to Other Theories . . . . .	175
10.3.1 Relation to Regularization Theory . . . . .	175
10.3.2 Relation to Interpolation Theory . . . . .	177
10.3.3 Relation to Kernel Based Method . . . . .	178
10.4 Classification . . . . .	178
10.5 Network Learning . . . . .	180
10.5.1 Radial Basis Functions . . . . .	180
10.5.2 Output Layer Weights . . . . .	182
<b>11 Error Functions</b>	<b>185</b>
11.1 Generalities . . . . .	185
11.2 Sum-of-Squares Error . . . . .	186
11.2.1 Linear Output Units . . . . .	187
11.2.2 Linear Sum Rules . . . . .	188
11.2.3 Significance of Network Output . . . . .	189
11.2.4 Outer product approximation of Hessian . . . . .	191
11.3 Minkowski Error . . . . .	192
11.4 Input-dependent Variance . . . . .	193
11.5 Modeling Conditional Distributions . . . . .	194
11.6 Classification using Sum-of-Squares . . . . .	197
11.6.1 Hidden Neurons . . . . .	198
11.6.2 Weighted Sum-of-Squares . . . . .	199
11.6.3 Loss Matrix . . . . .	201
11.7 Cross Entropy . . . . .	202
11.7.1 Two Classes Case . . . . .	202
11.7.2 Sigmoidal Activation Functions . . . . .	203
11.7.3 Cross-Entropy Properties . . . . .	203

11.7.4 Multiple Independent Features . . . . .	204
11.7.5 Multiple Classes Case . . . . .	205
11.8 Entropy . . . . .	207
11.9 Outputs as Probabilities . . . . .	212
<b>12 Parameter Optimization</b>	<b>215</b>
12.1 Error Surfaces . . . . .	215
12.2 Local Quadratic Approximation . . . . .	216
12.3 Initialization and Termination of Learning . . . . .	217
12.4 Gradient Descent . . . . .	218
12.4.1 Learning Parameter and Convergence . . . . .	220
12.4.2 Momentum . . . . .	221
12.4.3 Other Gradient Descent Improvement Techniques . . . . .	223
12.5 Line Search . . . . .	225
12.6 Conjugate Gradients . . . . .	225
12.6.1 Conjugate Search Directions . . . . .	225
12.6.2 Quadratic Error Function . . . . .	226
12.6.3 The Algorithm . . . . .	229
12.6.4 Scaled Conjugated Gradients . . . . .	231
12.7 Newton's Method . . . . .	233
12.8 Levenberg-Marquardt Algorithm . . . . .	234
<b>13 Feature Extraction</b>	<b>237</b>
13.1 Pre/Post-processing . . . . .	237
13.2 Input Normalization . . . . .	238
13.3 Missing Data . . . . .	239
13.4 Time Series Prediction . . . . .	240
13.5 Feature Selection . . . . .	241
13.6 Dimensionality Reduction . . . . .	243
13.6.1 Principal Component Analysis . . . . .	243
13.6.2 Non-linear Dimensionality Reduction Trough ANN . . . . .	245
13.7 Invariance . . . . .	247
13.7.1 The Tangent Prop Method . . . . .	247
13.7.2 Preprocessing . . . . .	248
13.7.3 Shared Weights . . . . .	250
13.7.4 Higher-order ANNs . . . . .	250

<b>14 Learning Optimization</b>	<b>253</b>
14.1 The Bias-Variance Tradeoff . . . . .	253
14.2 Regularization . . . . .	255
14.2.1 Weight Decay . . . . .	255
14.2.2 Linear Transformation And Weight Decay . . . . .	257
14.2.3 Early Stopping . . . . .	258
14.2.4 Curvature Smoothing . . . . .	259
14.2.5 Choosing weight decay hyperparameter . . . . .	259
14.3 Adding Noise . . . . .	260
14.4 Soft Weight Sharing . . . . .	261
14.5 Growing And Pruning Methods . . . . .	264
14.5.1 Cascade Correlation . . . . .	264
14.5.2 Pruning Techniques . . . . .	266
14.5.3 Neuron Pruning . . . . .	268
14.6 Committees of Networks . . . . .	268
14.7 Mixture Of Experts . . . . .	271
14.8 Other Training Techniques . . . . .	272
14.8.1 Cross-validation . . . . .	272
14.8.2 Stacked Generalization . . . . .	273
14.8.3 Complexity Criteria . . . . .	273
14.8.4 Model For Mixed Discrete And Continuous Data . . . . .	274
<b>15 Bayesian Techniques</b>	<b>275</b>
15.1 Bayesian Learning . . . . .	275
15.1.1 Weight Distribution . . . . .	275
15.1.2 Gaussian Prior Weight Distribution . . . . .	276
15.1.3 Application — Simple Classifier . . . . .	276
15.1.4 Gaussian Noise Model . . . . .	279
15.1.5 Gaussian Posterior Weight Distribution . . . . .	280
15.1.6 Consistent Prior Weight Distribution . . . . .	280
15.1.7 Approximation Of Weight Distribution . . . . .	280
15.2 Network Outputs Distribution . . . . .	281
15.2.1 Generalized Linear Networks . . . . .	283
15.3 Classification . . . . .	284
15.4 The Evidence Approximation For $\alpha$ And $\beta$ . . . . .	287
15.5 Integration Over $\alpha$ And $\beta$ . . . . .	291

15.6 Model Comparison . . . . .	292
15.7 Committee Of Networks . . . . .	294
15.8 Monte Carlo Integration . . . . .	295
15.9 Minimum Description Length . . . . .	297
15.10 Performance Of Models . . . . .	298
15.10.1 Risk Averaging . . . . .	298
<b>16 Tree Based Classifiers</b>	<b>301</b>
16.1 Tree Classifiers . . . . .	301
16.2 Splitting . . . . .	302
16.2.1 Impurity based method . . . . .	302
16.2.2 Deviance based method . . . . .	303
16.3 Pruning . . . . .	304
16.4 Missing Data . . . . .	306
<b>17 Belief Networks</b>	<b>307</b>
17.1 Graphs . . . . .	307
17.1.1 Markov Properties . . . . .	308
17.1.2 Markov Trees . . . . .	310
17.1.3 Decomposable Trees . . . . .	312
17.2 Casual Networks . . . . .	313
17.3 The Boltzmann Machine . . . . .	314
<b>III Advanced Topics</b>	<b>317</b>
<b>18 Matrix Operations on ANN</b>	<b>319</b>
18.1 New Matrix Operations . . . . .	319
18.2 Algorithms . . . . .	320
18.2.1 Backpropagation . . . . .	320
18.2.2 SOM/Kohonen Networks . . . . .	321
18.2.3 BAM/Hopfield Networks . . . . .	322
18.3 Conclusions . . . . .	323
<b>A Mathematical Sidelines</b>	<b>325</b>
A.1 Distances . . . . .	325
A.1.1 Euclidean Distance . . . . .	325
A.1.2 Hamming Distance . . . . .	325

A.2	Generalized Spherical Coordinates . . . . .	326
A.3	Properties of Symmetric Matrices . . . . .	327
A.3.1	Eigenvectors and Eigenvalues . . . . .	327
A.3.2	Rotation . . . . .	329
A.3.3	Quadratic Forms . . . . .	329
A.4	The Gaussian Integrals . . . . .	329
A.4.1	The Unidimensional Case . . . . .	329
A.4.2	The Multidimensional Case . . . . .	330
A.4.3	The multidimensional Gaussian integral with a linear term . . . . .	331
A.5	The Euler Functions . . . . .	331
A.5.1	The Euler function . . . . .	331
A.5.2	The sphere volume in the n-dimensional space . . . . .	332
A.6	The Lagrange Multipliers . . . . .	333
A.7	Useful Mathematical equations . . . . .	334
A.7.1	Combinatorics . . . . .	334
A.7.2	The Jensen's inequality . . . . .	334
A.7.3	The Stirling Formula . . . . .	336
A.8	Calculus of Variations . . . . .	337
A.9	Principal Components . . . . .	338
<b>B</b>	<b>Statistical Sidelines</b>	<b>341</b>
B.1	Probabilities . . . . .	341
B.1.1	Probabilities and Bayes Theorem . . . . .	341
B.1.2	Probability Density, Expectation and Variance . . . . .	343
B.2	Modeling the Density of Probability . . . . .	344
B.2.1	The Parametric Method . . . . .	345
B.2.2	The non-parametric method . . . . .	350
B.2.3	The Semi-Parametric Method . . . . .	354
B.3	The Bayesian Inference . . . . .	361
B.4	The Robbins–Monro algorithm . . . . .	364
B.5	Learning vector quantization . . . . .	366
<b>Bibliography</b>		<b>369</b>
<b>Index</b>		<b>371</b>



# **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 for natural brains. Much later it was discovered that ANN are a very general statistical<sup>1</sup> framework for modelling posterior probabilities given a set of samples (the input data).

The basic building block of a (artificial) neural network (ANN) is the neuron. A neuron is a processing unit which have 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  $\sum_{\text{all inputs}} w_i x_i = a$ . Then a activation function  $f$  is applied to the result  $a$ . The neuronal output is taken to be  $f(a)$ .

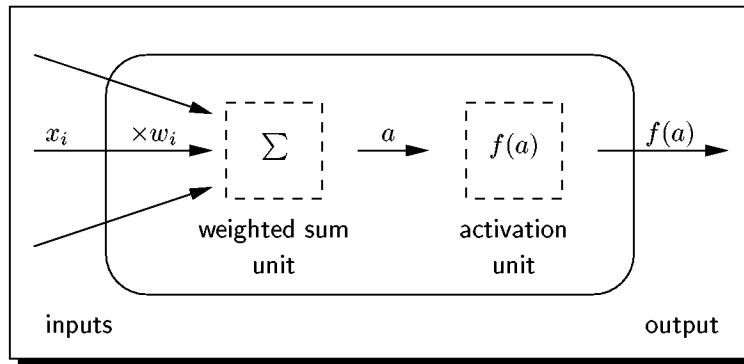
Generally the ANN are build by putting the neurons in layers and connecting the outputs of neurons from one layer to the inputs of the neurons from the next layer. See figure 1.2 on the next page. The type of network depicted there is also named *feedforward* (a feedforward network do not have feedbacks, 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 on neuron from one layer are going to the inputs of neurons from previous layers then the network is called *recurrent*, this providing feedback; lateral

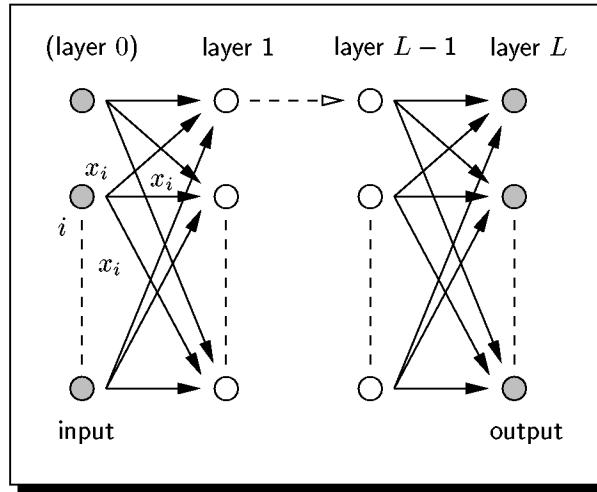
---

<sup>1</sup>\* For more information see also [BB95], it provides with some detailed theoretical neuronal models for true neurons.

<sup>1</sup>In the second part of this book it is explained in greater detail how the ANN output have a statistical significance.



**Figure 1.1:** The neuron



**Figure 1.2:** The general layout of a (feedforward) neural network. 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).

feedback is done when the output of one neuron goes to the other neurons on the same layer<sup>2</sup>.

So, to compute the output, an “activation function” is applied on the weighted sum of inputs:

$$\text{total input} \equiv a = \sum_{\text{all inputs}} w_i \cdot x_i$$

$$\text{output} = \text{activation function} \left( \sum_{\text{all inputs}} w_i \cdot x_i \right) = f(a)$$

<sup>2</sup>This is used into the SOM/Kohonen architecture.

More general designs are possible, e.g. higher order ANNs where the total input to a neurons contains also higher order combinations between inputs (e.g. 2-nd order terms, of the form  $w_{ij}x_i x_j$ ); however these are seldom used in practice as it involves huge computational efforts without clear-cut benefits.

The tunable parameters of an ANN are the weights  $\{w_i\}$ . They are found by different mathematical procedures<sup>3</sup> by using a given set of data. The procedure of finding the weights is named *learning* or *training*. The data set is called *learning* or *training set* and contain pairs of input vectors associated with the *desired* output vectors:  $\{(x_i, y_i)\}$ . 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 being *supervised*).



### 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 which neurons receive each one single component of the total input and distribute it to all neurons of the input layer. This layer may be seen as a **sensor layer** and (usually) doesn't do any (real) processing. In some other architectures the input layer is also the sensor layer. Unless otherwise specified it will be omitted.
- The last layer is called the **output layer**. The output set of the output neurons is (commonly) the desired output (of the network).
- The layers between input and output are called **hidden layers**.

## 1.2 Neurons as Functions

Neurons behave as functions. Neurons transduce an unbounded input activation  $x(t)$  at a time  $t$  into a bounded output **signal**  $f(x(t))$ . Usually a sigmoidal or S-shaped curve, as in figure 1.3 on the next page describes the transduction. This function ( $f$ ) is called the **activation** or **signal function**.

The most used function is the *logistic* signal function:

$$f(a) = \frac{1}{1 + e^{-ca}}$$

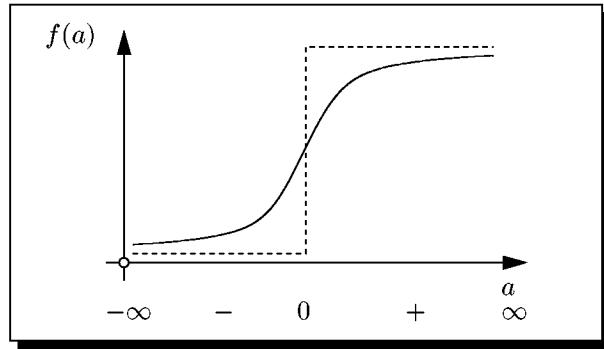
which is sigmoidal and strictly increases for positive scaling constant  $c > 0$ . Strict monotonicity implies that the activation derivative of  $f$  is positive:  $\diamond c$

$$f' \equiv \frac{df}{da} = cf(1 - f) > 0$$

The threshold signal function (dashed line) in figure 1.3 on the following page illustrates a non-differentiable signal function. The family of logistic signal function, indexed by  $c$ , approaches asymptotically the threshold function as  $c \rightarrow +\infty$ . Then  $f$  transduce positive activations signals  $a$  to unity signals and negative activations to zero signals. A discontinuity

---

<sup>3</sup>Most usual is the gradient-descent method and derivatives.



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

occurs at the zero activation value (which equals the signal function's "threshold"). Zero activation values seldom occur in large neural networks<sup>4</sup>.

❖  $\dot{f}$

The *signal velocity*  $df/dt$ , denoted  $\dot{f}$ , measures the signal's instantaneous time change.  $\dot{f}$  is the product between the change in the signal due to the activation and the change in the activation with time:

$$\dot{f} = \frac{df}{da} \frac{da}{dt} = f' \dot{a}$$

## ► 1.3 Common Signal Functions

The following activation functions are more often encountered in practice:

### 1. Logistic:

$$f(a) = \frac{1}{1 + e^{-ca}}$$

where  $c > 0$ ,  $c = \text{const.}$  is a positive scaling constant. The activation derivative is  $f' = \frac{df}{da} = cf(1 - f)$  and so  $f$  is monotone increasing ( $f > 0$ ). This function is the most common one.

### 2. Hyperbolic-tangent:

$$f(a) = \tanh(ca) = \frac{e^{ca} - e^{-ca}}{e^{ca} + e^{-ca}}$$

where  $c > 0$ ,  $c = \text{const.}$  is a positive scaling constant. The activation derivative is  $f' = \frac{df}{da} = c(1 - f^2) > 0$  and so  $f$  is monotone increasing ( $f < 1$ ).

<sup>4</sup>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.

**3. Threshold:**

$$f(a) = \begin{cases} 1 & \text{if } a \geq \frac{1}{c} \\ 0 & \text{if } a < 0 \\ ca & \text{otherwise } (x \in [0, 1/c]) \end{cases}$$

where  $c > 0$ ,  $c = \text{const.}$  is a positive scaling constant. The activation derivative is:

$$f'(a) = \frac{df}{da} = \begin{cases} 0 & \text{if } a \in (-\infty, 0) \cup [1/c, \infty) \\ c & \text{otherwise} \end{cases}$$

Note that is not a true threshold function as it have a non-infinite slope between 0 and  $c$ .

**4. Exponential-distribution:**

$$f(a) = \max(0, 1 - e^{-ca})$$

where  $c > 0$ ,  $c = \text{const.}$  is a positive scaling constant. The activation derivative is:

$$f'(a) = \frac{df}{da} = ce^{-ca}$$

and for  $a > 0$ , supra-threshold signals are monotone increasing ( $f' > 0$ ). Note: since the second derivative  $f'' = -c^2e^{-ca}$  the exponential-distribution function is strictly convex.

**5. Ratio-polynomial:**

$$f(a) = \max\left(0, \frac{a^n}{c + a^n}\right) \quad \text{for } n > 1$$

where  $c > 0$ ,  $c = \text{const.}$ . The activation derivative is:

$$f' = \frac{df}{da} = \frac{cna^{n-1}}{(c + a^n)^2}$$

and for positive activation supra-threshold signals are monotone increasing.

**6. Pulse-coded:** 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).

The exponentially weighted time average of sampled binary pulses function is:

$$f(t) = \int_{-\infty}^t g(s) e^{s-t} ds$$

$t$  being time, where the function  $g$  is:

$$g(t) = \begin{cases} 1 & \text{if a pulse occurs at } t \\ 0 & \text{if no pulse at } t \end{cases}$$

❖  $t, g$

and equals one if a pulse arrives at time  $t$  or zero if no pulse arrives.

The pulse-coded signal function is:  $f(t) : [0, 1] \rightarrow [0, 1]$ .

*Proof.* If  $g(t) = 0, \forall t$  then  $f(t) = 0$  (trivial).

If  $g(t) = 1, \forall t$  then

$$f(t) = \int_{-\infty}^t e^{s-t} ds = e^{t-t} - \lim_{s \rightarrow -\infty} e^{s-t} = 1$$

When the number of arriving pulses increase then the “pulse count” can only increase so  $f$  is monotone nondecreasing.  $\square$

## CHAPTER 2

# The Backpropagation Network

The backpropagation network represents one of the most classical example of an ANN, being also one of the most simple in terms of the overall design.

## 2.1 Network Structure

The network consists of several layers of neurons. The first one (let it be layer 0) distributes the inputs to the input layer 1. There is no processing in layer 0, it can be seen just as a *sensory* layer — each neuron receive just one component of the input (vector)  $x$  which gets distributed, unchanged, to all neurons from the input layer. The last layer is the output layer which outputs the processed data; each output of individual output neurons being a component of the output vector  $y$ . The layers between the input one and the output one are hidden layers.

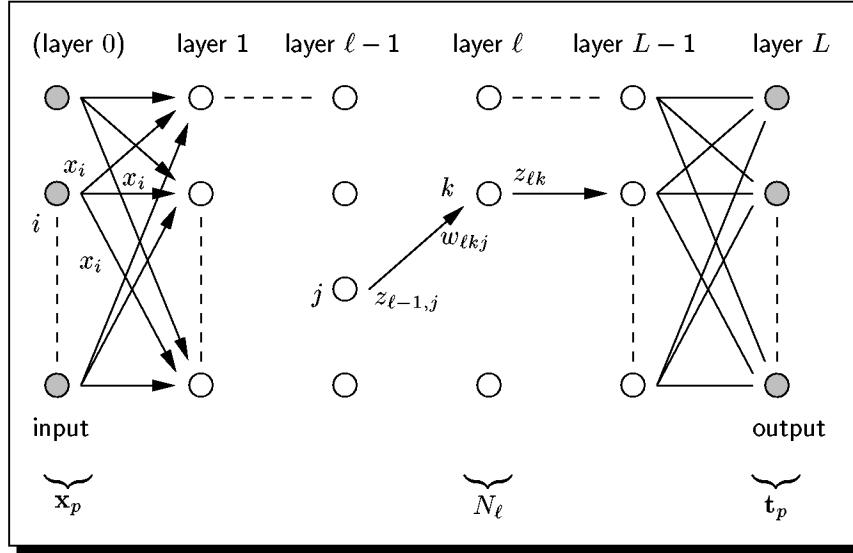


### Remarks:

- Layer 0 have to have the same number of neurons as the number of input components (dimension of input vector  $x$ ).
- The output layer have to have the same number of neurons as the desired output have (i.e. the dimension of the output vector  $y$  dictates the number of neurons on the output layer).
- In general the input and hidden layers may have any number of neurons, however their number may be chosen to achieve some special effects in some practical cases.

The network is a straight feedforward network: each neuron receives as input the outputs

feedforward  
network



**Figure 2.1:** The backpropagation network structure.

of all neurons from the previous layer (excepting the first sensory layer). See figure 2.1.

❖  $z_{\ell k}$ ,  $w_{\ell k j}$ ,  $\mathbf{x}_p$ ,  
 $t_p(\mathbf{x}_p)$ ,  $z_{0i}$ ,  $N_\ell$ ,  $L$ ,  
 $P$

The following notations are used:

- $z_{\ell k}$  is the output of neuron  $j$  from layer  $\ell$ .
- $w_{\ell k j}$  is the weight by which output of neuron  $j$  from layer  $\ell - 1$  contribute to input of neuron  $k$  from layer  $\ell$ .
- $\mathbf{x}_p$  is training input vector no.  $p$ .
- $t_p(\mathbf{x}_p)$  is the target (desired output) vector no.  $p$  (at training time).
- $z_{0i}$  is the  $i$  component of input vector. By notation, at training time,  $z_{0i} \equiv x_i$ , where  $x_i$  is the component  $i$  of one of input vectors, for some  $p$ .
- $N_\ell$  is the number of neurons in layer  $\ell$ .
- $L$  is the number of layers (the input layer is no. 0, the output layer is no.  $L$ ).
- $P$  is the number of training vectors,  $p = \overline{1, P}$

The learning set is (according to the above notation)  $\{(\mathbf{x}_p, \mathbf{t}_p)\}_{p=\overline{1, P}}$ .

## 2.2 Network Dynamics

### 2.2.1 Neuron Output Function

The activation function used is usually the logistic:

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

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

but note that the backpropagation algorithm is not particularly tied up to it.

### 2.2.2 Network Running Function

Each neuron compute at output the weighted sum of its input to which it applies the signal function (see also figure 2.1 on the facing page):

$$z_{\ell k} = f \left( \sum_{j=1}^{N_{\ell-1}} w_{\ell k j} z_{\ell-1, j} \right) \quad (2.3)$$

It must be computed in succession for each layer, starting from input and going through all layer in succession till the output layer is reached.

A more compact matrix notation may be developed as follows. For each layer (except 0), a matrix of weights  $W_\ell$  is build as:

$$W_\ell = \begin{pmatrix} w_{\ell 11} & \cdots & w_{\ell 1 N_{\ell-1}} \\ \vdots & \ddots & \vdots \\ w_{\ell N_\ell 1} & \cdots & w_{\ell N_\ell N_{\ell-1}} \end{pmatrix}$$

(note that all weights associated with a particular neuron are on the same row) then, considering the output vector of the previous layer  $\mathbf{z}_{\ell-1}$

$$\mathbf{z}_{\ell-1}^T = (z_{\ell-1,1} \ \cdots \ z_{\ell-1,N_{\ell-1}})$$

the output of the actual layer  $\ell$  may be calculated as:

$$\mathbf{z}_\ell^T = f(\mathbf{a}_\ell^T) = (f(a_{\ell 1}) \ \cdots \ f(a_{\ell N_\ell}))$$

where  $\mathbf{a}_\ell = W_\ell \mathbf{z}_{\ell-1}$ .

$\diamond W_\ell$

$\diamond \mathbf{z}_\ell$

$\diamond \mathbf{a}_\ell$

### 2.2.3 Network Learning Function

The network learning process is supervised i.e. the network receives (at training phase) both the raw data as inputs and the targets as output. *The learning involves adjusting weights so that errors will be minimized.* The function used to measure errors is usually the sum-of-squares defined below but note that backpropagation algorithm is not particularly tied up to it.

**Definition 2.2.1.** For an input pattern  $\mathbf{x}$  and the associated target  $\mathbf{t}$ , the **sum-of-squares error function**  $E(W)$  ( $E$  is dependent on all weights  $W$ ) is defined as:

$$E(W) \equiv \frac{1}{2} \sum_{q=1}^{N_L} [z_{Lq}(\mathbf{x}) - t_q(\mathbf{x})]^2$$

where  $z_{Lq}$  is the output of neuron  $q$  from the output layer i.e. the component  $q$  of the output vector.

$\diamond z_{Lq}$

Note that *all* components of input vector will influence *any* component of output vector, thus  $z_{Lq} = z_{Lq}(\mathbf{x})$ .

**Remarks:**❖  $E_{\text{tot.}}$ 

- Considering all learning samples (the full training set) then the total *sum-of-squares error sum-of-squares error*  $E_{\text{tot.}}(W)$  ( $E_{\text{tot.}}$  is also dependent of all weights as  $E$ ) is defined as:

$$E_{\text{tot.}}(W) \equiv \frac{1}{2} \sum_{p=1}^P E(W) = \sum_{p=1}^P \sum_{q=1}^{N_L} [z_{Lq}(\mathbf{x}_p) - t_q(\mathbf{x}_p)]^2 \quad (2.4)$$

The network weights are found (weights are adapted/changed) step by step. Considering  $N_W$  the total number of weights then the error function  $E : \mathbb{R}^{N_W} \rightarrow \mathbb{R}$  may be represented

as a surface in the  $\mathbb{R}^{N_W+1}$  space. The gradient vector  $\nabla E = \left\{ \frac{\partial E(W)}{\partial w_{\ell ji}} \right\}$  shows the direction of (local) maximum of square mean error and  $\{w_{\ell ji}\}$  are to be changed in the opposite direction (so “-” have to be used) — see also figure 2.3 on the next page.

❖  $t$ 

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

$$w_{\ell ji}(t+1) = w_{\ell ji}(t) - \mu \frac{\partial E(W)}{\partial w_{\ell ji}} \Big|_{W(t)} = w_{\ell ji}(t) - \mu \sum_{p=1}^P \frac{\partial E_p(W)}{\partial w_{\ell ji}} \Big|_{W(t)}$$

❖  $\mu$ 

where  $\mu = \text{const.}$ ,  $\mu > 0$  is named the learning constant and it is used for tuning the speed and quality of the learning process.

In matrix notation the above equation may be written simply as:

$$W(t+1) = W(t) - \mu \nabla E \quad (2.5)$$

because the error gradient may also be considered as a matrix or tensor, it have *the same dimensions* as  $W$ .

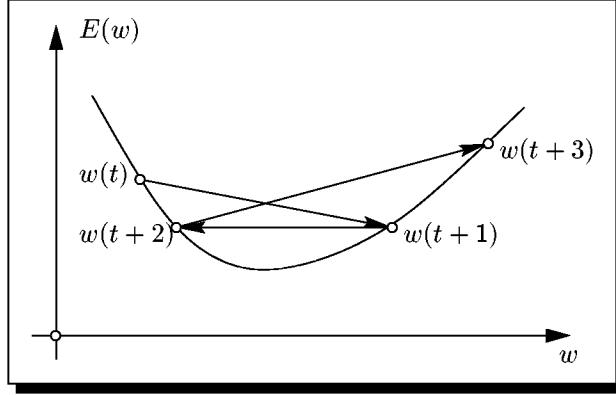
**Remarks:**

delta rule

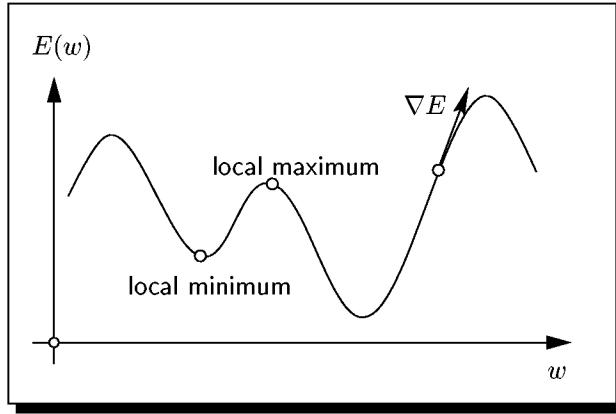
- The above method does not provide for a starting point. In practice, weights are initialized with small random values (usually in the  $[-1, 1]$  interval).
- The (2.5) equation represents the basics of weight adjusting, i.e. learning, in many ANN architectures; it is known as the **delta rule**:

$$\Delta W = W(t+1) - W(t) \propto -\nabla E$$

- If  $\mu$  is too small then the learning is slow and it may stop the learning process into a *local* minimum (of  $E$ ), being unable to overtake a *local* maximum. See figure 2.3 on the facing page.
- If  $\mu$  is too large then the learning is fast but it may jump over *local* minimum (of  $E$ ) which may be deeper than the next one. See figure 2.3 on the next page (Note that in general that is a **surface**).
- 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



**Figure 2.2:** Oscillations in learning process. The weights move around  $E$  minima without being able to reach it (arrows show the jumps made by the learning process).



**Figure 2.3:**  $E(w)$  — Total square error as function of weights.  $\nabla E$  points towards the (local) maximum.

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 2.2. This problem is particularly likely to occur for deep and narrow minima because in this case  $\nabla E$  is large (deep  $\equiv$  steep) and subsequently  $\Delta W$  is large (narrow  $\equiv$  easy to overshoot).

The problem is to find the error gradient  $\nabla E$ . Considering the “standard” approach (i.e.  $\{\nabla E\}_{\ell j i} \approx \frac{\Delta E}{\Delta w_{\ell j i}}$  for some small  $\Delta w_{\ell j i}$ ) this would require an computational time of the order  $\mathcal{O}(N_W^2)$ , because each calculation of  $E$  require  $\mathcal{O}(N_W)$  and it have to be repeated for each  $w_{\ell j i}$  in turn.

The importance of the backpropagation algorithm resides in the fact that it reduces the computational time of  $\nabla E$  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:

backpropagation  
❖  $(\nabla E)_\ell$

$$(\nabla E)_\ell \equiv \begin{pmatrix} \frac{\partial E}{\partial w_{\ell 11}} & \cdots & \frac{\partial E}{\partial w_{\ell 1 N_{\ell-1}}} \\ \vdots & \ddots & \vdots \\ \frac{\partial E}{\partial w_{\ell N_\ell 1}} & \cdots & \frac{\partial E}{\partial w_{\ell N_\ell N_{\ell-1}}} \end{pmatrix}, \quad \ell = \overline{1, L}$$

❖  $\nabla_{\mathbf{z}_\ell} E$   
For each layer, except  $L$  the error gradient, with respect to neuronal outputs, may be defined as:

$$\nabla_{\mathbf{z}_\ell} E \equiv \left( \frac{\partial E}{\partial z_{\ell 1}} \quad \cdots \quad \frac{\partial E}{\partial z_{\ell N_\ell}} \right), \quad \ell = \overline{1, L-1}$$

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  $\{\mathbf{t}_p\}$ .

$$\nabla_{\mathbf{z}_L} E = \text{known.}$$

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

$$\nabla_{\mathbf{z}_l} E = W_{l+1}^T \cdot [\nabla_{\mathbf{z}_{l+1}} E \odot f'(\mathbf{a}_{l+1})] \quad \text{calculated recursively from } L-1 \text{ to } 1 \quad (2.6a)$$

$$(\nabla E)_\ell = [\nabla_{\mathbf{z}_\ell} E \odot f'(\mathbf{a}_\ell)] \cdot \mathbf{z}_{\ell-1}^T \quad \text{for layers } \ell = \overline{1, L} \quad (2.6b)$$

❖  $\mathbf{z}_0 \equiv \mathbf{x}$ .

*Proof.* The error  $E(W)$  is dependent on  $w_{\ell j i}$  through the output of neuron  $(j, i)$  i.e.  $z_{\ell j}$ :

$$\frac{\partial E}{\partial w_{\ell j i}} = \frac{\partial E}{\partial z_{\ell j}} \frac{\partial z_{\ell j}}{\partial w_{\ell j i}}$$

and each derivative is computed separately

$$\begin{aligned} \square \text{ term } \frac{\partial z_{\ell j}}{\partial w_{\ell j i}} \\ \frac{\partial z_{\ell j}}{\partial w_{\ell j i}} &= \frac{\partial}{\partial w_{\ell j i}} \left[ f \left( \sum_{m=1}^{N_{\ell-1}} w_{\ell j m} z_{\ell-1, m} \right) \right] = \\ &= f' \left( \sum_{m=1}^{N_{\ell-1}} w_{\ell j m} z_{\ell-1, m} \right) \cdot \frac{\partial}{\partial w_{\ell j i}} \left( \sum_{m=1}^{N_{\ell-1}} w_{\ell j m} z_{\ell-1, m} \right) = \\ &= f'(a_{\ell j}) \cdot z_{\ell-1, i} \end{aligned}$$

because weights are mutually independent.

$$\square \text{ term } \frac{\partial E}{\partial z_{\ell j}}$$

Neuron  $z_{\ell j}$  affect  $E$  through all following layers that are intermediate between layer  $\ell$  and output (the influence being exercised through the interposed neurons).

First affected is next,  $\ell + 1$ , layer, through term  $\frac{\partial z_{\ell+1, m}}{\partial z_{\ell j}}$  (and then the dependency is carried on next successive layers):

$$\frac{\partial E}{\partial z_{\ell j}} = \sum_{m=1}^{N_{\ell+1}} \frac{\partial E}{\partial z_{\ell+1, m}} \frac{\partial z_{\ell+1, m}}{\partial z_{\ell j}} =$$

$$\begin{aligned}
&= \sum_{m=1}^{N_{\ell+1}} \frac{\partial E}{\partial z_{\ell+1,m}} \cdot \frac{\partial}{\partial z_{\ell j}} \left[ f \left( \sum_{q=1}^{N_\ell} w_{\ell+1,mq} z_{\ell q} \right) \right] = \\
&= \sum_{m=1}^{N_{\ell+1}} \frac{\partial E}{\partial z_{\ell+1,m}} \cdot f' \left( \sum_{q=1}^{N_\ell} w_{\ell+1,mq} z_{\ell q} \right) \cdot \frac{\partial}{\partial z_{\ell j}} \left( \sum_{q=1}^{N_\ell} w_{\ell+1,mq} z_{\ell q} \right) = \\
&= \sum_{m=1}^{N_{\ell+1}} \frac{\partial E}{\partial z_{\ell+1,m}} f'(a_{\ell+1,m}) w_{\ell+1,mj}
\end{aligned}$$

which represents exactly the element  $j$  of column matrix  $\nabla_{\mathbf{z}_\ell} E$  as build from (2.6a). The above formula applies iteratively from layer  $L - 1$  to 1, for layer  $L$ ,  $\nabla_{\mathbf{z}_L} E$  is assumed known.

Finally, the desired derivative is:

$$\frac{\partial E}{\partial w_{\ell j i}} = \frac{\partial E}{\partial z_{\ell j}} \frac{\partial z_{\ell j}}{\partial w_{\ell j i}} = \left[ \sum_{m=1}^{N_{\ell+1}} \frac{\partial E}{\partial z_{\ell+1,m}} f'(a_{\ell+1,m}) w_{\ell+1,mj} \right] f'(a_{\ell j}) z_{\ell-1,i}$$

representing the element found at row  $j$ , column  $i$  of matrix  $(\nabla E)_\ell$  as build from (2.6b).  $\square$

**Proposition 2.2.1.** *If using the logistic activation function and sum-of-squares error function then the error gradient may be computed recursively according to the formulas:*

$$\nabla_{\mathbf{z}_L} E = \mathbf{z}_L(\mathbf{x}) - \mathbf{t} \quad (2.7a)$$

$$\nabla_{\mathbf{z}_\ell} E = c W_{\ell+1}^T \cdot \left[ \nabla_{\mathbf{z}_{\ell+1}} E \odot \mathbf{z}_{\ell+1} \odot (\hat{\mathbf{1}} - \mathbf{z}_{\ell+1}) \right] \quad \text{for } \ell = \overline{1, L-1} \quad (2.7b)$$

$$(\nabla E)_\ell = c \left[ \nabla_{\mathbf{z}_\ell} E \odot \mathbf{z}_\ell \odot (\hat{\mathbf{1}} - \mathbf{z}_\ell) \right] \cdot \mathbf{z}_{\ell-1}^T \quad \text{for } \ell = \overline{1, L} \quad (2.7c)$$

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

*Proof.* From definition 2.2.1:

$$\frac{\partial E}{\partial z_{Lj}} = z_{Lj} - t_j \Rightarrow \nabla_{\mathbf{z}_L} E = \mathbf{z}_L - \mathbf{t}$$

By using (2.2) in the main results (2.6a) and (2.6b) of theorem 2.2.1, and considering that  $f(\mathbf{a}_\ell) = \mathbf{z}_\ell$  the other two formulas are deducted immediately.  $\square$

## 2.2.4 Initialization and Stop

Weights are initialized (in practice) with small random values and the adjusting process continue by iteration.

The stopping of the learning process can be done by one of the following methods:

- ① choosing a fixed number of steps  $t = \overline{1, T}$ .
- ② the learning process continue until the adjusting quantity  $\Delta w_{\ell ji} = w_{\ell ji(\text{at time } t+1)} - w_{\ell ji(\text{at time } t)}$  is under some specified value,  $\forall \ell, \forall j, \forall i$ .
- ③ learning stops when the total error, e.g. the total sum-of-squares  $E_{\text{tot.}}$ , attain a minima on a *test set*, *not* used for learning.

### Remarks:

- If the trained network performs well on the training set but have bad results on previously unseen patterns (i.e. it have poor generalization capabilities) then

this is usually a sign of overtraining (assuming, of course, that the network is reasonably build and there are a sufficient number of training patterns).

## → 2.3 The Algorithm

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

The activation and error functions and the stop condition are presumed to be chosen (known) and fixed.

**Network running procedure:**

1. The *input* layer is initialised, i.e. the output of input layer is made to be  $\mathbf{x}$ :

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

For all layers  $\ell = \overline{1, L}$  — starting with first hidden layer 1 — do:

$$\mathbf{z}_\ell = f(W_\ell \mathbf{z}_{\ell-1})$$

2. The output of the network is taken to be the output of the output layer, i.e.  $\mathbf{y} \equiv \mathbf{z}_L$ .

**Network learning procedure:**

1. Initialize all  $\{w_{\ell j i}\}$  weights with (small) random values.

2. For all training sets  $(\mathbf{x}_p, \mathbf{t}_p)$  (as long as the *stop* condition is not met) do:

- (a) Run the network to find the activations on all neurons  $\mathbf{a}_\ell$  and then the derivatives  $f'(\mathbf{a}_\ell)$ . The network output  $\mathbf{y}_p \equiv \mathbf{z}_L(\mathbf{x}_p) = f(\mathbf{a}_L)$  will also be required on next step.

**NOTE:** The algorithm require the derivatives of activation functions for all neurons. For most activation functions this may be expressed in terms of activation itself, i.e.  $f'(\mathbf{a}_\ell) = g(\mathbf{z}_\ell)$ , as is the case for logistic, see (2.2). This approach may reduce the memory usage or increase speed (or both in case of logistic function).

- (b) Using  $(\mathbf{y}_p, \mathbf{t}_p)$ , calculate  $\nabla_{\mathbf{z}_L} E$ , e.g. for sum-of-squares use (2.7a).
- (c) Compute the error gradient.
  - For output layer  $(\nabla E)_L$  is calculated directly from (2.6b) (or from (2.7c) for sum-of-squares and logistic).
  - For all other layers  $\ell = \overline{1, L-1}$ , going *backwards* from  $L-1$  to 1, calculate first  $\nabla_{\mathbf{z}_\ell} E$  using (2.6a) (or (2.7b) for sum-of-squares and logistic).

Then calculate  $(\nabla E)_\ell$  using (2.6b) (or respectively (2.7c)).

- (d) Update the  $W$  weights according to the *delta rule* (2.5).
- (e) Check the *stop* condition and exit if it have been met.



**Remarks:**

- In most cases<sup>1</sup> a better performance is obtained when training repeatedly with the whole training set. A shuffling of patterns is recommended, between repeats.

<sup>1</sup>But e.g. *not* when patterns form a time series.

- Trying to stop error backpropagation when is below some threshold value  $\delta$  may also improve learning, e.g. in case of sum-of-squares the  $\nabla_{\mathbf{z}_L} E$  may be changed to:

$$\frac{\partial E}{\partial z_{Lq}} = \begin{cases} z_{Lq} - t_q & \text{if } |z_{Lq} - t_q| > \delta \\ 0 & \text{otherwise} \end{cases} \quad \text{for } q = \overline{1, N_L}$$

i.e. rounding towards 0 the elements of  $\nabla_{\mathbf{z}_L} E$  smaller than  $\delta$ .

- The classical way of calculating the gradient, i.e.

$$\frac{\partial E}{\partial w_{\ell ji}} \approx \frac{E(w_{\ell ji} + \varepsilon) - E(w_{\ell ji} - \varepsilon)}{2\varepsilon} , \quad \varepsilon \gtrsim 0$$

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 (constants)  $\mu$  and  $\delta$ . The practical, hands-on, approach is still the best. Usual values for  $\mu$  are in the range  $[0.1, 1]$  (but some networks may learn even faster with  $\mu > 1$ ) and  $[0, 0.1]$  for  $\delta$ .
- In accordance with neuron output function (2.1) the output of the neuron have 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 transforming function may be used:

$$y(x) = 1 - \exp(-\alpha x) , \quad \alpha > 0 , \alpha = \text{const.}$$

which have the inverted:

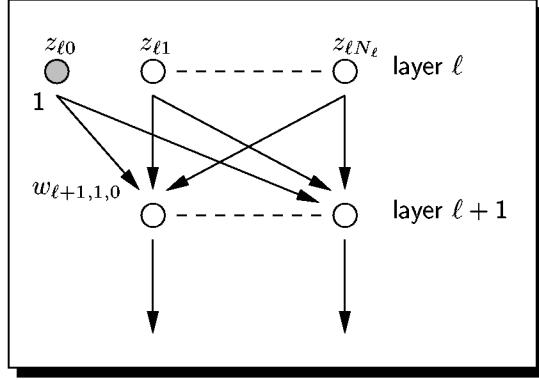
$$y^{-1}(x) = \frac{1}{\alpha} \ln \frac{1}{1-x}$$

The same procedure described above may be used for inputs.

This kind of transformation can be used each time the desired input/output falls beyond neuron activation function range.

- By no means reaching the *absolute* minima of  $E$  is guaranteed. First the training set is limited and the error minima with respect to the learning set may 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 have a symmetry, 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 trough the whole weight space but rather a small area of it. This is also the reason for which the starting point, given randomly, will not affect substantially the learning process.



**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).

## 2.4 Bias

Some problems, while having an obvious solution, cannot be solved with the architecture described above<sup>2</sup>. To solve these, the neuronal activation (2.3) is changed to:

$$z_{\ell k} = f \left( w_{\ell k 0} + \sum_{j=1}^{N_{\ell-1}} w_{\ell k j} z_{\ell-1, j} \right) \quad (2.8)$$

bias

❖  $w_{\ell k 0}$

and the new parameter  $w_{\ell k 0}$  introduced is named bias.

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

The required changes in neuronal outputs and weight matrices are:

$$\tilde{\mathbf{z}}_\ell^T = (1 \ z_{\ell 1} \ \dots \ z_{\ell N_\ell}) \quad \text{and} \quad \widetilde{W}_\ell = \begin{pmatrix} w_{\ell 1 0} & w_{\ell 1 1} & \dots & w_{\ell 1 N_{\ell-1}} \\ \vdots & \vdots & \ddots & \vdots \\ w_{\ell N_\ell 0} & w_{\ell N_\ell 1} & \dots & w_{\ell N_\ell N_{\ell-1}} \end{pmatrix}$$

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

❖  $(\widetilde{\nabla E})_\ell$

The error gradient matrix  $(\widetilde{\nabla E})_\ell$  associated with  $\widetilde{W}_\ell$  is:

$$(\widetilde{\nabla E})_\ell = \begin{pmatrix} \frac{\partial E}{\partial w_{\ell 1 0}} & \frac{\partial E}{\partial w_{\ell 1 1}} & \dots & \frac{\partial E}{\partial w_{\ell 1 N_{\ell-1}}} \\ \vdots & \vdots & \dots & \vdots \\ \frac{\partial E}{\partial w_{\ell N_\ell 0}} & \frac{\partial E}{\partial w_{\ell N_\ell 1}} & \dots & \frac{\partial E}{\partial w_{\ell N_\ell N_{\ell-1}}} \end{pmatrix}$$

Following the changes from above, the backpropagation theorem becomes:

<sup>2</sup>E.g. the tight encoder described later.

**Theorem 2.4.1. Backpropagation with biases.** If the error gradient with respect to neuronal outputs  $\nabla_{\mathbf{z}_L} E$  is known, and depends only on (actual) network outputs  $\{\mathbf{z}_L(\mathbf{x}_p)\}$  and targets  $\{\mathbf{t}_p\}$ :

$$\nabla_{\mathbf{z}_L} E = \text{known}.$$

then the error gradient (with respect to weights) may be calculated recursively according to formulas:

$$\nabla_{\mathbf{z}_\ell} E = W_{\ell+1}^T \cdot [\nabla_{\mathbf{z}_{\ell+1}} E \odot f'(\mathbf{a}_{\ell+1})] \quad \text{calculated recursively from } L-1 \text{ to } 1 \quad (2.9a)$$

$$\widetilde{(\nabla E)}_\ell = [\nabla_{\mathbf{z}_\ell} E \odot f'(\mathbf{a}_\ell)] \cdot \tilde{\mathbf{z}}_{\ell-1}^T \quad \text{for layers } \ell = \overline{1, L} \quad (2.9b)$$

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

*Proof.* See theorem 2.2.1 and its proof.

Equation (2.9a) results directly from (2.6a).

Columns 2 to  $N_{\ell-1} + 1$  of  $\widetilde{(\nabla E)}_\ell$  represents  $(\nabla E)_\ell$  given by (2.6b).

The only terms to be calculated remains those of first column of  $\widetilde{(\nabla E)}_\ell$ , i.e. terms of the form  $\frac{\partial E}{\partial w_{\ell j_0}}$ ,  $j$  being the row index. But these terms may be written as (see proof of theorem 2.2.1):

$$\frac{\partial E}{\partial w_{\ell j_0}} = \frac{\partial E}{\partial z_{\ell j}} \frac{\partial z_{\ell j}}{\partial w_{\ell j_0}}$$

where  $\frac{\partial E}{\partial z_{\ell j}}$  is term  $j$  of  $\nabla_{\mathbf{z}_\ell}$ , already calculated, and from (2.8):

$$\frac{\partial z_{\ell j}}{\partial w_{\ell j_0}} = f'(a_{\ell j}) \cdot 1 = f'(a_{\ell j})$$

As  $\tilde{\mathbf{z}}_{\ell-1,1} \equiv 1$  (by construction) then formula (2.9b) proves correct.  $\square$

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

$$\nabla_{\mathbf{z}_L} E = \mathbf{z}_L(\mathbf{x}) - \mathbf{t} \quad (2.10a)$$

$$\nabla_{\mathbf{z}_\ell} E = c W_{\ell+1}^T \cdot [\nabla_{\mathbf{z}_{\ell+1}} E \odot \mathbf{z}_{\ell+1} \odot (\hat{\mathbf{1}} - \mathbf{z}_{\ell+1})] \quad \text{for } \ell = \overline{1, L-1} \quad (2.10b)$$

$$\widetilde{(\nabla E)}_\ell = c [\nabla_{\mathbf{z}_\ell} E \odot \mathbf{z}_\ell \odot (\hat{\mathbf{1}} - \mathbf{z}_\ell)] \cdot \tilde{\mathbf{z}}_{\ell-1}^T \quad \text{for } \ell = \overline{1, L} \quad (2.10c)$$

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

*Proof.* It is proved the same way as proposition 2.2.1 but using theorem 2.4.1 instead.  $\square$

### Remarks:

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

## → 2.5 Algorithm Enhancements

### 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 is avoided by taking into account the previous adaptations in learning process (see also (2.5)):

$$\Delta W(t) = W(t+1) - W(t) = -\mu \nabla E|_{W(t)} + \alpha \Delta W(t-1) \quad (2.11)$$

momentum  
❖  $\alpha$

This procedure is named backpropagation with momentum and  $\alpha \in [0, 1]$  is named the momentum (learning) parameter.

The algorithm is very similar (*mutatis mutandi*) to the one described in section 2.3. As the main memory consumption is given by the requirement to store the weight matrix (especially true for large ANN), the momentum algorithm requires double the amount of standard backpropagation, to store  $\Delta W$  for next step.

#### Remarks:

- ➔ When choosing the momentum parameter the following results have to be considered:
  - if  $\alpha \geq 1$  then the contribution of each  $\Delta w_{\ell ji}$  grows infinitely.
  - if  $\alpha \lesssim 0$  then the momentum contribution is insignificant.
 so  $\alpha$  should be chosen somewhere in  $[0.5, 1]$  (in practice, usually  $\alpha \approx 0.9$ ).
- ➔ 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.
- ➔ Another improvement over momentum is the flat spot elimination. If the error surface is very flat then  $\nabla E \approx \tilde{0}$  and subsequently  $\Delta W \approx \tilde{0}$ . 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 (2.6b) may be performed as follows:

$$(2.6b) \quad \rightarrow \quad (\nabla E)_{\ell, \text{pseudo}} = \left\{ \nabla_{\mathbf{z}_\ell} E \odot \left[ f'(\mathbf{a}_\ell) + c_f \cdot \hat{\mathbf{1}} \right] \right\} \cdot \mathbf{z}_{\ell-1}^T$$

❖  $c_f$

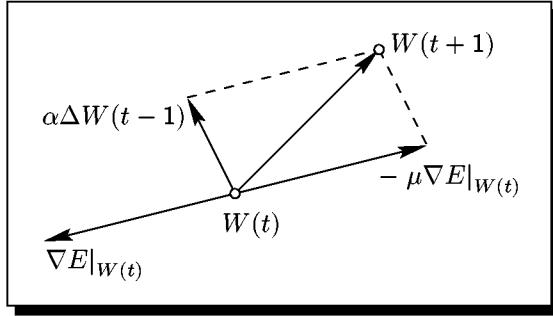
where  $(\nabla E)_{\ell, \text{pseudo}}$  is no more the real  $(\nabla E)_\ell$ . The  $c_f$  is named flat spot elimination constant.

Several points to note here:

- The procedure of adding a term to  $f'$  instead of multiplying it means that  $(\nabla E)_{\ell, \text{pseudo}}$  is more affected when  $f'$  is smaller — a desirable effect.

---

<sup>2.5.1</sup>[BTW95] p. 50



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

- The error gradient terms corresponding to a weight close to input layer is smaller than a similar term for a weight more closely to the output layer because the effect of changing the weight gets attenuated when propagated through layers. So another effect of  $c_f$  is the speed up of weight adaptation in layers close to input, again a desirable effect.
- The formulas (2.7c), (2.9b) and (2.10c) change the same way:

$$(\nabla E)_{\ell, \text{pseudo}} = c \left\{ \nabla_{z_\ell} E \odot \left[ z_\ell \odot (\hat{\mathbf{1}} - z_\ell) + c_f \cdot \hat{\mathbf{1}} \right] \right\} \cdot z_{\ell-1}^T \quad (2.12a)$$

$$\widetilde{(\nabla E)}_{\ell, \text{pseudo}} = \left\{ \nabla_{z_\ell} E \odot \left[ f'(\mathbf{a}_\ell) + c_f \cdot \hat{\mathbf{1}} \right] \right\} \cdot \tilde{z}_{\ell-1}^T \quad (2.12b)$$

$$\widetilde{(\nabla E)}_{\ell, \text{pseudo}} = c \left\{ \nabla_{z_\ell} E \odot \left[ z_\ell \odot (\hat{\mathbf{1}} - z_\ell) + c_f \cdot \hat{\mathbf{1}} \right] \right\} \cdot \tilde{z}_{\ell-1}^T \quad (2.12c)$$

→ In physical terms: The set of weights  $W$  may be thought as a set of coordinates defining a point in the space  $\mathbb{R}^{N_w}$ . During learning, this point is moved towards reducing the error  $E$ . The moment introduce an “inertia” proportional to  $\alpha$ , such that when changing direction under the influence of  $\nabla E$  “force” it have a tendency to keep the old direction of movement and “overshot” the point given by  $-\mu \nabla E$ . 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 (2.11) twice, for  $t-1$  and  $t$ ) it gives:

$$\Delta W(t) = -\mu \nabla E|_{W(t)} - \alpha \mu \nabla E|_{W(t-1)} + \alpha^2 \Delta W(t-2)$$

and it can be seen that the contributions of previous  $\Delta W$  gradually disappear with the increase of power of  $\alpha$  (as  $\alpha < 1$ ).

### 2.5.2 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 speed up learning over flat spot areas.

<sup>2.5.2</sup>[BTW95] p. 50

- If the slope of the error surface is step then a small learning parameter *should* be used to avoid overshooting the error minima.
- In general the slopes are gentle in some directions and step in the other ones.

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

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

$$\mu_{\ell j i}(t) = \begin{cases} \mathcal{I}\mu_{\ell j i}(t-1) & \text{if } \Delta w_{\ell j i}(t)\Delta w_{\ell j i}(t-1) \geq 0 \\ \mathcal{D}\mu_{\ell j i}(t-1) & \text{if } \Delta w_{\ell j i}(t)\Delta w_{\ell j i}(t-1) < 0 \end{cases} \quad (2.13)$$

❖  $\mathcal{I}, \mathcal{D}$  where  $\mathcal{I} \geq 1$  and  $\mathcal{D} \in (0, 1)$ . The  $\mathcal{I}$  parameter is named the adaptive increasing factor and the  $\mathcal{D}$  parameter is named the adaptive decreasing factor.

In matrix form, equation (2.13) may be written considering the matrix of  $\Delta w_{\ell j i}$  sign changes, i.e.  $\text{sign}[\Delta W(t) \odot \Delta W(t-1)]$ :

$$\mu(t) = \left\{ (\mathcal{I} - \mathcal{D}) \text{ sign} [\text{sign}(\Delta W(t) \odot \Delta W(t-1)) + \tilde{1}] + \mathcal{D} \cdot \tilde{1} \right\} \odot \mu(t-1) \quad (2.14)$$

*Proof.* The problem is to build a matrix containing 1-es corresponding to each  $\Delta w_{\ell j i}(t)\Delta w_{\ell j i}(t-1) \geq 0$  and 0-es in rest. This matrix multiplied by  $\mathcal{I}$  will be used to increase the corresponding  $\mu_{\ell j i}$  elements. The complementary matrix will be used to modify the matching  $\mu_{\ell j i}$  which have to be decreased.

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

$$\text{sign} [\text{sign}(\Delta W(t) \odot \Delta W(t-1)) + \tilde{1}]$$

while its complementary is

$$\tilde{1} - \text{sign} [\text{sign}(\Delta W(t) \odot \Delta W(t-1)) + \tilde{1}]$$

Then the updating formula for  $\mu$  finally becomes:

$$\begin{aligned} \mu(t) = & \mathcal{I}\mu(t-1) \odot \text{sign} [\text{sign}(\Delta W(t) \odot \Delta W(t-1)) + \tilde{1}] \\ & + \mathcal{D}\mu(t-1) \odot \left\{ \tilde{1} - \text{sign} [\text{sign}(\Delta W(t) \odot \Delta W(t-1)) + \tilde{1}] \right\} \quad \square \end{aligned}$$



### Remarks:

❖  $\mu_0$

- $\{\mu_{\ell j i}\}$  is initialized with a constant  $\mu_0$  and  $\Delta W(t-1) = \tilde{0}$ . Learning parameter matrix  $\mu$  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 apply.

Note that after initialization, when  $\mu(0) = \mu_0$  and  $\Delta W(0) = \tilde{0}$ , the first step will lead automatically to the increase  $\mu(1) = \mathcal{I}\mu_0$ , so  $\mu_0$  should be chosen accordingly.

Also, this algorithm requires three times as much memory compared to standard backpropagation, to store  $\mu$  and  $\Delta W$  for next step, both being of the same size as  $W$ .

- ➡ If  $\mathcal{I} = 1$  and  $\mathcal{D} = 1$  then the effect of algorithm is obviously void.
- In practice  $\mathcal{I} \in [1.1, 1.3]$  and  $\mathcal{D} \lesssim 1/\mathcal{I}$  gives the best results for a wide spectrum of applications.
- ➡ Note that  $\text{sign}(\Delta W(t) \odot \Delta W(t - 1))$  could be replaced by  $\text{sign}(\Delta W(t)) \odot \text{sign}(\Delta W(t - 1))$ . This is a tradeoff between one floating point multiplication followed by a sign versus two sign operations followed by an integer multiplication; whatever is faster may 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  $\nabla E$  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; then the adaptation process can be resumed at next step.

### 2.5.3 SuperSAB

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

The algorithm uses adaptive backpropagation for the  $w_{\ell ji}$  terms who continue to move in the same direction and momentum for the others, i.e.:

- If  $\Delta w_{\ell ji}(t) \Delta w_{\ell ji}(t - 1) \geq 0$  then:

$$\mu_{\ell ji}(t) = \mathcal{I} \mu_{\ell ji}(t - 1)$$

$$\Delta w_{\ell ji}(t + 1) = -\mu_{\ell ji}(t) \left. \frac{\partial E}{\partial w_{\ell ji}} \right|_{W(t)}$$

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

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

$$\mu_{\ell ji}(t) = \mathcal{D} \mu_{\ell ji}(t - 1)$$

$$\Delta w_{\ell ji}(t + 1) = -\mu_{\ell ji} \left. \frac{\partial E}{\partial w_{\ell ji}} \right|_{W(t)} - \alpha \Delta w_{\ell ji}(t)$$

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

In matrix notation SuperSAB rules are written as:

$$\mu(t) = \left\{ (\mathcal{I} - \mathcal{D}) \text{ sign} [\text{sign}(\Delta W(t) \odot \Delta W(t - 1)) + \tilde{1}] + \mathcal{D} \cdot \tilde{1} \right\} \odot \mu(t - 1)$$

$$\Delta W(t + 1) = -\mu(t) \odot \nabla E - \alpha \Delta W(t) \odot \left\{ \tilde{1} - \text{sign} [\text{sign}(\Delta W(t) \odot \Delta W(t - 1)) + \tilde{1}] \right\}$$

---

<sup>2.5.3</sup>[BTW95] p. 51

*Proof.* The first equation came directly from (2.14).

For the second equation, the matrix

$$\tilde{I} - \text{sign} [\text{sign}(\Delta W(t) \odot \Delta W(t-1)) + \tilde{I}]$$

contains, as elements, 1 if  $\Delta w_{\ell j i}(t)\Delta w_{\ell j i} < 0$  and zero in rest, so momentum terms are added exactly to the  $w_{\ell j i}$  requiring it, see proof of (2.14).  $\square$



### Remarks:

- ➔ While this algorithm uses the same main algorithm as described in section 2.3 (of course with the required changes) note however that the memory requirement is four times higher than for standard backpropagation, to store supplementary  $\mu$  and two  $\Delta W$ .
- ➔ Arguably, the matrix notation for this algorithm may be less beneficial in terms of speed. However: there is a benefit of splitting the effort of implementation into two levels, a lower one, dealing with matrix operations and a higher one dealing with the implementation of the algorithm itself. Beyond this an efficient matrix operations implementation may be already developed for the targeted system (e.g. an efficient matrix multiplication algorithm may be *several times* faster than the classical one when written specifically for the system used<sup>3</sup>, there is also the possibility of taking advantage of the hardware, threaded matrix operation on multiprocessor systems e.t.c.).

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

## 2.6 Applications

### 2.6.1 Identity Mapping Network

The network consists of 1 input, 1 hidden and 1 output neurons with 2 weights:  $w_1$  and  $w_2$ . See figure 2.6 on the next page.

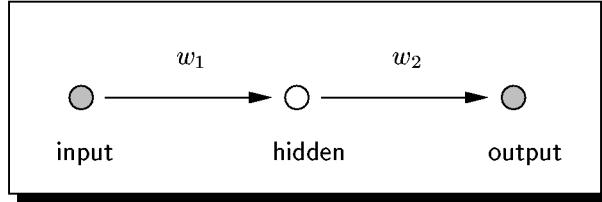
This particular network, while of little practical usage, it presents some interesting features:

- there are only 2 weights so it is possible to visualize exactly the error surface see figure 2.7 on the facing page;
- the error surface have a local maxima and a local minima, note that if the weights are “trapped” into the local minima the standard backpropagation can’t move forward as  $\nabla E$  becomes zero there.

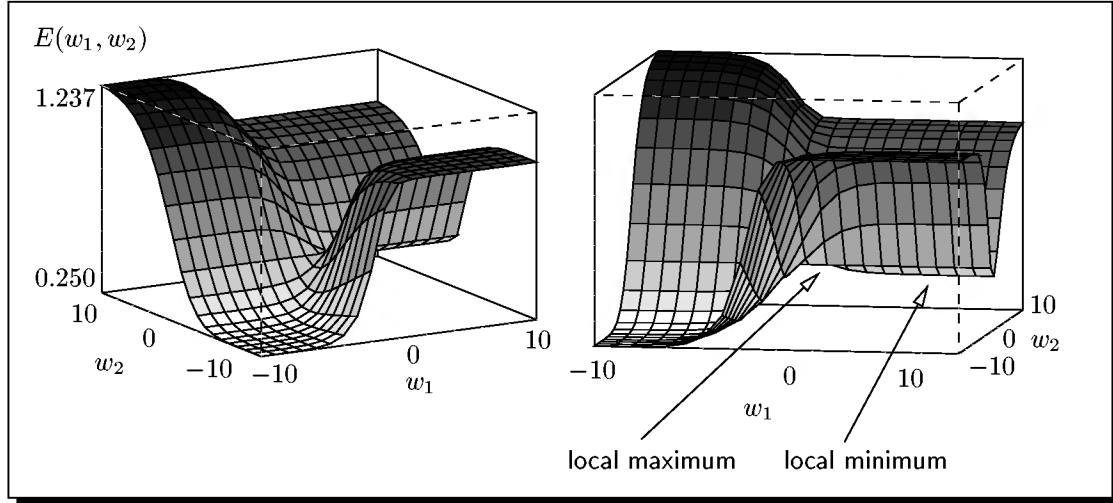
The problem is to configure  $w_1$  and  $w_2$  such that the identity mapping is realized for binary input.

---

<sup>3</sup>For a fast implementation of a matrix multiplication on a RISC processor , 8 times speed increase, see [Mos97]. For a multi-threaded matrix multiplication see [McC97].  
2.6.1 [BTW95] pp. 48–49



**Figure 2.6:** The identity mapping network



**Figure 2.7:** The error surface for identity mapping network.

The output of input neuron is  $x_p = z_{01}$  (by notation). The output of hidden neuron  $z_{11}$  is (see (2.3)):

$$z_{11} = \frac{1}{1 + \exp(-cw_1 z_{01})}$$

The output of the output neuron is:

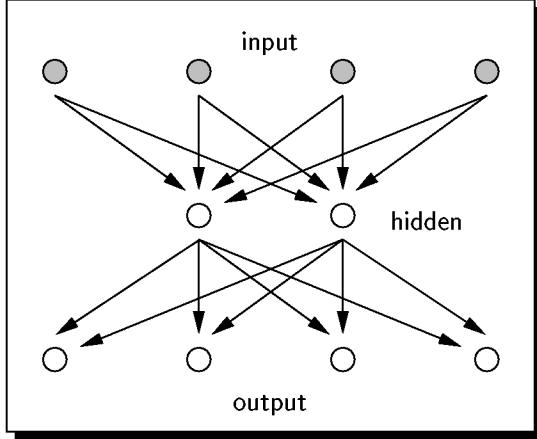
$$z_{21} = \frac{1}{1 + \exp(-cw_2 z_{11})} = \frac{1}{1 + \exp\left(\frac{cw_2}{1 + \exp(-cw_1 z_{01})}\right)}$$

The identity mapping network tries to map its input to output i.e.

$$\begin{aligned} \text{for } x_1 = z_{01} = 0 &\Rightarrow t_1(z_{01}) = 0 \\ \text{for } x_2 = z_{01} = 1 &\Rightarrow t_2(z_{01}) = 1 \end{aligned}$$

The square mean error is (2.4), where  $P = 2$  and  $N_L = 1$ :

$$E_{\text{tot.}}(w_1, w_2) = \frac{1}{2} \sum_{p=1}^P \sum_{q=1}^1 [z_{2q}(x_p) - t_q(x_p)]^2$$



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

$$= \frac{1}{2} \left[ \frac{1}{1 + \exp\left(\frac{-cw_2}{2}\right)} \right]^2 + \left[ \frac{1}{1 + \exp\left(\frac{-cw_2}{1+\exp(-cw_1)}\right)} - 1 \right]^2$$

For  $c = 1$  the error surface is shown in figure 2.7 on the page before. The surface have a local minimum and a local maximum.

### 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 layers. See figure 2.8.

Beyond any possible practical applications, this network shows the following:

- the architecture of an backpropagation ANN may be important with respect to its purpose;
- the output of a hidden layer is not necessary meaningless,
- the importance of biases.

The input vectors and targets are:

$$\mathbf{x}_1 = \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{x}_2 = \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix}, \quad \mathbf{x}_3 = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix}, \quad \mathbf{x}_4 = \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}, \quad \mathbf{t}_i \equiv \mathbf{x}_i, \quad i = \overline{1, 4}$$

The idea is that the inputs have to be “squeezed” through the bottleneck represented by hidden layer, before being reproduced at output. The network have to find a way to encode the 4-component vectors on a 2-component vector, the output of hidden layer. Obviously the encoding is given by:

$$\mathbf{z}_1 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad \mathbf{z}_2 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}, \quad \mathbf{z}_3 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad \mathbf{z}_4 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}$$

Note that one of  $x_i$  vectors will be encoded by  $z_1$ . On a network without biases this means that the output layer will receive total input  $a$  and considering the logistic activation function then the corresponding output will **always** be  $y^T = (0.5 \ 0.5 \ 0.5 \ 0.5)$  and this particular output will be weights-independent. One of the input vectors may never be learned by the encoder. In practice usually (but not always) the net will enter in oscillation trying to learn two vectors on one encoding so there will be two unlearn vectors. When using biases this do not happen as the output layer will always receive something weight-dependent.

An ANN was trained with the following parameters:

$$\text{learning parameter} = \mu = 2.5$$

$$\text{momentum} = \alpha = 0.9$$

$$\text{flat spot elimination} = c_f = 0.25$$

and after 200 epochs the outputs of hidden layer became:

$$z_1 = \begin{pmatrix} 1 \\ 0 \end{pmatrix}, \quad z_2 = \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \quad z_3 = \begin{pmatrix} 1 \\ 1 \end{pmatrix}, \quad z_4 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

and the corresponding output:

$$y_1 = \begin{pmatrix} 0.9975229 \\ 0.0047488 \\ 0.0015689 \\ 1.876 \cdot 10^{-8} \end{pmatrix}, \quad y_2 = \begin{pmatrix} 0.0000140 \\ 0.9929489 \\ 8.735 \cdot 10^{-9} \\ 0.0045821 \end{pmatrix},$$

$$y_3 = \begin{pmatrix} 0.0020816 \\ 7.241 \cdot 10^{-11} \\ 0.997392 \\ 0.0010320 \end{pmatrix}, \quad y_4 = \begin{pmatrix} 7.227 \cdot 10^{-11} \\ 0.0000021 \\ 0.0021213 \\ 0.9960705 \end{pmatrix}$$



### Remarks:

- The encoders with  $N_1 = \log_2 N_0$  are called *tight* encoders, those with  $N_1 < \log_2 N_0$  are *loose* and those with  $N_1 > \log_2 N_0$  are *supertight*.
- It is possible to train a loose encoder 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 represents an example of an ANN with unsupervised learning.

### 3.1 Network Structure

A SOM (Self Organizing Map, known also as Kohonen) network have **one** single layer, let name this one the output layer. The additional input (“sensory”) layer just distribute the inputs to output layer, there is no data processing on it. Into the output layer a lateral (feedback) interaction is provided (see also section 3.3). The number of neurons on input layer is  $N$  — equal to the dimension of input vector and for output layer is  $K$ . See figure 3.1 on the next page.

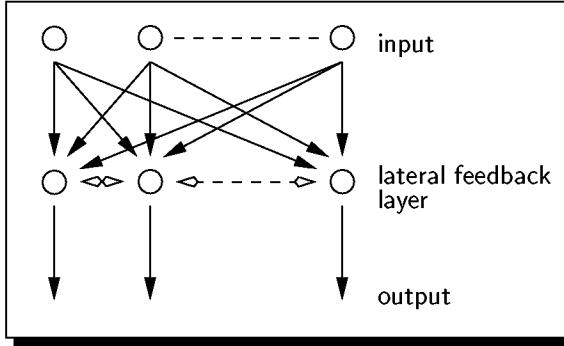
❖  $N, K$

#### Remarks:

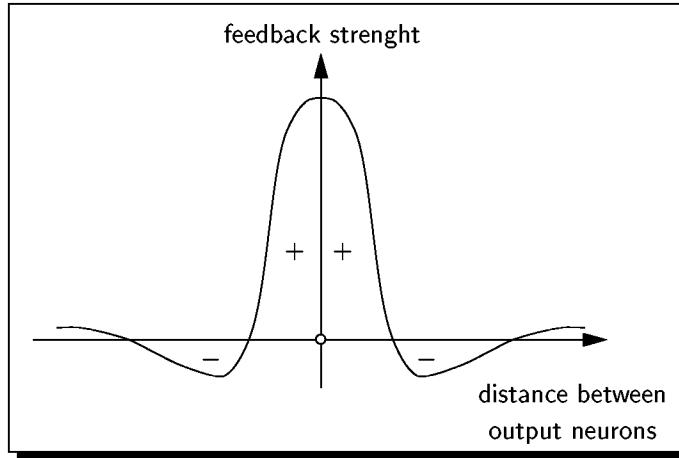
- Here the output layer have been considered unidimensional. Taking into account the “mapping” feature of the Kohonen networks the output layer may be considered — more convenient for some particular applications — multidimensional.
- A multidimensional output layer may be trivially mapped to a unidimensional one and the discussion below will remain the same.  
E.g. a bidimensional layer  $K \times K$  may be mapped to a unidimensional layer having  $K^2$  neurons just by finding a function  $f : K \times K \rightarrow K$  to do the relabeling/numbering of neurons. Such a function may be e.g.  $f(j, \ell) = (\ell - 1)K + j$  which maps first row of neurons  $(1, 1) \dots (1, K)$  to the first  $K$  unidimensional chunk and so on ( $j, \ell = \overline{1, K}$ ).

---

<sup>3.1</sup>See [BTW95] pp. 83–89 and [Koh88] pp. 119–124.



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



**Figure 3.2:** The lateral feedback interaction function of the “mexican hat” type.

The important thing 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 closest neurons receive a positive feedback, the more distant ones receive negative feedback and the far away ones are not affected.



### Remarks:

inter-neuronal distances

- ➔ The distance between neuron neighbors in output layer is (obvious) a discrete one. 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 obvious one being the Euclidean.
- ➔ 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).

- ➡ The area affected by the lateral feedback is named **neuronal neighborhood**. neuronal neighborhood
- ➡ For sufficiently large neighborhoods the distance may be considered continue when carrying some types of calculations.

## 3.2 Types of Neuronal Learning

### 3.2.1 The Learning Process

Let  $W = \{w_{ji}\}_{\substack{j=1, K \\ i=1, N}}$  be the weight matrix and  $\mathbf{x} = \{x_i\}_{i=1, N}$  be the input vector for a

(output) neuron, i.e. the total 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  $W(j, :)$  of the *same size* and from the *same space*  $\mathbb{R}^N$  as the input vector  $\mathbf{x}$ .  $\mathbb{R}^N$  is named the *weight space*.

❖  $W(j, :)$   
weight 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. the one for which:

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

is declared “winner”. All neurons included in its vicinity (neuronal neighborhood), including itself, will participate to the “learning” of  $\mathbf{x}$ . The other ones are not affected.

The learning process consists of changing the weight vectors  $W(j, :)$  towards the input vector (positive feedback). There is also a “forgetting” process which tries to slow down the progress (negative feedback).

It can be immediately seen why the feedback is indirect: the neurons are affected by being in the neuronal neighborhood of the winner, *not* by receiving directly the winner’s output.

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

$$\frac{dW}{dt} = \phi(\mathbf{x}, W) - \gamma(\mathbf{x}, W)$$

where  $\phi$  and  $\gamma$  are scalar (possibly nonlinear) functions,  $\phi$  representing the positive feedback,  $\gamma$  being the negative one. These two functions have to be build in such a way as to affect only the neuronal neighborhood of winning neuron  $k$ , which vary in time as the input vector is function of time  $\mathbf{x} = \mathbf{x}(t)$ . Note that here the winning neuron appears implicitly as it may be determined from  $\mathbf{x}$  and  $W$ .

❖  $\phi, \gamma$

Various adaptation models (differential equations for  $W$ ) can be build for the neurons (from the output layer) of the Kohonen network. 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 the neuronal neighborhood is sufficiently large to contain the whole network. Later it will be shown how to limit weight change/adaptation to targeted neurons by using the lateral feedback function.

<sup>3.2</sup>See [Koh88] pp. 92–98.

### 3.2.2 The Trivial Equation

❖  $\alpha, \beta$

One of the most simple equations is a linear differential:

$$\frac{dW(j,:)}{dt} = \alpha \mathbf{x}^T - \beta W(j,:) ; \quad \alpha, \beta > 0 ; \quad \alpha, \beta = \text{const.} ; \quad j = \overline{1, K}$$

which in matrix form becomes:

$$\frac{dW}{dt} = \alpha \hat{\mathbf{1}} \mathbf{x}^T - \beta W \quad (3.1)$$

and with initial condition  $W(0) \equiv W_0$  the solution is:

$$W(t) = \left[ \alpha \hat{\mathbf{1}} \left( \int_0^t \mathbf{x}^T(t') e^{\beta t'} dt' \right) + W_0 \right] e^{-\beta t}$$

which shows that for  $t \rightarrow \infty$ ,  $W(j,:)$  is the *exponentially* weighted average of  $\mathbf{x}(t)$  and do not produce any interesting effects.

*Proof.* The equation is solved by the method of variation of parameters. First, the homogeneous equation:

$$\frac{dW}{dt} + \beta W = 0$$

have a solution of the form:

$$W(t) = C e^{-\beta t} , \quad C = \text{matrix of constants}$$

The general solution for the nonhomogeneous equation (3.1) is found by considering  $C = C(t)$ . Then:

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

and by replacing in (3.1) it gives:

$$\begin{aligned} \frac{dC}{dt} e^{-\beta t} - \beta C(t) e^{-\beta t} &= \alpha \hat{\mathbf{1}} \mathbf{x}^T(t) - \beta C(t) e^{-\beta t} \Rightarrow \\ \Rightarrow \frac{dC}{dt} &= \alpha \hat{\mathbf{1}} \mathbf{x}^T(t) e^{\beta t} \Rightarrow C(t) = \alpha \hat{\mathbf{1}} \int_0^t \mathbf{x}^T(t') e^{\beta t'} dt' + C' \quad (C' = \text{matrix of constants}) \end{aligned}$$

and, at  $t = 0$ ,  $W(0) = C' \equiv W_0$ . □

### 3.2.3 The Simple Equation

The simple equation is defined as:

$$\frac{dW(j,:)}{dt} = \alpha a_j(t) \mathbf{x}^T - \beta W(j,:) , \quad \alpha, \beta > 0 , \quad \alpha, \beta = \text{const.} , \quad j = \overline{1, K}$$

and in matrix notation it becomes:

$$\frac{dW}{dt} = \alpha \mathbf{a}(t) \mathbf{x}^T - \beta W$$

and consequently, as  $\mathbf{a} = W \mathbf{x}$  then:

$$\frac{dW}{dt} = W(\alpha \mathbf{x} \mathbf{x}^T - \beta I)$$

In the time-discrete approximation:

$$\frac{dW}{dt} \rightarrow \frac{\Delta W}{\Delta t} = \frac{W(t+1) - W(t)}{(t+1) - t} = W(t) [\alpha \mathbf{x}(t) \mathbf{x}^T(t) - \beta I]$$

$$\Rightarrow W(t+1) = W(t) [\alpha \mathbf{x}(t) \mathbf{x}^T(t) - \beta I + I], \quad t \in \mathbb{N}^+, \quad W(0) \equiv W_0 \text{ (initial condition)}$$

so the general solution is:

$$W(t) = W_0 \prod_{t'=0}^{t-1} [\alpha \mathbf{x}(t') \mathbf{x}^T(t') - \beta I + I] \quad (3.2)$$

### Remarks:

- For most cases the solution (3.2) 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  $\alpha$  relatively small, such that the superior order terms  $\mathcal{O}(\alpha^2)$  may be neglected, and considering  $b = 0$  (no “forgetting” effect) then from (3.2):

$$W(t) \simeq W_0 \left[ I + \alpha \sum_{t'=0}^{t-1} \mathbf{x}(t') \mathbf{x}^T(t') \right]$$

#### 3.2.4 The Riccati Equation

The Riccati equation is defined as:

$$\frac{dW(j,:)}{dt} = \alpha \mathbf{x}^T - \beta a_j W(j,:) , \quad \alpha, \beta > 0 , \quad \alpha, \beta = \text{const.} , \quad j = \overline{1, K} \quad (3.3)$$

and after the replacement  $a_j = W(j,:) \mathbf{x} = \mathbf{x}^T W(j,:)^T$  (note that  $[W(j,:)]^T \equiv W(j,:)^T$  ♦♦  $W(j,:)^T$  for brevity), it becomes:

$$\frac{dW(j,:)}{dt} = \mathbf{x}^T [\alpha I - \beta W(j,:)^T W(j,:)] \quad (3.4)$$

or in matrix notation:

$$\frac{dW}{dt} = \alpha \hat{\mathbf{1}} \mathbf{x}^T - \beta (W \mathbf{x} \hat{\mathbf{1}}^T) \odot W \quad (3.5)$$

*Proof.* Equation (3.3) may be written as:  $\frac{dW}{dt} = \alpha \hat{\mathbf{1}} \mathbf{x}^T - \beta (\mathbf{a} \hat{\mathbf{1}}^T) \odot W$  and  $\mathbf{a} = W \mathbf{x}$ .  $\square$

For general  $\mathbf{x} = \mathbf{x}(t)$ , the Riccati equation is not integrable directly (of course beyond the trivial  $\mathbf{x}^T = W(j,:) = \hat{\mathbf{0}}$ ). However a statistical approach may be performed.

**Proposition 3.2.1.** *Considering a statistical approach to the Riccati equation (3.4), if there is a solution, i.e.  $\lim_{t \rightarrow \infty} W$  exists, then the solution of  $W$  is of the form:*

$$\lim_{t \rightarrow \infty} W = \sqrt{\frac{\alpha}{\beta}} \hat{\mathbf{1}} \frac{\langle \mathbf{x} \rangle^T}{\| \langle \mathbf{x} \rangle \|} \quad \text{if } \langle \mathbf{x} \rangle \neq \hat{\mathbf{0}}$$

where  $\langle \mathbf{x} \rangle = \mathcal{E}\{\mathbf{x}|W\} = \text{const.}, \text{ independent of } W \text{ and time}; \text{ i.e. all } W(j,:) \text{ became parallel with } \langle \mathbf{x} \rangle \text{ in } \mathbb{R}^N \text{ and will have the norm } \|W(j,:)\| = \sqrt{\alpha/\beta} \text{ (the Euclidean metric being used here).}$

❖  $\theta$  Proof. As  $\mathbf{x}$  and  $W(j,:)$  may be seen as vectors in  $\mathbb{R}^N$  space, then let  $\theta$  be the angle between them. From the scalar product,  $\cos \theta$  is:

$$\cos \theta = \frac{W(j,:)\mathbf{x}}{\|W(j,:)\|\|\mathbf{x}\|}$$

❖  $\|\mathbf{x}\|, \|W(j,:)\|$  where  $\|\mathbf{x}\|^2 = \mathbf{x}^T \mathbf{x}$  and  $\|W(j,:)\|^2 = W(j,:)^T W(j,:)$ , the Euclidean metric being used here.

When bringing under the  $\mathcal{E}\{\cdot|W\}$  operator, as  $\mathbf{x}$  is obviously independent of  $W$  (is the input vector), it goes to  $\mathbf{x} \rightarrow \langle \mathbf{x} \rangle$ . Then the expected value of  $d \cos \theta / dt$  is:

$$\mathcal{E}\left\{\frac{d \cos \theta}{dt} \middle| W\right\} = \mathcal{E}\left\{\frac{\frac{dW(j,:)\langle \mathbf{x} \rangle}{dt}}{\|W(j,:)\|\|\langle \mathbf{x} \rangle\|} - \frac{W(j,:)\langle \mathbf{x} \rangle \frac{d\|W(j,:)\|}{dt}}{\|W(j,:)\|^2\|\langle \mathbf{x} \rangle\|} \middle| W\right\} \quad (3.6)$$

□ Term  $\frac{dW(j,:)\langle \mathbf{x} \rangle}{dt}$

First  $\frac{dW(j,:)\langle \mathbf{x} \rangle}{dt} = \frac{dW(j,:)}{dt} \langle \mathbf{x} \rangle$ , as  $\langle \mathbf{x} \rangle$  is time independent. Then, by multiplying (3.4) to the right by  $\mathbf{x}$ :

$$\frac{dW(j,:)}{dt} \mathbf{x} = \alpha \mathbf{x}^T \mathbf{x} - \beta \mathbf{x}^T W^T(j,:) W(j,:) \mathbf{x} = \alpha \|\mathbf{x}\|^2 - \beta [W(j,:)] \mathbf{x}^2$$

(as for two matrices  $(AB)^T = B^T A^T$  is true), and then this term becomes:

$$\frac{dW(j,:)\langle \mathbf{x} \rangle}{dt} = \frac{\alpha \|\langle \mathbf{x} \rangle\|^2 - \beta [W(j,:)] \langle \mathbf{x} \rangle^2}{\|W(j,:)\|\|\langle \mathbf{x} \rangle\|}$$

(as  $\mathbf{x} \rightarrow \langle \mathbf{x} \rangle$  under  $\mathcal{E}\{\cdot|W\}$  operator).

□ Term  $\frac{W(j,:)\langle \mathbf{x} \rangle \frac{d\|W(j,:)\|}{dt}}{\|W(j,:)\|^2\|\langle \mathbf{x} \rangle\|}$

First the derivative  $d\|W(j,:)\|/dt$  is found as follows:

$$\begin{aligned} \frac{d\|W(j,:)\|^2}{dt} &= \begin{cases} 2\|W(j,:)\| \frac{d\|W(j,:)\|}{dt} \\ 2 \frac{dW(j,:)}{dt} W(j,:)^T \text{ (because } \|W(j,:)\|^2 = W(j,:)^T W(j,:)) \end{cases} \Rightarrow \\ \frac{d\|W(j,:)\|}{dt} &= \frac{dW(j,:)}{dt} \frac{W(j,:)^T}{\|W(j,:)\|} \end{aligned}$$

and by using (3.4) then

$$\begin{aligned} \frac{d\|W(j,:)\|}{dt} &= \left[ \alpha \mathbf{x}^T - \beta \mathbf{x}^T W(j,:)^T W(j,:) \right] \frac{W(j,:)^T}{\|W(j,:)\|} \\ &= \left[ \alpha \mathbf{x}^T W(j,:)^T - \beta \mathbf{x}^T W(j,:)^T W(j,:) W(j,:)^T \right] \frac{1}{\|W(j,:)\|} \\ &= \frac{1}{\|W(j,:)\|} \mathbf{x}^T W(j,:)^T \left[ \alpha - \beta W(j,:)^T W(j,:)^T \right] \\ &= \frac{W(j,:)\mathbf{x}}{\|W(j,:)\|} [\alpha - \beta \|\mathbf{x}\|^2] \end{aligned} \quad (3.7)$$

(as  $\mathbf{x}^T W(j,:)^T = W(j,:)^T \mathbf{x}$  and  $W(j,:)^T W(j,:)^T \equiv \|\mathbf{x}\|^2$ ). By replacing back into the wanted term and as  $\mathbf{x} \rightarrow \langle \mathbf{x} \rangle$ :

$$\frac{dW(j,:)\langle \mathbf{x} \rangle}{dt} = \frac{[W(j,:)\langle \mathbf{x} \rangle]^2(\alpha - \beta \|\mathbf{x}\|^2)}{\|W(j,:)\|^3\|\langle \mathbf{x} \rangle\|}$$

Replacing back into (3.6) gives

$$\begin{aligned}\mathcal{E} \left\{ \frac{d \cos \theta}{dt} \middle| W \right\} &= \mathcal{E} \left\{ \frac{\alpha \|\langle \mathbf{x} \rangle\|^2 - \beta [W(j,:) \langle \mathbf{x} \rangle]^2}{\|W(j,:)\| \|\langle \mathbf{x} \rangle\|} - \frac{[W(j,:) \langle \mathbf{x} \rangle]^2 (\alpha - \beta \|W(j,:)\|^2)}{\|W(j,:)\|^3 \|\langle \mathbf{x} \rangle\|} \middle| W \right\} \\ &= \mathcal{E} \left\{ \frac{\alpha \|\langle \mathbf{x} \rangle\|^2 \|W(j,:)\|^2 - \alpha [W(j,:) \langle \mathbf{x} \rangle]^2}{\|W(j,:)\|^3 \|\langle \mathbf{x} \rangle\|} \middle| W \right\} \\ &= \alpha \|\langle \mathbf{x} \rangle\| \mathcal{E} \left\{ \frac{1 - \cos^2 \theta}{\|W(j,:)\|} \middle| W \right\} = \alpha \|\langle \mathbf{x} \rangle\| \mathcal{E} \left\{ \frac{\sin^2 \theta}{\|W(j,:)\|} \middle| W \right\}\end{aligned}\quad (3.8)$$

Existence of  $\lim_{t \rightarrow \infty} W$  means that  $\theta$  stabilizes in time and then its derivative limit is zero and the expected value of the derivative is also zero (as it will remain zero after reaching the limit):

$$\lim_{t \rightarrow \infty} \frac{d \cos \theta}{dt} = 0 = \mathcal{E} \left\{ \frac{d \cos \theta}{dt} \middle| W \right\}$$

By using (3.8), it follows immediately that  $\mathcal{E}\{\sin \theta | W\} = 0$  and then  $\mathcal{E}\{\theta | W\} = 0$ , i.e. all  $\mathcal{E}\{W(j,:) | W\}$  are parallel to  $\langle \mathbf{x} \rangle$ .

The norm of  $W(j,:)$  is found from (3.7). If  $\lim_{t \rightarrow \infty} W(j,:)$  does exist then the expectation of  $d\|W(j,:)\|/dt$  have to be zero:

$$\mathcal{E} \left\{ \frac{d\|W(j,:)\|}{dt} \middle| W \right\} = \mathcal{E} \left\{ \frac{W(j,:) \mathbf{x}}{\|W(j,:)\|} [\alpha - \beta \|W(j,:)\|^2] \middle| W \right\} = 0$$

but as  $W(j,:) \neq \hat{0}$ , this may happen only if

$$\mathcal{E} \left\{ \alpha - \beta \|W(j,:)\|^2 \middle| W \right\} = 0 \Rightarrow (\mathcal{E}\{\|W(j,:)\|\})^2 = \frac{\alpha}{\beta}$$

Finally, combining all previously obtained results:

$$\begin{cases} \lim_{t \rightarrow \infty} \cos(\widehat{W(j,:) \cdot \langle \mathbf{x} \rangle}) = 1 \Rightarrow \lim_{t \rightarrow \infty} \frac{W(j,:)}{\|W(j,:)\|} = \frac{\langle \mathbf{x} \rangle^T}{\|\langle \mathbf{x} \rangle\|} \\ \lim_{t \rightarrow \infty} \|W(j,:)\| = \sqrt{\frac{\alpha}{\beta}} \end{cases} \Rightarrow \lim_{t \rightarrow \infty} W(j,:) = \sqrt{\frac{\alpha}{\beta}} \frac{\langle \mathbf{x} \rangle^T}{\|\langle \mathbf{x} \rangle\|} \quad \square$$

### 3.2.5 More General Equations

**Theorem 3.2.1.** Let  $\alpha > 0$ ,  $\mathbf{a} = W\mathbf{x}$  and  $\gamma(\mathbf{a})$  an arbitrary function such that  $\mathcal{E}\{\gamma(\mathbf{a}) | W\}$  exists. Let  $\mathbf{x} = \mathbf{x}(t)$  a vector with stationary statistical properties (and independent of  $W$ ).  $\diamond \alpha, \mathbf{a}, \gamma$

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

$$\frac{dW(j,:)}{dt} = \alpha \mathbf{x}^T - \gamma(a_j) W(j,:) \quad , \quad j = \overline{1, K} \quad (3.9)$$

or, in matrix notation:

$$\frac{dW}{dt} = \alpha \hat{\mathbf{1}} \mathbf{x}^T - [\gamma(\mathbf{a}) \hat{\mathbf{1}}^T] \odot W$$

have nonzero bounded  $W$  solutions for  $t \rightarrow \infty$ , then it must be of the form:

$$\lim_{t \rightarrow \infty} W \propto \hat{\mathbf{1}} \cdot \langle \mathbf{x} \rangle^T$$

where  $\langle \mathbf{x} \rangle$  is the mean of  $\mathbf{x}(t)$ ; i.e. all  $W(j,:)$  become parallel to  $\langle \mathbf{x} \rangle$  in  $\mathbb{R}^N$ .

*Proof.* Let  $\theta$  be the angle between  $\langle \mathbf{x} \rangle$  and  $W(j,:)$  vectors in  $\mathbb{R}^N$  then, from the scalar product:  $\cos \theta = \frac{W(j,:)\langle \mathbf{x} \rangle}{\|W(j,:)\| \|\langle \mathbf{x} \rangle\|}$ . The  $\mathcal{E}\{d \cos \theta / dt | W\}$  is calculated in similar way as in proof of proposition 3.2.1.  $\diamond \theta$

$$\mathcal{E}\left\{\frac{d \cos \theta}{dt} \middle| W\right\} = \mathcal{E}\left\{\frac{\frac{d W(j,:)}{dt} \langle \mathbf{x} \rangle}{\|W(j,:)\| \|\langle \mathbf{x} \rangle\|} - \frac{[W(j,:)\langle \mathbf{x} \rangle] \frac{d \|W(j,:)\|}{dt}}{\|W(j,:)\|^2 \|\langle \mathbf{x} \rangle\|} \middle| W\right\} \quad (3.10)$$

Multiplying (3.9) by  $\mathbf{x}$ , to the right, gives:

$$\frac{d W(j,:)}{dt} \mathbf{x} = \alpha \mathbf{x}^T \mathbf{x} - \gamma(a_j) W(j,:) \mathbf{x} = \alpha \|\mathbf{x}\|^2 - \gamma(a_j) [W(j,:)] \mathbf{x} \quad (3.11)$$

The  $d\|W(j,:)\|/dt$  derivative is calculated in similar way as in proof of proposition 3.2.1 to give:

$$\frac{d\|W(j,:)\|}{dt} = \frac{d W(j,:)}{dt} \frac{W(j,:)^T}{\|W(j,:)\|}$$

and then, by using (3.9):

$$\frac{d\|W(j,:)\|}{dt} = \alpha \frac{\mathbf{x}^T W(j,:)^T}{\|W(j,:)\|} - \gamma(a_j) \frac{W(j,:)^T W(j,:)}{\|W(j,:)\|} = \alpha \frac{[W(j,:)\mathbf{x}]}{\|W(j,:)\|} - \gamma(a_j) \|W(j,:)\| \quad (3.12)$$

The (3.11) and (3.12) results are used in (3.10) (and also  $\mathbf{x} \rightarrow \langle \mathbf{x} \rangle$ ) to give:

$$\begin{aligned} \mathcal{E}\left\{\frac{d \cos \theta}{dt} \middle| W\right\} &= \mathcal{E}\left\{\frac{\alpha \|\langle \mathbf{x} \rangle\|^2 - \gamma(a_j) [W(j,:)\langle \mathbf{x} \rangle]}{\|W(j,:)\| \|\langle \mathbf{x} \rangle\|} \right. \\ &\quad \left. - \frac{[W(j,:)\langle \mathbf{x} \rangle] (\alpha \frac{[W(j,:)\mathbf{x}]}{\|W(j,:)\|} - \gamma(a_j) \|W(j,:)\|)}{\|W(j,:)\|^2 \|\langle \mathbf{x} \rangle\|} \middle| W\right\} \end{aligned}$$

and, after simplification, it becomes:

$$\mathcal{E}\left\{\frac{d \cos \theta}{dt} \middle| W\right\} = \alpha \mathcal{E}\left\{\frac{\|\langle \mathbf{x} \rangle\|^2 \|W(j,:)\|^2 - [W(j,:)\langle \mathbf{x} \rangle]^2}{\|W(j,:)\|^3 \|\langle \mathbf{x} \rangle\|} \middle| W\right\}$$

Existence of  $\lim_{t \rightarrow \infty} W$  means that the  $\theta$  stabilizes in time and then  $\lim_{t \rightarrow \infty} (d \cos \theta / dt) = 0$ , and then the expected value is zero as well:  $\mathcal{E}\{d \cos \theta / dt | W\} = 0$ . But this may happen only if:

$$\mathcal{E}\left\{\frac{d \cos \theta}{dt} \middle| W\right\} = 0 \Leftrightarrow \mathcal{E}\{\|\langle \mathbf{x} \rangle\|^2 \|W(j,:)\|^2 - [W(j,:)\langle \mathbf{x} \rangle]^2 | W\} = 0 \Leftrightarrow$$

$$\mathcal{E}\left\{\frac{W(j,:)\langle \mathbf{x} \rangle}{\|W(j,:)\| \|\langle \mathbf{x} \rangle\|} \middle| W\right\} = 1 \Leftrightarrow \mathcal{E}\{\cos \theta | W\} = 1$$

i.e.  $\lim_{t \rightarrow \infty} \theta = 0$  and then  $W(j,:)$  and  $\langle \mathbf{x} \rangle$  become parallel for  $t \rightarrow \infty$ , i.e.  $\lim_{t \rightarrow \infty} W(j,:) \propto \langle \mathbf{x} \rangle^T$ .  $\square$

**Theorem 3.2.2.** Let  $\alpha > 0$ ,  $\mathbf{a} = W\mathbf{x}$  and  $\gamma(\mathbf{a})$  an arbitrary function such that  $\mathcal{E}\{\gamma(\mathbf{a}) | W\}$  exists. Let  $\langle \mathbf{x}\mathbf{x}^T \rangle = \mathcal{E}\{\mathbf{x}\mathbf{x}^T | W\}$  (in fact  $\mathbf{x}\mathbf{x}^T$  does not depend on  $W$  as is the covariance matrix of input vector). Let  $\lambda_{max} = \max_\ell \lambda_\ell$  the maximum eigenvalue of  $\langle \mathbf{x}\mathbf{x}^T \rangle$  and  $\mathbf{u}_{max}$  the associated eigenvector.

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

$$\frac{d W(j,:)}{dt} = \alpha a_j \mathbf{x}^T - \gamma(a_j) W(j,:) \quad (3.13)$$

or, in matrix notation:

$$\frac{d W}{dt} = \alpha \mathbf{a} \mathbf{x}^T - [\gamma(\mathbf{a}) \hat{\mathbf{1}}^T] \odot W$$

<sup>3.2.5</sup>See [Koh88] pp. 98–101.

have nonzero bounded  $W$  solutions for  $t \rightarrow \infty$ , they have to be of the form:

$$W \propto \hat{\mathbf{1}} \mathbf{u}_{\max}^T$$

provided that  $W \mathbf{u}_{\max} \neq \hat{\mathbf{0}}$ , where  $W(0) \equiv W_0$ ; i.e. all  $W(j,:)$  become parallel to  $\mathbf{u}_{\max}$  in  $\mathbb{R}^N$ .  $\diamond W_0$

*Proof.* Let  $\lambda_\ell$  be an eigenvalue and  $\mathbf{u}_\ell$  the corresponding eigenvector of  $\langle \mathbf{x} \mathbf{x}^T \rangle$  such that  $\langle \mathbf{x} \mathbf{x}^T \rangle \mathbf{u}_\ell = \lambda_\ell \mathbf{u}_\ell$ . Let  $\theta_\ell$  be the angle between  $W(j,:)$  and  $\mathbf{u}_\ell$  such that  $\cos \theta_\ell = \frac{W(j,:) \mathbf{u}_\ell}{\|W(j,:)\| \|\mathbf{u}_\ell\|}$ .  $\diamond \lambda_\ell, \mathbf{u}_\ell$   $\diamond \theta_\ell$

$\mathcal{E}\{d \cos \theta_\ell / dt | W\}$  is calculated the same way as in proof of theorem 3.2.1:

$$\mathcal{E} \left\{ \frac{d \cos \theta_\ell}{dt} \middle| W \right\} = \mathcal{E} \left\{ \frac{\frac{dW(j,:)}{dt} \mathbf{u}_\ell}{\|W(j,:)\| \|\mathbf{u}_\ell\|} - \frac{[W(j,:) \mathbf{u}_\ell] \frac{d\|W(j,:)\|}{dt}}{\|W(j,:)\|^2 \|\mathbf{u}_\ell\|} \middle| W \right\} \quad (3.14)$$

Note that  $\mathbf{x} \mathbf{x}^T \rightarrow \langle \mathbf{x} \mathbf{x}^T \rangle$  when passing under the  $\mathcal{E}\{\cdot | W\}$  operator.

From (3.13), knowing that  $a_j = W(j,:) \mathbf{x}$  then:

$$\frac{dW(j,:)}{dt} = \alpha W(j,:) \mathbf{x} \mathbf{x}^T - \gamma(a_j) W(j,:)$$

then, multiplying by  $\mathbf{u}_\ell$  to the right and knowing that  $\langle \mathbf{x} \mathbf{x}^T \rangle \mathbf{u}_\ell = \lambda_\ell \mathbf{u}_\ell$ , it follows that

$$\begin{aligned} \frac{dW(j,:)}{dt} \mathbf{u}_\ell &= \alpha W(j,:) \mathbf{x} \mathbf{x}^T \mathbf{u}_\ell - \gamma(a_j) W(j,:) \mathbf{u}_\ell \\ &\xrightarrow[\mathcal{E}\{\cdot | W\}]{} \alpha W(j,:) \langle \mathbf{x} \mathbf{x}^T \rangle \mathbf{u}_\ell - \gamma(a_j) W(j,:) \mathbf{u}_\ell = (\alpha \lambda_\ell - \gamma(a_j)) [W(j,:) \mathbf{u}_\ell] \end{aligned} \quad (3.15)$$

The other required term is  $d\|W(j,:)\|/dt$  which again is calculated in similar way as in proof of theorem 3.2.1:

$$\frac{d\|W(j,:)\|}{dt} = \frac{dW(j,:)}{dt} \frac{W(j,:)^T}{\|W(j,:)\|}$$

and, by using (3.13),  $a_j = W(j,:) \mathbf{x}$  and  $W(j,:) W(j,:)^T = \|W(j,:)\|^2$ , then:

$$\begin{aligned} \frac{d\|W(j,:)\|}{dt} &= \alpha \frac{W(j,:) \mathbf{x} \mathbf{x}^T W(j,:)^T}{\|W(j,:)\|} - \gamma(a_j) \frac{W(j,:) W(j,:)^T}{\|W(j,:)\|} \\ &\xrightarrow[\mathcal{E}\{\cdot | W\}]{} \alpha \frac{W(j,:) \langle \mathbf{x} \mathbf{x}^T \rangle W(j,:)^T}{\|W(j,:)\|} - \gamma(a_j) \|W(j,:)\| \end{aligned} \quad (3.16)$$

Replacing (3.15) and (3.16) results back into (3.14) gives:

$$\begin{aligned} \mathcal{E} \left\{ \frac{d \cos \theta_\ell}{dt} \middle| W \right\} &= \mathcal{E} \left\{ \frac{(\alpha \lambda_\ell - \gamma(a_j)) [W(j,:) \mathbf{u}_\ell]}{\|W(j,:)\| \|\mathbf{u}_\ell\|} \right. \\ &\quad \left. - \frac{[W(j,:) \mathbf{u}_\ell] \left( \alpha \frac{W(j,:) \langle \mathbf{x} \mathbf{x}^T \rangle W(j,:)^T}{\|W(j,:)\|} - \gamma(a_j) \|W(j,:)\| \right)}{\|W(j,:)\|^2 \|\mathbf{u}_\ell\|} \middle| W \right\} \end{aligned}$$

and, after simplification, it may be written as:

$$\mathcal{E} \left\{ \frac{d \cos \theta_\ell}{dt} \middle| W \right\} = \alpha \mathcal{E} \left\{ \left( \lambda_\ell - \frac{W(j,:) \langle \mathbf{x} \mathbf{x}^T \rangle W(j,:)^T}{\|W(j,:)\|^2} \right) \frac{[W(j,:) \mathbf{u}_\ell]}{\|W(j,:)\| \|\mathbf{u}_\ell\|} \middle| W \right\}$$

Lets take  $\mathbf{u}_\ell = \mathbf{u}_{\max}$  and the corresponding  $\lambda_\ell = \lambda_{\max}$ . The above formula becomes:

$$\mathcal{E} \left\{ \frac{d \cos \theta_{\max}}{dt} \middle| W \right\} = \alpha \mathcal{E} \left\{ \left( \lambda_{\max} - \frac{W(j,:) \langle \mathbf{x} \mathbf{x}^T \rangle W(j,:)^T}{\|W(j,:)\|^2} \right) \frac{[W(j,:) \mathbf{u}_{\max}]}{\|W(j,:)\| \|\mathbf{u}_{\max}\|} \middle| W \right\}$$

The existence of  $\lim_{t \rightarrow \infty} W$  means that  $\theta_{\max}$  stabilizes in time and thus  $\lim_{t \rightarrow \infty} d \cos \theta_{\max} / dt = 0$  and so is its expected value. As  $W(j,:) \mathbf{u}_{\max} \neq 0$  then:

$$\mathcal{E} \left\{ \lambda_{\max} - \frac{W(j,:) \langle \mathbf{x} \mathbf{x}^T \rangle W(j,:)^T}{\|W(j,:)\|^2} \middle| W \right\} = 0 \Leftrightarrow \mathcal{E} \left\{ \frac{W(j,:) \langle \mathbf{x} \mathbf{x}^T \rangle W(j,:)^T}{\|W(j,:)\|^2} \middle| W \right\} = \lambda_{\max} \quad (3.17)$$

the equality being possible only for  $\lambda_{\max}$ , in accordance to the Rayleigh quotient (See the mathematical appendix).

As the matrix  $\mathbf{x}\mathbf{x}^T$  is symmetrical (and so is  $\langle \mathbf{x}\mathbf{x}^T \rangle$ , i.e.  $\langle \mathbf{x}\mathbf{x}^T \rangle = (\langle \mathbf{x}\mathbf{x}^T \rangle)^T$ ) then an orthogonal set of eigenvectors may be build (see the mathematical appendix). A transformation of coordinates to the system  $\{\mathbf{u}_\ell\}_{\ell=1,N}$  may be performed by using the matrix  $U$  build using the set of eigenvectors as *columns* (and then  $U^T U = I$  as  $\mathbf{u}_\ell^T \mathbf{u}_k = \delta_{\ell k}$ ,  $\delta_{\ell k}$  being the Kronecker symbol). Then  $W(j,:)^T \rightarrow W'(j,:)^T = UW(j,:)^T$ ,  $W(j,:) \rightarrow W'(j,:) = W(j,:)^T U^T$ , also

$$\|W(j,:)\|^2 \rightarrow \|W'(j,:)\|^2 = W(j,:)^T UW(j,:)^T = W(j,:)^T IW(j,:)^T = \|W(j,:)\|^2$$

and  $W'(j,:)$  may be represented as a linear combination of  $\{\mathbf{u}_\ell\}$ :

$$W'(j,:) = \sum_\ell \omega_\ell \mathbf{u}_\ell^T$$

$\omega_\ell$  being the coefficients of the linear combination ( $\mathbf{u}_\ell$  appear transposed because  $W(j,:)$  is a *row* matrix).

Knowing that  $U^T \langle \mathbf{x}\mathbf{x}^T \rangle U$  is a diagonal matrix with eigenvalues on the main diagonal (and all others being zero, see again the mathematical appendix) and using again the orthogonality of  $\{\mathbf{u}_\ell\}$  (i.e.  $\mathbf{u}_\ell^T \mathbf{u}_k = \delta_{\ell k}$ ) then:

$$W'(j,:) \langle \mathbf{x}\mathbf{x}^T \rangle W'(j,:)^T = \sum_\ell \lambda_\ell \omega_\ell^2 \quad \text{and} \quad \|W'(j,:)\|^2 = \sum_\ell \omega_\ell^2$$

Replacing back into (3.17) (with  $W'(j,:)$  replacing  $W(j,:)$ ) gives:

$$\mathcal{E} \left\{ \frac{\sum_\ell \lambda_\ell \omega_\ell^2}{\sum_\ell \omega_\ell^2} \middle| W \right\} = \lambda_{\max}$$

which may happen only if all  $\omega_\ell \rightarrow 0$  except the one  $\omega_{\max}$  corresponding to  $\mathbf{u}_{\max}$ , i.e.  $\lim_{t \rightarrow \infty} W(j,:) = \omega_{\max} \mathbf{u}_{\max}^T \propto \mathbf{u}_{\max}^T$ .  $\square$

At first glance the condition  $W \mathbf{u}_{\max} \neq 0$  at all  $t$ , met in theorem 3.2.2 seems to be very hard. In fact it is, but in practice have a smaller importance and this deserves a discussion.

#### ❖ $W_0, \mathcal{P}_{\perp \mathbf{u}_{\max}}$

First, the initial value of  $W$ , let  $W(0) \equiv W_0$  be that one, should be chosen such that  $W_0 \mathbf{u}_{\max} \neq 0$ . But  $W(j,:) \mathbf{u}_{\max} \neq 0$  means that  $W(j,:) \notin \mathcal{P}_{\perp \mathbf{u}_{\max}}$ , i.e.  $W(j,:)$  is *not* contained in a hyperplane  $\mathcal{P}_{\perp \mathbf{u}_{\max}}$  perpendicular on  $\mathbf{u}_{\max}$  (in  $\mathbb{R}^N$ ). Even a random selection on  $W_0$  would have good chances to stand this condition, even more so as  $\mathbf{u}_{\max}$  is seldom known exactly in practice, being dependent on the stochastic input variable  $\mathbf{x}$  (an exact knowledge of  $\mathbf{u}_{\max}$  would mean a knowledge of an infinite series of  $\mathbf{x}(t)$ ).

Second, theorem 3.2.2 says that, *statistically*,  $W(j,:)$  vectors will move towards either  $\mathbf{u}_{\max}$  or  $-\mathbf{u}_{\max}$  depending upon what side of  $\mathcal{P}_{\perp \mathbf{u}_{\max}}$  is  $W_0(j,:)$ , i.e. *away from*  $\mathcal{P}_{\perp \mathbf{u}_{\max}}$ . However the proof is statistical in nature and there is a *small but finite probability* that, at same  $t$ ,  $W(j,:)(t)$  falls into  $\mathcal{P}_{\perp \mathbf{u}_{\max}}$ . What happens then, tell us (3.15): as  $W(j,:) \mathbf{u}_{\max} = 0$  then  $\frac{dW(j,:)}{dt} \mathbf{u}_{\max} = 0$  and this means that all further changes in  $W(j,:)$  are contained in  $\mathcal{P}_{\perp \mathbf{u}_{\max}}$ , i.e.  $W(j,:)$  becomes trapped in  $\mathcal{P}_{\perp \mathbf{u}_{\max}}$ .

The conclusion is then that the condition  $W \mathbf{u}_{\max} \neq 0, \forall t$ , may be neglected, with the remark that there is a small probability that some  $W(j,:)$  weight vectors may become trapped and then learning will be incomplete.

## 3.3 Network Dynamics

### 3.3.1 Network Running Function

As previously discussed,  $\mathbf{x}, W(j, :) \in \mathbb{R}^N$ , the weight space.

For each input vector  $\mathbf{x}$ , the neuron  $k$  for which:

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

is declared winner, i.e. the one with for which the associated weight vector  $W(k, :)$  is closest to  $\mathbf{x}$  (in weight space). The winner is used to decide which weights get changed using the current input vector  $\mathbf{x}$ . All and only the neurons found into the winner's neighborhood participate to learning, i.e. will have their weights changed/adapted. All other weights remain unchanged at this stage; later a new input vector may change the winner and thus the area of change.

#### Remarks:

- $\|\mathbf{x} - W(j, :)^T\|$  is the (mathematical) distance between vectors  $\mathbf{x}$  and  $W(j, :)$ .  
This distance is user definable but most used is the Euclidean  $\sqrt{\sum_{i=1}^N (x_i - w_{ji})^2}$ .
- As the learning of the network is unsupervised, i.e. there are *no targets*.
- If the input vectors  $\mathbf{x}$  and weight vectors  $\{W(j, :)\}_{j=1, K}$  are normalized  $\|\mathbf{x}\| = \|W(j, :)\|$  (to the same value, not necessary 1), e.g. in an Euclidean space:

$$\sqrt{\sum_{i=1}^N x_i^2} = \sqrt{\sum_{i=1}^N w_{ji}^2} \quad , \quad j = \overline{1, K}$$

i.e.  $\mathbf{x}(t)$  and  $W(j, :)$  are points on a hyper-sphere in  $\mathbb{R}^N$ , then the dot vector product can be used to find the matching. The winner neuron is that  $k$  one for which:

$$W(k, :) \mathbf{x} = \max_{j=i, K} W(j, :) \mathbf{x}$$

i.e. the winner is that neuron for which the weight vector  $W(k, :)$  points to the closest direction to that one to which points the input vector  $\mathbf{x}$ .

This operation is a little faster as it skips a subtraction operation of type:  $\mathbf{x} - W(j, :)^T$ , however it requires normalization of  $\mathbf{x}$  and  $W(j, :)$  which is not always desirable in practice.

### 3.3.2 Network learning function

The learning process is an unsupervised one. Time is considered to be discrete  $t = 1, 2, \dots$ . The weights are time dependent  $W = W(t)$ . The learning network is feed with data  $\mathbf{x}(t)$ .

At time  $t = 0$  the weights are initialized with (small) random values. The weights at time  $t$  are updated as follows:

- ① For  $\mathbf{x}(t)$  find the winning (output) neuron  $k$ . See section 3.3.1.
- ② Update weights according to the model chosen, see section 3.2 for a selection of learning models:

$$dW = \left( \frac{dW}{dt} \right) \cdot dt$$

which in discrete time approximation ( $dt \rightarrow t - (t - 1) = 1$ ) becomes:

$$\Delta W = W(t) - W(t - 1) = \frac{dW}{dt} \quad (3.18)$$

### 3.3.3 Initialization and Stop condition

Weights are initialized (in practice) with small random values (normalized or not) and the adjusting process continue by iteration.

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

- ① choosing a fixed number of steps  $t = \overline{1, T}$ .
- ② the learning process continue until the adjusting quantity  $\Delta w_{ji} = w_{ji}(t + 1) - w_{ji}(t)$  falls under some specified value, i.e.  $\Delta w_{ji} \leq \varepsilon$ , where  $\varepsilon$  is the threshold.

### 3.3.4 Remarks

#### *The mapping feature of Kohonen networks*

SOM

Due to the fact that the Kohonen algorithm “moves” the weights vectors towards the 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. **The mapping occurs in the weight space**. See section 3.5.2 for an example. For this reason Kohonen networks are also called **self ordering maps** or SOM.

#### *Activation Function*

Note that the activation function, as well as the neuronal output is irrelevant *to the learning process*.

#### *Incomplete Learning*

Even if the learning is unsupervised, in fact, a poor choice of learning parameters may lead to an incomplete learning (so, in fact, a full successful learning is “supervised” at a “highest” level). See section 3.5.2 and figure 3.6 for an example of an incomplete learning.

## 3.4 The algorithm

1. For all neurons in output layer: initialize weights with random values.
2. If working with normalized vectors then normalize the weights.
3. Choose a model — type of neuronal learning. See section 3.2 for some examples.

4. Choose a model for the neuronal neighborhood — lateral feedback function. See also section 3.5 for some examples.
5. Choose a *stop* condition. See section 3.3.3.
6. Knowing the learning model, the neuronal neighborhood function and the stop condition, build the final equation giving the weight adaptation formula. See section 3.5.1 for an example of how to do this.
7. In discrete time approximation repeat the following steps till the *stop* condition is met:
  - (a) Get the input vector  $\mathbf{x}(t)$ .
  - (b) For all neurons  $j$  in output layer, find the “winner” — the neuron  $k$  for which:

$$\|\mathbf{x}(t) - W(k, :)^T\| = \min_{j=1, K} \|\mathbf{x}(t) - W(j, :)^T\|$$

or, if working with normalized vectors:

$$W(k, :) \mathbf{x} = \max_{j=1, K} W(j, :) \mathbf{x}$$

- (c) Knowing the winner, change weights by using the adaptation formula built at step 6.

## 3.5 Applications

### 3.5.1 The Trivial Model with Forgetting Function

This application is without practical value but it shows how to build a weight adaptation formula. It also gives some examples of neuronal neighborhood models. The topics discussed here apply to many types of Kohonen networks.

Let choose the trivial equation (3.1) as learning model:

$$\frac{dW}{dt} = \alpha \hat{\mathbf{1}} \mathbf{x}^T - \beta W \quad (3.19)$$

Next let consider  $h(k, j)$  the function modelling neuronal neighborhood, i.e. the lateral feedback. This function should be of “mexican hat” type, i.e.

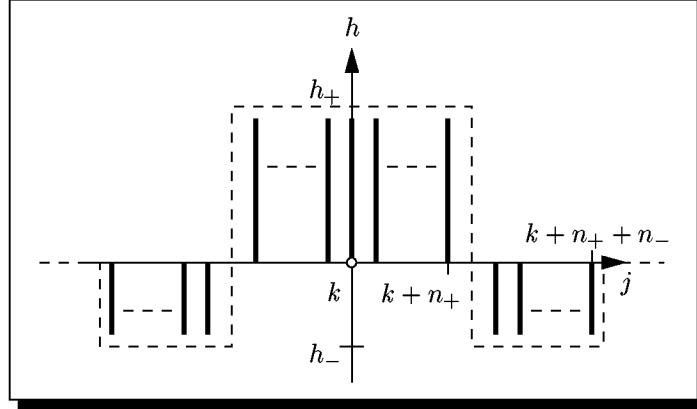
$$h(k, j) \begin{cases} > 0 & \text{for } j \text{ relatively close to } k \\ \leq 0 & \text{for } j \text{ far, but not too much, from } k \\ = 0 & \text{for } j \text{ far away from } k \end{cases}$$

❖  $h$   
neuronal  
neighborhood  
lateral feedback

This function will be generally a function of “distance” between  $k$  and  $j$ , the distance being user definable. Considering  $x_k$  and  $x_j$  the “coordinates” then  $h = h(|x_j - x_k|)$ . Let  $\mathbf{x}_{(K)}^T =$

$(x_1 \cdots x_K)$  be the vector containing the neuron coordinates, then  $h(|\mathbf{x}_{(K)} - x_{(K)k} \hat{\mathbf{1}}|)$  will give the vector containing adaptation height around winner  $k$ , for the whole network.

❖  $\mathbf{x}_{(K)}$



**Figure 3.3:** The simple lateral feedback function.

To account for the neuronal neighborhood model chosen, equation (3.19) have to be changed to:

$$\frac{dW}{dt} = [h(|\mathbf{x}_{(K)} - \mathbf{x}_{(K)k}\hat{\mathbf{1}}|) \hat{\mathbf{1}}^T] \odot [\alpha\hat{\mathbf{1}}\mathbf{x}^T - \beta W] \quad (3.20)$$

Note that the elements of  $h(|\mathbf{x}_{(K)} - \mathbf{x}_{(K)k}\hat{\mathbf{1}}|)$  corresponding to neurons outside neuronal neighborhood of winner are zero and thus, for these neurons:  $\frac{dW(j,:)}{dt} = 0$ , i.e. their weights remain unchanged for current  $\mathbf{x}$ .

Various neuronal neighborhood may be of the form:

- Simple lateral feedback function:

$$h(j, k) = \begin{cases} h_+ & \text{for } j \in \{k - n_+, \dots, k, k + n_+\} \text{ (positive feedback)} \\ -h_- & \text{for } j \in \{k - n_+ - n_-, \dots, k - n_+ - 1\} \cup \\ & \quad \{k + n_+ + 1, \dots, k + n_+ + n_-\} \\ & \text{(negative feedback)} \\ 0 & \text{in rest} \end{cases}$$

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

- Exponential lateral feedback function:

$$h(j, k) = h_+ e^{-(k-j)^2}, \quad \text{for } |j - k| \leq n$$

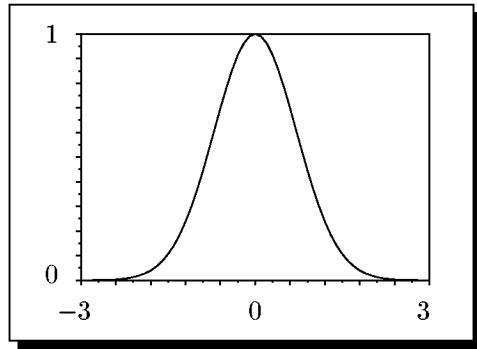
where  $h_+ > 0$ ,  $h_+ = \text{const.}$  defines the positive feedback and there is no negative one and  $n > 0$ ,  $n = \text{const.}$  defines the neuronal neighborhood. See figure 3.4 on the facing page.

stop condition

❖  $\tau$

Finally the stop condition may be implemented by multiplying the right side of equation (3.20) by a function  $\tau(t)$  with the property  $\lim_{t \rightarrow \infty} \tau(t) = 0$ . This way (3.20) becomes:

$$\frac{dW}{dt} = \tau(t) [h(|\mathbf{x}_{(K)} - \mathbf{x}_{(K)k}\hat{\mathbf{1}}|) \hat{\mathbf{1}}^T] \odot [\alpha\hat{\mathbf{1}}\mathbf{x}^T - \beta W] \quad (3.21)$$



**Figure 3.4:** The exponential lateral feedback function  $h(x) = e^{-x^2}$ .

The convergence of  $\tau(t)$  to zero ensures that  $\lim_{t \rightarrow \infty} \frac{dW}{dt} = 0$  and thus the weight adaptation, i.e. learning process, stops eventually.

Various stop functions may be of the form:

- Geometrical progression function:

$$\tau(t) = \tau_{\text{init}} \tau_{\text{ratio}}^t, \quad \tau_{\text{init}}, \tau_{\text{ratio}} \in (0, 1), \quad \tau_{\text{init}}, \tau_{\text{ratio}} = \text{const.}$$

where  $\tau_{\text{init}}$  and  $\tau_{\text{ratio}}$  are the initial/ratio values.

- Exponential function:

$$\tau(t) = \tau_{\text{init}} e^{-f(t)}$$

where  $f(t) : \mathbb{N} \rightarrow [0, \infty)$  is a monotone increasing function.

Note that for  $f(t) = t$  and  $t \in \mathbb{N}^+$  this function is a geometrical progression.

In discrete time approximation, using (3.21) into (3.18) gives:

$$W(t+1) = W(t) + \tau(t) \left[ h(|\mathbf{x}_{(K)} - \mathbf{x}_{(K)k} \hat{\mathbf{1}}|) \hat{\mathbf{1}}^T \right] \odot \left[ \alpha \hat{\mathbf{1}} \mathbf{x}^T - \beta W \right] \quad (3.22)$$

(note also that the winner  $k$  is also time dependent, i.e.  $\mathbf{x}_{(K)k} = \mathbf{x}_{(K)k}(\mathbf{x}(t))$ ).

### Remarks:

- The above considerations are easily extensible to multidimensional Kohonen networks. E.g. for a bidimensional  $K \times K$  layer, considering the winner  $k$ , the interneuronal Euclidean distance, squared, will be:

$$(\mathbf{x}_{(K)} - \mathbf{x}_{(K)k} \hat{\mathbf{1}})^{\odot 2} + (\mathbf{y}_{(K)} - \mathbf{y}_{(K)k} \hat{\mathbf{1}})^{\odot 2}$$

$\mathbf{y}_{(K)}$  holding the second coordinate. Of course  $h$  also changes to:

$$h = h((\mathbf{x}_{(K)} - \mathbf{x}_{(K)k} \hat{\mathbf{1}})^{\odot 2} + (\mathbf{y}_{(K)} - \mathbf{y}_{(K)k} \hat{\mathbf{1}})^{\odot 2})$$

### 3.5.2 Square mapping

This example shows the mapping feature of a Kohonen network, a way (among many other possibilities) to build multidimensional networks and possible defects in learning process.

Let be a Kohonen network with 2 inputs and  $8 \times 8$  neurons (bidimensional output layer with  $K = 8$ ). The trivial equation model, in discrete time approximation, will be used here, see section 3.5.1.

As the network is bidimensional will do the following changes:

❖  $X_{(K)}$

- The “ $x$ ” coordinates of neurons, in terms of interneuronal distances, are kept in a matrix  $X_{(K)}$  of the form:

$$X_{(K)} = \widehat{\mathbf{1}} \begin{pmatrix} 1 & 2 & \cdots & 8 \end{pmatrix} = \begin{pmatrix} 1 & 2 & \cdots & 8 \\ 1 & 2 & \cdots & 8 \\ \dots & \dots & \dots & \dots \\ 1 & 2 & \cdots & 8 \end{pmatrix}$$

❖  $Y_{(K)}$

while the “ $y$ ” coordinates are:

$$Y_{(K)} = \begin{pmatrix} 1 \\ 2 \\ \vdots \\ 8 \end{pmatrix} \widehat{\mathbf{1}}^T = \begin{pmatrix} 1 & 1 & \cdots & 1 \\ 2 & 2 & \cdots & 2 \\ \dots & \dots & \dots & \dots \\ 8 & 8 & \cdots & 8 \end{pmatrix}$$

and it becomes immediately that the layout of network is (coordinates  $(x, y)$  are in parentheses):

$$\begin{matrix} (1, 1) & \cdots & (8, 1) \\ \vdots & & \vdots \\ (1, 8) & \cdots & (8, 8) \end{matrix}$$

A particular neuron will be then identified by two numbers  $(j_x, j_y)$ .

❖  $D(k)$

- Considering  $(x_{(K)k_x k_y}, y_{(K)k_x k_y})$  the coordinates of winner then the interneuronal squared distances may be kept in a matrix  $D(k_x, k_y)$  as:

$$D(k_x, k_y) = (X_{(K)} - x_{(K)k_x k_y} \widetilde{\mathbf{1}})^{\odot 2} + (Y_{(K)} - y_{(K)k_x k_y} \widetilde{\mathbf{1}})^{\odot 2}$$

- The lateral feedback function is

$$h((k_x, k_y), t) = h_+ \exp \left( -\frac{D(k_x, k_y)}{(d_{\text{init}} d_{\text{rate}})^2} \right)$$

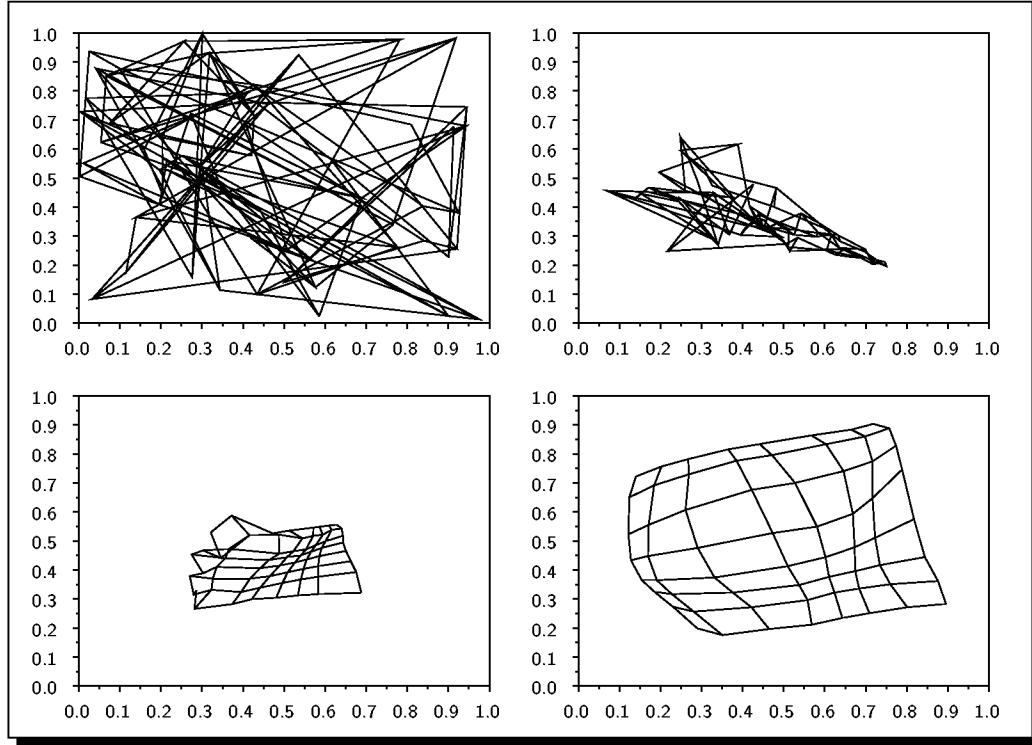
where  $h_+ = 0.6$ ,  $d_{\text{init}} = 5$  and  $d_{\text{rate}} = 0.993$ .

- The stop function is

$$\tau(t) = \tau_{\text{init}} \tau_{\text{rate}}^t$$

where  $\tau_{\text{init}} = 1$  and  $\tau_{\text{rate}} = 0.993$ .

- The weights are kept in two matrices  $W_1$  and  $W_2$  corresponding to the two components of input vector. The weight vector associated to a particular neuron  $(j_x, j_y)$  is  $(w_{1j_x j_y}, w_{2j_x j_y})$ .



**Figure 3.5:** Mapping of the  $[0, 1] \times [0, 1]$  square by an  $8 \times 8$  network.

These are snapshots taken at  $t = 0$  (upper-left),  $t = 7$  (upper-right),  $t = 15$  (lower-left) and  $t = 120$  (lower-right).

- The constants of trivial equation are taken to be  $\alpha = 1$  and  $\beta = 1$ .

Then the general weights updating formulas are:

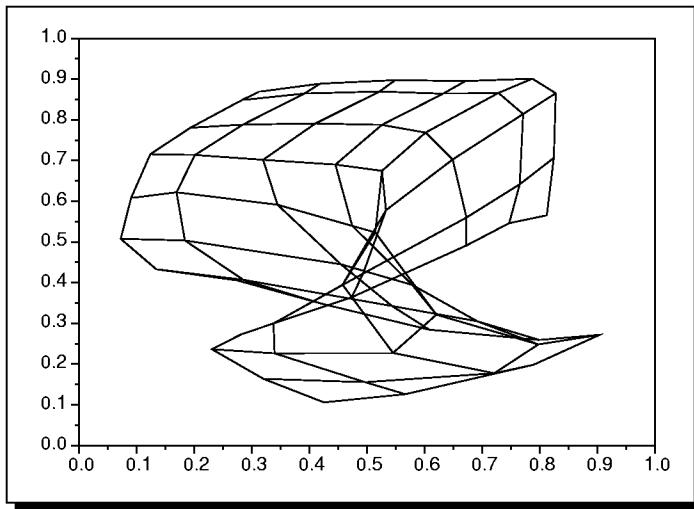
$$W_1(t+1) = W_1(t) + \tau(t) h((k_x, k_y), t) \odot [x_1(t)\tilde{1} - W_1(t)]$$

$$W_2(t+1) = W_2(t) + \tau(t) h((k_x, k_y), t) \odot [x_2(t)\tilde{1} - W_2(t)]$$

The pair of weights associated with each neuron  $(w_{1j_xj_y}, w_{2j_xj_y})$  may also be represented as points in the  $[0, 1] \times [0, 1]$  square. A successful learning will try to cover as much as possible of the area. See figure 3.5 where the evolution of training is shown from  $t = 0$  (upper-left) to final stage (down-right). Lines are drawn between closest neighbors (network topology wise). The weights are the points at intersections.

### Remarks:

- ➔ Even if the learning is unsupervised, in fact, a poor choice of learning parameters ( $\alpha$ ,  $h_+$ , e.t.c.) may lead to an incomplete learning. See figure 3.6 on the following page: small values of feedback function at the beginning of learning makes the network to be unable to “deploy” itself fast enough leading to the



**Figure 3.6:** Incomplete learning of the  $[0, 1] \times [0, 1]$  square by an  $8 \times 8$  network.

appearance of a ‘twist’ in the mapping. The network was the same as the one used to generate figure 3.5 including same inputs and same weights initial values. The only parameters changed were:  $h_+ = 0.35$ ,  $d_{\text{init}} = 3.5$ ,  $d_{\text{rate}} = 0.99$  and  $\tau_{\text{rate}} = 0.9999$ .

## CHAPTER 4

# The BAM/Hopfield Memory

This network illustrate an associative memory. Unlike the classical von Neumann systems where there is no link between memory address and its contents, in ANN, part of information is used to retrieve the rest associated with it. This kind of memory is named *associative*.

## 4.1 Associative Memory

**Definition 4.1.1.** Let be  $P$  pairs of vectors  $\{(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_P, \mathbf{y}_P)\}$  with  $\mathbf{x}_p \in \mathbb{R}^N$  and  $\mathbf{y}_p \in \mathbb{R}^K$ ,  $N, K, P \in \mathbb{N}^+$  called exemplars.  $\diamond \mathbf{x}_p, \mathbf{y}_p, P$

Then the mapping  $\mathcal{M} : \mathbb{R}^N \rightarrow \mathbb{R}^K$  is said to implement an **heteroassociative memory** if:

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

$$\mathcal{M}(\mathbf{x}) = \mathbf{y}_p \quad \forall \mathbf{x} \text{ such that } \|\mathbf{x} - \mathbf{x}_p\| < \|\mathbf{x} - \mathbf{x}_\ell\| \quad \forall \ell = \overline{1, P}, \ell \neq p$$

heteroassociative  
memory

**Definition 4.1.2.** Let be  $P$  pairs of vectors  $\{(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_P, \mathbf{y}_P)\}$  with  $\mathbf{x}_p \in \mathbb{R}^N$  and  $\mathbf{y}_p \in \mathbb{R}^K$ ,  $N, K, P \in \mathbb{N}^+$  called exemplars.

Then the mapping  $\mathcal{M} : \mathbb{R}^N \rightarrow \mathbb{R}^K$  is said to implement an **interpolative associative memory** if:

interpolative asso-  
ciative memory

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

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

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

---

<sup>4.1</sup>See [FS92] pp. 130–131.

The interpolative associative memory may be build from a set of orthonormal set of exemplars  $\{\mathbf{x}_p\}$ . The  $\mathcal{M}$  function is then defined as:

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

Kronecker symbol

*Proof.* Orthogonality of  $\{\mathbf{x}_p\}$  means that  $\mathbf{x}_p^T \mathbf{x}_\ell = \delta_{p\ell}$ , where  $\delta_{p\ell}$  is the Kronecker symbol.

From equation 4.1:

$$\mathcal{M}(\mathbf{x}_\ell) = \left( \sum_{p=1}^P \mathbf{y}_p \mathbf{x}_p^T \right) \mathbf{x}_\ell = \sum_{p=1}^P \mathbf{y}_p \mathbf{x}_p^T \mathbf{x}_\ell = \sum_{p=1}^P \mathbf{y}_p \delta_{p\ell} = \mathbf{y}_\ell$$

and for some  $\mathbf{x} = \mathbf{x}_\ell + \mathbf{d}$ :

$$\mathcal{M}(\mathbf{x}) = \mathcal{M}(\mathbf{x}_\ell + \mathbf{d}) = \mathbf{y}_\ell + \mathbf{e} \quad \text{where } \mathbf{e} = \sum_{p=1}^P \mathbf{y}_p \mathbf{x}_p^T \mathbf{d}$$

(obviously  $\mathcal{M}(\mathbf{x}_\ell + \mathbf{d}) = \mathcal{M}(\mathbf{x}_\ell) + \mathcal{M}(\mathbf{d})$ ).  $\square$

**Definition 4.1.3.** Let be a set of  $P$  vectors  $\{\mathbf{x}_1, \dots, \mathbf{x}_P\}$  with  $\mathbf{x}_p \in \mathbb{R}^N$  and  $N, P \in \mathbb{N}^+$  called exemplars.

autoassociative  
memory

Then, the mapping  $\mathcal{M} : \mathbb{R}^N \rightarrow \mathbb{R}^N$  is said to implement an **autoassociative memory** if:

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

$$\mathcal{M}(\mathbf{x}) = \mathbf{x}_p \quad \forall \mathbf{x} \text{ such that } \|\mathbf{x} - \mathbf{x}_p\| < \|\mathbf{x} - \mathbf{x}_\ell\| \quad \forall \ell = \overline{1, P}, \ell \neq p$$

In general  $\mathbf{x}$  will be used to denote the input vector and  $\mathbf{y}$  the output vector into an associative memory.

## 4.2 The BAM Architecture

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

The figure 4.1 on the facing page 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).

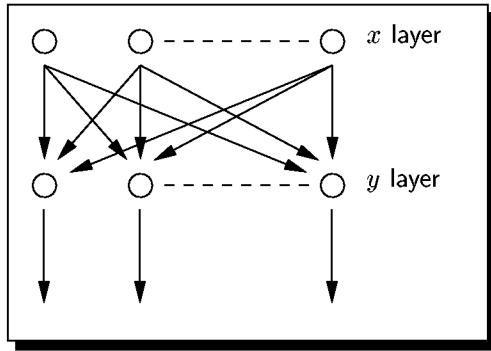
Considering the weight matrix  $W$  then the network output is  $\mathbf{y} = W\mathbf{x}$ , i.e. the activation function is identity  $f(\mathbf{x}) = \mathbf{x}$ .

According to (4.1), the weight matrix may be build using a set of orthogonal  $\{\mathbf{x}_p\}$  and the associated  $\{\mathbf{y}_p\}$ , as:

$$W = \sum_{p=1}^P \mathbf{y}_p \mathbf{x}_p^T \quad (4.2)$$

---

<sup>4.2</sup>See [FS92] pp. 131–132.



**Figure 4.1:** The BAM network structure.

If the  $\{\mathbf{y}_p\}$  are also orthogonal then the network is reversible. Considering  $y$  layer as input then:

$$\mathbf{x} = W^T \mathbf{y}$$

*Proof.* As for two matrices it is true that  $(AB)^T = B^T A^T$  then from (4.2):

$$W^T = \sum_{p=1}^P \mathbf{x}_p \mathbf{y}_p^T$$

By using the orthogonality property  $\mathbf{y}_p^T \mathbf{y}_\ell = \delta_{p\ell}$ , the proof is very similar to the one for (4.1) (replacing  $x \leftrightarrow y$ ).  $\square$



#### Remarks:

- According to (4.2) the weights can be computed exactly (within the limitations of rounding errors).
- The activation function of neurons was assumed to be the identity:  $f(\mathbf{a}) = \mathbf{a}$ . Because the output function of a neuron should be bounded so should be the data network is working with (i.e.  $\mathbf{x}$  and  $\mathbf{y}$  vectors).
- The network can be used as autoassociative memory considering  $\mathbf{x} \equiv \mathbf{y}$ . The weight matrix becomes:

$$W = \sum_{p=1}^P \mathbf{x}_p \mathbf{x}_p^T$$

## 4.3 BAM Dynamics

### 4.3.1 Network Running

The BAM functionality differ from others by the fact that *weights are not adjusted* during a training period but calculated from the start from the set of vectors to be stored  $\{\mathbf{x}_p, \mathbf{y}_p\}_{p=1,P}$ .

<sup>4.3.1</sup>See [FS92] pp. 132–136.

The procedure is developed for vectors belonging to Hamming space<sup>1</sup>  $\mathbf{H}$ . Due to the fact that most information can 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 components having values either +1 or -1). A transition to and from binary vectors (having component values either 0 or 1) can be easily done using the following relation:  $\mathbf{x} = 2\tilde{\mathbf{x}} - \hat{\mathbf{1}}$  where  $\mathbf{x}$  is a bipolar (Hamming) vector and  $\tilde{\mathbf{x}}$  is a binary vector.

From a set of vectors  $\{\mathbf{x}_p, \mathbf{y}_p\}$  the weight matrix is calculated by using (4.2). Both  $\{\mathbf{x}_p\}$  and  $\{\mathbf{y}_p\}$  have to be orthogonal because the network works in both directions.

The procedure works in discrete time approximation. An initial  $\mathbf{x}(0)$  is applied to the input. The goal is to retrieve a vector  $\mathbf{y}_\ell$  corresponding to the closest  $\mathbf{x}_\ell$  to  $\mathbf{x}(0)$ , where  $\{\mathbf{x}_\ell, \mathbf{y}_\ell\}$  are from the exemplars set (stored into the net, at the initialization time, by the mean of the calculated weight matrix).

The information is propagated forward and back between layers  $x$  and  $y$  till a stable state is reached and subsequently a pair  $\{\mathbf{x}_\ell, \mathbf{y}_\ell\}$  belonging to the set of exemplars is found (at the output of  $x$  respectively  $y$  layers).

The procedure is as follows:

- At  $t = 0$  the  $\mathbf{x}(0)$  is applied to the net and the corresponding  $\mathbf{y}(0) = W\mathbf{x}(0)$  is calculated.
- The outputs of  $x$  and  $y$  layers are propagated back and forward, till a stable state is reached, according to the formulas (for convenience  $[W(:, i)]^T \equiv W(:, i)^T$ ):

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

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

Note that the activation function  $f$  is *not* the identity.

In matrix notation the formulas may be written as:

$$\begin{aligned} \mathbf{x}(t+1) &= \text{sign}(W^T \mathbf{y}(t)) + |\text{sign}(W^T \mathbf{y}(t))|^C \odot \mathbf{x}(t) \\ \mathbf{y}(t+1) &= \text{sign}(W \mathbf{x}(t+1)) + |\text{sign}(W \mathbf{x}(t+1))|^C \odot \mathbf{y}(t) \end{aligned} \tag{4.3}$$

and the stable condition means:  $\text{sign}(W^T \mathbf{y}(t)) = \text{sign}(W \mathbf{x}(t+1)) = \hat{\mathbf{0}}$ .

*Proof.*  $\text{sign}(W^T \mathbf{y}(t))$  gives the correct ( $\pm 1$ ) values of  $\mathbf{x}(t+1)$  for the changing components and make  $x_i(t+1) = 0$  if  $W(:, i)^T \mathbf{y} = 0$ .

The vector  $|\text{sign}(W \mathbf{x}(t+1))|^C$  have 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).

---

<sup>1</sup>See math appendix.

The proof for second formula is similar.  $\square$

Convergence of the process is ensured by theorem 4.3.1.

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

$$\begin{aligned}\mathbf{y}(t+1) &= \text{sign}(W\mathbf{x}(t)) + |\text{sign}(W\mathbf{x}(t))|^C \odot \mathbf{y}(t) \\ \mathbf{x}(t+1) &= \text{sign}(W^T \mathbf{y}(t+1)) + |\text{sign}(W^T \mathbf{y}(t+1))|^C \odot \mathbf{x}(t)\end{aligned}\tag{4.4}$$

### 4.3.2 The BAM Energy Function

**Definition 4.3.1.** The following function:

$$E(\mathbf{x}, \mathbf{y}) = -\mathbf{y}^T W \mathbf{x} \quad \begin{matrix} \diamond E \\ \text{BAM energy} \\ \text{function} \end{matrix} \tag{4.5}$$

is called **BAM energy function**<sup>2</sup>.

**Theorem 4.3.1.** The **BAM energy function** have the following properties:

1. Any change in  $\mathbf{x}$  or  $\mathbf{y}$  during BAM running results in a decrease in  $E$  i.e.:

$$E_{t+1}(\mathbf{x}(t+1), \mathbf{y}(t+1)) \leq E_t(\mathbf{x}(t), \mathbf{y}(t))$$

2.  $E$  is bounded below by  $E_{\min} = -\sum_{j,i} |w_{ji}|$ .

3. When  $E$  changes it must change by an finite amount, i.e.  $\Delta E = E_{t+1} - E_t$  is finite.

*Proof.* 1. Let consider that just one component  $k$  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):

$$\begin{aligned}\Delta E &= E_{t+1} - E_t = \\ &= \left( -\sum_{i=1}^N y_k(t+1) w_{ki} x_i - \sum_{\substack{j=1 \\ j \neq k}}^K \sum_{i=1}^N y_j w_{ji} x_i \right) - \left( -\sum_{i=1}^N y_k(t) w_{ki} x_i - \sum_{\substack{j=1 \\ j \neq k}}^K \sum_{i=1}^N y_j w_{ji} x_i \right) \\ &= [y_k(t) - y_k(t+1)] \sum_{i=1}^N w_{ki} x_i = [y_k(t) - y_k(t+1)] W(k,:) \mathbf{x}\end{aligned}$$

According to the updating procedure, see section 4.3.1:

$$y_k(t+1) = \begin{cases} +1 & \text{if } W(k,:) \mathbf{x} > 0 \\ y_k(t) & \text{if } W(k,:) \mathbf{x} = 0 \\ -1 & \text{if } W(k,:) \mathbf{x} < 0 \end{cases}$$

As it was assumed that  $y_k$  changes then there are two cases

- $y_k(t) = +1$  and it changes to  $y_k(t+1) = -1$ . Then  $y_k(t) - y_k(t+1) > 0$  and  $W(k,:) \mathbf{x} < 0$  (according to the algorithm) so  $\Delta E < 0$ .
- $y_k(t) = -1$  and it changes to  $y_k(t+1) = +1$ . Analogous the preceding case:  $\Delta E < 0$ .

<sup>4.3.2</sup>See [FS92] pp. 136–141.

<sup>2</sup>This is the Liapunov function for BAM. All state change, with respect to time ( $\mathbf{x} = \mathbf{x}(t)$  and  $\mathbf{y} = \mathbf{y}(t)$ ) involves a *decrease* in the value of the function.

If more than one term is changing then  $\Delta E$  is of the form:

$$\Delta E = E_{t+1} - E_t = \sum_{k=1}^K \Delta y_k W(k,:) \mathbf{x} < 0$$

which represents a sum of negative terms.

A similar discussion may be performed for an  $\mathbf{x}$  change.

2. The  $\{y_j\}_{j=1,K}$  and  $\{x_i\}_{i=1,N}$  have all values either +1 or -1.

The lowest possible value for  $E$  is obtained when all sum terms  $y_j w_{ji} x_i$  (see (4.5)) are positive.

$$E_{\min} = - \sum_{j,i} |y_j w_{ji} x_i| = - \sum_{j,i} |y_j| |w_{ji}| |x_i| = - \sum_{j,i} |w_{ji}|$$

3. The energy function decreases, it doesn't increase, so  $\Delta E \neq +\infty$ . On the other hand the energy function is limited on the low end (according to the second part of the theorem) so it cannot decrease by an infinite amount:  $\Delta E \neq -\infty$ .

Also the value of  $\Delta E$  can't be infinitesimally small resulting into an infinite amount of time before it reaches its minimum. The minimum amount by which  $E$  may change is that  $k$  for which  $W(k,:) \mathbf{x}$  is minimum and occurs when  $y_k$  is changing; the minimum amount being:

$$\Delta E = -2|W(k,:) \mathbf{x}|$$

because  $|y_k(t) - y_k(t+1)| = 2$ .  $\square$

**Proposition 4.3.1.** *If the input pattern  $\mathbf{x}_\ell$  is exactly one of stored  $\{\mathbf{x}_p\}$  then the corresponding  $\mathbf{y}_\ell$  is retrieved.*

*Proof.* Theorem 4.3.1 ensures convergence of the process.

According to the procedure, eventually a vector  $\mathbf{y} = \text{sign}(W \mathbf{x}_\ell)$  is obtained. The zeros generated by sign disappear by procedure definition: previous values of  $y_j$  are kept instead. But as  $\mathbf{x}_p \mathbf{x}_\ell = \delta_{p\ell}$  (orthogonality) then:

$$\mathbf{y} = \text{sign}(W \mathbf{x}_\ell) = \text{sign} \left( \sum_{p=1}^P \mathbf{y}_p \mathbf{x}_p^T \mathbf{x}_\ell \right) = \text{sign} \left( \sum_{p=1}^P \mathbf{y}_p \delta_{p\ell} \right) = \text{sign}(\mathbf{y}_\ell) = \mathbf{y}_\ell \quad \square$$



### Remarks:

- ➔ The existence of BAM energy function with the outlined properties ensures that the running process is *convergent* and a for any input vector and solution is reached in *finite time*.
- ➔ If an input vector is slightly different from one of the stored, i.e. there is noise in data, then the corresponding associated vector is eventually retrieved. However the process is not guaranteed. Results may vary depending upon the amount of noise and the saturation of memory (see below).
- ➔ The theoretical upper limit (number of vectors to be stored) of BAM is  $2^{N-1}$  (i.e.  $2^N/2$  because the  $\mathbf{x}$  and  $-\mathbf{x}$  carry the same amount of information due to symmetry). But if the possibility to work with noisy data is sought then the real capacity is much lower. A crosstalk may appear (a different vector from the desired one is retrieved).
- ➔ The Hamming vectors are symmetric with respect to the  $\pm 1$  notation. For this reason an Hamming vector  $\mathbf{u}$  carry the same information as its complement  $\mathbf{x}^C$

and a BAM network stores automatically both, because  $\mathbf{x}^C = -\mathbf{x}$  and  $\mathbf{y}_p = W\mathbf{x}_p$ ,  $\forall p \in \overline{1, P}$  so:

$$\mathbf{y}_p^C = -\mathbf{y}_p = -W\mathbf{x}_p = W(-\mathbf{x}_p) = W\mathbf{x}_p^C$$

such that the same  $W$  matrix is used.

- When trying to retrieve a vector, if the initial one  $\mathbf{x}(0)$  is closer to the complement of an stored one  $\mathbf{x}_p^C$  then the complement pair will be retrieved  $\{\mathbf{x}^C, \mathbf{y}^C\}$  (because both *exemplars* and their complements are stored with equal precedence).

The conclusion is that BAM stores the **direction** of the *exemplar* vectors and not their values.

## 4.4 The BAM Algorithm

### **Network initialization**

The weight matrix is calculated directly from the desired set to be stored:

$$W = \sum_{p=1}^P \mathbf{y}_p \mathbf{x}_p^T$$

Note that there is no learning process. Weights are directly initialized with their final values.

### **Network running forward**

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

Propagate: repeat the following steps

$$\mathbf{x}(t+1) = \text{sign}(W^T \mathbf{y}(t)) + |\text{sign}(W^T \mathbf{y}(t))|^C \odot \mathbf{x}(t)$$

$$\mathbf{y}(t+1) = \text{sign}(W \mathbf{x}(t+1)) + |\text{sign}(W \mathbf{x}(t+1))|^C \odot \mathbf{y}(t)$$

till network stabilize, i.e.  $\text{sign}(W^T \mathbf{y}(t)) = \text{sign}(W \mathbf{x}(t+1)) = \hat{\mathbf{0}}$ .

Note that in both forward and backward running cases the intermediate vectors  $W\mathbf{x}$  or  $W^T\mathbf{y}$  may not be of Hamming type.

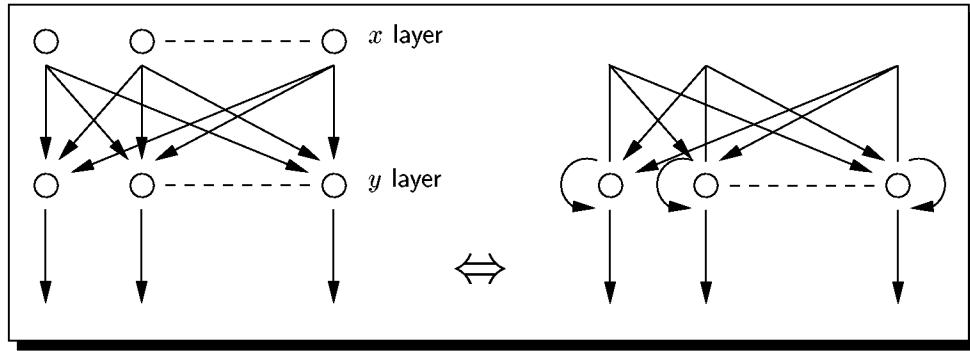
### **Network running backwards**

In the same discrete time approximation, given  $\mathbf{y}(0)$ , calculate  $\mathbf{x}(0) = W^T \mathbf{y}(0)$ . Then propagate using the formulas:

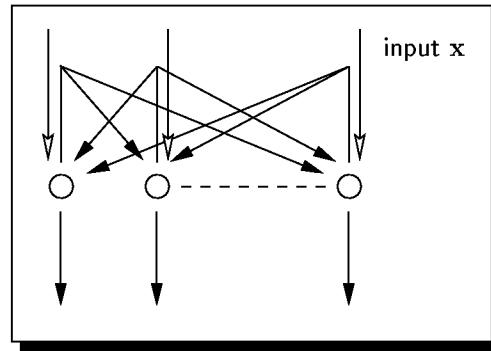
$$\mathbf{y}(t+1) = \text{sign}(W \mathbf{x}(t)) + |\text{sign}(W \mathbf{x}(t))|^C \odot \mathbf{y}(t)$$

$$\mathbf{x}(t+1) = \text{sign}(W^T \mathbf{y}(t+1)) + |\text{sign}(W^T \mathbf{y}(t+1))|^C \odot \mathbf{x}(t)$$

till the network stabilize, i.e.  $\text{sign}(W^T \mathbf{y}(t)) = \text{sign}(W \mathbf{x}(t+1)) = \hat{\mathbf{0}}$ .



**Figure 4.2:** The autoassociative memory structure.



**Figure 4.3:** The Hopfield network structure.

## 4.5 The Hopfield Memory

### 4.5.1 The Discrete Hopfield Memory

Let consider an autoassociative memory. The weight matrix, build from a set  $\{\mathbf{y}_p\}$  is:

$$W = \sum_{p=1}^P \mathbf{y}_p \mathbf{y}_p^T$$

and it's *square* ( $K \times K$ ) and *symmetric* ( $W = W^T$ ).

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.

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  $x$  — see figure 4.3.

❖ x

The differences from autoassociative memory or BAM are as follows:

<sup>4.5.1</sup>See [FS92] pp. 141–144.

- ① The discrete Hopfield memory is working with *binary* vectors rather than bipolar ones — see section 4.3.1 — so here and below the  $\mathbf{y}$  vectors are considered *binary* and so are the input  $\mathbf{x}$  vectors.
- ② The weight matrix is obtained from the following matrix:

$$\sum_{p=1}^P (2\mathbf{y}_p - \hat{\mathbf{1}})(2\mathbf{y}_p - \hat{\mathbf{1}})^T$$

by replacing the diagonal values with 0 (zeroing the diagonal elements of  $W$  is important for a efficient matrix notation<sup>3</sup>).

- ③ The algorithm is similar with the BAM one but the updating formula is:

$$y_j(t+1) = \begin{cases} +1 & \text{if } \sum_{\substack{i=1 \\ i \neq j}}^K w_{ji}y_i + x_j > t_j \\ y_j(t) & \text{if } \sum_{\substack{i=1 \\ i \neq j}}^K w_{ji}y_i + x_j = t_j \\ 0 & \text{if } \sum_{\substack{i=1 \\ i \neq j}}^K w_{ji}y_i + x_j < t_j \end{cases} \quad (4.6)$$

where the  $\{t_j\}_{j=1,K} = \mathbf{t}$  is named the threshold vector. ♦ t

In matrix notation the equation become:

$$A(t) = \text{sign}(W\mathbf{y}(t) + \mathbf{x} - \mathbf{t})$$

$$\mathbf{y}(t+1) = \frac{1}{2} [A(t) + \hat{\mathbf{1}} - |A(t)|^C] + |A(t)|^C \odot \mathbf{y}(t)$$

*Proof.* First as diagonal elements of  $W$  are zero ( $w_{ii} = 0$ ) then  $\sum_{\substack{i=1 \\ i \neq j}}^K w_{ji}y_i = W(j,:) \mathbf{y}$ . Also the elements of  $A(t) + \hat{\mathbf{1}}$  are:

$$\{A(t) + \hat{\mathbf{1}}\}_j = \begin{cases} 2 & \text{if } W(j,:) \mathbf{y} + x_j > t_j \\ 1 & \text{if } W(j,:) \mathbf{y} + x_j = t_j \\ 0 & \text{if } W(j,:) \mathbf{y} + x_j < t_j \end{cases}$$

and the elements of  $|A|^C$  are:

$$\{|A|^C\}_j = \begin{cases} 1 & \text{if } W(j,:) \mathbf{y} + x_j = t_j \\ 0 & \text{otherwise} \end{cases} \quad \square$$

**Definition 4.5.1.** The following function:

♦ E

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

is named the discrete Hopfield memory energy function.

---

<sup>3</sup>This helps towards an efficient simulation implementation as well.

 **Remarks:**

- Comparing to the BAM energy function the discrete Hopfield energy function have 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).

**Theorem 4.5.1.** *The discrete Hopfield energy function have the following properties:*

1. Any change in  $\mathbf{y}$  (during running) results in a decrease in  $E$ :

$$E_{t+1}(\mathbf{y}(t+1)) \leq E_t(\mathbf{y}(t))$$

2.  $E$  is bounded below by:

$$E_{\min} = -\frac{1}{2} \sum_{j,i} |w_{ji}| - K$$

3. When  $E$  changes it must change by an finite amount, i.e.  $\Delta E = E_{t+1} - E_t$  is finite.

*Proof.* 1. Consider that, from  $t$  to  $t + 1$ , just one component of vector  $\mathbf{y}$  changes:  $y_k$ . Then from (4.7):

$$\begin{aligned} \Delta E &= E_{t+1} - E_t = \\ &= 2[y_k(t+1) - y_k(t)] \left( -\frac{1}{2} \sum_{\substack{i=1 \\ i \neq k}}^K w_{ki} y_i \right) - (x_k + t_k)[y_k(t+1) - y_k(t)] \end{aligned}$$

because in the sum  $\sum_{\substack{i,j=1 \\ i \neq j}}^K y_j w_{ji} y_i$ ,  $y_k$  appears twice: once at the left and once at the right and  $w_{ij} = w_{ji}$ .

According to the updating procedure (4.6):

$$y_k(t+1) = \begin{cases} +1 & \text{if } -\sum_{\substack{i=1 \\ i \neq k}}^K w_{ki} y_i - x_k + t_k < 0 \\ y_k(t) & \text{if } -\sum_{\substack{i=1 \\ i \neq k}}^K w_{ki} y_i - x_k + t_k = 0 \\ 0 & \text{if } -\sum_{\substack{i=1 \\ i \neq k}}^K w_{ki} y_i - x_k + t_k > 0 \end{cases}$$

there are 2 cases (it was assumed that  $y_k(t+1) \neq y_k(t)$ ):

- $y_k(t) = +1$  and it changes to  $y_k(t+1) = 0$ . Then  $[y_k(t+1) - y_k(t)] < 0$  and  $-\sum_{\substack{i=1 \\ i \neq k}}^K w_{ki} y_i - x_k + t_k > 0$  (according to the algorithm) so  $\Delta E < 0$ .
- $y_k(t) = 0$  and it changes to  $y_k(t+1) = +1$ . Analogous the preceding case:  $\Delta E < 0$ .

If more than one term is changing then  $\Delta E$  is of the form:

$$\Delta E = E_{t+1} - E_t = \sum_{j=1}^K \Delta y_j \left( \sum_{\substack{i=1 \\ i \neq j}}^K w_{ji} y_i - x_j + t_j \right) < 0$$

which represents a sum of negative terms.

2. The  $\{y_i\}_{i=1,K}$  have all values either  $+1$  or  $0$ . The lowest possible value for  $E$  is obtained when  $\{y_i\}_{i=1,K} = 1$ , the input vector is also  $\mathbf{x} = \hat{\mathbf{1}}$  and the threshold vector is  $\mathbf{t} = \hat{\mathbf{0}}$  such that the negative

terms are maximum and the positive term is minimum (see (4.7)), assuming that all  $w_{ji} > 0$ ,  $i, j = \overline{1, K}$ .

$$E_{\min} = -\frac{1}{2} \sum_{\substack{j,i=1 \\ j \neq i}}^K |w_{ji}| - K$$

3. The energy function decreases, it doesn't increase, so  $\Delta E \neq +\infty$ . On the other hand the energy function is limited on the low end (according to the second part of the theorem) so it cannot decrease by an infinite amount:  $\Delta E \neq -\infty$ .

Also the value of  $\Delta E$  can't be infinitesimally small resulting into an infinite amount of time before it reaches its minimum. The minimum amount by which  $E$  may change is when just one component  $k$  is changing, for which  $W(k, :)y$  is minimum,  $x_k = 1$  and  $t_k = 0$ , the amount being:

$$\Delta E = - \left| \sum_{\substack{i=1 \\ i \neq k}}^K w_{ki} y_i \right| - x_k$$

( $y_k$  appears twice: once at the left and once at the right and  $w_{ij} = w_{ji}$ ).  $\square$

### Remarks:

- 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.

## 4.5.2 The Continuous Hopfield 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:

$$f(a) = \frac{1 + \tanh(\lambda a)}{2}, \quad \lambda = \text{const.}, \lambda \in \mathbb{R}^+$$

where  $\lambda$  is called the *gain parameter*. See figure 4.4 on the next page. The inverse of activation is:

❖  $f, \lambda, f^{(-1)}$   
gain parameter

$$f^{(-1)}(y) = \frac{1}{2\lambda} \ln \frac{y}{1-y} \quad (4.8)$$

See figure 4.5 on the following page.

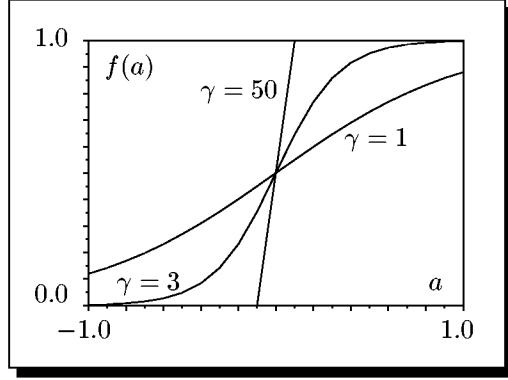
The differential equation governing the evolution of continuous Hopfield memory is defined as:

$$\frac{da_j}{dt} = \sum_{\substack{i=1 \\ i \neq j}}^K w_{ji} y_i + x_j - \frac{1}{\lambda} t_j a_j \quad (4.9)$$

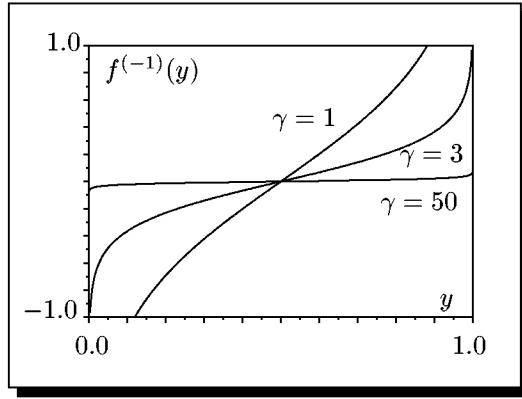
or in matrix notation:

$$\frac{d\mathbf{a}}{dt} = W\mathbf{y} + \mathbf{x} - \frac{1}{\lambda} \mathbf{t} \odot \mathbf{a}$$

<sup>4.5.2</sup>See [FS92] pp. 144–148.



**Figure 4.4:** The neuron activation function for continuous Hopfield memory (for different  $\gamma$  values).



**Figure 4.5:** The inverse of neuron activation function for continuous Hopfield memory (for different  $\gamma$  values).

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

$$\mathbf{y}(t+1) = \mathbf{y}(t) + \left( W\mathbf{y}(t) + \mathbf{x} - \frac{1}{\lambda} \mathbf{t} \odot \ln \frac{\mathbf{y}(t)}{\mathbf{1} - \mathbf{y}(t)} \right) \odot \mathbf{y}(t) \odot [\mathbf{1} - \mathbf{y}(t)] \quad (4.10)$$

(of course operations under  $\ln$  function are done on each  $y_j$  separately).

*Proof.* From (4.9):

$$\begin{aligned} \frac{df^{(-1)}(y_j)}{dt} &= \sum_{\substack{i=1 \\ i \neq j}}^K w_{ji} y_i + x_j - \frac{1}{\lambda} t_j f^{(-1)}(y_j) \\ df^{(-1)}(y_j) &= d \left( \ln \frac{y_j}{1 - y_j} \right) = \left( \frac{1}{y_j} + \frac{1}{1 - y_j} \right) dy_j = \frac{1}{y_j(1 - y_j)} dy_j \Rightarrow \\ dy_j &= \left( \sum_{\substack{i=1 \\ i \neq j}}^K w_{ji} u_i + x_j - \frac{1}{\lambda} t_j \ln \frac{y_j}{1 - y_j} \right) y_j(1 - y_j) dt \end{aligned}$$

and in discrete time approximation  $dt \rightarrow \Delta t = t + 1 - t = 1$ .  $\square$

**Definition 4.5.2.** *The following function:*

❖ E

$$E = -\frac{1}{2} \sum_{\substack{i,j=1 \\ i \neq j}}^K y_j w_{ji} y_i - \sum_{j=1}^K x_j y_j + \frac{1}{\gamma} \sum_{j=1}^K t_j \int_0^{y_j} f^{(-1)}(y') dy' \quad (4.11)$$

*is named the continuous Hopfield memory energy function.*

**Theorem 4.5.2.** *The continuous Hopfield energy function have the following properties:*

1. Any change in  $y$  as a result of running (evolution) results in a decrease in  $E$ , i.e.

$$\frac{dE}{dt} \leq 0$$

2.  $E$  is bounded below by:

$$E_{min} = -\frac{1}{2} \sum_{\substack{j,i=1 \\ j \neq i}}^K |w_{ji}| - K$$

*Proof.* 1. First:

$$\begin{aligned} \int \ln x dx &= x \ln x - x \Rightarrow \int \ln \frac{x}{1-x} dx = \ln e^{-1} x^x (1-x)^{1-x} \\ \lim_{x \searrow 0} \ln x^x &= \lim_{x \searrow 0} \frac{\ln x}{\frac{1}{x}} \stackrel{(L'Hospital)}{=} \lim_{x \searrow 0} -x = 0 \\ \int_0^{y_i} \ln \frac{y'}{1-y'} dy' &= \lim_{y_0 \searrow 0} \int_{y_0}^{y_i} \ln \frac{y'}{1-y'} dy' = \\ &= \ln y_i^{y_i} (1-y_i)^{1-y_i} - \lim_{y_0 \searrow 0} \ln y_0^{y_0} (1-y_0)^{1-y_0} = \ln y_i^{y_i} (1-y_i)^{1-y_i} \\ \frac{d}{dt} \ln y_i^{y_i} (1-y_i)^{1-y_i} &= \frac{d}{dy_i} (\ln y_i^{y_i} (1-y_i)^{1-y_i}) \frac{dy_i}{dt} = \\ &= (\ln y_i - \ln(1-y_i)) \frac{dy_i}{dt} = f^{(-1)}(y_i) \frac{dy_i}{dt} \end{aligned}$$

then, from (4.11) and using (4.9):

$$\begin{aligned} \frac{dE}{dt} &= - \sum_{j=1}^K \left( \sum_{\substack{i=1 \\ i \neq j}}^K w_{ji} u_i + x_j - \frac{1}{\gamma} t_j a_j \right) \frac{dy_j}{dt} \\ &= - \sum_{j=1}^K \frac{da_j}{dt} \frac{dy_j}{dt} = - \sum_{j=1}^K \frac{df^{(-1)}(y_j)}{dy_j} \left( \frac{dy_j}{dt} \right)^2 < 0 \end{aligned}$$

because  $\frac{df^{(-1)}(y_j)}{dy_j} = \frac{1}{y_j(1-y_j)} > 0$  ( $y_j \in (0, 1)$ ) such that  $\frac{dE}{dt}$  is always negative and  $E$  decreases in time.

2. The lowest possible value for  $E$  is obtained when  $\{y_j\}_{j=1,K} = 1$ , the input vector is also  $x = \hat{1}$  and the threshold vector is  $t = \hat{0}$ , such that the negative terms are maximum and the positive term is minimum (see (4.11)), assuming that all  $w_{ji} > 0$ ,  $i, j = \overline{1, K}$ .

$$E_{min} = -\frac{1}{2} \sum_{\substack{j,i=1 \\ j \neq i}}^K |w_{ji}| - K \quad \square$$

**Remarks:**

- The existence of continuous Hopfield energy function with the outlined properties ensures that the running process is *convergent*.
- While the process is convergent there is no guarantee that the process will converge to the lowest energy value.
- For  $\gamma \rightarrow +\infty$  then the continuous Hopfield becomes identical to the discrete one. Otherwise:
  - For  $\gamma \rightarrow 0$  then there is only one stable state for the network when  $y = \frac{1}{2} \hat{\mathbf{1}}$ .
  - For  $\gamma \in (0, +\infty)$  the stable states are somewhere between the corners of Hamming hypercube (having its center at  $\frac{1}{2} \hat{\mathbf{1}}$ ) and its center such that as the gain decreases from  $+\infty$  to 0 the stable points moves from corners towards the center and at some point they may merge.

## 4.6 Applications

### 4.6.1 The 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 other have a cost e.g. the intercity distance associated. The cost/distance traveled must be minimized. The salesperson have to return to the starting point.

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

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

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 a arithmetic progression growth of the problem the space of possible solutions grows *exponentially*.  $\square$

❖  $C_i$

Let  $C_1, \dots, C_K$  be the cities involved. To each of the  $K$  cities is attached a vector which represents a number, converted to a binary form, of the order of visiting in the current tour, i.e. the first one to be visited have:  $(1 \ 0 \ \dots \ 0)$ , the second one have:  $(0 \ 1 \ 0 \ \dots \ 0)$  and so on, the last one to be visited having attached the vector:  $(0 \ \dots \ 0 \ 1)$ ; i.e. each vector have one digit "1", all other elements being "0" (this format is different from the binary representation of the city number  $j$  as cities are not visited in their numbering order).

Having the cities  $C_1, \dots, C_K$ , a squared matrix can be build using their associate vectors as rows. Because the cities aren't necessary visited in their listed order the matrix is not

---

<sup>4.6.1</sup>See [FS92] pp. 148–156.

necessary diagonal.

$$\begin{array}{ccccccccc}
 1 & \dots & j_1 & \dots & j_2 & \dots & K \\
 \hline
 1 & \dots & 0 & \dots & 0 & \dots & 0 & C_1 \\
 \dots & \dots & \dots & \dots & \dots & \dots & \vdots \\
 0 & \dots & 1 & \dots & 0 & \dots & 0 & C_{k_1} \\
 \vdots & & \vdots & & \vdots & & \vdots \\
 0 & \dots & 0 & \dots & 1 & \dots & 0 & C_{k_2} \\
 \dots & \dots & \dots & \dots & \dots & \dots & \vdots \\
 0 & \dots & 0 & \dots & 0 & \dots & 1 & C_K
 \end{array} \tag{4.12}$$

This matrix defines the tour: for each column  $j = \overline{1, K}$  pickup as next city the one having the corresponding row element equal to 1.

The idea is to build a bidimensional Hopfield memory such that its output is a matrix  $Y$  (not a vector) having the layout (4.12) and this will give the solution (as each row will represent the visiting order number in binary format of the respective city).

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

- ① Each city must not be visited more than once  $\Leftrightarrow$  Each *row* of the matrix (4.12) should have no more than one “1”, all others elements should be “0”.
- ② Two cities can not be visited at the same time (can’t have the same order number)  $\Leftrightarrow$  Each *column* of the matrix (4.12) should have no more than one “1” all others elements should be “0”.
- ③ All cities should be visited  $\Leftrightarrow$  Each *row or column* of the matrix (4.12) should have at least one “1”.
- ④ The total distance/cost of the tour should be minimised. Let  $d_{k_1 k_2}$  be the distance/cost between cities  $C_{k_1}$  and  $C_{k_2}$ . ❖  $d_{k_1 k_2}$

As the network is bidimensional, each weight have 4 subscripts:  $w_{k_2 j_2 k_1 j_1}$  is the weight from neuron {row  $k_1$ , column  $j_1$ } to neuron {row  $k_2$ , column  $j_2$ }. See figure 4.6 on the next page.

### Remarks:

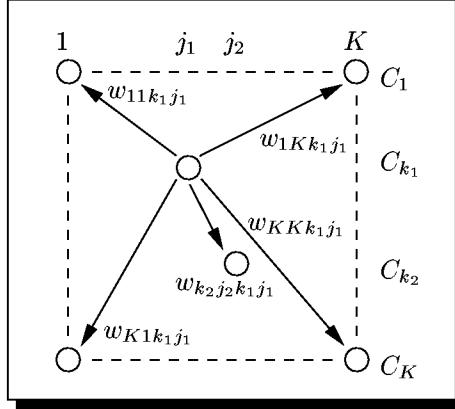
- ➔ When using bidimensional Hopfield networks all the established results will be kept but *each subscript will split in 2* giving the row and column position (instead of one giving the column position).

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

- 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 have 0. Then the weight should have a term of the form: ❖  $A$

$$w_{k_2 j_2 k_1 j_1}^{(1)} = -A \delta_{k_1 k_2} (1 - \delta_{j_1 j_2}), \quad A \in \mathbb{R}^+, \quad A = \text{const.}$$

where  $\delta_{\alpha\beta}$  is the Kroneker symbol. This means all  $w_{k_2 j_2 k_1 j_1}^{(1)} = 0$  for neurons on



**Figure 4.6:** The bidimensional Hopfield memory and its weight representation.

different rows,  $w_{k_1 j_2 k_1 j_1}^{(1)} < 0$  for a given row  $k_1$  if  $j_1 \neq j_2$  and  $w_{k_1 j_1 k_1 j_1}^{(1)} = 0$  for feedback.

- 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 have 0. Then the weight should have a term of the form:

$$w_{k_2 j_2 k_1 j_1}^{(2)} = -B \delta_{j_1 j_2} (1 - \delta_{k_1 k_2}), \quad B \in \mathbb{R}^+, \quad B = \text{const.}$$

This means all  $w_{k_2 j_2 k_1 j_1}^{(2)} = 0$  for neurons on different columns,  $w_{k_2 j_1 k_1 j_1}^{(2)} < 0$  for a given column if  $k_1 \neq k_2$  and  $w_{k_1 j_1 k_1 j_1}^{(2)} = 0$  for feedback.

- ③ Most of the neurons should have output 0 so a global inhibition may be used. Then the weight should have a term of the form:

$$w_{k_2 j_2 k_1 j_1}^{(3)} = -C, \quad C \in \mathbb{R}^+, \quad C = \text{const.}$$

i.e. all neurons receive the same global inhibition  $\propto C$ .

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

$$w_{k_2 j_2 k_1 j_1}^{(4)} = -D d_{k_1 k_2} (\delta_{j_1, j'_2+1} + \delta_{j_1, j''_2-1}), \quad D \in \mathbb{R}^+, \quad D = \text{const.}$$

where  $j'_2 = \begin{cases} 0 & \text{if } j_1 = 1 \text{ and } j_2 = K \\ j_2 & \text{in rest} \end{cases}$  and  $j''_2 = \begin{cases} K+1 & \text{if } j_1 = K \text{ and } j_2 = 1 \\ j_2 & \text{in rest} \end{cases}$

- ❖  $j'_2, j''_2$  to take care of special cases  $j_1 = 1$  and  $j_1 = K$ . The term  $\delta_{j_1, j'_2+1}$  takes care of the case when column  $j_2$  came *before*  $j_1$  (column  $K$  came “before” column 1) while  $\delta_{j_1, j''_2-1}$  operate similar for the case when  $j_2$  came *after*  $j_1$ .

Finally, the weights are:

$$\begin{aligned} w_{k_2 j_2 k_1 j_1} &= w_{k_2 j_2 k_1 j_1}^{(1)} + w_{k_2 j_2 k_1 j_1}^{(2)} + w_{k_2 j_2 k_1 j_1}^{(3)} + w_{k_2 j_2 k_1 j_1}^{(4)} \\ &= -A\delta_{k_1 k_2}(1 - \delta_{j_1 j_2}) - B\delta_{j_1 j_2}(1 - \delta_{k_1 k_2}) - C - Dd_{k_1 k_2}(\delta_{j_1, j'_2+1} + \delta_{j_1, j''_2-1}) \end{aligned} \quad (4.13)$$

Taking  $Y = \{y_{kj}\}$  be the network output (matrix) and considering  $X = \tilde{C}\tilde{1}$  as input (again matrix) and  $T = \tilde{0}$  as the threshold (matrix again) then from the definition of continuous Hopfield energy function (4.11) and using the weights (4.13):

$$\begin{aligned} E &= \frac{1}{2} A \sum_{k=1}^K \sum_{\substack{j_1, j_2=1 \\ j_1 \neq j_2}}^K y_{kj_1} y_{kj_2} + \frac{1}{2} B \sum_{\substack{k_1, k_2=1 \\ k_1 \neq k_2}}^K \sum_{j=1}^K y_{k_1 j} y_{k_2 j} + \\ &\quad + \frac{1}{2} C \sum_{k_1, k_2=1}^K \sum_{j_1, j_2=1}^K y_{k_1 j_1} y_{k_2 j_2} + \frac{1}{2} D \sum_{\substack{k_1, k_2=1 \\ k_1 \neq k_2}}^K \sum_{j=1}^K d_{k_1 k_2} y_{k_1 j} (y_{k_2, j'+1} + y_{k_2, j''-1}) \\ &\quad - CK \sum_{k=1}^K \sum_{j=1}^K y_{kj} \\ &= \frac{1}{2} A \sum_{k=1}^K \sum_{\substack{j_1, j_2=1 \\ j_1 \neq j_2}}^K y_{kj_1} u_{kj_2} + \frac{1}{2} B \sum_{j=1}^K \sum_{\substack{k_1, k_2=1 \\ k_1 \neq k_2}}^K y_{k_1 j} y_{k_2 j} + \\ &\quad + \frac{1}{2} C \left( \sum_{k=1}^K \sum_{j=1}^K y_{kj} - K \right)^2 + \frac{1}{2} D \sum_{\substack{k_1, k_2=1 \\ k_1 \neq k_2}}^K \sum_{j=1}^K d_{k_1 k_2} y_{k_1 j} (y_{k_2, j'-1} + y_{k_2, j''+1}) \\ &\quad - \frac{1}{2} CK^2 \\ &= E_1 + E_2 + E_3 + E_4 - \frac{1}{2} CK^2 \end{aligned} \quad (4.14)$$

$$\text{where } j' = \begin{cases} j & \text{if } j \neq 1 \\ K & \text{if } j = 1 \end{cases} \text{ and } j'' = \begin{cases} j & \text{if } j \neq K \\ 1 & \text{if } j = K \end{cases}. \quad \diamondsuit j', j''$$

According to theorem 4.5.2, during network running the energy (4.14) 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:

$$E_1 = \frac{1}{2} A \sum_{k=1}^K \sum_{\substack{j_1, j_2=1 \\ j_1 \neq j_2}}^K y_{kj_1} y_{kj_2}$$

which reaches minimum  $E_1 = 0$  if and only if each city appears only once in the tour such that the products  $y_{kj_1} y_{kj_2}$  are either of type  $1 \cdot 0$  or  $0 \cdot 0$ , i.e. there is only one 1 in each row of the matrix (4.12).

The  $1/2$  factor means that the terms  $y_{kj_1} y_{kj_2} = y_{kj_2} y_{kj_1}$  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 city  $C_{k_1}$  is the  $k_2$ -th to be visited then any other city can't be in the same  $k_2$ -th position in the tour:

$$E_2 = \frac{1}{2} B \sum_{j=1}^K \sum_{\substack{k_1, k_2=1 \\ k_1 \neq k_2}}^K y_{k_1 j} y_{k_2 j}$$

which reaches minimum  $E_2 = 0$  if and only if each city have different order number in the tour such that the products  $y_{k_1 j} y_{k_2 j}$  are either of type  $1 \cdot 0$  or  $0 \cdot 0$ , i.e. there is only one 1 in each column of the matrix (4.12).

The  $1/2$  factor means that the terms  $y_{k_1 j} y_{k_2 j} = y_{k_2 j} y_{k_1 j}$  will be added only once, not twice.

- ③ Energy minimum will favor states that have all cities in the tour:

$$E_3 = \frac{1}{2} C \left( \sum_{k=1}^K \sum_{j=1}^K y_{kj} - K \right)^2$$

reaching minimum  $E_3 = 0$  if all cities are represented in the tour, i.e.  $\sum_{k=1}^K \sum_{j=1}^K y_{kj} = K$

— the fact that *if* a city was present, it was once and only once was taken care in previous terms (there are  $K$  and only  $K$  “ones” in the whole matrix (4.12)).

The squaring shows that the *module* of the difference is important (otherwise energy may decrease for an increase of the number of missed cities, i.e. either  $\sum_{k=1}^K \sum_{j=1}^K y_{kj} \leq K$  is bad).

- ④ Energy minimum will favor states with minimum distance/cost of the tour:

$$E_4 = \frac{1}{2} D \sum_{\substack{k_1, k_2=1 \\ k_1 \neq k_2}}^K \sum_{j=1}^K d_{k_1 k_2} y_{k_1 j} (y_{k_2, j'+1} + y_{k_2, j''-1})$$

If  $y_{k_1 j} = 0$  then no distance will be added. If  $y_{k_1 j} = 1$  then 2 cases arises:

- (a)  $y_{k_2, j+1} = 1$  that means that the city  $C_{k_2}$  is the next one in the tour and the distance  $d_{k_1 k_2}$  will be added:  $d_{k_1 k_2} y_{k_1 j} y_{k_2, j+1} = d_{k_1 k_2}$ .
- (b)  $y_{k_2, j+1} = 0$  that means that the city  $C_{k_2}$  isn't the next one on the tour and the corresponding distance will not be added:  $d_{k_1 k_2} y_{k_1 j} y_{k_2, j+1} = 0$ .

Similar discussion for  $y_{k_2, j-1}$ .

The  $1/2$  means that the distances  $d_{k_1 k_2} = d_{k_2 k_1}$  will be added only once, not twice.

From previous terms it should be only one digit “1” on each row so a distance  $d_{k_1 k_2}$  should appear only once (the factor  $1/2$  was considered).

- ⑤ The term  $-\frac{1}{2} CK^2$  is just an additive constant, used to create the square in  $E_3$ .

To be able to use the running procedure (4.10), a way to convert  $\{w_{k_2 j_2 k_1 j_1}\}$  to a matrix and  $\{y_{kj}\}$  to a vector should be found. As indices  $\{k, j\}$  work in pairs this can be easily done using the lexicographic convention:

$$\begin{aligned} w_{k_2 j_2 k_1 j_1} &\rightarrow \tilde{w}_{(k_2-1)K+j_2, (k_1-1)K+j_1} \\ y_{kj} &\rightarrow \tilde{y}_{(k-1)K+j} \end{aligned}$$

The graphical representation of  $Y \rightarrow \tilde{y}$  transformation is very simple: take each column of  $Y^T$  and “glue” it at the bottom of previous one. The inverse transformation of  $\tilde{y}$  to get  $Y$  is also very simple:

$$\tilde{y}_\ell \rightarrow y_{\text{mod } K \ell + 1, \ell - K \text{ mod } K \ell}$$

The updating formula (4.10) then becomes:

$$\hat{y}(t+1) = \hat{y}(t) + [\tilde{W}\hat{y}(t) + C\hat{1}] \odot \hat{y}(t) \odot [\hat{1} - \hat{y}(t)]$$

and the  $A$ ,  $B$ ,  $C$  and  $D$  constants are used to tune the process.



## 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<sup>1</sup>.

CPN

❖  $\mathcal{C}_k$ ,  $\langle \mathbf{x} \rangle_k$ ,  $\langle \mathbf{y} \rangle_k$

### 5.1 The CPN Architecture

Let be a set of vector pairs  $(\mathbf{x}_1, \mathbf{y}_1), \dots, (\mathbf{x}_P, \mathbf{y}_P)$  who may be classified into several classes  $\mathcal{C}_1, \dots, \mathcal{C}_H$ . The CPN associate an input  $\mathbf{x}$  vector with an  $\langle \mathbf{y} \rangle_k \in \mathcal{C}_k$  for which the corresponding  $\langle \mathbf{x} \rangle_k$  is closest to input  $\mathbf{x}$ .  $\langle \mathbf{x} \rangle_k$  and  $\langle \mathbf{y} \rangle_k$  are the averages over those  $\mathbf{x}_p$ , respectively  $\mathbf{y}_p$  who are from *the same class*  $\mathcal{C}_k$ .

CPN may also work in reverse, inputting an  $\mathbf{y}$  and retrieving an  $\langle \mathbf{x} \rangle_k$ .

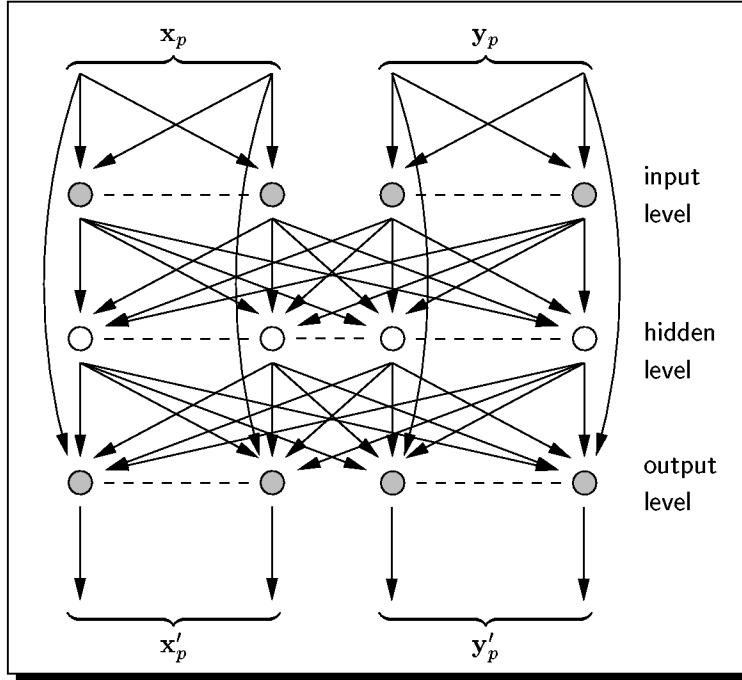
The CPN architecture consists of 5 layers on 3 levels. The input level contains  $x$  and  $y$  layers; the middle level contains the hidden layer and the output level contains the  $x'$  and  $y'$  layers. See figure 5.1 on the following page. Note that each neuron on  $x$  layer, input level, receive 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,  $\mathbf{y} = \hat{\mathbf{0}}$  at input level and the corresponding vector  $\langle \mathbf{y} \rangle_k$  is retrieved. When running in reverse an  $\mathbf{y}$  vector is applied,  $\mathbf{x} = \hat{\mathbf{0}}$  at input level and the  $\langle \mathbf{y} \rangle_k$  is retrieved. Both cases are identical, only the first will be discussed in detail.

---

<sup>1</sup>See "The BAM/Hopfield Memory" chapter for definition.

<sup>5.1</sup>See [FS92] pp. 213–234.



**Figure 5.1:** The CPN architecture.

This functionality is achieved as follows:

- The first level normalize the input vector.
- The second level (hidden layer) does a classification of input vector, outputting an one-of- $k$  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).
- Based on the classification performed on hidden layer, the output layer actually retrieve a “representative” vector.

All three subnetworks are quasi-independent and training at one level is performed only *after* the training at previous level have been finished.

### 5.1.1 The Input Layer

❖  $\mathbf{z}_x$

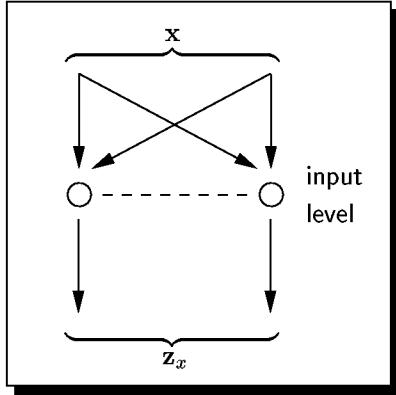
Let consider the  $x$  input layer<sup>2</sup> Let be  $N$  the dimension of vector  $\mathbf{x}$  and  $K$  the dimension of vector  $\mathbf{y}$ . Let  $\mathbf{z}_x$  be the output vector of the input  $x$  layer. See figure 5.2 on the next page.

❖  $B$

The input layer have to achieve a normalization of input; this may be done if the neuronal activity on the input layer is defined as follows:

- each neuron receive a positive excitation proportional to it's corresponding input, i.e.  $+Bx_i$ ,  $B = \text{const.}$ ,  $B > 0$ ,

<sup>2</sup>An identical discussion goes for  $y$  layer, as previously specified.



**Figure 5.2:** The input layer.

- each neuron receive a negative excitation from all neurons on the same layer, including itself, equal to  $-z_{xi}x_j$ ,
- the input vector  $\mathbf{x}$  is applied at time  $t = 0$  and removed ( $\mathbf{x}$  becomes  $\hat{\mathbf{0}}$ ) at time  $t = t'$ , and
- in the absence of input  $x_i$ , the neuronal output  $z_{xi}$  decrease to zero following an exponential, defined by  $A = \text{const.}, A > 0$ , i.e.  $z_{xi} \propto e^{-At}$ .  $\diamond A$

Then the neuronal behaviour may be summarized into:

$$\begin{aligned}\frac{dz_{xi}}{dt} &= -Az_{xi} + Bx_i - z_{xi} \sum_{j=1}^N x_j && \text{for } t \in [0, t') \\ \frac{dz_{xi}}{dt} &= -Az_{xi} && \text{for } t \in [t', \infty)\end{aligned}$$

or in matrix notation:

$$\frac{d\mathbf{z}_x}{dt} = -A\mathbf{z}_x + B\mathbf{x} - z_{xi}(\mathbf{x}^T \hat{\mathbf{1}}) \hat{\mathbf{1}} \quad \text{for } t \in [0, t') \quad (5.1a)$$

$$\frac{d\mathbf{z}_x}{dt} = -A\mathbf{z}_x \quad \text{for } t \in [t', \infty) \quad (5.1b)$$

The boundary conditions are  $\mathbf{z}_x(0) = \hat{\mathbf{0}}$  (starts from  $\hat{\mathbf{0}}$ , no previously applied signal) and  $\lim_{t \rightarrow \infty} \mathbf{z}_x(t) = \hat{\mathbf{0}}$  (returns to  $\hat{\mathbf{0}}$  after the signal have been removed). For continuity purposes the condition  $\lim_{t \nearrow t'} \mathbf{z}_x(t) = \lim_{t \searrow t'} \mathbf{z}_x(t)$  should be imposed. With these limit conditions, the solutions to (5.1a) and (5.1b) are:

$$\mathbf{z}_x = \frac{B\mathbf{x}}{A + \mathbf{x}^T \hat{\mathbf{1}}} \left\{ 1 - \exp \left[ - \left( A + \mathbf{x}^T \hat{\mathbf{1}} \right) t \right] \right\} \quad \text{for } t \in [0, t') \quad (5.2a)$$

$$\mathbf{z}_x = \frac{B\mathbf{x}}{A + \mathbf{x}^T \hat{\mathbf{1}}} \left\{ 1 - \exp \left[ - \left( A + \mathbf{x}^T \hat{\mathbf{1}} \right) t' \right] \right\} e^{-A(t-t')} \quad \text{for } t \in [t', \infty) \quad (5.2b)$$

*Proof.* 1. From (5.1a), for  $t \in [0, t']$ :

$$\frac{dz_x}{dt} + (A + \mathbf{x}^T \hat{\mathbf{1}}) z_{xi} \hat{\mathbf{1}} = B\mathbf{x} \Rightarrow \frac{dz_{xi}}{dt} + \left( A + \sum_{j=1}^N x_j \right) z_{xi} = Bx_i$$

First a solution for the homogeneous equation is to be found:

$$\begin{aligned} \frac{dz_{xi}}{dt} + \left( A + \sum_{j=1}^N x_j \right) z_{xi} = 0 &\Rightarrow \frac{dz_{xi}}{z_{xi}} = - \left( A + \sum_{j=1}^N x_j \right) dt \Rightarrow \\ z_{xi} &= z_{xi0} \exp \left[ - \left( A + \sum_{j=1}^N x_j \right) t \right] \end{aligned}$$

♦  $z_{xi0}$

where  $z_{xi0}$  is the integral constant.

The general solution to the non-homogeneous equation is found considering  $z_{xi0} = z_{xi0}(t)$ . Then from (5.1a) ( $x_i = \text{const.}$ ):

$$\begin{aligned} \frac{dz_{xi0}}{dt} \exp \left[ - \left( A + \sum_{j=1}^N x_j \right) t \right] &= Bx_i \Rightarrow \\ z_{xi0} &= \int Bx_i \exp \left[ \left( A + \sum_{j=1}^N x_j \right) t \right] dt = \frac{Bx_i}{A + \sum_{j=1}^N x_j} \exp \left[ \left( A + \sum_{j=1}^N x_j \right) t \right] \Rightarrow \\ z_{xi} &= \frac{Bx_i}{\left( A + \sum_{j=1}^N x_j \right)} = \text{const.} \end{aligned}$$

This solution is not convenient because it will mean an instant jump from 0 to maximal value for  $z_{xi}$  when  $\mathbf{x}$  is applied (see the initial condition) or it will be the trivial solution  $\mathbf{z}_x = \mathbf{x} = 0$ . As it was obtained in the assumption that  $\frac{dz_{xi0}}{dt} \neq 0$ , this means that this is not valid and thus  $z_{xi0}$  have to be considered constant. Then the general solution to (5.1a) is:

$$z_{xi} = z_{xi0} \exp \left[ - \left( A + \sum_{j=1}^N x_j \right) t \right] + \text{const.}, \quad z_{xi0} = \text{const.}$$

and then, replacing back into (5.1a) and using first boundary condition gives (5.2a).

2. From equation (5.1a), by separating variables and integrating, for  $t \in [t', \infty)$ :

$$z_{xi} = z_{xi0} e^{-A(t-t')} \quad z_{xi0} = \text{const.}$$

Then, from the continuity condition and (5.2a), the  $z_{xi0}$  is:

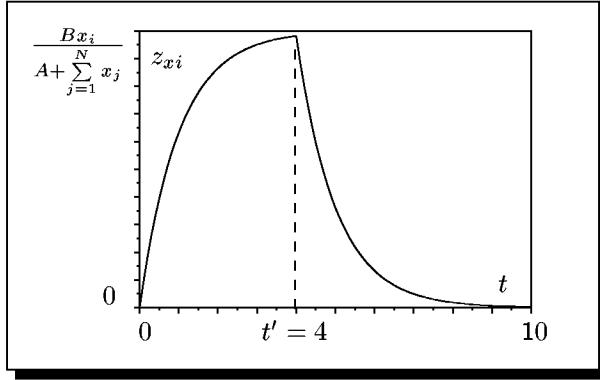
$$z_{xi0} = \frac{Bx_i}{A + \sum_{j=1}^N x_j} \left\{ 1 - \exp \left[ - \left( A + \sum_{j=1}^N x_j \right) t' \right] \right\} \quad \square$$

The output of a neuron from the input layer as function of time is shown in figure 5.3 on the facing page. The maximum value attainable on output is  $z_{x\max} = \frac{B\mathbf{x}}{A + \mathbf{x}^T \hat{\mathbf{1}}}$ , for  $t = t' \rightarrow \infty$ .



### Remarks:

- In practice, due to the exponential nature of the output, close values to  $z_{x\max}$  are obtained for  $t'$  relatively small, see again figure 5.3 on the next page, about 98% of the maximum value was attained at  $t = t' = 4$ .



**Figure 5.3:** The output of a neuron from the input layer as function of time.

→ Even if the input vector  $x$  is big ( $x_i \rightarrow \infty$ ,  $i = \overline{1, N}$ ) the output is limited but proportional with the input:

$$\mathbf{z}_{x\max} = \frac{B\mathbf{x}}{A + \mathbf{x}^T \hat{\mathbf{1}}} = \zeta_x \frac{B\mathbf{x}^T \hat{\mathbf{1}}}{A + \mathbf{x}^T \hat{\mathbf{1}}} \propto \zeta_x \quad \text{where} \quad \zeta_{xi} = \frac{x_i}{\sum_{j=1}^N x_j} \propto x_i$$

$\zeta_x$  being named *the reflectance pattern* and is “normalized” in the sense that  $\diamond \zeta_x$   
sums to unity:  $\sum_{j=1}^N \zeta_{xj} = 1$ .

### 5.1.2 The 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<sup>3</sup> (number of neurons) and  $\mathbf{z}_H = \{z_{Hk}\}_{k=\overline{1, H}}$  the output vector of the hidden layer.

Let  $\{w_{ki}\}_{\substack{k=\overline{1, H} \\ i=\overline{1, N+K}}}$  be the weight matrix (by which  $\mathbf{z}$  enters the hidden layer) such that the input to neuron  $k$  is  $W(k, :) \mathbf{z}$ .

The equations governing the behavior of a hidden neuron  $k$  are defined in a similar way as those of input layer ( $\mathbf{z}$  is applied from  $t = 0$  to  $t = t'$ ):  $\diamond a, b$

$$\frac{dz_{Hk}}{dt} = -az_{Hk} + bW(k, :) \mathbf{z} \quad \text{for } t \in [0, t') \tag{5.3a}$$

$$\frac{dz_{Hk}}{dt} = -az_{Hk} \quad \text{for } t \in [t', \infty) \tag{5.3b}$$

<sup>3</sup>The fact that this equals the number of classes is not a coincidence, later it will be shown that there have to be at least one hidden neuron to represent each class.

where  $a, b \in \mathbb{R}^+$ ,  $a, b = \text{const.}$ , and boundary conditions are  $z_{Hk}(0) = 0$ ,  $\lim_{t \rightarrow \infty} z_{Hk}(t) = 0$  and, for continuity purposes  $\lim_{t \nearrow t'} z_{Hk}(t) = \lim_{t \searrow t'} z_{Hk}(t)$ .

The weight matrix is defined to change as well (the network is learning) according to the following equation:

$$\frac{dW(k,:)}{dt} = \begin{cases} -c [W(k,:) - \mathbf{z}^T] & \text{if } z_k \neq 0 \\ 0 & \text{if } z_k = 0 \end{cases} \quad (5.4)$$

where  $c, d \in \mathbb{R}^+$ ,  $c, d = \text{const.}$  and boundary condition  $W(k,:)(0) = \widehat{\mathbf{0}}$ , second case being introduced to avoid the forgetting process.



### Remarks:

- In the absence of the input vector  $\mathbf{z} = 0$  if the learning process would continue then:

$$\frac{dW(k,:)}{dt} = -cW(k,:) \Rightarrow W(k,:) = Ce^{-ct} \xrightarrow{t \rightarrow \infty} 0$$

( $C$  being a constant row matrix).

Assuming that *the weight matrix is changing much slower than the neuron output* then  $W(k,:)\mathbf{z} \simeq \text{const.}$  and the solution to (5.3a) and (5.3b) are:

$$z_{Hk} = \frac{b}{a} W(k,:) \mathbf{z} (1 - e^{-at}) \quad \text{for } t \in [0, t') \quad (5.5a)$$

$$z_{Hk} = \frac{b}{a} W(k,:) \mathbf{z} (1 - e^{-at'}) e^{-a(t-t')} \quad \text{for } t \in [t', \infty) \quad (5.5b)$$

*Proof.* It is proven in a similar way as for the input layer, see section 5.1.1, proof of equations (5.2a) and (5.2b).  $\square$

The output of hidden neuron is similar to the output of input layer, see also figure 5.3 on the preceding page, with the remark that the maximal possible value for  $z_{Hk}$  is  $z_{Hk\max} = \frac{b}{a} W(k,:) \mathbf{z}$  for  $t, t' \rightarrow \infty$ .

Assuming that an input vector  $\mathbf{z}$  is applied and kept sufficiently long then the solution to (5.4) is of the form:

$$W(k,:) = \mathbf{z}^T (1 - e^{-ct})$$

i.e.  $W(k,:)$  moves towards  $\mathbf{z}$ . If  $\mathbf{z}$  is kept applied sufficiently long then  $W(k,:) \xrightarrow{t \rightarrow \infty} \mathbf{z}^T$ .

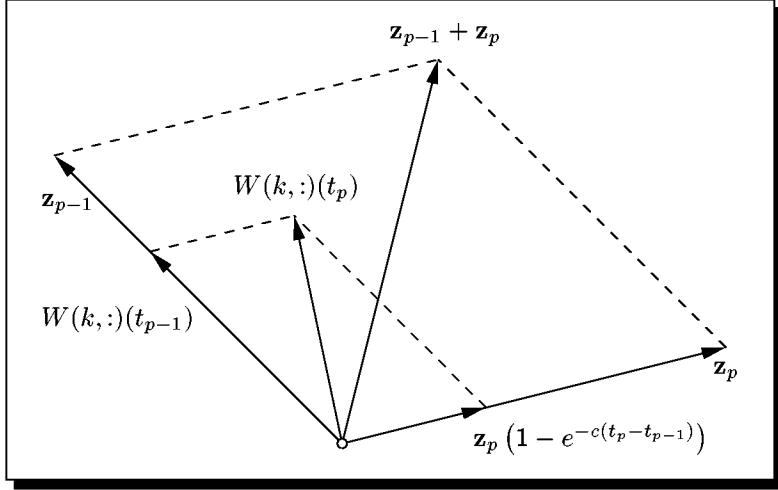
*Proof.* The differential (5.4) is very similar to previous encountered equations. It may be proven also by direct replacement.  $\square$

Let be a set of input vectors  $\{\mathbf{z}_p\}_{p=1,P}$  applied as follows:  $\mathbf{z}_1$  between  $t = 0$  and  $t = t_1$ ,  $\dots$ ,  $\mathbf{z}_P$  between  $t = t_{P-1}$  and  $t = t_P$ . Then the learning procedure is:

① Initialize the weight matrix:  $W(k,:) = \widehat{\mathbf{0}}$ .

② Considering  $t_0 \equiv 0$ , calculate the next value for weights:

$$W(k,:)(t_1) = \mathbf{z}_1 (1 - e^{-ct_1})$$



**Figure 5.4:** The directions taken by weight vectors, relatively to input, in hidden layer.

$$\begin{aligned}
 W(k,:)(t_2) &= \mathbf{z}_2 \left( 1 - e^{-c(t_2 - t_1)} \right) + W(k,:)(t_1) \dots \\
 W(k,:)(t_P) &= \mathbf{z}_P \left( 1 - e^{-c(t_P - t_{P-1})} \right) + W(k,:)(t_{P-1}) \\
 &= \sum_{p=1}^P \mathbf{z}_p \left( 1 - e^{-c(t_p - t_{p-1})} \right)
 \end{aligned}$$

The final weight vector  $W(k,:)$  represents a linear combination of all input vectors  $\{\mathbf{z}_p\}_{p=1,P}$ . Because the coefficients  $(1 - e^{-c(t_p - t_{p-1})})$  are all positive then the final direction of  $W(k,:)$  will point to an “average” direction pointed by  $\{\mathbf{z}_p\}_{p=1,P}$  and this is exactly the purpose of defining (5.4) as it was. See figure 5.4.



#### Remarks:

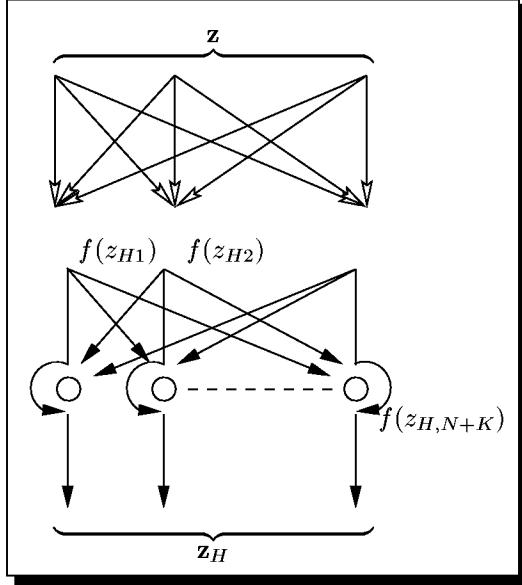
- ➔ It is also possible to give each  $\mathbf{z}_p$  a time-slice  $dt$  and when finishing with  $\mathbf{z}_P$  to start over with  $\mathbf{z}_1$  till some (external) stop conditions are met.

The trained instar is able to classify the direction of input vectors (see (5.5a)):

$$z_{Hk} \propto W(k,:) \mathbf{z} = \|W(k,:)\| \|\mathbf{z}\| \cos(\widehat{W(k,:)}, \mathbf{z}) \propto \cos(\widehat{W(k,:)}, \mathbf{z})$$

#### 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 hidden layer is to classify the normalized input vector  $\mathbf{z}$  (who is proportional to input). Its output is of the one-of- $k$  encoding type, i.e. all neurons have output zero except a neuron  $k$ . Note that it is assumed that classes do *not* overlap, i.e. an input vector may belong to one class only.



**Figure 5.5:** The competitive — hidden — layer.

The property of instars that their associate weight vector moves towards an average of input have to be preserved, but a feedback function is to be added, to ensure the required output. Let  $f = f(z_{Hk})$  be the feedback function of the instars, i.e. the value added at the neuron input. See figure 5.5

❖  $f$

Then the instar equations (5.3a) and (5.3b) are redefined as:

$$\begin{aligned} \frac{dz_{Hk}}{dt} &= -az_{Hk} + b[f(z_{Hk}) + W(k,:) \mathbf{z}] \\ &\quad - z_{Hk} \sum_{\ell=1}^{N+K} [f(z_{H\ell}) + W(\ell,:) \mathbf{z}] \quad \text{for } t \in [0, t') \end{aligned} \quad (5.6a)$$

$$\frac{dz_{Hk}}{dt} = -az_{Hk} \quad \text{for } t \in [t', \infty) \quad (5.6b)$$

where  $a, b \in \mathbb{R}^+$ ,  $a, b = \text{const.}$ ; the expression  $W(k,:) \mathbf{z} + \sum_{\ell=1}^{N+K} f(z_{H\ell})$  representing the total input of hidden neuron  $k$ . In matrix notation is:

$$\frac{d\mathbf{z}_H}{dt} = -a\mathbf{z}_H + b[f(\mathbf{z}_H) + W\mathbf{z}] - \mathbf{z}_H \odot [f(\mathbf{z}_H) + W\mathbf{z}] \quad \text{for } t \in [0, t')$$

$$\frac{d\mathbf{z}_H}{dt} = -a\mathbf{z}_H \quad \text{for } t \in [t', \infty)$$

The feedback function  $f$  have to be selected such that the hidden layer performs as a competitive 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”.

For a feedback function of type  $f(z) = z^r$  where  $r > 1$ , equations (5.6) defines a competitive layer,

*Proof.* First a *change of variable* is performed as follows:

❖  $\tilde{z}_{Hk}, z_{H,\text{tot}}$

$$\begin{aligned} \tilde{z}_{Hk} &\equiv \frac{z_{Hk}}{\sum_{\ell=1}^{N+K} z_{H\ell}} \quad \text{and} \quad z_{H,\text{tot}} \equiv \sum_{\ell=1}^{N+K} z_{H\ell} \\ \Rightarrow \quad z_{Hk} &= \tilde{z}_{Hk} z_{H,\text{tot}} \quad \text{and} \quad \frac{dz_{Hk}}{dt} = \frac{d\tilde{z}_{Hk}}{dt} z_{H,\text{tot}} + \tilde{z}_{Hk} \frac{dz_{H,\text{tot}}}{dt} \end{aligned} \quad (5.7)$$

By making the sum over  $k$  on (5.6a) (and  $f(z_{H\ell}) = f(\tilde{z}_{H\ell} z_{H,\text{tot}})$ ):

$$\frac{dz_{H,\text{tot}}}{dt} = -az_{H,\text{tot}} + (b - z_{H,\text{tot}}) \sum_{\ell=1}^{N+K} [f(\tilde{z}_{H\ell} z_{H,\text{tot}}) + W(\ell,:) \mathbf{z}] \quad (5.8)$$

and substituting  $z_{Hk} = \tilde{z}_{Hk} z_{H,\text{tot}}$  in (5.6a):

$$\frac{dz_{Hk}}{dt} = -a\tilde{z}_{Hk} z_{H,\text{tot}} + b[f(\tilde{z}_{Hk} z_{H,\text{tot}}) + W(k,:) \mathbf{z}] - \tilde{z}_{Hk} z_{H,\text{tot}} \sum_{\ell=1}^{N+K} [f(\tilde{z}_{H\ell} z_{H,\text{tot}}) + W(\ell,:) \mathbf{z}] \quad (5.9)$$

As  $\frac{d\tilde{z}_{Hk}}{dt} z_{H,\text{tot}} = \frac{dz_{Hk}}{dt} - \tilde{z}_{Hk} \frac{dz_{H,\text{tot}}}{dt}$  and from (5.8) and (5.9):

$$\frac{d\tilde{z}_{Hk}}{dt} z_{H,\text{tot}} = b[f(\tilde{z}_{Hk} z_{H,\text{tot}}) + W(k,:) \mathbf{z}] - b\tilde{z}_{Hk} \left[ \sum_{\ell=1}^{N+K} f(\tilde{z}_{H\ell} z_{H,\text{tot}}) + \sum_{\ell=1}^{N+K} W(\ell,:) \mathbf{z} \right] \quad (5.10)$$

The following cases, with regard to the feedback function, may be discussed:

- The identity function:  $f(\tilde{z}_{Hk} z_{H,\text{tot}}) = \tilde{z}_{Hk} z_{H,\text{tot}}$ . Then, by using  $\sum_{\ell=1}^{N+K} h_i = 1$ , the above equation become:

$$\frac{d\tilde{z}_{Hk}}{dt} z_{H,\text{tot}} = bW(k,:) \mathbf{z} - b\tilde{z}_{Hk} \sum_{\ell=1}^{N+K} W(\ell,:) \mathbf{z}$$

The stable value, obtained by stating  $\frac{d\tilde{z}_{Hk}}{dt} = 0$ , is:

$$\tilde{z}_{Hk} = \frac{W(k,:) \mathbf{z}}{\sum_{\ell=1}^{N+K} W(k,:) \mathbf{z}} \quad \Rightarrow \quad z_{Hk} \propto \frac{W(k,:) \mathbf{z}}{\sum_{\ell=1}^{N+K} W(\ell,:) \mathbf{z}}$$

- The square function:  $f(\tilde{z}_{Hk} z_{H,\text{tot}}) = (\tilde{z}_{Hk} z_{H,\text{tot}})^2$ . Again, as  $\sum_{\ell=1}^{N+K} \tilde{z}_{H\ell} = 1$ , (5.10) can be rewritten in the form:

$$\begin{aligned} \frac{d\tilde{z}_{Hk}}{dt} z_{H,\text{tot}} &= b \sum_{\ell=1}^{N+K} \tilde{z}_{H\ell} f(\tilde{z}_{H\ell} z_{H,\text{tot}}) - \sum_{\ell=1}^{N+K} \tilde{z}_{H\ell} f(\tilde{z}_{H\ell} z_{H,\text{tot}}) \\ &\quad + bW(k,:) \mathbf{z} - b\tilde{z}_{Hk} \sum_{\ell=1}^{N+K} W(\ell,:) \mathbf{z} \\ &= bz_{H,\text{tot}} \tilde{z}_{Hk} \sum_{\ell=1}^{N+K} \tilde{z}_{H\ell} \left[ \frac{f(\tilde{z}_{Hk} z_{H,\text{tot}})}{\tilde{z}_{Hk} z_{H,\text{tot}}} - \frac{f(\tilde{z}_{H\ell} z_{H,\text{tot}})}{\tilde{z}_{H\ell} z_{H,\text{tot}}} \right] \\ &\quad + bW(k,:) \mathbf{z} - b\tilde{z}_{Hk} \sum_{\ell=1}^{N+K} W(\ell,:) \mathbf{z} \end{aligned}$$

Then, considering the expression of  $f$ , the term:

$$\frac{f(\tilde{z}_{Hk} z_{H,\text{tot}})}{\tilde{z}_{Hk} z_{H,\text{tot}}} - \frac{f(\tilde{z}_{H\ell} z_{H,\text{tot}})}{\tilde{z}_{H\ell} z_{H,\text{tot}}}$$

reduces to  $z_{H,\text{tot}}(\tilde{z}_{Hk} - \tilde{z}_{H\ell})$  which for  $\tilde{z}_{Hk} > \tilde{z}_{H\ell}$  represents an amplification while for  $\tilde{z}_{Hk} < \tilde{z}_{H\ell}$  it represents an inhibition.

The term  $bW(k,:) \mathbf{z}$  is a constant with respect to  $\tilde{z}_{Hk}$  and the term  $-b\tilde{z}_{Hk} \sum_{\ell=1}^{N+K} W(\ell,:) \mathbf{z}$  represents an inhibitory term.

So the differential equations describe the behaviour of a network where all neurons inhibit all others with less output than theirs and are inhibited by neurons which have greater 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  $\tilde{z}_{Hk}$  will have a non zero output.

The same discussion occurs for  $f(\tilde{z}_{Hk} z_{H,\text{tot}}) = (\tilde{z}_{Hk} z_{H,\text{tot}})^\ell$  where  $\ell > 1$ .  $\square$

And finally, only the winning neuron, let  $k$  be the one, have to be affected by the learning process: this neuron will represent the class  $\mathcal{C}_k$  to which input vector belongs and its associated weight vector  $W(k,:)$  have to be moved towards the average “representative”  $\{\langle \mathbf{x} \rangle_k, \langle \mathbf{y} \rangle_k\}$  (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) or for cases with deep intricacy between various classes. See also figure 5.6 on page 81.

### 5.1.3 The 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 output level is to retrieve a pair  $\{\langle \mathbf{x} \rangle_k, \langle \mathbf{y} \rangle_k\}$ , where  $\langle \mathbf{x} \rangle_k$  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.

❖  $A', B', C'$

The differential equations governing the behavior of outstars are defined as:

$$\frac{dx'_i}{dt} = -A'x'_i + B'x_i + C'W'(i,:) \mathbf{z}_H \quad \text{for } t \in [0, t')$$

$$\frac{dx'_i}{dt} = -A'x'_i + C'W'(i,:) \mathbf{z}_H \quad \text{for } t \in [t', \infty)$$

where  $A', B', C' \in \mathbb{R}^+$ ,  $A', B', C' = \text{const.}$ , or in matrix notation:

$$\frac{d\mathbf{x}'}{dt} = -A'\mathbf{x}' + B'\mathbf{x} + C'W'(1:N,:) \mathbf{z}_H \quad \text{for } t \in [0, t') \quad (5.11a)$$

$$\frac{d\mathbf{x}'}{dt} = -A'\mathbf{x}' + C'W'(1:N,:) \mathbf{z}_H \quad \text{for } t \in [t', \infty) \quad (5.11b)$$

with boundary condition  $\mathbf{x}'(0) = \hat{\mathbf{0}}$ .

The weight matrix is changing — the network is learning — by construction, according to  $\diamond D', E'$  the following equation:

$$\frac{dW'(1:N,k)}{dt} = \begin{cases} -D'W'(1:N,k) + E'\mathbf{x} & \text{if } z_{Hk} \neq 0 \\ \hat{\mathbf{0}} & \text{if } z_{Hk} = 0 \end{cases} \quad (5.12)$$

with the boundary condition  $W'(1:N,k)(0) = \hat{\mathbf{0}}$  (the part for  $z_{Hk} = 0$  being defined in order to avoid weight “decaying”). 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 other 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:

$$\mathbf{x}' = \left[ \frac{B'}{A'} \mathbf{x} + \frac{C'}{A'} W'(1:N,:) \mathbf{z}_H \right] (1 - e^{-A't})$$

*Proof.* For each  $x'_i$ :  $B'x_i + C'W'(i,:) \mathbf{z}_H = \text{const.}$  and then the solution is built the same way as for previous differential equations by starting to search for the solution for homogeneous equation. See also proof of equations (5.2). The boundary condition is used to find the integration constant.  $\square$

The solution to weights update formula (5.12) is:

$$W'(1:N,k) = \frac{E'}{D'} \mathbf{x} (1 - e^{-D't})$$

*Proof.* Same way as for  $\mathbf{x}'$ , above, for each  $w_{ik}$  separately.  $\square$

The asymptotic behaviour for  $t \rightarrow \infty$  of  $\mathbf{x}'$  and  $W'(1:N,k)$  are (from the solutions found):

$$\lim_{t \rightarrow \infty} \mathbf{x}' = \frac{B'}{A'} \mathbf{x} + \frac{C'}{A'} W(1:N,:) \mathbf{z}_H \quad \text{and} \quad \lim_{t \rightarrow \infty} W'(1:N,k) = \frac{E'}{D'} \mathbf{x} \quad (5.13)$$

After a training with a set of  $\{\mathbf{x}_p, \mathbf{y}_p\}$  vectors, *from the same class k*, the weights will be:  $W'(1:N,k) \propto \langle \mathbf{x} \rangle_k$  respectively  $W'(N+1:N+K,k) \propto \langle \mathbf{y} \rangle_k$ .

At runtime  $\mathbf{x} \neq \hat{\mathbf{0}}$  and  $\mathbf{y} = \hat{\mathbf{0}}$  to retrieve an  $\mathbf{y}'$  (or vice-versa to retrieve an  $\mathbf{x}'$ ). But (similar to  $\mathbf{x}'$ )

$$\lim_{t \rightarrow \infty} \mathbf{y}' = \frac{B'}{A'} \mathbf{y} + \frac{C'}{A'} W(N+1:N+K,:) \mathbf{z}_H$$

and, as  $\mathbf{z}_H$  represents an one-of-k encoding, then  $W(N+1:N+K,:) \mathbf{z}_H$  selects the column  $k$  out of  $W'$  (for the corresponding winner) and as  $\mathbf{y} = \hat{\mathbf{0}}$  then

$$\mathbf{y}' \propto W'(N+1:N+K,k) \propto \langle \mathbf{y} \rangle_k$$

## 5.2 CPN Dynamics

### Remarks:

- ➔ In simulations on digital systems the normalization of vectors and the decision over the winner in the hidden layer may be done in separate processes and such simplifying and speeding up the network running.
- ➔ The process uses the asymptotic (equilibrium) values to avoid the actual solving of differential equations.
- ➔ The vector norm may be written as  $\|\mathbf{x}\| = \sqrt{\mathbf{x}^T \mathbf{x}}$ .

### 5.2.1 Network Running

To generate the corresponding  $\mathbf{y}$  vector for the input  $\mathbf{x}$ :

- ① The input layer normalizes the input vectors and distributes the result to the hidden layer. For the  $y$  part a null vector  $\hat{\mathbf{0}}$  is applied:

$$\begin{aligned} \mathbf{z}(1 : N) &= \frac{\mathbf{x}}{\sqrt{\mathbf{x}^T \mathbf{x}}} \quad (\text{as } \|\mathbf{y}\| = 0) \\ \mathbf{z}(N + 1 : N + K) &= \hat{\mathbf{0}} \end{aligned}$$

i.e.  $\mathbf{z}$  is the combination of vectors  $\frac{\mathbf{x}}{\|\mathbf{x}\|}$  and  $\hat{\mathbf{0}}$  to form a single vector.

- ② The hidden layer is of “winner-takes-all” type.

To avoid a differential equation calculation the weight vectors  $W(\ell, :)$  are *normalized*. This way the closest one to *normalized* input  $\mathbf{z}$  will be found by doing a simple scalar product:  $W(\ell, :) \mathbf{z} \propto \cos(\widehat{W(\ell, :) \mathbf{z}})$ .

The “raw” output of the hidden neurons is calculated first:  $z_{H\ell} \propto W(\ell, :) \mathbf{z}$ , then the neuron with the largest output is declared winner and it gets the output 1, all others get output 0. Let  $k$  be the winning neuron, i.e.  $W(k, :) \mathbf{z} = \max_{\ell} W(\ell, :) \mathbf{z}$ . Then initialize  $\mathbf{z}_H$  to  $\hat{\mathbf{0}}$  and afterwards make  $z_{Hk} = 1$ :

$$\mathbf{z}_H = \hat{\mathbf{0}} \quad \text{and afterwards} \quad z_{Hk} = 1$$

and so all outstars receive an input vector of the form:

$$\mathbf{z}_H^T = (0 \quad \dots \quad 1 \quad \dots \quad 0) \tag{5.14}$$

where all  $z_{H\ell}$  are zero, except  $z_{Hk}$ .

Note that as  $\mathbf{y} = \hat{\mathbf{0}}$  is as all things happen in the space  $\mathbb{R}^N \subset \mathbb{R}^{N+K}$ . The scalar product may be replaced with the scalar product between  $\frac{\mathbf{x}}{\|\mathbf{x}\|}$  and the projection of  $W(\ell, :)$ , i.e.  $W(\ell, 1 : N)$ .

---

<sup>5.2</sup>See [FS92] pp. 235–239.

- ③ From (5.13) and making  $\mathbf{y} = \hat{\mathbf{0}}$ ,  $C' = A'$  and  $E' = D'$  then the output of the  $y'$  layer is  $\mathbf{y}' = W'(N+1 : N+K, k)$ , i.e. the winner of hidden layer selects what column of  $W'$  will be chosen to be the output (the  $W'(1 : N, k)$  represents the  $\mathbf{x}'$  while  $W'(:, k)$  represents the joining of  $\mathbf{x}'$  and  $\mathbf{y}'$ ) as the multiplication between  $W'$  and a vector of type (5.14) selects column  $k$  out of  $W'$ .

To generate the corresponding  $\mathbf{x}$  from  $\mathbf{y}$ , i.e. working in reverse, the procedure is the same (by changing  $\mathbf{x} \leftrightarrow \mathbf{y}$ ).

### 5.2.2 Network Learning

- ① An input vector, randomly selected, is applied to the input layer.
- ② The input layer normalize the input vector and distribute it to the hidden layer.

$$\begin{aligned}\mathbf{z}(1 : N) &= \frac{\mathbf{x}}{\sqrt{\mathbf{x}^T \mathbf{x} + \mathbf{y}^T \mathbf{y}}} \\ \mathbf{z}(N+1 : N+K) &= \frac{\mathbf{y}}{\sqrt{\mathbf{x}^T \mathbf{x} + \mathbf{y}^T \mathbf{y}}}\end{aligned}$$

i.e.  $\mathbf{z}$  is the normalized combination of vectors  $\mathbf{x}$  and  $\mathbf{y}$  to form a single vector.

- ③ *The training of the hidden layer is done first.* The weights are initialized with randomly selected normalized input vectors. This ensure both the normalization of weight vectors  $W(\ell, :)$  and a good spread of them.

The hidden layer is of “winner-takes-all” type. To avoid a differential equation calculation, and as  $W(\ell, :)$  are normalized, the closest  $W(\ell, :)$  to  $\mathbf{z}$  is found by using the scalar product  $W(\ell, :) \mathbf{z} \propto \cos(\widehat{W(\ell, :) \mathbf{z}})$ .

The “raw” output of the hidden neurons is calculated first as scalar products:  $z_{H\ell} \propto W(\ell, :) \mathbf{z}$ , then the neuron with the largest output, i.e. the  $k$  one for which  $W(k, :) \mathbf{z} = \max_{\ell} W(\ell, :) \mathbf{z}$ , is declared winner and it gets the output 1, all others get output 0:

$$\mathbf{z}_H = \hat{\mathbf{0}} \quad \text{and afterwards} \quad z_{Hk} = 1$$

and so all outstars receive an input vector of the form

$$\mathbf{z}_H^T = (0 \quad \dots \quad 1 \quad \dots \quad 0)$$

where all  $z_{H\ell}$  are zero with one exception  $z_{Hk}$ .

The weight of the winning neuron is updated according to the equation (5.4). In discrete time approximation: such that:

$$\begin{aligned}dt \rightarrow \Delta t = 1 \quad \text{and} \quad dW(k, :) \rightarrow \Delta W(k, :) &= W(k, :)(t+1) - W(k, :)(t) \quad \Rightarrow \\ W(k, :)(t+1) &= W(k, :)(t) + c[\mathbf{z}^T - W(k, :)(t)]\end{aligned}$$

The above updating is repeated for all input vectors.

The training is repeated until the input vectors are recognized correctly, e.g. till the angle between the input vector and the output vector is less than some maximum error specified:  $\cos(\widehat{W(k, :) \mathbf{z}}) < \varepsilon$ .

The network may be tested with some input vectors not used before. If the classification is good (the error is under the maximal one) the the training of hidden layer is done, else the training continues.

- ④ *After the training of the hidden layer is finished the training of the output layer begins.*

An input vector is applied, the input layer normalizes it and distribute it to the *trained* hidden layer. On the hidden layer only one neuron is winner and have output non-zero, let  $k$  be that one, such that the output vector becomes  $\mathbf{z}_H = (0 \dots 1 \dots 0)$  with 1 on the  $k$  position.

The weight of the winning neuron is updated according to the equation (5.12). In discrete time approximation:

$$\begin{aligned} dt \rightarrow \Delta t = 1 \quad \text{and} \quad dW'(k,:) \rightarrow \Delta W'(k,:) = W'(k,:)(t+1) - W'(k,:)(t) \Rightarrow \\ W'(1:N,k)(t+1) = W'(1:N,k)(t) + E'[\mathbf{x} - W'(1:N,k)(t)] \quad (5.15) \\ W'(N+1:N+K,k)(t+1) = W'(N+1:N+K,k)(t) \\ + E'[\mathbf{y} - W'(N+1:N+K,k)(t)] \end{aligned}$$

The above updating is repeated for all input vectors.

The training is repeated until the input vectors are recognized correctly, e.g. till the error is less than some maximum error specified:  $w'_{ik} - x_i < \varepsilon$  for  $i = \overline{1, N}$  and similar for  $\mathbf{y}$ .



### Remarks:

- ➔ From the learning procedure it becomes clear that the CPN is in fact a system composed of several semi-independent subnetworks:

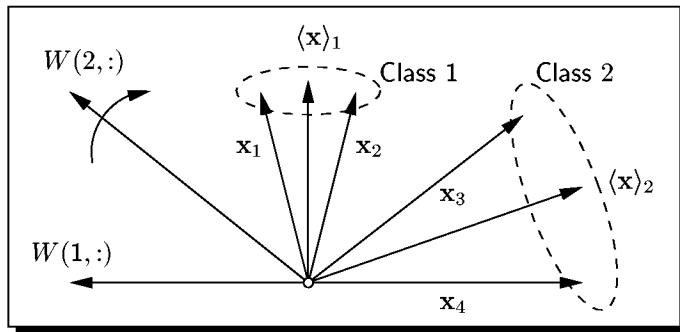
- the input level who normalize input,
- the hidden layer of “winner-takes-all” type and
- the output level who generate the actual required output.

Each level is independent and the training of next layer starts only *after* the learning in precedent layer have been done.

- ➔ Usually the CPN is used to classify an input vector  $\mathbf{x}$  as belonging to a class represented by  $\langle \mathbf{x} \rangle_k$ . A set of input vectors  $\{\mathbf{x}_p\}$  will be used to train the network such that it will have the output  $\{\langle \mathbf{x} \rangle_k, \langle \mathbf{y} \rangle_k\}$  if  $\mathbf{x} \in \mathcal{C}_k$  (see also figure 5.6 on the facing page).

- ➔ Unfortunate choice of weight vectors for the hidden layer  $W(\ell,:)$  may lead to the “stuck-vector” situation when one neuron from the hidden layer may never win. See figure 5.6 on the next page: the vector  $W(2,:)$  will move towards  $\mathbf{x}_{1,2,3,4}$  during learning and will become representative for both classes 1 and 2 — the corresponding hidden neuron 2 will be a winner for both classes.

The “stuck vector” situation can 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 than the number of



**Figure 5.6:** The “stuck vector” situation.

training vectors from the class it suppose to represents — then it will shut down allowing other neurons to win and corresponding weights to be changed.

- 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.
- For the output layer the critical point is to select an adequate learning constant  $E'$ : the learning constant can be chosen small  $0 \lesssim E' \ll 1$  at the beginning and increased later when  $x_i - w'_{ik}(t)$  decreases, see equation (5.15).
- Obviously the hidden layer may be replaced by any other system able to perform an one-of- $k$  encoding.

## 5.3 The Algorithm

### The running procedure

1. The  $\mathbf{x}$  vector is assumed to be known and the corresponding  $\langle \mathbf{y} \rangle_k$  is to be retrieved. For the reverse situation — when  $\mathbf{y}$  is known and  $\langle \mathbf{x} \rangle_k$  is to be retrieved — the algorithm is similar changing  $\mathbf{x} \leftrightarrow \mathbf{y}$ .

Make the  $\mathbf{y}$  null ( $\hat{\mathbf{0}}$ ) at input and compute the normalized input vector  $\mathbf{z}$ :

$$\mathbf{z}(1 : N) = \frac{\mathbf{x}}{\sqrt{\mathbf{x}^T \mathbf{x}}} \quad \text{and} \quad \mathbf{z}(N + 1 : N + K) = \hat{\mathbf{0}}$$

2. Find the winning neuron on hidden layer, the  $k$  for which:  $W(k, :) \mathbf{z} = \max_{\ell} W(\ell, :) \mathbf{z}$ .
3. Find the  $\mathbf{y}$  vector in  $W'$  matrix:

$$\mathbf{y} = W'(N + 1 : N + K, k)$$

### The learning procedure

1. Let  $\mathbf{x}$  and  $\mathbf{y}$  be one set of input vectors

Let  $N$  be the dimension of the “ $x$ ” part and  $K$  the dimension of the “ $y$ ”. Then  $N + K$  is the number of neurons in the input layer.

2. For all  $\{\mathbf{x}_p, \mathbf{y}_p\}$  training sets, normalize the input vector  $\Leftrightarrow$  compute  $\mathbf{z}_p$ .

$$\mathbf{z}_p(1 : N) = \frac{\mathbf{x}_p}{\sqrt{\mathbf{x}_p^T \mathbf{x}_p + \mathbf{y}_p^T \mathbf{y}_p}}$$

$$\mathbf{z}_p(N + 1 : N + K) = \frac{\mathbf{y}_p}{\sqrt{\mathbf{x}_p^T \mathbf{x}_p + \mathbf{y}_p^T \mathbf{y}_p}}$$

Note that the input layer does just a normalisation of the input vectors. No further training is required.

3. Initialize weights on hidden layer. For all  $\ell$  neurons ( $\ell = \overline{1, H}$ ) in hidden layer select an representative input vector  $\mathbf{z}_\ell$  for class  $\ell$  and then  $W(\ell, :) = \mathbf{z}_\ell^T$  (this way the weight vectors become automatically normalized).

Note that in extreme case there may be just one vector available for training for each class. In this case that vector becomes the “representative”.

4. Train the hidden layer. For all normalized training vectors  $\mathbf{z}_p$  find the winning neuron on hidden layer, the  $k$  one for which  $W(k, :) \mathbf{z}_p = \max_\ell W(\ell, :) \mathbf{z}_p$ .

Update the winner weights:

$$W_{\text{new}}(k, :) = W_{\text{old}}(k, :) + c[\mathbf{z}_p^T - W_{\text{old}}(k, :)]$$

The training of hidden layer have to be finished before moving forward to the output layer.

5. Initialize the weights on output layer. As for the hidden layer, select an representative input vector pair  $\{\mathbf{x}_k, \mathbf{y}_k\}$  for each class  $C_k$ .

$$W'(1 : N, \ell) = \mathbf{x}_k$$

$$W'(N + 1 : N + K, \ell) = \mathbf{y}_k$$

Another possibility would be to make an average over several  $\{\mathbf{x}_p, \mathbf{y}_p\}$  belonging to the *same* class.

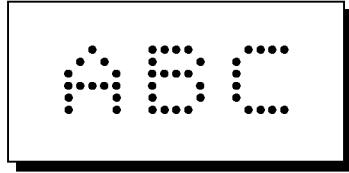
6. Train the output layer. For all training vectors  $\mathbf{z}_p$  find the winning neuron on hidden layer, the  $k$  one for which  $W(k, :) \mathbf{z}_p = \max_\ell W(\ell, :) \mathbf{z}_p$ .

Update the winner’s output weights:

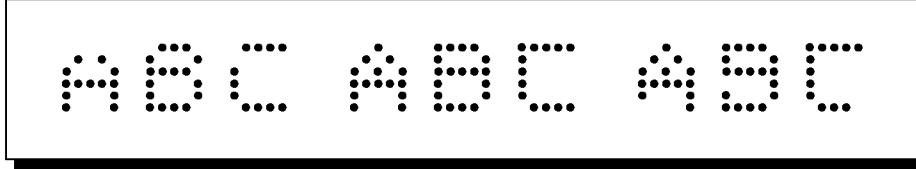
$$W'_{\text{new}}(1 : N, k) = W'_{\text{old}}(1 : N, k) + E'[\mathbf{x}_p - W'_{\text{old}}(1 : N, k)]$$

$$W'_{\text{new}}(N + 1 : N + K, k) = W'_{\text{old}}(N + 1 : N + K, k)$$

$$+ E'[\mathbf{y}_p - W'_{\text{old}}(N + 1 : N + K, k)]$$



**Figure 5.7:** The representative set for letters A, B, C.



**Figure 5.8:** The training set for letters A, B, C.

## ► 5.4 Applications

### 5.4.1 Letter classification

Being given a set of letters as binary image into a  $5 \times 6$  matrix the network have 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 corresponding 3 neurons on the hidden layer. The representative letters are in figure 5.7.

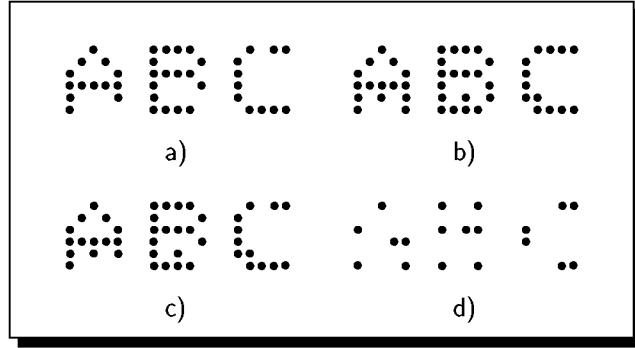
The  $x$  vectors are created by reading the graphical representation of the characters on rows; a “dot” gets an 1, its absence gets an 0:

$$\begin{aligned} x_{\text{rep.}}^T(A) = & \begin{pmatrix} 0 & 0 & 1 & 0 & 0 & , \\ 0 & 1 & 0 & 1 & 0 & , \\ 1 & 0 & 0 & 0 & 1 & , \\ 1 & 1 & 1 & 1 & 1 & , \\ 1 & 0 & 0 & 0 & 1 & , \\ 1 & 0 & 0 & 0 & 1 & ) \end{pmatrix}, \quad x_{\text{rep.}}^T(B) = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & , \\ 1 & 0 & 0 & 0 & 1 & , \\ 1 & 1 & 1 & 1 & 0 & , \\ 1 & 0 & 0 & 0 & 1 & , \\ 1 & 0 & 0 & 0 & 1 & , \\ 1 & 1 & 1 & 1 & 0 & ) \end{pmatrix}, \\ x_{\text{rep.}}^T(C) = & \begin{pmatrix} 0 & 1 & 1 & 1 & 1 & , \\ 1 & 0 & 0 & 0 & 0 & , \\ 1 & 0 & 0 & 0 & 0 & , \\ 1 & 0 & 0 & 0 & 0 & , \\ 0 & 1 & 1 & 1 & 1 & ) \end{pmatrix} \end{aligned}$$

such that they are 30-dimensional vectors.

The ASCII codes are 65 for A, 66 for B and 67 for C. They are converted to binary format and to a  $y$  8-dimensional vector:

$$\begin{aligned} y^T(A) &= (0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 0 \ 1) \\ y^T(B) &= (0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 0) \end{aligned}$$



**Figure 5.9:** The testing set for letters A, B, C.

$$\mathbf{y}^T(C) = (0 \ 1 \ 0 \ 0 \ 0 \ 0 \ 1 \ 1)$$

The training letters are depicted in figure 5.8 on the page before — the first set contains a “dot” less (information missing), the second one contains a supplementary “dot” (noise added) while the third set contains both.

The test letters are depicted in figure 5.9 — first 3 sets are similar, but not identical, to the training sets and they were 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.
- ➔ The training was done just once for the training set (one epoch) with the following constants:

$$c = 0.1 \quad \text{and} \quad E' = 0.1$$

- ➔ At run-time the  $\mathbf{y}$  vector becomes  $\hat{\mathbf{0}}$ .
- ➔ 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 (ART)

The ART networks are an example of ANN composed of several subsystems and able to resume learning at a later stage *without* having to restart from scratch.

## ► 6.1 The ART1 Architecture

The ART1 network is made from 2 main layers of neurons:  $F_1$  and  $F_2$ , a gain control unit (neuron) and a reset unit (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 build such that they become active if and only if 2 out of 3 input sources are active.

2/3 rule

### Remarks:

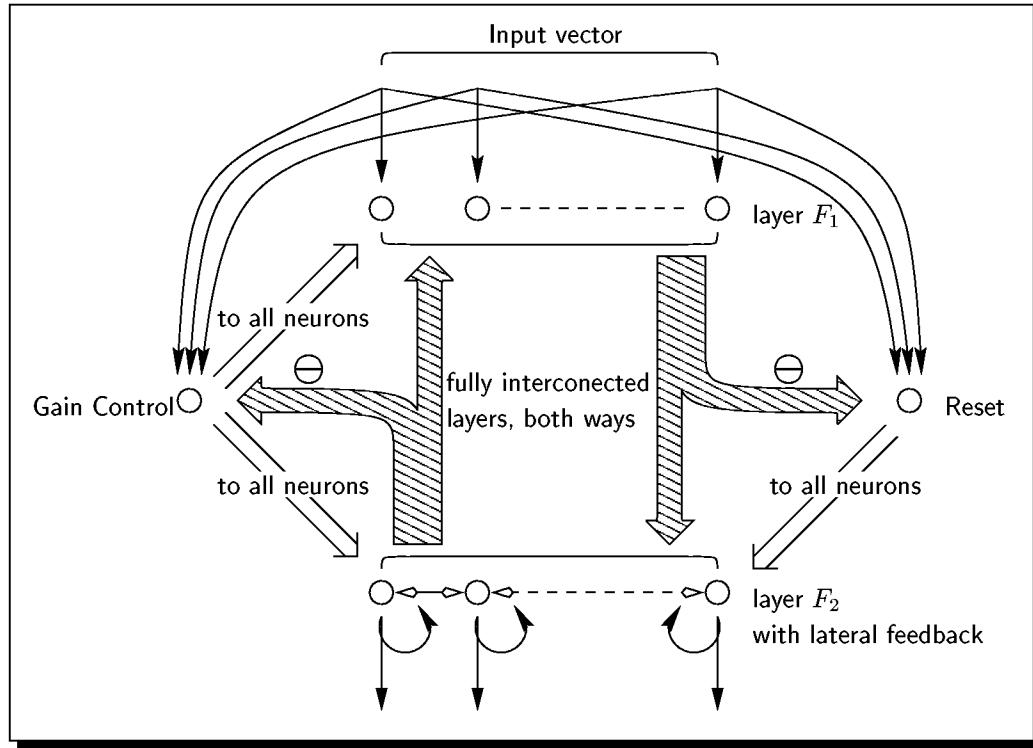
- ➔ The input is considered to be active when the input vector is non-zero i.e. it have at least one non-zero component.
- ➔ The  $F_2$  layer is considered active when its output vector is non-zero i.e. it have at least one non-zero component.

The propagation of signals trough the network is done as follows:

- ① The input vector is distributed to  $F_1$  layer, gain control unit and reset unit. Each component of input vector is distributed to a different  $F_1$  neuron —  $F_1$  have the same dimension  $N$  as input  $x$ .

❖  $x, N$

<sup>6.1</sup>See [FS92] pp. 293–298.



**Figure 6.1:** The ART1 network architecture. Inhibitory input is marked with  $\ominus$  symbol.

- ② The output of  $F_1$  is sent as inhibitory signal to the reset unit. The design of the network is such that the inhibitory signal from  $F_1$  cancels the input vector and the reset unit remains inactive.
- ③ The gain control unit send a nonspecific excitatory signal to  $F_1$  layer (an identical signal to all neurons from  $F_1$ ).
- 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 contrast enhancement type: only one neuron should trigger for a given pattern (or, in a more generalized case only few neurons should “fire” — have a nonzero output).
- ⑤ The output of  $F_2$  is sent back as excitatory signal to  $F_1$  and as inhibitory signal to the gain control unit. The design of the network is such that if the gain control unit receives an inhibitory signal from  $F_2$  it ceases activity.
- ⑥ Then  $F_1$  receives signals from  $F_2$  and input (the gain control unit have been deactivated). The output of the  $F_1$  layer changes such that it isn't anymore identical to the first one, because the overall input had changed: the gain control unit ceases activity and — instead — the  $F_2$  sends its output to  $F_1$ . Also there is the *2/3 rule* which have to be taken into account: 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 unit becomes active.
- ⑦ The reset unit send a reset signal to the active neuron(s) from the  $F_2$  layer which forces

it (them) to become inactive for a *long* period of time, i.e. they do not participate into the next network pattern matching cycle(s). 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 the reset unit, because is identical to the first one, or — no match was found, the output of  $F_2$  is zero — a learning process begins in  $F_2$ .

The action of the reset unit (see previous step) ensures that a neuron already used in the “past” will not be used again for pattern matching.



#### 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  $F_2$  to 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*.

With the presence with the gain unit  $F_2$  can't trigger a process by itself but it can precondition  $F_1$  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.

expectation

## ► 6.2 ART1 Dynamics

The equation describing the activation (total input) of a neuron  $j$  from  $F_{1,2}$  layers is of the form:

$$\frac{da_j}{dt} = -a_j + (1 - Aa_j) \times \text{excitatory input} - (B + Ca_j) \times \text{inhibitory input} \quad (6.1)$$

where  $A, B, C$  are positive constants.



#### Remarks:

- These equations do not describe the actual output of neurons with will be obtained from the activation by applying a “digitizing” function which will transform the activation into a binary vector.

### 6.2.1 The $F_1$ layer

The neuron on  $F_1$  layer receives input  $x$ , input  $z'$  from  $F_2$  and input from the gain control unit as *excitatory* input. The *inhibitory* input is set to 1. See (6.1).

Let be  $a'_k$  the *activation* of the neuron  $k$  from the  $F_2$  layer,  $f_2(a'_k)$  its output ( $f_2$  being the

❖  $a_k, f_2, W, y$

<sup>6.2</sup>See [FS92] pp. 298–310.

activation function on layer  $F_2$  and  $f_2(\mathbf{a}') = \mathbf{y}$ ) and  $w_{jk}$  the weight when entering neuron  $j$  on layer  $F_1$ . Then the total input received by the  $F_1$  neuron from  $F_2$  is  $W(j,:) f_2(\mathbf{a}')$ .

The  $F_2$  layer is of competitive (“winner-takes-all”) type — there is only one winning neuron which have a non-zero output, all others will have null output. The output activation function for  $F_2$  neurons is a binary function<sup>1</sup>:

$$\text{output of } F_2 \text{ neuron} = f_2(a'_k) = \begin{cases} 1 & \text{if winner is } k \\ 0 & \text{otherwise} \end{cases}$$

❖  $g$  The gain control unit is set such that if input vector  $\mathbf{x} \neq \hat{\mathbf{0}}$  and the vector from  $F_2$  is  $f_2(\mathbf{a}') = \hat{\mathbf{0}}$  then its output is 1, otherwise is 0:

$$g = \begin{cases} 1 & \text{if } \mathbf{x} \neq \hat{\mathbf{0}} \text{ and } f_2(\mathbf{a}') = \hat{\mathbf{0}} \\ 0 & \text{otherwise} \end{cases}$$

Finally the dynamic equation for a neuron  $j$  from the  $F_1$  layer becomes (from (6.1)):

$$\frac{da_j}{dt} = -a_j + (1 - A_1 a_j)[x_j + D_1 W(j,:) f_2(\mathbf{a}') + B_1 g] - (B_1 + C_1 a_j) \quad (6.2)$$

❖  $A_1, B_1, C_1, D_1$  where the constants  $A, B, C$  and  $D$  have been given the subscript 1 to denote that they are for  $F_1$  layer. Obviously here  $D_1$  controls the amplitude of  $W$  weights, it should be chosen such that all weights are  $w_{j\ell} \in [0, 1]$ .

The following cases may be considered:

- ① Input is inactive ( $\mathbf{x} = \hat{\mathbf{0}}$ ) and  $F_2$  is inactive ( $f_2(\mathbf{a}') = \hat{\mathbf{0}}$ ). Then  $g = 0$  and (6.2) becomes:

$$\frac{da_j}{dt} = -a_j - (B_1 + C_1 a_j)$$

At equilibrium  $\frac{da_j}{dt} = 0$  and  $a_j = -\frac{B_1}{1+C_1}$  i.e. *inactive  $F_1$  neurons have negative activation*.

- ② Input is active ( $\mathbf{x} \neq \hat{\mathbf{0}}$ ) but  $F_2$  is still inactive ( $f_2(\mathbf{a}') = \hat{\mathbf{0}}$ ) — there was no time for the signal to travel from  $F_1$  to  $F_2$  and back (and deactivating the gain control unit on the way back). The gain control unit is activated:  $g = 1$  and (6.2) becomes:

$$\frac{da_j}{dt} = -a_j + (1 - A_1 a_j)(x_j + B_1) - (B_1 + C_1 a_j)$$

At equilibrium  $\frac{da_j}{dt} = 0$  and

$$a_j = \frac{x_j}{1 + A_1(x_j + B_1) + C_1} \quad (6.3)$$

i.e. neurons who received non-zero input ( $x_j \neq 0$ ) have a positive activation ( $a_j > 0$ ) and the neurons who received a zero input have their activation *raised* to zero.

---

<sup>1</sup>See also the  $F_2$  section, below

- ③ Input is active ( $\mathbf{x} \neq \hat{\mathbf{0}}$ ) and  $F_2$  is also active ( $f_2(\mathbf{a}') \neq \hat{0}$ ). Then the gain control unit is deactivated ( $g = 0$ ) and (6.2) becomes:

$$\frac{da_j}{dt} = -a_j + (1 - A_1 a_j)[x_j + D_1 W(j, :) f_2(\mathbf{a}')] - (B_1 + C_1 a_j)$$

At equilibrium  $\frac{da_j}{dt} = 0$  and

$$a_j = \frac{x_j + D_1 W(j, :) f_2(\mathbf{a}') - B_1}{1 + A_1(x_j + D_1 W(j, :) f_2(\mathbf{a}')) + C_1} \quad (6.4)$$

The following cases may be discussed here:

- (a) Input is maximum:  $x_j = 1$  and input from  $F_2$  is minimum:  $\mathbf{a}' \rightarrow \hat{\mathbf{0}}$ . Because the gain control unit have been deactivated and the activity of  $F_2$  layer is dropping to  $\hat{0}$  then — according to the *2/3 rule* — the neuron have to switch to inactive state and consequently  $a_j$  have to switch to a negative value. From (6.4):

$$\lim_{\mathbf{a}' \rightarrow \hat{\mathbf{0}}} a_j = \lim_{\mathbf{a}' \rightarrow \hat{\mathbf{0}}} \left[ \frac{x_j + D_1 W(j, :) f_2(\mathbf{a}') - B_1}{1 + A_1(x_j + D_1 W(j, :) f_2(\mathbf{a}')) + C_1} \right] < 0 \Rightarrow B_1 > 1 \quad (6.5)$$

(as all constants  $A_1, B_1, C_1, D_1$  were positive definite).

- (b) Input is maximum:  $x_j = 1$  and input from  $F_2$  is non-zero.  $F_2$  layer is of a contrast enhancement type ("winner takes all") and it have only (maximum) one winner, let  $k$  be that one, i.e.  $W(j, :) f_2(\mathbf{a}') = w_{jk} f_2(a'_k)$ . Then according to the *2/3 rule* the neuron is active and the activation value should be  $a_j > 0$  and (6.4) becomes:

$$1 + D_1 w_{jk} f_2(a'_k) - B_1 > 0 \Rightarrow w_{jk} f_2(a'_k) > \frac{B_1 - 1}{D_1} \quad (6.6)$$

There is a discontinuity between this condition and the preceding one: from (6.6), if  $w_{jk} f_2(a'_k) \rightarrow 0$  then  $B_1 - 1 < 0$  which seems to be in contradiction with the previous (6.5) condition. Consequently this condition will be imposed on  $W$  weights and *not* on constants  $B_1$  and  $D_1$ .

- (c) Input is maximum  $x_j = 1$  and input from  $F_2$  is maximum, i.e.  $w_{jk} f_2(a'_k) = 1$  (see above,  $k$  is the  $F_2$  winning neuron). Then (6.4) gives:

$$1 + D_1 - B_1 > 0 \Rightarrow B_1 < D_1 + 1 \quad (6.7)$$

As  $f_2(a'_k) = 1$  (maximum) and because of the choosing of  $D_1$  constant:  $w_{jk} \in [0, 1]$  then  $w_{jk} = 1$ .

- (d) Input is minimum  $\mathbf{x} \rightarrow \hat{\mathbf{0}}$  and input from  $F_2$  is maximum. Similarly to the first case above (and (6.4)):

$$\lim_{\mathbf{x} \rightarrow \hat{\mathbf{0}}} a_j = \lim_{\mathbf{x} \rightarrow \hat{\mathbf{0}}} \left[ \frac{x_j + D_1 W(j, :) f_2(\mathbf{a}') - B_1}{1 + A_1[x_j + D_1 W(j, :) f_2(\mathbf{a}')] + C_1} \right] < 0 \Rightarrow D_1 < B_1 \quad (6.8)$$

(because of the *2/3 rule* at limit the  $F_1$  neuron have to switch to negative state and subsequently have a negative activation).

- (e) Input is minimum ( $x \rightarrow \hat{0}$ ) and input from  $F_2$  is also minimum ( $a' \rightarrow \hat{0}$ ). 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:

$$\lim_{\substack{x \rightarrow \hat{0} \\ a' \rightarrow \hat{0}}} a_j < 0 \Rightarrow -B_1 < 0$$

which is useless because anyway  $B_1$  constant is positive definite.

Combining all of the above requirements (6.5), (6.7), and (6.8) in one gives:

$$\max(1, D_1) < B_1 < D_1 + 1$$

which represents one of the condition to be put on  $F_1$  constants such that the *2/3 rule* will operate.

❖  $f_1$  The output value for the  $j$ -th  $F_1$  neuron is obtained by applying the following activation function:

$$f_1(a_j) = \begin{cases} 1 & \text{if activation } a_j > 0 \\ 0 & \text{if activation } a_j \leq 0 \end{cases} \quad (6.9)$$

### 6.2.2 The $F_2$ layer

- ① Initially the network is started at a “0” state. There are no signals traveling internally and no input ( $x = \hat{0}$ ). So, the output of the gain control unit is 0. Even if the  $F_2$  layer receive a direct input from the outside environment (another network, e.t.c.) the *2/3 rule* stops  $F_2$  from sending any output.
- ② Once an input have arrived on  $F_1$  the output of the gain control unit switches to 1, because the output of  $F_2$  ( $z'$ ) is still 0.

Now the output of  $F_2$  is allowed. There are two cases:

- (a) There is already an input from outside, then the  $F_2$  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 have to wait for the output of  $F_1$  before being able to send an output.

The conclusion is that on  $F_2$  level the gain control unit is used just to turn on/off the right of  $F_2$  to send an output. Because the output of the gain control unit (i.e. 1) is sent uniformly to all neurons it doesn't play any other active role and can be left out from the equations describing the behavior of  $F_2$  units.



#### Remarks:

- In fact the equation describing the activation of the  $F_2$  neuron is of the form:

$$\text{output equation : } \begin{cases} \frac{da'_k}{dt} = -a'_k + (1 - A_2 a'_k) \times \text{excitatory input} & \text{if } g = 1 \\ & - (B_2 + C_2 a'_k) \times \text{inhibitory input} \\ a'_k = 0 & \text{otherwise} \end{cases}$$

where  $a'_k$  is the neuron activation and  $A_2$ ,  $B_2$  and  $C_2$  are positive constants (see (6.1)). The first part is analyzed below.

The neuron  $k$  on  $F_2$  layer receives an *excitatory* input from  $F_1$ :  $W'(k,:) f_1(\mathbf{a})$  (where  $W'$  is the weight matrix of connections from  $F_1$  to  $F_2$ ) and from itself:  $h(a'_k)$ ;  $h$  being the feedback function:

$$\text{excitatory input} = W'(k,:) f_1(\mathbf{a}) + h(a'_k)$$

The same  $k$  neuron receive an *direct inhibitory* input from all other  $F_2$  neurons:  $\sum_{\substack{\ell=1 \\ \ell \neq k}}^K h(a'_\ell)$

and an *indirect inhibitory* input:  $\sum_{\substack{\ell=1 \\ \ell \neq k}}^K W'(\ell,:) f_1(\mathbf{a})$ , where  $K$  is the number of neurons on  $F_2$ . The latter term represents the indirect inhibition (feedback) due to the fact that others neurons will have an positive output (because of their input), while the former is due to *direct* inter-connections (lateral feedback) between neurons in  $F_2$  layer.

$$\text{inhibitory input} = \sum_{\substack{\ell=1 \\ \ell \neq k}}^K h(a'_\ell) + \sum_{\substack{\ell=1 \\ \ell \neq k}}^K W'(\ell,:) f_1(\mathbf{a})$$

Eventually, from (6.1):

$$\begin{aligned} \frac{da'_k}{dt} = & -a'_k + (1 - A_2 a'_k)(D_2 W(k,:) f_1(\mathbf{a}) + h(a'_k)) \\ & - (B_2 + C_2 a'_k) \sum_{\substack{\ell=1 \\ \ell \neq k}}^K [h(a'_\ell) + W'(\ell,:) f_1(\mathbf{a})] \end{aligned}$$

where  $D_2$  is a multiplying positive constant.

Let  $B_2 = 0$ ,  $C_2 = A_2$  and  $D_2 = 1$ . Then:

$$\frac{da'_k}{dt} = -a'_k + h(a'_k) + W'(k,:) f_1(\mathbf{a}) - A_2 a'_k \sum_{\ell=1}^K [h(a'_\ell) + W'(\ell,:) f_1(\mathbf{a})] \quad (6.10)$$

or in matrix notation:

$$\frac{d\mathbf{a}'}{dt} = -\mathbf{a}' + h(\mathbf{a}') + W' f_1(\mathbf{a}) - A_2 \hat{\mathbf{1}}^T [h(\mathbf{a}') + W' f_1(\mathbf{a})] \mathbf{a}'$$

For an feedback function of the form  $h(a'_k) = a'^m_k$ , where  $m > 1$ , the above equations define an competitive layer.

*Proof.* First the following *change of variable* is performed:

$\diamond \tilde{a}'_k, a'_{\text{tot}}$

$$\begin{aligned} \tilde{a}'_k &= \frac{a'_k}{\sum_{\ell=1}^K a'_\ell} \quad \text{and} \quad a'_{\text{tot}} = \sum_{\ell=1}^K a'_\ell \Rightarrow \\ a'_k &= \tilde{a}'_k a'_{\text{tot}} \quad \text{and} \quad \frac{da'_k}{dt} = \frac{d\tilde{a}'_k}{dt} a'_{\text{tot}} + \tilde{a}'_k \frac{da'_{\text{tot}}}{dt} \end{aligned}$$

By doing a summation for all  $k$  in (6.10) and using the change of variable just introduced:

$$\frac{da'_{\text{tot}}}{dt} = -a'_{\text{tot}} + \sum_{\ell=1}^K h(\tilde{a}'_{\ell} a'_{\text{tot}}) + \sum_{\ell=1}^K W'(\ell, :) f_1(\mathbf{a}) - A_2 a'_{\text{tot}} \sum_{\ell=1}^K [h(\tilde{a}'_{\ell} a'_{\text{tot}}) + W'(\ell, :) f_1(\mathbf{a})] \quad (6.11)$$

Then, from the substitution introduced and (6.10) and (6.11):

$$\frac{d\tilde{a}'_k}{dt} a'_{\text{tot}} = \frac{da'_k}{dt} - \tilde{a}'_k \frac{da'_{\text{tot}}}{dt} = h(\tilde{a}'_k a'_{\text{tot}}) + W'(k, :) f_1(\mathbf{a}) - \tilde{a}'_k \sum_{\ell=1}^K [h(\tilde{a}'_{\ell} a'_{\text{tot}}) + W'(\ell, :) f_1(\mathbf{a})]$$

As  $\sum_{\ell=1}^K \tilde{a}'_k = 1$ , the above equation may be rewritten as:

$$\begin{aligned} \frac{d\tilde{a}'_k}{dt} a'_{\text{tot}} &= \sum_{\ell=1}^K \tilde{a}'_{\ell} h(\tilde{a}'_k a'_{\text{tot}}) - \sum_{\ell=1}^K \tilde{a}'_k h(\tilde{a}'_{\ell} a'_{\text{tot}}) + W(k, :) f_1(\mathbf{a}) - \tilde{a}'_k \sum_{\ell=1}^K W(\ell, :) f_1(\mathbf{a}) \\ &= \tilde{a}'_k a'_{\text{tot}} \sum_{\ell=1}^K \tilde{a}'_{\ell} \left[ \frac{h(\tilde{a}'_k a'_{\text{tot}})}{\tilde{a}'_k a'_{\text{tot}}} - \frac{h(\tilde{a}'_{\ell} a'_{\text{tot}})}{\tilde{a}'_{\ell} a'_{\text{tot}}} \right] + W(k, :) f_1(\mathbf{a}) - \tilde{a}'_k \sum_{\ell=1}^K W(\ell, :) f_1(\mathbf{a}) \end{aligned}$$

and on this formula the following cases may be considered:

- Identity function:  $h(\tilde{a}'_k a'_{\text{tot}}) = \tilde{a}'_k a'_{\text{tot}}$  then:

$$\frac{d\tilde{a}'_k}{dt} a'_{\text{tot}} = W'(k, :) f_1(\mathbf{a}) - \tilde{a}'_k \sum_{\ell=1}^K W'(\ell, :) f_1(\mathbf{a})$$

and the stable value (obtained from  $\frac{d\tilde{a}'_k}{dt} = 0$ ) is:

$$\tilde{a}'_k = \frac{W'(k, :) f_1(\mathbf{a})}{\sum_{\ell=1}^K W'(\ell, :) f_1(\mathbf{a})} \Rightarrow a'_k \propto \frac{W'(k, :) f_1(\mathbf{a})}{\sum_{\ell=1}^K W'(\ell, :) f_1(\mathbf{a})}$$

i.e. the output is proportional to the weighted sum of inputs.

- Square function:  $h(\tilde{a}'_k a'_{\text{tot}}) = (\tilde{a}'_k a'_{\text{tot}})^2$  then  $\frac{h(\tilde{a}'_k a'_{\text{tot}})}{\tilde{a}'_k a'_{\text{tot}}} - \frac{h(\tilde{a}'_{\ell} a'_{\text{tot}})}{\tilde{a}'_{\ell} a'_{\text{tot}}}$  reduces to  $a'_{\text{tot}}(\tilde{a}'_k - \tilde{a}'_{\ell})$  which for  $\tilde{a}'_k > \tilde{a}'_{\ell}$  represents an amplification while for  $\tilde{a}'_k < \tilde{a}'_{\ell}$  represents an inhibition. The  $W'(k, :) f_1(\mathbf{a})$  is constant.

The  $F_2$  layer acts as an “winner-takes-all”<sup>2</sup> network; the distance between neurons with large output and those with small output widens — eventually only one neuron will have a non-zero output.

The same discussion goes for  $h(\tilde{a}'_k a'_{\text{tot}}) = (\tilde{a}'_k a'_{\text{tot}})^m$  where  $m > 1$ .  $\square$

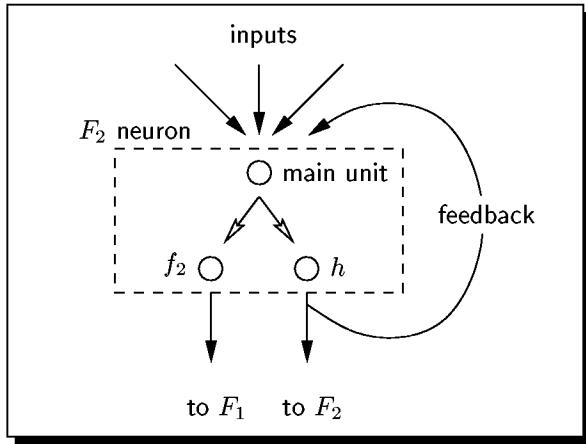
❖  $f_2$  The winning  $F_2$  neuron sends a value of 1 to  $F_1$  layer, all other send 0. Let  $f_2(a')$  be the output (activation) function which value is sent to  $F_1$ :

$$f_2(a'_k) = \begin{cases} 1 & \text{if } a'_k = \max_{\ell=1,K} a'_{\ell} \\ 0 & \text{otherwise} \end{cases} \quad (6.12)$$

### Remarks:

- ➔ It seems — at first sight — that the  $F_2$  neuron have two outputs: one sent to the  $F_2$  layer —  $h$  function and one sent to the  $F_1$  layer —  $f_2$  function. This runs counter 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 calculate the activation  $a'_k$  and send the

<sup>2</sup>There is a strong similarity with the functionality of the hidden layer of a counterpropagation network



**Figure 6.2:** The  $F_2$  neuron structure.

result (it have the identity function as activation function) 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.

### 6.2.3 Learning on $F_1$ : The $W$ weights

The differential equations describing the  $F_1$  learning process, (i.e.  $W$  weights adaptation) are:

$$\frac{dw_{jk}}{dt} = [-w_{jk} + f_1(a_j)]f_2(a'_k) \quad , \quad j = \overline{1, N}, k = \overline{1, K} \quad (6.13)$$

There is just one (at most) “winner” — let  $k$  be that one — on  $F_2$  for which  $f_2(a'_k) \neq 0$ , for all others  $f_2(a'_\ell) = 0$ , i.e. only the weights related to the winning  $F_2$  neuron are adapted on  $F_1$  layer, all other remain unchanged (for a given input).



#### Remarks:

- During the learning process only one (at most) component of the weight vector  $W(j, :)$  changes for each  $F_1$  neuron, i.e. only column  $k$  of  $W$  changes,  $k$  being the  $F_2$  winner.

Because of the definition of the  $f_1$  and  $f_2$  functions (see (6.9) and (6.12)) the following cases may be considered:

- ①  $F_2:k$  neuron winner ( $f_2(a'_k) = 1$ ) and  $F_1:j$  neuron active ( $f_1(a_j) = 1$ ), then:

$$\frac{dw_{jk}}{dt} = -w_{jk} + 1 \quad \Rightarrow \quad w_{jk} = 1 - e^{-t}$$

(solution found by searching first for the solution for the homogeneous equation and then making the “constant” time dependent to find the general solution). The weight asymptotically approaches 1 for  $t \rightarrow \infty$ .

②  $F_2:k$  neuron winner ( $f_2(a'_k) = 1$ ) and  $F_1:j$  neuron non-active ( $f_1(a_j) = 0$ ), then:

$$\frac{dw_{jk}}{dt} = -w_{jk} \Rightarrow w_{jk} = w_{jk}(0) e^{-t}$$

where  $w_{jk}(0)$  is the initial value at  $t = 0$ . The weight asymptotically decreases to 0 for  $t \rightarrow \infty$ .

③  $F_2:k$  neuron non-winner ( $f_2(a'_k) = 0$ ), then:

$$\frac{dw_{jk}}{dt} = 0 \Rightarrow w_{jk} = \text{const.}$$

the weights do not change.

A supplementary condition for  $W$  weights is required in order for 2/3 rule to function, see (6.6):

$$w_{jk} > \frac{B_1 - 1}{D_1} \quad (6.14)$$

i.e. all weights have to be initialized to a value greater than  $\frac{B_1 - 1}{D_1}$ . Otherwise the  $F_1:i$  neuron is kept into an inactive state and the weights decrease to 0 (or do not change).

**Fast Learning:** If the  $F_2:k$  and  $F_1:j$  neurons are both active then the weight  $w_{jk} \rightarrow 1$ , otherwise it decays towards 0 or remain unchanged. A fast way to achieve the learning is to set the weights to their asymptotic values as soon as possible, i.e. knowing the neuronal activities:

$$w_{jk} = \begin{cases} 1 & \text{if } j, k \text{ neurons are active} \\ \text{no change} & \text{if } k \text{ neuron is non-active} \\ 0 & \text{otherwise} \end{cases} \quad (6.15)$$

or in matrix notation:

$$W(:, k)_{\text{new}} = f_1(\mathbf{a}) \quad \text{and} \quad W(:, \ell)_{\text{new}} = W(:, \ell)_{\text{old}} \text{ for } \ell \neq k$$

*Proof.* Only column  $k$  of  $W$  is to be changed (weights related to  $F_2$  winner).  $f_1(\mathbf{a})$  is 1 for active neuron, 0 otherwise, see (6.9).  $\square$

#### 6.2.4 Learning on $F_2$ : The $W'$ weights

The differential equations describing the  $F_2$  learning process are:

$$\frac{dw'_{kj}}{dt} = E \left[ F(1 - w'_{kj})f_1(a_j) - w'_{kj} \sum_{\substack{\ell=1 \\ \ell \neq j}}^N f_1(a_\ell) \right] f_2(a'_k)$$

❖  $E, F = \text{const.}$  and, because  $\sum_{\substack{\ell=1 \\ \ell \neq j}}^N f_1(a_\ell) = \sum_{\ell=1}^N f_1(a_\ell) - f_1(a_j)$ , the equations may be

rewritten as:

$$\frac{dw'_{kj}}{dt} = E \left[ F(1 - w'_{kj})f_1(a_j) - w'_{kj} \left( \sum_{\ell=1}^N f_1(a_\ell) - f_1(a_j) \right) \right] f_2(a'_k)$$

For all neurons  $\ell$  on  $F_2$  except winner  $f_2(a'_\ell) = 0$  so only the winner's weights are adapted, all other remain unchanged (for a given input).

Analogous previous  $W$  weights, and see also (6.9) and (6.12), the following cases are discussed:

- ①  $F_2:k$  neuron winner ( $f_2(a'_k) = 1$ ) and  $F_1:j$  neuron active ( $f_1(a_j) = 1$ ) then:

$$\frac{dw'_{kj}}{dt} = E \left[ F - w'_{kj} \left( F - 1 + \sum_{\ell=1}^N f_1(a_\ell) \right) \right]$$

and the solution is of the form:

$$w'_{kj} = \frac{F}{F - 1 + \sum_{\ell=1}^N f_1(a_\ell)} - \exp \left[ -E \left( F - 1 + \sum_{\ell=1}^N f_1(a_\ell) \right) t \right]$$

(found analogous  $W$ ). The weight asymptotically approaches  $\frac{F}{F - 1 + \sum_{\ell=1}^N f_1(a_\ell)}$  for  $t \rightarrow \infty$ .

In extreme case it may be possible that  $\sum_{\ell=1}^N f_1(a_\ell) = 0$  such that the condition  $F > 1$  have to be imposed to keep weights positive.

- ②  $F_2:k$  neuron winner ( $f_2(a'_k) = 1$ ) and  $F_1:j$  neuron non-active ( $f_1(a_j) = 0$ ), then:

$$\frac{dw'_{kj}}{dt} = -Ew'_{kj} \sum_{\ell=1}^N f_1(a_\ell) \Rightarrow w'_{kj} = w'_{kj}(0) \exp \left( -Et \sum_{\ell=1}^N f_1(a_\ell) \right)$$

where  $w'_{kj}(0)$  is the initial value at  $t = 0$ . The weight asymptotically decreases to 0 for  $t \rightarrow \infty$ .

- ③  $F_2:k$  neuron non-winner ( $f_2(a'_k) = 0$ ) then:

$$\frac{dw'_{kj}}{dt} = 0 \Rightarrow w'_{kj} = \text{const.}$$

the weight do not change.

**Fast learning:** If the  $F_2:k$  and  $F_1:j$  neurons are both active then the weight  $w'_{kj} \rightarrow 1$ , otherwise it decays towards 0 or remain unchanged. A fast way to achieve the learning is to set the weights to their asymptotic values as soon as possible, i.e. knowing the neuronal activities:

$$w'_{kj} = \begin{cases} \frac{F}{F - 1 + \sum_{\ell=1}^N f_1(a_\ell)} & \text{if } k, j \text{ neurons are active} \\ \text{no change} & \text{if } k \text{ non-active} \\ 0 & \text{otherwise} \end{cases} \quad (6.16)$$

or, in matrix notation:

$$W'(k, :)_{\text{new}} = \frac{F}{F - 1 + \hat{\mathbf{1}}^T f_1(\mathbf{a})} f_1(\mathbf{a}^T) \quad \text{and} \quad W'(\ell, :)_{\text{new}} = W'(\ell, :)_{\text{old}} \text{ for } \ell \neq k \quad (6.17)$$

*Proof.* Only row  $k$  of  $W'$  is to be changed (weights related to  $F_2$  winner).  $f_1(a)$  is 1 for active neuron, 0 otherwise, see (6.9).  $\square$

### 6.2.5 Subpatterns

❖  $\mathbf{x}', \mathbf{x}'', k', k''$

The input vector is binary  $x_i = 0$  or 1. All patterns are subpatterns of the unit input vector  $\hat{\mathbf{1}}$ . Also it is possible that a pattern  $\mathbf{x}'$  may be a subpattern of another input vector  $\mathbf{x}''$ :  $\mathbf{x}' \subset \mathbf{x}''$ , i.e. either  $x'_i = x''_i$  or  $x'_i = 0$ . The ART1 network ensures that the proper  $F_2$  neuron will win such case may occur, i.e. the winning neurons must be different for  $\mathbf{x}'$  and  $\mathbf{x}''$  — let  $k'$  and  $k''$  be those ones.  $k' \neq k''$  if  $\mathbf{x}'$  and  $\mathbf{x}''$  correspond to different classes.

*Proof.* When first presented with an input vector the  $F_1$  layer outputs the same vector and distribute it to  $F_2$  layer, see (6.3) and (6.9) (the change in  $F_1$  activation pattern later is used just to reset the  $F_2$  layer). So at this stage  $f_1(a) = \mathbf{x}$ .

Assuming that  $k'$  neuron have learned  $\mathbf{x}'$ , its weights should be (from (6.17)):

$$W'(k', :) = \frac{F}{F - 1 + \hat{\mathbf{1}}^T \mathbf{x}'} \mathbf{x}'^T \quad (6.18)$$

and for  $k''$  neuron which have learned  $\mathbf{x}''$  its weights should be:

$$W'(k'', :) = \frac{F}{F - 1 + \hat{\mathbf{1}}^T \mathbf{x}''} \mathbf{x}''^T$$

When  $\mathbf{x}'$  is presented as input, total input to  $k'$  neuron is (output of  $F_1$  is  $\mathbf{x}'$ ):

$$a'_{k'} = W'(k', :) \mathbf{x}' = \frac{F}{F - 1 + \hat{\mathbf{1}}^T \mathbf{x}'} \mathbf{x}'^T \mathbf{x}'$$

while the total input to  $k''$  neuron is

$$a'_{k''} = W'(k'', :) \mathbf{x}' = \frac{F}{F - 1 + \hat{\mathbf{1}}^T \mathbf{x}''} \mathbf{x}''^T \mathbf{x}'$$

Because  $\mathbf{x}' \subset \mathbf{x}''$  then  $\mathbf{x}''^T \mathbf{x}' = \mathbf{x}'^T \mathbf{x}'$  but  $\hat{\mathbf{1}}^T \mathbf{x}'' > \hat{\mathbf{1}}^T \mathbf{x}'$  and then  $a'_{k''} < a'_{k'}$  and  $k'$  wins as it should.

Similarly when  $\mathbf{x}''$  is presented as input, the output of  $k'$  neuron is

$$a'_{k'} = W'(k', :) \mathbf{x}'' = \frac{F}{F - 1 + \hat{\mathbf{1}}^T \mathbf{x}'} \mathbf{x}'^T \mathbf{x}''$$

while the total input to  $k''$  neuron is

$$a'_{k''} = W'(k'', :) \mathbf{x}'' = \frac{F}{F - 1 + \hat{\mathbf{1}}^T \mathbf{x}''} \mathbf{x}''^T \mathbf{x}''$$

As  $\mathbf{x}' \subset \mathbf{x}''$  and are binary vectors then

$$\mathbf{x}'^T \mathbf{x}'' = \mathbf{x}'^T \mathbf{x}' = \hat{\mathbf{1}}^T \mathbf{x}' \quad , \quad \mathbf{x}''^T \mathbf{x}'' = \hat{\mathbf{1}}^T \mathbf{x}'' \quad \text{and} \quad \hat{\mathbf{1}}^T \mathbf{x}' < \hat{\mathbf{1}}^T \mathbf{x}'' \Rightarrow$$

$$a'_{k'} = \frac{F}{\frac{F-1}{\hat{\mathbf{1}}^T \mathbf{x}'} + 1} < a'_{k''} = \frac{F}{\frac{F-1}{\hat{\mathbf{1}}^T \mathbf{x}''} + 1}$$

and neuron  $k''$  wins as it should.  $\square$

#### Remarks:

- ➔ The input patterns are assumed non zero otherwise  $\mathbf{x} = \hat{\mathbf{0}}$  means no activation.
- ➔ All inputs are subpatterns of the unit vector  $\hat{\mathbf{1}}$  and the neuron who have learned the unit vector have the smallest weights:

$$W'(k, :) = \frac{F}{F - 1 + N} \hat{\mathbf{1}}^T$$

and smallest output (when the unit vector is presented as input)

$$a'_k = \frac{F}{\frac{F-1}{N} + 1}$$

- The  $F_2$  neurons which aren't used yet should not win over neurons which were already committed to an input vector.

Then the weights of unused neurons have to be initialized such that they do not win in the worst case, i.e. when  $\hat{\mathbf{1}}$  have been already committed to a neuron. Uncommitted neuronal weights have to be initialized with values:

$$w'_{kj} \in \left(0, \frac{F}{F - 1 + N}\right)$$

(the 0 have to be avoided because it will 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 won then only one of the uncommitted neurons wins.

### 6.2.6 The Reset Unit

The reset neuron is set to detect mismatches between the input vector and the output of the  $F_1$  layer.

At start, when an input is present, the output of  $F_1$  is identical to the input and the reset unit should not activate.

Also the reset unit 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 86.

Let  $Q$  be the weight(s) for inputs and  $-S$  the weight(s) for  $F_1$  connections ( $Q, S > 0$ ). ♦  $Q, S, a_R$   
The total input to the reset unit is  $a_R$ :

$$a_R = Q \sum_{i=1}^N x_i - S \sum_{i=1}^N f_1(a_i) , \quad Q, S = \text{const.} , \quad Q, S \in \mathbb{R}^+$$

The reset unit activates if the net input is positive:

$$Q \sum_{i=1}^N x_i - S \sum_{i=1}^N f_1(a_i) > 0 \Leftrightarrow \frac{\sum_{i=1}^N f_1(a_i)}{\sum_{i=1}^N x_i} < \frac{Q}{S} = \rho$$

where  $\rho$  is called the *vigilance parameter*. For  $\frac{\sum_{i=1}^N f_1(a_i)}{\sum_{i=1}^N x_i} \geq \rho$  the reset unit does not trigger. ♦  $\rho$

Because at the beginning (before  $F_2$  activate)  $f_1(\mathbf{a}) = \mathbf{x}$  then the vigilance parameter should be  $\rho \leq 1$ , i.e.  $Q \leq S$  (otherwise it will always trigger).

### **Noise in data**

The vigilance parameter self scale the difference between the actual input pattern and the stored/learned one. Depending upon the input pattern the difference (noise, distance, e.t.c.) between the two may or may not trigger the reset unit.

For the same difference (same set of ones in input vector) the ratio between noise and information varies depending upon the number of ones in the input vector, i.e. assuming that the noise vector is the smallest one  $\mathbf{x}^T = (0 \dots 1 \dots 0)$  (just one "1") and the input stored vector is similar (have also just one "1") then for an input with noise the ratio between noise and data is at least 1 : 1; for an input having two ones the ratio drops to half 0.5 : 1 and so on.

### **New pattern learning**

If the reset unit is activated then it deactivates the winning  $F_2$  neuron for a sufficient 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) neuron  $k$  (from  $F_2$ ) win. The activity of the  $F_1$  neurons are:

$$a_j = \frac{x_j + D_1 W(j,:) f_2(\mathbf{a}') - B_1}{1 + A_1[x_j + D_1 W(j,:) f_2(\mathbf{a}')] + C_1} = \frac{x_j + D_1 w_{jk} - B_1}{1 + A_1(x_j + D_1 w_{jk}) + C_1}$$

(see (6.4) and because  $f_2(\mathbf{a}')$  is 1 just for winner  $k$  and zero in rest and then  $W(j,:) f_2(\mathbf{a}') = w_{jk}$ ). For newly committed  $F_2$  neurons the weights (from  $F_2$  to  $F_1$ ) are initialized to a value  $w_{jk} > \frac{B_1 - 1}{D_1}$  (see (6.14)) and then:

$$x_j + D_1 w_{jk} - B_1 > x_j - 1$$

which means that for  $x_j = 1$  the activity  $a_j$  is positive and  $f_1(a_j) = 1$  while for  $x_j = 0$ , because of the 2/3 rule, the activity is negative and  $f_1(a_j) = 0$ .

Conclusion: when a new  $F_2$  neuron is committed to learn the input, the output of  $F_1$  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 have enough neurons for all classes to be learned, 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**

Let assume that an input vector, which is similar to a stored/learned one, is presented to the network. Let consider the case where some data is missing, i.e. some components of input are 0 where the stored one have 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 winning  $F_1$  neuron to the  $F_2$  layer will have more 1-s than the input pattern (after one transmission cycle between  $F_{1,2}$ ,

layers). The corresponding weights (see (6.15):  $j$  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 are 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 unassigned  $F_2$  neuron may be committed to learn.

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.

## 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 require to be divided into “subclasses” with different assigned  $F_2$  winning neurons.
2. Select the constants:  $A_1 > 0$ ,  $C_1 > 0$ ,  $D_1 > 0$ ,  $\max(1, D_1) < B_1 < D_1 + 1$ ,  $F > 1$  and  $\rho \in (0, 1]$ .
3. Initialize the  $W$  weights with random values such that:

$$w_{jk} > \frac{B_1 - 1}{D_1} \quad , \quad j = \overline{1, N}, k = \overline{1, K}$$

4. Initialize  $W'$  weights with random values such that:

$$w'_{kj} \in \left(0, \frac{F}{F - 1 + n}\right) \quad , \quad k = \overline{1, K}, j = \overline{1, N}$$

### **Network running and learning**

The algorithm uses the fast learning method (asymptotic values for weights).

1. Apply an input vector  $\mathbf{x}$ . The  $F_1$  output becomes

$$f_1(\mathbf{a}) = \mathbf{x}$$

(at first run,  $\mathbf{x}$  is a binary vector).

2. Calculate the activities of  $F_2$  neurons and find the winner. The neuron with the biggest input from  $F_1$  wins (and all other will have zero output). For  $k$  the  $F_2$  winner it is true that:

$$W'(k, :) f_1(\mathbf{a}) = \max_{\ell} W'(\ell, :) f_1(\mathbf{a})$$

3. Calculate the new activities of  $F_1$  neurons caused by inputs from  $F_2$ . The  $F_2$  output is a vector which have all its components 0 with one exception for winner  $k$ , multiplying  $W$  by such a vector means that column  $W(:, k)$  is selected to become the activity of  $F_1$  and the new  $F_1$  output becomes:

$$f_1(\mathbf{a})_{\text{new}} = \text{sign}(W(:, k))$$

4. Calculate the “degree of match” between input and the new output of  $F_1$  layer

$$\text{degree of match} = \frac{\sum_{j=1}^N f_1(a_j)_{\text{new}}}{\sum_{j=1}^N x_j} = \frac{\hat{\mathbf{1}}^T f_1(\mathbf{a})_{\text{new}}}{\hat{\mathbf{1}}^T \mathbf{x}}$$

5. Compare the “degree of match” computed previously with the vigilance parameter  $\rho$ .

If the vigilance parameter is bigger than the “degree of match” then

- (a) Mark the  $F_2:k$  neuron as inactive for the rest of the cycle 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  $k$ .

6. Update the weights (if learning is enabled; it have to be always enabled for new classes). See (6.15) and (6.16).

$$W(:, k)_{\text{new}} = f_1(\mathbf{a})_{\text{new}}$$

$$W'(k, :)_{\text{new}} = \frac{F}{F - 1 + \hat{\mathbf{1}}^T f_1(\mathbf{a})_{\text{new}}} f_1(\mathbf{a}^T)_{\text{new}}$$

only the weights related to the winning  $F_2$  neuron being updated.

7. 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 The ART2 Architecture

Unlike ART1, the ART2 network is designed to work with analog *positive* inputs. There is a broad similarity with ART1 architecture: there is a  $F_1$  layer which sends its output to a  $F_2$  layer and a reset *layer*. The  $F_2$  layer is of “winner-takes-all” type and the reset unit have the same role as in ART1. However the  $F_1$  layer is made of 6 sublayers labeled  $w, s, u, v, p$  and  $q$ .

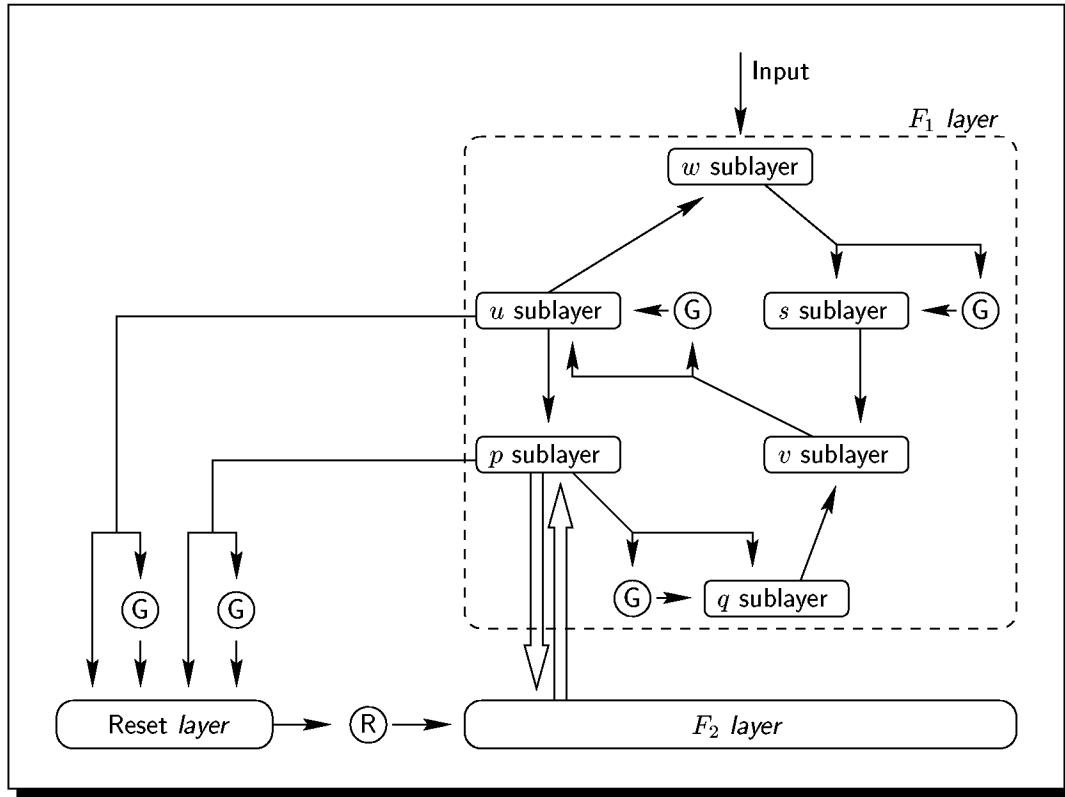
❖  $w, s, u, v, p, q$

See figure 6.3 on the facing page. Each of the sublayers have the same number of neurons as the number of components in the input vector.

- ① The input vector is sent to the  $w$  sublayer.
- ② The output to 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.
- ⑤ The output of the  $u$  sublayer is sent to the  $p$  sublayer, to the reset layer  $r$  and *back* to the  $w$  sublayer.

---

<sup>6.4</sup>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  $G$  units are gain-control neurons which sent an inhibitory signal; the  $R$  unit is the reset neuron.

- 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  $j$  from one layer to the corresponding neuron  $j$  from the other layer)
- ➔ All (sub)layers receive input from 2 sources, a supplementary gain-control neuron have been added where necessary such that the layers may be compliant with the 2/3 rule.
- ➔ The gain-control unit have the role to normalize the output of the corresponding layers (see also (6.20) equations); note that all layers have 2 sources of input either from 2 layers or from a layer and a gain-control unit. The gain-control neuron receive input from all neurons from the corresponding sublayer and send the sum of its input as *inhibitory* input (see also (6.21) and table 6.1), while the other layers sent an *excitatory* input.

## 6.5 ART2 Dynamics

### 6.5.1 The $F_1$ layer

❖  $a, b, e$

The differential equations governing the  $F_1$  sublayers behavior are:

$$\begin{aligned}\frac{dw_j}{dt} &= -w_j + x_j + au_j & \frac{dv_j}{dt} &= -v_j + f(s_j) + bf(q_j) \\ \frac{ds_j}{dt} &= -es_j + w_j - s_j\|\mathbf{w}\| & \frac{dp_j}{dt} &= -p_j + u_j + W(j,:)f_2(\mathbf{a}') \\ \frac{du_j}{dt} &= -eu_j + v_j - u_j\|\mathbf{v}\| & \frac{dq_j}{dt} &= -eq_j + p_j - q_j\|\mathbf{p}\|\end{aligned}$$

❖  $f, \alpha$

where  $a, b, e = \text{const.}$ . The  $f$  function determines the contrast enhancement which takes place inside  $F_1$  layer, a possible definition would be

$$f(x) = \begin{cases} 0 & \text{for } x < \alpha \\ x & \text{otherwise} \end{cases} \quad (6.19)$$

where  $\alpha \in (0, 1)$ ,  $\alpha = \text{const.}$ . The norm of a vector  $\mathbf{x}$  is here defined as the Euclidean. When applied to a vector the contrast enhancement function may be written in matrix format as:

$$f(\mathbf{x}) = \mathbf{x} \odot \text{sign}(\text{sign}(\mathbf{x} - \alpha\hat{\mathbf{1}}) + \hat{\mathbf{1}})$$

The equilibrium values (from  $\frac{d(\cdot)}{dt} = 0$ ) are:

$$\begin{aligned}w_j &= s_j + au_j & v_j &= f(s_j) + bf(q_j) \\ s_j &= \frac{w_j}{e + \|\mathbf{w}\|} & p_j &= u_j + W(j,:)f_2(\mathbf{a}') \\ u_j &= \frac{v_j}{e + \|\mathbf{v}\|} & q_j &= \frac{p_j}{e + \|\mathbf{p}\|}\end{aligned} \quad (6.20)$$

These results may be described by the means of one single equation with different parameters for different sub-layers — see table 6.1:

$$\begin{aligned}\frac{d(\text{neuron output})}{dt} &= -C_1 \text{neuron output} + \text{excitatory input} \\ &\quad - C_2 \text{neuron output} \times \text{inhibitory input}\end{aligned} \quad (6.21)$$



#### Remarks:

- ➔ The same (6.21) is applicable to the reset layer with the parameters in table 6.1 ( $c = \text{const.}$ ).
- ➔ The purpose of the  $e$  constant is to limit the output of  $s, q, u$  and  $r$  (sub)layers when their input is 0 and consequently  $e$  should be chosen  $e \gtrsim 0$  and may be neglected when real data is presented to the network.

<sup>6.5</sup>See [FS92] pp. 318–324 and [CG87].

Layer	Neuron output	$C_1$	$C_2$	Excitatory input	Inhibitory input
$w$	$w_j$	1	1	$x_j + au_j$	0
$s$	$s_j$	$e$	1	$w_j$	$\ \mathbf{w}\ $
$u$	$u_j$	$e$	1	$v_j$	$\ \mathbf{v}\ $
$v$	$v_j$	1	1	$f(s_j) + bf(q_j)$	0
$p$	$p_j$	1	1	$u_j + W(j,:) f_2(\mathbf{a}')$	0
$q$	$q_j$	$e$	1	$p_j$	$\ \mathbf{p}\ $
$r$	$r_j$	$e$	1	$u_j + cp_j$	$\ \mathbf{u}\  + c\ \mathbf{p}\ $

**Table 6.1:** The parameters for the general, ART2 differential equation (6.21).

### 6.5.2 The $F_2$ Layer

The  $F_2$  layer of ART2 network is identical to the  $F_2$  layer of ART1 network. The total input into neuron  $k$  is  $a'_k = W'(k,:) f_1(\mathbf{a})$  and the output is:

$$f_2(a'_k) = \begin{cases} d & \text{if } a'_k = \max_\ell a'_\ell \\ 0 & \text{otherwise} \end{cases} \quad (6.22)$$

where  $d = \text{const.}$ ,  $d \in (0, 1)$ . ♦  $d$

Then the output of  $p$  sublayer becomes:

$$p_j = \begin{cases} u_j & \text{if } F_2 \text{ is inactive} \\ u_j + dw_{jk} & \text{for } k \text{ neuron winner on } F_2 \end{cases} \quad (6.23)$$

### 6.5.3 The Reset Layer

The differential equation defining the reset layer running is of the form given by (6.21) with the parameters defined in table 6.1:

$$\frac{dr_j}{dt} = -er_j + u_j + cp_j - r_j\|\mathbf{u}\| - cr_j\|\mathbf{p}\|$$

with the inhibitory input given by the 2 gain-control neurons.

The equilibrium value is:

$$r_j = \frac{u_j + cp_j}{e + \|\mathbf{u}\| + c\|\mathbf{p}\|} \simeq \frac{u_j + cp_j}{\|\mathbf{u}\| + c\|\mathbf{p}\|} \Leftrightarrow \mathbf{r} = \frac{\mathbf{u} + c\mathbf{p}}{\|\mathbf{u}\| + c\|\mathbf{p}\|} \quad (6.24)$$

(see also the remarks regarding the value of  $e$ ).

By definition — considering  $\rho$  the vigilance parameter — the reset occurs when:

♦  $\rho$

$$\|\mathbf{r}\| < \rho \quad (6.25)$$

The reset should not activate before an output from  $F_2$  layer have arrived — this condition is used in the ART2 algorithm — and indeed from (6.20) equations, if  $f_2(\mathbf{a}') = \hat{\mathbf{0}}$  then  $\mathbf{p} = \mathbf{u}$  and then (6.24) gives  $\|\mathbf{r}\| = 1$ .

### 6.5.4 Learning and Initialization

The differential equations governing the weights adaptation are defined as:

$$\frac{dw_{jk}}{dt} = f_2(a'_k)(p_j - w_{jk}) \quad \text{and} \quad \frac{dw'_{kj}}{dt} = f_2(a'_k)(p_j - w'_{kj})$$

and, considering (6.22) and (6.23):

$$\frac{dw_{jk}}{dt} = \begin{cases} d(u_j + dw_{jk} - w_{jk}) & \text{for } k \text{ winner on } F_2 \\ 0 & \text{otherwise} \end{cases}$$

$$\frac{dw'_{kj}}{dt} = \begin{cases} d(u_j + dw_{jk} - w'_{kj}) & \text{for } k \text{ winner on } F_2 \\ 0 & \text{otherwise} \end{cases}$$

#### **Fast Learning**

The weights related to winning  $F_2$  neuron are updated, all others remain unchanged. The equilibrium values are obtained from  $\frac{d(\cdot)}{dt} = 0$  condition. Assuming that  $k$  is the winner:

$$u_j + dw_{jk} - w_{jk} = 0 \quad \text{and} \quad u_j + dw_{kj} - w'_{kj} = 0$$

and then

$$w_{jk} = w'_{kj} = \frac{u_j}{1-d} \Rightarrow \begin{cases} W(:, k) = \frac{\mathbf{u}}{1-d} \\ W'(k, :) = \frac{\mathbf{u}^T}{1-d} = W(:, k)^T \end{cases} \quad (6.26)$$

(so this is why the condition  $0 < d < 1$  is necessary). Eventually the  $W'$  weight matrix becomes the transposed of  $W$  matrix — when all the  $F_2$  neurons have been used to learn new data.

#### **New pattern learning**

The reset unit should not activate when a learning process takes place.

Using the fact that  $u_j = \frac{v_j}{\|\mathbf{v}\|}$  ( $e \simeq 0$ ) and then  $\|\mathbf{u}\| = 1$  and also  $\|\mathbf{r}\| = \sqrt{\mathbf{r}^T \mathbf{r}}$ , from (6.24):

$$\|\mathbf{r}\| = \frac{\sqrt{1 + 2c\|\mathbf{p}\| \cos(\widehat{\mathbf{u}, \mathbf{p}}) + c^2\|\mathbf{p}\|^2}}{1 + c\|\mathbf{p}\|} \quad (6.27)$$

where  $\widehat{\mathbf{u}, \mathbf{p}}$  is the angle between  $\mathbf{u}$  and  $\mathbf{p}$  vectors and if  $\mathbf{p} \parallel \mathbf{u}$  then a reset does not occur because  $\rho < 1$  and the reset condition (6.25) is not met ( $\|\mathbf{r}\| = 1$ ).

If the  $W$  weights are initialized to  $\tilde{\mathbf{0}}$  then the output of  $F_2$ , at the beginning of the learning process, is zero and  $\mathbf{p} = \mathbf{u}$  (see (6.23)) such that the reset does not occur.

During the learning process the weight vector  $W(:, k)$ , associated with connection from  $F_2$  winner to  $F_1$  ( $p$  layer), becomes parallel to  $\mathbf{u}$  (see (6.26)) and then, from (6.23),  $\mathbf{p}$  moves

(during the learning process) towards becoming parallel with  $\mathbf{u}$  and again a reset does not occur.

**Conclusion:** The  $W$  weights have to be initialized to  $\tilde{\mathbf{0}}$ .

### Initialization of $W'$ weights

Let assume that a  $k$  neuron from  $F_2$  have learned a input vector and, after some time, is presented again to the network. The same  $k$  neuron should win, and not one of the uncommitted (yet)  $F_2$  neurons. This means that the output of the  $k$  neuron, i.e.  $a'_k = W'(k, :) \mathbf{p}$  should be bigger than an  $a'_\ell = W'(\ell, :) \mathbf{p}$  for all  $\ell$  unused  $F_2$  neurons:

$$W'(k, :) \mathbf{p} > W(\ell, :) \mathbf{p} \Rightarrow \|W'(k, :)\| \|\mathbf{p}\| > \|W'(\ell, :)\| \|\mathbf{p}\| \cos(\widehat{W'(\ell, :)} \mathbf{p})$$

because  $\|\mathbf{p}\| \|\mathbf{u}\| \|\widehat{W'(\ell, :)}\|$ , see (6.23) and (6.26) (the learning process — weight adaptation — has been done already previously for  $k$  neuron).

The worst possible case is when  $\|\widehat{W'(\ell, :)}\| \|\mathbf{p}\|$  such that  $\cos(\widehat{W'(\ell, :)} \mathbf{p}) = 1$ . To ensure that no other neuron but  $k$  wins, the condition:

$$\|W'(\ell, :)\| < \|W'(k, :)\| = \frac{1}{1-d} = \left\| \frac{\widehat{\mathbf{1}}^T}{\sqrt{K}(1-d)} \right\| \quad (6.28)$$

have to be imposed for unused  $\ell$  neurons (for a committed neuron  $W'(k, :) = \frac{\mathbf{u}^T}{1-d}$  and  $\|\mathbf{u}\| = 1$  as  $u_j \simeq \frac{v_j}{\|\mathbf{v}\|}$ ,  $e \simeq 0$ ).

To maximize the unused neurons input  $a'_\ell$  such that the network will be more sensitive to new patterns the weights of (unused) neurons have to be uniformly initialized with maximum values allowed by the condition (6.28), i.e.  $w'_{\ell j} \lesssim \frac{1}{\sqrt{K}(1-d)}$ .

**Conclusion:** The  $W'$  weights have to be initialized with:  $w'_{kj} \lesssim \frac{1}{\sqrt{K}(1-d)}$ .

### Constants Restraints

As  $\mathbf{p} = \mathbf{u} + dW(:, k)$  and  $\|\mathbf{u}\| = 1$  then:

$$\begin{aligned} \mathbf{u}\mathbf{p} &= \|\mathbf{p}\| \cos(\widehat{\mathbf{u}, \mathbf{p}}) = 1 + d\|W(:, k)\| \cos(\widehat{\mathbf{u}, W(:, k)}) \\ \|\mathbf{p}\| &= \sqrt{\mathbf{p}^T \mathbf{p}} = \sqrt{1 + 2d\|W(:, k)\| \cos(\widehat{\mathbf{u}, W(:, k)}) + d^2\|W(:, k)\|^2} \end{aligned}$$

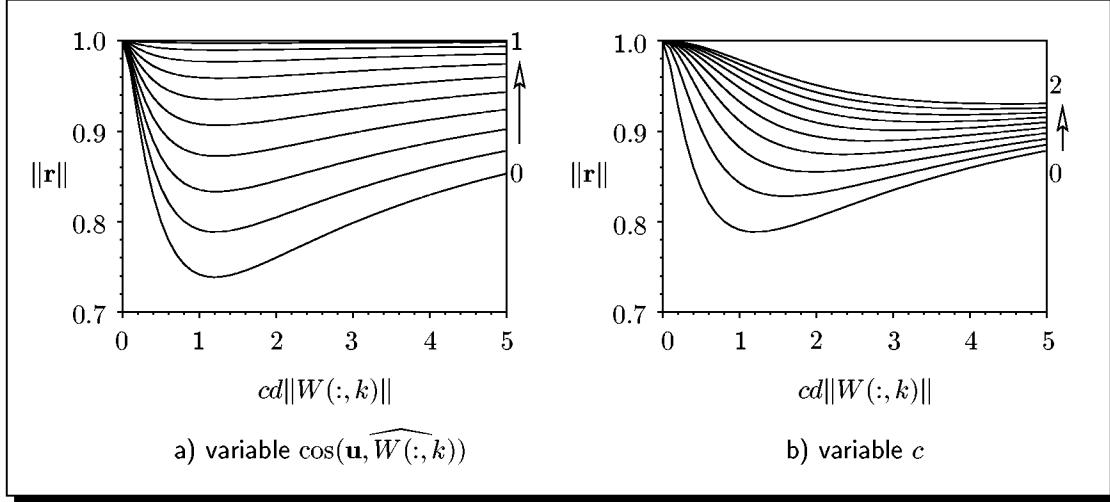
( $k$  being the  $F_2$  winner). Replacing  $\|\mathbf{p}\| \cos(\widehat{\mathbf{u}, \mathbf{p}})$  and  $\|\mathbf{p}\|$  into (6.27) gives:

$$\|\mathbf{r}\| = \frac{\sqrt{(1+c)^2 + 2(1+c)cd\|W(:, k)\| \cos(\widehat{\mathbf{u}, W(:, k)}) + c^2d^2\|W(:, k)\|^2}}{1 + \sqrt{c^2 + 2c \cdot cd\|W(:, k)\| + c^2d^2\|W(:, k)\|^2}}$$

Figure 6.4 on the next page shows the dependency of  $\|\mathbf{r}\|$  as function of  $cd\|\mathbf{w}_{(1)k}\|$ ,  $\cos(\widehat{\mathbf{u}, \mathbf{w}_{(1)k}})$  and  $c$  — note that  $\|\mathbf{r}\|$  decreases for  $cd\|\mathbf{w}_{(1)k}\| < 1$ .

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,



**Figure 6.4:**  $\|r\|$  as function of  $cd\|W(:, k)\|$ . Figure (a) shows dependency for various angles  $\mathbf{u}, \widehat{W(:, k)}$ , from  $\pi/2$  to  $0$  in  $\pi/20$  steps, and  $c = 0.1$  constant. Figure (b) shows dependency for various  $c$  values, from  $0.1$  to  $1.9$  in  $0.2$  steps, and  $\mathbf{u}, \widehat{W(:, k)} = \pi/2 - \pi/20$  constant.

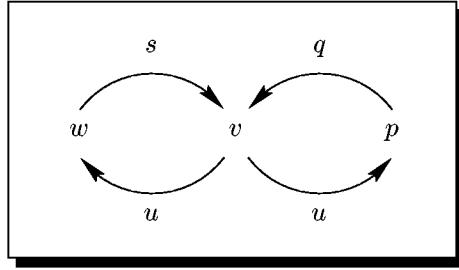
the network should be able to discern better between different classes of input. This means that, while the network learns a new input class and  $W(:, k)$  increases from initial value  $\widehat{\mathbf{0}}$  to the final value when  $\|W(:, k)\| = \frac{1}{1-d}$ , the  $\|r\|$  value (defining the network sensitivity) have to decrease (such that reset condition (6.25) became easier to met). In order to achieve this the following condition have to be imposed:

$$\frac{cd}{1-d} \leq 1$$

(at the end of the learning  $W(:, k) = \frac{1}{1-d}$ ).

Or, in other words, when there is a perfect fit  $\|r\|$  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  $\|r\|$  will first decrease before increasing back to 1. When decreasing it may happen that  $\|r\| < \rho$  and a rest 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  $\|r\|$  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 into  $F_1$  layer may be explained now:
  - $s, u$  and  $q$  just normalize  $w, v$  and  $p$  outputs before sending it further.



**Figure 6.5:** The  $F_1$  dynamics: data communication between sublayers.

- There are connection between  $p \rightarrow v$  (via  $q$ ) and  $w \rightarrow v$  (via  $s$ ) and also back  $v \rightarrow w, p$  (via  $u$ ). See figure 6.5. Obviously  $v$  layer acts as a mediator between input  $x$  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  $W(:, k)$  ( $\|\mathbf{r}\|$  drops). If it moves too far then a reset occurs ( $\|\mathbf{r}\|$  becomes smaller than  $\rho$ ) and the process starts over with another  $F_2$  neuron and a new  $W(:, k')$ . Note that  $\mathbf{u}$  represents eventually a normalized combination (filtered through  $f$ ) of  $x$  and  $p$  (see (6.20)).

## 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:  $\|\mathbf{x}\| = \sqrt{\mathbf{x}^T \mathbf{x}}$ .

1. Select network learning constants such that:

$$a, b > 0 \quad \rho \in (0, 1)$$

$$d \in (0, 1) \quad c \in (0, 1)$$

$$\frac{cd}{1-d} \leq 1$$

and the size of  $F_2$  (similar to ART1 algorithm).

2. Choose a contrast enhancement function, e.g. (6.19).
3. Initialize the weights:

$$W = \tilde{\mathbf{0}} \quad w'_{kj} \lesssim \frac{1}{\sqrt{K}(1-d)}$$

$W'$  to be initialized with random values such that the above condition is met.

### Network running and learning

1. Pickup an input vector  $x$ .

2. First initialize:

$$\mathbf{u} = \hat{\mathbf{0}} \quad \text{and} \quad \mathbf{q} = \hat{\mathbf{0}}$$

and then iterate the following steps till the output values of  $F_1$  sublayers stabilize:

$$\begin{aligned} \mathbf{w} &= \mathbf{x} + a\mathbf{u} & \rightarrow \mathbf{s} &= \frac{\mathbf{w}}{\|\mathbf{w}\|} \rightarrow \\ \rightarrow \mathbf{v} &= f(\mathbf{s}) + bf(\mathbf{q}) & \rightarrow \mathbf{u} &= \frac{\mathbf{v}}{\|\mathbf{v}\|} \rightarrow \\ \rightarrow \mathbf{p} &= \begin{cases} \mathbf{u} & \text{at first iteration} \\ \mathbf{u} + dW(:, k) & \text{on next iterations} \end{cases} & \rightarrow \mathbf{q} &= \frac{\mathbf{p}}{\|\mathbf{p}\|} \end{aligned}$$

( $F_2$  is inactive at first iteration). Note that usually two iterations are enough.

3. Calculate the output of the  $r$  layer:

$$\mathbf{r} = \frac{\mathbf{u} + c\mathbf{p}}{\|\mathbf{u}\| + \|\mathbf{p}\|}$$

If there is a reset, i.e.  $\|\mathbf{r}\| > \rho$  then the  $F_2$  winner (there should be no reset at first pass as  $\|\mathbf{r}\| = 1 > \rho$ ) is made inactive for the current input vector and go back to step number 2. If there is no reset (step 4 is always executed at least once) — a winner was found on  $F_2$  — then the resonance was found and jump to step 6.

4. Find the winner on  $F_2$ . First calculate total inputs  $\mathbf{a}' = W'\mathbf{p}$  then find the winner  $k$  for which  $a'_k = \max_{\ell} a'_{\ell}$  and finally

$$f_2(\mathbf{a}') = \hat{\mathbf{0}} \quad \text{and afterwards} \quad f_2(a'_k) = d$$

as  $F_2$  is a contrast enhancement layer (see (6.22)).

5. Go back to step 2 (find the new output values for  $F_1$  layers).

6. Update the weights (if learning is allowed):

$$W(:, k) = W'(k, :)^T = \frac{\mathbf{u}}{1 - d}$$

7. 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.

# **Basic Principles**

---



## CHAPTER 7

# Pattern Recognition

## ► 7.1 Patterns: The Statistical Approach

### 7.1.1 Patterns and Classification

In most general way an intelligent behavior (living organism, artificial intelligence machines) is represented by the characteristic of being able to recognize/classify patterns — taken in its broadest definition. A pattern may represent a class of objects, a sequence of movements or even a mixture of feelings. The way the intelligent system reacts to the recognition of a pattern may also be considered a pattern.

A quantized pattern is represented by a vector  $\mathbf{x}$ . Let  $\mathcal{C}_k$ ,  $k = \overline{1, K}$  be the classes with respect to which the pattern  $\mathbf{x}$  have to be classified.

❖  $\mathcal{C}_k$

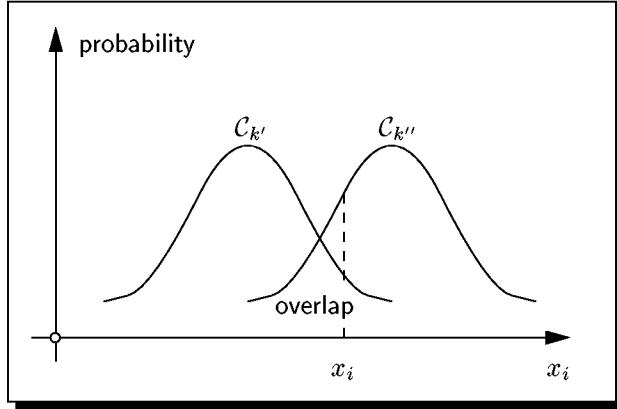


#### Remarks:

- ➔ Many classes of patterns overlap and, in many cases, it may be quite difficult to classify a pattern to a certain class. For this reason a statistical approach is taken, i.e. a class membership probability is attached to patterns.
- ➔ One of the things which may be hard sometimes is to quantize the pattern into a set of numbers such that a processing can be done on them. An image may be quantized into a set of pixels, a sound may be quantized into a set of frequencies associated width pitch, volume and time duration, e.t.c.

The pattern  $\mathbf{x}$  have associated a certain probability to being of class  $\mathcal{C}_k$  — different for each class. This probability is a function of variables  $\{x_i\}_{i=\overline{1, N}}$ . The main problem is to find/build these functions as to be able to give reliable results on previously unseen patterns

7.1.1 See [Bis95] pp. 1–6 and [Rip96] pp. 5, 24.



**Figure 7.1:** Overlapping probability: Considering only the  $x_i$  component it's more probable that the  $\mathbf{x}$  pattern is of class  $C_{k''}$ , however it is possible for  $\mathbf{x}$  to be of class  $C_{k'}$ .

(generalization), i.e. to build a statistical model. The probabilities may overlap — see figure 7.1.

In ANN field, in most cases the probabilities are given by a vector  $\mathbf{y}$  representing the network output and it's a function of its input pattern  $\mathbf{x}$  and some parameters named weights collected together in a matrix  $W$  (or several matrices):

❖  $W$

$$\mathbf{y} = \mathbf{y}(\mathbf{x}, W)$$

❖  $X, Y$   
learning

Usually the mapping from the pattern space  $X$  to the output space  $Y$  is non-linear. ANN offers a very general framework for finding out the mapping  $\mathbf{y} : X \rightarrow Y$  (and are particularly efficient on building nonlinear models). The process of finding the adequate  $W$  weights is called *learning*.

training set,  
generalization

The probabilities and the classes are usually determined from a *learning data set* already classified by a supervisor. The ANN learns to classify from the data set — this kind of learning is called *supervised learning*. At the end of the learning process the network should be able to correctly classify an previously *unseen* pattern, i.e. to *generalize*. The data set is called a *sample* or a *training set* — because the ANN trains/learns using it — and the supervisor is usually a human, however there are neural networks with *unsupervised learning* capabilities. There is also another type called *reinforced learning* where there is no desired output present into the data set but there is a positive or negative feedback depending on output (desired/undesired).

classification,  
regression

If the output variables are part of a discrete, finite set then the process of neural network computing will be called *classification*; for continuous output variables it will be called *regression*<sup>1</sup>

outliers

### Remarks:

- ➔ It is also possible to have a (more or less) special class containing all patterns  $\mathbf{x}$  which were classified (with some confidence) as *not* belonging to any other class.

<sup>1</sup>Regression refers to functions defined as an average over a random quantity.

These patterns are called *outliers*. The outliers usually appear due to insufficient data.

- ➔ In some cases there is a “cost” associated to a (mis)classification. If the cost is too high and the probability of misclassification is also (relatively) high then the classifier may decline to classify the input pattern. These patterns are called *rejects* or *doubts*.
- ➔ Two more “classes” with special meaning may be considered:  $\mathcal{O}$  for outliers and  $\mathcal{D}$  for rejected patterns (doubts).

❖  $\mathcal{O}, \mathcal{D}$

### 7.1.2 Feature Extraction

Usually the straight usage of a pattern (like taking all the pixels from a image and transforming it into a vector) may end up into a very large pattern vector. This may pose some problems to the neural networks in the process of classification because the training set is limited.

Let assume that the pattern vector is unidimensional  $\mathbf{x} = (x_1)$  and there are only 2 classes such that if  $x_1$  is less than some threshold value  $\tilde{x}_1$  then is of class  $\mathcal{C}_1$  otherwise is of class  $\mathcal{C}_2$ . See figure 7.2-a.

Let now add a new feature/dimension to the pattern vector:  $x_2$  such that it becomes bidimensional  $\mathbf{x} = (x_1, x_2)$ . There are 2 cases: either  $x_2$  is relevant to the classification or is not.

If  $x_2$  is not relevant to the classification (does not depend on its value) then it shouldn’t have been added; it just increases the useless/useful data ratio, i.e. it just increase the noise (useless data *is* noise and each  $x_i$  component may bear some noise embedded in its value). Adding more of irrelevant components may increase the noise to a level where it will exceed the useful information.

If is relevant, then it must have a threshold value  $\tilde{x}_2$  such that if  $x_2$  is lower then it is classified in one class, let  $\mathcal{C}_1$  be that one (the number is irrelevant for the justification of the argument, classes may be just renumbered in a convenient way) otherwise is of  $\mathcal{C}_2$ . Now, instead of just 2 cases to be considered ( $x_1$  less or greater than  $\tilde{x}_1$ ) there are four cases (for  $x_1$  and  $x_2$ ). See figure 7.2-b. The number of patterns into the training set being constant, *the number of training patterns per each case have halved* (assuming a large number of training pattern vectors spread evenly). A further addition of a new feature  $x_3$  increases the number of cases to 8. See figure 7.2-c.

In general the number of cases to be considered increases exponentially, i.e.  $K^N$ . The training set spreads thinner into the pattern space. The performance of the neural network with respect to the dimension of the pattern space have a peak and increasing the dimension further may decreases it. See figure 7.2-d. The phenomena of performance decreasing as dimensionality of pattern space increases is known as *the course of dimensionality*.

course of dimensionality

#### Remarks:

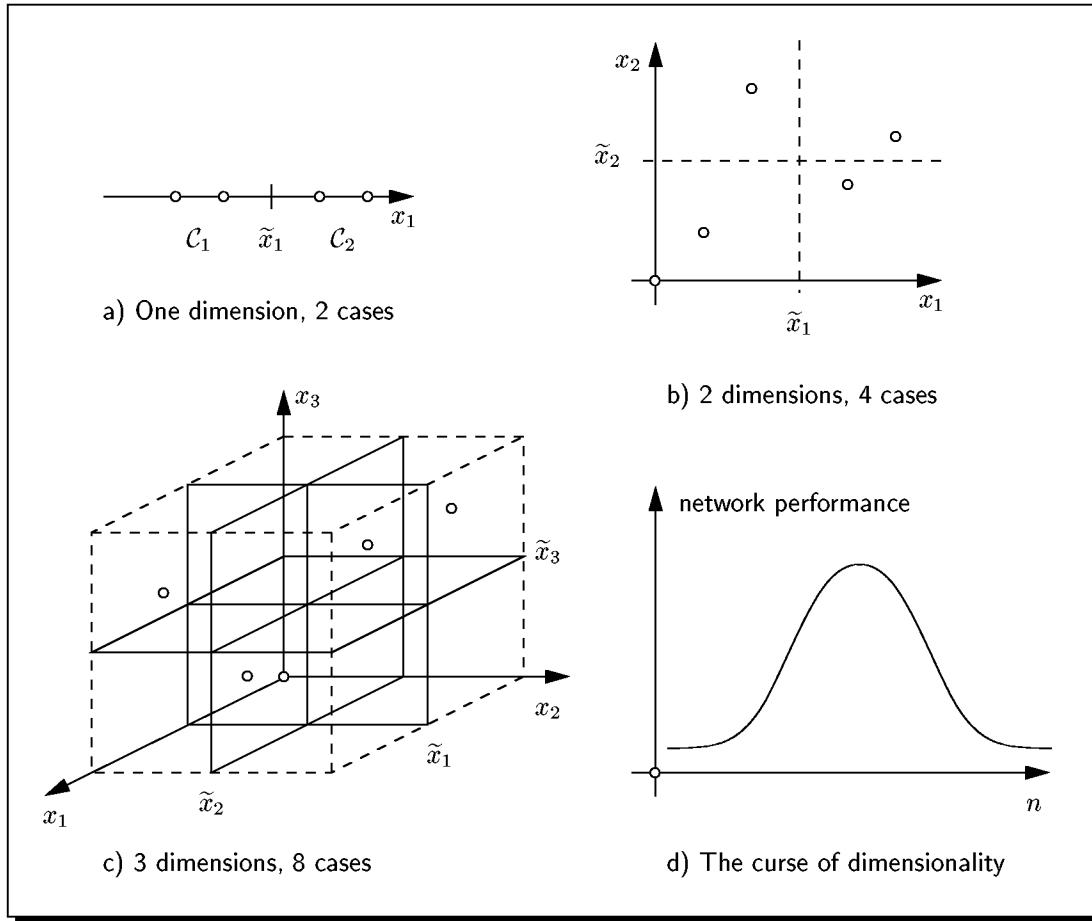
- ➔ The performance of a network can be measured as the percent of correct outputs (correct classification, e.t.c.) with regard to the total number of inputs — after the learning process have finished.

network performance

The process of extracting the useful information and translating the pattern into a vector is

feature extraction

<sup>7.1.2</sup>See [Bis95] pp. 6–9.



**Figure 7.2:** *The curse of dimensionality: The increase of pattern vector dimension  $n$  may cause a worsening in neural network performance. Patterns are represented as dots in the pattern space. The same number of training patterns have to be spread “thinner” if there are more dimensions.*

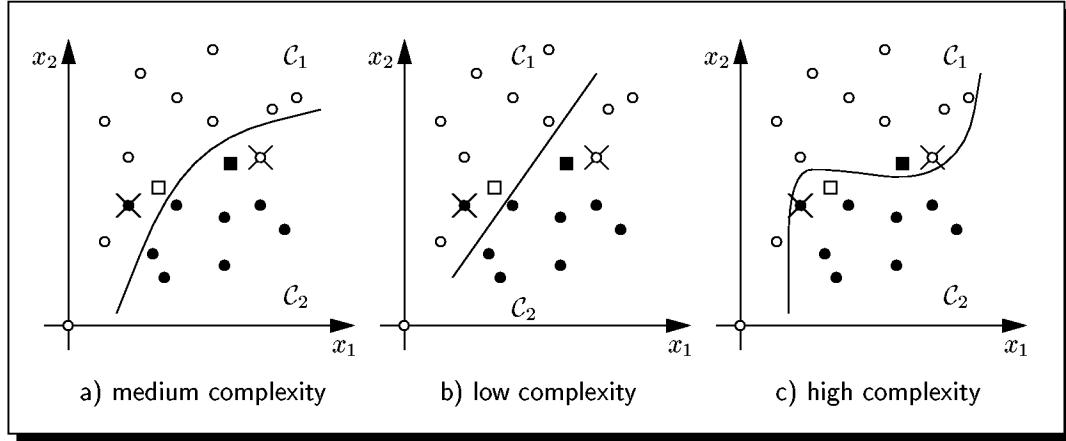
called *feature extraction*. This process may be manually, automatic or even done by another neural network and takes place before entering the actual neural network.

### 7.1.3 Model Complexity

- ❖  $\mathbf{x}_p, \mathbf{t}_p$   
For a supervised learning, the training set consists of pairs  $\{\mathbf{x}_p, \mathbf{t}_p\}_{p=1, \dots, P}$ , where  $\mathbf{t}_p$  is the desired network target output vector given input  $\mathbf{x}_p$ .
- ❖  $E$   
The  $W$  weights are to be found by trying to minimize an error function  $E = E(\mathbf{x}, \mathbf{t}, W)$ . The most widely used error function is the sum-of-squares defined as:

$$E = \frac{1}{2} \sum_{p=1}^P [\mathbf{y}(\mathbf{x}_p, W) - \mathbf{t}_p]^2$$

<sup>7.1.3</sup>See [Bis95] pp. 9-15.



**Figure 7.3:** Model complexity. That training set patterns are marked with circles, new patterns are marked with squares, exceptions are marked with  $\times$ .

another one being root-mean-square  $E_{\text{RMS}}$ :

$$E_{\text{RMS}} = \sqrt{\frac{1}{P} \sum_{p=1}^P [\mathbf{y}(\mathbf{x}_p, W) - \mathbf{t}_p]^2}$$

❖  $E_{\text{RMS}}$

In solving this problem an arbitrary complex model may be build. However there is a trade-off between exception handling and generalization.



### Remarks:

- Exceptions are patterns which are more likely to be member of one class but in fact are from another. Misclassification usually happens due to overlapping of probability (see also figure 7.1) noise in data or missing data.
- There may be also a fuzziness between different classes, i.e. they may not be well defined.

A reasonably complex model will be able to handle (recognize/classify) new patterns and consequently to generalize. See figure 7.3-a. A too simple model will have a low performance — many patterns misclassified. See figure 7.3-b. A too complex model may well handle the exceptions *present in the training set* but may have poor generalization capabilities. See figure 7.3-c.

One widely used way to control the complexity of the model is to add a *regularization term*  $\Omega$  to the error function:

regularization  
❖  $\Omega, \nu$

$$\tilde{E} = E + \nu \Omega$$

which is high for complex models and low for simple models. The  $\nu$  parameter controls the weight by which  $\Omega$  influences  $E$ .

decision  
boundaries

Bayes rule

### 7.1.4 Classification: Making Decisions and Minimizing Risk

A neural network maps the pattern space  $X$  to the classes of patterns. The pattern space is divided into a  $K$  number of areas  $X_k$  (which may be of any possible form) such that if the pattern vector  $\mathbf{x} \in X_k$  it is classified as being of class  $k$ . These areas are named *decision regions* and the boundaries between them are named *decision boundaries*.

The problem consists in finding the decision boundaries such that the errors of misclassification are minimized or the correctness of the classification is maximized.

**Theorem 7.1.1. Bayes rule.** *Being given a pattern vector  $\mathbf{x}$  to be classified into one of the classes  $\{\mathcal{C}_k\}_{k=1,K}$  — the probability of misclassification is minimized if it is classified into class  $\mathcal{C}_k$  for which the posterior probability is maximal:*

$$P(\mathcal{C}_k|\mathbf{x}) = \max_{\ell=1,K} P(\mathcal{C}_\ell|\mathbf{x})$$

*Proof.* The decision boundaries are found by maximizing the correctness of the classification. Let consider one finite decision region  $X_1 \subset X$  such that all pattern vectors belonging to it will be classified as  $\mathcal{C}_1$ . The probability of making a correct classification if  $\mathbf{x} \in X_1$  is the joint-probability associated with that class and decision region  $P(\mathcal{C}_1, X_1)$ . Considering two decision region and two classes all patterns with their pattern vectors in  $X_1$  and belonging to class  $\mathcal{C}_1$  will be classified correctly and the same happens for  $X_2$  and  $\mathcal{C}_2$  — such that the probability of making a correct classification is the sum of the two joint-probabilities, i.e.  $P(\mathcal{C}_1, X_1) + P(\mathcal{C}_2, X_2)$ . In general, for  $K$  classes and respectively decision regions:

$$P_{\text{correct}} = \sum_{k=1}^K P(\mathcal{C}_k, X_k)$$

The joint probability may be written as the product between class-conditional and prior probability<sup>2</sup>:

$$P(\mathcal{C}_k, X_\ell) = P(X_\ell|\mathcal{C}_k) P(\mathcal{C}_k)$$

also as  $P(X_\ell|\mathcal{C}_k) = \int_{X_\ell} p(\mathbf{x}|\mathcal{C}_k) d\mathbf{x}$  then:

$$P_{\text{correct}} = \sum_{k=1}^K P(X_k|\mathcal{C}_k) P(\mathcal{C}_k) = \sum_{k=1}^K \int_{X_k} p(\mathbf{x}|\mathcal{C}_k) P(\mathcal{C}_k) d\mathbf{x}$$

Each  $X_k$  should be chosen such that, inside it, the integrand  $p(\mathbf{x}|\mathcal{C}_k) P(\mathcal{C}_k)$  is greater than any other integrand  $p(\mathbf{x}|\mathcal{C}_\ell) P(\mathcal{C}_\ell)$ , for  $\ell \neq k$ :

$$\begin{aligned} p(\mathbf{x}|\mathcal{C}_k) P(\mathcal{C}_k) &> p(\mathbf{x}|\mathcal{C}_\ell) P(\mathcal{C}_\ell) \quad , \quad \forall \ell \in \{1, \dots, K | \ell \neq k\} \\ \Leftrightarrow \frac{p(\mathbf{x}|\mathcal{C}_k) P(\mathcal{C}_k)}{p(\mathbf{x})} &> \frac{p(\mathbf{x}|\mathcal{C}_\ell) P(\mathcal{C}_\ell)}{p(\mathbf{x})} \quad (\text{because } p(\mathbf{x}) > 0) \\ \Leftrightarrow P(\mathcal{C}_k|\mathbf{x}) &> P(\mathcal{C}_\ell|\mathbf{x}) \quad (\text{from Bayes theorem}) \quad \square \end{aligned}$$

#### Remarks:

- The decision boundaries are placed at the points where the highest posterior probability  $P(\mathcal{C}_k|\mathbf{x})$  becomes smaller than another  $P(\mathcal{C}_\ell|\mathbf{x})$ . See figure 7.4 on the facing page.

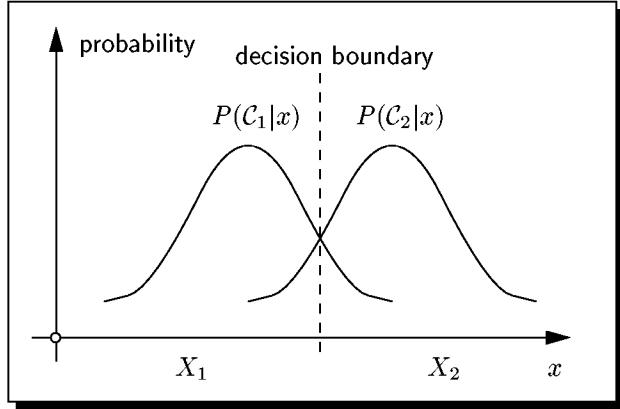
❖  $\mathcal{M}, P_{mc}$

**Definition 7.1.1.** *Given a mapping (classification procedure)  $\mathcal{M} : X \rightarrow \{1, \dots, K, \mathcal{D}\}$  the probability of misclassification a vector  $\mathbf{x}$  of class  $\mathcal{C}_k$  is*

$$P_{mc}(k) = P(\mathcal{M}(\mathbf{x}) \neq \mathcal{C}_k, \mathcal{M}(\mathbf{x}) \in \{\mathcal{C}_1, \dots, \mathcal{C}_K\} | \mathbf{x} \in \mathcal{C}_k) = \sum_{\substack{\ell=1 \\ \ell \neq k}}^K P(X_\ell|\mathcal{C}_k)$$

---

<sup>2</sup>See the statistical appendix.



**Figure 7.4:** Decision boundary for a unidimensional pattern space and two classes.

The doubt probability  $P_d(i)$  is defined similarly:

❖  $P_d$

$$P_d(k) = P(\mathcal{M}(\mathbf{x}) = \mathcal{D} | \mathbf{x} \in \mathcal{C}_k) = \int_{X_{\mathcal{D}}} p(\mathbf{x} | \mathcal{C}_k) d\mathbf{x}$$

The total doubt probability is the probability for a pattern from any class to be unclassified (i.e. classified as doubt) ❖  $P(\mathcal{D} | \mathbf{x})$

$$P(\mathcal{D} | \mathbf{x}) = \sum_{k=1}^K P_d(k)$$

More general, the decision boundaries may be defined with the help of a set of *discriminant functions*  $\{y_k(\mathbf{x})\}_{k=\overline{1,K}}$  such that a pattern vector  $\mathbf{x}$  is assigned to class  $\mathcal{C}_k$  if

discriminant functions

$$y_k(\mathbf{x}) = \max_{\ell=\overline{1,K}} y_{\ell}(\mathbf{x})$$

and in particular case  $y_k(\mathbf{x}) = P(\mathcal{C}_k | \mathbf{x})$  as in theorem 7.1.1.

In particular cases (in practice, e.g. in medical field) there may be necessary to increase the penalty of misclassification of a pattern from one class to another. Then a *loss matrix* defining penalties for misclassification is used: let  $L_{k\ell}$  be the penalty associated to misclassification of a pattern belonging to class  $\mathcal{C}_k$  as being of class  $\mathcal{C}_{\ell}$ .

loss matrix

### Remarks:

- ➔ Penalty may be associated with *risk*: high penalty means high risk in case of misclassification. risk

The diagonal elements of the loss matrix should be  $L_{kk} = 0$  because, in this case, there is no misclassification so no penalty.

The penalty associated with misclassification of a pattern  $\mathbf{x} \in \mathcal{C}_k$  into a particular class  $\mathcal{C}_{\ell}$  is  $L_{k\ell}$  multiplied by the probability of misclassification  $P(X_{\ell} | \mathcal{C}_k)$  (the probability that the

pattern vector is in  $X_\ell$  but is of class  $\mathcal{C}_k$ ).

$$R_{k\ell} = L_{k\ell} P(X_\ell | \mathcal{C}_k) = L_{k\ell} \int_{X_\ell} p(\mathbf{x} | \mathcal{C}_k) d\mathbf{x}$$

 **Remarks:**

- The loss term  $L_{k\ell}$  have the role of increasing the effect of misclassification probability  $P(X_\ell | \mathcal{C}_k)$ .

The total penalty associated with misclassification of a pattern  $\mathbf{x} \in \mathcal{C}_k$  in any other class is the sum, over all classes, of penalties for misclassification of  $\mathbf{x}$  into another class

$$R_k = \sum_{\ell=1}^K R_{k\ell} = \sum_{\ell=1}^K L_{k\ell} P(X_\ell | \mathcal{C}_k) = \sum_{\ell=1}^K L_{k\ell} \int_{X_\ell} p(\mathbf{x} | \mathcal{C}_k) d\mathbf{x} \quad (7.1)$$

❖  $P_{X|C}$  or, considering the matrix  $P_{X|C} = \{P(X_\ell | \mathcal{C}_k)\}_{\ell,k}$  ( $\ell$  row,  $k$  column index) then

$$R_k = L(k, :) P_{X|C}(:, k)$$

( $R_k$  are the diagonal terms of  $L P_{X|C}$  product).

The total penalty for misclassification is the sum of penalties associated with misclassification of a pattern  $\mathbf{x} \in \mathcal{C}_k$  into any other class multiplied by the probability that such penalty may occur, i.e.  $P(\mathcal{C}_k)$ . Defining the vector  $\mathbf{P}_C^T = (P(\mathcal{C}_1) \ \dots \ )$  then:

$$R = \sum_{k=1}^K R_k P(\mathcal{C}_k) = P_C^T [(L \odot P_{X|C}^T) \hat{\mathbf{1}}] = \sum_{\ell=1}^K \int_{X_\ell} \left[ \sum_{k=1}^K L_{k\ell} p(\mathbf{x} | \mathcal{C}_k) P(\mathcal{C}_k) \right] d\mathbf{x} \quad (7.2)$$

*Proof.*  $R$  represents the multiplication between  $\mathbf{P}_C$  and  $\mathbf{R}^T = (R_1 \ \dots \ )$ .  $L \odot P_{X|C}^T$  creates the elements of sum appearing in (7.1) while multiplication by  $\hat{\mathbf{1}}$  sums the  $L \odot P_{X|C}^T$  matrix on rows. □

The penalty is minimized when the  $X_\ell$  areas are chosen such that the integrand in (7.1) is minimum:

$$\mathbf{x} \in X_\ell \Rightarrow \sum_{k=1}^K L_{k\ell} p(\mathbf{x} | \mathcal{C}_k) P(\mathcal{C}_k) = \min_m \sum_{k=1}^K L_{km} p(\mathbf{x} | \mathcal{C}_k) P(\mathcal{C}_k) \quad (7.3)$$

 **Remarks:**

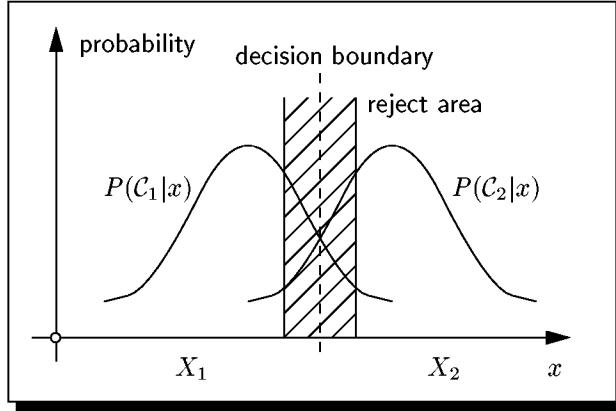
- (7.3) is equivalent with theorem 7.1.1 if the penalty is 1 for any misclassification, i.e.  $L_{k\ell} = 1 - \delta_{k\ell}$  ( $\delta_{ij}$  being the Kronecker symbol).

*Proof.* Indeed, in this case (7.3) becomes:

$$\sum_{\substack{k=1 \\ k \neq \ell}}^K p(\mathbf{x} | \mathcal{C}_k) P(\mathcal{C}_k) < \sum_{\substack{k=1 \\ k \neq m}}^K p(\mathbf{x} | \mathcal{C}_k) P(\mathcal{C}_k) \quad \text{for } \mathbf{x} \in X_\ell, \forall m \neq \ell$$

and by subtracting the identity  $\sum_{k=1}^K p(\mathbf{x} | \mathcal{C}_k) P(\mathcal{C}_k) = \sum_{k=1}^K p(\mathbf{x} | \mathcal{C}_k) P(\mathcal{C}_k)$  from above equation, finally

$$p(\mathbf{x} | \mathcal{C}_\ell) P(\mathcal{C}_\ell) > p(\mathbf{x} | \mathcal{C}_m) P(\mathcal{C}_m); \forall \ell \in \{1, \dots, K\} \ell \neq m\}$$



**Figure 7.5:** Reject area around a decision boundary between two classes, into a unidimensional pattern space.

which is equivalent to

$$P(\mathcal{C}_\ell|\mathbf{x}) > P(\mathcal{C}_m|\mathbf{x})$$

(see the proof for theorem 7.1.1).  $\square$

In general, most of the misclassification occur in the vicinity of decision boundaries where the difference between the top-most posterior probability and the next one is relatively low. If the penalty/risk of misclassification is very high then is better to define a *reject area* around the decision boundary such that all pattern vectors inside it are rejected from the classification process (to be analyzed by a higher instance, e.g. a human or a slower but more accurate ANN), i.e. are classified in class  $\mathcal{D}$ . See figure 7.5.

Considering the doubt class  $\mathcal{D}$  as well, with the associated loss  $L_{k\mathcal{D}}$ , then the risk terms change to:

$$R_k = \sum_{\ell=1}^K R_{k\ell} + L_{k\mathcal{D}} P_d(k)$$

and the total penalty (7.2) also changes accordingly.

To accommodate the reject area and the loss matrix, the Bayes rule 7.1.1 is changed as in the proposition below.

**Proposition 7.1.1. (Bayes rule with reject area and loss matrix).** Let consider a pattern vector  $\mathbf{x}$  to be classified in one of the classes  $\{\mathcal{C}_k\}_{k=1,\overline{K}}$  or  $\mathcal{D}$ . Let consider the loss matrix  $\{L_{k\ell}\}$  and also the loss  $L_{k\mathcal{D}} = d = \text{const.}, \forall k$ , for the doubt class. Then:

reject area

$\diamond L_{k\mathcal{D}}$

$\diamond d$

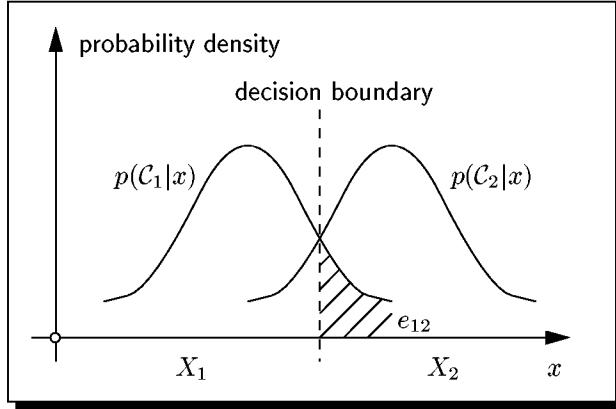
- Neglecting loss, the best classification is obtained when  $\mathbf{x}$  is classified as  $\mathcal{C}_k$  if

$$P(\mathcal{C}_k|\mathbf{x}) = \max_{\ell=1,\overline{K}} P(\mathcal{C}_\ell|\mathbf{x}) > P(\mathcal{D}|\mathbf{x})$$

or  $\mathbf{x}$  is classified as  $\mathcal{D}$  otherwise, i.e.  $P(\mathcal{C}_\ell|\mathbf{x}) < P(\mathcal{D}|\mathbf{x}), \forall \ell = \overline{1, K}$ .

- Considering loss, the best classification is obtained when  $\mathbf{x}$  is classified as  $\mathcal{C}_k$  if

$$R_k = \min_{\ell=1,\overline{K}} R_\ell < R_{\mathcal{D}}$$



**Figure 7.6:** The graphical representation for the elements of the confusion matrix. The hatched area represents element  $e_{12}$ .

♦  $R_D$  or  $x$  is classified as  $\mathcal{D}$  otherwise, i.e.  $R_\ell > R_D$ ,  $\forall \ell = \overline{1, K}$ , where  $R_D$  is the risk associated to a classification in the doubt category.

*Proof.* 1. This is simply an extension to theorem 7.1.1 considering a supplementary class  $\mathcal{D}$  with an associated decision area  $X_D$ .

2. This represents just the rule of classification according to the minimum risk.  $\square$

confusion matrix Another useful tool for estimating the capabilities of a classifier is the *confusion matrix* whose elements are defined as:

$$e_{k\ell} = P(\mathbf{x} \text{ classified as } \mathcal{C}_\ell | \mathbf{x} \in \mathcal{C}_k)$$

As a simple geometrical representation, the element  $e_{k\ell}$  represents the integral of  $p(\mathcal{C}_k|\mathbf{x})$  over the decision area of class  $\mathcal{C}_\ell$ :

$$e_{k\ell} = \int_{X_\ell} p(\mathcal{C}_k|\mathbf{x}) d\mathbf{x}$$

See figure 7.6. Note that  $e_{kk}$  represents the probability of a correct classification for class  $\mathcal{C}_k$ .

## 7.2 Likelihood Function

### 7.2.1 The Discriminant Functions

Instead of the most obvious discriminant function  $y_k(\mathbf{x}) = P(\mathcal{C}_k|\mathbf{x})$  its logarithm may be chosen:

$$y_k(\mathbf{x}) = \ln P(\mathcal{C}_k|\mathbf{x}) = \ln \frac{p(\mathbf{x}|\mathcal{C}_k) P(\mathcal{C}_k)}{p(\mathbf{x})} = \ln p(\mathbf{x}|\mathcal{C}_k) + \ln P(\mathcal{C}_k) + \text{const.}$$

(see the Bayes theorem and theorem 7.1.1). The  $p(\mathbf{x})$  being class independent (normalization factor) is just an additive constant.

**Remarks:**

- Considering each class-conditional probability density  $p(\mathbf{x}|\mathcal{C}_k)$  as independent multidimensional Gaussian distribution<sup>3</sup>

$$p(\mathbf{x}|\mathcal{C}_k) = \frac{1}{(2\pi)^{N/2} \sqrt{|\Sigma_k|}} \exp \left[ -\frac{(\mathbf{x} - \boldsymbol{\mu}_k)^T \Sigma_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)}{2} \right]$$

(each having its  $\boldsymbol{\mu}_k$  and  $\Sigma_k$  parameters) then

$$y_k = -\frac{(\mathbf{x} - \boldsymbol{\mu}_k)^T \Sigma_k^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)}{2} - \frac{1}{2} \ln |\Sigma_k| + \ln P(\mathcal{C}_k) + \text{const.}$$

Let  $\Sigma_k = \Sigma$ ,  $\forall k \in \{1, \dots, K\}$ . Then  $\ln |\Sigma|$  is class independent (constant) and  $\mathbf{x}^T \Sigma^{-1} \boldsymbol{\mu}_k = \boldsymbol{\mu}_k^T \Sigma^{-1} \mathbf{x}$ . Eventually:

$$y_k(\mathbf{x}) = \boldsymbol{\mu}_k^T \Sigma^{-1} \mathbf{x} - \frac{\boldsymbol{\mu}_k^T \Sigma^{-1} \boldsymbol{\mu}_k}{2} + \ln P(\mathcal{C}_k) + \mathbf{x}^T \Sigma^{-1} \mathbf{x} + \text{const.}$$

Because  $\mathbf{x}^T \Sigma^{-1} \mathbf{x}$  is class independent then it may be dropped from the discriminant function  $y_k(\mathbf{x})$  (being an additive factor, equal for all discriminants). Eventually:

$$y_k(\mathbf{x}) = (\boldsymbol{\mu}_k^T \Sigma^{-1}) \mathbf{x} - \frac{\boldsymbol{\mu}_k^T \Sigma^{-1} \boldsymbol{\mu}_k}{2} + \ln P(\mathcal{C}_k)$$

Let consider  $\Xi$  the matrix built using  $\boldsymbol{\mu}_k$  as columns and the matrix  $W$  defined as:

$$W(1 : K, 1 : N) = \Xi^T \Sigma^{-1} \quad \text{and} \quad w_{0k} = -\frac{\boldsymbol{\mu}_k^T \Sigma^{-1} \boldsymbol{\mu}_k}{2} + \ln P(\mathcal{C}_k)$$

and  $\tilde{\mathbf{x}}^T = (1 \ x_1 \ \dots \ x_N)$ ; then the discriminant functions may be written simply as

$$\mathbf{y} = W \tilde{\mathbf{x}}$$

(i.e. the general sought form  $\mathbf{y} = \mathbf{y}(W, \mathbf{x})$ ).

The above equation represents a linear form, such that the decision boundaries are hyper-planes. The equation describing the hyper-plane decision boundary between class  $\mathcal{C}_k$  and  $\mathcal{C}_\ell$  is found by formulating the condition  $y_k(\mathbf{x}) = y_\ell(\mathbf{x})$ . See figure 7.7 on the next page.

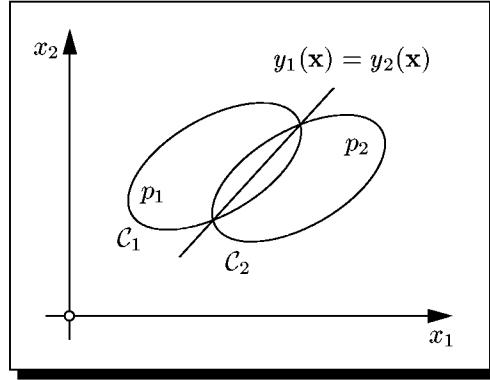
### 7.2.2 Likelihood Function and Maximum Likelihood Procedure

The maximum likelihood method tries to find the best values for the parameters by maximizing a function named likelihood, using the training set. The procedure below is repeated for each class  $\mathcal{C}_k$  in turn.

Let consider a probability density function depending on  $\mathbf{x}$  and a set of parameters  $W$ :  $p = p(\mathbf{x}, W)$ . Let also the training set be  $\{\mathbf{x}_p\}_P = \{\mathbf{x}_1, \dots, \mathbf{x}_P\}$ , all  $\mathbf{x}_p$  being taken from  $\diamond \{\mathbf{x}_p\}_P$

---

<sup>3</sup>See statistical appendix



**Figure 7.7:** The linear discriminant function for a bidimensional probability density and two classes. At the upper-left corner  $p_1 > p_2$ , at the lower-right corner  $p_1 < p_2$ .

the same class.

Considering the vectors from the training set randomly selected (training set statistically significant), the joint probability density for the whole  $\{\mathbf{x}_p\}_P$  is:

$$p(\{\mathbf{x}_p\}_P|W) = \prod_{p=1}^P p(\mathbf{x}_p|W) \equiv \mathcal{L}(W) \quad (7.4)$$

likelihood function  
❖  $\mathcal{L}(W)$  where  $\mathcal{L}(W)$  is named *likelihood function*. The method is to try to find the  $W$  set of parameters for which  $\mathcal{L}(W)$  is maximum.

### Remarks:

► In the case of a Gaussian distribution the  $W$  parameters are defined<sup>4</sup> through  $\mu$  and  $\Sigma$ :  $\mu = \mathcal{E}\{\mathbf{x}\}$  and  $\Sigma = \mathcal{E}\{(\mathbf{x} - \mu)(\mathbf{x} - \mu)^T\}$ . Then:

$$\begin{aligned} \tilde{\mu} &= \frac{1}{P} \sum_{p=1}^P \mathbf{x}_p \xrightarrow[P \rightarrow \infty]{} \mu \\ \tilde{\Sigma} &= \frac{1}{P} \sum_{i=1}^P (\mathbf{x}_p - \tilde{\mu})(\mathbf{x}_p - \tilde{\mu})^T \xrightarrow[P \rightarrow \infty]{} \Sigma \end{aligned}$$

❖  $\tilde{\mu}, \tilde{\Sigma}$  i.e.  $\mu$  — the mean of  $\mathbf{x}$  is replaced with  $\tilde{\mu}$  — the mean of the training set (considered *statistically representative*); and the same happens for  $\Sigma$ .

► Assuming an unidimensional Gaussian distribution then

$$\tilde{\mu} = \frac{1}{P} \sum_{p=1}^P x_p \quad \text{and} \quad \tilde{\sigma^2} = \frac{1}{P} \sum_{p=1}^P (x_p - \tilde{\mu})^2$$

---

<sup>4</sup>See statistical appendix

Considering  $\tilde{\mu}$  as  $\mu$  and the true value for standard deviation  $\sigma$  then the expectation of *assumed* standard deviation, compared to the true one, is:

$$\mathcal{E}\{\tilde{\sigma}^2\} = \frac{P}{P-1} \sigma^2$$

*Proof.*

$$\mathcal{E}\{\tilde{\sigma}^2\} = \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} \frac{1}{P} \sum_{p=1}^P (x_p - \tilde{\mu})^2 \exp\left[-\frac{(x_p - \tilde{\mu})^2}{2\sigma^2}\right] dx_p$$

The change of variable  $y_p = \frac{x_p - \tilde{\mu}}{\sigma}$  is done, then  $x_p - \tilde{\mu} = \frac{P-1}{P} x_p - \frac{1}{P} \sum_{\substack{q=1 \\ q \neq p}}^N x_q$  such that

$$dy_p = \frac{P-1}{\sigma P} dx_p \text{ and}$$

$$\mathcal{E}\{\tilde{\sigma}^2\} = \frac{\sigma^2}{\sqrt{2\pi}(P-1)} \sum_{p=1}^P \int_{-\infty}^{\infty} y_p^2 \exp\left(-\frac{y_p^2}{2}\right) dy_p = \frac{P}{P-1} \sigma^2$$

(integral made by parts, similar to the calculus of  $\mathcal{E}[(x - \mu)^2]$  — see unidimensional Gaussian distribution in statistical appendix).  $\square$

The build probability distribution have a *bias* with tends to 0 for  $P \rightarrow \infty$ .

bias

- ➡ In this case the Gaussian distribution is also suited for *sequential parameter estimation*.

sequential parameter estimation

Assuming that not all patterns from the training set are known at once then new patterns may be added later as they become available, i.e. the  $W$  parameters may be adapted to the new training set

For the Gaussian distribution, adding a new pattern changes  $\tilde{\mu}_P = \frac{1}{P} \sum_{p=1}^P x_p$  to the new value:

$$\tilde{\mu}_{P+1} = \frac{1}{P+1} \sum_{p=1}^{P+1} x_p = \tilde{\mu}_P + \frac{x_{P+1} - \tilde{\mu}_P}{P+1}$$

For multiple classes the likelihood function is defined as:

$$\mathcal{L}(W) = \prod_{p=1}^P p(\mathbf{x}_p | W) = \prod_{k=1}^K \prod_{\mathbf{x}_p \in \mathcal{C}_k} p(\mathbf{x}_p | \mathcal{C}_k, W) P(\mathcal{C}_k | W)$$

Maximizing likelihood function is equivalent to minimizing its negative logarithm, which is the usual way in practice. Considering that in the training set there are  $P_k$  vector patterns for each class  $\mathcal{C}_k$  then:

❖  $P_k$

$$E = -\ln \mathcal{L} = -\sum_{k=1}^K \sum_{p=1}^{P_k} \ln p(\mathbf{x}_{(k)p} | \mathcal{C}_k, W) - \sum_{k=1}^K P_k \ln P(\mathcal{C}_k | W) \quad (7.5)$$

where  $\mathbf{x}_{(k)p} \in \mathcal{C}_k$ .

❖  $\mathbf{x}_{(k)p}$

For a given training set the above expression may be reduced to:

$$E = - \sum_{k=1}^K \sum_{p=1}^{P_k} \ln p(\mathbf{x}_{(k)p} | \mathcal{C}_k, W) + \text{const.} \quad (7.6)$$

*Proof.* The expression is reduced by using the Lagrange multipliers method<sup>5</sup> using as minimization condition the normalization of  $P(\mathcal{C}_k | W)$ :

$$\sum_{k=1}^K P(\mathcal{C}_k | W) = 1 \quad (7.7)$$

which leads to the Lagrange function:

$$L = E + \lambda \left( \sum_{k=1}^K P(\mathcal{C}_k | W) - 1 \right)$$

and then

$$\frac{\partial L}{\partial P(\mathcal{C}_k | W)} = -\frac{P_k}{P(\mathcal{C}_k | W)} + \lambda = 0 \quad k = 1, K$$

and replacing into (7.7) gives

$$\lambda = \sum_{k=1}^K P_k \Rightarrow P(\mathcal{C}_k | W) = \frac{P_k}{\sum_{k=1}^K P_k} = \frac{P_k}{P}$$

As the training set is fixed then  $\{P_k\}$  are constant and then the likelihood (7.5) becomes (7.6).  $\square$

As it can be seen, the formula (7.5) contains the sum both over classes and inside a class. If the data gathering is easy and the classification (by some supervisor) is difficult ("expensive") then the function (7.6) may be replaced by:

$$E = - \sum_{k=1}^K \sum_{p=1}^{P_k} \ln p(\mathbf{x}_{(k)p} | \mathcal{C}_k, W) - \sum_{p=1}^{P'} \ln p(\mathbf{x}'_p | W)$$

The unclassified training set  $\{\mathbf{x}'_p\}$  may still be very useful in finding the appropriate set of  $W$  parameters.

### Remarks:

- ➔ The maximum likelihood method is based on finding the  $W$  parameters for which the likelihood function  $\mathcal{L}(W)$  is maximum, i.e. where its derivative is 0 —  $W$  are the roots of the derivative; The Robbins–Monro algorithm<sup>6</sup> may be used to find them.

The maximum of likelihood function  $\mathcal{L}(W)$  (see equation (7.4)) may be found from the condition:

$$\nabla_W \left[ \prod_{p=1}^P p(\mathbf{x}_p | W) \right] \Big|_{\widetilde{W}} = 0 \quad (7.8)$$

❖  $\nabla_W, N_W$

where  $\nabla_W$  is the vector  $\left( \frac{\partial}{\partial w_1} \dots \frac{\partial}{\partial w_{N_W}} \right)^T$ ,  $N_W$  being the dimension of  $W$

---

<sup>5</sup>See mathematical appendix.

<sup>6</sup>See statistical appendix.

parameter space.

Because  $\ln$  is a monotone function then  $\ln \mathcal{L}$  may be used instead of  $\mathcal{L}$  and the above condition becomes:

$$\frac{1}{P} \nabla_W \left[ \sum_{p=1}^P \ln p(\mathbf{x}_p|W) \right] \Big|_{\widetilde{W}} = 0$$

(the factor  $1/P$  is constant and does not affect the result).

Considering the vectors from the training set randomly selected then

$$\lim_{P \rightarrow \infty} \frac{1}{P} \nabla_W \left[ \sum_{p=1}^P \ln p(\mathbf{x}_p|W) \right] = \mathbb{E}\{\nabla_W \ln p(\mathbf{x}|W)\}$$

and the condition (7.8) becomes:

$$\mathbb{E}\{\nabla_W \ln p(\mathbf{x}|W)\} = 0$$

The roots  $\widetilde{W}$  of this function are found using the Robbins–Monro algorithm.

- It is *not* possible to chose the  $W$  parameters such that the likelihood function  $\mathcal{L}(W) = \prod_{p=1}^P p(\mathbf{x}_p|W)$  (see (7.4)) is maximized, because the  $\mathcal{L}(W)$  may be increased indefinitely by overfitting the training set such that the estimated probability density is reduced to a function similar to  $\delta$  Dirac function, having the value 1 at the training set points and 0 elsewhere.

## 7.3 Statistical Models

It is important to note that the statistical model built as  $p(\mathbf{x}|W)$  generally differ from the true probability density  $p_{\text{true}}(\mathbf{x})$ , which is also independent of  $W$  parameters. The estimated probability density will give the best fit of true  $p(\mathbf{x})$  for some  $W_0$  parameters. These parameters may be found given an *infinite* training set. However, in practice, as the learning set ( $P$ ) is finite then only an estimate  $\widetilde{W}$  of  $W_0$  may be found.

❖  $p_{\text{true}}(\mathbf{x})$ ,  
 $W_0$ ,  $\widetilde{W}$

It is possible to build a function such that it will measure the “distance” between the estimated and the real probability densities. Then the  $W$  parameters have to be found such that this function will be minimum.

Let consider the expected value of the minus logarithm of the likelihood function:

$$\mathbb{E}\{-\ln \mathcal{L}\} = - \lim_{P \rightarrow \infty} \frac{1}{P} \sum_{p=1}^P \ln p(\mathbf{x}_p|W) = - \int_X \ln[p(\mathbf{x}|W)] p_{\text{true}}(\mathbf{x}) d\mathbf{x}$$

which, for  $p(\mathbf{x}|W) = p_{\text{true}}(\mathbf{x})$ , have the value:  $-\int_X p_{\text{true}}(\mathbf{x}) \ln p_{\text{true}}(\mathbf{x}) d\mathbf{x}$ .

The the following function — called *asymmetric divergence*<sup>7</sup> — is defined:

asymmetric  
divergence

<sup>7.3</sup>See [Rip96] pp. 32–34.

<sup>7</sup>also known as Fullback–Leiber distance

$$L = \mathcal{E}\{-\ln \mathcal{L}\} + \int_X p(\mathbf{x}) \ln p(\mathbf{x}) d\mathbf{x} = - \int_X \ln \frac{p(\mathbf{x}|W)}{p_{\text{true}}(\mathbf{x})} p_{\text{true}}(\mathbf{x}) d\mathbf{x} \quad (7.9)$$

**Proposition 7.3.1.** *The asymmetric divergence  $L$  is positive definite, i.e.  $L \geq 0$ , the equality being for  $p(\mathbf{x}|W) = p_{\text{true}}(\mathbf{x})$ .*

*Proof.* Let consider the function  $f(x) = -\ln x + x - 1$ . Its derivative is  $\frac{df}{dx} = -\frac{1}{x} + 1$  and is negative for  $x < 1$  and positive for  $x > 1$ . It follows that the  $f(x)$  function have a minimum for  $x = 1$  where  $f(1) = 0$ . Because  $\lim_{x \rightarrow \infty} f(x) = \infty$  then the function  $f(x)$  is positive definite, i.e.  $f(x) \geq 0, \forall x$ , the equality happening for  $x = 1$ .

Let now consider the function:

$$f\left(\frac{p(\mathbf{x}|W)}{p_{\text{true}}(\mathbf{x})}\right) = -\ln \frac{p(\mathbf{x}|W)}{p_{\text{true}}(\mathbf{x})} + \frac{p(\mathbf{x}|W)}{p_{\text{true}}(\mathbf{x})} - 1 \geq 0$$

and because it is positive definite then its expectation is positive:

$$\begin{aligned} \mathcal{E}\left\{f\left(\frac{p(\mathbf{x}|W)}{p_{\text{true}}(\mathbf{x})}\right)\right\} &= - \int_X \ln \frac{p(\mathbf{x}|W)}{p_{\text{true}}(\mathbf{x})} p_{\text{true}}(\mathbf{x}) d\mathbf{x} + \int_X \frac{p(\mathbf{x}|W)}{p_{\text{true}}(\mathbf{x})} p_{\text{true}}(\mathbf{x}) d\mathbf{x} - 1 \\ &= - \int_X \ln \frac{p(\mathbf{x}|W)}{p_{\text{true}}(\mathbf{x})} p_{\text{true}}(\mathbf{x}) d\mathbf{x} \geq 0 \end{aligned}$$

( $p(\mathbf{x}|W)$  is normalized such that  $\int_X p(\mathbf{x}|W) d\mathbf{x} = 1$ ) and then  $L \geq 0$ , being 0 when the probability distributions are equal.  $\square$

As previously discussed, usually the model chosen for probability density  $p(\mathbf{x}|W)$  is *not* even from the same class of models as the “true” probability density  $p_{\text{true}}(\mathbf{x})$ , i.e. they may have totally different functional expressions. However there is a set of parameters  $W_0$  for which the asymmetric divergence (7.9) is minimized:

$$\min_W \left\{ \int_X [\ln p_{\text{true}}(\mathbf{x}) - \ln p(\mathbf{x}|W)] p(\mathbf{x}) d\mathbf{x} \right\} = \int_X \ln \frac{p_{\text{true}}(\mathbf{x})}{p(\mathbf{x}|W_0)} p_{\text{true}}(\mathbf{x}) d\mathbf{x} \quad (7.10)$$

The minimization of the negative logarithm of the likelihood function involves finding a set of (“optimal”) parameters  $\widetilde{W}$ . Due to the limitation of training set, in general  $\widetilde{W} \neq W_0$  but at the limit  $P \rightarrow \infty$  ( $P$  being the number of training patterns)  $\widetilde{W} \xrightarrow{P \rightarrow \infty} W_0$ .

Considering the Nabla operator  $\nabla$  and the negative logarithm of likelihood  $E$ :

$$\nabla^T = \begin{pmatrix} \frac{\partial}{\partial w_1} & \cdots & \frac{\partial}{\partial w_W} \end{pmatrix}, \quad E = -\ln \mathcal{L}(W) = -\ln \prod_{p=1}^P p(\mathbf{x}_p|W)$$

❖  $J, K$

then the following matrices are defined:

$$J = \mathcal{E}\{(\nabla \nabla^T) E\} = -\mathcal{E}\left\{\sum_{p=1}^P (\nabla \nabla^T) \ln p(\mathbf{x}_p|W)\right\} = -\sum_{p=1}^P (\nabla \nabla^T) \ln p(\mathbf{x}_p|W_0)$$

and

$$\begin{aligned} K &= \mathcal{E}\{(\nabla E)(\nabla E)^T\} = \mathcal{E}\left\{\left(\sum_{p=1}^P \nabla \ln p(\mathbf{x}|W)\right) \left(\sum_{p=1}^P \nabla \ln p(\mathbf{x}|W)\right)^T\right\} \\ &= \left(\sum_{p=1}^P \nabla \ln p(\mathbf{x}|W_0)\right) \left(\sum_{p=1}^P \nabla \ln p(\mathbf{x}|W_0)\right)^T \end{aligned}$$

For  $P$  sufficiently large it is possible to approximate the distribution of  $\widetilde{W} - W_0$  by the (Gaussian) normal distribution  $N_N(\widehat{\mathbf{0}}, J^{-1}KJ^{-1})$  (here and below  $W$  are seen as vectors, i.e. column matrices).

*Proof.*  $E$  is minimized with respect to  $W$ , then:

$$\nabla E|_{\widetilde{W}} = \widehat{\mathbf{0}} \Rightarrow \sum_{p=1}^P \nabla \ln p(\mathbf{x}_p|\widetilde{W}) = \widehat{\mathbf{0}}$$

Considering  $P$  reasonably large then  $\widetilde{W}$  and  $W_0$  are sufficiently close and a Taylor series development may be done around  $W_0$  for  $\nabla E|_{\widetilde{W}}$ :

$$\widehat{\mathbf{0}} = \nabla E|_{\widetilde{W}} = \nabla E|_{W_0} + H|_{W_0}(\widetilde{W} - W_0) + \mathcal{O}((\widetilde{W} - W_0)^2)$$

where  $H = (\nabla \nabla^T) E$  and is named the Hessian. Note that here and below  $\widetilde{W}$  and  $W_0$  are to be seen as vectors (column matrices). Finally:

$$\widetilde{W} - W_0 \simeq H^{-1}|_{W_0} \nabla E|_{W_0} \Rightarrow \mathcal{E}\{\widetilde{W} - W_0\} = \lim_{P \rightarrow \infty} H^{-1}|_{W_0} \nabla E|_{W_0} = \widehat{\mathbf{0}}$$

as  $\widetilde{W} \xrightarrow{P \rightarrow \infty} W_0$ .

Also:

$$\mathcal{E}\{(\widetilde{W} - W_0)(\widetilde{W} - W_0)^T\} = \mathcal{E}\{H^{-1} \nabla E \nabla^T E H^{-1 T}\} = J^{-1} K J^{-1} \quad (7.11)$$

(by using the matrix property  $(AB)^T = B^T A^T$ ).  $\square$

**Definition 7.3.1.** The **deviance**  $D$  of a pattern vector is defined as being twice the expectancy of log-likelihood of the best model minus the log-likelihood of current model, the best model being the true model or an exact fit, also named a saturated model:

$$D = 2\mathcal{E}\{\ln p_{true}(\mathbf{x}) - \ln p(\mathbf{x}|\widetilde{W})\}$$

The deviance may be approximated by:

$$D \simeq 2L + \text{Tr}(KJ^{-1})$$

*Proof.* Considering the Taylor series development of  $\ln p(\mathbf{x}|\widetilde{W})$  around  $W_0$ :

$$\ln p(\mathbf{x}|\widetilde{W}) \simeq \ln p(\mathbf{x}|W_0) + \nabla^T \ln p(\mathbf{x}|W)|_{W_0}(\widetilde{W} - W_0) + \frac{1}{2}(\widetilde{W} - W_0)^T H|_{W_0}(\widetilde{W} - W_0)$$

It is assumed that the gradient of asymptotic divergence (7.10)(see also (7.9)) is zero at  $W_0$ :

$$\nabla L = \mathcal{E}\{\nabla \ln p(\mathbf{x}|W_0)\} = \widehat{\mathbf{0}}$$

(as it hits an minima). The following matrix relation is also true:

$$(\widetilde{W} - W_0)^T H|_{W_0}(\widetilde{W} - W_0) = \text{Tr}(H|_{W_0}(\widetilde{W} - W_0)(\widetilde{W} - W_0)^T)$$

(may be proven by making  $H|W_0$  diagonal, using its eigenvectors, is a symmetrical matrix, see mathematical appendix) and then, from the definition, the deviance approximation is:

$$D \simeq 2L - \mathcal{E}\{\text{Tr}(H|W_0)(\widetilde{W} - W_0)(\widetilde{W} - W_0)^T\}$$

and finally, using also (7.11):

$$D \simeq 2L + \text{Tr}\left[J\mathcal{E}\{(\widetilde{W} - W_0)(\widetilde{W} - W_0)^T\}\right] = 2L + \text{Tr}(KJ^{-1}) \quad \square$$

Considering a sum over training examples, instead of integration over whole  $X$ , the deviance  $D_N$  for a given training set may be approximated as

$$D_N \simeq 2 \sum_{p=1}^P \ln \frac{p_{\text{true}}(\mathbf{x}_p)}{p(\mathbf{x}_p|\widetilde{W})} + \text{Tr}(KJ^{(-1)})$$

information criterion

where the left term of above equation is named *information criterion*.

## CHAPTER 8

# Single Layer Neural Networks

### 8.1 Linear Separability

#### 8.1.1 Discriminant Functions

##### *Two Classes Case*

Let consider the problem of classification in two classes with linear decision boundary such that the classes are separated by a hyperplane in the pattern space. Then the discriminant function is the equation describing that hyperplane and so it is linear in  $\mathbf{x}$ . See also figure 8.1 on the following page.

$$y(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + w_0 \quad (8.1)$$

The  $\mathbf{w}$  is the parameter vector and  $w_0$  is named *bias*. For  $y(\mathbf{x}) > 0$  the pattern  $\mathbf{x}$  is assigned to one class, for  $y(\mathbf{x}) < 0$  it is assigned to the other class, the hyperplane being defined by  $y(\mathbf{x}) = 0$ .

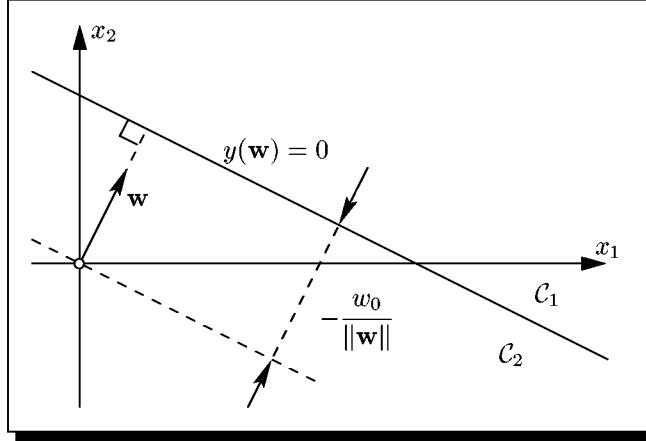
❖  $\mathbf{w}, w_0$

Considering two vectors  $\mathbf{x}_1$  and  $\mathbf{x}_2$  contained within hyperplane (decision boundary) then  $y(\mathbf{x}) = y(\mathbf{x}_2) = 0$ . From (8.1)  $\mathbf{w}^T(\mathbf{x}_1 - \mathbf{x}_2) = 0$ , i.e.  $\mathbf{w}$  is normal to any vector in the hyperplane and then it is normal to the hyperplane  $y(\mathbf{w}) = 0$ . See figure 8.1 on the next page.

The distance between the origin and the hyperplane is given by the scalar product between a versor perpendicular on the plane  $\frac{\mathbf{w}}{\|\mathbf{w}\|}$  and a vector pointing to a point in the plane  $\mathbf{x}$ ,

---

<sup>8.1</sup>See [Bis95] pp. 77-89.



**Figure 8.1:** Linear discriminant in a two dimensional pattern space with two classes.

defined by  $y(\mathbf{x}) = 0$ . Then:

$$\text{distance} = \frac{\mathbf{w}^T \mathbf{x}}{\|\mathbf{w}\|} = -\frac{w_0}{\|\mathbf{w}\|}$$

such that the *bias* defines the shift of the hyperplane from origin.

❖ N

Considering  $N$  to be the dimension of the pattern space, the whole classification problem may be transferred to a  $N + 1$  dimensional pattern space by considering the translations

$$\mathbf{x} \rightarrow \tilde{\mathbf{x}} = (1, \mathbf{x}) \quad \text{and} \quad \mathbf{w} \rightarrow \tilde{\mathbf{w}} = (w_0, \mathbf{w})$$

such that the discriminant equation becomes:

$$y(\mathbf{x}) = \tilde{\mathbf{w}}^T \tilde{\mathbf{x}}$$

defining a hyperplane in the  $N + 1$  dimension space, *passing trough the origin* (bias is 0 now).

The whole process may be expressed in terms of one neuron which have  $N + 1$  inputs and one output being the weighted sum of its inputs  $y(\tilde{\mathbf{x}}) = 1 \cdot w_0 + \sum_{i=1}^N w_i x_i$ . See figure 8.2 on the facing page.

### Multiple Classes Case

Let consider several classes  $\{\mathcal{C}_k\}_{k=\overline{1,K}}$  and one linear discriminant function for each class:

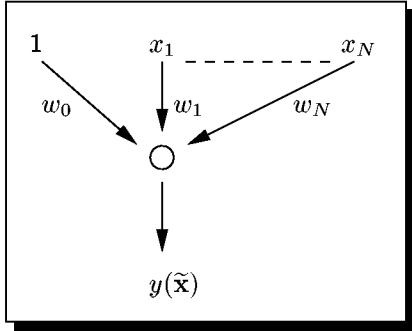
$$y_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} + w_{k0} \quad , \quad k = \overline{1, K} \quad (8.2)$$

❖ K

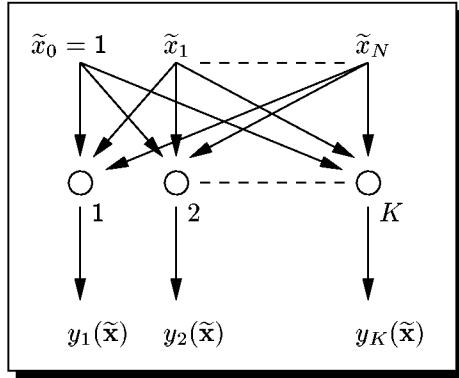
such that a pattern  $\mathbf{x}$  is assigned to class  $\mathcal{C}_k$  if  $y_k(\mathbf{x}) = \max_{\ell=1, K} y_\ell(\mathbf{x})$ ,  $K$  being the dimension of the output space.

The decision boundary between classes  $\mathcal{C}_k$  and  $\mathcal{C}_\ell$  is given by the equation  $y_k(\mathbf{x}) = y_\ell(\mathbf{x})$ :

$$(\mathbf{w}_k - \mathbf{w}_\ell)^T \mathbf{x} + (w_{k0} - w_{\ell0}) = 0$$



**Figure 8.2:** A single, simple, neuron performs the weighted sum of its inputs and may act as a linear classifier.



**Figure 8.3:** A single layer of neurons may act as a linear classifier for \$K\$ classes. The connection between input \$i\$ and neuron \$k\$ is weighted by \$\tilde{w}\_{ki}\$ being the component \$i\$ of vector \$\tilde{\mathbf{w}}\_k\$.

which is the equation of a hyperplane in the pattern space.

The distance from the hyperplane \$(k, \ell)\$ to the origin is:

$$\text{distance}_{(k, \ell)} = -\frac{w_{k0} - w_{\ell0}}{\|\mathbf{w}_k - \mathbf{w}_\ell\|}$$

Similarly to the previous two classes case it is possible to move to the \$N + 1\$ space by the transformation:

$$\mathbf{x} \rightarrow \tilde{\mathbf{x}} = (1, \mathbf{x}) \quad \text{and} \quad \mathbf{w}_k \rightarrow \tilde{\mathbf{w}}_k = (w_{k0}, \mathbf{w}_k) \quad , \quad k = \overline{1, K}$$

and then the whole process may be represented by a neural network having one layer of \$K\$ neurons and \$N + 1\$ inputs. See figure 8.3.

Note that if a matrix \$W\$ is built using \$\tilde{\mathbf{w}}\_k^T\$ as rows then network output is simply written as:

$$\mathbf{y}(\tilde{\mathbf{x}}) = W\tilde{\mathbf{x}}$$

The training of the network consists in finding the adequate \$W\$. A new vector \$\mathbf{x}\$ is assigned to the class \$\mathcal{C}\_k\$ for which the corresponding neuron \$k\$ have the biggest output \$y\_k(\mathbf{x})\$.

linear  
separability

**Definition 8.1.1.** Considering a set of pattern vectors, they are called **linearly separable** if they may be separated by a set of hyperplanes as decision boundaries in the pattern space.

**Proposition 8.1.1.** The regions  $X_k$  defined by linear discriminant functions  $y_k(\mathbf{x})$  are simply connected and convex.

*Proof.* Let consider two vector patterns:  $\mathbf{x}_a, \mathbf{x}_b \in X_k$ .

Then  $y_k(\mathbf{x}_a) = \max_{\ell=1,K} y_\ell(\mathbf{x}_a)$  and  $y_k(\mathbf{x}_b) = \max_{\ell=1,K} y_\ell(\mathbf{x}_b)$ . Any point on the line connecting  $\mathbf{x}_a$  and  $\mathbf{x}_b$  may be defined by the vector:

$$\mathbf{x}_c = t\mathbf{x}_a + (1-t)\mathbf{x}_b \quad \text{where } t \in [0, 1]$$

(see also Jensen's inequality in mathematical appendix).

Also  $y_k(\mathbf{x}_c) = ty_k(\mathbf{x}_a) + (1-t)y_k(\mathbf{x}_b)$  and then  $y_k(\mathbf{x}_c) = \max_{\ell=1,K} y_\ell(\mathbf{x}_c)$ , i.e.  $\mathbf{x}_c \in X_k$  (because  $y_\ell$  are linear).

Then any line connecting two points, from the same  $X_k$ , is contained in the  $X_k$  domain  $\Leftrightarrow X_k$  is convex and simple connected.  $\square$

### 8.1.2 Neuronal Memory Capacity

Let consider one neuron with  $N$  inputs and one output which have to learn  $P$  pattern vectors. All input vectors belong to one of two classes, i.e. either  $\mathcal{C}_1$  or  $\mathcal{C}_2$  and the output of neuron indicates to which class the input belongs (e.g.  $y(\mathbf{x}) = 1$  for  $\mathbf{x} \in \mathcal{C}_1$  and  $y(\mathbf{x}) = -1$  for  $\mathbf{x} \in \mathcal{C}_2$ ) Considering that the input vectors are points in  $\mathbb{R}^N$  space the neuron may learn only those cases where the inputs are linearly separable by a hyperplane. As the number of linearly separable cases is limited so is the learning capacity/memory of a single neuron (and of course the learning capacity of the network is limited as well).

general position

Let consider that there are  $P$  fixed points in  $\mathbb{R}^N$ , in *general position*, i.e. for  $N \geq 2$  there are not  $N$  or fewer points linearly dependent. Let consider that either of these points may belong to class  $\mathcal{C}_1$  or  $\mathcal{C}_2$ , the total number of combinations is  $2^P$  (as each point brings up 2 cases, independently of the others). From the  $2^P$  cases some are linearly separable and some are not, let  $F(P, N)$  be the number of linearly separable cases. Then the probability of linear separability is:

$$\text{Probability of linear separability} = \frac{F(P, N)}{2^P}$$

**Proposition 8.1.2.** The number of linearly separable cases is given by:

$$F(P, N) = 2 \sum_{i=0}^N \binom{P-1}{i} \tag{8.3}$$

(where  $0! = 1$  by definition).

*Proof.* It is proven by induction.

A hyperplan in  $\mathbb{R}^N$  is defined by the equation  $\mathbf{a}^T \mathbf{x} + b = 0$  (where  $\mathbf{x}$  is a point contained in the hyperplan), i.e. is defined by  $N+1$  parameters.

<sup>8.1.2</sup>See [Rip96] pp. 119–120.

Let consider first the case:  $P \leq N + 1$ . Then the set of equations:

$$\mathbf{a}^T \mathbf{x}_i + b = y_i \begin{cases} > 0 & \text{one side of hyperplan} \\ < 0 & \text{other side of hyperplan} \\ = 0 & \text{contained in the hyperplan} \end{cases}$$

define the hyperplan parameters. As there are  $N + 1$  parameters and at most  $N + 1$  equations and the points are in general position then the system of equations with unknowns  $\mathbf{a}$  and  $b$  (hyperplan parameters) have always a solution, i.e. there is always a way to separate the two classes with a hyperplan (there may be several solutions); then:

$$F(P, N) = 2^P \quad \text{for } P \leq N + 1$$

Let now consider the case  $P \geq N + 1$ . A recurrent formula is attempted for  $F(P + 1, N)$ . Let consider that  $P$  linearly separable points are already "in position", i.e. one case from  $F(P, N)$ , and a new one is added. There are two cases to be considered:

- The new point sides on one side only of any separating (the  $P$  set) hyperplane. Then the new set is separable only if the new point is on the "correct" side, i.e. the same as for its class.
- If the above it's not true then it is possible to choose the separating hyperplane as to pass trough the new point. Then no matter to which class the new point is assigned, the new set is linearly separable.

Let considering again only the  $P$  points set and the hyperplane as chosen above. If all points are projected into a (new) hyperplane perpendicular on the separating one, then the points in the new hyperplane are linearly separate (by the hyperline given by the intersection of the two hyperplanes). This means that the number of possibilities in this situation is  $F(P, N - 1)$  and the number of combinations is  $2F(P, N - 1)$ , as the  $P + 1$ -th point may be assigned to either class.

Finally, the first case analyzed above gives  $F(P, N)$  minus the number of possibilities in the second case (i.e.  $F(P, N - 1)$ ) and the second case gives  $2F(P, N - 1)$ . Thus the wanted recurrent formula is:

$$F(P + 1, N) = [F(P, N) - F(P, N - 1)] + 2F(P, N - 1) = F(P, N) + F(P, N - 1) \quad (8.4)$$

Induction: for  $F(P, N - 1)$ , from (8.3), the expression is:

$$F(P, N - 1) = 2 \sum_{i=0}^{N-1} \binom{P-1}{i} = 2 \sum_{i=1}^N \binom{P-1}{i-1}$$

and then, using (8.4), (8.3) and the above equation, the expression for  $F(P + 1, N)$  is:

$$F(P + 1, N) = F(P, N) + F(P, N - 1) = 2 + 2 \sum_{i=1}^N \left[ \binom{P-1}{i} + \binom{P-1}{i} \right] = 2 \sum_{i=0}^N \binom{P}{i}$$

i.e. is of the same form as (8.3) (the property  $\binom{P-1}{i} + \binom{P-1}{i} = \binom{P}{i}$  was used here<sup>1</sup>).

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 8.4 on the following page, the other one is its mirror image. So the formula (8.3) checks also for an initial case.  $\square$

The probability of linear separability is then:

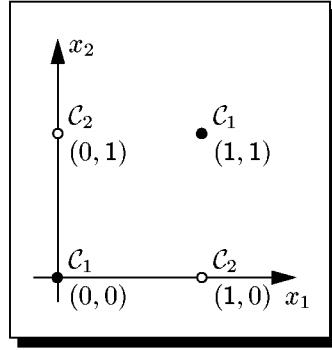
$$P_{\text{linear separability}} = \begin{cases} 1 & P \leq N + 1 \\ \frac{1}{2^{P-1}} \sum_{i=0}^N \binom{P-1}{i} & P \geq N + 1 \end{cases}$$

and then

$$P_{\text{linear separability}} = \begin{cases} > 0.5 & \text{for } P < 2(N + 1) \\ = 0.5 & \text{for } P = 2(N + 1) \\ < 0.5 & \text{for } P > 2(N + 1) \end{cases}$$

---

<sup>1</sup>See mathematical appendix.



**Figure 8.4:** The XOR problem. The vectors marked with black circles are from one class; the vectors marked with white circles are from the other class.  $C_1$  and  $C_2$  are not linearly separable.

i.e. the memory capacity of a single neuron is around  $2(N + 1)$ .

### Remarks:

- As the points (pattern vectors) from the same class are usually (to some extent) correlated then the memory capacity of a single neuron is much higher than the above reasoning suggests.
- A simple problem of classification where the pattern vectors are not linearly separable is the *exclusive-or* (XOR), in the bidimensional space.

The vectors  $(0,0)$  and  $(1,1)$  are from one class ( $0 \text{ xor } 0 = 0, 1 \text{ xor } 1 = 0$ ); while the vectors  $(0,1)$  and  $(1,0)$  are from the other ( $1 \text{ xor } 0 = 1, 0 \text{ xor } 1 = 1$ ). See figure 8.4.

### 8.1.3 Logistic discrimination

The discriminant functions may be generalized by replacing the linear functions (8.2) with a monotone function applied to the linear combination of  $\mathbf{w}$  and  $\mathbf{x}$

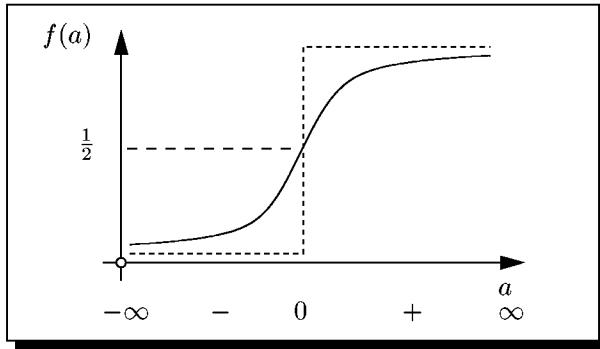
$$y_k(\mathbf{x}) = f(\mathbf{w}_k^T \mathbf{x} + w_{k0}) \quad , \quad k = \overline{1, K}$$

activation  
function

where  $f$  is named *activation function*.

From the Bayesian theorem:

$$\begin{aligned} p(\mathcal{C}_k | \mathbf{x}) &= \frac{p(\mathbf{x} | \mathcal{C}_k) P(\mathcal{C}_k)}{p(\mathbf{x})} = \frac{p(\mathbf{x} | \mathcal{C}_k) P(\mathcal{C}_k)}{\sum_{\ell=1}^K p(\mathbf{x} | \mathcal{C}_\ell) P(\mathcal{C}_\ell)} = \\ &= \frac{1}{1 + \frac{\sum_{\ell=1, \ell \neq k}^K p(\mathbf{x} | \mathcal{C}_\ell) P(\mathcal{C}_\ell)}{p(\mathbf{x} | \mathcal{C}_k) P(\mathcal{C}_k)}} \equiv \frac{1}{1 + e^{-a}} \end{aligned} \quad (8.5)$$



**Figure 8.5:** The logistic signal activation function. The particular case of step function is drawn with a dashed line. The maximum value of the function is 1.

where:

$$a \equiv \ln \frac{p(\mathbf{x}|\mathcal{C}_k)}{\sum_{\substack{\ell=1 \\ \ell \neq k}}^K p(\mathbf{x}|\mathcal{C}_\ell)} + \ln \frac{P(\mathcal{C}_k)}{\sum_{\substack{\ell=1 \\ \ell \neq k}}^K P(\mathcal{C}_\ell)} \quad (8.6)$$



#### Remarks:

- ➡ For the Gaussian model with the same variance matrix  $\Sigma$  for all classes:

$$p(\mathbf{x}|\mathcal{C}_k) = \frac{1}{(2\pi)^{n/2} \sqrt{|\Sigma|}} \exp \left[ -\frac{(\mathbf{x} - \boldsymbol{\mu}_k)^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu}_k)}{2} \right], \quad k = \overline{1, K}$$

and two classes  $\mathcal{C}_1$  and  $\mathcal{C}_2$  then the expression of  $a$  becomes:

$$a = \mathbf{w}^T \mathbf{x} + w_0$$

where ( $\Sigma$  is symmetric):

$$\mathbf{w} = \Sigma^{-1} (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)$$

$$w_0 = -\frac{\boldsymbol{\mu}_1^T \Sigma^{-1} \boldsymbol{\mu}_1 - \boldsymbol{\mu}_2^T \Sigma^{-1} \boldsymbol{\mu}_2}{2} + \ln \frac{P(\mathcal{C}_1)}{P(\mathcal{C}_2)}$$

Then, by choosing the *logistic sigmoid activation function* as  $f(a) = \frac{1}{1+e^{-a}}$  — sigmoid function see figure 8.5 — the meaning of the neuron output becomes simply the posterior probability  $P(\mathcal{C}_k|\mathbf{x})$ .

- ➡ The logistic sigmoid activation function have also the property of mapping the interval  $(-\infty, \infty)$  into  $[0, 1]$  and thus limiting the neuron output.

Another choice for the activation function of neuron is the threshold (step) function:

$$f(a) = \begin{cases} 1 & \text{for } a \geq 0 \\ 0 & \text{for } a < 0 \end{cases} \quad (8.7)$$

threshold  
function

perceptron  
adaline

and the neurons having it are called *perceptrons* or *adaline*.



#### Remarks:

- The step function is the particular case of the logistic signal activation function  $f(a) = \frac{1}{1+e^{-ca}}$  when  $c \rightarrow \infty$ . See figure 8.5 on the page before.

### 8.1.4 Binary pattern vectors

A binary pattern vector  $\mathbf{x}$  have its components  $x_i \in \{0, 1\}$ .

Let  $P_{ki}$  be the probability that the component  $i$  of vector  $\mathbf{x} \in \mathcal{C}_k$  is  $x_i = 1$ , respectively  $(1 - P_{ki})$  is the probability of  $x_i = 0$ . Then:

$$p(x_i | \mathcal{C}_k) = P_{ki}^{x_i} (1 - P_{ki})^{1-x_i}$$

Bernoulli distribution

also named Bernoulli distribution. Assuming that the components of the pattern vector  $\mathbf{x}$  are *statistically* independent then:

$$p(\mathbf{x} | \mathcal{C}_k) = \prod_{i=1}^N P_{ki}^{x_i} (1 - P_{ki})^{1-x_i}$$

By taking the discriminant function in the form of:

$$y_k(\mathbf{x}) = \ln P(\mathbf{x} | \mathcal{C}_k) + \ln P(\mathcal{C}_k)$$

then it may be written as:

$$y_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} + w_{0k}$$

where:

$$w_{ki} = \ln P_{ki} - \ln(1 - P_{ki}) \quad , \quad i = \overline{1, N} \quad \text{and}$$

$$w_{0k} = \sum_{i=1}^N \ln(1 - P_{ki}) + \ln P(\mathcal{C}_k)$$

Similar as above, from the Bayesian theorem, see (8.5) and (8.6), — for two classes  $\mathcal{C}_1$  and  $\mathcal{C}_2$  — the posterior probability  $P(\mathcal{C}_k | \mathbf{x})$  may be expressed as the output of the neuron having the logistic sigmoidal activation function:

$$P(\mathcal{C}_1 | \mathbf{x}) = f(\mathbf{w}^T \mathbf{x} + w_0) = \frac{1}{1 + \exp[-(\mathbf{w}^T \mathbf{x} + w_0)]}$$

where:

$$w_i = \ln \frac{P_{1i}}{P_{2i}} - \ln \frac{1 - P_{1i}}{1 - P_{2i}} \quad , \quad i = \overline{1, N} \quad \text{and}$$

$$w_0 = \sum_{i=1}^N \ln \frac{1 - P_{1i}}{1 - P_{2i}} + \ln \frac{P(\mathcal{C}_1)}{P(\mathcal{C}_2)}$$

and  $P(\mathcal{C}_2 | \mathbf{x})$  have a similar form (obtainable by swapping  $1 \leftrightarrow 2$ ).

### 8.1.5 Generalized linear discriminants

Generalized linear discriminant functions are obtained by replacing  $\mathbf{x}$  with a vectorial function of it:  $\varphi : X \rightarrow X$  having the same dimension. The discriminant functions then becomes

$$y_k(\mathbf{x}) = \mathbf{w}_k^T \varphi(\mathbf{x}) + w_{0k} \quad (8.8)$$

and, by switching to the  $N + 1$  space:

$$y_k(\tilde{\mathbf{x}}) = \tilde{\mathbf{w}}_k^T \tilde{\varphi}(\mathbf{x})$$

where  $\tilde{\varphi}_0(\mathbf{x}) \equiv 1$ .

By building the  $(N + 1) \times K$  matrix  $W$  having the weights  $\tilde{\mathbf{w}}_k$  associated with each output neuron as rows:  $\diamond W$

$$W = \begin{pmatrix} \tilde{w}_{10} & \cdots & \tilde{w}_{1N} \\ \vdots & \ddots & \vdots \\ \tilde{w}_{K0} & \cdots & \tilde{w}_{KN} \end{pmatrix}$$

then:

$$\mathbf{y}(\mathbf{x}) = W\tilde{\varphi}(\mathbf{x})$$

## 8.2 The Least Squares Technique

### 8.2.1 The Error Function

Let  $\{\mathbf{x}_p\}_{p=1,P}$  be the training set and  $\{\mathbf{t}_p\}_{p=1,P}$  the desired output of the network. Then the *sum-of-squares error function* is:

$$E(W) = \frac{1}{2} \sum_{p=1}^P \sum_{k=1}^K [y_k(\mathbf{x}_p, \mathbf{w}_k) - t_{kp}]^2 \quad (8.9)$$

where  $t_{kp}$  is the component  $k$  of desired vector  $\mathbf{t}_p$ .  $\diamond t_{kp}$

Considering the generalized linear discriminants of the form (8.8) then  $E(W)$  is a quadratic function of weights and its derivatives are linear function of weights and then the minimum of the error function may be found exactly in closed form.

$$E(W) = \frac{1}{2} \sum_{p=1}^P \sum_{k=1}^K [\tilde{\mathbf{w}}_k^T \tilde{\varphi}(\mathbf{x}_p) - t_{kp}]^2 = \frac{1}{2} \sum_{p=1}^P [W\tilde{\varphi}(\mathbf{x}_p) - \mathbf{t}_p]^T [W\tilde{\varphi}(\mathbf{x}_p) - \mathbf{t}_p] \quad (8.10)$$

(here  $W$  contains  $\tilde{\mathbf{w}}^T$  on rows).

#### The geometrical interpretation of error function

Let consider the  $P$ -dimensional vector  $\vec{y}_k$  which components are the outputs of the same  $\diamond \vec{y}_k$

<sup>8.2</sup>See [Bis95] pp. 89–98.

neuron  $k$  while the network is presented with the  $\mathbf{x}_p$  training vectors:

$$\vec{y}_k^T = (\tilde{\mathbf{w}}_k^T \tilde{\varphi}(\mathbf{x}_1) \quad \dots \quad \tilde{\mathbf{w}}_k^T \tilde{\varphi}(\mathbf{x}_P))$$

Using the components of the  $\tilde{\varphi}(\mathbf{x})$  vectorial function:  $\tilde{\varphi}(\mathbf{x})^T = (\tilde{\varphi}_0(\mathbf{x}) \quad \dots \quad \tilde{\varphi}_N(\mathbf{x}))$  and its value for the training set vectors  $\mathbf{x}_p$ , it is possible to build the vector:

$$\vec{\varphi}_i^T = (\tilde{\varphi}_i(\mathbf{x}_1) \quad \dots \quad \tilde{\varphi}_i(\mathbf{x}_P))$$

As the component  $p$  of vector  $\vec{y}_k$  is  $\{\vec{y}_k\}_p = \sum_{i=0}^N \tilde{w}_{ki} \tilde{\varphi}_i(\mathbf{x}_p)$  then  $\vec{y}_k$  may be written as a linear combination of  $\vec{\varphi}_i$  in the  $N + 1$  space:

$$\vec{y}_k = \sum_{i=0}^N \tilde{w}_{ki} \vec{\varphi}_i \quad (8.11)$$

( $\tilde{w}_{ki}$  being the component  $i$  of vector  $\tilde{\mathbf{w}}_k$ ).

The sum of  $\vec{y}_k$  vectors is  $\vec{y}_{\text{total}} = \sum_{k=1}^K \vec{y}_k = \sum_{i=0}^N \left( \sum_{k=1}^K \tilde{w}_{ki} \right) \vec{\varphi}_i$  and, again, is a linear combination of  $\vec{\varphi}_i$ .

Similar, the vector  $\vec{t}_k$  may be build, using the target values for output neuron  $k$  given the input vector  $\mathbf{x}_p$  as being  $t_{kp}$ :

$$\vec{t}_k^T = (t_{k1} \quad \dots \quad t_{kP})$$

Finally, the sum-of-squares error function (8.9) may be written as:

$$\begin{aligned} E &= \frac{1}{2} \sum_{p=1}^P \sum_{k=1}^K \left( \sum_{\ell=0}^N \tilde{w}_{k\ell} \tilde{\varphi}_\ell(\mathbf{x}_p) - t_{kp} \right)^2 = \frac{1}{2} \sum_{k=1}^K \sum_{p=1}^P (\{\vec{y}_k\}_p - t_{kp})^2 \\ &= \frac{1}{2} \sum_{k=1}^K \|\vec{y}_k - \vec{t}_k\|^2 \end{aligned} \quad (8.12)$$

Let make the reasonable assumption that the number of inputs is smaller than the number of sample vectors in the training set, i.e.  $N + 1 \leq P$  (what happens if this is not true is discussed later). Let consider the space of dimension  $P$ : the set of  $N + 1$  vectors  $\{\vec{\varphi}_i\}$  define a subspace in which all  $\vec{y}_k$  are contained, the vector  $\vec{t}_k$  being in general not included.

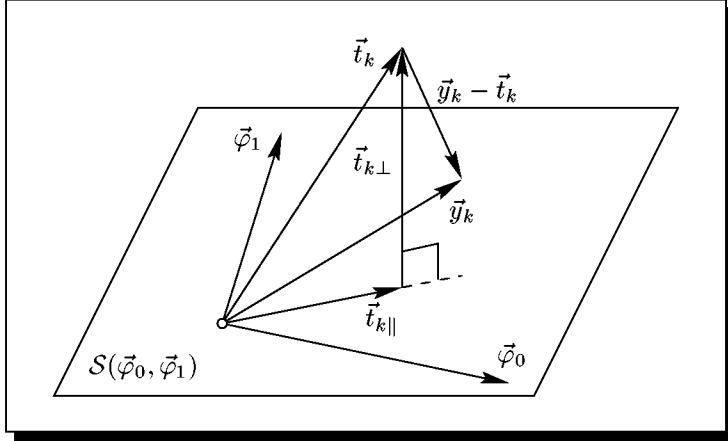
Then the sum-of-squares error function (8.12) represents simply the sum of all distances between  $\vec{y}_k$  and  $\vec{t}_k$ . See figure 8.6 on the facing page.

The  $\vec{t}_k$  may be decomposed into two components:

$$\vec{t}_{k\parallel} \in \mathcal{S}(\vec{\varphi}_0, \dots, \vec{\varphi}_N) \quad \text{and} \quad \vec{t}_{k\perp} \perp \mathcal{S}(\vec{\varphi}_0, \dots, \vec{\varphi}_N)$$

❖  $\mathcal{S}$

where  $\mathcal{S}(\vec{\varphi}_0, \dots, \vec{\varphi}_N)$  is the sub-space defined by the set of functions  $\{\vec{\varphi}_i\}$ . See figure 8.6.



**Figure 8.6:** The error vector  $\vec{y}_k - \vec{t}_k$  for output neuron  $k$ .  $S(\vec{\varphi}_0, \vec{\varphi}_1)$  represents the subspace defined by the set of functions  $\{\vec{\varphi}_i\}$  in a bidimensional case — one input neuron plus bias. The  $\vec{y}_k$  and  $\vec{t}_{k\parallel}$  are included in  $S(\vec{\varphi}_0, \vec{\varphi}_1)$  space.

The minimum of error function:  $E = \frac{1}{2} \sum_{k=1}^K (\vec{y}_k - \vec{t}_k)^T (\vec{y}_k - \vec{t}_k)$  (see (8.11) and (8.12)) with respect to  $\tilde{w}_{ki}$  is found from the condition that its derivatives are zero:

$$\frac{\partial E}{\partial \tilde{w}_{ki}} = 0 \Leftrightarrow \vec{\varphi}_i^T (\vec{y}_k - \vec{t}_k) = 0, \quad k = \overline{1, K}, \quad i = \overline{0, N} \quad (8.13)$$

and because  $\vec{t}_k = \vec{t}_{k\parallel} + \vec{t}_{k\perp}$  and  $\vec{\varphi}_i^T \vec{t}_{k\perp} = 0$  (by choice of  $\vec{t}_{k\parallel}$  and  $\vec{t}_{k\perp}$ ) then the above condition is equivalent with:

$$\vec{\varphi}_i^T (\vec{y}_k - \vec{t}_{k\parallel}) = 0 \Leftrightarrow \vec{y}_k = \vec{t}_{k\parallel}, \quad k = \overline{1, K} \quad (8.14)$$

i.e.  $\{\tilde{w}_{ki}\}$  should be chosen such that  $\vec{y}_k = \vec{t}_{k\parallel}$  — see also figure 8.6.

Note that there is *always* a “residual” error due to the  $\vec{t}_{k\perp}$ .

Assuming that the network is optimized such that  $\vec{y}_k = \vec{t}_{k\parallel}$ ,  $\forall k \in \{1, \dots, K\}$  then  $\vec{y}_k - \vec{t}_k = -\vec{t}_{k\perp}$  and the error function (8.12) becomes:

$$E_{\min} = \frac{1}{2} \sum_{k=1}^K \|\vec{y}_k - \vec{t}_k\|^2 = \frac{1}{2} \sum_{k=1}^K \|\vec{t}_{k\perp}\|^2$$

### 8.2.2 The Pseudo-inverse solution

Let build the  $P \times (N + 1)$  matrix  $\Phi$  from  $\vec{\varphi}_i$  used as columns:

$$\Phi = \begin{pmatrix} \tilde{\varphi}_0(\mathbf{x}_1) & \cdots & \tilde{\varphi}_N(\mathbf{x}_1) \\ \vdots & \ddots & \vdots \\ \tilde{\varphi}_0(\mathbf{x}_P) & \cdots & \tilde{\varphi}_N(\mathbf{x}_P) \end{pmatrix}$$

❖  $\Phi$

❖  $T$

also the  $P \times K$  matrix  $T$  using  $\vec{t}_k$  vectors as columns:

$$T = \begin{pmatrix} t_{11} & \cdots & t_{K1} \\ \vdots & \ddots & \\ t_{1P} & \cdots & t_{KP} \end{pmatrix}$$

From the above matrices and (8.11):  $\vec{y}_k = \Phi W(k,:)^T$ . Then using the above notations, the set of minima conditions (8.14) may be written in matrix form as:

$$(\Phi^T \Phi) W^T - \Phi^T T = \tilde{0}$$

Assuming that the square  $(N + 1) \times (N + 1)$  matrix  $\Phi^T \Phi$  is inversable then the solution for the weights is:

$$W^T = (\Phi^T \Phi)^{-1} \Phi^T T = \Phi^\dagger T \quad (8.15)$$

pseudo-inverse matrix

where  $\Phi^\dagger$  is the pseudo-inverse matrix of  $\Phi$  (which generally is not square) and is defined as:

$$\Phi^\dagger = (\Phi^T \Phi)^{-1} \Phi^T$$

If the  $\Phi^T \Phi$  is not inversable then taking an  $\varepsilon \in \mathbb{R}$  the pseudo-inverse matrix may be defined as:

$$\Phi^\dagger = \lim_{\varepsilon \rightarrow 0} (\Phi^T \Phi + \varepsilon I)^{-1} \Phi^T$$



### Remarks:

- ➔ As the  $\Phi$  matrix is built from the  $\vec{\varphi}_i$  set of vectors, if two of them are parallel (or nearly parallel) then the  $\Phi^T \Phi$  is singular (or nearly singular) — the rank of the matrix will be lower.
- ➔ The case of nearly singularity also leads to large weights necessary to represent the solution  $\vec{y}_k = \vec{t}_{k\parallel}$ . See figure 8.7 on the facing page.
- ➔ In case of two parallel vectors  $\vec{\varphi}_i \parallel \vec{\varphi}_\ell$  the one of them is proportional with another:  $\vec{\varphi}_i \propto \vec{\varphi}_\ell$  and then they may be combined together in the error function and thus reducing the number of dimensions of  $\mathcal{S}$  space.
- ➔ By writing explicitly the minima conditions (8.14) for biases  $w_{k0}$ :

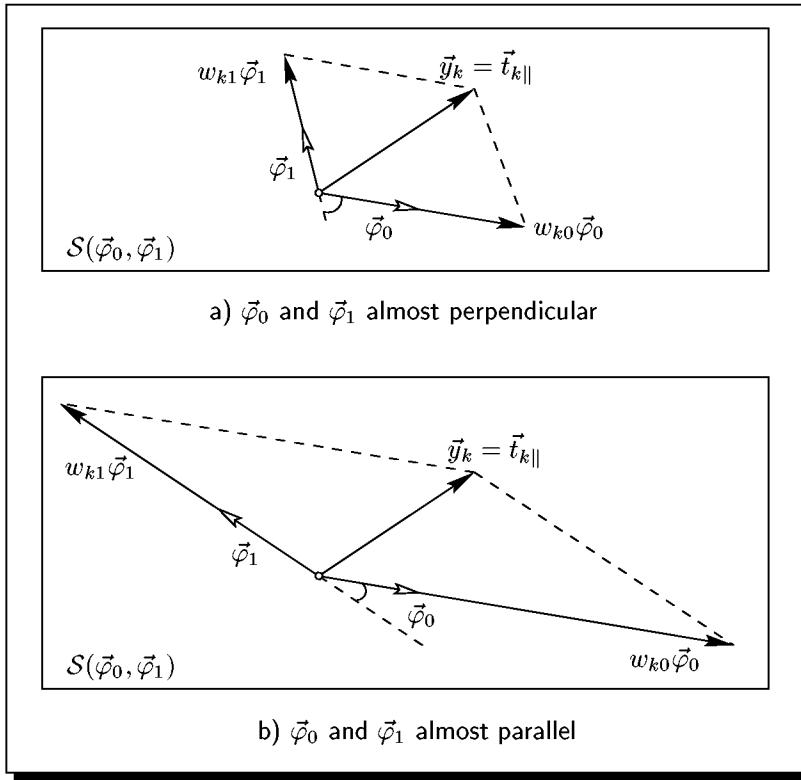
$$\frac{\partial E}{\partial w_{k0}} = \sum_{p=1}^P \left( \sum_{\ell=1}^N w_{k\ell} \tilde{\varphi}_\ell(\mathbf{x}_p) + w_{k0} - t_{kp} \right) = 0$$

$(\tilde{\varphi}_0(\mathbf{x}_p) = 1$ , by construction of  $\tilde{\varphi}_0$ ) the bias may be written as:

$$w_{k0} = \bar{t}_k - \sum_{\ell=1}^N w_{k\ell} \bar{\varphi}_\ell$$

❖  $\bar{t}_k$ ,  $\bar{\varphi}_\ell$

where:



**Figure 8.7:** The solution  $\vec{y}_k = \vec{t}_{k\parallel}$  in a bidimensional space  $S(\vec{\varphi}_0, \vec{\varphi}_1)$ .  
 Figure a presents the case of nearly orthogonal set, figure b presents the case of a nearly parallel set of  $\{\vec{\varphi}_i\}$  functions.  $\vec{t}_{k\parallel}$  and  $\vec{\varphi}_0$  were kept the same in both cases

$$\bar{t}_k = \frac{1}{P} \sum_{p=1}^P t_{kp} \quad \text{and} \quad \bar{\varphi}_\ell = \frac{1}{P} \sum_{p=1}^P \tilde{\varphi}_\ell(\mathbf{x}_p)$$

i.e. the bias compensate the difference between the mean of targeted output  $\bar{t}_k$  and mean of the actual output — over the training set.

- ➡ If the number of vectors in the training set  $P$  is equal with the number of inputs  $N + 1$  then the  $\Phi$  matrix is square and it have an inverse. By multiplying with  $(\Phi^T)^{-1}$  to the left in (8.15):  $\Phi W^T = T \Rightarrow W^T = \Phi^{-1}T$ . Geometrically speaking  $\vec{t}_k \in \mathcal{S}$  and  $\vec{t}_{k\perp} = 0$  — see figure 8.6, i.e. the network is capable to learn perfectly the target and the error function is zero (after training).

If  $P \leq N + 1$  then the  $\vec{t}_k$  vectors are included into a *subspace* of  $\mathcal{S}$  and to minimize the error to zero it us enough to make the *projection* of  $\vec{y}_k$  (into that subspace) equal with  $\vec{t}_k$ . (a situation *similar* — mutatis mutandis — to that represented in figure 8.6 on page 139 but with  $\vec{y}_k$  and  $\vec{t}_k$  swapping places).

This means that just a part of weights are to be adapted (found); the other ones do not count (there are an infinity of solutions).

To have  $P \leq N + 1$  is not normally a good idea because the network acts more as a *memory* rather than as a generalizer (the network have a strong tendency to overadapt).

- ➔ The solution developed in this section does not work if the neuron does not have a linear activation, e.g. sigmoid activation function.

### 8.2.3 The Gradient Descent Solution

The neuronal activation function is supposed to be differentiable and the error function may be expressed as function of weights  $E = E(W)$ . Then an initial value for  $\{w_{ki}\}$  parameters is chosen (usually weights are initialized randomly) and the parameters are modified by small values in the direction of decrease of  $E$ , i.e. in the direction of  $-\nabla E$  (with respect to weights), in small steps:

$$\Delta w_{ki} = w_{(s+1)ki} - w_{(s)ki} = -\eta \frac{\partial E}{\partial w_{ki}} \Big|_{W(s)}, \quad \eta = \text{const.} \in \mathbb{R}^+$$

❖  $t, \eta$   
learning rate

$t$  being the step of iteration (discrete time).  $\eta$  is a positive constant called *learning rate* and governs the speed by which the  $\{w_{ki}\}$  parameters are changed.

Obviously  $-\nabla E$  may be represented as a matrix of the same dimensions as  $W$  and then the above equation may be written simply as:

$$\Delta W = W_{(t+1)} - W_{(t)} = -\eta \nabla E \quad (8.16)$$

and is known as the *delta rule*.

❖  $E_p$

Usually the error function is expressed as a sum over the training set of a  $P$  error terms:  $E = \sum_{p=1}^P E_p(W)$ , then the weight adjustment may be done in steps, for each training vector in turn:

$$\Delta w_{ki} = w_{(t+1)ki} - w_{(t)ki} = -\eta \frac{\partial E_p}{\partial w_{ki}} \Big|_{W(t)}, \quad p = \overline{1, P}$$

#### Remarks:

- ➔ The above procedure is especially useful if the training set is not available from start but rather the vectors are arriving as a time series.
- ➔ The learning parameter  $\eta$  may be chosen to decrease in time, e.g.  $\eta = \frac{\eta_0}{t}$ .

This procedure is very similar to the Robbins–Monro algorithm for finding the root of *derivative* of  $E$ , i.e. the minima of  $E$ .

Assuming general linear discriminant (8.8) then considering the sum-of-squares error function (8.10):

$$E(W) = \frac{1}{2} \sum_{p=1}^P \sum_{k=1}^K \left[ \sum_{i=0}^N \tilde{w}_{ki} \tilde{\varphi}_i(\mathbf{x}_p) - t_{kp} \right]^2, \quad E_p(W) = \frac{1}{2} \sum_{k=1}^K \left[ \sum_{i=0}^N \tilde{w}_{ki} \tilde{\varphi}_i(\mathbf{x}_p) - t_{kp} \right]^2$$

then:

$$\frac{\partial E_p}{\partial w_{k\ell}} = \left[ \sum_{i=0}^N \tilde{w}_{ki} \tilde{\varphi}_i(\mathbf{x}_p) - t_{kp} \right] \tilde{\varphi}_\ell(\mathbf{x}_p) = [y_k(\mathbf{x}_p) - t_{kp}] \tilde{\varphi}_\ell(\mathbf{x}_p)$$

or, in matrix notation:

$$\nabla E_p = [\mathbf{y}(\mathbf{x}_p) - \mathbf{t}_p] \tilde{\varphi}^T(\mathbf{x}_p)$$

where  $\mathbf{y}(\mathbf{x}_p) = W \tilde{\varphi}(\mathbf{x}_p)$ . Then the delta rule<sup>2</sup> (8.16) becomes:

delta rule

$$\Delta W = -\eta [W \tilde{\varphi}(\mathbf{x}_p) - \mathbf{t}_p] \tilde{\varphi}^T(\mathbf{x}_p)$$

So far only networks with identity activation function were discussed. In general neurons have a differentiable activation function  $f$  (the perceptron being an exception) then total input to neuron  $k$  is  $a_{kp} = W(k,:) \tilde{\varphi}(\mathbf{x}_p)$  and:

❖  $f$

$$\mathbf{a}_p = W \tilde{\varphi}(\mathbf{x}_p) , \quad \mathbf{y}(\mathbf{x}_p) = f(\mathbf{a}_p)$$

The sum-of-squares error (8.9), for each training pattern  $p$  is:

$$E_p(W) = \frac{1}{2} [f(\mathbf{a}_p) - \mathbf{t}_p]^T [f(\mathbf{a}_p) - \mathbf{t}_p]$$

and then:

$$\nabla E_p = \{[f(\mathbf{a}_p) - \mathbf{t}_p] \odot f'(\mathbf{a}_p)\} \tilde{\varphi}^T(\mathbf{x}_p)$$

where  $f'$  is the total derivative of  $f$ .

❖  $f'$

*Proof.* From the expression of  $E_p$ :

$$E_p(W) = \frac{1}{2} \sum_{k=1}^K [f(a_{kp}) - t_{kp}]^2 = \frac{1}{2} \sum_{k=1}^K \left[ f \left( \sum_{i=1}^N w_{ki} \tilde{\varphi}_i(\mathbf{x}_p) \right) - t_{kp} \right]^2$$

and then:

$$\frac{\partial E_p}{\partial w_{k\ell}} = [f(a_{kp}) - t_{kp}] f'(a_{kp}) \tilde{\varphi}_\ell(\mathbf{x}_p)$$

which leads directly to the matrix formula above.  $\square$

### Remarks:

► In the case of sigmoid function  $f(x) = \frac{1}{1+e^{-x}}$  the derivative is:

$$f'(x) = \frac{df}{dx} = f(x)[1 - f(x)]$$

In this case writing  $f'$  in terms of  $f$  speeds up the calculation and save some memory on digital simulations.

► It is easily seen that the derivatives are “local”, i.e. depend only on parameters linked to the particular neuron in focus and do not depend on the values linked to other neurons.

---

<sup>2</sup>This equation is also known as the least-mean-square (LMS) rule, the adaline rule and the Widrow-Hoff rule.

- The total derivative over whole training set may be found easily from:

$$\frac{\partial E}{\partial w_{k\ell}} = \sum_{p=1}^P \frac{\partial E_p}{\partial w_{k\ell}}$$

## 8.3 The Perceptron

The *perceptron* (or *adaline*<sup>3</sup>) represents a single layer neural network with threshold activation function (see (8.7)). Usually the activation function is chosen as being odd:

$$f(a) = \begin{cases} +1 & \text{for } a \geq 0 \\ -1 & \text{for } a < 0 \end{cases}$$

### 8.3.1 The Error Function

Considering just one neuron, its output is  $y(\mathbf{x}) = f(\tilde{\mathbf{w}}^T \tilde{\varphi}(\mathbf{x}))$ . Because the output of the single neuron is either  $+1$  or  $-1$  then it may classify just a set of two classes: if  $\tilde{\mathbf{w}}^T \tilde{\varphi} \geq 0$  then the output is  $+1$  and  $\mathbf{x} \in \mathcal{C}_1$ , else  $\tilde{\mathbf{w}}^T \tilde{\varphi} < 0$ , the output is  $-1$  and  $\mathbf{x} \in \mathcal{C}_2$ .

Then, for a *correct classification*,  $t\tilde{\mathbf{w}}^T \tilde{\varphi} > 0, \forall \mathbf{x}$ ; where  $t$  is the target value given the input vector  $\mathbf{x}$ . For a *misclassified* input vector either  $\tilde{\mathbf{w}}^T \tilde{\varphi} > 0$  while  $t = -1$  or vice-versa, i.e.  $-t\tilde{\mathbf{w}}^T \tilde{\varphi} < 0$ .

A good choice for the error function will be:

$$E(\mathbf{w}) = - \sum_{\mathbf{x}_p \in \mathcal{M}} t\tilde{\mathbf{w}}^T \tilde{\varphi}(\mathbf{x}_p) \quad (8.17)$$

❖  $\mathcal{M}$

where  $\mathcal{M}$  is the set of *misclassified* vectors  $\mathbf{x}_p$ .

#### Remarks:

- From the discussion in section 8.1.1 it follows that  $\tilde{\mathbf{w}}^T \tilde{\varphi}(\mathbf{x}_p)$  is proportional to the distance from the misclassified vector  $\tilde{\varphi}(\mathbf{x}_p)$  to the decision boundary.

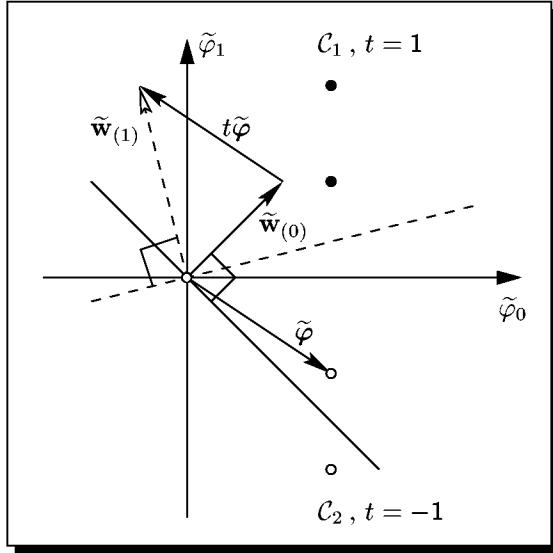
The process of minimizing the function (8.17) is equivalent to shifting the decision boundary such that misclassification becomes minimum.

During the shifting process  $\mathcal{M}$  changes as some previously misclassified vectors becomes correctly classified and vice-versa.

---

<sup>8.3</sup>See [Bis95] pp. 98–105.

<sup>3</sup>From ADaptive LINear Element.



**Figure 8.8:** The learning process for perceptron. White circles are from one class, black ones are from the other. Initially the parameter is  $\tilde{w}_{(0)}$  and the pattern shown by  $\tilde{\varphi}$  is misclassified. Then  $\tilde{w}_{(1)} = \tilde{w}_{(0)} - t\tilde{\varphi}$ ;  $\eta$  was chosen 1 and  $t = -1$ . The decision boundary is always perpendicular to  $\mathbf{w}$  vector, see section 8.1.1. The other case of a misclassified  $\mathbf{x}_p \in \mathcal{C}_1$  is similar.

### 8.3.2 The Learning Procedure

The gradient descent solution (section 8.2.3) is used to find the weight vector:

$$\frac{\partial E_p}{\partial w_i} = \begin{cases} -t\tilde{\varphi}_i(\mathbf{x}_p) & \text{if } \mathbf{x}_p \in \mathcal{M} \\ 0 & \text{if } \mathbf{x}_p \text{ is correctly classified} \end{cases}$$

and then  $\nabla E_p = -t\tilde{\varphi}(\mathbf{x}_p)$  if  $\mathbf{x}_p \in \mathcal{M}$  or  $\nabla E_p = \hat{\mathbf{0}}$  otherwise.

The delta rule (8.16) becomes:

$$\Delta \tilde{\mathbf{w}} = \tilde{\mathbf{w}}_{(t+1)} - \tilde{\mathbf{w}}_{(t)} = \begin{cases} \eta t \tilde{\varphi}(\mathbf{x}_p) & \text{if } \mathbf{x}_p \in \mathcal{M} \\ \hat{\mathbf{0}} & \text{if } \mathbf{x}_p \text{ is correctly classified} \end{cases} \quad (8.18)$$

i.e. all training vectors are tested: if the  $\mathbf{x}_p$  is *correctly classified* then  $\mathbf{w}$  is left *unchanged*, otherwise it is “adapted” and the process is repeated until all vectors from the training set are classified correctly. See figure 8.8.

### 8.3.3 Convergence of Learning

The error function (8.17) decreases by using the learning rule (8.18).

*Proof.* The terms from  $E$  (8.17), after one learning step using (8.18), are:

$$-t\tilde{\mathbf{w}}_{(t+1)}^T \tilde{\varphi} = -t\tilde{\mathbf{w}}_{(t)}^T \tilde{\varphi} - \eta t^2 \|\tilde{\varphi}\|^2 < -\tilde{\mathbf{w}}_{(t)}^T \tilde{\varphi}$$

where  $\|\tilde{\varphi}\|^2 = \tilde{\varphi}^T \tilde{\varphi}$ ; then  $E_{(t+1)} < E_{(t)}$ , i.e.  $E$  decreases.  $\square$

❖  $\hat{w}$

Let consider a linearly separable problem. Then it exists a solution  $\hat{w}$  such that:

$$t_p \hat{w}^T \tilde{\varphi}(x_p) > 0 \quad , \quad p = \overline{1, P}$$

The process of updating  $w$ , using the delta rule (8.18) is convergent.

*Proof.* Let consider that the initial vector — in the above learning procedure — is chosen as  $\tilde{w}_{(0)} = \hat{0}$  and let the learning parameter be  $\eta = 1$  (without any loss of generality); then:

$$\tilde{w}_{(t+1)} = \tilde{w}_{(t)} + t_p \tilde{\varphi}(x_p)$$

where  $x_p$  is a misclassified training vector. (see (8.18)). Then the weight vector may be written as:

$$\tilde{w}_{(t+1)} = \sum_{\ell} \tau_{\ell} t_{\ell} \tilde{\varphi}(x_{\ell})$$

❖  $\tau_{\ell}$

where  $\tau_{\ell}$  is the number of misclassification of  $x_{\ell}$  vector — 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 sum is done over the training cycle — the training set may be used several times in any order.

By multiplying with  $\hat{w}^T$  to the left:

$$\hat{w}^T \tilde{w} = \sum_{\ell} \tau_{\ell} t_{\ell} \hat{w}^T \tilde{\varphi}(x_{\ell}) \geq \left( \sum_{\ell} \tau_{\ell} t_{\ell} \right) \min_{\ell} \hat{w}^T \tilde{\varphi}(x_{\ell})$$

❖  $\tau$

such that the product is limited from below by a function linear in  $\tau = \sum_k \tau_k t_k$  — and thus  $\tilde{w}_{(t+1)}$  is limited from below as well ( $\hat{w}$  is constant).

On the other hand:

$$\|\tilde{w}_{(t+1)}\|^2 = \|\tilde{w}_{(t)}\|^2 + t_{\ell}^2 \|\tilde{\varphi}(x_{\ell})\|^2 + 2t_{\ell} \tilde{w}_{(t)}^T \tilde{\varphi}(x_{\ell}) \leq \|\tilde{w}_{(t)}\|^2 + \|\tilde{\varphi}(x_{\ell})\|^2$$

Therefore:

$$\Delta \|\tilde{w}\|^2 = \|\tilde{w}_{(t+1)}\|^2 - \|\tilde{w}_{(t)}\|^2 \leq \max_{\ell} \|\tilde{\varphi}(x_{\ell})\|^2$$

and then:

$$\|\tilde{w}_{(t+1)}\|^2 \leq \tau \max_{\ell} \|\tilde{\varphi}(x_{\ell})\|^2$$

i.e.  $\|\tilde{w}_{(t+1)}\|$  is limited from above by a function linear in  $\sqrt{\tau}$ .

Considering both limitations (below by  $\tau$  and above by  $\sqrt{\tau}$ ) it follows that no matter how large  $t$  is, i.e. no matter how many update steps are taken,  $\tau$  have to be limited (because  $\tau$  from below grows faster than  $\sqrt{\tau}$  from above, during training) and then it means that at some stage  $\tau_{\ell}$  becomes stationary for all  $\ell \in \{1, P\}$  — thus (because  $\hat{w}$  was presumed to exists) all training vectors becomes correctly classified.  $\square$

### Remarks:

- ➔ The learning algorithm is good at generalization as long as the training set is statistically significant.
- ➔ The perceptron may be successfully used only for linearly separable classes.

## 8.4 Fisher Linear Discriminant

### 8.4.1 Two Classes Case

A very simple way to reduce the dimensionality it to apply a linear projection, into a unidimensional space, of the form:

$$y = \mathbf{w}^T \mathbf{x} \quad (8.19)$$

where  $\mathbf{w}$  is the vector of parameters chosen such as to maximize separability.

Let consider two classes and a training set containing  $P_1$  vectors of class  $\mathcal{C}_1$  and  $P_2$  vectors of class  $\mathcal{C}_2$ . The mean vectors of class distribution are:

$$\mathbf{m}_1 = \frac{1}{P_1} \sum_{\mathbf{x}_p \in \mathcal{C}_1} \mathbf{x}_p \quad \text{and} \quad \mathbf{m}_2 = \frac{1}{P_2} \sum_{\mathbf{x}_p \in \mathcal{C}_2} \mathbf{x}_p$$

Then a natural choice for  $\mathbf{w}$  would be such that it will maximize the distance between the unidimensional projection of means, i.e.  $\mathbf{w}^T(\mathbf{m}_1 - \mathbf{m}_2)$  — on the other hand this distance may be arbitrary increased by increasing  $\mathbf{w}$ ; to avoid this a constraint on the size of  $\mathbf{w}$  should be imposed, e.g. a normalization:  $\|\mathbf{w}\|^2 = \mathbf{w}^T \mathbf{w} = 1$ .

The Lagrange multiplier method is applied (see mathematical appendix) the Lagrange function (using the normalization on  $\mathbf{w}$ ) is:

$$L(\mathbf{w}, \lambda) = \mathbf{w}^T(\mathbf{m}_1 - \mathbf{m}_2) + \lambda(\|\mathbf{w}\|^2 - 1)$$

and the required solution is found from the conditions:

$$\frac{\partial L}{\partial w_i} = m_{1i} - m_{2i} + 2\lambda w_i = 0 \quad \text{and} \quad \frac{\partial L}{\partial \lambda} = \sum_i w_i^2 - 1 = 0$$

which gives  $\mathbf{w} \propto \mathbf{m}_1 - \mathbf{m}_2$ . However this solution is not generally good because it considers only the relative positions of the distributions, not their form, e.g. for Gaussian distribution this means the matrix  $\Sigma$ . See figure 8.9 on the next page.

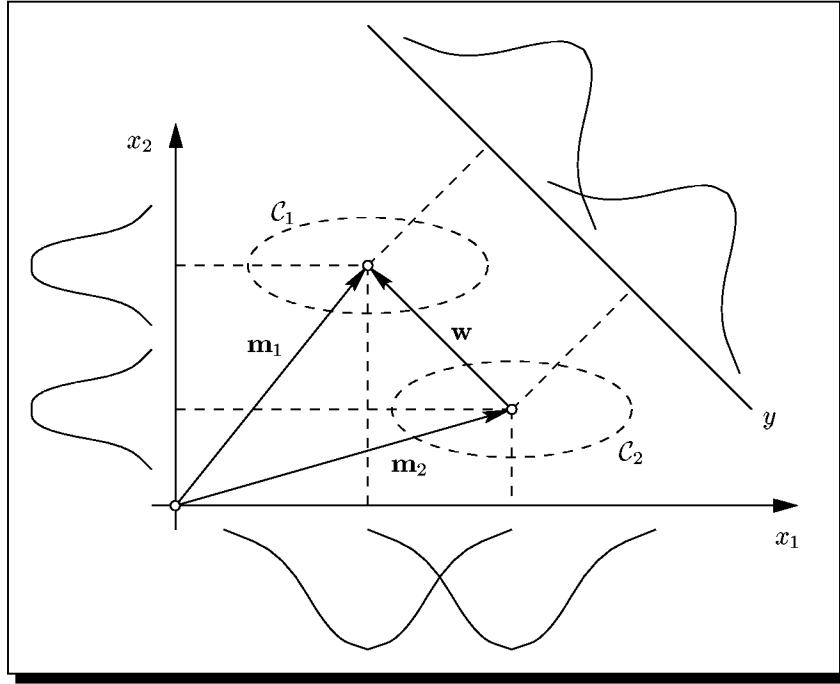
One way to measure the within class scatter, of the uni-dimensional projection of the data, is

$$\begin{aligned} s_k^2 &= \sum_{\mathbf{x}_p \in \mathcal{C}_k} [y(\mathbf{x}_p) - \mathbf{w}^T \mathbf{m}_k]^2 = \sum_{\mathbf{x}_p \in \mathcal{C}_k} [\mathbf{w}^T \mathbf{x}_p - \mathbf{w}^T \mathbf{m}_k] [\mathbf{x}_p^T \mathbf{w} - \mathbf{m}_k^T \mathbf{w}] \\ &= \mathbf{w}^T \left[ \sum_{\mathbf{x}_p \in \mathcal{C}_k} (\mathbf{x}_p - \mathbf{m}_k) (\mathbf{x}_p - \mathbf{m}_k)^T \right] \mathbf{w} \end{aligned}$$

and the total scatter, for two classes, would be  $s_{\text{total}}^2 = s_1^2 + s_2^2$ . Then a criteria to search for  $\mathbf{w}$  would be to minimize the scattering.

The Fisher technique takes the approach of maximizing the inverse of total scattering. The Fisher criterion

<sup>8.4</sup>See [Bis95] pp. 105–112.



**Figure 8.9:** The unidimensional reduction in  $y$  space using the Lagrange multiplier: Gaussian distribution, two classes, two dimensions. While the separation is better than a projection on  $x_1$  axis, it is worse than that one in the  $x_2$  axis because the  $\Sigma$  parameter was not considered.

Fisher criterion is defined as:

$$J(\mathbf{w}) = \frac{(\mathbf{w}^T \mathbf{m}_1 - \mathbf{w}^T \mathbf{m}_2)^2}{s_1^2 + s_2^2} = \frac{\mathbf{w}^T (\mathbf{m}_1 - \mathbf{m}_2)(\mathbf{m}_1 - \mathbf{m}_2)^T \mathbf{w}}{s_1^2 + s_2^2}$$

and it may be expressed also as:

$$J(\mathbf{w}) = \frac{\mathbf{w}^T S_b \mathbf{w}}{\mathbf{w}^T S_w \mathbf{w}}$$

❖  $S_b$

where  $S_b$  is named *between-class covariance matrix*:

$$S_b = (\mathbf{m}_1 - \mathbf{m}_2)(\mathbf{m}_1 - \mathbf{m}_2)^T \quad (8.20)$$

❖  $S_w$

and  $S_w$  is named *within-class covariance matrix*:

$$S_w = \sum_{\mathbf{x}_p \in \mathcal{C}_1} (\mathbf{x}_p - \mathbf{m}_1)(\mathbf{x}_p - \mathbf{m}_1)^T + \sum_{\mathbf{x}_p \in \mathcal{C}_2} (\mathbf{x}_p - \mathbf{m}_2)(\mathbf{x}_p - \mathbf{m}_2)^T$$

The gradient of  $J$  with respect to  $\mathbf{w}$  is zero at the desired maximum:

$$\nabla J = \frac{(\mathbf{w} S_w \mathbf{w}^T) S_b \mathbf{w} - (\mathbf{w} S_b \mathbf{w}^T) S_w \mathbf{w}}{(\mathbf{w} S_w \mathbf{w}^T)^2} = 0$$

$$\Rightarrow (\mathbf{w} S_w \mathbf{w}^T) S_b \mathbf{w} = (\mathbf{w} S_b \mathbf{w}^T) S_w \mathbf{w} \quad (8.21)$$

From (8.20) it gives that:

$$S_b \mathbf{w} = (\mathbf{m}_1 - \mathbf{m}_2)(\mathbf{m}_1 - \mathbf{m}_2)^T \mathbf{w} \propto (\mathbf{m}_1 - \mathbf{m}_2) \quad (8.22)$$

Because only the *direction* of  $\mathbf{w}$  matters then any scalar terms may be dropped; replacing (8.22) into (8.21) gives:

$$\mathbf{w} \propto S_w^{-1}(\mathbf{m}_1 - \mathbf{m}_2)$$

known as the *Fisher discriminant*.

Fisher  
discriminant

### 8.4.2 Connections With The Least Squares Technique

For the particular case of transformation (8.19), the sum-of-squares error function (8.9) becomes:

$$E = \frac{1}{2} \sum_{p=1}^P (\mathbf{w}^T \mathbf{x}_p + w_0 - t_p)^2$$

The target values are chosen as follows:

$$t_i = \begin{cases} \frac{P}{P_1} & \text{if } \mathbf{x}_p \in \mathcal{C}_1 \\ -\frac{P}{P_2} & \text{if } \mathbf{x}_p \in \mathcal{C}_2 \end{cases} \quad (8.23)$$

where  $P_1$  is the number of training patterns in  $\mathcal{C}_1$  and similar for  $P_2$ , obviously  $P_1 + P_2 = P$ .  $\diamond P_1, P_2$

The minima of  $E$  with respect to  $\mathbf{w}$  and  $w_0$  is found by zeroing its derivatives:

$$\nabla E = \sum_{p=1}^P (\mathbf{w}^T \mathbf{x}_p + w_0 - t_p) \mathbf{x}_p = 0 \quad (8.24a)$$

$$\frac{\partial E}{\partial w_0} = \sum_{p=1}^P (\mathbf{w}^T \mathbf{x}_p + w_0 - t_p) = 0 \quad (8.24b)$$

The sum in (8.24b) may be split on two sums following the membership of  $\mathbf{x}_p$ :  $\sum_{\mathbf{x}_p \in \mathcal{C}_1}$  and  $\sum_{\mathbf{x}_p \in \mathcal{C}_2}$ ; from the particular choice (8.23) for  $t_p$  and because  $P_1 + P_2 = P$  then:

$$w_0 = -\mathbf{w}^T \mathbf{m} \quad \text{where} \quad \mathbf{m} = \frac{1}{P} \sum_{p=1}^P \mathbf{x}_p = \frac{P_1}{P} \mathbf{m}_1 + \frac{P_2}{P} \mathbf{m}_2 \quad (8.25)$$

i.e.  $\mathbf{m}$  represents the mean of  $\mathbf{x}$  over the whole training set.

$\diamond \mathbf{m}$

The sum from (8.24a) may be split in 4 terms — separate summation over each class,

replacing  $w_0$  from (8.25) and replacing  $t_p$  values from (8.23):

$$\begin{aligned}\nabla E = & \sum_{\mathbf{x}_p \in \mathcal{C}_1} \left( \mathbf{w}^T \mathbf{x}_p - \frac{P_1}{P} \mathbf{w}^T \mathbf{m}_1 - \frac{P}{P_1} \right) \mathbf{x}_p - \frac{P_2}{P} \mathbf{w}^T \mathbf{m}_2 \sum_{\mathbf{x}_p \in \mathcal{C}_1} \mathbf{x}_p \\ & + \sum_{\mathbf{x}_p \in \mathcal{C}_2} \left( \mathbf{w}^T \mathbf{x}_p - \frac{P_2}{P} \mathbf{w}^T \mathbf{m}_2 + \frac{P}{P_2} \right) \mathbf{x}_p - \frac{P_1}{P} \mathbf{w}^T \mathbf{m}_1 \sum_{\mathbf{x}_p \in \mathcal{C}_2} \mathbf{x}_p = 0\end{aligned}\quad (8.26)$$

This relation reduces to:

$$\left( S_w + \frac{P_1 P_2}{P} S_b \right) \mathbf{w} = P(\mathbf{m}_1 - \mathbf{m}_2)$$

*Proof.* Using the definition of  $\mathbf{m}_{1,2}$ , (8.26) becomes:

$$\begin{aligned}\sum_{\mathbf{x}_p \in \mathcal{C}_1} (\mathbf{w}^T \mathbf{x}_p) \mathbf{x}_p - \sum_{\mathbf{x}_p \in \mathcal{C}_1} \frac{P_1}{P} (\mathbf{w}^T \mathbf{m}_1) \mathbf{x}_p - \sum_{\mathbf{x}_p \in \mathcal{C}_1} \frac{P}{P_1} \mathbf{x}_p - \frac{P_2}{P} (\mathbf{w}^T \mathbf{m}_2) P_1 \mathbf{m}_1 \\ + \sum_{\mathbf{x}_p \in \mathcal{C}_2} (\mathbf{w}^T \mathbf{x}_p) \mathbf{x}_p - \sum_{\mathbf{x}_p \in \mathcal{C}_2} \frac{P_2}{P} (\mathbf{w}^T \mathbf{m}_2) \mathbf{x}_p + \sum_{\mathbf{x}_p \in \mathcal{C}_2} \frac{P}{P_2} \mathbf{x}_p - \frac{P_1}{P} (\mathbf{w}^T \mathbf{m}_1) P_2 \mathbf{m}_2 = 0\end{aligned}$$

As  $\mathbf{w}^T \mathbf{x}_p = \mathbf{x}_p^T \mathbf{w}$  and the same for other  $\mathbf{w}^T$  products:

$$\begin{aligned}\sum_{\mathbf{x}_p \in \mathcal{C}_1} \mathbf{x}_p (\mathbf{x}_p^T \mathbf{w}) - \sum_{\mathbf{x}_p \in \mathcal{C}_1} \frac{P_1}{P} \mathbf{x}_p (\mathbf{m}_1^T \mathbf{w}) - \sum_{\mathbf{x}_p \in \mathcal{C}_1} \frac{P}{P_1} \mathbf{x}_p - \frac{P_1 P_2}{P} \mathbf{m}_1 (\mathbf{m}_2^T \mathbf{w}) \\ + \sum_{\mathbf{x}_p \in \mathcal{C}_2} \mathbf{x}_p (\mathbf{x}_p^T \mathbf{w}) - \sum_{\mathbf{x}_p \in \mathcal{C}_2} \frac{P_2}{P} \mathbf{x}_p (\mathbf{m}_2^T \mathbf{w}) + \sum_{\mathbf{x}_p \in \mathcal{C}_2} \frac{P}{P_2} \mathbf{x}_p - \frac{P_1 P_2}{P} \mathbf{m}_2 (\mathbf{m}_1^T \mathbf{w}) = 0\end{aligned}$$

and using again the definitions of  $\mathbf{m}_{1,2}$ :

$$\begin{aligned}\sum_{\mathbf{x}_p \in \mathcal{C}_1} \mathbf{x}_p (\mathbf{x}_p^T \mathbf{w}) - \frac{P_1^2}{P} \mathbf{m}_1 (\mathbf{m}_1^T \mathbf{w}) - P \mathbf{m}_1 - \frac{P_1 P_2}{P} \mathbf{m}_1 (\mathbf{m}_2^T \mathbf{w}) \\ + \sum_{\mathbf{x}_p \in \mathcal{C}_2} \mathbf{x}_p (\mathbf{x}_p^T \mathbf{w}) - \frac{P_2^2}{P} \mathbf{m}_2 (\mathbf{m}_2^T \mathbf{w}) + P \mathbf{m}_2 - \frac{P_1 P_2}{P} \mathbf{m}_2 (\mathbf{m}_1^T \mathbf{w}) = 0\end{aligned}$$

As matrix multiplication is associative,  $\mathbf{w}$  is a common factor. A  $\frac{P_1 P_2}{P} \mathbf{m}_1 \mathbf{m}_1^T + \frac{P_1 P_2}{P} \mathbf{m}_2 \mathbf{m}_2^T$  is added and then subtracted to help form  $S_b$  (it's moved to the right of equality):

$$\begin{aligned}\left[ \sum_{\mathbf{x}_p \in \mathcal{C}_1} \mathbf{x}_p \mathbf{x}_p^T - \frac{P_1^2}{P} \mathbf{m}_1 \mathbf{m}_1^T - \frac{P_1 P_2}{P} \mathbf{m}_1 \mathbf{m}_1^T \right. \\ \left. + \sum_{\mathbf{x}_p \in \mathcal{C}_2} \mathbf{x}_p \mathbf{x}_p^T - \frac{P_2^2}{P} \mathbf{m}_2 \mathbf{m}_2^T - \frac{P_1 P_2}{P} \mathbf{m}_2 \mathbf{m}_2^T \right] \mathbf{w} = P(\mathbf{m}_1 - \mathbf{m}_2) - \frac{P_1 P_2}{P} S_b \mathbf{w}\end{aligned}$$

Terms 2, 3, 5, 6 reduces ( $P = P_1 + P_2$ ) to give:

$$\left[ \sum_{\mathbf{x}_p \in \mathcal{C}_1} \mathbf{x}_p \mathbf{x}_p^T - P_1 \mathbf{m}_1 \mathbf{m}_1^T + \sum_{\mathbf{x}_p \in \mathcal{C}_2} \mathbf{x}_p \mathbf{x}_p^T - P_2 \mathbf{m}_2 \mathbf{m}_2^T \right] \mathbf{w} = P(\mathbf{m}_1 - \mathbf{m}_2) - \frac{P_1 P_2}{P} S_b \mathbf{w}$$

and expanding  $\mathbf{m}_{1,2}$  shows that the square parenthesis is  $S_w$ .  $\square$

Because  $S_b \mathbf{w} \propto (\mathbf{m}_1 - \mathbf{m}_2)$  (see (8.22)) and only the direction of  $\mathbf{w}$  counts then, by dropping the irrelevant constant factors, the Fisher discriminant is obtained:

$$\mathbf{w} \propto S_w^{-1} (\mathbf{m}_1 - \mathbf{m}_2)$$

### 8.4.3 Multiple Classes Case

It is assumed that the dimensionality of pattern vector space is greater than the number of classes, i.e.  $N > K$ .

A set of  $K$  transformations is considered

$$y_k(\mathbf{x}) = \mathbf{w}_k^T \mathbf{x} \quad , \quad j = \overline{1, K} \quad \Leftrightarrow \quad \mathbf{y}(\mathbf{x}) = W\mathbf{x} \quad (8.27)$$

where the matrix  $W$  is build using  $\mathbf{w}_k^T$  as rows.

❖  $W$

Let  $P_k$  the number of training vectors (from the whole set) being of class  $\mathbf{C}_k$ , and  $\mathbf{m}_k$  be the mean vector of that class:

❖  $P_k, \mathbf{m}_k$

$$\mathbf{m}_k = \frac{1}{P_k} \sum_{\mathbf{x}_p \in \mathcal{C}_k} \mathbf{x}_p \quad , \quad k = \overline{1, K} \quad \text{and} \quad \mathbf{m} = \frac{1}{P} \sum_{p=1}^P \mathbf{x}_p = \frac{1}{P} \sum_{k=1}^K P_k \mathbf{m}_k$$

where  $P = \sum_{k=1}^K P_k$  is the total number of training vectors, and  $\mathbf{m}$  is the mean over the whole training set.

❖  $P, \mathbf{m}$

The generalization of within-class covariance matrix (8.20) is easily performed as:

❖  $S_w$

$$S_w = \sum_{k=1}^K S_{wk} \quad \text{where} \quad S_{wk} = \sum_{\mathbf{x}_p \in \mathcal{C}_k} (\mathbf{x}_p - \mathbf{m}_k)(\mathbf{x}_p - \mathbf{m}_k)^T$$

The *total covariance matrix*  $S_t$  is

❖  $S_t$

$$S_t = \sum_{p=1}^P (\mathbf{x}_p - \mathbf{m})(\mathbf{x}_p - \mathbf{m})^T = \sum_{k=1}^K \sum_{\mathbf{x}_p \in \mathcal{C}_k} (\mathbf{x}_p - \mathbf{m})(\mathbf{x}_p - \mathbf{m})^T$$

and may be written as:

❖  $S_b$

$$S_t = S_w + S_b \quad \text{where} \quad S_b = \sum_{k=1}^K P_k (\mathbf{m}_k - \mathbf{m})(\mathbf{m}_k - \mathbf{m})^T$$

where  $S_t, S_w$  and  $S_b$  are defined in  $X$  pattern space.

*Proof.*

$$\begin{aligned} S_t &= \sum_{k=1}^K \left[ \sum_{\mathbf{x}_p \in \mathcal{C}_k} \mathbf{x}_p \mathbf{x}_p^T - \left( \sum_{\mathbf{x}_p \in \mathcal{C}_k} \mathbf{x}_p \right) \mathbf{m}^T - \mathbf{m} \sum_{\mathbf{x}_p \in \mathcal{C}_k} \mathbf{x}_p^T + P_k \mathbf{m} \mathbf{m}^T \right] \\ &= \sum_{k=1}^K \left[ \sum_{\mathbf{x}_p \in \mathcal{C}_k} \mathbf{x}_p \mathbf{x}_p^T - P_k \mathbf{m}_k \mathbf{m}_k^T - \mathbf{m} P_k \mathbf{m}_k^T + P_k \mathbf{m} \mathbf{m}^T \right] \end{aligned}$$

By adding, and then subtracting, a term of the form  $P_k \mathbf{m}_k \mathbf{m}_k^T$ , the  $S_b$  is formed and then:

$$S_t = \sum_{k=1}^K \left[ \sum_{\mathbf{x}_p \in \mathcal{C}_k} \mathbf{x}_p \mathbf{x}_p^T - P_k \mathbf{m}_k \mathbf{m}_k^T \right] + S_b = S_w + S_b \quad \square$$

Similar matrices may be expresses in the  $Y$  output space.

- ❖  $\mu_k, \mu$  Let  $\mu_k$  and  $\mu$  be the mean over class  $\mathcal{C}_k$  and, respectively, over all training set of output  $\mathbf{y}(\mathbf{x}_p)$ :

$$\mu_k = \frac{1}{P_k} \sum_{\mathbf{x}_p \in \mathcal{C}_k} \mathbf{y}(\mathbf{x}_p) \quad , \quad k = \overline{1, K} \quad \text{and} \quad \mu = \frac{1}{P} \sum_{p=1}^P \mathbf{y}(\mathbf{x}_p) = \frac{1}{P} \sum_{k=1}^K P_k \mu_k$$

The covariance matrices in  $Y$  space are:

$$S_{(Y)w} = \sum_{k=1}^K \sum_{\mathbf{x}_p \in \mathcal{C}_k} [\mathbf{y}(\mathbf{x}_p) - \mu_k][\mathbf{y}(\mathbf{x}_p) - \mu_k]^T$$

$$S_{(Y)b} = \sum_{k=1}^K P_k [\mu_k - \mu][\mu_k - \mu]^T$$

One possibility<sup>4</sup> for the Fisher criterion is:

$$J(W) = \text{Tr}(S_{(Y)w}^{-1} S_{(Y)b}) = \text{Tr}(W S_w W^T W S_b W^T)$$

(considering (8.27)).

### Remarks:

- ➡  $S_b$  is a sum of  $K$  matrices, each of rank 1 — because it represents a product of 2 vectors. Also there is a relation between all  $\mathbf{m}_k$  given by the definition of  $\mathbf{m}$ . Then the rank of  $S_b$  is  $K - 1$  at most, and, consequently it have only  $K - 1$  eigenvectors/values.

By the means of Fisher criterion it is possible to find only  $K - 1$  transformations.

---

<sup>4</sup>There are several choices.

## CHAPTER 9

# Multi Layer Neural Networks

### ► 9.1 Feed-Forward Networks

Feedforward networks do not contain feedback connections. Also between the input units  $x_i$  and the output units  $y_k$  there are (usually) some hidden units  $z_j$ . Let  $N$  be the dimension (number of neurons) of input layer,  $H$  the dimension of hidden layer and  $K$  the one of output layer. See figure 9.1 on the following page.

❖  $N, H, K$

Assuming that the weights for the hidden layer are  $\{w_{(1)ji}\}_{\substack{j=1, H \\ i=0, N}}$  (characterizing the connection to  $z_j$  from  $x_i$ ) and the activation function is  $f_1$  then the output of the hidden neurons is:

❖  $w_{(1)ji}, f_1$

$$z_j = f_1 \left( \sum_{i=0}^N w_{(1)ji} x_i \right)$$

where  $x_0 = 1$  at all times —  $w_{j0}$  being the bias (characteristic to hidden neuron  $j$ ). Note that there are no connections from  $x_i$  to  $z_0$ , they would be irrelevant as  $z_0$  represents the bias and its output is 1 at all times (regardless of its input).

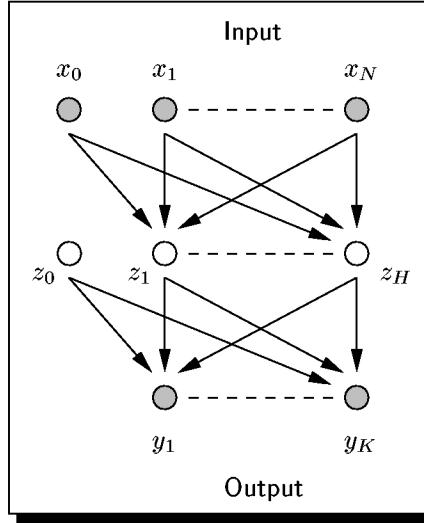
On similar grounds, let  $\{w_{(2)kj}\}_{\substack{k=1, K \\ j=0, H}}$  be the weights of the output layer, and  $f_2$  its activation function. Then the output of the output neuron is:

❖  $w_{(2)ki}, f_2$

$$y_k = f_2 \left( \sum_{j=0}^H w_{(2)kj} z_j \right) = f_2 \left( \sum_{j=0}^H w_{(2)kj} f_1 \left( \sum_{i=0}^N w_{(1)ji} x_i \right) \right)$$

---

<sup>9.1</sup>See [Bis95] pp. 116–121.



**Figure 9.1:** The feedforward network architecture. Between the input units  $x_i$  and output units  $y_k$  there are some hidden units  $z_j$ .  $x_0$  and  $z_0$  represents the bias.

The above formula may be easily generalized to a network with multiple hidden layers.



### Remarks:

- While the network depicted in figure 9.1 appears to have 3 layers, there are in fact only 2 *processing* layers: hidden and output. So this network will be said to have 2 layers.
- The input layer  $x$  plays the role of distributing the inputs to all neurons in subsequent layer, i.e. it plays the role of a *sensory* layer.
- In general a network is of feedforward type if there is a possibility to label all neurons (input, hidden and output) such that any neuron will receive inputs only from those with lower number label.
- By the above definition, more general neural networks may be build (than the one from figure 9.1).

## 9.2 Threshold Neurons

A threshold neuron have the activation function of the form:

$$f(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{if } a < 0 \end{cases} \quad (9.1)$$

and the  $a = 0$  value may be assigned to either case.

---

<sup>9.2</sup>See [Bis95] pp. 121–126.

### 9.2.1 Binary Vectors

Let consider the case of binary pattern vectors, i.e.  $x_i \in \{0, 1\}$ ,  $\forall i$ . On the other end, let the outputs be also binary. One output neuron will be considered, the discussion being easily generalisable to multiple output neurons.

Then the problem is to model a Boolean function  $f_B : \{0, 1\}^N \rightarrow \{0, 1\}$ . The total number of possible inputs is  $2^N$ . The function  $f_B$  is totally defined once the output value is given for all input combinations.  $\diamond f_B$

Then the network may be build as follows:

- The number of hidden neurons is equal to the number of input patterns which should result in an output equal to 1.
- The activation function for hidden neurons is defined as:

$$f_1(a) = \begin{cases} 1 & \text{if } a > 0 \\ 0 & \text{if } a \leq 0 \end{cases}$$

Each hidden neuron is set to be activated just by one pattern: For that pattern the weights are set up:  $w_{ji} = \begin{cases} 1 & \text{if } x_i = 1 \\ -1 & \text{if } x_i = 0 \end{cases}$  and  $w_{j0} = 1 - n_x$ ; where  $n_x = \sum_{i=1}^N x_i$ , i.e. is equal with the number of "ones" into the  $x$  pattern vector.

Then the total input to a hidden neuron is  $1 \cdot n_x + 1 \cdot (1 - n_x) = 1$  and then the output of the hidden neuron is 1 when presented with the "learned" pattern vector.

The total input is at most  $(n_x - 1) + (1 - n_x) = 0$  when presented with another pattern vector (one  $x_i$  component changed from 1 to 0 or vice-versa); such that the output will be 0.

- The activation function for the output neuron is:

$$f_2(a) = \begin{cases} 1 & \text{if } a \geq 0 \\ 0 & \text{if } a < 0 \end{cases}$$

The weights to the output neuron are set to 1. The bias  $w_{(2)0}$  is set to  $-1$  such that when a pattern for which the output should be 1 is presented to the net, the total input in  $y$  is 0; otherwise is  $-1$  and thus the correct output is ensured at all times.

A vectorial output function may be split into components and each component may be assigned to a output neuron.

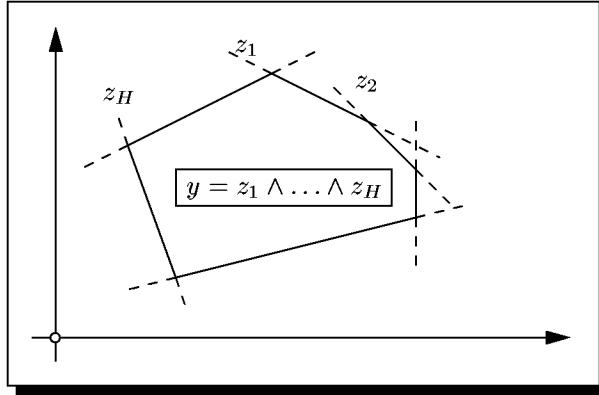


#### Remarks:

- ➔ While not very useful by itself (it does not have a generalization capability) the above architecture illustrate the possible importance of singular neuron "firing" — used extensively in CPN and ART neural networks.

### 9.2.2 Continuous Vectors

The two propositions below assume a 2 class problem (either  $x \in \mathcal{C}_1$  or  $x \in \mathcal{C}_2$ ) and thus one output neuron is enough for classification. The solution is easily extensible to multi-class



**Figure 9.2:** In a two layer network the hidden layer may represent the hyperplane decision boundaries and then the output layer may do a logical AND to establish if the input vector is within the decision region. Thus any convex decision area may be represented by a two layer network.

problems by assigning each output neuron to a class.

**Proposition 9.2.1.** A two-layer neural network may have arbitrary convex decision boundary.

*Proof.* A single layer neural network (with one output) have a decision boundary which is a hyperplane. Let the hidden layer represent the hyperplanes (see chapter “Single Layer Neural Networks”).

Then the output layer may perform an logical AND between the outputs of the hidden layer to decide if the pattern vector is inside the decision region or not — each output neuron representing a class. See figure 9.2.  $\square$

**Proposition 9.2.2.** A 3-layer neural network may have arbitrary decision boundary (it may be non-convex and/or disjoint).

*Proof.* The pattern space is divided into sufficiently small hypercubes such that the decision region may be approximated using them (i.e. each hypercube will be included either in the  $C_1$  decision region or in that of  $C_2$ 's). The decision area may be approximated with arbitrary precision by making the hypercubes smaller.

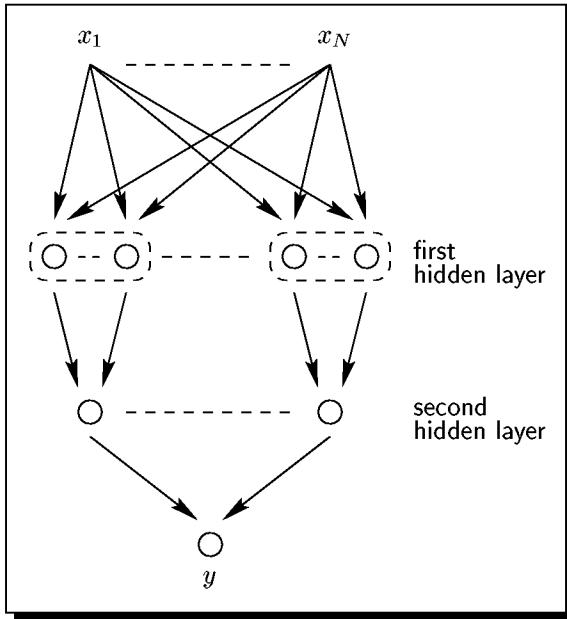
The neural network is built as follows: The first hidden layer contains a group of  $2N$  neurons for each hypercube of the same one class (2 hyperplanes for each dimension to define the hypercube). The second hidden layer contains  $N$  neurons who receive the input from the corresponding group of  $2N$  neurons from the first hidden layer. The output layer receive its inputs from all  $N$  neurons from the second hidden layer. The architecture of the network is depicted in figure 9.3 on the next page.

By the same method as described in the previous proposition a neuron from the second layer may decide if the input pattern vector is inside the hypercube it represents.

An logical OR on the output layer, between the outputs of the second hidden layer, decides if the pattern vector belongs to any of hypercubes represented by the first layer and thus to the class selected; if not then the input vector belongs to the other class.  $\square$

### Remarks:

- ➔ The network architecture described in proof of proposition 9.2.2 have the disadvantage that require large hidden layers.



**Figure 9.3:** The 3 layer neural network architecture for arbitrary decision areas.

## 9.3 Sigmoidal Neurons

The possible sigmoidal activation functions are:

$$y = f(a) = \frac{1}{1 + e^{-ca}} \quad \text{and} \quad y = f(a) = \tanh(a) = \frac{e^{ca} - e^{-ca}}{e^{ca} + e^{-ca}}$$

where  $c = \text{const.}$

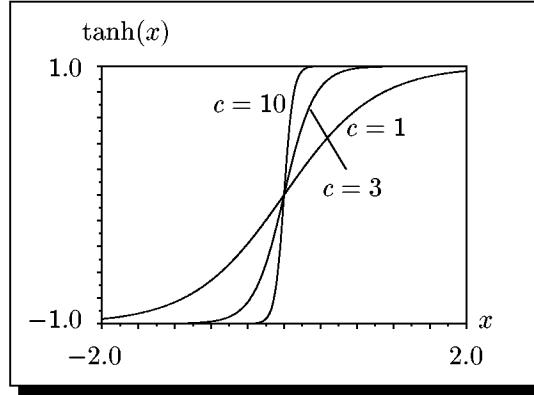
Because  $\frac{\tanh(a/2)+1}{2} = \frac{1}{1+e^{-ca}}$  then using the tanh function instead of the logistic one is equivalent to apply a linear transformation  $\tilde{a} = a/2$  before and (again a linear transformation)  $y = \frac{1}{2}\tilde{y} + 1$  after the processing on neural layer (this is equivalent in a linear transformation of weights and biases).

The tanh function have the advantage of being symmetrical with respect to the origin. See figure 9.4 on the following page.

### Remarks:

- The output of a logistic neuron is limited to the interval  $[0, 1]$ . However the logistic function is easily inversable and then the inverse  $f^{-1}(y) = \frac{1}{c} \ln \frac{y}{1-y}$ .
- In the vicinity of the origin the logistic function is almost linear and thus it can approximate a linear neuron.

<sup>9.3</sup>See [Bis95] pp. 126–132.



**Figure 9.4:** The graph of  $\tanh$  function for various  $c$  constants.

### 9.3.1 Three Layer Networks

**Proposition 9.3.1.** *A three layer network may approximate with any accuracy a smooth function (mapping)  $X \rightarrow Y$ .*

*Proof.* The logistic activation function is:

$$y = \frac{1}{1 + \exp(-\mathbf{w}^T \mathbf{x} - w_0)}$$

and is represents the output of a first-layer neuron. See figure 9.5-a .

By making linear combinations, i.e. when entering the second neuronal layer, it is possible to get a function with an absolute maximum. See figure 9.5-b and 9.5-c.

By applying the sigmoidal 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 9.5-d.

The third layer may combine the localized outputs of the second layer to perform the approximation of any smooth function — it should have a linear activation function.  $\square$

### 9.3.2 Two Layer Networks

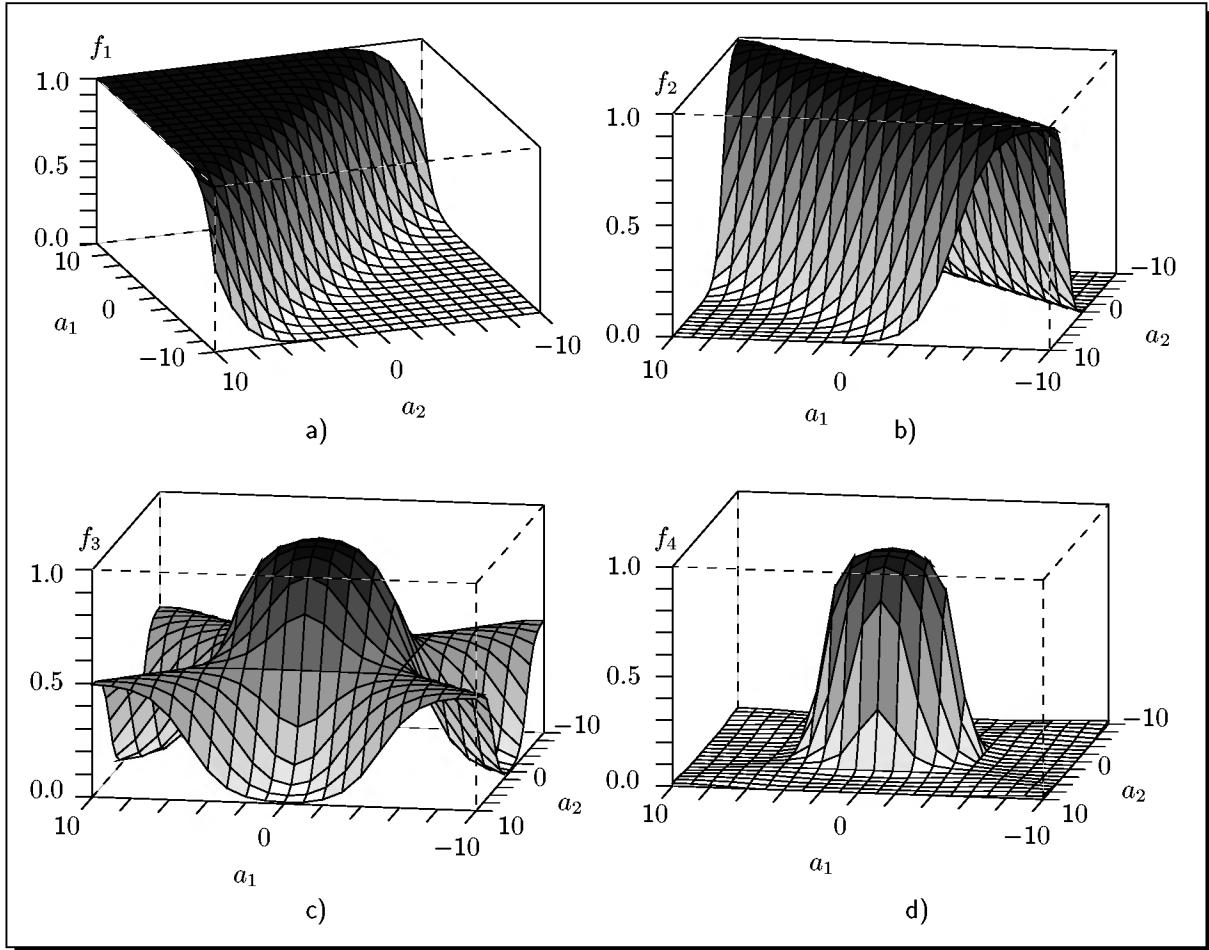
**Proposition 9.3.2.** *A two layer neuronal network can approximate, arbitrary well, any function (mapping)  $X \rightarrow Y$ , provided that  $X$  and  $Y$  spaces are finite-dimensional and there are enough hidden neurons.*

*Proof.* Any function may be decompose into a Fourier series:

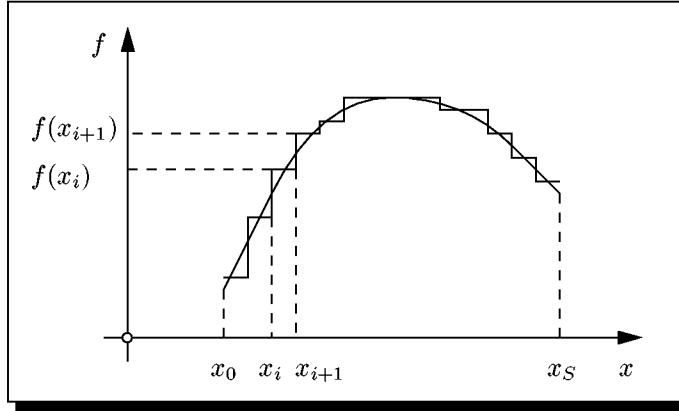
$$\begin{aligned} y(x_1, \dots, x_N) &= \sum_{i_1} c_{i_1}(x_2, \dots, x_N) \cos(i_1 x_1) = \dots \\ &= \sum_{i_1} \dots \sum_{i_N} C_{i_1 \dots i_N} \prod_{\ell=1}^N \cos(i_\ell x_\ell) \end{aligned}$$

(by developing in series all  $c$  parameters).

Any product of 2 cosines may be transformed into a sum, using the formula  $\cos \alpha \cos \beta = \frac{1}{2} \cos(\alpha + \beta) + \frac{1}{2} \cos(\alpha - \beta)$ . Then by applying this procedure  $N - 1$  times, the product from the equation above may be changed to a sum of cosines (the values of the angles and the constants don't have to be specified for the proof).



**Figure 9.5:** Two dimensional space. Figure a) shows the sigmoidal function  $f_1 = \frac{1}{1+e^{-a_1-a_2}}$ . Figure b) shows the linear combination  $f_2 = \frac{1}{1+e^{-a_1-a_2-5}} - \frac{1}{1+e^{-a_1-a_2+5}}$  — they are displaced relatively one each other. Figure c) shows a linear combination of 4 sigmoidal functions  $f_3 = \frac{0.5}{1+e^{-a_1-a_2-5}} - \frac{0.5}{1+e^{-a_1-a_2+5}} + \frac{0.5}{1+e^{-a_1+a_2-5}} - \frac{0.5}{1+e^{-a_1+a_2+5}}$  — the second pair is rotated by  $\pi/2$ . Figure d) shows the output after applying the sigmoidal function again  $f_4 = \frac{1}{1+e^{14f_3+1.6}}$  — only the central maximum remains.



**Figure 9.6:** The approximation of a function by  $S$  steps.

Using the Heaviside step function:

$$H(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

any function may be approximated as:

$$f(x) \simeq f(x_0) + \sum_{i=1}^S [f(x_i) - f(x_{i-1})] H(x - x_i)$$

❖  $S$

where  $S$  is the number of steps by which the function was approximated. See figure 9.6.

Then the network is build/perform as follows:

- The first layer have threshold activation functions and calculate the values of cosine functions.
- The second layer performs the linear combination from the Fourier series development. □

## → 9.4 Weight-Space Symmetry

Considering the tanh activation function then by changing the sign on all weights and the bias the output of neuron have reverted sign (tanh is symmetrical with respect to origin, see also figure 9.4 on page 158).

If the network have two layers and the weights and biases on the second layer have also the signs reverted then the final output of network remains unchanged.

Then the two sets of weights and biases leads to same result, i.e. there is a symmetry of the weights towards origin.

Assuming a 2-layer network with  $H$  hidden neurons then the total number of weights and biases sets which gives the same final result is  $2^H$

Also, if all weights and the bias are interchanged between two neurons on one layer then the output of the next layer remains unchanged (assuming that the layers are fully interconnected).

---

<sup>9.4</sup>See [Bis95] pg. 133.

There are  $H!$  such possible combinations on the hidden layer.

Finally there are  $H!2^H$  sets of weights which gives the same output on a 2-layer network. On more complex networks there may be even more symmetries.

*The symmetry in weights leads directly to a symmetry in error function and then the error will have several equivalent minima. Eventually this means that the minima point of error may be much closer than it looks at first sight, regardless of the starting point, usually randomly selected.*

## 9.5 Higher-Order Neuronal Networks

The neurons studied so far performed a linear combination of their inputs before applying the activation function:

$$\{\text{total input}\}_j = a_j = \sum_{i=0}^N w_{ji}x_i \quad \text{then} \quad \{\text{output}\}_j = y_j = f(a_j)$$

(or in matrix notation:  $\mathbf{a} = W\mathbf{x}$  and  $\mathbf{y} = f(\mathbf{a})$ ).

It is possible to design a neuron which performs a higher-degree combination of its inputs, e.g. a second-order:

$$\{\text{total input}\}_j = w_{j0} + \sum_{i=1}^N w_{ji}x_i + \sum_{i=1}^N \sum_{\ell=1}^N w_{jil}x_ix_\ell$$



### Remarks:

- The main difficulty in dealing with such neurons consists in the tremendous increase in the number of  $W$  parameters.

A first-ordered neuron will have  $N + 1$  parameters while a second order neuron will have  $N^2 + N + 1$  parameters and for  $N \gg 1 \Rightarrow N^2 \gg N$ .

On the other hand higher-order neurons may be built to be invariant to some transformations of the pattern space, e.g. translations, rotation and scaling. This property may make them usable into the *first* layer of network.

## 9.6 Backpropagation Algorithm

### 9.6.1 Error Backpropagation

It is assumed that the error function may be written as a sum over all training vector patterns

$$E = \sum_{p=1}^P E_p$$

---

<sup>9.5</sup> See [Bis95] pp. 133–135.

<sup>9.6</sup> See [Bis95] pp. 140–148.

and also that the error function  $E_p = E_p(\mathbf{y})$  is differentiable with respect to the output variables. Then just one vector pattern is considered at a time.

The output of network is calculated by forward propagation from:

$$a_j = \sum_i w_{ji} z_i \quad \text{and} \quad z_j = f(a_j) \quad (9.2)$$

$$\mathbf{a} = W(j,:) \mathbf{z} \quad \text{and} \quad \mathbf{z} = f(\mathbf{a})$$

❖  $a_j$  where  $a_j$  is the weighted sum of all inputs entering neuron  $z_j$ ,  $z_i$  being here any neuron, hidden or output.

Then  $E_p$  also depends on  $w_{ji}$  through  $a_j$  and then the derivative of  $E_p$  with respect to  $w_{ji}$  is

$$\frac{\partial E_p}{\partial w_{ji}} = \frac{\partial E_p}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \frac{\partial E_p}{\partial a_j} z_i = \delta_j z_i \Rightarrow \quad (9.3)$$

$$\nabla E_p = \nabla_{\mathbf{a}} E_p \mathbf{z}^T = \boldsymbol{\delta} \mathbf{z}^T$$

❖  $\delta_j$ ,  $\boldsymbol{\delta}$  where  $\frac{\partial E_p}{\partial a_j} \equiv \delta_j$  is named *error* — it's a factor determinant in weight adjusting. (as shown below);  $\nabla_{\mathbf{a}} E_p \equiv \boldsymbol{\delta}$ , note that here  $\nabla E_p$  represents just the part linked to  $W$  from the whole error gradient

$\mathbf{z}$  for all layers is found through a forward propagation through the network.  $\boldsymbol{\delta}$  is found by backpropagation (from output to input) as follows:

- For the output layer:  $E_p = E_p(f(\mathbf{a}))$  and then:

$$\delta_{\text{out},k} \equiv \frac{\partial E_p}{\partial a_k} = \frac{\partial E_p}{\partial y_k} \frac{df(a_k)}{da_k} = \frac{\partial E_p}{\partial y_k} f'(a_k) \quad (9.4)$$

$$\boldsymbol{\delta}_{\text{out}} = \nabla_{\mathbf{y}} E_p \odot f'(\mathbf{a})$$

❖  $f'$  where  $f'$  is the total derivative of activation function  $f$ .

- For other layers: neuron  $z_j$  affects the error  $E_p$  through all other neurons to which it sends its output:

$$\delta_j = \frac{\partial E_p}{\partial a_j} = \sum_{\ell} \frac{\partial E_p}{\partial a_{\ell}} \frac{\partial a_{\ell}}{\partial a_j} = \sum_{\ell} \delta_{\text{next},\ell} \frac{\partial a_{\ell}}{\partial a_j} \quad (9.5)$$

where the sum is done over all neurons to which  $z_j$  send connections ( $\delta_{\text{next},\ell} \equiv \frac{\partial E_p}{\partial a_{\ell}}$ ) and from the expression of  $a_{\ell}$  and definition of  $\delta_{\ell}$  finally the *back-propagation formula*:

$$\delta_j = f'(a_j) \sum_{\ell} w_{\ell j} \delta_{\text{next},\ell} \quad (9.6)$$

by the means of which the derivatives may be calculated backwards — from the output layer to the input layer. In matrix notation:

$$\boldsymbol{\delta} = f'(\mathbf{a}) \odot (W^T \boldsymbol{\delta}_{\text{next}})$$

By knowing the derivatives of  $E$  with respect to the weights, the  $W$  parameters may be adjusted in the direction of minimizing the error, using the delta rule.

### 9.6.2 Application: Sigmoidal Neurons and Sum-of-squares Error

Let consider a network having logistic activation function for its neurons and the sum-of-squares error function.

Then the derivative of the activation function is:

$$f(x) = \frac{1}{1 + e^{-x}} \Rightarrow \frac{df}{dx} = f(x)(1 - f(x)) \quad (9.7)$$

and the sum-of-squares error function for pattern vector  $\mathbf{x}_p$  is:

$$E_p = \frac{1}{2} \sum_{k=1}^K [y_k(\mathbf{x}_p) - t_{pk}]^2 = \frac{1}{2} [\mathbf{y}(\mathbf{x}_p) - \mathbf{t}_p]^T [\mathbf{y}(\mathbf{x}_p) - \mathbf{t}_p] \quad (9.8)$$

(it is assumed that  $y_k$  and  $t_{pk}$  are the components of the respective vectors  $\mathbf{y}$  and  $\mathbf{t}_p$ , when the network is presented with the pattern vector  $\mathbf{x}_p$ ).

From (9.8), (9.4) and (9.7), for output layer:

$$\delta_{\text{out}} = \mathbf{y}(\mathbf{x}_p) \odot [\hat{\mathbf{1}} - \mathbf{y}(\mathbf{x}_p)] \odot [\mathbf{y}(\mathbf{x}_p) - \mathbf{t}_p]$$

and similar, for the hidden layers:

$$\delta = \mathbf{z} \odot [\hat{\mathbf{1}} - \mathbf{z}] \odot (W^T \delta_{\text{next}})$$

where the  $\delta$  errors are calculated backwards starting with the hidden layer closest to the output and ending with the first hidden layer.

The error gradient (the part linked to  $W$ )  $\nabla E_p$  is:

$$\nabla E_p = \delta \mathbf{z}^T$$

To minimize the error, the weights have to be changed in direction contrary to that pointed by the gradient vector, i.e. the amount will be:

$$\Delta W \propto -\nabla E_p = -\eta \nabla E_p$$

where  $\eta$  governs the overall speed of weight adaptation, i.e. the speed of learning, and thus is named *learning constant*.  $\delta_i$  is a determining factor in weight change and thus is named *error*. The above equation represents the delta rule.



#### Remarks:

► The choice of order regarding training pattern vectors is optional:

- Consider one at a time (including random selection for the next one).
- Consider all together and then change the weights with the sum of all individual weight adjustments

$$\Delta W = -\eta \sum_{p=1}^P \nabla E_p$$

This represents the *batch* backpropagation.

❖  $W, \mathcal{O}$

- Any pattern vector may be selected multiple times for training/weight adjusting.
  - Any of the above in any order.
- $\eta$  have to be sufficiently large to avoid the trap of local minima of  $E$  but on the other hand it have to be sufficiently small such that the “absolute” minima will not be jumped over.
- In numerical simulation the most computational intensive terms are the matrix operations found in (9.6). Then the computational time is proportional with the number of weights  $W$ :  $\mathcal{O}(W)$  — where  $\mathcal{O}$  is a linear function in  $W$  — the number of neurons being usually much lower (unless there is a very sparse network). Because the algorithm require in fact one forward and one backward propagation trough the net the dependency is  $\mathcal{O}(2W)$ .

On the other hand to explicitly calculate the derivatives of  $E_p$  would require one pass trough the net for each one and then the computational time will be proportional with  $\mathcal{O}(W^2)$ . The importance of backpropagation algorithm resides in the fact that reduces computational time from  $\mathcal{O}(W^2)$  to  $\mathcal{O}(W)$ . However the classical way of calculating  $\nabla E_p$  by perturbing each  $w_{ji}$  by a small amount  $\varepsilon \gtrsim 0$  is still good for checking the correctness of a digital implementation on a particular system:

$$\frac{\partial E_p}{\partial w_{ji}} \simeq \frac{E_p(w_{ji} + \varepsilon) - E_p(w_{ji} - \varepsilon)}{2\varepsilon}$$

Another approach would be to calculate the derivatives:

$$\frac{\partial E_p}{\partial a_j} \simeq \frac{E_p(a_j + \varepsilon) - E_p(a_j - \varepsilon)}{2\varepsilon}$$

This approach still needs two steps for each neuron and then the computational time is proportional with  $\mathcal{O}(2MW)$ , where assuming  $M$  is the total number of neurons.

Note that because the derivative is calculated with the aid of two values centered around  $w_{ji}$ , respectively  $a_j$ , the terms  $\mathcal{O}(\varepsilon)$  are canceled (the bigger non-zero terms neglected in the above approximations are  $\mathcal{O}(\varepsilon^2)$ ).

## 9.7 Jacobian Matrix

The following matrix:

$$J \equiv \left\{ \frac{\partial y_k}{\partial x_i} \right\}_{\substack{k=1, K \\ i=1, N}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_K}{\partial x_1} & \dots & \frac{\partial y_K}{\partial x_N} \end{pmatrix}$$

Jacobian

is named the *Jacobian* matrix and it provides a measure of the local sensitivity of the network

---

<sup>9.7</sup> See [Bis95] pp. 148–150.

output to a change in the inputs, i.e. for small perturbation in input vector  $\Delta \mathbf{x} \ll \mathbf{x}$  the perturbation in the output vector is:

$$\Delta \mathbf{y} \simeq J \Delta \mathbf{x}$$

Considering the first layer to which inputs *send* connections (i.e. the first hidden layer):

$$J_{ki} = \frac{\partial y_k}{\partial x_i} = \sum_{\ell} \frac{\partial y_k}{\partial a_{\ell}} \frac{\partial a_{\ell}}{\partial x_i} = \sum_{\ell} \frac{\partial y_k}{\partial a_{\ell}} w_{\ell i} \quad (9.9)$$

(the sum is made over all neurons to which inputs send connections). To use a matrix notation, the following matrix is defined:

❖  $\nabla_{\mathbf{a}} \mathbf{y}$

$$\nabla_{\mathbf{a}} \mathbf{y} \equiv \left\{ \frac{\partial y_k}{\partial a_{\ell}} \right\}$$

where  $\mathbf{a}$  refers to a particular layer.

The derivatives  $\frac{\partial y_k}{\partial a_{\ell}}$  are found by a procedure similar to the backpropagation algorithm.

A perturbation/change in  $a_{\ell}$  is propagated through all neurons to which neuron  $\ell$  send connections (its output), then:

$$\frac{\partial y_k}{\partial a_{\ell}} = \sum_q \frac{\partial y_k}{\partial a_q} \frac{\partial a_q}{\partial a_{\ell}}$$

and, because  $a_q = \sum_s w_{qs} z_s = \sum_s w_{qs} f(a_s)$  then:

$$\frac{\partial y_k}{\partial a_{\ell}} = f'(a_{\ell}) \sum_q \frac{\partial y_k}{\partial a_q} w_{q\ell} \quad (9.10)$$

i.e. the derivatives  $\frac{\partial y_k}{\partial a_{\ell}}$  may be evaluated in terms of the same derivatives of the next layer.

- For the output layer:

$$\frac{\partial y_k}{\partial a_{\ell}} = \delta_{k\ell} \frac{df(a_k)}{da_k}$$

where  $a_{\ell}$  here are those received by the output neurons ( $\delta_{k\ell}$  is the Kronecker symbol), i.e. all partial derivatives are 0 except  $\frac{\partial y_k}{\partial a_k}$  and the matrix  $\nabla_{\mathbf{a}} \mathbf{y}$  have just one non-zero element per each row/column.

- For other layers: the derivatives are calculated backward using (9.10), then:

$$\nabla_{\mathbf{a}} \mathbf{y} = [\widehat{\mathbf{1}} f'(\mathbf{a}^T)] \odot [\nabla_{\mathbf{a}_{\text{next}}} \mathbf{y} W]$$

When the first layer is reached then the Jacobian is found from (9.9).



### Remarks:

- The same remarks as for backpropagation algorithm regarding the computing time (see section 9.9) applies here.

## 9.8 Hessian Tensor

The elements of the Hessian tensor are the second derivatives of the error function with respect to weights:

$$H = \left\{ \frac{\partial^2 E}{\partial w_{ji} \partial w_{kl}} \right\}_{jikl}$$

### Remarks:

- The Hessian is used in several non-linear learning algorithms by analyzing the surface  $E = E(W)$ ; the inverse of the Hessian is used to determine the least significant weights in order to “prune” the network and it is also used to find the output uncertainties.
- Considering that there are  $W$  weights then the Hessian have  $W^2$  elements and then the computational time is at least of  $\mathcal{O}(W^2)$  order.

The error function is considered additive with respect to the set of training pattern vectors.

### 9.8.1 Diagonal Approximation

Here the Hessian is calculated by considering it diagonal, then only the diagonal elements  $\frac{\partial^2 E}{\partial w_{ji}^2}$  have to be found (all others are zero).

The error function is considered a sum of elements over the training set as in the backpropagation algorithm:  $E = \sum_p E_p$ .

From the expression of  $a_j$ , in (9.2), the operator  $\frac{\partial}{\partial w_{ji}}$  may be written as:

$$\frac{\partial}{\partial w_{ji}} = \frac{\partial}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} = \frac{\partial}{\partial a_j} z_i$$

and, because  $z_i$  does not depend on  $w_{ji}$  then:

$$\frac{\partial^2}{\partial w_{ji}^2} = \frac{\partial^2}{\partial a_j^2} z_i^2$$

and the diagonal elements of the Hessian for pattern vector  $x_p$  becomes  $\frac{\partial^2 E_p}{\partial w_{ji}^2} = \frac{\partial^2 E_p}{\partial a_j^2} z_i^2$ .

From (9.6) (and on similar grounds):

$$\frac{\partial^2 E_p}{\partial a_j^2} = \frac{d^2 f(a_j)}{da_j^2} \sum_\ell w_{\ell j} \frac{\partial E_p}{\partial a_\ell} + \left( \frac{df(a_j)}{da_j} \right)^2 \sum_\ell \sum_{\ell'} w_{\ell j} w_{\ell' j} \frac{\partial^2 E_p}{\partial a_\ell \partial a_{\ell'}}$$

---

<sup>9.8</sup>See [Bis95] pp. 150–160.

(where  $\frac{\partial}{\partial a_j} \left( \frac{\partial E_p}{\partial a_\ell} \right) = \sum_{\ell'} \frac{\partial^2 E_p}{\partial a_\ell \partial a_{\ell'}} \frac{\partial a_{\ell'}}{\partial a_j}$ ) the sum over  $\ell$  and  $\ell'$  being done over all neurons to which neuron  $i$  is sending connections. In matrix notation it may be written directly as:

$$(\nabla_a \odot \nabla_a) E_p = f''(\mathbf{a}) \odot (W^T \delta_{\text{next}}) + [f'(\mathbf{a}) \odot f'(\mathbf{a})] \odot (W^T \nabla_{\mathbf{a}_{\text{next}}} \odot W^T \nabla_{\mathbf{a}_{\text{next}}}) E_p$$

By neglecting the non-diagonal terms:

$$\frac{\partial^2 E_p}{\partial a_j^2} \approx \frac{d^2 f(a_j)}{da_j^2} \sum_{\ell} w_{\ell j} \frac{\partial E_p}{\partial a_{\ell}} + \left( \frac{df(a_j)}{da_j} \right)^2 \sum_{\ell} w_{\ell j}^2 \frac{\partial^2 E_p}{\partial^2 a_{\ell}}$$

or in matrix notation:

$$\begin{aligned} (\nabla_a \odot \nabla_a) E_p &= f''(\mathbf{a}) \odot (W^T \delta_{\text{next}}) \\ &\quad + [f'(\mathbf{a}) \odot f'(\mathbf{a})] \odot W^{\odot 2T} \odot (\nabla_{\mathbf{a}_{\text{next}}} \odot \nabla_{\mathbf{a}_{\text{next}}}) E_p \end{aligned}$$

and thus, the computational time is reduced from  $\mathcal{O}(W^2)$  to  $\mathcal{O}(W)$  (by neglecting the off diagonal terms  $\frac{\partial^2 E_p}{\partial a_{\ell} \partial a_{\ell'}}$ ,  $\ell \neq \ell'$ ). Note however that in practice the Hessian is quite far from being diagonal.

### 9.8.2 Outer Product Approximation

Considering the sum-of-squares error function for pattern vector  $\mathbf{x}_p$ :

$$E_p = \frac{1}{2} \sum_{s=1}^K [y_s(\mathbf{x}_p) - t_{ps}]^2 = \frac{1}{2} [\mathbf{y}(\mathbf{x}_p) - \mathbf{t}_p]^T [\mathbf{y}(\mathbf{x}_p) - \mathbf{t}_p]$$

then the Hessian is calculated immediately as:

$$\frac{\partial^2 E_p}{\partial w_{ji} \partial w_{k\ell}} = \sum_{s=1}^K \frac{\partial y_s}{\partial w_{ji}} \frac{\partial y_s}{\partial w_{k\ell}} + \sum_{s=1}^K (y_s - t_s) \frac{\partial^2 y_s}{\partial w_{ji} \partial w_{k\ell}}$$

Considering a well trained network and the amount of noise small then the terms  $y_s - t_s$  have to be small and may be neglected; then:

$$\frac{\partial^2 E_p}{\partial w_{ji} \partial w_{k\ell}} \approx \sum_{s=1}^K \frac{\partial y_s}{\partial w_{ji}} \frac{\partial y_s}{\partial w_{k\ell}}$$

and  $\frac{\partial y_s}{\partial w_{ji}}$  may be found by backpropagation procedure.

### 9.8.3 Inverse Hessian

Let consider the Nabla vectorial operator with respect to the weights  $\nabla = \left\{ \frac{\partial}{\partial w_{ji}} \right\}_{ji}$  then the Hessian may be written as a square  $W \times W$  matrix

$$H_P = \sum_{p=1}^P \nabla \cdot \nabla^T E_p$$

( $P$  being the number of vectors in the training set).

By adding a new training vector:

$$H_{P+1} = H_P + \nabla \cdot \nabla^T E_{P+1}$$

A similar recurrent formula may be developed for the inverse Hessian:

$$H_{P+1}^{-1} = H_P^{-1} - \frac{H_P^{-1} \nabla \nabla^T E_{P+1} H_P^{-1}}{1 + \nabla^T H_P^{-1} \nabla E_{P+1}}$$

*Proof.* Considering 3 matrices  $A$ ,  $B$  and  $C$ ,  $A$  inversable, then it is true that:

$$(A + BC)^{-1} = A^{-1} - A^{-1}B(I + CA^{-1}B)^{-1}CA^{-1}$$

(the identity may be verified by multiplication by  $(A + BC)$  to the left respectively to the right).

By putting  $H_P = A$ ,  $\nabla = B$  and  $\nabla^T E_P = C$  then  $\nabla^T H_P^{-1} \nabla E_{P+1}$  have dimension 1 (as the product between a row and column matrix) and the required formula is obtained.  $\square$

Using the above formula and starting with  $H_0 \propto I$  the Hessian may be computed by just one pass trough all training set.

#### 9.8.4 Finite Differences

Another possibility to find the Hessian is by applying small perturbations to weights and calculate the error; then:

$$\begin{aligned} \frac{\partial^2 E}{\partial w_{ji} \partial w_{k\ell}} &= \frac{1}{4\varepsilon^2} [E(w_{ji} + \varepsilon, w_{k\ell} + \varepsilon) - E(w_{ji} - \varepsilon, w_{k\ell} + \varepsilon) \\ &\quad - E(w_{ji} + \varepsilon, w_{k\ell} - \varepsilon) + E(w_{ji} - \varepsilon, w_{k\ell} - \varepsilon)] + \mathcal{O}(\varepsilon^2) \\ &\simeq \frac{1}{4\varepsilon^2} [E(w_{ji} + \varepsilon, w_{k\ell} + \varepsilon) - E(w_{ji} - \varepsilon, w_{k\ell} + \varepsilon) \\ &\quad - E(w_{ji} + \varepsilon, w_{k\ell} - \varepsilon) + E(w_{ji} - \varepsilon, w_{k\ell} - \varepsilon)] \end{aligned} \quad (9.11)$$

where  $0 \lesssim \varepsilon \ll 1$ . By choosing an interval centered around  $(w_{ij}, w_{k\ell})$  the terms  $\mathcal{O}(\varepsilon)$  cancels one each other.

Another approach is to use the gradient  $\nabla E$ :

$$\frac{\partial^2 E}{\partial w_{ji} \partial w_{k\ell}} = \frac{1}{2\varepsilon} \left[ \frac{\partial E}{\partial w_{ji}} \Big|_{w_{ji}+\varepsilon} (w_{k\ell} + \varepsilon) - \frac{\partial E}{\partial w_{ji}} \Big|_{w_{ji}-\varepsilon} (w_{k\ell} + \varepsilon) \right] + \mathcal{O}(\varepsilon^2) \quad (9.12)$$

$$\simeq \frac{1}{2\varepsilon} \left[ \frac{\partial E}{\partial w_{ji}} \Big|_{w_{k\ell}+\varepsilon} - \frac{\partial E}{\partial w_{ji}} \Big|_{w_{k\ell}-\varepsilon} \right] \quad (9.13)$$

#### Remarks:

- ➔ The “brute force” attack used in (9.11) require  $\mathcal{O}(4W)$  computing time for each Hessian element (one pass for each of four  $E$  terms), i.e.  $\mathcal{O}(4W^3)$  computing time for the whole tensor. However it represent a good way of checking other algorithm implementations.

- The second approach, used in (9.11) require  $\mathcal{O}(2W)$  computing time to calculate the gradient and thus  $\mathcal{O}(2W^2)$  computing time for the whole tensor.

### 9.8.5 Exact Hessian

From (9.3):  $\frac{\partial E_p}{\partial w_{k\ell}} = \delta_k z_\ell$  and then by differentiating again:

$$\frac{\partial^2 E_p}{\partial w_{ji} \partial w_{k\ell}} = \frac{\partial}{\partial a_j} \left( \frac{\partial E_p}{\partial w_{k\ell}} \right) \frac{\partial a_j}{\partial w_{ji}} = \frac{\partial}{\partial a_j} \left( \frac{\partial E_p}{\partial w_{k\ell}} \right) z_i = z_i \frac{\partial(\delta_k z_\ell)}{\partial a_j} \quad (9.14)$$

(using also (9.2)).

Because  $z_\ell = f(a_\ell)$  then:

$$\frac{\partial^2 E_p}{\partial w_{ji} \partial w_{k\ell}} = z_i \delta_k f'(a_\ell) h_{\ell j} + z_i z_\ell b_{kj}$$

where:

❖  $h_{\ell j}, b_{kj}$

$$h_{\ell j} \equiv \frac{\partial a_\ell}{\partial a_j} \quad \text{and} \quad b_{kj} \equiv \frac{\partial \delta_k}{\partial a_j}$$

#### The $h_{\ell j}$ coefficients

$a_\ell$  depends over all  $a_s$  from neurons  $s$  which connects to neuron  $\ell$ , then:

$$h_{\ell j} = \sum_s \frac{\partial a_\ell}{\partial a_s} \frac{\partial a_s}{\partial a_j} \quad (9.15)$$

and (as  $a_\ell = \sum_s w_{\ell s} f(a_s)$ ) then:

$$h_{\ell j} = \sum_s w_{\ell s} f'(a_s) h_{sj}$$

i.e. the  $h_{\ell j}$  coefficients may be calculated in terms of the previous ones till the first layer which receive directly the input and for which the coefficients do not have to be calculated.

#### Remarks:

- For the same neuron obviously  $h_{\ell\ell} = 1$ .
- For two different neurons, by continuing the development in (9.15): if a forward propagation path can't be established from neuron  $\ell$  to neuron  $j$ , then  $a_\ell$  and  $a_j$  are independent and, consequently  $h_{\ell j} = 0$

#### The $b_{kj}$ coefficients

The  $b_{kj}$  are calculated using a backpropagation method.

The neuron  $k$  affects the error through all neurons  $s$  to which it sends connections:

$$\delta_k = f'(a_k) \sum_s w_{sk} \delta_s$$

(see also the backpropagation formula (9.6)). Then:

$$\begin{aligned} b_{kj} &= \frac{\partial}{\partial a_j} \left( f'(a_k) \sum_s w_{sk} \delta_s \right) \\ &= f''(a_k) h_{kj} \sum_s w_{sk} \delta_s + f'(a_k) \sum_s w_{sk} b_{sj} \end{aligned} \quad (9.16)$$

❖  $f''$  where  $f''(a) = \frac{d^2 f}{da^2}$  is the second derivative of the activation function.



#### Remarks:

► The derivative  $\frac{\partial}{\partial a_j}$  proceed from the derivative  $\frac{\partial}{\partial w_{ji}}$ , see (9.14). Its application is correct only as long  $w_{ji}$  does not appear explicitly in the expression to be derived in (9.16).

The weight  $w_{ji}$  represents the connection *from* neuron  $i$  to neuron  $j$  while the sum in (9.16) is done over all neurons to which neuron  $j$  send connections *to*.

Thus — according to the feedforward network definition —  $w_{ji}$  can't be among the set of weights  $w_{sk}$  and the formula is correct.

For the  $s$  neuron on output layer:

$$\delta_s = \frac{\partial E_p}{\partial a_s} = f'(a_s) \frac{\partial E_p}{\partial y_s}$$

(as  $y_s = f(a_s)$ ) and:

$$\begin{aligned} b_{ss} &= \frac{\partial \delta_s}{\partial a_s} = \frac{\partial}{\partial a_s} \left( f'(a_s) \frac{\partial E_p}{\partial y_s} \right) = f''(a_s) \frac{\partial E_p}{\partial y_s} + f'(a_s) \frac{\partial}{\partial y_s} \frac{\partial E_p}{\partial a_s} \\ &= f''(a_s) \frac{\partial E_p}{\partial y_s} + f''(a_s) \frac{da_s}{dy_s} \frac{\partial E_p}{\partial y_s} + f'(a_s) \frac{\partial^2 E_p}{\partial y_s^2} \\ &= f''(a_s) [1 + f^{-1'}(y_s)] \frac{\partial E_p}{\partial y_s} + f'(a_s) \frac{\partial^2 E_p}{\partial y_s^2} \end{aligned}$$

(because  $\frac{\partial}{\partial a_s} \leftrightarrow \frac{\partial}{\partial y_s}$  switch places, using the expression of  $\delta_s$  above and  $a_s = f^{-1}(y_s)$ ).

For all other layers the  $b_{kj}$  coefficients may be found from (9.16) by backpropagation and by considering the condition  $w_{jj} = 0$ , i.e. into a feedforward network there is no connection from a neuron to itself.

### 9.8.6 Multiplication with Hessian

Considering the Hessian as a matrix  $\nabla \cdot \nabla^T E_p$  (see section 9.8.3) the problem is to find an easy way of multiplying it with a vector  $\mathbf{v}$  having  $W$  components:

$$\mathbf{v}^T H \equiv \mathbf{v}^T \cdot (\nabla \nabla^T) E_p$$

Using the finite difference method on a interval centered around the current set of weights  $W$  ( $W$  is seen here as a vector):

$$\mathbf{v}^T \nabla(\nabla^T E_p) = \frac{1}{2\varepsilon} [\nabla^T E_p(W + \varepsilon\mathbf{v}) - \nabla^T E_p(W - \varepsilon\mathbf{v})] + \mathcal{O}(\varepsilon^2)$$

where  $0 \lesssim \varepsilon \ll 1$ .

### **Application.**

Let consider a two layer feedforward neural network with the sum-of-squares error function. Then:

❖  $\mathbf{a}, \mathbf{z}, \mathbf{y}$

$$\mathbf{a} = W_{(1)}\mathbf{x}_p \quad , \quad \mathbf{z} = f(\mathbf{a}) \quad \text{and} \quad \mathbf{y} = W_{(2)}\mathbf{z}$$

Let define the operator  $\mathcal{R}(\cdot) = \mathbf{v}^T \nabla$ , then  $\mathcal{R}(W) = \mathbf{v}$  ( $\mathcal{R}$  is a scalar operator, when applied to the components of  $W$  the results are the components of  $\mathbf{v}$  as  $\nabla W = \hat{\mathbf{1}}$ ). Note that  $W$  represents the *total* set of weights, i.e.  $W_{(1)} \cup W_{(2)}$ ; also the  $\mathbf{v}$  vector may be represented, here, in the form of two matrices  $\mathbf{v}_{(1)}$  and  $\mathbf{v}_{(2)}$ , the same way  $W$  is represented by  $W_{(1)}$  and  $W_{(2)}$ .

❖  $\mathcal{R}(\cdot)$

❖  $\mathbf{v}_{(1)}, \mathbf{v}_{(2)}$

By applying  $\mathcal{R}$  to  $\mathbf{a}$ ,  $\mathbf{z}$  and  $\mathbf{y}$ :

$$\mathcal{R}(\mathbf{a}) = \mathbf{v}_{(1)}\mathbf{x}$$

$$\mathcal{R}(\mathbf{z}) = f'(\mathbf{a})\mathcal{R}(\mathbf{a})$$

$$\mathcal{R}(\mathbf{y}) = W_{(2)}\mathcal{R}(\mathbf{z}) + \mathbf{v}_{(2)}\mathbf{z}$$

From the sum-of-squares error function:

❖  $\mathbf{a}_{\text{out}}$

$$\delta_{(2)} = f'(\mathbf{a}_{\text{out}}) \odot [\mathbf{y} - \mathbf{t}_p] \quad \text{for output layer, } \mathbf{a}_{\text{out}} = W_{(2)}\mathbf{z}$$

$$\delta_{(1)} = f'(\mathbf{a}) \odot W_{(2)}\delta_{(2)} \quad \text{for hidden layer}$$

(see (9.4), (9.6) and (9.8)).

By applying again the  $\mathcal{R}$  operator:

$$\mathcal{R}(\delta_{(2)}) = f''(\mathbf{a}_{\text{out}}) \odot \mathcal{R}(\mathbf{a}_{\text{out}}) \odot (\mathbf{y} - \mathbf{t}_p) + f'(\mathbf{a}_{(2)}) \odot \mathcal{R}(\mathbf{y})$$

$$\mathcal{R}(\delta_{(1)}) = f''(\mathbf{a}) \odot \mathcal{R}(\mathbf{a}) \odot W_{(2)}\delta_{(2)} + f'(\mathbf{a}) \odot \mathbf{v}_{(2)}\delta_{(2)} + f'(\mathbf{a}) \odot W_{(2)}\mathcal{R}(\delta_{(2)})$$

Finally, from (9.3):

$$\nabla_{W_{(2)}} E_p = \delta_{(2)}\mathbf{z}^T \quad \text{for output layer}$$

$$\nabla_{W_{(1)}} E_p = \delta_{(1)}\mathbf{x}^T \quad \text{for hidden layer}$$

and then the components of  $\mathbf{v}^T H$  vector are found trough:

$$\mathcal{R}(\nabla_{W_{(2)}} E_p) = \mathcal{R}(\delta_{(2)})\mathbf{z}^T + \delta_{(2)}\mathcal{R}(\mathbf{z}^T) \quad \text{for output layer}$$

$$\mathcal{R}(\nabla_{W_{(1)}} E_p) = \mathcal{R}(\delta_{(1)})\mathbf{x}^T \quad \text{for hidden layer}$$

**Remarks:**

- ➔ The above result may also be used to find the Hessian, by making  $\mathbf{v}$  of the form  $\mathbf{v}^T = (0 \ \dots \ 0 \ 1 \ 0 \ \dots \ 0)$ ; and then the expression  $\mathbf{v}^T \mathcal{R}(E_p)$  gives a row of the Hessian “matrix”.

## CHAPTER 10

# Radial Basis Function Networks

### 10.1 Exact Interpolation

For a unidimensional output network let consider a set of training vectors  $\{\mathbf{x}_p\}_{p=1,\overline{P}}$ , the corresponding set of targets  $\{t_p\}_{p=1,\overline{P}}$  and a function  $h : X \rightarrow Y$  which tries to map exactly the training set to the targets such that:

$$h(\mathbf{x}_p) = t_p \quad , \quad p = \overline{1, P} \quad (10.1)$$

It is assumed that the  $h(\mathbf{x})$  function may be found by the means of a linear combination of a set of  $P$  basis functions of the form  $\varphi(\|\mathbf{x} - \mathbf{x}_p\|)$ :

$$h(\mathbf{x}) = \sum_{p=1}^P w_p \varphi(\|\mathbf{x} - \mathbf{x}_p\|) \quad (10.2)$$

By building the *symmetrical* matrix:

$$\Phi = \begin{pmatrix} \varphi(\|\mathbf{x}_1 - \mathbf{x}_1\|) & \dots & \varphi(\|\mathbf{x}_P - \mathbf{x}_1\|) \\ \vdots & \ddots & \vdots \\ \varphi(\|\mathbf{x}_1 - \mathbf{x}_P\|) & \dots & \varphi(\|\mathbf{x}_P - \mathbf{x}_P\|) \end{pmatrix}$$

❖  $\Phi$

then from (10.1) and (10.2) it follows that:

$$\vec{\mathbf{w}}^T \Phi = \vec{\mathbf{t}}^T \quad (10.3)$$

where  $\vec{\mathbf{w}}^T = (w_1 \quad \dots \quad w_P)$  and  $\vec{\mathbf{t}}^T = (t_1 \quad \dots \quad t_P)$ .

❖  $\vec{\mathbf{w}}, \vec{\mathbf{t}}$

---

<sup>10.1</sup>See [Bis95] pp. 164–167.

The  $\vec{w}$  set of parameters is found immediately if the square matrix  $\Phi$  is inversable (which usually is); the solution being:

$$\vec{w} = \vec{t}^T \Phi^{-1}$$

Examples of basis functions are:

- Gaussian function:  $\varphi(x) = \exp\left(-\frac{x^2}{2\sigma^2}\right)$
- $\varphi(x) = (x^2 + \sigma^2)^{-\alpha}$ ,  $\alpha > 0$
- $\varphi(x) = x^2 \ln x$
- Multi-quadratic function:  $\varphi(x) = \sqrt{x^2 + \sigma^2}$
- Cubic and linear functions:  $\varphi(x) = x^3$ ,  $\varphi(x) = x$

For the multidimensional network output the established relations are immediately extendible as follows:

$$h_k(\mathbf{x}_p) = t_{kp} \quad , \quad p = \overline{1, P} \quad , \quad k = \overline{1, K}$$

$$h_k(\mathbf{x}) = \sum_{p=1}^P w_{kp} \varphi(\|\mathbf{x} - \mathbf{x}_p\|) \quad , \quad k = \overline{1, K}$$

❖ **h** Let  $\mathbf{h} \equiv (h_1 \dots h_K)$  then  $\mathbf{h}(\mathbf{x}_p) = \mathbf{t}_p$  and also by building  $W$  using all  $\vec{w}^T$  as rows and  $T$  using all  $\vec{t}^T$  again as rows then  $W\Phi = T$  and thus  $W = T\Phi^{-1}$  (assuming  $\Phi$  inversable, of course).

## → 10.2 Radial Basis Function Networks

The radial basis function network is built by considering the basis function as an neuronal activation function and the  $w_{kp}$  parameters as weights.

### Remarks:

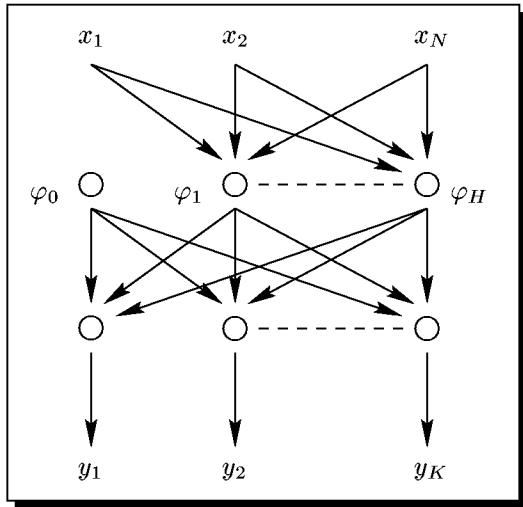
- ➔ To perform a exact interpolation of the training set is not only unnecessary but a bad thing as well — see the course of dimensionality problem. For this reason some modifications are made.

When using the basis functions in neural networks, the following changes are performed:

- The number  $H$  of basis functions do not need to be equal to the number  $P$  of training vectors — usually is much less. So  $\{w_{kp}\} \rightarrow \{w_{kj}\}$ .
- The basis functions do not have to be centered around the training vectors.
- The basis functions may have themselves some tunable (during training) parameters.
- There may be a bias parameter.

---

<sup>10.2</sup>See [Bis95] pp. 167–169.



**Figure 10.1:** The radial basis function network architecture. On the hidden layer each neuron have a radial basis activation function. The activation function of the output layer is the identity. For bias  $\varphi_0 = 1$ .

Then the output of the network looks like:

$$y_k(\mathbf{x}) = \sum_{j=1}^H w_{kj} \varphi_j(\mathbf{x}) + w_{k0} \Leftrightarrow \mathbf{y}(\mathbf{x}) = \widetilde{\mathbf{W}} \widetilde{\varphi}(\mathbf{x}) \quad (10.4)$$

where  $\widetilde{\varphi}^T \equiv (\varphi_0 \dots \varphi_K)$  and  $\widetilde{\mathbf{W}}$  holds both weights and bias.

❖  $\widetilde{\varphi}, \widetilde{\mathbf{W}}$

### ✎ Remarks:

- If the basis functions are of Gaussian type the they are of the form:

$$\varphi_j(\mathbf{x}) = \exp \left[ -\frac{(\mathbf{x} - \boldsymbol{\mu}_j)^T \Sigma_j^{-1} (\mathbf{x} - \boldsymbol{\mu}_j)}{2} \right]$$

where  $\Sigma_j$  is a covariant symmetrical matrix.

The model may be represented in the form of a two layer network where the hidden neurons have the basis functions as activation function. Note that the weight matrix from input to hidden layer is <sup>10.3</sup> and the output layer have the identity activation function. See figure 10.1. The basis function associated with bias is the constant function  $\varphi_0(\mathbf{x}) = 1$ .

## 10.3 Relation to Other Theories

### 10.3.1 Relation to Regularization Theory

A way to control the smoothing of the model (in order to avoid large oscillations in the output after a small variation of input) — and thus to control the complexity of the model

<sup>10.3</sup>See [Bis95] pp. 171–179.

— would be through an additional term in the error function which penalizes un-smooth network mappings.

For a unidimensional output, the sum-of-squares error function will be written as:

$$E = \frac{1}{2} \sum_{p=1}^P [y(\mathbf{x}_p) - t_p]^2 + \frac{\nu}{2} \int_X |\mathcal{P}y|^2 d\mathbf{x} \quad (10.5)$$

❖  $\nu, \mathcal{P}$

where  $\nu$  is a constant named *regularization parameter* and  $\mathcal{P}$  is a differential operator such that large curvatures of  $y(\mathbf{x})$  gives rise to large values of  $|\mathcal{P}y|^2$ .

❖  $\delta_D$

By replacing  $y(\mathbf{x}_p)$  with  $y(\mathbf{x}_p) \delta_D(\mathbf{x} - \mathbf{x}_p)$  — where  $\delta_D$  is the Dirac function — the error function (10.5) may be expressed as a functional of  $y(\mathbf{x})$  (see the mathematical appendix).

The condition of stationarity for  $E$  is to set to 0 its derivative with respect to  $y(\mathbf{x})$ :

$$\frac{\delta E}{\delta y} = \sum_{p=1}^P [y(\mathbf{x}_p) - t_p] \delta_D(\mathbf{x} - \mathbf{x}_p) + \nu \widehat{\mathcal{P}} \mathcal{P} y(\mathbf{x}) = 0 \quad (10.6)$$

Euler-Lagrange equation

where  $\widehat{\mathcal{P}}$  is the adjoint differential operator of  $\mathcal{P}$  (see also the mathematical appendix). (10.6) is named the *Euler-Lagrange equation*.

Green's functions

The solutions to (10.6) are found in terms of the *Green's functions*  $G$  of the operator  $P$  which are defined as being the solutions to the equation:

$$\widehat{\mathcal{P}} \mathcal{P} G(\mathbf{x}, \mathbf{x}') = \delta_D(\mathbf{x} - \mathbf{x}')$$

and then the solutions  $y(\mathbf{x})$  are searched in the form:

$$y(\mathbf{x}) = \sum_{p=1}^P w_p G(\mathbf{x}, \mathbf{x}_p) \quad (10.7)$$

where  $\{w_p\}_{p=1}^P$  are parameters found by replacing solution (10.7) back into (10.6), giving:

$$\sum_{p=1}^P [y(\mathbf{x}_p) - t_p] \delta_D(\mathbf{x} - \mathbf{x}_p) + \nu \sum_{p=1}^P w_p \delta_D(\mathbf{x} - \mathbf{x}_p)$$

and by integrating around a small enough vicinity of  $\mathbf{x}_p$  (small enough such that it will not contain any other  $\mathbf{x}_{p'}$ ), for all  $p = \overline{1, P}$ :

$$y(\mathbf{x}_p) - t_p + \nu w_p = 0 \quad p = \overline{1, P} \quad (10.8)$$

(because of the way  $\delta_D$  is defined).

❖  $G$

By replacing the solution (10.7) in the above equation and considering the  $G$  matrix and  $\vec{w}$  and  $\vec{t}$  vectors:

$$G = \begin{pmatrix} G(\mathbf{x}_1, \mathbf{x}_1) & \cdots & G(\mathbf{x}_P, \mathbf{x}_1) \\ \vdots & \ddots & \vdots \\ G(\mathbf{x}_1, \mathbf{x}_P) & \cdots & G(\mathbf{x}_P, \mathbf{x}_P) \end{pmatrix}$$

finally (10.8) becomes:

$$\vec{w}^T(G + \nu I) = \vec{t}^T$$

By comparing (10.4) and (10.7) it becomes clear that the Green's functions plays the same role as the basis functions (also the above equation may be compared to (10.3)).



#### Remarks:

- If the  $\mathcal{P}$  operator is invariant to translation and rotation then  $G$  functions are dependent only on the norm of  $\mathbf{x}$ :  $G(\mathbf{x}, \mathbf{x}') = G(\|\mathbf{x} - \mathbf{x}'\|)$ .

### 10.3.2 Relation to Interpolation Theory

Let consider a mapping  $h : X \rightarrow Y$ . A noise is added to input  $x$ . Let  $p(x)$  be the probability density of input and  $\tilde{p}(\xi)$  the probability density of the noise.

Considering the sum-of-squares error function:

$$E = \frac{1}{2} \iint_X [y(x + \xi) - h(x)]^2 p(x) \tilde{p}(\xi) dx d\xi$$

where  $h(x)$  is the desired output (target) for  $x + \xi$ .

By making the change of variable  $x + \xi = z$ :

$$E = \frac{1}{2} \iint_X [y(z) - h(x)]^2 p(x) \tilde{p}(z - x) dx dz$$

and the  $y(x)$  is found by setting  $E$  functional derivative (see the mathematical appendix) with respect to  $y(z)$  to 0:

$$\frac{\delta E}{\delta y} = \int_X [y(z) - h(x)] p(x) \tilde{p}(z - x) dx = 0 \Rightarrow y(z) = \frac{\int_X h(x) p(x) \tilde{p}(z - x) dx}{\int_X p(x) \tilde{p}(z - x) dx}$$

Considering a sufficiently large set of training vectors then the integrals may be approximated by sums:

$$y(z) = \sum_{p=1}^P h(x_p) \frac{\tilde{p}(z - x_p)}{\sum_{q=1}^P \tilde{p}(z - x_q)}$$

and by comparing to (10.4) it becomes obvious that the above expression of  $y(x)$  represents an expansion in basis functions form.

### 10.3.3 Relation to Kernel Based Method

Considering a set of training data  $\{\mathbf{x}_p, \mathbf{t}_p\}_{p=1,P}$  then the *estimated* probability density of a pair  $\{\mathbf{x}, \mathbf{t}\}$ , assuming an exponential smoothing kernel function, is<sup>1</sup>:

$$\tilde{p}(\mathbf{x}, \mathbf{t}) = \frac{1}{P} \sum_{p=1}^P \frac{1}{(2\pi L^2)^{(N+K)/2}} \exp \left[ -\frac{\|\mathbf{x} - \mathbf{x}_p\|^2 + \|\mathbf{t} - \mathbf{t}_p\|^2}{2L^2} \right] \quad (10.9)$$

❖  $L$  where  $L$  is the length of the hypercube side ( $N$  being the dimensionality of the input  $X$  space and  $K$  the dimensionality of the  $Y$  output space).

The optimal mapping  $y(\mathbf{x})$  is given by the average over the desired output conditioned by the input:

$$\mathbf{y}(\mathbf{x}) = \int_Y \mathbf{t} p(\mathbf{t}|\mathbf{x}) d\mathbf{t} = \frac{\int_Y \mathbf{t} p(\mathbf{x}, \mathbf{t}) d\mathbf{t}}{\int_Y p(\mathbf{x}, \mathbf{t}) d\mathbf{t}}$$

(by using also the Bayes theorem;  $\int_Y p(\mathbf{x}, \mathbf{t}) d\mathbf{t} = p(\mathbf{x})$ ).

By replacing the  $p(\mathbf{x}, \mathbf{t})$  with the value from (10.9) and integrating (see also the mathematical appendix regarding Gaussian integrals):

$$\mathbf{y}(\mathbf{x}) = \frac{\sum_{p=1}^P \mathbf{t}_p \exp \left[ -\frac{\|\mathbf{x} - \mathbf{x}_p\|^2}{2L^2} \right]}{\sum_{p=1}^P \exp \left[ -\frac{\|\mathbf{x} - \mathbf{x}_p\|^2}{2L^2} \right]}$$

Nadaraya-Watson estimator known also as *Nadaraya-Watson estimator* — a formula similar to (10.4).

## → 10.4 Classification

For classification into a set of  $\mathcal{C}_k$  classes the Bayes theorem gives:

$$P(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)P(\mathcal{C}_k)}{p(\mathbf{x})} = \frac{p(\mathbf{x}|\mathcal{C}_k)P(\mathcal{C}_k)}{\sum_{q=1}^K p(\mathbf{x}|\mathcal{C}_q)P(\mathcal{C}_q)} \quad (10.10)$$

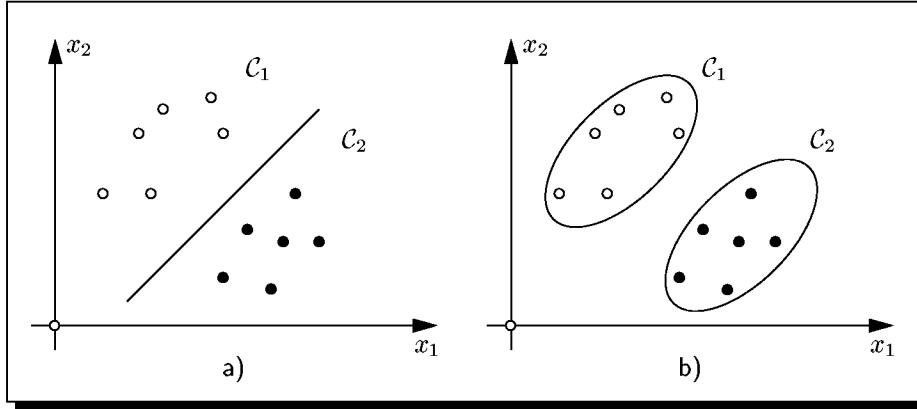
(because  $p(\mathbf{x})$  plays the role of a normalization factor).

Because the posterior probabilities  $P(\mathcal{C}_k|\mathbf{x})$  is what the model should calculate, (10.10) may be compared with (10.4), the basis functions being:

$$\varphi_k(\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)}{\sum_{q=1}^K p(\mathbf{x}|\mathcal{C}_q)P(\mathcal{C}_q)}$$

---

<sup>1</sup>See the non-parametric/kernel based method in the statistical appendix.  
<sup>10.4</sup>See [Bis95] pp.179–182.



**Figure 10.2:** Bidimensional pattern vector space. The difference between perceptron based classifiers and radial basis function classification: a) The perceptron represents the decision boundaries; b) the radial basis function networks represents the classes through the basis functions.

and the output layer having only one connection per neuron from the hidden layer, the weight being  $P(\mathcal{C}_k)$  (in this case the hidden layer/number of basis function is also  $K$ ).

The interpretation of this method is that each basis function represents a class — while the perceptron network represents the hyperplanes acting as decision boundaries. See figure 10.2.

It is possible to improve the model by considering a mixture of basis functions  $m = \overline{1, M}$  (instead of one single basis function per class):

$$p(\mathbf{x}|\mathcal{C}_k) = \sum_{m=1}^M p(\mathbf{x}|m) P(m|\mathcal{C}_k)$$

(i.e. the mixture model). The total probability density for  $\mathbf{x}$  also changes to:

$$p(\mathbf{x}) = \sum_{k=1}^K p(\mathbf{x}|\mathcal{C}_k) P(\mathcal{C}_k) = \sum_{m=1}^M p(\mathbf{x}|m) P(m)$$

where  $P(m)$  represents the prior probability of mixture component and it can be expressed as:

$$P(m) = \sum_{k=1}^K P(m|\mathcal{C}_k) P(\mathcal{C}_k)$$

By replacing the above expressions into the Bayes theorem:

$$P(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k) P(\mathcal{C}_k)}{p(\mathbf{x})} = \frac{\sum_{m=1}^M P(m|\mathcal{C}_k) p(\mathbf{x}|m) P(\mathcal{C}_k) \frac{P(m)}{P(m)}}{\sum_{m=1}^M p(\mathbf{x}|m) P(m)} = \sum_{m=1}^M w_{km} \varphi_m(\mathbf{x})$$

❖  $w_{km}$ ,  $\varphi_m$  ( $\frac{P(m)}{P(m)} = 1$  being added on purpose) where:

$$\varphi_m(\mathbf{x}) = \frac{p(\mathbf{x}|m) P(m)}{\sum_{n=1}^M p(\mathbf{x}|n) P(n)} = P(m|\mathbf{x}) \quad \text{and}$$

$$w_{km} = \frac{P(m|\mathcal{C}_k) P(\mathcal{C}_k)}{P(m)} = P(\mathcal{C}_k|m)$$

(by using the Bayes theorem), i.e. the basis functions represents the posterior probability of  $\mathbf{x}$  being generated by the component  $m$  of the mixture model and the weights represent the posterior probability of class  $\mathcal{C}_k$  membership given a pattern vector generated by component  $m$  of the mixture.

## ► 10.5 Network Learning

The learning procedure consists of two steps:

- The basis functions are established and their parameters are found, usually by an unsupervised learning procedure, i.e. only the inputs  $\{\mathbf{x}_p\}$  from the training set are considered (the targets  $\{\mathbf{t}_p\}$  are ignored). The basis functions are usually defined to depend only over a distance between the pattern vector  $\mathbf{x}$  and the training vectors  $\{\mathbf{x}_p\}$ , i.e.  $\|\mathbf{x} - \mathbf{x}_p\|$ , such that they have a radial symmetry.
- Then, having the basis functions properly established, the weights on output layer are to be found.

### 10.5.1 Radial Basis Functions

The relation between radial basis function network and other statistical methods described in section 10.3, suggest that the basis functions should represent the probability density of input vectors. Then an unsupervised method may be envisaged to find the parameters of the basis functions, several are described below.

#### ***Subsets of data points***

This procedure builds the basis functions as Gaussians:

$$\varphi_j(\mathbf{x}) = \exp \left[ -\frac{\|\mathbf{x} - \boldsymbol{\mu}_j\|^2}{2\sigma_j^2} \right] \quad (10.11)$$

The  $\{\boldsymbol{\mu}_j\}$  vectors are chosen as a subset of the training set  $\{\mathbf{x}_p\}$ .

The  $\{\sigma_j\}$  parameters are chosen all equal to a multiple of average distances between the centers of radial functions (as defined by vectors  $\boldsymbol{\mu}_i$ ). They should be chosen such as to allow for a small overlap between radial functions.

---

<sup>10.5</sup>See [Bis95] pp. 170–171 and pp. 183–191.

 **Remarks:**

- This ad hoc algorithm usually gives poor results but it may be useful as a starting point for further optimization.

**Clustering**

The basis functions are built as Gaussian, as above.

The training set is divided into a number of subsets  $\mathcal{S}_j$ , equal to the number of basis functions, such that an error function associated with the clustering is minimized:

$$E = \sum_{j=1}^H \sum_{\mathbf{x}_p \in \mathcal{S}_j} \|\mathbf{x}_p - \boldsymbol{\mu}_j\|^2$$

At the beginning the learning set is divided into subsets at random. Then the mean  $\boldsymbol{\mu}_j$  is calculated inside each subset and each training pattern vector  $\mathbf{x}_p$  is reassigned to the subset having the closest mean. The procedure is repeated until there are no more reassignments.

 **Remarks:**

- Alternatively the  $\boldsymbol{\mu}_j$  vectors may be found by an on-line stochastic procedure. First they are chosen at random from the training set. Then they are updated according to:

$$\Delta \boldsymbol{\mu}_j = \mu (\mathbf{x}_p - \boldsymbol{\mu}_j)$$

for all  $\mathbf{x}_p$ , where  $\mu$  represents a “learning constant”. Note that this is similar to finding the root of  $\nabla_{\boldsymbol{\mu}_j} E$ , i.e. the minima of  $E$ , through the Robbins-Monro algorithm.

**Gaussian mixture model**

The radial basis functions are considered of the Gaussian form (10.11).

The probability density of the pattern vector is considered a mixture model of the form:

$$p(\mathbf{x}) = \sum_{j=1}^H P(j) \varphi_j(\mathbf{x})$$

It is possible to use the EM (expectation-maximisation) algorithm to find  $\boldsymbol{\mu}_j$ ,  $\sigma_j$  and  $P(j)$  at step  $s+1$  from the values at step  $s$ , starting with some initial values  $\boldsymbol{\mu}_{0j}$ ,  $\sigma_{0j}$  and  $P_0(j)$ :

$$\boldsymbol{\mu}_{(s+1)j} = \frac{\sum_{p=1}^P P_{(s)}(j|\mathbf{x}_p) \mathbf{x}_p}{\sum_{p=1}^P P_{(s)}(j|\mathbf{x}_p)}$$

$$\sigma_{(s+1)j} = \sqrt{\frac{\sum_{p=1}^P P_{(s)}(j|\mathbf{x}_p) \|\mathbf{x}_p - \boldsymbol{\mu}_{(s+1)j}\|^2}{N \sum_{p=1}^P P_{(s)}(j|\mathbf{x}_p)}}$$

$$P_{(s+1)}(j) = \frac{1}{P} \sum_{p=1}^P P_{(s)}(j|\mathbf{x}_p)$$

where  $j = \overline{1, H}$  and:

$$P(j|\mathbf{x}) = \frac{P(j) \varphi(\mathbf{x})}{p(\mathbf{x})}$$

### **Supervised learning**

It is possible to envisage also a supervised learning for the hidden layer. E.g. considering the basis function of Gaussian form (10.11),  $y_k = \sum_{j=1}^H w_{kj} \varphi_j$  and the sum-of-squares error

$E = \frac{1}{2} \sum_{p=1}^P [y_k(\mathbf{x}_p) - t_{kp}]^2$  which have to be minimized then the conditions are:

$$\frac{\partial E}{\partial \sigma_j} = \sum_{p=1}^P \sum_{k=1}^K w_{kj} [y_k(\mathbf{x}_p) - t_{kp}] \exp\left(-\frac{\|\mathbf{x}_p - \mu_j\|^2}{2\sigma_j^2}\right) \frac{\|\mathbf{x}_p - \mu_j\|^2}{\sigma_j^3} = 0$$

$$\frac{\partial E}{\partial \mu_{sj}} = \sum_{p=1}^P \sum_{k=1}^K w_{kj} [y_k(\mathbf{x}_p) - t_{kp}] \exp\left(-\frac{\|\mathbf{x}_p - \mu_j\|^2}{2\sigma_j^2}\right) \frac{x_{sp} - \mu_{sj}}{\sigma_j^2} = 0$$

However, to solve the above equations is very computational intensive and the solutions have to be checked against the possibility of a local minima of  $E$ .

### **10.5.2 Output Layer Weights**

Considering the matrix  $\widetilde{W} = \{w_{kj}\}_{\substack{i=1, K \\ j=0, H}}$  (biases included) and the vectorial function  $\widetilde{\varphi}^T = (\varphi_0 \dots \varphi_H)$  then (10.4) gives:

$$\mathbf{y} = \widetilde{W} \widetilde{\varphi}(\mathbf{x}) \quad (10.12)$$

and it may be considered as a single layer network. Then considering the sum-of-squares error function  $E = \frac{1}{2} \sum_{p=1}^P [\mathbf{y}(\mathbf{x}_p) - \mathbf{t}_p]^T [\mathbf{y}(\mathbf{x}_p) - \mathbf{t}_p]$  the least squares technique is applied to find the weights.

❖  $\Phi, T$

Considering that  $\mathbf{y}$  is obtained by the form of a generalized linear discriminant, see (10.12), then the following matrices are build:

$$\Phi = \begin{pmatrix} \varphi_0(\mathbf{x}_1) & \cdots & \varphi_0(\mathbf{x}_P) \\ \vdots & \ddots & \vdots \\ \varphi_H(\mathbf{x}_1) & \cdots & \varphi_H(\mathbf{x}_P) \end{pmatrix} \quad \text{and} \quad T = \begin{pmatrix} t_{11} & \cdots & t_{1P} \\ \vdots & \ddots & \vdots \\ t_{K1} & \cdots & t_{KP} \end{pmatrix}$$

❖  $\Phi^\dagger$

Then (according to the least squares technique)  $\widetilde{W} \Phi = T$  which have the solution:

$$\widetilde{W} = T \Phi^\dagger \quad \text{where} \quad \Phi^\dagger = \Phi^T (\Phi \Phi^T)^{-1}$$

( $\Phi^\dagger$  being the pseudo-inverse of  $\Phi$ ).

*Proof.*  $\widetilde{W}\Phi = T \Rightarrow \widetilde{W}(\Phi\Phi^T) = T\Phi^T \Rightarrow \widetilde{W} = T\Phi^\dagger$ .

□



## CHAPTER 11

# Error Functions

### ► 11.1 Generalities

Usually neural networks are used to generalize (not to memorize) from a set of training data.

The most general way to describe the modeller (the neural network) is in terms of joint probability  $p(\mathbf{x}, \mathbf{t})$ , where  $\mathbf{t}$  is the desired output (target) given  $\mathbf{x}$  as input. The joined probability may be decomposed in the conditional probability of  $\mathbf{t}$ , given  $\mathbf{x}$ , and the probability of  $\mathbf{x}$ :

$$p(\mathbf{x}, \mathbf{t}) = p(\mathbf{t}|\mathbf{x}) p(\mathbf{x})$$

where the unconditional probability of  $\mathbf{x}$  may be written also as:

$$p(\mathbf{x}) = \int_Y p(\mathbf{x}, \mathbf{t}) d\mathbf{t}$$

Most error functions may be expressed in terms of the maximum likelihood function  $\mathcal{L}$ , given the  $\{\mathbf{x}_p, \mathbf{t}_p\}_{p=1}^P$  set of training vectors:

$$\mathcal{L} = \prod_{p=1}^P p(\mathbf{x}_p, \mathbf{t}_p) = \prod_{p=1}^P p(\mathbf{t}_p|\mathbf{x}_p) p(\mathbf{x}_p)$$

which represents the probability of observing the training set  $\{\mathbf{x}_p, \mathbf{t}_p\}$ .

---

<sup>11.1</sup>See [Bis95] pp. 194–195.

The ANN parameters are tuned such that  $\mathcal{L}$  is maximized. It may be convenient instead to minimize a derived function:

$$\tilde{E} = -\ln \mathcal{L} = -\sum_{p=1}^P \ln p(\mathbf{t}_p|\mathbf{x}_p) - \sum_{p=1}^P \ln p(\mathbf{x}_p) \Rightarrow E = -\sum_{p=1}^P \ln p(\mathbf{t}_p|\mathbf{x}_p) \quad (11.1)$$

error function

where, in  $E$ , the terms  $p(\mathbf{x}_p)$  were dropped because they don't depend on network parameters. The  $E$  function is called *error function*.

## ► 11.2 Sum-of-Squares Error

It is assumed that the  $K$  components of the output vector are statistically independent, i.e.

$$p(\mathbf{t}|\mathbf{x}) = \prod_{k=1}^K p(t_k|\mathbf{x}) \quad (11.2)$$

( $t_k$  being the  $k$ -th component of vector  $\mathbf{t}$ ).

It is assumed that the distribution of target data is Gaussian, i.e. it is composed from a deterministic function  $h(\mathbf{x})$  and some Gaussian noise  $\varepsilon$ :

$$t_k = h_k(\mathbf{x}) + \varepsilon_k \quad \text{where} \quad p(\varepsilon_k) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{\varepsilon_k^2}{2\sigma^2}\right) \quad (11.3)$$

Then  $\varepsilon_k = t_k - h_k(\mathbf{x})$ ,  $h_k(\mathbf{x}) = y_k(\mathbf{x}, W)$  because it's the model represented by the neural network ( $W$  being the network parameters), and  $p(\varepsilon_k) = p(t_k|\mathbf{x})$  (as  $h_k(\mathbf{x})$  are purely deterministic):

$$p(t_k|\mathbf{x}) = \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{[y_k(\mathbf{x}, W) - t_k]^2}{2\sigma^2}\right) \quad (11.4)$$

By using the above expression in (11.2) and then in (11.1), the error function becomes:

$$E = \frac{1}{2\sigma^2} \sum_{p=1}^P \sum_{k=1}^K [y_k(\mathbf{x}_p, W) - t_{kp}]^2 + PK \ln \sigma + \frac{PK}{2} \ln 2\pi$$

❖  $t_{kp}$

where  $t_{kp}$  is the component  $k$  of vector  $\mathbf{t}_p$ . By dropping the last two terms which are weight ( $W$ ) independent, as well as the  $1/\sigma^2$  from the first term, the error function may be written as:

$$E = \frac{1}{2} \sum_{p=1}^P \sum_{k=1}^K \|y_k(\mathbf{x}_p, W) - t_{kp}\|^2 \quad (11.5)$$

### Remarks:

► It is sometimes convenient to use one error function for network training, e.g. sum-

RMS

<sup>11.2</sup>See [Bis95] pp. 195–208.

of-squares, and another to test over the performance achieved after training, e.g. the root-mean-square (RMS) error:

$$E_{\text{RMS}} = \frac{\sum_{p=1}^P \|\mathbf{y}_p - \mathbf{t}_p\|}{\sum_{p=1}^P \|\mathbf{t}_p - \langle \mathbf{t} \rangle\|} \quad \text{where} \quad \langle \mathbf{t} \rangle = \frac{1}{P} \sum_{p=1}^P \mathbf{t}_p$$

which have the advantage that is not growing with the number of tests  $P$ .

### 11.2.1 Linear Output Units

Generally, the output neuron  $k$  is applying the activation function  $f$  to the weighted sum of  $z_j$  — inputs from (last) hidden layer — in order to compute its own output:

$$y_k(\mathbf{x}, W) = f(a_k) \quad \text{where} \quad a_k = \sum_{j=0}^H w_{kj} z_j(\mathbf{x}, \widetilde{W})$$

where  $\widetilde{W}$  is the set of all weights except those involving the output layer,  $H$  is the number of neurons on hidden layer and  $w_{k0}$  is the bias corresponding to  $z_0 = 1$  ( $w_{kj}$  being the weight for connection from neuron  $j$  to neuron  $k$ ).

❖  $\widetilde{W}, H, a_k, z_j$

Then, the derivative of error (11.5), with respect to the total input  $a_k$  into the neuron  $k$ , is:

$$\frac{\partial E}{\partial a_k} = \sum_{p=1}^P [y_p(\mathbf{x}_p, W) - t_{kp}] \frac{df(a_k)}{da_k}$$

Assuming a linear activation function  $f(x) = x$  then:

$$\frac{\partial E}{\partial a_k} = \sum_{p=1}^P [y_p(\mathbf{x}_p, W) - t_{kp}] = \frac{\partial E}{\partial y_k}$$

and the optimization of output layer weights becomes simple. Let:

❖  $\langle \mathbf{t} \rangle, \langle \mathbf{z} \rangle$

$$\langle \mathbf{t} \rangle = \frac{1}{P} \sum_{p=1}^P \mathbf{t}_p \quad \text{and} \quad \langle \mathbf{z} \rangle = \frac{1}{P} \sum_{p=1}^P \mathbf{z}(\mathbf{x}_p)$$

and the following  $W_{\text{out}}$ ,  $\widetilde{Z}$  and  $\widetilde{T}$  matrices are defined:

❖  $W_{\text{out}}, \widetilde{Z}, \widetilde{T}$

$$W_{\text{out}} = \begin{pmatrix} w_{11} & \cdots & w_{1H} \\ \vdots & \ddots & \vdots \\ w_{K1} & \cdots & w_{KH} \end{pmatrix}, \quad \widetilde{Z} = \begin{pmatrix} \widetilde{z}_{11} & \cdots & \widetilde{z}_{1P} \\ \vdots & \ddots & \vdots \\ \widetilde{z}_{H1} & \cdots & \widetilde{z}_{HP} \end{pmatrix}, \quad \widetilde{T} = \begin{pmatrix} \widetilde{t}_{11} & \cdots & \widetilde{t}_{1P} \\ \vdots & \ddots & \vdots \\ \widetilde{t}_{K1} & \cdots & \widetilde{t}_{KP} \end{pmatrix}$$

where  $W_{\text{out}}$  holds the weights associated with links between (last) hidden and output layers,  $\widetilde{z}_{jp} = z_j(\mathbf{x}_p) - \langle z_j \rangle$  and  $\widetilde{t}_{kp} = t_{kp} - \langle t_k \rangle$ . Then the solution for weights matrix is:

❖  $\widetilde{z}_{jp}, \widetilde{t}_{kp}$

$$W_{\text{out}} = \widetilde{T} \widetilde{Z}^\dagger \quad \text{where} \quad \widetilde{Z}^\dagger = \widetilde{Z}^T (\widetilde{Z} \widetilde{Z}^T)^{-1} \quad (11.6)$$

❖  $\tilde{Z}^\dagger$ ,  $w_0$   $\tilde{Z}^\dagger$  being the pseudo-inverse of  $\tilde{Z}$ . The biases  $w_0^T = (w_{10} \dots w_{K0})$  are found trough:

$$w_0 = \langle t \rangle - W_{\text{out}} \langle z \rangle \quad (11.7)$$

*Proof.* By writing explicitly the bias:

$$y_k = \sum_{j=1}^H w_{kj} z_j + w_{k0} \quad (11.8)$$

and putting the condition of minimum of  $E$  with respect to biases:

$$\frac{\partial E}{\partial w_{k0}} = \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial w_{k0}} = \sum_{p=1}^P \left[ \sum_{j=1}^H w_{kj} z_j(x_p) + w_{k0} - t_{kp} \right] = 0$$

❖  $\langle t_k \rangle$ ,  $\langle z_j \rangle$

( $z_j(x_p)$  being the output of the hidden neuron  $j$  when the input of the network is  $x_p$ ) and then:

$$w_{k0} = \langle t_k \rangle - \sum_{j=1}^H w_{kj} \langle z_j \rangle \quad \text{where} \quad \langle t_k \rangle = \frac{1}{P} \sum_{p=1}^P t_{kp}, \quad \langle z_j \rangle = \frac{1}{P} \sum_{p=1}^P z_j(x_p)$$

bias

i.e. the bias compensate for the difference between the average of target and the weighted average of (last) hidden layer output.

By replacing the biases found, back into (11.5) (through (11.8)):

$$E = \frac{1}{2} \sum_{p=1}^P \sum_{k=1}^K \left[ \sum_{j=1}^H w_{kj} \tilde{z}_{jp} - \tilde{t}_{kp} \right]^2 \quad (11.9)$$

The minimum of  $E$  with respect to  $w_{kj}$  is found from:

$$\frac{\partial E}{\partial w_{kj}} = \sum_{p=1}^P \left[ \sum_{s=1}^{\ell} w_{ks} \tilde{z}_{sp} - \tilde{t}_{kp} \right] \tilde{z}_{jp} = 0, \quad k = \overline{1, K}, \quad j = \overline{1, H} \quad (11.10)$$

and the set of equations (11.10) may be condensed into one matrix equation:

$$W_{\text{out}} \tilde{Z} \tilde{Z}^T - \tilde{T} \tilde{Z}^T = \tilde{0} \quad (11.11)$$

which yields the desired result.  $\square$



### Remarks:

outliers

- ➔ It was assumed that  $\tilde{Z} \tilde{Z}^T$  is invertible.
- ➔ The solution (11.6) was found by maintaining the  $\tilde{W}$  weights fixed. However if they change, the optimal solution (11.6) changes as well.
- ➔ The sum-of-squares error function is sensitive to training vectors with high error — called *outliers*. It is also sensitive to misslabeled data because it leads to high error.

### 11.2.2 Linear Sum Rules

❖  $\mathbf{u}$ ,  $u_0$

Let assume that for all targets (desired outputs) from the training set it is true that:

$$\mathbf{u}^T \mathbf{t}_p + u_0 = 0 \quad \text{where} \quad \mathbf{u}, u_0 = \text{const.}, \forall p$$

By summing over all training vectors it follows that  $u_0 = -\mathbf{u}^T \langle \mathbf{t} \rangle$  and then:

$$\mathbf{u}^T \mathbf{t}_p = -\mathbf{u}^T \langle \mathbf{t} \rangle \quad (11.12)$$

Then it is also true that:

$$\mathbf{u}^T \mathbf{y} = \mathbf{u}^T \langle \mathbf{t} \rangle$$

i.e. if the training set have the property (11.12) then any network output will have the same property.

*Proof.* The network output is  $\mathbf{y} = W_{\text{out}} \mathbf{z} + \mathbf{w}_0$  and also from (11.7):  $\mathbf{w}_0 = \langle \mathbf{t} \rangle - W_{\text{out}} \langle \mathbf{z} \rangle$ . Then:

$$\mathbf{u}^T \mathbf{y} = \mathbf{u}^T (W_{\text{out}} \mathbf{z} + \mathbf{w}_0) = \mathbf{u}^T \tilde{T} \tilde{Z}^\dagger (\mathbf{z} - \langle \mathbf{z} \rangle) + \mathbf{u}^T \langle \mathbf{t} \rangle$$

But the elements of the row matrix (vector)  $\mathbf{u}^T \tilde{T}$  are:

$$\{\mathbf{u}^T \tilde{T}^T\}_p = \mathbf{u}^T \tilde{T}(:, p) = \mathbf{u}^T (\mathbf{t}_p - \langle \mathbf{t} \rangle) = 0$$

by virtue of (11.12).  $\square$

### 11.2.3 Significance of Network Output

Let consider that the number of training data sets grows to infinity and the following error function (using Euclidean norm):

$$\begin{aligned} E &= \lim_{P \rightarrow \infty} \frac{1}{2P} \sum_{p=1}^P \|\mathbf{y}(\mathbf{x}_p, W) - \mathbf{t}_p\|^2 \\ &= \frac{1}{2} \sum_{k=1}^K \iint_{X, Y_k} [y_k(\mathbf{x}_p, W) - t_{kp}]^2 p(t_k, \mathbf{x}) dt_k d\mathbf{x} \\ &= \frac{1}{2} \sum_{k=1}^K \iint_{X, Y_k} [y_k(\mathbf{x}_p, W) - t_k]^2 p(t_k | \mathbf{x}) p(\mathbf{x}) dt_k d\mathbf{x} \end{aligned} \quad (11.13)$$

where  $Y_k$  is the unidimensional component of the output space  $Y$  related to  $t_k$ .

The following conditional averages are defined:

$$\langle t_k | \mathbf{x} \rangle \equiv \int_{Y_k} t_k p(t_k | \mathbf{x}) dt_k \quad \text{and} \quad \langle t_k^2 | \mathbf{x} \rangle \equiv \int_{Y_k} t_k^2 p(t_k | \mathbf{x}) dt_k \quad (11.14)$$

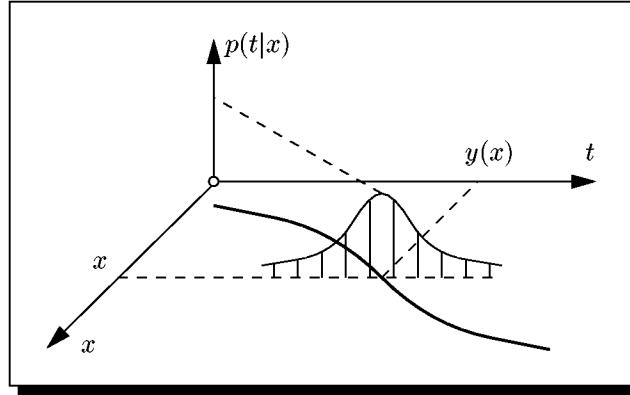
Then, by using the above definitions, it's possible to write:

$$\begin{aligned} [y_k - t_k]^2 &= [y_k - \langle t_k | \mathbf{x} \rangle + \langle t_k | \mathbf{x} \rangle - t_k]^2 \\ &= [y_k - \langle t_k | \mathbf{x} \rangle]^2 + 2(y_k - \langle t_k | \mathbf{x} \rangle)(\langle t_k | \mathbf{x} \rangle - t_k) + [\langle t_k | \mathbf{x} \rangle - t_k]^2 \end{aligned}$$

and by replacing into the expression of  $E$ , the middle term cancels after integration over  $t_k$  ( $\langle t_k | \mathbf{x} \rangle - t_k \rightarrow \langle t_k | \mathbf{x} \rangle - \langle t_k | \mathbf{x} \rangle = 0$ )

$$E = \frac{1}{2} \sum_{k=1}^K \int_X [y_k(\mathbf{x}, W) - \langle t_k | \mathbf{x} \rangle]^2 p(\mathbf{x}) d\mathbf{x} + \frac{1}{2} \sum_{k=1}^K \int_X [\langle t_k^2 | \mathbf{x} \rangle - \langle t_k | \mathbf{x} \rangle^2] p(\mathbf{x}) d\mathbf{x} \quad (11.15)$$

(upon integration over  $t_k$ : the first term is independent of  $p(t_k | \mathbf{x})$  and  $\int_{Y_k} p(t_k | \mathbf{x}) dt_k = 1$  as  $p(t_k | \mathbf{x})$  is assumed normated, while for the second term  $[\langle t_k | \mathbf{x} \rangle - t_k]^2 \rightarrow \langle t_k | \mathbf{x} \rangle^2 - 2\langle t_k | \mathbf{x} \rangle^2 + \langle t_k^2 | \mathbf{x} \rangle$ ).



**Figure 11.1:** The network output significance. Unidimensional input and output space. Note that  $\langle t|x \rangle$  doesn't necessarily coincide with the maximum of  $p(t|x)$ .

In the expression (11.15) of the error function, the second term does not depend upon network parameters  $W$  and may be dropped; the first term is always positive and then  $E \geq 0$ . The error becomes minimum (zero) when the integrand in the first term is zero, i.e.

$$y_k(\mathbf{x}, W^*) = \langle t_k | \mathbf{x} \rangle \quad (11.16)$$

❖  $W^*$

where  $W^*$  denotes the final weights, after the learning process have been finished (i.e. optimal  $W$  value).

The ANN output with sum-of-squares error function represents the average of target conditioned on input. See also figure 11.1.

To obtain the above result the following assumptions were made:

- The training set is sufficiently large:  $P \rightarrow \infty$ .
- The number of weights ( $w$  parameters) is sufficiently large.
- The absolute minimum of error function was found.



#### 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 who try to minimize the sum-of-squares error function.
- ➔ As  $p(t_k | \mathbf{x})$  is normated then each term in the sum appearing in last expression of  $E$  in (11.13) may be multiplied conveniently by  $\int_{Y_{k'}} p(t_{k'} | \mathbf{x}) dk' = 1$  such that  $E$  may be written as:

$$E = \frac{1}{2} \iint_{X,Y} \|\mathbf{y}(\mathbf{x}_p, W) - \mathbf{t}\|^2 p(\mathbf{t} | \mathbf{x}) p(\mathbf{x}) dt dx$$

( $\{t_k\}$  were assumed statistically independent,  $p(\mathbf{t} | \mathbf{x}) = \prod_{k=1}^K p(t_k | \mathbf{x})$ ) and then the

proof of (11.16):

$$y(\mathbf{x}, W) = \langle \mathbf{t} | \mathbf{x} \rangle$$

may be expressed similarly as above but directly in vectorial terms.

The same result may be obtained faster by considering the functional derivative of  $E$  with respect to  $y_k(\mathbf{x})$  which is set to zero to find its minima:

$$\frac{\delta E}{\delta y_k(\mathbf{x})} = \iint_{X, Y_k} [y_k(\mathbf{x}, W) - t_k] p(t_k | \mathbf{x}) p(\mathbf{x}) dt_k d\mathbf{x} = 0$$

The right term may be split in two integrals,  $y_k(\mathbf{x}, W)$  is independent of  $t_k$ ,  $p(t_k | \mathbf{x})$  is normated and then:

$$y_k(\mathbf{x}, W) = \int_{Y_k} t_k p(t_k | \mathbf{x}) dt_k = \langle t_k | \mathbf{x} \rangle$$

(the integrand of  $\int_X$  have to be zero).

In general the result (11.16) represents the optimal solution: assuming that the data is generated from a set of deterministic functions  $h_k(\mathbf{x})$  with superimposed zero-mean noise then:

$$t_k = h_k(\mathbf{x}) + \varepsilon_k \quad \Rightarrow$$

$$y_k(\mathbf{x}) = \langle t_k | \mathbf{x} \rangle = \langle h_k(\mathbf{x}) + \varepsilon_k | \mathbf{x} \rangle = h_k(\mathbf{x})$$

The variance of the target data, as function of  $\mathbf{x}$ , is:

$$\sigma_k^2(\mathbf{x}) = \int_{Y_k} [t_k - \langle t_k | \mathbf{x} \rangle]^2 p(t_k | \mathbf{x}) dt_k = \langle t_k^2 | \mathbf{x} \rangle - \langle t_k | \mathbf{x} \rangle^2 \quad (11.17)$$

$([t_k - \langle t_k | \mathbf{x} \rangle]^2 \rightarrow \langle t_k^2 | \mathbf{x} \rangle - 2\langle t_k | \mathbf{x} \rangle^2 + \langle t_k | \mathbf{x} \rangle^2)$  i.e. is exactly the residual term of the error function (11.15) and it may be used to assign error bars to the network prediction.



#### Remarks:

- ➔ Using the sum-of-squares error function, the network outputs are  $\mathbf{x}$ -dependent means of the distribution and the average variance is the residual value of the error function at its minimum. Thus the sum-of-squares error function cannot distinguish between the true distribution and a Gaussian distribution with the same  $\mathbf{x}$ -dependent mean and average variance.

#### 11.2.4 Outer product approximation of Hessian

From the definition (11.13) of error function, the Hessian terms are:

$$\frac{\partial^2 E}{\partial w_s \partial w_q} = \sum_{k=1}^K \int_X \frac{\partial y_k}{\partial w_s} \frac{\partial y_k}{\partial w_q} p(\mathbf{x}) d\mathbf{x} + \sum_{k=1}^K \int_X \frac{\partial^2 y_k}{\partial w_s \partial w_q} (y_k - \langle t_k | \mathbf{x} \rangle) p(\mathbf{x}) d\mathbf{x}$$

where  $w_s, w_q$  are some two weights, also the integration over  $t_k$  was performed ( $y_k$  is independent of  $p(t_k|\mathbf{x})$  which is normated, i.e.  $\int_{Y_k} p(t_k|\mathbf{x}) dt_k = 1$ ).

The second term, after integration over  $\mathbf{x}$ , cancels because of the result (11.16), such that the Hessian becomes — after reverting back from the integral to the discrete sum:

$$\frac{\partial^2 E}{\partial w_s \partial w_q} = \frac{1}{P} \sum_{p=1}^P \sum_{k=1}^K \frac{\partial y_k(\mathbf{x}_p, W^*)}{\partial w_s} \frac{\partial y_k(\mathbf{x}_p, W^*)}{\partial w_q}$$

## → 11.3 Minkowski Error

A more general Gaussian distribution for the noise  $\varepsilon$ , see (11.3), is:

$$p(\varepsilon_k) = \frac{R\beta^{1/R}}{2\Gamma_E(1/R)} \exp(-\beta|\varepsilon_k|^R) = p(t_k|\mathbf{x}) \quad , \quad R = \text{const.} \quad (11.18)$$

where  $\Gamma_E$  is the Euler function<sup>1</sup>.

By a similar procedure as used in section 11.2 (and using the likelihood function) the error function becomes:

$$E = \sum_{p=1}^P \sum_{k=1}^K |y_k(\mathbf{x}_p, W) - t_{kp}|^R$$

Minkowski error

which is named *Minkowski error*.

The derivative of the error function, with respect to the weights, is

$$\frac{\partial E}{\partial w_s} = \sum_{p=1}^P \sum_{k=1}^K |y_k(\mathbf{x}_p, W) - t_{kp}|^{R-1} \text{sign}(y_k(\mathbf{x}_p, W) - t_{kp}) \frac{\partial y_k(\mathbf{x}_p, W)}{\partial w_s}$$

which may be evaluated by the means of backpropagation algorithm ( $w_s$  being here some weight).

### Remarks:

- ➔ The constant in front of  $\exp$  function in (11.18) ensures the normalization of probability density:  $\int_{Y_k} p(\varepsilon_k) d\varepsilon_k = 1$ .
- ➔ Obviously, for  $R = 2$  it reduces to the Gaussian distribution. For  $R = 1$  the distribution is called *Laplacian* and the corresponding error function *city-block metric*.

$L_R$  norm

More generally the distance  $|y - t|^R$  is named  $L_R$  norm.

---

<sup>11.3</sup>See [Bis95] pp. 208–210.

<sup>1</sup>See the mathematical appendix.

► The use of  $R < 2$  reduces the sensitivity to outliers. E.g. by considering a network with one output and  $R = 1$  then:

$$E = \sum_{p=1}^P |y(\mathbf{x}_p, W) - t_p|$$

and, at minimum, the derivative is zero:

$$\sum_{p=1}^P \text{sign}(y(\mathbf{x}_p, W) - t_p) = 0$$

condition which is satisfied if there are an equal number of points  $t_p > y$  as there are  $t_p < y$ , *irrespective of the value of difference*.

## 11.4 Input-dependent Variance

Considering the variance as depending on the input vector  $\sigma_k = \sigma_k(\mathbf{x})$  and a Gaussian distribution of the noise then, from (11.4):

$$p(t_k|\mathbf{x}) = \frac{1}{\sqrt{2\pi}\sigma_k(\mathbf{x})} \exp\left(-\frac{[y_k(\mathbf{x}, W) - t_k]^2}{2\sigma_k^2(\mathbf{x})}\right)$$

By the same means as used in section 11.2 for the sum-of-squares function (by using the likelihood function), the error may be written as:

$$E = \sum_{p=1}^P \sum_{k=1}^K \left[ \frac{[y_k(\mathbf{x}_p, W) - t_{kp}]^2}{2\sigma_k^2(\mathbf{x}_p)} + \ln \sigma_k(\mathbf{x}_p) \right]$$

Dividing by  $1/P$  and considering the limit  $P \rightarrow \infty$  (infinite training set) then:

$$E = \sum_{k=1}^K \iint_{X, Y_k} \left[ \frac{[y_k(\mathbf{x}, W) - t_k]^2}{2\sigma_k^2(\mathbf{x})} + \ln \sigma_k(\mathbf{x}) \right] p(t_k|\mathbf{x}) p(\mathbf{x}) dt_k d\mathbf{x}$$

and, the condition of minima with respect to  $y_k$  is:

$$\frac{\delta E}{\delta y_k} = p(\mathbf{x}) \int_{Y_k} \frac{y_k(\mathbf{x}, W) - t_k}{\sigma_k^2(\mathbf{x})} p(t_k|\mathbf{x}) dt_k = 0$$

which means that the output of network, after the training was done, is:

$$y_k(\mathbf{x}, W^*) = \langle t_k | \mathbf{x} \rangle$$

Similarly, the condition of minima for  $E$  with respect to  $\sigma_k$  is:

$$\frac{\delta E}{\delta \sigma_k(\mathbf{x})} = p(\mathbf{x}) \int_{Y_k} \left[ \frac{1}{\sigma_k(\mathbf{x})} - \frac{[y_k(\mathbf{x}, W) - t_k]^2}{\sigma_k^3(\mathbf{x})} \right] p(t_k|\mathbf{x}) dt_k = 0$$

---

<sup>11.4</sup>See [Bis95] pp. 211–212.

and, after the training ( $E$  minimum), the variance is (see also (11.17)):

$$\sigma_k^2(\mathbf{x}) = \langle [t_k - \langle t_k | \mathbf{x} \rangle]^2 | \mathbf{x} \rangle$$

The above result may be used to find the variance as follows:

- First a network is trained to minimize the sum-of-squares error function, using the  $\{\mathbf{x}_p, \mathbf{t}_p\}_{p=1,P}$  training set.
- The outputs of the above network are  $y_k = \langle t_k | \mathbf{x} \rangle$ . These values are subtracted from the target values  $t_k$  and the result is squared. Together with the input vectors  $\mathbf{x}_p$  they form a new training set  $\{\mathbf{x}_p, (t_p - \langle t_p | \mathbf{x} \rangle)^2\}_{p=1,P}$ .
- The new set is used to train a new network with a sum-of-squares error function. The outputs of this network are  $\sigma_k^2 = \langle [t_k - \langle t_k | \mathbf{x} \rangle]^2 | \mathbf{x} \rangle$ .

## 11.5 Modeling Conditional Distributions

A very general framework of modeling conditional distributions is to build a model in two stages:

- The first stage uses the input vectors  $\mathbf{x}_p$  to model — through an ANN — some parameters  $\theta(\mathbf{x})$ .
- The  $\theta$  parameters are used into a parametric model (non ANN) to find the conditional probability density  $p(\mathbf{t}|\mathbf{x})$ .

### Remarks:

- The above approach may deal well with complex distributions. By comparison, a neural network with sum-of-squares error function may model just Gaussian distributions with a global variance parameter and a  $\mathbf{x}$ -dependent mean.

❖  $M$

As parametric model, a good choice is the mixture model. In this approach, the distribution of  $p(\mathbf{x})$  is considered of the form:  $p(\mathbf{x}) = \sum_{m=1}^M p(\mathbf{x}|m) P(m)$  where  $M$  is the number of mixture components  $m$ .

On similar grounds, the probability distribution  $p(\mathbf{t}|\mathbf{x})$  may be expressed as:

$$p(\mathbf{t}|\mathbf{x}) = \sum_{m=1}^M \alpha_m(\mathbf{x}) \varphi_m(\mathbf{t}|\mathbf{x}) \quad (11.19)$$

❖  $\alpha_m$

where  $\alpha_m(\mathbf{x})$  are prior probabilities, conditioned on  $\mathbf{x}$ , of the target vector  $\mathbf{t}$  being generated by the  $m$ -th component of the mixture. Being probabilities they have to satisfy the constraint of normality:

$$\sum_{m=1}^M \alpha_m(\mathbf{x}) = 1 \quad \text{and} \quad \alpha_m(\mathbf{x}) \in [0, 1]$$

---

<sup>11.5</sup>See [Bis95] pp. 212–222.

The kernel functions  $\varphi_m(\mathbf{t}|\mathbf{x})$  represents the conditional density of the target vector  $\mathbf{t}$  for the  $m$ -th mixture component. One possibility is to choose them to be of Gaussian type:

$$\varphi_m(\mathbf{t}|\mathbf{x}) = \frac{1}{(2\pi)^{K/2}\sigma_m^K(\mathbf{x})} \exp\left(-\frac{\|\mathbf{t} - \mu_m(\mathbf{x})\|^2}{2\sigma_m^2(\mathbf{x})}\right) \quad (11.20)$$

and then the outputs of the neural network may be defined as follows:

- A set of outputs  $\{y_{\alpha m}\}$  for the  $\alpha_m$  parameters which will be calculated through a *softmax function*:

$$\alpha_m = \frac{\exp(y_{\alpha m})}{\sum_{n=1}^M \exp(y_{\alpha n})} \quad (11.21)$$

$\diamond \varphi_m$   
 $y_{\alpha m}, y_{\sigma m},$   
 $y_{\mu km}$   
softmax function

- A set of outputs  $\{y_{\sigma m}\}$  for the  $\sigma_m$  parameters which will be calculated through the exponential function:

$$\sigma_m = \exp(y_{\sigma m}) \quad (11.22)$$

- A set of outputs  $\{y_{\mu km}\}$  for the  $\mu_m$  parameters which will be represented directly by the network outputs:

$$\mu_{km} = y_{\mu km}$$

The error function is build from the likelihood:

$$E = -\ln \mathcal{L} = -\ln \left( \prod_{p=1}^P p(\mathbf{t}_p|\mathbf{x}_p) \right) = -\sum_{p=1}^P \ln \left( \sum_{m=1}^M \alpha_m(\mathbf{x}_p) \varphi_m(\mathbf{t}_p|\mathbf{x}_p) \right)$$

The problem is to find the network weights in order to minimize the error function. This is done by finding an expression for the derivatives of  $E$  with respect to network outputs  $y_{(\cdot)}$  and then the weights are updated by using the backpropagation algorithm. The error function will be considered as a sum of  $P$  components  $E_p$ :

$$E_p = -\ln \left( \sum_{m=1}^M \alpha_m(\mathbf{x}_p) \varphi_m(\mathbf{t}_p|\mathbf{x}_p) \right) \quad (11.23)$$

The posterior probability that the pair  $(\mathbf{x}, \mathbf{t})$  was generated by the component  $m$  of the mixture is:

$$\pi_m(\mathbf{x}, \mathbf{t}) = \frac{\alpha_m(\mathbf{x}) \varphi_m(\mathbf{t}|\mathbf{x})}{\sum_{n=1}^M \alpha_n(\mathbf{x}) \varphi_n(\mathbf{t}|\mathbf{x})}$$

and, obviously, is normated ( $\sum_{m=1}^M \pi_m = 1$ ).

$\diamond E_p$

$\diamond \pi_m$

**The  $\frac{\partial E}{\partial y_{\alpha m}}$  derivatives.**

From (11.23):  $\frac{\partial E_p}{\partial \alpha_n} = -\frac{\pi_n}{\alpha_n}$  and from (11.21):

$$\frac{\partial \alpha_n}{\partial y_{\alpha m}} = \delta_{nm} \alpha_n - \alpha_n \alpha_m$$

( $\delta_{nm}$  being the Kronecker symbol).  $E_p$  depends upon  $y_{\alpha m}$  through all  $\alpha_n$  parameters, then:

$$\frac{\partial E_p}{\partial y_{\alpha m}} = \sum_{n=1}^M \frac{\partial E_p}{\partial \alpha_n} \frac{\partial \alpha_n}{\partial y_{\alpha m}} = \alpha_m - \pi_m$$

(by using the property of normation for  $\pi_k$  as well).

**The  $\frac{\partial E}{\partial y_{\sigma m}}$  derivatives.**

From (11.23) and (11.20):

$$\frac{\partial E_p}{\partial \sigma_m} = -\pi_m \left( \frac{\|\mathbf{t} - \boldsymbol{\mu}_m\|^2}{\sigma_m^3} - \frac{K}{\sigma_m} \right)$$

Also from (11.22)  $\frac{d\sigma_m}{dy_{\sigma m}} = \sigma_m$  and then:

$$\frac{\partial E_p}{\partial y_{\sigma m}} = \frac{\partial E_p}{\partial \sigma_m} \frac{d\sigma_m}{dy_{\sigma m}} = -\pi_m \left( \frac{\|\mathbf{t} - \boldsymbol{\mu}_m\|^2}{\sigma_m^2} - K \right)$$

**The  $\frac{\partial E}{\partial y_{\mu km}}$  derivatives.**

Since  $\mu_{km} = y_{\mu km}$ , then from (11.23) and (11.20) (Euclidean norm):

$$\frac{\partial E_p}{\partial y_{\mu km}} = \frac{\partial E_p}{\partial \mu_{km}} = \pi_m \frac{\mu_{km} - t_k}{\sigma_m^2}$$

The conditional average of the target data is:

$$\langle \mathbf{t} | \mathbf{x} \rangle = \int_Y \mathbf{t} p(\mathbf{t} | \mathbf{x}) d\mathbf{t} = \sum_{m=1}^M \alpha_m(\mathbf{x}) \int_Y \mathbf{t} \varphi_m(\mathbf{t} | \mathbf{x}) d\mathbf{t} = \sum_{m=1}^M \alpha_m(\mathbf{x}) \boldsymbol{\mu}_m(\mathbf{x}) \quad (11.24)$$

(from (11.19) and (11.20); it was also assumed that  $Y = \mathbb{R}^K$ , see Gaussian integrals in the mathematical appendix).

❖  $s$

The variance  $s(\mathbf{x})$  is:

$$s^2(\mathbf{x}) \equiv \langle \|\mathbf{t} - \langle \mathbf{t} | \mathbf{x} \rangle\|^2 | \mathbf{x} \rangle = \sum_{m=1}^M \alpha_m(\mathbf{x}) \left[ \sigma_m^2(\mathbf{x}) + \left\| \sum_{n=1}^M \alpha_n(\mathbf{x}) \boldsymbol{\mu}_n(\mathbf{x}) - \boldsymbol{\mu}_m(\mathbf{x}) \right\|^2 \right]$$

*Proof.* From the definition of average and using (11.19) and (11.20) (also: Euclidean norm and  $Y_k = \mathbb{R}$ ):

$$s^2(\mathbf{x}) = \int_{\mathbb{R}^K} \|\mathbf{t} - \langle \mathbf{t} | \mathbf{x} \rangle\|^2 p(\mathbf{t} | \mathbf{x}) d\mathbf{t} = \sum_{k=1}^K \int_{\mathbb{R}} (t_k - \langle t_k | \mathbf{x} \rangle)^2 p(t_k | \mathbf{x}) dt_k$$

$$\begin{aligned}
&= \sum_{k=1}^K \sum_{m=1}^M \alpha_m(\mathbf{x}) \int_{\mathbb{R}} (t_k - \langle t_k | \mathbf{x} \rangle)^2 \varphi_m(t_k | \mathbf{x}) dt_k \\
&= \sum_{k=1}^K \sum_{m=1}^M \frac{\alpha_m(\mathbf{x})}{(2\pi)^{K/2} \sigma_m^K(\mathbf{x})} \int_{\mathbb{R}} (t_k - \langle t_k | \mathbf{x} \rangle)^2 \exp \left\{ -\frac{[t_k - \mu_{km}(\mathbf{x})]^2}{2\sigma_m^2(\mathbf{x})} \right\} dt_k
\end{aligned}$$

To calculate the Gaussian integral above first make a change of variable  $\tilde{t}_k = t_k - \langle t_k | \mathbf{x} \rangle$  and then a second one  $\hat{t}_k = \tilde{t}_k + \langle t_k | \mathbf{x} \rangle - \mu_{km}(\mathbf{x})$  forcing also a squared  $\hat{t}_k$ . This leads to:  $\diamond \tilde{t}_k, \hat{t}_k$

$$\begin{aligned}
&\int_{\mathbb{R}} (t_k - \langle t_k | \mathbf{x} \rangle)^2 \exp \left\{ -\frac{[t_k - \mu_{km}(\mathbf{x})]^2}{2\sigma_m^2(\mathbf{x})} \right\} dt_k = \int_{\mathbb{R}} \tilde{t}_k \exp \left\{ -\frac{(\tilde{t}_k + \langle t_k | \mathbf{x} \rangle - \mu_{km}(\mathbf{x}))^2}{2\sigma_m^2(\mathbf{x})} \right\} d\tilde{t}_k \\
&= \int_{\mathbb{R}} \hat{t}_k \exp \left( -\frac{\hat{t}_k^2}{2\sigma_m^2(\mathbf{x})} \right) d\hat{t}_k - (\langle t_k | \mathbf{x} \rangle - \mu_{km})^2 \int_{\mathbb{R}} \exp \left( -\frac{\hat{t}_k^2}{2\sigma_m^2(\mathbf{x})} \right) d\hat{t}_k \\
&\quad - 2(\langle t_k | \mathbf{x} \rangle - \mu_{km}) \int_{\mathbb{R}} \hat{t}_k \exp \left( -\frac{\hat{t}_k^2}{2\sigma_m^2(\mathbf{x})} \right) d\hat{t}_k
\end{aligned}$$

First term, after an integration by parts, leads to a Gaussian (see mathematical appendix) and equals  $\sqrt{2\pi} \sigma_m^3(\mathbf{x})$ ; the second one is directly a Gaussian giving  $\sqrt{2\pi} \sigma_m^2(\mathbf{x})$  while in the third term the integrand is an odd function, integrated over a domain symmetrical around origin, so it is zero. The sought result is obtained after the (11.24) replacement.  $\square$

## 11.6 Classification using Sum-of-Squares

If the sum-of-squares is used as error function then the outputs of neurons represent the conditional average of the target data  $\mathbf{y} = \langle \mathbf{t} | \mathbf{x} \rangle$  as proven in (11.16). In problems of classification the  $\mathbf{t}$  vectors represent the class labels. Then the most simple way of labeling is to assign a network output to each class and to consider that an input vector  $\mathbf{x}$  is belonging to that class  $\mathcal{C}_k$  represented by the (output) neuron  $k$  with the biggest output. This means that the target for a vector  $\mathbf{x} \in \mathcal{C}_k$  is  $\{t_k = \delta_{kq}\}_{q=1,K}$  ( $\delta_{kq}$  being the Kronecker symbol). This method is named one-of- $k$  encoding. The  $p(t_k | \mathbf{x})$  probability may be expressed as:

$$p(t_k | \mathbf{x}) = \sum_{q=1}^K \delta_D(t_k - \delta_{kq}) P(\mathcal{C}_q | \mathbf{x})$$

( $\delta_D$  being the Dirac function and  $P(\mathcal{C}_q | \mathbf{x})$  is the probability of the  $\mathbf{x} \in \mathcal{C}_q$  event, for a given  $\mathbf{x}$ ).

From (11.16) and the above expression of  $p(t_k | \mathbf{x})$ :

$$y_k(\mathbf{x}) = \sum_{q=1}^K \int_{Y_k} \delta_{kq} \delta_D(t_k - \delta_{kq}) P(\mathcal{C}_q | \mathbf{x}) dt_k = P(\mathcal{C}_k | \mathbf{x})$$

i.e. when using the one-of- $k$  encoding the network outputs represent the posterior probability.

---

<sup>11.6</sup>See [Bis95] pp. 225–230.

The outputs of network being probabilities must sum to unity and be in the range  $[0, 1]$ :

- By the way the targets are chosen (1-of- $k$ ) and the linear sum rules (see section 11.2.2) the outputs of the network will sum to unity.
- The range of the outputs may be ensured by the way the activation function is chosen (e.g. logistic signal activation function).

### 11.6.1 Hidden Neurons

The error function (11.9) may be written in a matrix form:

$$E = \frac{1}{2} \text{Tr}\{(W_{\text{out}} \tilde{Z} - \tilde{T})^T (W_{\text{out}} \tilde{Z} - \tilde{T})\} \quad (11.25)$$

❖  $S_B, S_T$

and, by replacing the solution (11.6):

$$E = \frac{1}{2} \text{Tr}(\tilde{T}^T \tilde{T} - S_B S_T^{-1}) \quad \text{where } S_B = \tilde{Z} \tilde{T}^T \tilde{T} \tilde{Z}^T, S_T = \tilde{Z} \tilde{Z}^T \quad (11.26)$$

*Proof.* As  $\tilde{Z}^\dagger = \tilde{Z}^T (\tilde{Z} \tilde{Z}^T)$  and for two matrices  $(AB)^T = (B^T A^T)$  then:

$$\begin{aligned} E &= \frac{1}{2} \text{Tr} \left\{ (\tilde{T} \tilde{Z}^\dagger \tilde{Z} - \tilde{T})^T (\tilde{T} \tilde{Z}^\dagger \tilde{Z} - \tilde{T}) \right\} = \frac{1}{2} \text{Tr} \left\{ (\tilde{Z}^T \tilde{Z}^\dagger T \tilde{T}^T - \tilde{T}^T) (\tilde{T} \tilde{Z}^\dagger \tilde{Z} - \tilde{T}) \right\} \\ &= \frac{1}{2} \text{Tr} \left\{ [\tilde{Z}^T (\tilde{Z} \tilde{Z}^T)^{-1} \tilde{Z} \tilde{T}^T - \tilde{T}^T] [\tilde{T} \tilde{Z}^T (\tilde{Z} \tilde{Z}^T)^{-1} \tilde{Z} - \tilde{T}] \right\} \\ &= \frac{1}{2} \text{Tr} \left\{ \tilde{Z}^T (\tilde{Z} \tilde{Z}^T)^{-1} \tilde{Z} \tilde{T}^T \tilde{T} \tilde{Z}^T (\tilde{Z} \tilde{Z}^T)^{-1} \tilde{Z} - \tilde{Z}^T (\tilde{Z} \tilde{Z}^T)^{-1} \tilde{Z} \tilde{T}^T \tilde{T} \tilde{Z}^T (\tilde{Z} \tilde{Z}^T)^{-1} \tilde{Z} + \tilde{T}^T \tilde{T} \right\} \end{aligned}$$

For two matrices  $A$  and  $B$ , such that  $B$  have same dimensions as  $A^T$ , it is true that  $\text{Tr}(AB) = \text{Tr}(BA)$ . This property is used in first and third terms, by moving  $\tilde{Z}$  from left to right, thus they become identical and cancel out. As  $\text{Tr}(A^T) = \text{Tr}(A)$ , the second term is transposed (again  $(AB)^T = B^T A^T$ ) and then,  $\tilde{Z}$  is moved from left to right:

$$E = \frac{1}{2} \text{Tr} \left\{ -\tilde{Z} \tilde{T}^T \tilde{T} \tilde{Z}^T (\tilde{Z} \tilde{Z}^T)^{-1} + \tilde{T}^T \tilde{T} \right\} \quad \square$$

Minimizing the error becomes equivalent to maximize the expression:

$$J = \frac{1}{2} \text{Tr} (S_B S_T^{-1}) \quad (11.27)$$

The  $S_T$  matrix may be written as (see the definition of  $\tilde{Z}$ ):

$$S_T = \tilde{Z} \tilde{Z}^T = \sum_{p=1}^P [\mathbf{z}(\mathbf{x}_p) - \langle \mathbf{z} \rangle][\mathbf{z}(\mathbf{x}_p) - \langle \mathbf{z} \rangle]^T \quad (11.28)$$

i.e. it represents the total covariance matrix of (last) hidden layer with respect to the training set.

❖  $P_k, \langle \mathbf{z} \rangle_{C_k}$

Let  $P_k$  be the number of  $\mathbf{x}_p \in C_k$  and  $\langle \mathbf{z} \rangle_{C_k} = \frac{1}{P_k} \sum_{\mathbf{x}_p \in C_k} \mathbf{z}(\mathbf{x}_p)$  (i.e. the mean of hidden neurons output over the training vectors belonging to a single class). Then:

$$S_B = \sum_{k=1}^K P_k^2 (\langle \mathbf{z} \rangle_{C_k} - \langle \mathbf{z} \rangle)(\langle \mathbf{z} \rangle_{C_k} - \langle \mathbf{z} \rangle)^T \quad (11.29)$$

*Proof.*  $S_B = \tilde{Z}\tilde{T}^T\tilde{T}\tilde{Z}^T = (\tilde{T}\tilde{Z}^T)^T(\tilde{T}\tilde{Z}^T)$  (as  $(AB)^T = B^T A^T$ ). Considering the one-of- $k$  encoding then  $\langle t_k \rangle = \frac{1}{P} \sum_{p=1}^P t_{kp} = \frac{P_k}{P}$ , also  $\langle z_j \rangle = \frac{1}{P} \sum_{p=1}^P z_j(\mathbf{x}_p)$  and then:

$$\tilde{t}_{kp} = t_{kp} - \frac{P_k}{P} \quad \text{and} \quad \tilde{Z}^T(p, j) = z_j(\mathbf{x}_p) - \frac{1}{P} \sum_{p=1}^P z_j(\mathbf{x}_p)$$

Each element of  $\tilde{T}\tilde{Z}^T$  is calculated directly, note that  $\sum_{p=1}^P t_{kp}z_j(\mathbf{x}_p) = P_k \langle z_j \rangle_{C_k}$  due to one-of- $k$  encoding:

$$(\tilde{T}\tilde{Z}^T)(k, j) = \sum_{p=1}^P \left[ t_{kp} - \frac{P_k}{P} \right] \left[ z_j(\mathbf{x}_p) - \frac{1}{P} \sum_{p'=1}^P z_j(\mathbf{x}_{p'}) \right] = P_k (\langle z_j \rangle_{C_k} - \langle z_j \rangle)$$

The final result is obtained by doing the  $(\tilde{T}\tilde{Z}^T)^T(\tilde{T}\tilde{Z}^T)$  multiplication.  $\square$

The processing on hidden neuron layer(s) may be seen as non-linear transformation such that (11.27) is maximized. The (11.27) expression have a strong resemblance with the Fisher criterion. *The output of (last) hidden layer have the role to generate maximum discrimination between classes, optimum for a linear transformation (performed by the output layer).*



#### Remarks:

- ➔ Note that  $S_B$  contains the multiplicative factor  $P_k^2$  under the sum, fact which strongly raises the importance of classes well represented *in the training set*, see section 11.6.2.

### 11.6.2 Weighted Sum-of-Squares

For the sum-of-squares networks, the minimization of the error function results into a maximisation of the  $J$  term (see (11.27)) which for the one-of- $k$  encoding is dependent upon the *observed* prior probabilities  $\tilde{P}(C_k) = P_k/P$  of the training set. If these prior probabilities differ in the training set from the *true*  $P(C_k)$  then the (trained) classifier (model) built will be suboptimal.

❖  $P(C_k), \tilde{P}(C_k)$

To correct the above problem the error function may be modified by inserting a weighting factor  $\kappa_p$  for each pattern; the error becoming:

❖  $\kappa_p$

$$E = \frac{1}{2} \sum_{p=1}^P \kappa_p \|\mathbf{y}(\mathbf{x}_p) - \mathbf{t}_p\|^2 \quad \text{where} \quad \kappa_p = \frac{P(C_k)}{\tilde{P}(C_k)} \text{ for } \mathbf{x}_p \in C_k \quad (11.30)$$



#### Remarks:

- ➔ The prior probabilities  $P(C_k)$  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 the adjusting coefficients  $\kappa_p$  may be even estimated by other means.

Considering the identity activation function  $f(\mathbf{a}) = \mathbf{a}$  then:

$$\frac{\partial E}{\partial a_k} = \frac{\partial E}{\partial y_k} \frac{df}{da_k} = \frac{\partial E}{\partial y_k} = \sum_{p=1}^P \kappa_p [y_k(\mathbf{x}_p) - t_{kp}]$$

where (bias was considered):

$$y_k = \sum_{q=1}^K w_{kj} z_j + w_{k0} \quad (11.31)$$

Then the condition of minima of  $E$  with respect to  $w_{k0}$  is:

$$\frac{\partial E}{\partial w_{k0}} = \frac{\partial E}{\partial y_k} \frac{\partial y_k}{\partial w_{k0}} = \sum_{p=1}^P \kappa_p \left( \sum_{j=1}^H w_{kj} z_j(\mathbf{x}_p) + w_{k0} - t_{kp} \right) = 0$$

❖  $\langle t_k \rangle, \langle z_j \rangle$

the solution being:

$$w_{k0} = \langle t_k \rangle - \sum_{j=1}^H w_{kj} \langle z_j \rangle \quad ; \quad \langle t_k \rangle = \frac{\sum_{p=1}^P \kappa_p t_{kp}}{\sum_{p=1}^P \kappa_p} \quad , \quad \langle z_j \rangle = \frac{\sum_{p=1}^P \kappa_p z_j(\mathbf{x}_p)}{\sum_{p=1}^P \kappa_p} \quad (11.32)$$

From (11.30), using (11.31) and (11.32), by same reasoning as for (11.9) and with the same meaning for  $\tilde{z}_{jp}$  and  $\tilde{t}_{kp}$ :

$$E = \frac{1}{2} \sum_{p=1}^P \kappa_p \sum_{k=1}^K \left( \sum_{j=1}^H w_{kj} \tilde{z}_{jp} - \tilde{t}_{kp} \right)^2 = \frac{1}{2} \sum_{p=1}^P \sum_{k=1}^K \left( \sum_{j=1}^H w_{kj} \tilde{z}_{jp} \sqrt{\kappa_p} - \tilde{t}_{kp} \sqrt{\kappa_p} \right)^2$$

❖  $K$

The matrix of  $\kappa_p$  coefficients is build as:

$$K = \begin{pmatrix} \sqrt{\kappa_1} & 0 & \cdots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \sqrt{\kappa_P} \end{pmatrix} \quad \Rightarrow \quad K^T K = K K^T = \begin{pmatrix} \kappa_1 & 0 & \cdots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \kappa_P \end{pmatrix}$$

and then, by the same reasoning as for (11.25):

$$E = \frac{1}{2} \text{Tr} \left[ \left( W_{\text{out}} \tilde{Z}' - \tilde{T}' \right)^T \left( W_{\text{out}} \tilde{Z}' - \tilde{T}' \right) \right] \quad (11.33)$$

❖  $\tilde{Z}', \tilde{T}'$

where  $\tilde{Z}' = \tilde{Z} K$  and  $\tilde{T}' = \tilde{T} K$ .

On the other hand, the condition of minimum for  $E$ , relative to the weights, is:

$$\begin{aligned} \frac{\partial E}{\partial w_{kj}} &= \sum_{p=1}^P \kappa_p \left( \sum_{q=1}^H w_{kq} \tilde{z}_{qp} - \tilde{t}_{kp} \right) \tilde{z}_{jp} \\ &= \sum_{p=1}^P \kappa_p \left( \sum_{q=1}^H w_{kq} \tilde{z}_{qp} \sqrt{\kappa_p} - \tilde{t}_{kp} \sqrt{\kappa_p} \right) \tilde{z}_{jp} \sqrt{\kappa_p} = 0 \end{aligned}$$

or, in matrix notation, similar to (11.11):

$$W_{\text{out}} \tilde{Z}' \tilde{Z}'^T - \tilde{T}' \tilde{Z}'^T = 0 \quad (11.34)$$

which leads to the solution (see also (11.6) and related calculations: results are similar, by making the substitution  $\tilde{Z} \leftrightarrow \tilde{Z}'$  and  $\tilde{T} \leftrightarrow \tilde{T}'$ ):

$$W_{\text{out}} = \tilde{T}' \tilde{Z}'^\dagger \quad \text{where} \quad \tilde{Z}'^\dagger = \tilde{Z}'^T (\tilde{Z}' \tilde{Z}'^T)^{-1}$$

and the error function may be written as (see (11.26)):

$$E = \frac{1}{2} \text{Tr} \left\{ \tilde{T}'^T \tilde{T}' - S'_B S'_T^{-1} \right\}$$

where  $S'_B = \tilde{Z}' \tilde{T}'^T \tilde{T}' \tilde{Z}'^T$  and  $S'_T = \tilde{Z}' \tilde{Z}'^T$ . ♦  $S'_B, S'_T$

Similar to (11.28) and (11.29), considering the definition of  $\kappa_p$  and the one-of- $k$  encoding, the  $S'_B$  and  $S'_T$  matrices may be written as:

$$\begin{aligned} S'_T &= \sum_{k=1}^K \frac{P(\mathcal{C}_k)}{P_k} \sum_{x_p \in \mathcal{C}_k} [\mathbf{z}(x_p) - \langle \mathbf{z} \rangle][\mathbf{z}(x_p) - \langle \mathbf{z} \rangle]^T \\ S'_B &= \sum_{k=1}^K P^2 P^2(\mathcal{C}_k) (\langle \mathbf{z} \rangle_{\mathcal{C}_k} - \langle \mathbf{z} \rangle)(\langle \mathbf{z} \rangle_{\mathcal{C}_k} - \langle \mathbf{z} \rangle)^T \end{aligned}$$

*Proof.* The proof for  $S'_B$  is very similar to the proof of (11.29). Note that in one-of- $k$  encoding:

$$\sum_{p=1}^P \kappa_p t_{kp} z_j(x_p) = \frac{P(\mathcal{C}_k)}{\tilde{P}(\mathcal{C}_k)} P_k(z_j)_{\mathcal{C}_k}$$

(of course, assuming that the training set is correctly classified) and so forth for the rest of terms. Also  $\tilde{P}(\mathcal{C}_k) = P_k/P$ . □

### 11.6.3 Loss Matrix

Penalties for misclassification may be introduced in the one-of- $k$  encoding by changing the targets to:

$$t_{kp} = 1 - L_{k\ell} \text{ for } x_p \in \mathcal{C}_\ell \quad \text{where} \quad L_{k\ell} = \begin{cases} 0 & \text{if } \ell = k \\ \in [0, 1] & \text{otherwise} \end{cases}$$

$L$  being the loss matrix. Note that for  $L_{k\ell} = 1 - \delta_{k\ell}$  the situation is reduced to the one-of- $k$  case.

Considering the loss matrix,  $S_B$  becomes:

$$S_B = \sum_{k=1}^K \left[ \sum_{\ell=1}^K P_\ell (1 - L_{k\ell}) (\langle \mathbf{z} \rangle_{\mathcal{C}_\ell} - \langle \mathbf{z} \rangle) \right] \left[ \sum_{\ell'=1}^K P_{\ell'} (1 - L_{k\ell'}) (\langle \mathbf{z} \rangle_{\mathcal{C}_{\ell'}} - \langle \mathbf{z} \rangle) \right]^T$$

*Proof.* Same as for proof of (11.29):

$$\sum_{p=1}^P t_{kp} z_j(x_p) = \sum_{\ell=1}^K \sum_{x_p \in \mathcal{C}_\ell} (1 - L_{k\ell}) z_j(x_p) = \sum_{\ell=1}^K P_\ell (1 - L_{k\ell}) \langle z_j \rangle_{\mathcal{C}_\ell}$$

and so forth. □

## → 11.7 Cross Entropy

### 11.7.1 Two Classes Case

Considering a two classes problem, a one network output is discussed, such that the target is either  $t = 1$ , for pattern vectors belonging to  $\mathcal{C}_1$ , or otherwise  $t = 0$  ( $\mathbf{x} \in \mathcal{C}_2$ ), i.e. the network output represents the posterior probabilities  $P(\mathcal{C}_1|\mathbf{x}) = y(\mathbf{x})$  and  $P(\mathcal{C}_2|\mathbf{x}) = 1 - y(\mathbf{x})$ . Then  $p(t|\mathbf{x})$  may be written as:

$$p(t|\mathbf{x}) = y^t(\mathbf{x}_p)[1 - y(\mathbf{x}_p)]^{1-t} \quad (11.35)$$

and for the whole training set — giving the likelihood:

$$\mathcal{L} = \prod_{p=1}^P p(t_p|\mathbf{x}_p) = \prod_{p=1}^P y^{t_p}(\mathbf{x}_p)[1 - y(\mathbf{x}_p)]^{1-t_p}$$

The error function may be taken as the negative logarithm of the likelihood function (as previously discussed):

$$E = -\ln \mathcal{L} = -\sum_{p=1}^P t_p \ln y(\mathbf{x}_p) + (1 - t_p) \ln[1 - y(\mathbf{x}_p)] \quad (11.36)$$

**cross-entropy**

also known as *cross-entropy*.

The error minima, with respect to  $y(\mathbf{x}_p)$ , is found by zeroing its derivatives:

$$\frac{\partial E}{\partial y(\mathbf{x}_p)} = \frac{t_p - y(\mathbf{x}_p)}{y(\mathbf{x}_p)[1 - y(\mathbf{x}_p)]}$$

❖  $E_{\min}$

the minimum occurring for  $y(\mathbf{x}_p) = t_p$ ,  $\forall p \in \{1, \dots, P\}$ ; the value being:

$$E_{\min} = -\sum_{p=1}^P t_p \ln t_p + (1 - t_p) \ln(1 - t_p) \quad (11.37)$$

#### Remarks:

► Considering the logistic activation function  $y = f(a)$ :

$$f(a) = \frac{1}{1 + e^{-a}} \quad , \quad \frac{df}{da} = f(a)[1 - f(a)]$$

❖  $a(\mathbf{x}_p)$

then the derivative of error with respect to  $a(\mathbf{x}_p)$  (total input to output neuron when presented with  $\mathbf{x}_p$ ) is:

$$\frac{\partial E}{\partial a(\mathbf{x}_p)} = \frac{\partial E}{\partial y(\mathbf{x}_p)} \frac{df(a(\mathbf{x}_p))}{da(\mathbf{x}_p)} = y(\mathbf{x}_p) - t_p$$

---

<sup>11.7</sup>See [Bis95] pp. 230–240.

For one-of- $k$  encoding, either  $t_p$  or  $1 - t_p$  is 0 and then  $E_{\min.} = 0$ . For the general case, the minimum value (11.37) may be subtracted, from the general expression (11.36) such that  $E$  becomes:

$$E = - \sum_{p=1}^P t_p \ln \frac{y(\mathbf{x}_p)}{t_p} + (1 - t_p) \ln \frac{1 - y(\mathbf{x}_p)}{1 - t_p} \quad (11.38)$$

### 11.7.2 Sigmoidal Activation Functions

Let assume that the posterior probabilities of outputs  $\mathbf{z}$  of hidden neurons, with respect to classes  $\mathcal{C}_k$  ( $k \in \{1, 2\}$ ) is of the exponential general form:

$$p(\mathbf{z}|\mathcal{C}_k) = \exp \left[ A(\boldsymbol{\theta}_k) + B(\mathbf{z}, \varphi) + \boldsymbol{\theta}_k^T \mathbf{z} \right] \quad (11.39)$$

where  $\boldsymbol{\theta}_k$  and  $\varphi$  are some parameters defining the form of distribution and  $A$  and  $B$  are some (fixed) functions.  $\diamond \boldsymbol{\theta}_k, \varphi, A, B$

From the Bayes theorem:

$$\begin{aligned} P(\mathcal{C}_1|\mathbf{z}) &= \frac{p(\mathbf{z}|\mathcal{C}_1)P(\mathcal{C}_1)}{p(\mathbf{z})} = \frac{p(\mathbf{z}|\mathcal{C}_1)P(\mathcal{C}_1)}{p(\mathbf{z}|\mathcal{C}_1)P(\mathcal{C}_1) + p(\mathbf{z}|\mathcal{C}_2)P(\mathcal{C}_2)} \\ &= \frac{1}{1 + \exp(-a)} \quad \text{where } a = \ln \frac{p(\mathbf{z}|\mathcal{C}_1)P(\mathcal{C}_1)}{p(\mathbf{z}|\mathcal{C}_2)P(\mathcal{C}_2)} \end{aligned}$$

By replacing (11.39) in the expression of  $a$  (above)

$\diamond a, \mathbf{w}, w_0$

$$\begin{aligned} P(\mathcal{C}_1|\mathbf{z}) &= \frac{1}{1 + \exp[-(\mathbf{w}^T \mathbf{z} + w_0)]} \quad \text{where } \mathbf{w} = \boldsymbol{\theta}_1 - \boldsymbol{\theta}_2 \\ w_0 &= A(\boldsymbol{\theta}_1) - A(\boldsymbol{\theta}_2) + \ln \frac{P(\mathcal{C}_1)}{P(\mathcal{C}_2)} \end{aligned}$$

i.e. the network output is represented by a logistic sigmoid activation function applied to the weighted sum of hidden neurons outputs (obviously only those connected to the network output).

### 11.7.3 Cross-Entropy Properties

Let  $\varepsilon_p = y(\mathbf{x}_p) - t_p$  be the error for input pattern  $\mathbf{x}_p$ . Then the cross-entropy (11.38)  $\diamond \varepsilon_p$  becomes:

$$E = - \sum_{p=1}^P t_p \ln \left( 1 + \frac{\varepsilon_p}{t_p} \right) + (1 - t_p) \ln \left( 1 - \frac{\varepsilon_p}{1 - t_p} \right) \quad (11.40)$$

and it can be seen that it depends on *relative error* ( $\varepsilon_p/t_p, \varepsilon_p/(1 - t_p)$ ).

#### Remarks:

- The cross-entropy error function will try to minimize the *relative* error and thus giving evenly results for any kind of targets (large or small) while compared with sum-of-squares error function which tries to minimize the *absolute* errors (thus giving better results with large target values).

Let consider binary outputs, i.e.  $t_p = 1$  for  $\mathbf{x}_p \in \mathcal{C}_1$  and  $t_p = 0$  for  $\mathbf{x}_p \in \mathcal{C}_2$ . Then the (11.40) error becomes:

$$E = - \sum_{\mathbf{x}_p \in \mathcal{C}_1} \ln(1 + \varepsilon_p) - \sum_{\mathbf{x}_p \in \mathcal{C}_2} \ln(1 - \varepsilon_p)$$

(by splitting the sum in (11.40) in two, separately for each class).

### Remarks:

- Considering the absolute error  $\varepsilon_p$  small, then  $\ln(1 \pm \varepsilon_p) \simeq \pm \varepsilon_p$ ; also for  $\mathbf{x}_p \in \mathcal{C}_1 \Rightarrow t_p = 1$  and  $y_p \in [0, 1]$ , then, obviously  $\varepsilon_p < 0$ ; similarly, for  $\mathbf{x}_p \in \mathcal{C}_2$ ,  $\varepsilon_p > 0$ , then  $E \simeq \sum_{p=1}^P |\varepsilon_p|$ .

For an infinitely large training set the sum in (11.36) transforms into an integral:

$$E = - \int_X \int_0^1 \{t \ln y(\mathbf{x}) + (1-t) \ln[1-y(\mathbf{x})]\} p(t|\mathbf{x}) p(\mathbf{x}) dt d\mathbf{x}$$

- ❖  $\langle t|\mathbf{x} \rangle$  is independent of  $t$  and then, by integration over  $t$ :

$$E = - \int_X [\langle t|\mathbf{x} \rangle \ln y(\mathbf{x}) + (1 - \langle t|\mathbf{x} \rangle) \ln(1 - y(\mathbf{x}))] p(\mathbf{x}) d\mathbf{x}$$

$$\text{where } \langle t|\mathbf{x} \rangle = \int_0^1 tp(t|\mathbf{x}) dt \quad (11.41)$$

The value of  $y(\mathbf{x})$  for which  $E$  is minimum is found by zeroing its functional derivative:

$$\frac{\delta E}{\delta y} = \int_X \left[ \frac{\langle t|\mathbf{x} \rangle}{y(\mathbf{x})} - \frac{1 - \langle t|\mathbf{x} \rangle}{1 - y(\mathbf{x})} \right] p(\mathbf{x}) d\mathbf{x} = 0$$

and then  $y(\mathbf{x}) = \langle t|\mathbf{x} \rangle$  ( $E = 0$  if and only if the integrand is 0), i.e. *the output of network represents the conditional average of the target data for the given input*.

For the particular encoding scheme chosen for  $t$ , the posterior probabilities  $p(t|\mathbf{x})$  may be written as:

$$p(t|\mathbf{x}) = \delta_D(1-t)P(\mathcal{C}_1|\mathbf{x}) + \delta_D(t)P(\mathcal{C}_2|\mathbf{x}) \quad (11.42)$$

( $\delta_D$  being the Dirac function). Substituting (11.42) in (11.41) and integrating gives  $y(\mathbf{x}) = P(\mathcal{C}_1|\mathbf{x})$  i.e. the output of network is exactly what it was supposed to represent.

### 11.7.4 Multiple Independent Features

So far only one property of the input vectors  $\mathbf{x}$  was present and studied in the network output — its membership to a particular class — property mutually exclusive.

To watch out for multiple, non exclusive, features a network with multiple outputs is required and then  $y_k(\mathbf{x})$  will represent the probability that the  $k$ -th feature is present in  $\mathbf{x}$  input vector.

Assuming independent features then:  $p(\mathbf{t}|\mathbf{x}) = \prod_{k=1}^K p(t_k|\mathbf{x})$ .

The presence or absence of the  $k$ -th feature may be used to classify  $\mathbf{x}$  as belonging to one of 2 classes, e.g.  $\mathcal{C}'_1$  if it does have it and  $\mathcal{C}'_2$  if it doesn't. Then, by the way the meaning of  $y_k$  was chosen, the posterior probabilities  $p(t_k|\mathbf{x})$  may be expressed as in (11.35) and:

$$p(t_k|\mathbf{x}) = y_k^{t_k} (1 - y_k)^{1-t_k} \Rightarrow p(\mathbf{t}|\mathbf{x}) = \prod_{k=1}^K y_k^{t_k} (1 - y_k)^{1-t_k}$$

$$\mathcal{L} = \prod_{p=1}^P p(\mathbf{t}_p|\mathbf{x}_p) = \prod_{p=1}^P \prod_{k=1}^K y_k^{t_{kp}}(\mathbf{x}_p) [1 - y_k(\mathbf{x}_p)]^{1-t_{kp}}$$

In the usual way, the error function is build as the negative logarithm of the likelihood function:

$$E = -\ln \mathcal{L} = -\sum_{p=1}^P \sum_{k=1}^K \{t_{kp} \ln y_k(\mathbf{x}_p) + (1 - t_{kp}) \ln [1 - y_k(\mathbf{x}_p)]\}$$

Because the network outputs are independent then for each  $y_k$  the analysis for one output network applies. Considering this, the error function may be changed the same way as (11.38):

$$E = -\sum_{p=1}^P \sum_{k=1}^K \left[ t_{kp} \ln \frac{y_k(\mathbf{x}_p)}{t_{kp}} + (1 - t_{kp}) \ln \frac{1 - y_k(\mathbf{x}_p)}{1 - t_{kp}} \right]$$

### 11.7.5 Multiple Classes Case

Let consider the problem of classification with a set of mutually exclusive classes  $\{\mathcal{C}_k\}_{k=\overline{1,K}}$ , and the one-of- $k$  encoding scheme, i.e.  $t_{kp} = \delta_{kl}$  for the input vector  $\mathbf{x}_p \in \mathcal{C}_l$ .

The output of (output) neuron  $k$  represents the posterior probability of  $\mathcal{C}_k$ :  $P(\mathcal{C}_k|\mathbf{x}) = y_k(\mathbf{x})$ ; thus the posterior probability density of  $\mathbf{t}_p$  is:  $p(\mathbf{t}_p|\mathbf{x}_p) = \prod_{k=1}^K y_k^{t_{kp}}(\mathbf{x}_p)$  (similarly to the two classes case, see (11.35)). Then:

$$\mathcal{L} = \prod_{p=1}^P p(\mathbf{t}_p|\mathbf{x}_p) = \prod_{p=1}^P \prod_{k=1}^K y_k^{t_{kp}}(\mathbf{x}_p) \Rightarrow E = -\ln \mathcal{L} = -\sum_{p=1}^P \sum_{k=1}^K t_{kp} \ln y_k(\mathbf{x}_p)$$

The error function have a minimum when all the output values  $y_k(\mathbf{x}_p)$  coincide with the targets  $t_{kp}$ :

$$E_{\min} = -\sum_{p=1}^P \sum_{k=1}^K t_{kp} \ln t_{kp}$$

and, as in (11.38), the minimum may be subtracted from the general expression, and the energy becomes:

$$E = - \sum_{p=1}^P \sum_{k=1}^K t_{kp} \ln \frac{y_k(\mathbf{x}_p)}{t_{kp}} = - \sum_{p=1}^P E_p \quad (11.43)$$

❖  $E_p$  where  $E_p = \sum_{k=1}^K t_{kp} \ln \frac{y_k(\mathbf{x}_p)}{t_{kp}}$ .

To represent the posterior probabilities  $P(\mathcal{C}_k|\mathbf{x})$ , the network outputs should be  $y_k \in [0, 1]$  and sum to unity  $\sum_{k=1}^K y_k = 1$ . To ensure this, the softmax function may be used:

$$y_k = \frac{\exp(a_k)}{\sum_{\ell=1}^K \exp(a_\ell)} = \frac{1}{1 + \exp(-A_k)} \quad \text{where} \quad A_k = a_k - \ln \sum_{\ell=1, \ell \neq k} \exp(a_\ell) \quad (11.44)$$

and it may be seen that the activation of output neurons is the logistic function.

Let assume that the posterior probabilities of the outputs of hidden neurons is of the general exponential form as (11.39)

$$p(\mathbf{z}|\mathcal{C}_k) = \exp \left[ A(\boldsymbol{\theta}_k) + B(\mathbf{z}, \varphi) + \boldsymbol{\theta}_k^T \mathbf{z} \right] \quad (11.45)$$

where  $A$ ,  $B$ ,  $\boldsymbol{\theta}_k$  and  $\varphi$  have the same significance as in section 11.7.2.

From Bayes theorem, the posterior probability  $p(\mathcal{C}_k|\mathbf{z})$  is

$$p(\mathcal{C}_k|\mathbf{z}) = \frac{p(\mathbf{z}|\mathcal{C}_k) P(\mathcal{C}_k)}{p(\mathbf{z})} = \frac{p(\mathbf{z}|\mathcal{C}_k) P(\mathcal{C}_k)}{\sum_{\ell=1}^K p(\mathbf{z}|\mathcal{C}_\ell) P(\mathcal{C}_\ell)} \quad (11.46)$$

By substituting (11.45) in (11.46), the posterior probability becomes:  $p(\mathcal{C}_k|\mathbf{z}) = \frac{\exp(a_k)}{\sum_{\ell=1}^K \exp(a_\ell)}$

❖  $a_k$ ,  $\mathbf{w}_k$ ,  $w_{k0}$  where:

$$a_k = \mathbf{w}_k^T \mathbf{z} + w_{k0} \quad \text{and} \quad \begin{cases} \mathbf{w}_k = \boldsymbol{\theta}_k \\ w_{k0} = A(\boldsymbol{\theta}_k) + \ln P(\mathcal{C}_k) \end{cases}$$

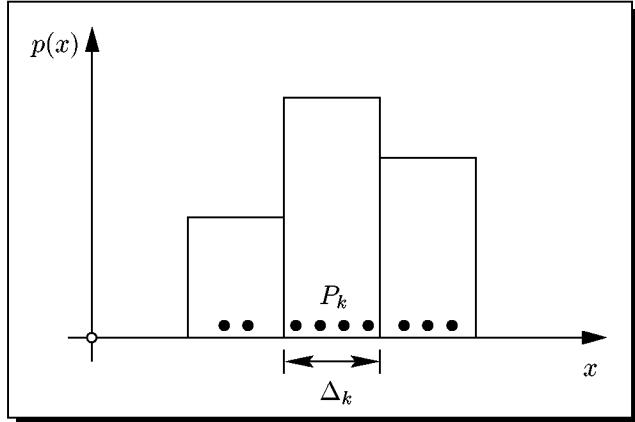
The derivative of error function with respect to the weighted sum of inputs  $a_k$  of the output neurons is:

$$\frac{\partial E_p}{\partial a_k} = \sum_{\ell=1}^K \frac{\partial E_p}{\partial y_\ell} \frac{\partial y_\ell}{\partial a_k}$$

(because  $E_p$  depends on  $a_k$  through all  $y_\ell$ , see (11.44)).

From (11.43) respectively (11.44)

$$\frac{\partial E_p}{\partial y_\ell} = -\frac{t_{\ell p}}{y_\ell(\mathbf{x}_p)} \quad \text{respectively} \quad \frac{\partial y_\ell}{\partial a_k} = y_\ell \delta_{\ell k} - y_\ell y_k$$



**Figure 11.2:** A histogram. It may be seen as a set of “bins”, each containing some “objects”.

and finally  $\frac{\partial E_p}{\partial a_k} = y_k(x_p) - t_k$ , same result as for sum-of-squares error with linear activation and two class with logistic activation cases.

## 11.8 Entropy

Let consider a random variable  $x$  and its probability density  $p(x)$ . Also let consider a set of  $P$  values  $\{x_p\}_{p=1,P}$ . By dividing the  $X$  axis ( $x \in X$ ) into “bins” of width  $\Delta_k$  (for each “bin”  $k$ ) and putting each  $p(x_p)$  into the corresponding “bin”, a histogram is obtained. See figure 11.2.

❖  $\Delta_k$

Let consider the number of ways the objects from bins are arranged. Let  $P_k$  be the number of “objects” from bin no.  $k$ . There are  $P$  ways to pick up the first object,  $P - 1$  ways to pick up the second one and so on, there are  $P!$  ways to pick up the set of  $P$  objects. Also there are  $P_k!$  ways to arrange the objects in each bin. Because the number of arrangements inside bins doesn’t count then the total number of arrangements will end up in the same histogram is:

$$W = \frac{P!}{\prod_k P_k!}$$

The particular way of arranging the objects in bins is called a *microstate*, while the arrangement with leads to the same  $p(x)$  is called *macrostate*. The parameter representing the number of microstates for one given macrostate is named *multiplicity*.

❖  $P_k$

microstate  
macrostate  
multiplicity

### ☞ Remarks:

- ➡ The notion of entropy came from physics where it’s a measure of the system “disorder” (higher “disorder” meaning higher entropy). In the information theory it is a measure of information content.

<sup>11.8</sup>See [Bis95] pp. 240–245.

→ There is an analogy which may be done between the above example and a physics related example. Consider a gas formed of molecules with different speeds (just one component of the speed will be considered).

From macroscopic (macrostate) point of view it doesn't matter which molecule have a (particular) given speed while from microscopic point of view if two molecules swap speeds there is.

→ If the number of microstates decrease for the same macrostate then the order in the system increases (till there is only one microstate corresponding to the given macrostate in which case the system is totally ordered — there is only one way to arrange it). As entropy measures the degree of "disorder" it shall be inversely proportional to the multiplicity.

The results from statistical physics corroborated with thermodynamics lead to the definition of entropy as being equal (up to a multiplicative constant) with the negative logarithm of multiplicity; the constant being the Boltzmann constant  $k$ .

Based on the above remarks, the entropy in the above situation is defined as:

$$S = -\frac{1}{P} \ln W = -\frac{1}{P} \left[ \ln P! - \sum_k P_k! \right]$$

and, by using the Stirling formula  $\ln P! \simeq P \ln P - P$  for  $P \gg 1$  and the relation  $\sum_k P_k = P$ , for  $P, P_k \rightarrow \infty$ , it becomes:

$$S = -\sum_k \frac{P_k}{P} \ln \frac{P_k}{P} = \sum_k p_k \ln p_k \quad (11.47)$$

❖  $p_k$

where  $p_k = P_k/P$  represents the — observed in histogram — probability density.

The lower entropy ( $S = 0$ ) will be when all "objects" are in one single "bin", i.e. all probabilities  $p_k = 0$  with the exception of one  $p_\ell = 1$ . The highest entropy will be when all  $p_k$  are equal.

*Proof.* The proof of the above statement is obtained by using the Lagrange multiplier technique to find the maximum (see mathematical appendix) but the minimum of  $S$  will be missed due to discontinuities at 0. The constraint is the condition of normation  $\sum_k p_k = 1$ ; then the Lagrange function will be:

$$L = \sum_k p_k \ln p_k + \lambda \left[ \sum_k p_k - 1 \right]$$

and the maximum is found by zeroing the derivatives:

$$\frac{\partial L}{\partial p_k} = \ln p_k + 1 + \lambda = 0 \quad \text{and} \quad \frac{\partial L}{\partial \lambda} = \sum_k p_k - 1 = 0$$

❖  $I$

then from the first equation:  $p_k = e^{-(1+\lambda)}$ ,  $\forall k$ , and, considering  $K$  as the total number of "bins", by introducing this value into the second one  $\lambda = -1 + \ln I$ . Then for  $p_k = 1/I$  the entropy is maximum. □

Considering the limit  $K \rightarrow \infty$  (number of bins) then the probability density is constant inside a "bin" and  $p_k = p(x_p \in \text{bin}_k) \Delta_k$ , where  $x_p$  is from inside "bin"  $k$ . Then the entropy is:

$$S = \sum_k p(x_p \in \text{bin}_k) \Delta_k \ln[p(x_p) \Delta_k] \simeq \int_X p(x) \ln p(x) dx + \lim_{K \rightarrow \infty} \ln \Delta_k$$

(because  $\Delta_k = \text{const.}$  and  $\int_X p(x) dx = 1$ ). For  $K \rightarrow \infty$  the width of "bins" will reduce to  $\Delta_k \rightarrow 0$  and thus the term  $\lim_{K \rightarrow \infty} \ln \Delta_k \rightarrow \infty$  and, to keep a meaning, it is dropped from the expression of entropy (anyway the most important is the change of entropy  $\Delta S$  and the divergent term disappears when calculating it; this is also the reason for the name "differential entropy").

**Definition 11.8.1.** *The general expression of differential entropy is:*

differential entropy

$$S = - \int_X p(\mathbf{x}) \ln p(\mathbf{x}) d\mathbf{x}$$

The distribution with maximum of entropy, subject to the following constraints:

- $\int_{-\infty}^{\infty} p(x) dx = 1$ , i.e. normalization of distribution.
- $\int_{-\infty}^{\infty} xp(x) dx = \mu$ , i.e. existence of a mean.
- $\int_{-\infty}^{\infty} (x - \mu)^2 p(x) dx = \sigma^2$ , i.e. existence of a variance.

is the Gaussian:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x - \mu)^2}{2\sigma^2}\right]$$

*Proof.* The distribution is found through the Lagrange multipliers method (see mathematical appendix). The Lagrange function is:

$$L = \int_{-\infty}^{\infty} p(x) [\ln p(x) + \lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2] dx - \lambda_1 - \lambda_2 x - \lambda_3 (x - \mu)^2$$

Then maximum of  $S$  is found by zeroing the derivatives of  $L$ :

$$\frac{\partial L}{\partial p} = \int_{-\infty}^{\infty} [\ln p(x) + 1 + \lambda_1 + \lambda_2 x + \lambda_3 (x - \mu)^2] dx = 0 \quad (11.48)$$

and the other derivative equations  $\frac{\partial L}{\partial \lambda_1}, \frac{\partial L}{\partial \lambda_2}, \frac{\partial L}{\partial \lambda_3}$  just lead to the imposed conditions. The derivative (11.48) may be zero only if the integrand is zero — because the integrand is an odd function<sup>2</sup> and the integration is made over an interval symmetric relatively to the origin. Then the searched probability density is of the form:

$$p(x) = \exp [-1 - \lambda_1 - \lambda_2 x - \lambda_3 (x - \mu)^2] \quad (11.49)$$

and the  $\lambda_1, \lambda_2, \lambda_3$  constants are found from the conditions imposed.

From the first condition:

$$\begin{aligned} 1 &= \int_{-\infty}^{\infty} \exp [-1 - \lambda_1 - \lambda_2 x - \lambda_3 (x - \mu)^2] dx \\ &= \exp \left( -1 - \lambda_1 + \frac{\lambda_2}{2\lambda_3} - \lambda_2 \mu \right) \int_{-\infty}^{\infty} \exp \left[ -\lambda_3 \left( x - \mu + \frac{\lambda_2}{2\lambda_3} \right)^2 \right] dx \end{aligned} \quad (11.50)$$

<sup>2</sup>A function  $f$  is odd if  $f(-x) = f(x)$ .

$$= \exp\left(-1 - \lambda_1 + \frac{\lambda_2}{2\lambda_3} - \lambda_2\mu\right) \sqrt{\frac{\pi}{\lambda_3}}$$

(see also the Gaussian integrals, mathematical appendix).

From the second condition, similar as above, and using the change of variable  $\tilde{x} = x - \mu$  (one of the integrals will cancel due to the fact that the integrand will be an odd function and the integration interval is symmetric relatively to origin):

$$\begin{aligned}\mu &= \int_{-\infty}^{\infty} x \exp[-1 - \lambda_1 - \lambda_2 x - \lambda_3(x - \mu)^2] dx \\ &= \exp\left(-1 - \lambda_1 + \frac{\lambda_2}{2\lambda_3} - \lambda_2\mu\right) \int_{-\infty}^{\infty} x \exp\left[-\lambda_3\left(x - \mu + \frac{\lambda_2}{2\lambda_3}\right)^2\right] dx \\ &= \exp\left(-1 - \lambda_1 + \frac{\lambda_2}{2\lambda_3} - \lambda_2\mu\right) \left(\mu - \frac{\lambda_2}{2\lambda_3}\right) \int_{-\infty}^{\infty} x \exp\left[-\lambda_3\left(x - \mu + \frac{\lambda_2}{2\lambda_3}\right)^2\right] dx\end{aligned}$$

and by using (11.50) (see also Gaussian integrals) it follows that  $\mu = \mu - \frac{\lambda_2}{2\lambda_3}$ , thus  $\lambda_2 = 0$ . Replacing back into (11.50) gives  $\exp(1 + \lambda_1) = \sqrt{\frac{\pi}{\lambda_3}}$ .

From the third condition, as  $\lambda_2 = 0$  (integration by parts):

$$\begin{aligned}\sigma^2 &= \int_{-\infty}^{\infty} (x - \mu)^2 \exp[-1 - \lambda_1 - \lambda_3(x - \mu)^2] dx \\ &= -\frac{e^{-1-\lambda_1}}{2\lambda_3} \int_{-\infty}^{\infty} (x - \mu) d\{\exp[-\lambda_3(x - \mu)^2]\} = -\frac{e^{-1-\lambda_1}}{2\lambda_3} \sqrt{\frac{\pi}{\lambda_3}}\end{aligned}$$

and then finally  $\lambda_3 = \frac{1}{2\sigma^2}$  and  $e^{-1-\lambda_1} = \frac{1}{\sqrt{2\pi}\sigma}$ . The wanted distribution is found by replacing the  $\lambda_{1,2,3}$  values back into (11.49).  $\square$

Another way of interpreting the entropy is to consider it as the amount of information. It is reasonable to consider that the amount of information and the probability are somehow interdependent, i.e.  $S = S(p)$ .

The entropy is a continuous, monotonically increasing function. On the other hand an *certain* event have probability  $p = 1$  and bears no information  $S(1) = 0$ .

Let now consider two events  $A$  and  $B$ , statistically independent and having probabilities  $p_A$  and  $p_B$ . The probability of both events occurring is  $p_A p_B$  and the entropy associated is  $S(p_A p_B)$ . If event  $A$  have occurred then the information given by event  $B$  is  $S(p_A p_B) - S(p_A)$  and on the other hand  $S(p_B)$ . Then:

$$S(p_A p_B) = S(p_A) + S(p_B)$$

From the above result it follows that  $S(p^2) = 2S(p)$  and, by induction,  $S(p^n) = nS(p)$ . Then, by the same means  $S(p) = S\{(p^{1/n})^n\} = nS(p^{1/n})$  which may be immediately extended to  $S(p^{n/m}) = \frac{n}{m}S(p)$ . Finally, from the continuity of  $S$ :

$$S(p^x) = xS(p)$$

and thus it also may be written as:

$$S(p) = S\{(1/2)^{-\log_2 p}\} = -S(1/2) \log_2 p$$

where  $S(1/2)$  is a multiplicative constant and by choosing it to be equal to 1 the entropy/information is being said to be expressed in *bits*. By choosing natural logarithm  $S(p) = -S(1/e) \ln p$  and  $S(1/e) = 1$  the information will be expressed in *nats*.

bits, nats

### Remarks:

- ➡ The above result may be explained by the fact that, for independent events, the information is additive (there may be information about event  $A$  and event  $B$ ) while the probability is multiplicative.

Let consider a discrete random variable  $x$  which may take one of the values  $\{x_p\}$  and have to be transmitted, or rather *information about it*. For each  $x_p$  the information is  $-\ln p(x_p)$ . Then the expected/average information/entropy to be transmitted (as to establish what of the possible  $x_p$  values  $x$  have now) is:

$$S = - \sum_p p(x_p) \ln p(x_p)$$

the result being known also as *noiseless coding theorem*.

For a continuous vectorial variable  $\mathbf{x}$ , and also

noiseless coding theorem

considering that usually the true distribution  $p(\mathbf{x})$  is not known but rather the estimated one  $\tilde{p}(\mathbf{x})$  (the associated information being  $-\ln \tilde{p}(\mathbf{x})$ ). Then the average/expected information relative to  $\mathbf{x}$  is:

$$S = - \int_X p(\mathbf{x}) \ln \tilde{p}(\mathbf{x}) d\mathbf{x} \quad (11.51)$$

### Remarks:

- ➡ The entropy may be written as:

$$S = - \int_X p(\mathbf{x}) \ln \frac{\tilde{p}(\mathbf{x})}{p(\mathbf{x})} d\mathbf{x} - \int_X p(\mathbf{x}) \ln p(\mathbf{x}) d\mathbf{x}$$

where the first term represents the asymmetric divergence between  $p(\mathbf{x})$  and  $\tilde{p}(\mathbf{x})$ <sup>3</sup>. It is shown in the "Pattern Recognition" chapter that  $S(\mathbf{x})$  is minimum for  $\tilde{p}(\mathbf{x}) = p(\mathbf{x})$ .

For a neural network which have the outputs  $y_k$  the entropy (11.51), for one  $\mathbf{x}_p$  is:

$$S_p = - \sum_{k=1}^K t_k \ln y_k(\mathbf{x}_p)$$

because  $t_k$  represents the true probability (desired output) while  $y_k(\mathbf{x})$  represents the estimated (calculated) probability (actual output). For the whole training set the cross-entropy is:

$$S = - \sum_{p=1}^P \sum_{k=1}^K t_{kp} \ln y_k(\mathbf{x}_p)$$

the result being valid for all networks for which  $t_{kp}$  and  $y_k$  represents probabilities.

<sup>3</sup>Known also as Kullback-Leibler distance.

## 11.9 Outputs as Probabilities

Due to the fact that many error functions lead to the interpretation of network output as probabilities, the problem is to find the general conditions which lead to this result.

It will be assumed that the error function is additive with respect to the number of training vectors  $E = \sum_{p=1}^P E_p$  and each  $E_p$  is a sum over the components of the target and actual output vectors:

$$E_p = \sum_{k=1}^K f(t_{kp}, y_k(\mathbf{x}_p))$$

❖  $f$  where  $f$  is a function to be found, assumed to be of the form:

$$f(t_{kp}, y_k(\mathbf{x}_p)) = f(|y_k(\mathbf{x}_p) - t_{kp}|)$$

i.e. it depends only on the distance between actual and desired output.

❖  $\langle E_p \rangle$  Considering an infinite training set, the expected (average) per-pattern error is:

$$\langle E_p \rangle = \sum_{k=1}^K \iint_{XY} f(|y_k(\mathbf{x}) - t_k|) p(\mathbf{t}|\mathbf{x}) p(\mathbf{x}) d\mathbf{t} d\mathbf{x}$$

For the one-of- $k$  encoding scheme, and considering statistically independent  $y_k$  (network outputs), the conditional probability of the target may be written as:

$$p(\mathbf{t}|\mathbf{x}) = \prod_{\ell=1}^K \left[ \sum_{q=1}^K \delta_D(t_\ell - \delta_{\ell q}) P(\mathcal{C}_q|\mathbf{x}) \right] = \prod_{\ell=1}^K p(t_\ell|\mathbf{x}) \quad (11.52)$$

and then the expectation of the error function becomes:

$$\langle E_p \rangle = \sum_{k=1}^K \int_X \left[ \prod_{\substack{\ell=1 \\ \ell \neq k}}^K \int_{Y_\ell} p(t_\ell|\mathbf{x}) dt_\ell \right] \left[ \int_{Y_k} f(|y_k(\mathbf{x}) - t_k|) p(t_k|\mathbf{x}) dt_k \right] p(\mathbf{x}) d\mathbf{x}$$

where the integrals over  $Y_{\ell, \ell \neq k}$  are equal to 1 due to the normalization of the probability (or, otherwise, by substituting the value of  $p(t_\ell|\mathbf{x})$  and doing the integral) and the  $p(t_k|\mathbf{x})$  probability density may be written as (see (11.52) — the cases  $k = q$  and  $k \neq q$  were considered):

$$\begin{aligned} p(t_k|\mathbf{x}) &= \delta_D(t_k - 1) P(\mathcal{C}_k|\mathbf{x}) + \sum_{\substack{q=1 \\ q \neq k}}^K \delta_D(t_k) P(\mathcal{C}_q|\mathbf{x}) = \\ &= \delta_D(t_k - 1) P(\mathcal{C}_k|\mathbf{x}) + \delta_D(t_k) [1 - P(\mathcal{C}_k|\mathbf{x})] \end{aligned}$$

(because  $\sum_{q=1}^K P(\mathcal{C}_q|\mathbf{x}) = 1$ , normalization).

---

<sup>11.9</sup>See [Bis95] pp. 245–247.

Finally the average error becomes:

$$\begin{aligned}
 \langle E_p \rangle &= \sum_{k=1}^K \int_X \left\{ \int_{Y_k} f(|y_k(\mathbf{x}) - t_k|) \delta_D(t_k - 1) P(\mathcal{C}_k | \mathbf{x}) dt_k \right. \\
 &\quad \left. + \int_{Y_k} f(|y_k(\mathbf{x}) - t_k|) \delta_D(t_k) [1 - P(\mathcal{C}_k | \mathbf{x})] dt_k \right\} p(\mathbf{x}) d\mathbf{x} \\
 &= \sum_{k=1}^K \int_X \left\{ \int_{Y_k} f(1 - y_k) \delta_D(0) P(\mathcal{C}_k | \mathbf{x}) dt_k \right. \\
 &\quad \left. + \int_{Y_k} f(y_k) \delta_D(0) [1 - P(\mathcal{C}_k | \mathbf{x})] dt_k \right\} p(\mathbf{x}) d\mathbf{x} \\
 &= \sum_{k=1}^K \int_X \{f(1 - y_k) P(\mathcal{C}_k | \mathbf{x}) + f(y_k) [1 - P(\mathcal{C}_k | \mathbf{x})]\} p(\mathbf{x}) d\mathbf{x}
 \end{aligned}$$

because the first integral over  $Y_k$  is for the case  $t_k = 1$  while the second is for the case  $t_k = 0$ ; obviously  $y_k \in [0, 1]$ .

The conditions of minima for  $\langle E_p \rangle$  are found by setting its functional derivative, with respect to  $y_k(\mathbf{x})$ , to zero:

$$\frac{\delta \langle E_p \rangle}{\delta y_k(\mathbf{x})} = f'(1 - y_k) P(\mathcal{C}_k | \mathbf{x}) + f'(y_k) [1 - P(\mathcal{C}_k | \mathbf{x})] = 0$$

where  $f'$  is the derivative of  $f$ . This leads to:

$$\frac{f'(1 - y_k)}{f'(y_k)} = \frac{1 - P(\mathcal{C}_k | \mathbf{x})}{P(\mathcal{C}_k | \mathbf{x})}$$

and, considering that the network outputs are probabilities (this was the hypothesis):

$$\frac{f'(1 - y_k)}{f'(y_k)} = \frac{1 - y_k}{y_k} \tag{11.53}$$

A general class of functions which satisfies (11.53) is:

$$f(y) = \int y^r (1 - y)^r dy \quad , \quad r = \text{const.}$$

and from this class ( $f'(1 - y) = df(1 - y)/d(1 - y)$ ):

- $r = 1 \Rightarrow f(y) = y^2/2$ , i.e. the sum-of-squares error function.
- $r = 0 \Rightarrow f(y) = -\ln(1 - y) = -\ln(1 - |y|)$ , i.e. the cross-entropy error function.



### Remarks:

- The Minkowski error function  $f(y) = y^R$  does not satisfy the (11.53) unless  $R = 2$  which leads to the already known, sum-of-squares error.

In this case the outputs of network are:

$$y_k = \frac{P(\mathcal{C}_k|\mathbf{x})^{1/(R-1)}}{P(\mathcal{C}_k|\mathbf{x})^{1/(R-1)} + [1 - P(\mathcal{C}_k|\mathbf{x})]^{1/(R-1)}}$$

On the other hand the decision boundaries still give the minimum of misclassification because  $y_k$  are monotonic functions of  $P(\mathcal{C}_k|\mathbf{x})$ .

## CHAPTER 12

# Parameter Optimization

### → 12.1 Error Surfaces

Generally the error may be represented as a surface  $E = E(W)$ <sup>1</sup> into the  $N_W + 1$  space where  $N_W$  is the total number of weights (see also figure 12.2 on page 219).

❖  $N_W$

The goal is to find the minima of error function, where  $\nabla E = 0$ ; however note that this condition is not enough to find the absolute minima because it is also true for local minimums, maximums and saddle-points.

In general it is not possible to find the solution  $W$  in a closed form. Then a numerical approach is taken, to find it by searching the weights space in incremental steps ( $t = 1, \dots$ ) of the form  $W_{(t+1)} = W_{(t)} + \Delta W_{(t)}$ . However, usually, the algorithms does not guarantee for the finding of absolute minima and even an saddle-point may stuck them.

On the other hand the weight space have a high degree of symmetry<sup>2</sup> and thus many local and global minimums which give the same value for the error function; then a relatively fast convergence may be achieved starting from a random point (i.e. the local/global minima will be relatively close wherever the starting point is).

It was shown<sup>3</sup> that the optimum value for the network is obtained when  $\langle y_k | x \rangle = \langle t_k | x \rangle$ , i.e. the actual average output  $y_k$  equals the desired output  $t_k$ , both conditioned on input  $x$ . However this equality was obtained by considering the training set as infinite while in practice the training set is always finite and covers just a tiny amount from all possible cases. Then even a local minima may give a good generalization.

---

<sup>12.1</sup>See [Bis95] pp. 253–256.

<sup>1</sup>Such a surface was drawn in the “Backpropagation” chapter for 2 weights.

<sup>2</sup>See the “Multi Layer Neural Networks” chapter.

<sup>3</sup>See the “Error Functions” chapter.

## → 12.2 Local Quadratic Approximation

Let consider the Taylor series development of error  $E$  around a point  $W_0$ , note that here  $W$  is being seen as a *vector*:

$$E(W) = E(W_0) + \nabla E|_{W_0} + \frac{1}{2}(W - W_0)^T H|_{W_0}(W - W_0) \quad (12.1)$$

❖  $H$  where  $\{H\}_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j}$  is the Hessian matrix, note that here the Hessian will be considered as a matrix. Then the gradient of  $E$  with respect to  $W$  may be approximated as:

$$\nabla E = \nabla E|_{W_0} + H|_{W_0}(W - W_0)$$

❖  $W^*$  Considering a minima point  $W^*$  then  $\nabla E|_{W^*} = 0$  and:

$$E(W) = E(W^*) + \frac{1}{2}(W - W^*)^T H|_{W^*}(W - W^*) \quad (12.2)$$

❖  $\mathbf{u}_i, \lambda_i$  Because the Hessian is a symmetric matrix then it is possible to find an orthonormal set of eigenvectors<sup>4</sup>  $\{\mathbf{u}_i\}$  with the corresponding eigenvalues  $\{\lambda_i\}$  such that:

$$H\mathbf{u}_i = \lambda_i \mathbf{u}_i \quad , \quad \mathbf{u}_i^T \mathbf{u}_j = \delta_{ij} \quad (12.3)$$

( $\lambda_i$  and  $\mathbf{u}_i$  being calculated at the minimum of  $E$ , given by the point  $W^*$ ).

By writing  $(W - W^*)$  in terms of  $\{\mathbf{u}_i\}$

$$W - W^* = \sum_i \alpha_i \mathbf{u}_i \quad (12.4)$$

❖  $\alpha_i$  (where  $\alpha_i$  are the coefficients of the  $\{\mathbf{u}_i\}$  development) and replacing in (12.2)

$$E(W) = E(W^*) + \frac{1}{2} \sum_i \lambda_i \alpha_i^2$$

i.e. *the error hyper-surfaces projected into the weights space are hyper-ellipsoids* oriented with the axes parallel with the orthonormated  $\{\mathbf{u}_i\}$  system and having the lengths inversely proportional with the square roots of Hessian eigenvalues. See figure 12.1 on the facing page.

### Remarks:

► From (12.2):

$$\Delta E = E(W) - E(W^*) = \frac{1}{2}(W - W^*)^T H|_{W^*}(W - W^*)$$

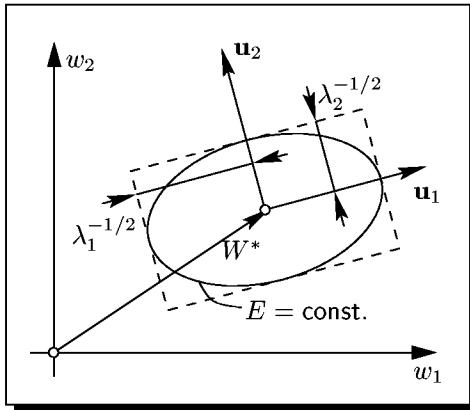
then  $W^*$  is a minimum point if only if  $H$  is positive definite<sup>5</sup> ( $\Leftrightarrow \Delta E \geq 0 \Leftrightarrow E(W) \geq E(W^*)$ ).

---

<sup>12.2</sup>See [Bis95] pp. 257–259.

<sup>4</sup>See mathematical appendix.

<sup>5</sup>See previous footnote.



**Figure 12.1:** A constant error surface projection into a bidimensional weights space is an ellipse having the axes parallel with the Hessian eigenvectors and lengths inversely proportional with the square root of Hessian eigen values.

- The quadratic approximation of error function (12.1) involves the knowledge of  $W(W+3)/2$  terms ( $W$  for  $\nabla E$  and  $W(W+1)/2$  for the symmetrical Hessian matrix). So, to find a minimum will require at least  $\mathcal{O}(W^2)$  equations, each needing at least  $\mathcal{O}(W)$  steps, i.e. a total of  $\mathcal{O}(W^3)$  steps.  
By using the gradient information and the backpropagation algorithm the required steps are reduced to  $\mathcal{O}(W^2)$ .
- It was shown<sup>6</sup> that for linear output neurons it is possible to find the related weights by one step.

## 12.3 Initialization and Termination of Learning

Usually the weights are initialized with random values to avoid problems due to weight space symmetry. However there are two restrictions:

- If the initial weights are too big then the activation functions  $f$  will have values into the saturation region (e.g. see sigmoidal activation function) and their derivatives  $f'$  will be small, leading to an small error gradient as well, i.e. a approximatively flat error surface and, consequently, a slow learning.
- If the initial weights are too small then the activation functions  $f$  will be linear and their derivatives will be quasi-constant, the second derivatives will be small and then the Hessian will be small meaning that around minimums the error surface will be approximatively flat and, consequently, a slow learning (see section 12.2).

which suggest that the weighted sum of inputs, for sigmoidal activation function, should be of order unity.

<sup>6</sup>See chapter “Single Layer Neural Networks”.

12.3 See [Bis95] pp. 260–263.

The weights are usually drawn from a symmetrical Gaussian distribution with 0 mean (there is no reason to choose any other mean, due to symmetry). To see the choice for the variance  $\sigma$  of the above distribution, let consider a set of inputs  $\{x_i\}_{i=1,N}$  with zero mean:  $\langle x_i \rangle = 0$  and unit variance:  $\langle x_i^2 \rangle = 1$  (calculated over the training set).

The output of neuron is:

$$z = f(a) \quad \text{where} \quad a = \sum_{i=0}^N w_i x_i$$

(for  $i = 0$  the weight  $w_0$  represents the bias and  $x_0 = 1$ ).

The weights are chosen randomly, independent of  $\{x_i\}$ , and then the mean of  $a$  is:

$$\langle a \rangle = \sum_{i=0}^N \langle w_i x_i \rangle = \sum_{i=0}^N \langle w_i \rangle \langle x_i \rangle = 0$$

(as  $\langle x_i \rangle = 0$ ) and, considering  $\{w_i\}$  statistically independent  $\langle w_i w_j \rangle = \delta_{ij} \sigma^2$ , the variance is:

$$\langle a^2 \rangle = \left\langle \left( \sum_{i=0}^N w_i x_i \right) \left( \sum_{j=0}^N w_j x_j \right) \right\rangle = \sum_{i=0}^N \langle w_i^2 \rangle \langle x_i^2 \rangle = (N+1)\sigma^2 \simeq N\sigma^2$$

(for non-bias the sum is from  $i = 1$  to  $N$  and then  $\langle a^2 \rangle = N\sigma^2$ ). Because  $\langle a^2 \rangle \simeq 1$  (as discussed above) then the variance should be chosen  $\sigma \simeq N^{-1/2}$ .

committee of networks

Another way to improve network performance is to train multiple instances of the same network, but with a different set of initial weights, and choosing among those who give best results. This method is called *committee of networks*.

The criteria for stopping the learning process may be one of the following:

- Stop after a fixed number of steps.
- Stop when the error function had become smaller than a specified amount.
- Stop when the change in the error function ( $\Delta E$ ) had become smaller than a specified amount.
- Stop when the error on an (independent) validation set begin to increase.

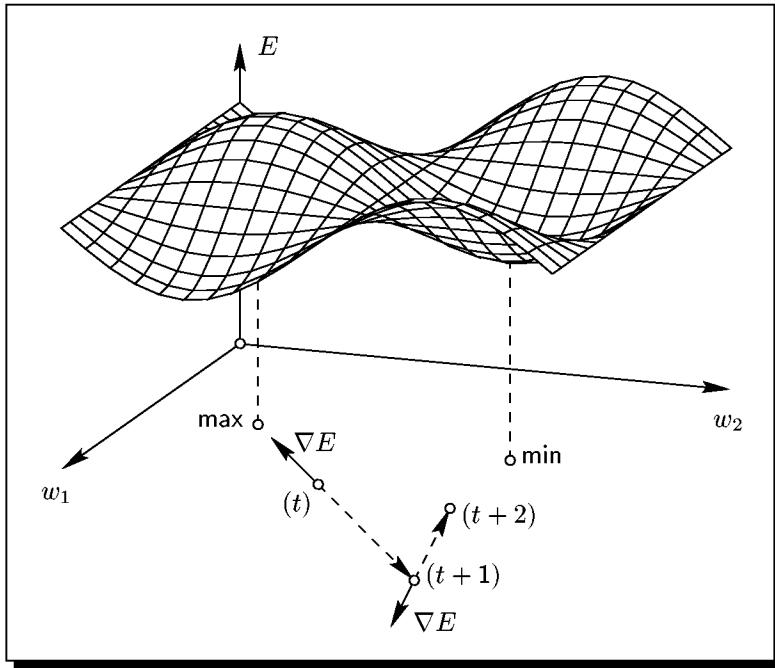
## 12.4 Gradient Descent

A simple, geometrical, explanation of the gradient descent method is the means of representation of the error surface into the weights space. See figure 12.2 on the next page.

The gradient of error function, relative to the weights,  $\nabla E$  will point to the set of weights which will give maximum error. Then the weights have to be moved against the direction of  $\nabla E$ ; note that this does *not* mean a movement towards the minima point. See figure 12.2 on the facing page.

---

<sup>12.4</sup>See [Bis95] pp. 263–272.



**Figure 12.2:** The error surface for a bidimensional weights space. The steps  $t$ ,  $t + 1$  and  $t + 2$  are also shown. The weights are moved against the error maximum (and **not** towards minimum).

At step  $t = 0$  (in discrete time approximation) the weights are initialized with the value  $W_0$  (usually randomly selected). Then at some step  $t + 1$  they are adjusted following the formula:

$$\Delta W_{(t+1)} = W_{(t+1)} - W_{(t)} = -\eta \nabla E|_{W_{(t)}} \quad (12.5)$$

where  $\eta$  is a parameter governing the speed of learning, named *learning rate/constant*, controlling the distance between  $W_{(t+1)}$  and  $W_{(t)}$  (see also figure 12.2). This type of learning is named a *batch process* (it uses all training vectors at once every time the gradient is calculated). (12.5) is also known as delta rule.

learning rate  
batch learning

Alternatively the same method (as above) may be used but with one pattern at a time:

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

where  $p$  denotes a pattern from the training set, e.g. the training patterns may be numbered in some way and then considered in the order  $p = t$  (first pattern taken at first step, ..., and so on). This type of learning is named *sequential process* (it uses just one training vector at a time).

sequential learning

### Remarks:

- ➡ The gradient descent technique is very similar to the Robbins-Monro algorithm<sup>7</sup> and it becomes identical if the learning rate is chosen of the form  $\eta \propto 1/t$  and thus

<sup>7</sup>See chapter "Pattern recognition".

the convergence is assured; however this alternative leads to very long training time.

- The choice of  $\eta$  may be critical to the learning process. A large value may lead to an over shooting of the minima especially if it's narrow and step (in terms of error surface) and/or oscillations between 2 areas (points) in weights space. A small value will lead to long learning time.
- Compared to the batch learning, the sequential process is less sensitive to training data redundancy (a duplicate of a training vector will be used twice, in two separate steps, and thus, usually, improving learning, rather than being used twice at each learning step).
- It is also possible to use a mixture learning, i.e. to divide the training set into subsets and use each subset for batch process. This technique is especially useful if the training algorithm is intrinsically of the batch type.

### 12.4.1 Learning Parameter and Convergence

Considering the quadratic approximation of the error function in the neighborhood of minima, from (12.2), (12.3) and (12.4):

$$\nabla E = \sum_i \alpha_i \lambda_i \mathbf{u}_i \quad (12.6)$$

From (12.4):

$$\Delta W_{(t+1)} = \sum_i \Delta \alpha_i \mathbf{u}_i \quad \text{where} \quad \Delta \alpha_i = \alpha_{i(t+1)} - \alpha_{i(t)} \quad (12.7)$$

and by replacing (12.6) and (12.7) in (12.5) and because the eigenvectors  $\mathbf{u}_i$  are orthonormalized, then:

$$\Delta \alpha_i = -\eta \lambda_i \alpha_{i(t)} \quad \Rightarrow \quad \alpha_{i(t+1)} = (1 - \eta \lambda_i) \alpha_{i(t)} \quad (12.8)$$

Again from (12.4), by multiplying with  $\mathbf{u}_i^T$  to the left, and from the orthonormalization of  $\{\mathbf{u}_i\}$ :

$$\mathbf{u}_i^T (W - W^*) = \alpha_i$$

i.e.  $\alpha_i$  represents the distance (into the weights space) to the minimum, along the direction given by  $\mathbf{u}_i$ .

After  $T$  steps the iterative usage of (12.8) gives:

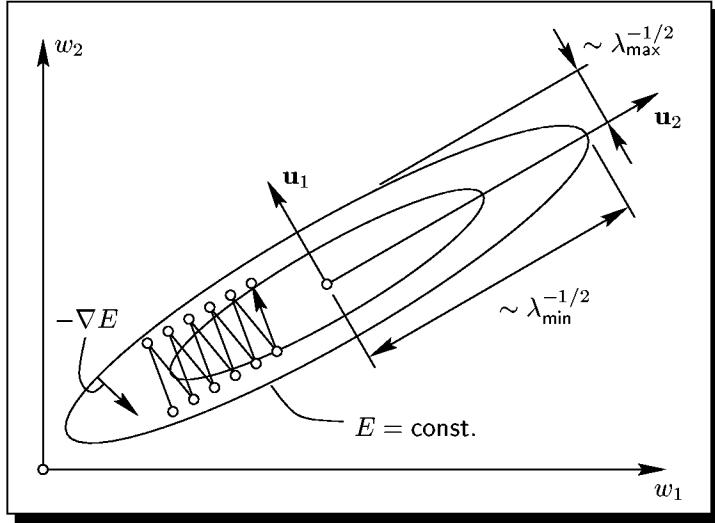
$$\alpha_{i(T)} = (1 - \eta \lambda_i)^T \alpha_{i(0)} \quad (12.9)$$

A convergence process means that  $\lim_{t \rightarrow \infty} W_{(t)} = W^*$ , i.e.  $\lim_{t \rightarrow \infty} \alpha_{i(t)} = 0$  (by considering the significance of  $\alpha_i$  as discussed above). Then, to have a convergent learning, formula (12.9) shows that it is necessary to impose the condition:

$$|1 - \eta \lambda_i| < 1, \forall i \quad \Rightarrow \quad \eta < \frac{2}{\lambda_{\max}}$$

❖  $\lambda_{\max}$

where  $\lambda_{\max}$  is the biggest eigenvalue of the Hessian.



**Figure 12.3:** Slow learning for a small  $\lambda_{\min}/\lambda_{\max}$  ratio. The longest axis of the  $E = \text{const.}$  ellipsoid surface is proportional with  $\lambda_{\min}^{-1/2}$  while the shortest one is proportional with  $\lambda_{\max}^{-1/2}$ . The gradient is perpendicular to the  $E = \text{const.}$  surface. The weight vector is moving slowly, with oscillations towards the minima of  $E$ , center of the ellipsoids.

Considering the maximum possible value  $\eta = 2/\lambda_{\max}$ , and replacing into (12.8), the speed of learning will be decided by the time needed for the convergence of  $\alpha_i$  corresponding to the smallest eigenvalue, i.e. the size of factor  $\left(1 - \frac{2\lambda_{\min}}{\lambda_{\max}}\right)$ . If the ratio  $\lambda_{\min}/\lambda_{\max}$  is very small then the convergence will be slow. See figure 12.3.

So far, in equation (12.5), the time, given by  $t$ , was considered discrete. By considering continuous time the equation governing weights change during learning becomes:

$$\frac{dW}{dt} = -\eta \nabla E$$

which represents the equation of movement for a point in weights space, which position is given by  $W$  and subject to a potential field  $E(W)$  and viscosity  $1/\eta$ .

## 12.4.2 Momentum

By adding a term to the equation (12.5) and changing it to:

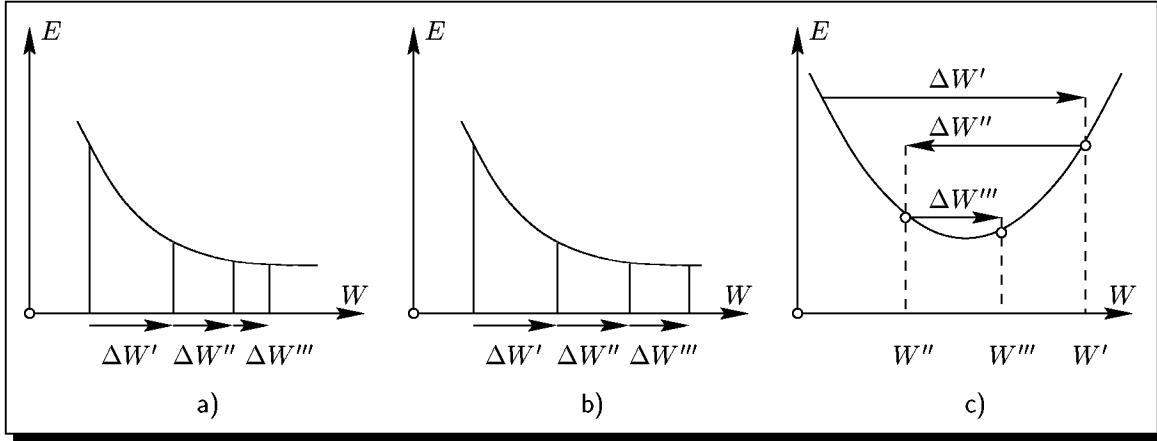
$$\Delta W_{(t+1)} = -\eta \nabla E|_{W_{(t)}} + \mu \Delta W_{(t)} \quad (12.10)$$

it is possible to increase the speed of convergence. The  $\mu$  parameter is named *momentum*.

❖  $\mu$

(12.10) may be rewritten into a continuous form as follows:

$$W_{(t+1)} - W_{(t)} = -\eta \nabla E|_{W_{(t)}} + \mu [W_{(t)} - W_{(t-1)}]$$



**Figure 12.4:** Learning with and without momentum ( $W'$ ,  $W''$  and  $W'''$  are 3 points, consecutive in time). Figure a) shows learning without momentum:  $\Delta W$  decreases in proportion to  $\nabla E$ , i.e. decreases around minima. Figure b) shows learning with momentum (without oscillations): the learning rate increases, compensating for the decrease of  $\nabla E$ . Figure c) shows oscillations; most of the additional quantities, introduced by momentum, cancels from one oscillation to the next.

$$\begin{aligned}
 W_{(t+1)} - W_{(t)} &= -\eta \nabla E|_{W_{(t)}} + \mu [W_{(t)} - W_{(t-1)}] \\
 &\quad - \mu [W_{(t+1)} - W_{(t)}] + \mu [W_{(t+1)} - W_{(t)}] \\
 \Delta W_{(t+1)} &= -\eta \nabla E|_{W_{(t)}} - \mu [\Delta W_{(t+1)} - \Delta W_{(t)}] + \mu \Delta W_{(t+1)} \\
 (1-\mu) \Delta W_{(t+1)} &= -\eta \nabla E|_{W_{(t)}} - \mu \Delta^2 W_{(t+1)}
 \end{aligned} \tag{12.11}$$

❖  $\Delta W$ ,  $\Delta^2 W$

where  $\Delta W_{(t+1)} = W_{(t+1)} - W_{(t)}$  and  $\Delta^2 W_{(t+1)} = \Delta W_{(t+1)} - \Delta W_{(t)}$ .

❖  $\tau$

By switching to the limit, the terms in (12.11) have to be of the same infinitesimal order, let  $\tau$  be equal to the unit of time (introduced for dimensionality correctness), then

$$(1-\mu) dW \frac{dt}{\tau} = -\eta \nabla E \frac{dt^2}{\tau^2} - \mu d^2 W$$

❖  $m$ ,  $\nu$

which gives finally the differential equation:

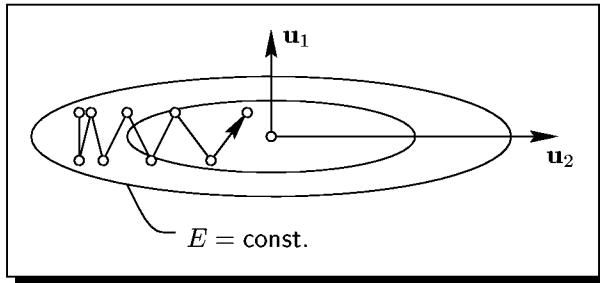
$$m \frac{d^2 W}{dt^2} + \nu \frac{dW}{dt} = -\nabla E, \quad \text{where } m = \frac{\mu \tau^2}{\eta}, \quad \nu = \frac{(1-\mu)\tau}{\eta} \tag{12.12}$$

pointing that the momentum shall be chosen  $\mu \in (0, 1)$ .



### Remarks:

- (12.12) 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  $\nu$  and into a conservative force field  $E$ .



**Figure 12.5:** The learning, alongside Hessian eigenvector corresponding to smallest eigenvalue, is accelerated comparing with plain gradient descent. See also figure 12.3 on page 221 for comparison.

$W$  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 fields.

To understand the effect given by the momentum — see also figure 12.4 on the preceding page — two cases may be analyzed:

- The gradient is constant  $\nabla E = \text{const.}$ . Then, by applying iteratively (12.10):

$$\Delta W = -\eta \nabla E (1 + \mu + \mu^2 + \dots) \simeq -\frac{\eta}{1 - \mu} \nabla E$$

(because  $\mu \in (0, 1)$  and then  $\lim_{n \rightarrow \infty} \mu^n = 0$ ), i.e. the learning rate effectively increases from  $\eta$  to  $\frac{\eta}{(1 - \mu)}$ .

- In a region with high curvature where the gradient change direction almost completely (opposite direction), generating weights oscillations, the effects of momentum will tend to cancel from one oscillation to the next.

The above discussion is also true on the components of vector  $W$ . The advancement in the direction of error minima alongside the direction of eigenvector corresponding to the smallest eigenvalue of Hessian is accelerated. See figure 12.5.

### 12.4.3 Other Gradient Descent Improvement Techniques

#### **Bold descent**

This algorithm improvement is based on the following idea:

- If, after a step, the error increases, i.e.  $\Delta E > 0$  this means that the minimum point was overshot. Then the change is rejected (reversed), because otherwise the weight value will be further from the minima point, and the learning rate is decreased.
- If, after a step, the error decreases, i.e.  $\Delta E < 0$  then the weight adjustment is accepted, but the learning rate is considered to be too small and consequently is increased.

The algorithm changes the learning rate, at each step, as follows:

$$\eta_{(t+1)} = \begin{cases} \rho\eta_{(t)} & \text{if } \Delta E < 0, \rho > 1 \\ \sigma\eta_{(t)} & \text{if } \Delta E > 0, \sigma < 1 \end{cases}$$

and the weight change is rejected if  $\Delta E > 0$ . In practice  $\rho \simeq 1.1$  and  $\sigma \simeq 0.5$ .

### Quick backpropagation

The quick backpropagation algorithm makes the following assumptions:

- The weights are independent. This is true only if the Hessian would be diagonal, i.e. its eigenvectors are parallel with the weights axes which in practice is seldom true (see also figure 12.3 on page 221 — the system  $\{\mathbf{u}_i\}$  is usually not parallel with the  $W$  system).
  - The error function may be approximated with a quadratic polynomial, i.e. a parabola  $E(w_i) = a + bw_i + cw_i^2$ ,  $a, b, c = \text{const.}$
- ❖  $a, b, c$

and then the weights are changed to the minima of error function by using 2 estimates of the error gradient.

Assuming the above hypotheses then:

$$\begin{aligned} E(w_i) = a + bw_i + cw_i^2 &\Rightarrow \frac{\partial E}{\partial w_i} = b + 2cw_i \\ \Rightarrow \begin{cases} \left. \frac{\partial E}{\partial w_i} \right|_t = b + 2cw_{i(t)} \\ \left. \frac{\partial E}{\partial w_i} \right|_{t-1} = b + 2cw_{i(t-1)} \end{cases} &\Rightarrow 2c = \frac{\left. \frac{\partial E}{\partial w_i} \right|_t - \left. \frac{\partial E}{\partial w_i} \right|_{t-1}}{w_{i(t)} - w_{i(t-1)}} \end{aligned}$$

At  $t+1$  the minimum is attended (as assumed):

$$\text{minimum} \Rightarrow \left. \frac{\partial E}{\partial w_i} \right|_t = 0 \Rightarrow w_{i(t+1)} = -\frac{b}{2c}$$

From the error gradient at  $t$ :

$$b = \left. \frac{\partial E}{\partial w_i} \right|_t - \frac{\left. \frac{\partial E}{\partial w_i} \right|_t - \left. \frac{\partial E}{\partial w_i} \right|_{t-1}}{w_{i(t)} - w_{i(t-1)}} w_{i(t)}$$

and then the weights update formula is:

$$\Delta w_{i(t+1)} = -\frac{\left. \frac{\partial E}{\partial w_i} \right|_t}{\left. \frac{\partial E}{\partial w_i} \right|_t - \left. \frac{\partial E}{\partial w_i} \right|_{t-1}} \Delta w_{i(t)}$$

$$(\Delta w_{i(t+1)} = w_{i(t+1)} - w_{i(t)}, \Delta w_{i(t)} = w_{i(t)} - w_{i(t-1)}).$$

#### Remarks:

- The algorithm assumes that the parabola have a minimum. If it have a maximum then the weights will be updated in the wrong direction.

- 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.

## 12.5 Line Search

The main idea of the algorithm is to search for the minimum of error function along the direction given by its negative gradient, instead of just move by a fixed amount given by the learning rate.

The weight vector at step  $t + 1$ , given the value at step  $t$ , is:

$$W_{(t+1)} = W_{(t)} - \lambda_{(t)} \nabla E|_{W_{(t)}}$$

where  $\lambda_{(t)}$  is a parameter, such that  $E(\lambda) = E(W_{(t)} - \lambda \nabla E|_{W_{(t)}})$  is minimum. ❖  $\lambda_{(t)}$

By the above approach the problem is reduced to a unidimensional case. The minimum of  $E(\lambda)$  may be found by searching for 3 points  $W'$ ,  $W''$  and  $W'''$  such that  $E(W') > E(W'')$  and  $E(W''') > E(W'')$ . Then the minimum is to be found somewhere between  $W'$  and  $W'''$  and may be found by approximation  $E$  with a quadratic polynomial, i.e. a parabola.



### Remarks:

- Another, less efficient but simpler, way would be to advance, along the direction given by  $-\nabla E$ , in small, fixed steps, and stop when error begin to increase.

## 12.6 Conjugate Gradients

### 12.6.1 Conjugate Search Directions

The method described in section 12.5 does not give the best direction for search of the minima in the weight space. Considering that the line search minimum have been reached at step  $t + 1$  then:

$$\frac{\partial E}{\partial \lambda} \Big|_{W_{(t+1)}} = \frac{\partial E}{\partial \lambda} \Big|_{W_{(t)} - \lambda_{(t)} \nabla E|_{W_{(t)}}} = 0 \Rightarrow (\nabla E|_{W_{(t+1)}})^T (\nabla E|_{W_{(t)}}) = 0$$

i.e. the gradients from two consecutive steps are orthogonal. See figure 12.6 on the following page. This behavior may lead to more (searching) steps than necessary.

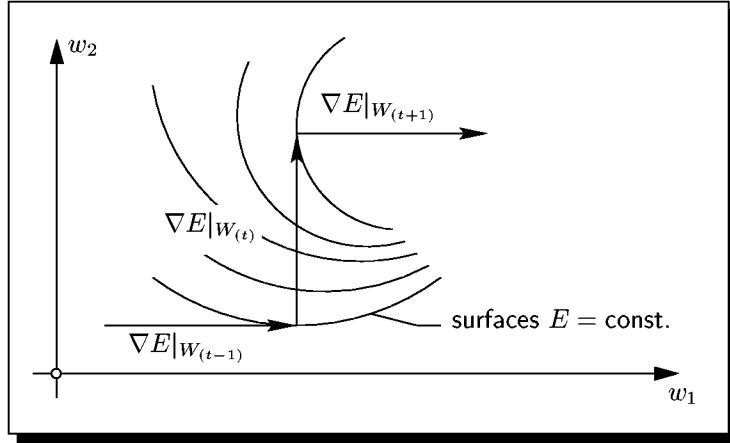
It is possible to continue the search from step  $t + 1$ , beyond, such that the component of the gradient parallel with the previous direction — and made 0 by the previous minimization step — is preserved to the lowest order. This means that a series of directions  $\{d_{(t)}\}$  have to be found such that:

$$(\nabla E|_{W_{(t+1)}})^T d_{(t)} = 0 \quad \text{and} \quad (\nabla E|_{W_{(t+1)} + \lambda d_{(t+1)}})^T d_{(t)} = 0$$

---

<sup>12.5</sup>See [Bis95] pp. 263–272.

<sup>12.6</sup>See [Bis95] pp. 274–285.



**Figure 12.6:** Line search learning. The error gradients from two consecutive steps are perpendicular.

and, by developing in series to the lowest order:

$$\left[ (\nabla E|_{W_{(t+1)}})^T + \lambda d_{(t+1)}^T H \right] d_{(t)} = 0 \Rightarrow \\ d_{(t+1)}^T H d_{(t)} = 0 \quad (12.13)$$

where  $H$  is the Hessian calculated at point  $W_{t+1}$ . Directions which respects condition (12.13) are named *conjugate* (or *interfering*).  
conjugate directions

### 12.6.2 Quadratic Error Function

A quadratic error function is of the form:

$$E(W) = E_0 + b^T W + \frac{1}{2} W^T H W \quad , \quad b, H = \text{const.}$$

and  $H$  (the Hessian) is symmetrical and positive definite.

The error gradient is:

$$\nabla E = b + HW \quad (12.14)$$

❖  $W^*$

and the minimum of  $E$  is achieved at the point  $W^*$  where:

$$\nabla E|_{W^*} = 0 \quad \Rightarrow \quad b + HW^* = 0 \quad (12.15)$$

❖  $N_W$

Let consider a set of  $\{d_i\}_{i=1, N_W}$  conjugate — with respect to  $H$  — directions ( $N_W$  being the total number of weights):

$$d_i^T H d_j = 0 \quad \text{for } i \neq j \quad (12.16)$$

and, of course,  $d_i \neq 0, \forall i$ .

**Proposition 12.6.1.** The  $\{d_i\}_{i=1, N_W}$  set of conjugate directions is linearly independent.

*Proof.* Let consider that there is a direction  $\mathbf{d}_\ell$  which is a linear combination of the other ones:

$$\mathbf{d}_\ell = \sum_{i,i \neq \ell} \alpha_i \mathbf{d}_i , \quad \alpha_i = \text{const.}$$

and then, from the way the set  $\{\mathbf{d}_i\}$  was chosen:

$$\mathbf{d}_\ell^T H \mathbf{d}_{q,q \neq \ell} = \left( \sum_{i,i \neq \ell} \alpha_i \mathbf{d}_i \right)^T H \mathbf{d}_{q,q \neq \ell} = \alpha_q \mathbf{d}_{q,q \neq \ell}^T H \mathbf{d}_{q,q \neq \ell} = 0$$

and as  $\mathbf{d}_{q,q \neq \ell}^T H \mathbf{d}_{q,q \neq \ell} \neq 0$  then  $\forall \alpha_{q,q \neq \ell} = 0$ , i.e.  $\mathbf{d}_\ell = 0$  which runs counter the way  $\{\mathbf{d}_i\}$  set was chosen.  $\square$



### Remarks:

- The above proposition ensures the existence of a set of  $N_W$  linear independent and  $H$  conjugate vectors.

As  $\{\mathbf{d}_i\}$  contains  $N_W$  vectors and is linear independent then it may be used as a reference system, i.e. any  $W$  vector from the weights space may be written as a linear combination of  $\{\mathbf{d}_i\}$ .

Let assume that the starting point for the search is  $W_1$  and the minimum point is  $W^*$ . Then it may be possible to write:

$$W^* - W_1 = \sum_{i=1}^{N_W} \alpha_i \mathbf{d}_i \quad (12.17)$$

where  $\alpha_i$  are some parameters. Finding  $W^*$  may be envisaged by a successive steps (of length  $\alpha_i$  along directions  $\mathbf{d}_i$ ) in the form:

$$W_{i+1} = W_i + \alpha_i \mathbf{d}_i \quad (12.18)$$

where  $i = \overline{1, N_W}$  and  $W_{N_W+1} = W^*$ .

By multiplying (12.17) with  $\mathbf{d}_\ell^T H$  to the left and using (12.15) plus the property (12.16):

$$\mathbf{d}_\ell^T H (W^* - W_1) = -\mathbf{d}_\ell^T \mathbf{b} - \mathbf{d}_\ell^T H W_1 = \sum_{i=1}^{N_W} \alpha_i \mathbf{d}_\ell^T H \mathbf{d}_i = \alpha_\ell \mathbf{d}_\ell^T H \mathbf{d}_\ell$$

and then the  $\alpha_\ell$  steps are:

$$\alpha_\ell = -\frac{\mathbf{d}_\ell^T (\mathbf{b} + H W_1)}{\mathbf{d}_\ell^T H \mathbf{d}_\ell} \quad (12.19)$$

The  $\alpha_\ell$  coefficients may be put into another form. From (12.18):  $W_{i+1} = W_1 + \sum_{j=1}^i \alpha_j \mathbf{d}_j$ ;

multiplying with  $\mathbf{d}_{i+1}^T H$  to the left and using again (12.16):

$$\mathbf{d}_{i+1}^T H W_{i+1} = \mathbf{d}_{i+1}^T H W_1 \Rightarrow$$

$$\mathbf{d}_{i+1}^T \nabla E|_{W_{i+1}} = \mathbf{d}_{i+1}^T (\mathbf{b} + H W_{i+1}) = \mathbf{d}_{i+1}^T (\mathbf{b} + H W_1)$$

and by using this result in (12.19):

$$\alpha_\ell = -\frac{\mathbf{d}_\ell^T \nabla E|_{W_\ell}}{\mathbf{d}_\ell^T H \mathbf{d}_\ell} \quad (12.20)$$

**Proposition 12.6.2.** If the weight vector is updated according to the procedure (12.18) the the gradient of the error function at step  $i + 1$  is orthogonal on all previous conjugate directions:

$$\mathbf{d}_j^T \nabla E|_{W_i} = 0, \forall j, i \text{ such that } j < i \leq N_W \quad (12.21)$$

*Proof.* From (12.14) and (12.18)

$$\nabla E|_{W_{i+1}} - \nabla E|_{W_i} = H(W_{i+1} - W_i) = \alpha_i H \mathbf{d}_i \quad (12.22)$$

and, by multiplying to the left with  $\mathbf{d}_i^T$  and replacing  $\alpha_i$  from (12.20), it follows that ( $\{\mathbf{d}_i\}$  are conjugate directions):

$$\mathbf{d}_i^T \nabla E|_{W_{i+1}} = 0 \quad (12.23)$$

On the other hand, by multiplying (12.22) with  $\mathbf{d}_j$  to the left:

$$\mathbf{d}_j^T (\nabla E|_{W_{i+1}} - \nabla E|_{W_i}) = \alpha_i \mathbf{d}_j^T H \mathbf{d}_i = 0, \forall j < i \leq N_W \quad (12.24)$$

The equation (12.24) is written for all instances  $i, i-1, \dots, j+1$  and then a summation is done over all equation obtained, which gives:

$$\mathbf{d}_j^T (\nabla E|_{W_{i+1}} - \nabla E|_{W_{j+1}}) = 0, \forall j < i \leq N_W$$

and, by using (12.23), finally:

$$\mathbf{d}_j^T \nabla E|_{W_{i+1}} = 0, \forall j < i \leq N_W$$

which combined with (12.23) (i.e. for  $j = i$ ) proves the desired result.  $\square$

The set of conjugate directions  $\{\mathbf{d}_i\}$  may be built as follows:

1. The first direction is chosen as:

$$\mathbf{d}_1 = -\nabla E|_{W_1}$$

2. The following directions are build incrementally as:

$$\mathbf{d}_{i+1} = -\nabla E|_{W_{i+1}} + \beta_i \mathbf{d}_i \quad (12.25)$$

❖  $\beta_i$

where  $\beta_i$  are coefficients to be found such that the newly build  $\mathbf{d}_{i+1}$  is conjugate with the previous  $\mathbf{d}_i$ , i.e.  $\mathbf{d}_{i+1}^T H \mathbf{d}_i = 0$ . By multiplying (12.25) with  $H \mathbf{d}_i$  to the right:

$$(-\nabla E|_{W_{i+1}} + \beta_i \mathbf{d}_i)^T H \mathbf{d}_i = 0 \Rightarrow \beta_i = \frac{(\nabla E|_{W_{i+1}})^T H \mathbf{d}_i}{\mathbf{d}_i^T H \mathbf{d}_i} \quad (12.26)$$

**Proposition 12.6.3.** By using the above method for building the set of directions, the error gradient at step  $j$  is orthogonal on all previous ones:

$$(\nabla E|_{W_j})^T \nabla E|_{W_i} = 0, \forall j, i \text{ such that } j < i \leq N_W \quad (12.27)$$

*Proof.* Obviously by the way of building, each direction vector represents a linear combination of all previously gradients, of the form:

$$\mathbf{d}_j = -\nabla E|_{W_j} + \sum_{\ell=1}^{j-1} \gamma_\ell \nabla E|_{W_\ell} \quad (12.28)$$

where  $\gamma_\ell$  are the coefficients of the linear combination (their exact value is not relevant to the proof).  $\diamond \gamma_\ell$

By multiplying (12.28) with  $\nabla_{\mathbf{w}} E|_{\mathbf{w}_i}$  to the right and using the result established in (12.21):

$$\left( \nabla E|_{W_j} \right)^T \nabla E|_{W_i} = \sum_{\ell=1}^{j-1} \gamma_\ell \left( \nabla E|_{W_\ell} \right)^T \nabla E|_{W_i} , \quad \forall j, i \text{ such that } j < i \leq N_W$$

For  $j = 1$  the error gradient equals the direction  $\mathbf{d}_1$  and, by using (12.21), the result (12.27) holds. For  $j = 2$ :

$$\left( \nabla E|_{W_2} \right)^T \nabla E|_{W_i} = \sum_{\ell=1}^{j-1} \gamma_\ell \left( \nabla E|_{W_\ell} \right)^T \nabla E|_{W_i} = \gamma_1 \left( \nabla E|_{W_1} \right)^T \nabla E|_{W_i} = 0$$

and so on, as long as  $j < i$  and thus the (12.27) is true.  $\square$

**Proposition 12.6.4.** *The (set of) directions build by the above method are mutually conjugate.*

*Proof.* It will be shown by induction.

By construction  $\mathbf{d}_2^T H \mathbf{d}_1 = 0$ , i.e. these directions are conjugate.

It is assumed (induction) that:

$$\mathbf{d}_i^T H \mathbf{d}_j = 0 , \quad \forall i, j \text{ such that } j < i \leq N_W$$

is true and it have to be shown that it holds for  $i + 1$  as well (assuming  $i + 1 \leq N_W$ , of course).

For  $\mathbf{d}_{i+1}$ , by using the above assumption and (12.25):

$$\mathbf{d}_{i+1}^T H \mathbf{d}_j = - \left( \nabla E|_{W_{i+1}} \right)^T H \mathbf{d}_j + \beta_i \mathbf{d}_i^T H \mathbf{d}_j = - \left( \nabla E|_{W_{i+1}} \right)^T H \mathbf{d}_j \quad (12.29)$$

$\forall j, i$  such that  $j < i \leq N_W$  (the second term disappears due to the induction assumption supposed true).

From (12.22),  $\nabla E|_{W_{j+1}} - \nabla E|_{W_j} = \alpha_j H \mathbf{d}_j$ . By multiplying this equation with  $\left( \nabla E|_{W_{i+1}} \right)^T$  to the left, and considering (12.27), i.e.  $\left( \nabla E|_{W_{i+1}} \right)^T \nabla E|_{W_j} = 0$  for  $\forall j < i + 1 < N_W$ , then:

$$\begin{aligned} & \left( \nabla E|_{W_{i+1}} \right)^T \left( \nabla E|_{W_{j+1}} - \nabla E|_{W_j} \right) \\ &= \left( \nabla E|_{W_{i+1}} \right)^T \nabla E|_{W_{j+1}} - \left( \nabla E|_{W_{i+1}} \right)^T \nabla E|_{W_j} = \alpha_j \nabla E|_{W_{i+1}}^T H \mathbf{d}_j = 0 \end{aligned}$$

and by inserting this result into (12.29) then:

$$\mathbf{d}_{i+1}^T H \mathbf{d}_j = 0 , \quad \forall j, i \text{ such that } j < i \leq N_W$$

and this result is extensible from  $j < i \leq N_W$  to  $j < i + 1 \leq N_W$  because of the way  $\mathbf{d}_{i+1}$  is build, i.e.  $\mathbf{d}_{i+1}^T H \mathbf{d}_i = 0$  by design.  $\square$



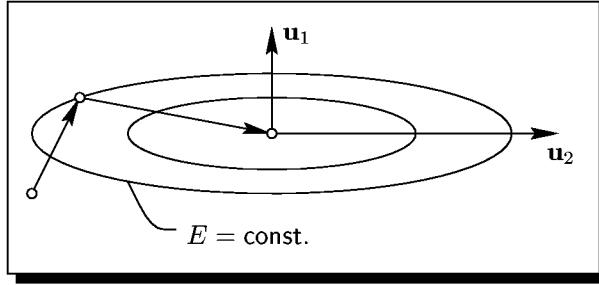
### Remarks:

- The method described in this section gives a very fast converging method for finding the error minima, i.e. the number of steps required equals the dimensionality of the weight space. See figure 12.7 on the next page.

### 12.6.3 The Algorithm

The previous section give the general method for fast finding the minima of  $E$ . However there are 2 remarks to be made:

- The error function was assumed to be quadratic.



**Figure 12.7:** Conjugate gradient learning. Into a bidimensional weight space it takes just 2 steps to reach the minimum.

- For a non-quadratic error function the Hessian is variable and then it have to be calculated at each  $W_i$  point which results into a very computational intensive process.

For the general algorithm it is possible to express the  $\alpha_i$  and  $\beta_i$  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  $\beta_i$  coefficients:

Hestenes-Stiefel

- *The Hestenes-Stiefel formula.* By replacing (12.22) into (12.26):

$$\beta_i = \frac{(\nabla E|_{W_{i+1}})^T (\nabla E|_{W_{i+1}} - \nabla E|_{W_i})}{\mathbf{d}_i^T (\nabla E|_{W_{i+1}} - \nabla E|_{W_i})} \quad (12.30)$$

Polak-Ribiere

- *The Polak-Ribiere formula.* From (12.25) and (12.21) and by making a multiplication to the right:

$$\begin{aligned} \mathbf{d}_i &= -\nabla E|_{W_i} + \beta_i \mathbf{d}_{i-1} \quad ; \quad \mathbf{d}_{i-1}^T \nabla E|_{W_i} = 0 \quad \Rightarrow \\ \mathbf{d}_i^T \nabla E|_{W_i} &= -(\nabla E|_{W_i})^T \nabla E|_{W_i} + \beta_i \mathbf{d}_{i-1}^T \nabla E|_{W_i} \\ &= -(\nabla E|_{W_i})^T \nabla E|_{W_i} \end{aligned}$$

and by using this result, together with (12.21) again, into (12.30), finally:

$$\beta_i = \frac{(\nabla E|_{W_{i+1}})^T (\nabla E|_{W_{i+1}} - \nabla E|_{W_i})}{(\nabla E|_{W_i})^T \nabla E|_{W_i}} \quad (12.31)$$

Fletcher-Reeves

- *The Fletcher-Reeves formula.* From (12.31) and using (12.27):

$$\beta_i = \frac{(\nabla E|_{W_{i+1}})^T \nabla E|_{W_{i+1}}}{(\nabla E|_{W_i})^T \nabla E|_{W_i}} \quad (12.32)$$



#### Remarks:

- In theory, i.e. for a quadratic error function, (12.30), (12.31) and (12.32) are equivalent. In practice, because the error function is seldom quadratic, they may give different results. Usually the Polak-Ribiere formula gives best result.

Let consider a quadratic error as function of  $\alpha_i$ :

$$E(W_i + \alpha_i \mathbf{d}_i) = E_0 + \mathbf{b}^T (W_i + \alpha_i \mathbf{d}_i) + \frac{1}{2} (W_i + \alpha_i \mathbf{d}_i)^T H (W_i + \alpha_i \mathbf{d}_i)$$

The minimum if error along the direction given by  $\mathbf{d}_i$  is found by imposing the cancellation of its derivative with respect to  $\alpha_i$ :

$$\frac{\partial E}{\partial \alpha_i} = 0 \Rightarrow \mathbf{b}^T \mathbf{d}_i + (W_i + \alpha_i \mathbf{d}_i)^T H \mathbf{d}_i = 0$$

and considering the property  $\mathbf{x}^T \mathbf{y} = \mathbf{y}^T \mathbf{x}$  and the fact that  $\nabla E = \mathbf{b} + HW$ , then:

$$\alpha_i = -\frac{\mathbf{d}_i^T \nabla E|_{W_i}}{\mathbf{d}_i^T H \mathbf{d}_i} \quad (12.33)$$

The fact that formula (12.33) coincide with expression (12.20) indicate that the procedure of finding these coefficients may be replaced with any procedure for finding the error minima along  $\mathbf{d}_i$  direction.

The general algorithm is:

1. Select an starting (into the weight space) point given by  $W_1$ .
2. Calculate  $\nabla E|_{W_1}$  and make:  $\mathbf{d}_1 = -\nabla E|_{W_1}$
3. For  $i = 1, \dots, (\text{max. value})$ :
  - (a) Find the minimum of  $E(W_i + \alpha_i \mathbf{d}_i)$  with respect to  $\alpha_i$  and move to the next point  $W_{i+1} = W_i + \alpha_{i(\min.)} \mathbf{d}_i$ .
  - (b) Evaluate the stop condition. It may be error drop under some specified value, a fixed number of steps, e.t.c.
  - (c) Calculate  $\nabla E|_{W_{i+1}}$  and then  $\beta_i$ , using one of the (12.30), (12.31) or (12.32). Finally calculate the new  $\mathbf{d}_{i+1}$  direction from (12.25).

(Cycle is to be repeated till the error minima have been found, or some maximal number of steps have been executed).

#### 12.6.4 Scaled Conjugated Gradients

The line search algorithm may have the following drawbacks:

- it may involve several calculations of the error function.
- it may be sensible on the accuracy of  $\alpha_i$  determination.

For a general network however it is possible to calculate the product between the Hessian and a vector, e.g.  $H \mathbf{d}_i$  in the conjugate gradient algorithm, in just  $\mathcal{O}(W)$  steps<sup>8</sup>. However there is a possibility that the Hessian is not positive definite, meaning that the denominator  $\mathbf{d}_\ell^T H \mathbf{d}_\ell$  of (12.20) may be negative and thus driving an increase of error.

The Hessian may be made positive definite by performing a translation of the form  $H \rightarrow \diamond \lambda$

---

<sup>8</sup>See chapter "Multi layer neural networks".

$H + \lambda I$  where  $I$  is the unit matrix and  $\lambda$  is a sufficiently large coefficient. The formula (12.20) then becomes:

$$\alpha_i = -\frac{\mathbf{d}_i^T \nabla E|_{W_i}}{\mathbf{d}_i^T H|_{W_i} \mathbf{d}_i + \lambda_i \|\mathbf{d}_i\|^2} \quad (12.34)$$

❖  $H|_{W_i}$ ,  $\lambda_i$

$H|_{W_i}$  being the Hessian calculated at point  $W_i$  and  $\lambda_i$  being the required value of  $\lambda$  to make the Hessian positive definite.

The problem is to choose the  $\lambda_i$  parameter. One way is to start from some value — which may be 0 as well — and to increase it till the denominator of (12.34) becomes positive. Let consider the denominator:

$$\delta_i = \mathbf{d}_i^T H \mathbf{d}_i + \lambda_i \|\mathbf{d}_i\|^2$$

❖  $\lambda'_i$ ,  $\delta'_i$

If  $\delta_i > 0$  then it can be used; otherwise ( $\delta_i < 0$ ) the  $\lambda_i$  parameter is increased to the new value  $\lambda'_i$  such that the new value of the denominator  $\delta'_i > 0$ :

$$\delta'_i = \delta_i + (\lambda'_i - \lambda_i) \|\mathbf{d}_i\|^2 > 0 \Rightarrow \lambda'_i > \lambda_i - \frac{\delta_i}{\|\mathbf{d}_i\|^2}$$

An interesting value to choose for  $\lambda'_i$  is  $\lambda'_i = 2 \left( \lambda_i - \frac{\delta_i}{\|\mathbf{d}_i\|^2} \right)$  which gives:

$$\delta'_i = -\delta_i + \lambda_i \|\mathbf{d}_i\|^2 = -\mathbf{d}_i^T H \mathbf{d}_i$$

### Remarks:

- ➡ If the error is quadratic then  $\lambda_i = 0$ . In the regions of the weights space where error is badly approximated by a quadratic  $\lambda_i$  have to be increased, in the regions where error is closer to the quadratic approximation  $\lambda_i$  may be decreased.
- ➡ One way to measure the degree of quadratic approximation is:

$$\Delta_i = \frac{E(W_i) - E(W_i + \alpha_i \mathbf{d}_i)}{E(W_i) - E_Q(W_i + \alpha_i \mathbf{d}_i)} \quad (12.35)$$

❖  $E_Q$

where  $E_Q$  is the local quadratic approximation:

$$E_Q(W_i + \alpha_i \mathbf{d}_i) = E(W_i) + \alpha_i \mathbf{d}_i^T \nabla E|_{W_i} + \frac{1}{2} \alpha_i^2 \mathbf{d}_i^T H|_{W_i} \mathbf{d}_i$$

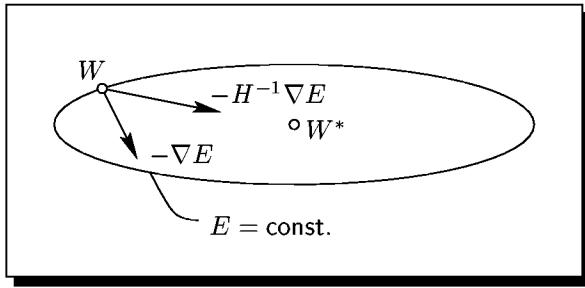
and, by replacing into (12.35) and using (12.20):

$$\Delta_i = \frac{2 [E(W_i) - E(W_i + \alpha_i \mathbf{d}_i)]}{\alpha_i \mathbf{d}_i^T \nabla E|_{W_i}}$$

- ➡ In practice the usual values used are:

$$\begin{cases} \lambda_{i+1} = \lambda_i / 2 & \text{if } \Delta_i > 0.75 \\ \lambda_{i+1} = \lambda_i & \text{if } 0.25 < \Delta_i < 0.75 \\ \lambda_{i+1} = 4\lambda_i & \text{if } \Delta_i < 0.25 \end{cases} \quad (12.36)$$

with the supplemental rule to *not* update the weights if  $\Delta_i < 0$  but just increase the value of  $\lambda_i$  and reevaluate  $\Delta_i$ .



**Figure 12.8:** Newton direction  $-H^{-1}\nabla E$  points directly to the error minima point  $W^*$ .

## 12.7 Newton's Method

For a local quadratic approximation around minima (12.2) of the error function, the gradient at some  $W$  point is  $\nabla E = H|_{W^*}(W - W^*)$  and then the minimum point is at:

$$W^* = W - H^{-1}|_{W^*} \nabla E \quad (12.37)$$

The vector  $-H^{-1}\nabla E$ , named *Newton direction*, points from the  $W$  point directly to the minimum  $W^*$ . See figure 12.8.

There are several points to be considered regarding the Newton's method:

- Since it is just an approximation, this method may require several steps to be performed to reach the *real* minimum.
- The exact evaluation of the Hessian is computationally intensive, of order  $\mathcal{O}(PW^2)$ ,  $P$  being the number of training vectors. The computation of the inverse Hessian  $H^{-1}$  is even more computationally intensive, i.e.  $\mathcal{O}(W^3)$ .
- The Newton direction may also point to a saddle point or maximum so checks should be made. This occurs if the Hessian is not positive definite.
- The point given by (12.37) may be outside the range of quadratic approximation, leading to instabilities in the learning process.

If the Hessian is positive definite then the Newton direction points towards a decrease of error — considering the derivative of error along Newton direction:

$$\begin{aligned} \frac{\partial E(W - \lambda H^{-1} \nabla E)}{\partial \lambda} &= - (H^{-1} \nabla E)^T \nabla E \\ &= - (\nabla E)^T H^{-1} \nabla E < 0 \end{aligned}$$

(the matrix property  $(AB)^T = B^T A^T$  was used here, as well as the fact that Hessian is symmetric).

If the Hessian is not positive definite the approach similar to that used in section 12.6.4 may be used, i.e.  $H \rightarrow H + \lambda I$ . This represents a compromise between gradient descent

<sup>12.7</sup>See [Bis95] pp. 285–287.

and Newton's direction:

- For  $\lambda \gtrsim 0 \Rightarrow H + \lambda I \simeq H$ , i.e. the new direction is close to Newton's direction.
- For  $\lambda \gg 0 \Rightarrow H + \lambda I \simeq \lambda I$ , and then  $-(\lambda I)^{-1} = -\frac{1}{\lambda} I$ , i.e. the new direction is close to  $\nabla E$ .

## 12.8 Levenberg-Marquardt Algorithm

This algorithm is specifically designed for “sum-of-squares” error function. Let  $\varepsilon_p$  be the error given by the  $p$ -th training pattern vector and  $\varepsilon^T = (\varepsilon_1 \dots \varepsilon_P)$ . The error function is then:

$$E = \frac{1}{2} \sum_{p=1}^P (\varepsilon_p)^2 = \frac{1}{2} \|\varepsilon\|^2 \quad (12.38)$$

❖  $\Upsilon$

Let consider the following matrix:

$$\Upsilon = \begin{pmatrix} \frac{\partial \varepsilon_1}{\partial W_1} & \dots & \frac{\partial \varepsilon_1}{\partial W_{N_W}} \\ \vdots & \ddots & \vdots \\ \frac{\partial \varepsilon_P}{\partial W_1} & \dots & \frac{\partial \varepsilon_P}{\partial W_{N_W}} \end{pmatrix}$$

then, considering a small variation in  $W$  weights from step  $t$  to  $t+1$ , the error vector  $\varepsilon$  may be developed in a Taylor series to the first order:

$$\varepsilon_{(t+1)} = \varepsilon_{(t)} + \Upsilon(W_{(t+1)} - W_{(t)})$$

and the error function at step  $t+1$  is:

$$E_{(t+1)} = \frac{1}{2} \|\varepsilon_{(t)} + \Upsilon(W_{(t+1)} - W_{(t)})\|^2 \quad (12.39)$$

Minimizing (12.39) with respect to  $W_{(t+1)}$  means:

$$\nabla E|_{W_{(t+1)}} = [\varepsilon_{(t)} + \Upsilon(W_{(t+1)} - W_{(t)})]^T \Upsilon = 0 \Rightarrow \varepsilon_{(t)} + \Upsilon(W_{(t+1)} - W_{(t)}) = 0$$

$\Upsilon$  is not square so first a multiplication with  $\Upsilon^T$  to the left is performed and then a multiplication by  $(\Upsilon^T \Upsilon)^{-1}$  again to the left which finally gives:

$$W_{(t+1)} = W_{(t)} - (\Upsilon^T \Upsilon)^{-1} \Upsilon^T \varepsilon_{(t)} \quad (12.40)$$

which represents the core of Levenberg-Marquardt weights update formula.

From (12.38) the Hessian matrix is:

$$\{H\}_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j} = \sum_{p=1}^P \left( \frac{\partial \varepsilon_p}{\partial w_i} \frac{\partial \varepsilon_p}{\partial w_j} + \varepsilon_p \frac{\partial^2 \varepsilon_p}{\partial w_i \partial w_j} \right)$$

---

<sup>12.8</sup>See [Bis95] pp. 290–292.

and by neglecting the second order derivatives the Hessian may be considered as:

$$H \simeq \Upsilon^T \Upsilon$$

i.e. the equation (12.40) essentially involves the inverse Hessian. However this is done through the computation of the error gradient with respect to weights which may be efficiently done by the backpropagation algorithm.

One problem should be taken care of: the formula (12.40) may give large values for  $\Delta W$ , i.e. so large that the (12.39) approximation no longer apply. To avoid this situation the following changed error function may be used instead:

$$E_{(t+1)} = \frac{1}{2} \|\varepsilon_{(t)} + \Upsilon(W_{(t+1)} - W_{(t)})\|^2 + \lambda \|W_{(t+1)} - W_{(t)}\|^2$$

where  $\lambda$  is a parameter governing the  $\Delta W$  size. By the same means as for (12.40) the new update formula becomes:

$$W_{(t+1)} = W_{(t)} - (\Upsilon^T \Upsilon + \lambda I)^{-1} \Upsilon^T \varepsilon_{(t)} \quad (12.41)$$

For  $\lambda \gtrsim 0$ , (12.41) approaches the Newton formula, for  $\lambda \gg 0$  (12.41) approaches the gradient descent formula.



### Remarks:

- For sufficiently large values of  $\lambda$  the error function is “guaranteed” to decrease since the direction of change is opposite to the gradient and the step is proportional with  $1/\lambda$ .
- The Levenberg-Marquardt algorithm is of *model trust region* type. The model — the linearized approximation error function — is “trusted” just around the current point  $W$ , the size of region being defined by  $\lambda$ .
- Practical values for  $\lambda$  are:  $\lambda \simeq 0.1$  at start then if error decrease multiply  $\lambda$  by 10; if the error increases go back (restore the old value of  $W$ , i.e. undo changes), divide  $\lambda$  by 10 and try again.



## CHAPTER 13

# Feature Extraction

### → 13.1 Pre/Post-processing

Usually the raw data is not feed directly into the ANN but rather processed before. The preprocessing have the following advantages:

- it allows for dimensionality reduction and thus avoid the course of dimensionality,
- it may include some *prior knowledge*, i.e. information additional to the training set.

Also the ANN outputs are also postprocessed to give the required output format. The pre and post processing may take any form, i.e. a statistical processing, some fixed transformation and/or even a processing involving another ANN.

The most important forms of preprocessing are:

- *dimensionality reduction* — it allows for building smaller ANN's and thus with less parameters and better suited to learn/generalize<sup>1</sup>,
- *feature extraction* — it involves making some combination of original training vector components called *features*; the process of calculating the features is named *feature extraction*, feature extraction

usually both processes going together, i.e. by dropping some vector components automatically those more “feature rich” will be kept and reciprocal the number of features extracted is usually much smaller than the dimension of the original pattern vector.

The preprocessing techniques described above will always drive to some loss of information. However the gain in accuracy neural computation outweighs this loss (of course, assuming that some care have been taken in the preprocessing phase). The main difficulty here is to find the right balance between the informational loss and the neural processing gain.

---

<sup>13.1</sup>See [Bis95] pp. 295–298.

<sup>1</sup>See the “Pattern Recognition” chapter, regarding the course of dimensionality

## 13.2 Input Normalization

One useful transformation is to scale the inputs such that they will be into the same order of magnitude.

❖  $\langle x_i \rangle, \sigma_i$

For each component  $x_i$ , of the input vector  $\mathbf{x}$ , the mean  $\langle x_i \rangle$  and variance  $\sigma_i^2$  are calculated:

$$\langle x_i \rangle = \frac{1}{P} \sum_{p=1}^P x_{ip} \quad , \quad \sigma_i^2 = \frac{1}{P-1} \sum_{p=1}^P (x_{ip} - \langle x_i \rangle)^2$$

and then the following transformation is applied:

$$\tilde{x}_{ip} = \frac{x_{ip} - \langle x_i \rangle}{\sigma_i} \quad (13.1)$$

❖  $\tilde{x}_{ip}$

where the new pattern introduced  $\tilde{x}_{ip}$  have zero mean and variance one:

$$\langle \tilde{x}_i \rangle = \frac{1}{P} \sum_{p=1}^P \tilde{x}_{ip} = 0 \quad , \quad \tilde{\sigma}_i^2 = \frac{1}{P-1} \sum_{p=1}^P (\tilde{x}_{ip} - \langle \tilde{x}_i \rangle)^2 = \frac{1}{P-1} \sum_{p=1}^P \tilde{x}_{ip}^2 = 1$$

A similar transformation may be applied to the target pattern vectors.

### Remarks:

- ➔ While the input normalization performed in (13.1) could be done into the first layer of the neural network, this preprocessing makes easier the initial choice of weights and thus learning: if the inputs and outputs are of order unity the the weights should also be of the same order of magnitude.

whitening

❖  $\langle \mathbf{x} \rangle, \Sigma$

Another, more sophisticated, transformation is *whitening*. The input training vector pattern set have the mean  $\langle \mathbf{x} \rangle$  and covariance matrix  $\Sigma$ :

$$\langle \mathbf{x} \rangle = \frac{1}{P} \sum_{p=1}^P \mathbf{x}_p \quad , \quad \Sigma = \frac{1}{P-1} \sum_{p=1}^P (\mathbf{x}_p - \langle \mathbf{x} \rangle)(\mathbf{x}_p - \langle \mathbf{x} \rangle)^T$$

❖  $U, \Lambda$

and considering the eigenvectors  $\{\mathbf{u}_i\}$  and eigenvalues  $\lambda_i$  of the covariance matrix:  $\Sigma \mathbf{u}_i = \lambda_i \mathbf{u}_i$  then the following transformation is performed:

$$\tilde{\mathbf{x}}_p = \Lambda^{-1/2} U^T (\mathbf{x}_p - \langle \mathbf{x} \rangle)$$

$$\text{where } U = \begin{pmatrix} u_{11} & \cdots & u_{1N} \\ \vdots & \ddots & \vdots \\ u_{N1} & \cdots & u_{NN} \end{pmatrix} \quad , \quad \Lambda = \begin{pmatrix} \lambda_1 & 0 & \cdots & 0 \\ 0 & \ddots & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \cdots & 0 & \lambda_N \end{pmatrix}$$

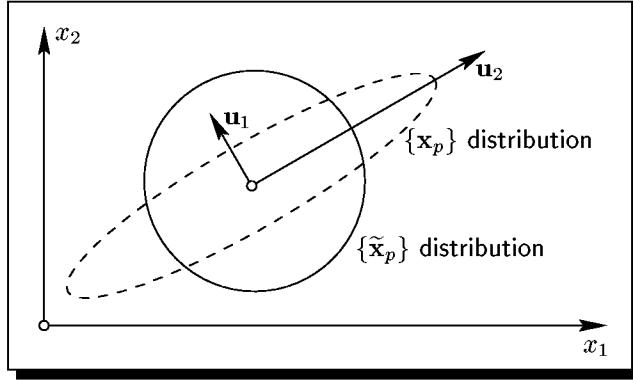
$u_{ij}$  being the component  $i$  of  $\mathbf{u}_j$ ;  $\{\Lambda^{-1/2}\}_{ij} = \frac{1}{\sqrt{\lambda_i}} \delta_{ij}$ . Because  $\Sigma$  is symmetric, it is possible to build an orthonormal  $\{\mathbf{u}_i\}$  set<sup>2</sup>:

$$\mathbf{u}_i \mathbf{u}_j = \delta_{ij} \quad \Rightarrow \quad U^T U = 1$$

---

<sup>13.2</sup>See [Bis95] pp. 298–300.

<sup>2</sup>See mathematical appendix.



**Figure 13.1:** The whitening process. The new distribution  $\{\tilde{\mathbf{x}}_p\}$  have a spherical symmetry and is centered in the origin — in the eigenvectors system of coordinates.

The mean of transformed pattern vectors is zero and the covariance matrix is unity:

$$\langle \tilde{\mathbf{x}} \rangle = \sum_{p=1}^P \tilde{\mathbf{x}}_p = 0 \quad (13.2)$$

$$\begin{aligned} \tilde{\Sigma} &= \frac{1}{P-1} \sum_{p=1}^P (\tilde{\mathbf{x}}_p - \langle \tilde{\mathbf{x}} \rangle)(\tilde{\mathbf{x}}_p - \langle \tilde{\mathbf{x}} \rangle)^T = \frac{1}{P-1} \sum_{p=1}^P \tilde{\mathbf{x}}_p \tilde{\mathbf{x}}_p^T = \Lambda^{-\frac{1}{2}} U^T \Sigma U \Lambda^{-\frac{1}{2} T} \\ &= \Lambda^{-\frac{1}{2}} U^T \Lambda U \Lambda^{-\frac{1}{2} T} = (\Lambda^{-\frac{1}{2}} U^T \Lambda^{\frac{1}{2} T}) (\Lambda^{\frac{1}{2}} U \Lambda^{-\frac{1}{2} T}) = U^T U = I \end{aligned}$$

( $\Lambda$  may be written as  $\Lambda = \Lambda^{\frac{1}{2}} \Lambda^{\frac{1}{2}}$ ;  $\Lambda^{\frac{1}{2}} U \Lambda^{-\frac{1}{2} T} = U$  is true — may be checked by direct multiplication — and, because of diagonal nature of  $\Lambda$  matrices they equal their transposed).

The (13.2) result shows that in the system of coordinates given by  $\{\mathbf{u}_i\}$  the transformed distribution of  $\tilde{\mathbf{x}}_p$  is centered in origin and have a spherical symmetry. See figure 13.1.



#### Remarks:

- The discussion in this section was around input vectors with a continuous spectrum. For discrete input values the above methods may be inappropriate. In these cases one possibility would be to use a one-of- $k$  encoding scheme similar to that used in classification.

## 13.3 Missing Data

The “missing data” problem appears when some components of some/all training vector are unknown.

<sup>13.3</sup>See [Bis95] pp. 301–302.

Several approaches may be taken to deal with the problem:

- *Discarding data.* If the process responsible of missing data is independent of data set and there is a (relatively) large quantity of data then the incomplete pattern vectors may be discarded from the training set. This is the most simple approach.
- *Filling in.* The missing data may be filled in with values calculated using various approaches:
  - Use the means over corresponding data for which values are known. However this approach is too simplistic and usually gives poor results.
  - Calculate the missing values by a regression method over known data. The drawback is that the regression function generated is noise-free and then it underestimates the covariance in data.
  - Use the EM (expectation-maximisation) algorithm<sup>3</sup>. where missing data may be treated as mixture components.
  - The general approach: integrate over the corresponding variables by weighting with the corresponding distribution. This involves the fact that the distribution itself is modeled.

Let consider that, from the pattern vector  $\mathbf{x}$ , one part  $\mathbf{x}_{(k)}$  is known and the rest  $\mathbf{x}_{(m)}$  is missing.

❖  $\mathbf{x}_{(k)}, \mathbf{x}_{(m)}$

Using the Bayes theorem, the posterior probability of  $\mathbf{x}_{(k)} \in \mathcal{C}_k$ , respectively  $\mathbf{x} \in \mathcal{C}_k$  are:

$$P(\mathcal{C}_k | \mathbf{x}_{(k)}) = \frac{p(\mathbf{x}_{(k)} | \mathcal{C}_k) P(\mathcal{C}_k)}{p(\mathbf{x}_{(k)})}$$

$$P(\mathcal{C}_k | \mathbf{x}_{(k)}, \mathbf{x}_{(m)}) = \frac{p(\mathbf{x}_{(k)}, \mathbf{x}_{(m)} | \mathcal{C}_k) P(\mathcal{C}_k)}{p(\mathbf{x}_{(k)}, \mathbf{x}_{(m)})}$$

Using the above equations, the posterior probability of class  $\mathcal{C}_k$ , while knowing only  $\mathbf{x}_{(k)}$ , may be expressed as:

$$\begin{aligned} P(\mathcal{C}_k | \mathbf{x}_{(k)}) &= \frac{p(\mathbf{x}_{(k)} | \mathcal{C}_k) P(\mathcal{C}_k)}{p(\mathbf{x}_{(k)})} = \frac{P(\mathcal{C}_k)}{p(\mathbf{x}_{(k)})} \int_{\mathbf{X}_{(m)}} p(\mathbf{x}_{(k)}, \mathbf{x}_{(m)} | \mathcal{C}_k) d\mathbf{x}_{(m)} \\ &= \frac{1}{p(\mathbf{x}_{(k)})} \int_{\mathbf{X}_{(m)}} P(\mathcal{C}_k | \mathbf{x}_{(k)}, \mathbf{x}_{(m)}) p(\mathbf{x}_{(k)}, \mathbf{x}_{(m)}) d\mathbf{x}_{(m)} \end{aligned}$$

## 13.4 Time Series Prediction

The time series prediction problem involves the prediction of a pattern vector  $\mathbf{x}(\tau)$  based of the knowledge of previous behavior of the variable.

<sup>3</sup>Described in “Pattern Recognition” chapter.

<sup>13.4</sup>See [Bis95] pp. 302–304.

The following approaches are common:

- *The static method.* It is assumed that the statistical properties of the data generator do not evolve in time.

The pattern vector is sampled in time, at regular interval, resulting a series of values, i.e. time is converted to a discrete form:  $\dots, \mathbf{x}_{\tau-2}, \mathbf{x}_{\tau-1}, \mathbf{x}_\tau, \dots$

The training set is build by using one value as output and some  $n$  previous values as input, e.g.  $\mathbf{x}_\tau$  is used as output and  $\mathbf{x}_{\tau-n}, \dots, \mathbf{x}_{\tau-1}$  as inputs; then  $\mathbf{x}_{\tau+1}$  is used as output and  $\mathbf{x}_{\tau-n+1}, \dots, \mathbf{x}_\tau$  as inputs, e.t.c.

The first predicted value may be used as output to make the second prediction, and so on.

- *Detrending.* The time series may have a simple trend, e.g. increasing or decreasing in time. This may lead to a poor prediction over time. However, by fitting a simple curve to the data and then removing the value predicted by this simple model the data are detrended. Only the more complex model (assumed constant in time) remains.
- *The dynamical approach.* This involves a method which allows for retraining in time and adaptation to the data generator as it changes in time.

## 13.5 Feature Selection

Usually, only a subset of the full feature set is used in the training process of ANN. As there may be many combinations of features, a selection criteria have to be applied in order to find the most fitted subset of features (the individual input vector components may be seen as features as well).

One procedure of selection is to train the network on different sets of features and then to test for generalization achieved on a set of pattern vectors not used at learning stage.

As the training and testing may be time consuming on a complex model, an alternative is to use a simplified one for this purpose.

It is to be expected — due to the curse of dimensionality — that there is some optimal minimum set of features which gives the best generalization. For less features there is too little information and for more features the dimensionality course intervene.

On the other hand usually a criteria  $J$  used in class separability increases with the number of features  $X$ :  $\diamond J, X$

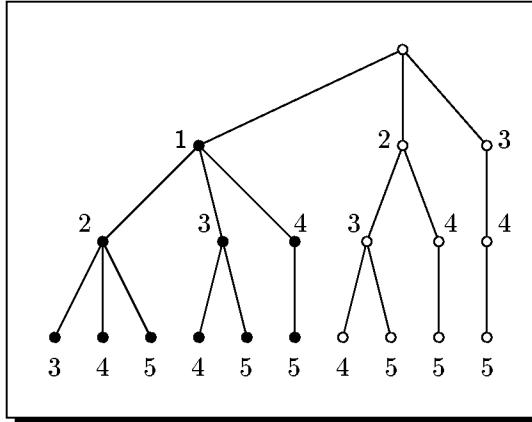
$$J(X_+) \geq J(X) \quad \text{if } X \subseteq X_+ \quad (13.3)$$

e.g. the Mahalanobis distance<sup>4</sup>  $\Delta^2$ . This means that this kind of criteria can't be used directly to compare the results given by two, different in size, feature sets.

Assuming that there are  $N$  features, there are  $2^N$  possible subsets (2 because a feature may be present or absent from the subset, the whole set may be also considered as a subset). Considering that the subset is required to have exactly  $\tilde{N}$  features the number of possible combination is still  $\frac{N!}{(N-\tilde{N})! \tilde{N}!}$ . In principle, an exhaustive search trough all possible

<sup>13.5</sup> See [Bis95] pp. 304–310.

<sup>4</sup>Defined in chapter "Pattern Recognition"



**Figure 13.2:** The branch and bound decision tree. Build for a particular case of 5 features. If at one level a feature, e.g. 1, can not be eliminated then the whole subtree, marked with black dots, is removed from the search.

combinations should be made in order to find the best subset of feature. In practice even a small number of features will generate a huge number of subset combinations, too big to be fully checked.

There are several methods to avoid evaluating the whole feature combination sets.

#### **The branch and bound method**

This method gives the optimal solution based on a criteria for which (13.3) is true.

Let assume that there are  $N$  features and  $\tilde{N}$  features to be selected, i.e. there are  $N - \tilde{N}$  features to be dropped.

A top-down decision tree is build. It starts with one node and have  $N - \tilde{N}$  levels (not counting the root itself). At each level one feature is dropped such that at the bottom there are only  $\tilde{N}$  left. See figure 13.2.

Note that as the number of features to remain is  $\tilde{N}$  then the first level have only  $N - \tilde{N}$ . It doesn't matter what features are present on this level as the order of elimination is not important. For the same reason one feature eliminated at one level does not appear into the sub-trees of other eliminated features.

The elimination works as follows:

- A random combination of features is selected, i.e. one point from the bottom of the decision tree. The corresponding criteria used in class separability  $J(X_0)$  is calculated. See (13.3).
- Then one feature is eliminated at a time, going from top to the bottom of the tree. The criteria  $J(X)$  is calculated at each level.

If at some level  $J(X) > J(X_0)$  then continue the search. Otherwise ( $J(X) < J(X_0)$ ) there are are the following possibilities:

- The node is at bottom of the tree. The new feature combination is better then the old one and becomes the new level of comparison, i.e.  $J(X) \rightarrow J(X_0)$ .

- If the node is not at the bottom of the tree then the whole subtree is eliminated from search as being suboptimal. The condition (13.3) is no more met, i.e. a feature (or combination of features) which shouldn't have been eliminated just was and all combinations which contain the elimination of that feature shouldn't be considered further.

### **The sequential search**

This method may give suboptimal solutions but is faster than the previous one. It is based on considering one feature at a time.

There are two ways of selection:

- At each step one feature — the one which gives biggest  $J(X)$  criterion — is added to the (initial empty) set. The method is named *sequential forward selection*.  
sequential forward selection
- At each step the least important feature — the one which gives the smaller decrease of  $J(X)$  criterion — is eliminated from the (initial full) set. The method is named *sequential backward elimination*.  
sequential backward elimination

## 13.6 Dimensionality Reduction

This procedure tries to reduce the dimensionality of input data by combining them using an unsupervised learning process.

### 13.6.1 Principal Component Analysis

The problem is to map a set of input vectors  $\{\mathbf{x}_p\}$ , which are  $N$  dimensional, into a set of corresponding vectors  $\{\mathbf{z}_p\}$  which have a lower dimensionality  $K < N$ .

The  $\mathbf{x}$  input vector may be represented by the means of an orthonormal set of vectors  $\{\mathbf{u}_i\}$ :

$$\mathbf{x} = \sum_{i=1}^N z_i \mathbf{u}_i \quad \Rightarrow \quad z_i = \mathbf{u}_i^T \mathbf{x} \quad \text{where} \quad \mathbf{u}_i^T \mathbf{u}_j = \delta_{ij}$$

A transformation  $\mathbf{x} \rightarrow \tilde{\mathbf{x}}$  is performed as follows: from the set  $\{z_i\}$  only  $K$  are retained (e.g. the first ones) and the others are replaced with constants  $b_i$

❖  $\tilde{\mathbf{x}}, b_i$

$$\tilde{\mathbf{x}} = \sum_{i=1}^K z_i \mathbf{u}_i + \sum_{i=K+1}^N b_i \mathbf{u}_i \quad , \quad b_i = \text{const.}$$

which practically represents a dimensionality reduction from  $N$  to  $K$ .

The problem is now to select the best  $K$  set of components from  $\mathbf{x}$ . This can be done by trying to minimize the error when switching from  $\mathbf{x}$  to  $\tilde{\mathbf{x}}$ , i.e. the difference between the two vectors:

$$\mathbf{x} - \tilde{\mathbf{x}} = \sum_{i=K+1}^N (z_i - b_i) \mathbf{u}_i$$

---

<sup>13.6</sup>See [Bis95] pp. 310–316.

❖  $E_K$

and for a set of  $P$  input vectors

$$E_K = \frac{1}{2} \sum_{p=1}^P \|\mathbf{x}_p - \tilde{\mathbf{x}}_p\|^2 = \frac{1}{2} \sum_{p=1}^P \sum_{i=K+1}^N (z_{ip} - b_i)^2 \quad (13.4)$$

From the condition of minimum of  $E_K$  with respect to  $b_i$

$$\frac{\partial E_K}{\partial b_i} = 0 \Rightarrow b_i = \frac{1}{P} \sum_{p=1}^P z_{ip} = \mathbf{u}_i^T \langle \mathbf{x} \rangle$$

❖  $\langle \mathbf{x} \rangle$

where:  $\langle \mathbf{x} \rangle = \frac{1}{P} \sum_{p=1}^P \mathbf{x}_p$  is the mean input vector.

Then the error (13.4) may be written as (use  $(AB)^T = B^T A^T$  matrix property):

$$E_K = \frac{1}{2} \sum_{p=1}^P \sum_{i=K+1}^N [\mathbf{u}_i^T (\mathbf{x}_p - \langle \mathbf{x} \rangle)]^2 = \frac{1}{2} \sum_{i=K+1}^N \mathbf{u}_i^T \Sigma \mathbf{u}_i \quad (13.5)$$

❖  $\Sigma$

where  $\Sigma$  is the covariance matrix of input vectors  $\{\mathbf{x}_p\}$ :

$$\Sigma = \sum_{p=1}^P (\mathbf{x}_p - \langle \mathbf{x} \rangle)(\mathbf{x}_p - \langle \mathbf{x} \rangle)^T \quad (13.6)$$

The minima of  $E_K$  with respect to  $\{\mathbf{u}_i\}$  occurs when the this set is made from the eigenvectors of covariance matrix<sup>5</sup>  $\Sigma$ :

$$\Sigma \mathbf{u}_i = \lambda_i \mathbf{u}_i \quad (13.7)$$

❖  $\lambda_i$

$\lambda_i$  being the eigenvalues. By replacing (13.7) into (13.5) and using the orthogonality of  $\{\mathbf{u}_i\}$  the error becomes:

$$E_K = \frac{1}{2} \sum_{i=K+1}^N \lambda_i \quad (13.8)$$

and is minimized by choosing the smallest  $N - K$  eigenvalues.

Let consider the translation of coordinates from whatever  $\{\mathbf{x}_p\}$  was represented to the one defined by the eigenvectors  $\{\mathbf{u}_i\}$  and in the same time a translation of the origin of the new system to  $\langle \mathbf{x} \rangle$ , i.e. in the new system  $\langle \mathbf{x} \rangle = \hat{\mathbf{0}}$ ; this means that each  $\mathbf{x}_p$  may be represented as linear combination of  $\{\mathbf{u}_i\}$ :

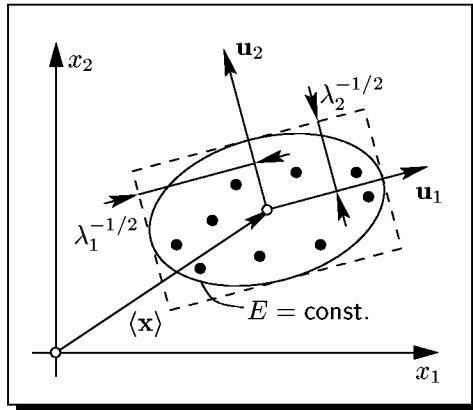
$$\mathbf{x}_p = \sum_{i=1}^N \alpha_{ip} \mathbf{u}_i$$

and by replacing in (13.5), considering also the orthogonality of  $\{\mathbf{u}_i\}$ ,  $E_K = \frac{1}{2} \sum_{p=1}^P \sum_{i=K+1}^N \alpha_{ip}^2$ ,

i.e.  $E_K$  is a quadratic form. From (13.8) it follows that the surfaces  $E_K = \text{const.}$  are hyperellipses with the axes proportional with  $1/\sqrt{\lambda_i}$  and the dimensionality reduction is done by

---

<sup>5</sup>See the mathematical appendix.



**Figure 13.3:** The constant-error surfaces are hyper-ellipses. The dimensionality reduction is done by projecting the data points (black dots) on the axes corresponding to the smallest eigenvalue  $\lambda_i$ , i.e. in this bidimensional case on  $\mathbf{u}_1$

dropping those dimensions corresponding to the smallest axes, i.e. by making a projection on the axes corresponding to largest  $\lambda_i$  representing the largest spread of data points. See figure 13.3.



#### Remarks:

- ➔ The method described is also known as the *Karhunen-Loéve transformation*. The  $\mathbf{u}_i$  are named principal components.

### 13.6.2 Non-linear Dimensionality Reduction Through ANN

The principal component analysis performed in the previous section may reduce the dimensionality only through a *linear* process.

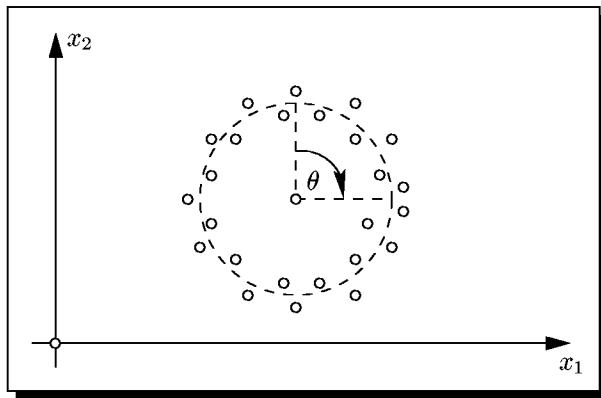
Sometimes the data have an intrinsic dimensionality which cannot be retrieved by linear methods. See figure 13.4 on the following page.

Auto-associative ANN may be used to perform dimensionality reduction. The input patterns are used also as targets (hence the “auto-associative” name) but the network have a “bottleneck” built into hidden layers. See figure 13.5 on the next page. The hidden layers have less neurons than input and output layers thus forcing the network to “squeeze” the data through, thus achieving a dimensionality reduction — the output of hidden layer representing the dimensionally reduced input data.

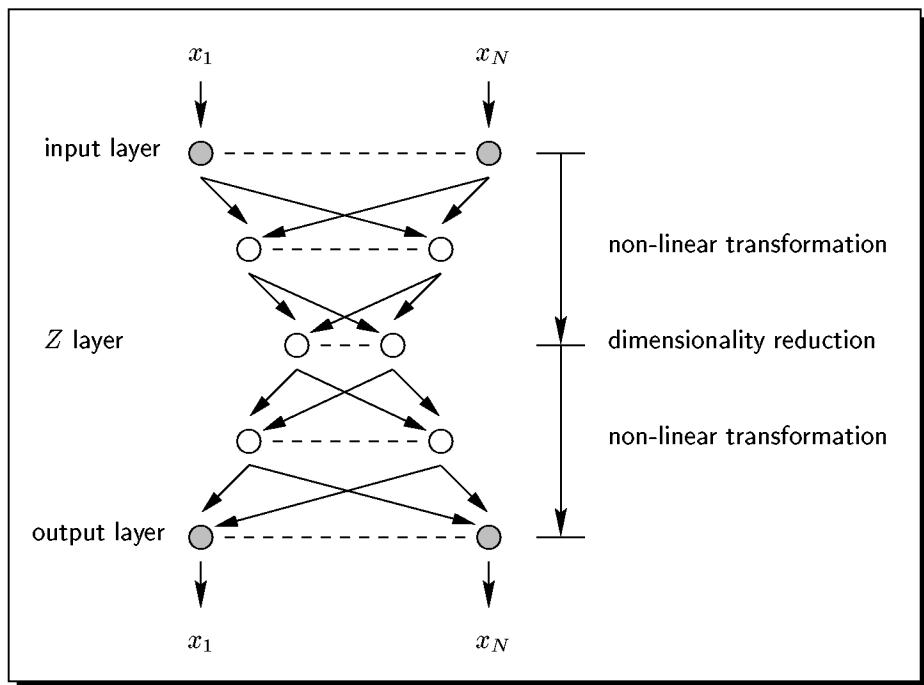


#### Remarks:

- ➔ The error function used is usually the sum-of-squares.
- ➔ If the network contains *only one hidden layer* then the transformation performed is *linear*, i.e. equivalent to principal component analysis. Unless it is “hardware” implemented there is no reason to use single hidden layer ANNs to perform dimensionality reduction.



**Figure 13.4:** Sometimes the data have an intrinsic dimensionality which cannot be revealed by a linear transformation. In this case the data points are distributed on a circular shape and could be described by one dimension, e.g. the angle  $\theta$  measured from one from a reference point.



**Figure 13.5:** The use of auto-associative ANNs for dimensionality reduction. The network have a bottleneck, i.e. the hidden layers have less neurons than input and output layers. The dimensionality reduction is achieved in layer  $Z$ .

► The same design may be used for data compression.

## 13.7 Invariance

In some applications of ANNs the output should be invariant to some transformation of the input, e.g. the classification of the shape of an object should be invariant to the position/rotation of it.

The most straightforward approach — and also the most inefficient — is to include into the training set as many examples of the transformed input as possible. This requires a large amount of training data and gives poor result for transformations uncovered well by the learning set. Various more effective alternatives have been developed.

### 13.7.1 The Tangent Prop Method

This method is applicable for continuous transformations, e.g. translations and rotations but not mirroring.

Let assume that the transformation of a vector  $\mathbf{x}$  leads to vector  $\mathbf{s}$  and is governed by one scalar parameter  $\alpha$ , e.g. rotation is defined by the angle parameter. Then  $\mathbf{s} = \mathbf{s}(\mathbf{x}, \alpha)$  and let  $\mathcal{M}$  be the reunion of all  $\mathbf{s}$  vectors for a given  $\mathbf{x}$  and all possible values of  $\alpha$ . Also the "natural" condition  $\mathbf{s}(\mathbf{x}, 0) = \mathbf{x}$  is imposed.

❖  $\mathbf{s}, \alpha, \mathcal{M}$

Let  $\tau$  be the vector defined as

$$\tau = \left. \frac{\partial \mathbf{s}(\mathbf{x}, \alpha)}{\partial \alpha} \right|_{\alpha=0}$$

❖  $\tau$

The change in network output, due to the transformation defined by  $\alpha$  is:

$$\frac{\partial y_j}{\partial \alpha} = \sum_{i=1}^N \frac{\partial y_j}{\partial x_i} \frac{\partial x_i}{\partial \alpha} = \sum_{i=1}^N \frac{\partial y_j}{\partial x_i} \tau_i = \sum_{i=1}^N J_{ji} \tau_i \quad (13.9)$$

❖  $J$

$$J \equiv \left\{ \frac{\partial y_j}{\partial x_i} \right\}_{\substack{i=1, N \\ j=1, K}} = \begin{pmatrix} \frac{\partial y_1}{\partial x_1} & \dots & \frac{\partial y_1}{\partial x_N} \\ \vdots & \ddots & \vdots \\ \frac{\partial y_K}{\partial x_1} & \dots & \frac{\partial y_K}{\partial x_N} \end{pmatrix}$$

If the network mapping function is invariant to the transformation in the vicinity of each pattern vector  $\mathbf{x}_p$  then the term (13.9) is heading towards zero and it may be used to build a regularization function<sup>7</sup> of the form:

$$\Omega = \frac{1}{2} \sum_{p=1}^P \sum_{j=1}^K (J_{jp} \tau_{ip})^2$$

where  $J_{jp}$  and  $\tau_{ip}$  are referring to the input vector  $\mathbf{x}_p$  from the training set  $\{\mathbf{x}_p\}_{p=1, P}$ .

❖  $J_{jp}, \tau_{ip}$

<sup>13.7</sup> See [Bis95] pp. 319–329.

<sup>6</sup>The Jacobian is defined in the "Multi layer neural networks" chapter.

<sup>7</sup>See chapter "Pattern recognition".

The new error function becomes:  $\tilde{E} = E + \nu\Omega$  where  $\nu$  is a regularization parameter determining the influence of  $\Omega$ .



### Remarks:

- In practice the derivative defining  $\tau$  is found by the mean of finite difference between  $x$  and  $s$  obtained from a (relative) small  $\alpha$ .

## 13.7.2 Preprocessing

The invariant features may be extracted at preprocessing stage. These features are named *moments*.

❖  $x, u_i$

Let consider one particular component  $x$  of the input vector, described in some system of coordinates and let  $\{u_i\}$  be the coordinates of  $x$ , e.g.  $x$  may be a point on a bidimensional image and  $\{u_1, u_2\}$  may be the Cartesian coordinates, the point being either “lit” ( $x = 1$ ) or not ( $x = 0$ ) depending upon coordinates  $(u_1, u_2)$ .

❖  $M$

The moment  $M$  is defined as:

$$M = \int_{U_1} \cdots \int_{U_N} x(u_1, \dots, u_N) K(u_1, \dots, u_N) du_1 \dots du_N$$

kernel function

where  $K(u_1, \dots, u_N)$  is named *kernel function* and determines the moments to be considered. For discrete spaces the integrals changes to sums:

$$M = \sum_{i_1} \cdots \sum_{i_N} x(u_{1(i_1)}, \dots, u_{N(i_N)}) K(u_{1(i_1)}, \dots, u_{N(i_N)}) \Delta u_{1(i_1)} \dots \Delta u_{N(i_N)}$$

(the values being taken at the points  $i_1, \dots, i_N$ ).

regular moments

One of the possible kernel functions is the power function which give rise to *regular moments*  $M_{i_1, \dots, i_N}$ :

$$M_{i_1, \dots, i_N} = \int_{U_1} \cdots \int_{U_N} x(u_1, \dots, u_N) u_1^{i_1} \dots u_N^{i_N} du_1 \dots du_N$$

❖  $\bar{u}_i$

and by defining  $\bar{u}_i = \frac{M_{0, \dots, 1, \dots, 0}}{M_{0, \dots, 0}}$  (1 in  $i$ -th position) then:

$$\widehat{M}_{i_1, \dots, i_N} = \int_{U_1} \cdots \int_{U_N} x(u_1, \dots, u_N) (u_1 - \bar{u}_1)^{i_1} \dots (u_N - \bar{u}_N)^{i_N} du_1 \dots du_N \quad (13.10)$$

central moments

which is named *central moment*.

The central moment, defined in (13.10), is invariant to translations, i.e. to transformations of type  $x(u_1, \dots, u_N) \rightarrow x(u_1 + \Delta u_1, \dots, u_N + \Delta u_N)$ , provided that the edge effects are neglected, or the  $U_i$  domains are infinite.

*Proof.* Indeed, the moment calculated for the new pattern is:

$$\widetilde{M}_{i_1, \dots, i_N} = \int_{U_1} \cdots \int_{U_N} x(u_1 + \Delta u_1, \dots, u_N + \Delta u_N) (u_1 - \bar{u}_1)^{i_1} \dots (u_N - \bar{u}_N)^{i_N} du_1 \dots du_N \quad (13.11)$$

and by making the change of variable  $u_i \rightarrow u_i + \Delta u_i = \tilde{u}_i$  then  $du_i = d\tilde{u}_i$ , also  $\tilde{u}_i = \bar{u}_i + \Delta u_i$ , and by replacing in (13.11):

$$\widetilde{\widehat{M}}_{i_1, \dots, i_N} = \int_{U_1} \cdots \int_{U_N} x(\tilde{u}_1, \dots, \tilde{u}_N) (\tilde{u}_1 - \bar{u}_1)^{i_1} \cdots (\tilde{u}_N - \bar{u}_N)^{i_N} d\tilde{u}_1 \cdots d\tilde{u}_N = \widehat{M}_{i_1, \dots, i_N} \quad \square$$

A moment  $\mu_{i_1, \dots, i_N}$  invariable to uniform scaling, i.e.  $x(u_1, \dots, u_N) \rightarrow x(\alpha u_1, \dots, \alpha u_N)$   $\diamond \mu_{i_1, \dots, i_N}, \alpha$  where  $\alpha$  is the scaling parameter, may be built as:

$$\mu_{i_1, \dots, i_N} = \frac{\widehat{M}_{i_1, \dots, i_N}}{\widehat{M}_{0, \dots, 0}^{1+(i_1+\dots+i_N)/N}} \quad (13.12)$$

Because the  $\mu_{i_1, \dots, i_N}$  moment is build using only the central moments  $\widehat{M}_{i_1, \dots, i_N}$  then it is automatically invariant to translations.

*Proof.* Let consider a scaling defined by  $\alpha$ . Similarly as for translation:

$$\begin{aligned} \widetilde{\widetilde{M}}_{i_1, \dots, i_N} &= \int_{U_1} \cdots \int_{U_N} x(\alpha u_1, \dots, \alpha u_N) (u_1 - \bar{u}_1)^{i_1} \cdots (u_N - \bar{u}_N)^{i_N} du_1 \cdots du_N \\ &= \frac{1}{\alpha^{N+i_1+\dots+i_N}} \int_{U_1} \cdots \int_{U_N} x(\alpha u_1, \dots, \alpha u_N) (\alpha u_1 - \alpha \bar{u}_1)^{i_1} \cdots (\alpha u_N - \alpha \bar{u}_N)^{i_N} \times \\ &\quad \times d(\alpha u_1) \cdots d(\alpha u_N) \\ &= \frac{1}{\alpha^{N+i_1+\dots+i_N}} \int_{U_1} \cdots \int_{U_N} x(\tilde{u}_1, \dots, \tilde{u}_N) (\tilde{u}_1 - \bar{u}_1)^{i_1} \cdots (\tilde{u}_N - \bar{u}_N)^{i_N} d\tilde{u}_1 \cdots d\tilde{u}_N \\ &= \frac{\widehat{M}_{i_1, \dots, i_N}}{\alpha^{N+i_1+\dots+i_N}} \end{aligned} \quad (13.13)$$

By the same means, for  $\widehat{M}_{0, \dots, 0}$  which is:  $\widehat{M}_{0, \dots, 0} = \int_{U_1} \cdots \int_{U_N} x(u_1, \dots, u_N) du_1 \cdots du_N$  it gives that:

$$\widetilde{\widetilde{M}}_{0, \dots, 0} = \frac{\widehat{M}_{0, \dots, 0}}{\alpha^N} \quad (13.14)$$

Finally, by using (13.13) and (13.14) into (13.12) it gives that  $\widetilde{\mu}_{i_1, \dots, i_N} = \mu_{i_1, \dots, i_N}$ , i.e. the  $\mu$  moment is invariant to scaling as well.  $\square$

It is possible also to build a moment which is independent to rotation. First the  $M$  moment is rewritten in generalized spherical coordinates<sup>8</sup>:

$$M_{i_1, \dots, i_N} = \int_0^\infty \int_0^{2\pi} \cdots \int_0^{2\pi} x(r, \theta_1, \dots, \theta_N) (r \cos \theta_1)^{i_1} \cdots (r \cos \theta_N)^{i_N} r dr d\theta_1 \cdots d\theta_N$$

As  $\sum_{i=1}^N \cos^2 \theta_i = 1$  then the moment:

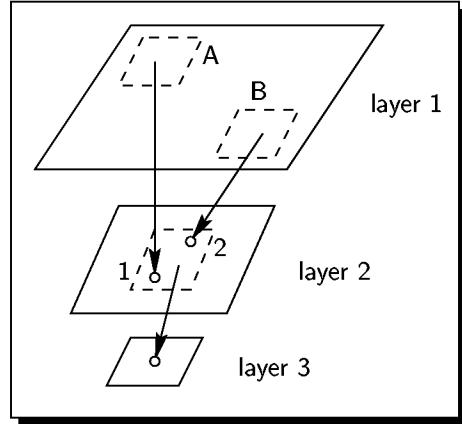
$$M_R = M_{2,0,\dots,0} + M_{0,2,0,\dots,0} + \dots + M_{0,\dots,0,2}$$

is rotation independent and thus the moment:

$$\mu_R = \mu_{2,0,\dots,0} + \mu_{0,2,0,\dots,0} + \dots + \mu_{0,\dots,0,2}$$

---

<sup>8</sup>See mathematical appendix.



**Figure 13.6:** The shared weights method for bidimensional pattern vectors. Region A from first layer activates neuron 1 from second layer, respectively region B activates neuron 2. The weights from A to 1, respectively from B to 2 are the same, i.e. the same input pattern in A respectively B will give same activation in 1 respectively 2. Layers 1, 2, 3, ... are in decreasing size order.

is independent to translation, scaling and rotation.



#### Remarks:

- ➔ There are two potential problems when using moments: one is that it may be computationally intensive and the second is that some information may be lost during preprocessing.

### 13.7.3 Shared Weights

This technique is using specially build ANNs to allow for a relative translation invariance.



#### Remarks:

- ➔ This technique is useful in image processing and bears some resemblance with mammalian vision.

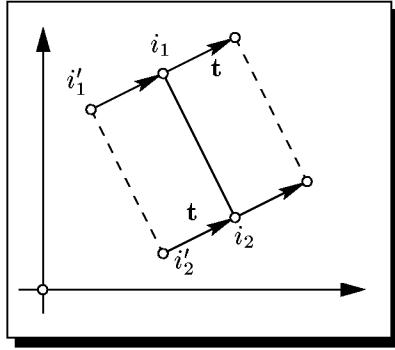
Considering a bidimensional input pattern the network is build such that a layer will receive excitatory input only from a small area of the previous layer. By having (sharing) the same weights, such areas send the same excitatory input to the next layer, when presented with the same input pattern. See figure 13.6.

### 13.7.4 Higher-order ANNs

A higher-order network have a neural activation of the form<sup>9</sup>:

$$z_j = f \left( w_{j0} + \sum_{i=1}^N w_{ji} x_i + \sum_{i_1=1}^N \sum_{i_2=1}^N w_{ji_1 i_2} x_{i_1} x_{i_2} + \dots \right)$$

<sup>9</sup>See chapter "Multi Layer Neural Networks"



**Figure 13.7:** Translation in bidimensional space.  $t$  represents the translation vector. The pattern values  $x_{i_1}$  is replaced by  $x_{i'_1}$ , respectively  $x_{i_2}$  by  $x_{i'_2}$ .

where neuron  $z_j$  receives input from  $x_i$ ,  $w_{j0}$  is the bias and  $f$  is the activation function.

By using second-order neural networks, i.e. of the form:

$$z_i = f \left( \sum_{i_1=1}^N \sum_{i_2=1}^N w_{j i_1 i_2} x_{i_1} x_{i_2} \right) \quad (13.15)$$

it is possible to built a network whose output is translation invariant for a bidimensional input pattern.

Considering a translation of a pattern in a bidimensional space then in the place  $i_1$  occupied previously by  $x_{i_1}$  now will be an input  $x_{i'_1}$  which have come from  $i'_1$ , the same happens for the second point  $i_2$ . See figure 13.7.

To keep the network output (13.15) the same it is necessary to impose the condition:

$$w_{j i_1 i_2} = w_{j i'_1 i'_2} \quad (13.16)$$

for each pair of points  $\{(i_1, i_2), (i'_1, i'_2)\}$  which may be correlated by a translation.



#### Remarks:

- The condition (13.16) greatly reduces the number of independent weights (such that a second order neural network becomes more manageable).
- One layer of second order neural network is sufficient to extract the translation invariant pattern information.
- Highest order networks may be used for more complex invariant information extraction, e.g. a third order network may be used to extract information invariant to translation, uniform scaling and rotation by correlating 3 points.



## CHAPTER 14

# Learning Optimization

### ► 14.1 The Bias-Variance Tradeoff

Let  $p(t|\mathbf{x})$  be the target  $t$  probability density, conditioned on input  $\mathbf{x}$ . The conditional average of the target is  $\langle t|\mathbf{x} \rangle = \int_Y tp(t|\mathbf{x}) dt$  and the conditional average of the square of targets  $\langle t^2|\mathbf{x} \rangle = \int_Y t^2 p(t|\mathbf{x}) dt$ .  $\diamond \langle t|\mathbf{x} \rangle, \langle t^2|\mathbf{x} \rangle$

For an infinite training set the sun-of-square error function may be written<sup>1</sup>, considering only one output, as:

$$E = \frac{1}{2} \int_X [y(\mathbf{x}) - \langle t|\mathbf{x} \rangle]^2 p(\mathbf{x}) d\mathbf{x} + \frac{1}{2} \int_X [\langle t^2|\mathbf{x} \rangle - \langle t|\mathbf{x} \rangle^2] p(\mathbf{x}) d\mathbf{x} \quad (14.1)$$

The second term in (14.1) is independent of  $y(\mathbf{x})$  and thus independent of weights — it represents an intrinsic noise which sets the minimum of  $E$ . The first term may be minimized to 0 in which case  $y(\mathbf{x}) = \langle t|\mathbf{x} \rangle$ .

Finite training data sets  $S$ , considered containing  $P$  training vector patterns (each), cannot cover the whole possibilities for input/output and then there will always be some difference between  $y(\mathbf{x})$  (after training) and  $\langle t|\mathbf{x} \rangle$  (considering an infinite set).

The integrand of the first term in (14.1), i.e.  $[y(\mathbf{x}) - \langle t|\mathbf{x} \rangle]^2$ , gives a measure of how close is the actual mapping  $y(\mathbf{x})$  to the desired target  $t$ . To avoid the dependency over a particular  $\diamond \mathcal{E}_S$

---

<sup>14.1</sup>See [Bis95] pp. 332–338.

<sup>1</sup>See chapter “Error Functions”, “Significance of network output”.

training set, the expectation (average)  $\mathcal{E}_S$  is taken as a measure of the mapping:

$$\text{measure of mapping} \equiv \mathcal{E}_S\{[y(\mathbf{x}) - \langle t|\mathbf{x} \rangle]^2\}$$

the average being done over the whole ensemble of training sets.

bias

**Definition 14.1.1.** *The **bias** represents the difference between the average of network mapping  $y(\mathbf{x})$  and the data generator, i.e.  $\langle t|\mathbf{x} \rangle$ :*

$$\text{bias} \equiv \mathcal{E}_S\{y(\mathbf{x})\} - \langle t|\mathbf{x} \rangle$$

*The **average bias** over the input  $\mathbf{x}$  is defined as:*

$$(\text{average bias})^2 \equiv \frac{1}{2} \int_X [\mathcal{E}_S\{y(\mathbf{x})\} - \langle t|\mathbf{x} \rangle]^2 p(\mathbf{x}) d\mathbf{x}$$

For a perfect model, the bias would be 0 (as  $\mathcal{E}_S\{y(\mathbf{x})\} = \langle t|\mathbf{x} \rangle$ ). Non-zero bias arises from two causes:

- difference between the function created by the model (e.g. ANN) and the true function who generated the data — this is the “true” bias,
- variance due to data sets — particular sensitivity to some training sets.

variance

**Definition 14.1.2.** *The **variance** represents the average sensitivity of the network mapping  $y(\mathbf{x})$  to a particular training set:*

$$\text{variance} \equiv \mathcal{E}_S\{[y(\mathbf{x}) - \mathcal{E}_S\{y(\mathbf{x})\}]^2\}$$

*The **average variance** over the input  $\mathbf{x}$  is defined as:*

$$\text{average variance} \equiv \frac{1}{2} \int_X \mathcal{E}_S\{[y(\mathbf{x}) - \mathcal{E}_S\{y(\mathbf{x})\}]^2\} p(\mathbf{x}) d\mathbf{x}$$

Let consider the measure of mapping  $[y(\mathbf{x}) - \langle t|\mathbf{x} \rangle]^2$ :

$$\begin{aligned} [y(\mathbf{x}) - \langle t|\mathbf{x} \rangle]^2 &= [y(\mathbf{x}) - \mathcal{E}_S\{y(\mathbf{x})\} + \mathcal{E}_S\{y(\mathbf{x})\} - \langle t|\mathbf{x} \rangle]^2 \\ &= [y(\mathbf{x}) - \mathcal{E}_S\{y(\mathbf{x})\}]^2 + [\mathcal{E}_S\{y(\mathbf{x})\} - \langle t|\mathbf{x} \rangle]^2 \\ &\quad + 2[y(\mathbf{x}) - \mathcal{E}_S\{y(\mathbf{x})\}][\mathcal{E}_S\{y(\mathbf{x})\} - \langle t|\mathbf{x} \rangle] \end{aligned}$$

When doing an average over the above equation, the third term cancels (because  $y(\mathbf{x}) \rightarrow \mathcal{E}_S\{y(\mathbf{x})\}$ ) and then:

$$\begin{aligned} \mathcal{E}_S\{[y(\mathbf{x}) - \langle t|\mathbf{x} \rangle]^2\} &= \mathcal{E}_S\{[y(\mathbf{x}) - \mathcal{E}_S\{y(\mathbf{x})\}]^2\} + [\mathcal{E}_S\{y(\mathbf{x})\} - \langle t|\mathbf{x} \rangle]^2 \\ &= \text{variance} + (\text{bias})^2 \end{aligned}$$

❖  $h(\mathbf{x}), \varepsilon$ 

To see the meaning of bias and variance let consider the targets as being generated from a function  $h(\mathbf{x})$  to which some random noise  $\varepsilon$ , with 0 mean, have been added:

$$t_p = h(\mathbf{x}_p) + \varepsilon_p \tag{14.2}$$

and the optimal mapping is then  $\langle t|x \rangle = h(x)$ .

There are two extreme possibilities for the  $y(x)$  mapping choice:

- The mapping is build to be some  $g(x)$  function, completely independent of data set. Then the variance is zero since  $\mathcal{E}_S\{y(x)\} = g(x) = y(x)$ .

However the bias may be very high (unless some other prior knowledge have been used to build  $g(x)$ ).

- The mapping is build to fit the data perfectly. Then the bias is zero since:

$$\mathcal{E}_S\{y(x)\} = \mathcal{E}\{h(x) + \varepsilon\} = h(x) = \langle t|x \rangle$$

However the variance is:

$$\mathcal{E}_S\{[y(x) - \mathcal{E}_S\{y(x)\}]^2\} = \mathcal{E}_S\{[y(x) - h(x)]^2\} = \mathcal{E}_S\{\varepsilon^2\}$$

and the variance of  $\varepsilon$  may be high.

The above discussion reveals that there is a tradeoff between the bias and the variance and, in practice, a balance between the two, should be found.

One way to reduce the bias *and* variance at the same time is to increase the complexity model by using larger training sets, i.e. the size of training set determines the size of the model such that the optimal balance between bias and variance is found (note however that this approach leads to the course of dimensionality and thus the model cannot be increased indefinitely).

Another way to reduce bias and variance at the same time is to use the prior knowledge (if any) about the data generator (14.2) when building the model.

## 14.2 Regularization

The error function  $E$  may be changed by adding a regularization term<sup>2</sup>  $\Omega$ :

$$\tilde{E} = E + \nu\Omega \tag{14.3}$$

where  $\Omega$  is a function which have a large value for smooth mapping functions  $y(x)$  and a small value otherwise, i.e. is large for simple models and small for complex models, thus encouraging less complex models. The  $\nu$  parameter controls the influence of  $\Omega$ . ❖  $\Omega, \nu$

Thus the regularization  $\Omega$  and the error  $E$  counterbalance one each other (as error generally increases in simple models) in the process of minimizing the changed error function  $\tilde{E}$  during learning process.

### 14.2.1 Weight Decay

In the case of a over-fitted model the mapping will have areas with large curvature which require large values for weights<sup>3</sup>.

<sup>14.2</sup>See [Bis95] pp.338–346.

<sup>2</sup>See chapter “Radial Basis Function Networks”

<sup>3</sup>See chapter “Parameter optimization”

By encouraging weights to be small the error hyper-surface is kept relatively smooth. This may be achieved by using a regularization of the form:

$$\Omega = \frac{1}{2} \sum_i w_i^2 \quad (14.4)$$

Many ANN training algorithm make use of the error gradient. From (14.3) and (14.4):

$$\nabla \tilde{E} = \nabla E + \nu W \quad (14.5)$$

( $W$  being seen as vector).

Considering just the part generated by the regularization term, the weight update formula in the gradient descent learning method<sup>4</sup> in the continuous time  $\tau$  limit is:

$$\frac{dW}{d\tau} = -\eta \nabla \tilde{E} = -\eta \nu W \quad (\nabla E \text{ neglected})$$

(where  $\eta$  is the learning parameter) which have a solution of the form:

$$W(\tau) = W(0) e^{-\eta \nu \tau}$$

i.e. the regularization term (14.4) favors weights decay toward 0, following an exponential rule.

❖  $\mathbf{b}, H$

Considering a quadratic error function<sup>5</sup> of the form:

$$E(W) = E_0 + \mathbf{b}^T W + \frac{1}{2} W^T H W \quad , \quad \mathbf{b}, H = \text{const.} \quad (14.6)$$

❖  $W^*$

where  $H$  is a symmetrical matrix (it is in fact the Hessian), the minima of  $E$  (from  $\nabla E = \hat{\mathbf{0}}$ ) is at  $W^*$  given by:

$$\nabla E = \mathbf{b} + HW^* = \hat{\mathbf{0}} \quad (14.7)$$

❖  $\widetilde{W}^*$

Similarly, the minima of  $\tilde{E}$  (from  $\nabla \tilde{E} = \hat{\mathbf{0}}$ ) occurs at  $\widetilde{W}^*$  given by:

$$\nabla \tilde{E} = \mathbf{b} + H\widetilde{W}^* + \nu W^* = \hat{\mathbf{0}} \quad (14.8)$$

(from (14.5) and (14.7)).

❖  $\mathbf{u}_i, \lambda_i$

Let  $\{\mathbf{u}_i\}$  be an orthogonal system of eigenvectors of  $H$  such that:

$$H\mathbf{u}_i = \lambda_i \mathbf{u}_i \quad , \quad \mathbf{u}_i^T \mathbf{u}_j = \delta_{ij} \quad (14.9)$$

where  $\lambda_i$  are the eigenvalues of  $H$  (such construction is possible due to the fact that  $H$  is symmetrical, see mathematical appendix).

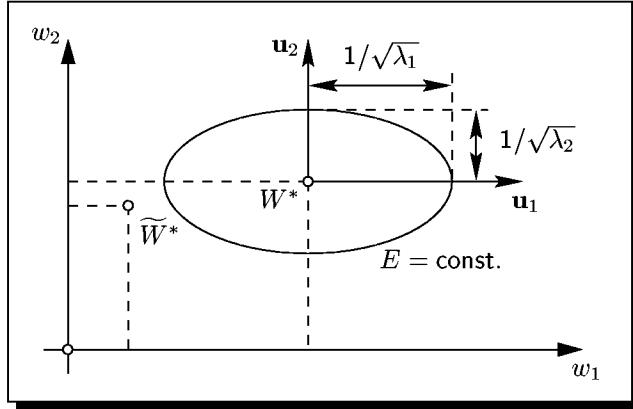
Let consider that the system of coordinates in the weights space is rotated such that it becomes parallel with  $\{\mathbf{u}_i\}$ . Then  $W^*$  and  $\widetilde{W}^*$  may be written as:

$$W^* = \sum_i w_i^* \mathbf{u}_i \quad , \quad \widetilde{W}^* = \sum_i \widetilde{w}_i^* \mathbf{u}_i$$

---

<sup>4</sup>See chapter “Single Layer Neural Networks”.

<sup>5</sup>See chapter “Parameter Optimization”



**Figure 14.1:** The hyper-surfaces  $E = \text{const.}$  are hyper-ellipses having axes proportional to  $1/\sqrt{\lambda_i}$ . The distance between  $W^*$  and  $\widetilde{W}^*$  is smaller along axes corresponding to larger eigenvalues  $\lambda_i$  of  $H$ .

and from (14.7) and (14.8)  $\nabla E = \hat{\mathbf{0}} = \nabla \tilde{E}$  and considering the orthogonality of  $\{\mathbf{u}_i\}$  it follows that:

$$\tilde{w}_i^* = \frac{\lambda_i}{\lambda_i + \nu} w_i^*$$

which means that  $\begin{cases} \lambda_i \gg \nu & \Rightarrow \quad \tilde{w}_i^* \simeq w_i^* \\ \lambda_i \ll \nu & \Rightarrow \quad |\tilde{w}_i^*| \ll |w_i^*| \end{cases}$ . As the surfaces  $E = \text{const.}$  are hyper-ellipses which have axes proportional with<sup>6</sup>  $1/\sqrt{\lambda_i}$  this means that  $\widetilde{W}^*$  is closer to  $W^*$  along  $\mathbf{u}_i$  directions with correspond to larger  $\lambda_i$ . See figure 14.1.

### 14.2.2 Linear Transformation And Weight Decay

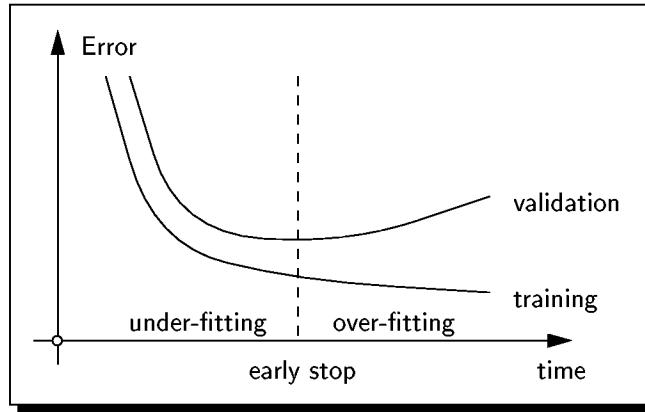
Let consider a multi-layer perceptron network having one hidden layer and one linear output layer. Then for the hidden layer  $z_j = f\left(w_{j0} + \sum_i w_{ji}x_i\right)$  and for the output layer  $y_k = w_{k0} + \sum_j w_{kj}z_j$ .

Considering a linear transformation of the form:  $x_i \rightarrow \tilde{x}_i = ax_i + b$  where  $a, b = \text{const.}$  then is is possible to get the same outputs by changing the weights of hidden layer as:

$$\begin{cases} w_{ji} & \rightarrow \quad \tilde{w}_{ji} = \frac{1}{a}w_{ji} \\ w_{j0} & \rightarrow \quad \tilde{w}_{j0} = w_{j0} - \frac{b}{a}\sum_i w_{ji} \end{cases}$$

(may be checked directly,  $z_j$  doesn't change).

<sup>6</sup>See chapter "Parameter Optimization".



**Figure 14.2:** The error given by the validation set increases from some point outwards. At that point the network is optimally fitted; beyond is over-fitted, before is under-fitted.

Similarly, for a transformation of the output layer:  $y_k \rightarrow \tilde{y}_k = cy_k + d$  where  $c, d = \text{const.}$  the weight changes of the output layer are:

$$\begin{cases} w_{kj} & \rightarrow \quad \tilde{w}_{kj} = cw_{kj} \\ w_{k0} & \rightarrow \quad \tilde{w}_{k0} = cw_{k0} + d \end{cases}$$

By training two identical networks, one with original data and one with transformed inputs and/or outputs the trained networks should be equivalent, with weights changed by the linear transformations as shown above. The regularization term (14.4) does not satisfy this requirement. However a weight decay regularization term of the form:

$$\Omega = \frac{\nu_1}{2} \sum_{\text{hidden layer}} w^2 + \frac{\nu_2}{2} \sum_{\text{output layer}} w^2$$

satisfies the invariance of linear transformation, provided that suitable rescaling is performed on  $\nu_1$  and  $\nu_2$ .

### 14.2.3 Early Stopping

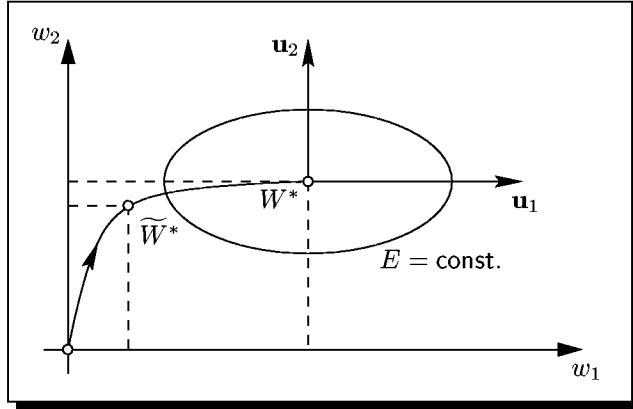
validation

From the whole set of available data for training, usually, some part is put aside and not used in training, for the purpose of *validation*, i.e. checking the generalization capability of network with some data unused during the learning process.

While the error function always decreases for the training set during learning, the error for the validation set decreases at the beginning then, later, begin to increase as the network becomes over-fitted. See figure 14.2.

The network is optimally fitted at the point where the validation set gives the lowest error; at this point training should be stopped as the generalization capabilities are best, even if further training will lower error with respect to the training set.

A justification of the early stop method may be given by the means of the weight decay procedure. As the weights adapt during learning, the weight vector  $W$  follows a path which



**Figure 14.3:** The “early stop” method gives similar result as the “weight decay” procedure as it involves stopping on the learning path — marked with an arrow — before reaching  $W^*$ . The particular form of the learning path may be explained by the faster advancement along directions with greater  $\nabla E$ , i.e. smaller  $\lambda_i$ . See also figure 14.1 on page 257.

leads to  $\tilde{W}^*$  before reaching  $W^*$ . See figure 14.3 and section 14.2.1.

#### 14.2.4 Curvature Smoothing

As over-complex neural models leads to network mappings with large curvatures (on error surface), a direct approach will be to build a regularization term which increases with curvature. As the second derivatives are a measure of the curvature then the regularization should be of the form:

$$\Omega = \frac{1}{2} \sum_{p=1}^P \sum_{i=1}^N \sum_{k=1}^K \left( \frac{\partial^2 y_k(\mathbf{x}_p)}{\partial x_{ip}^2} \right)^2$$

$N$  respectively  $K$  being the size of input respectively output vectors.

❖  $N, K$

#### 14.2.5 Choosing weight decay hyperparameter

Considering the weight decay regularization (14.4) then the prior probability density of weights is chosen usually as a Gaussian of the form  $p(\mathbf{w}) \propto \exp(-\nu\Omega)$  which have the variance  $\sigma = 1/\sqrt{\nu}$ .

Considering the logistic activation function<sup>7</sup>  $f(x) = \frac{1}{1+e^{-x}}$  this one is saturated around  $x = \pm 3$ , i.e.  $f(-3) \simeq 0.04$  and  $f(3) \simeq 0.95$  (very close to lower 0 and upper 1 asymptotes).

As the reason (among others) for weight decay regularization is to prevent saturation of neuronal outputs then the variance of total input is to be around 2. For a small number of

<sup>14.2.5</sup>See [Rip96] pg. 163.

<sup>7</sup>See also chapter “Pattern Recognition”.

inputs in the range  $[0, 1]$  a reasonable variance should be between 1 and 10, e.g. the middle value 5 may be considered, this corresponds to  $\nu \sim 0.04$

In practice is a good thing to have some neurons saturated (this corresponding to a sort of pruning<sup>8</sup>) then the base range for  $\nu$  is between 0.001 and 0.1. Experience indicate that a multiplication of  $\nu$  up to 5 times is not critical to the learning process.

## 14.3 Adding Noise

Another way to achieve better generalization is to add random noise to the training set before inputting it to the network.

❖  $\varepsilon, \tilde{p}(\varepsilon)$

Let consider a random vector  $\varepsilon$  which is being added to the input  $\mathbf{x}$  and have probability density  $\tilde{p}(\varepsilon)$ .

In the absence of noise, the error function is<sup>9</sup>:

$$E = \frac{1}{2} \sum_{k=1}^K \int_Y \int_X [y_k(\mathbf{x}) - t_k]^2 p(t_k | \mathbf{x}) p(\mathbf{x}) d\mathbf{x} dt_k$$

Considering an infinite number of training vectors, each containing an added noise term, then

$$\tilde{E} = \frac{1}{2} \sum_{k=1}^K \int_Y \int_X \int_{\varepsilon} [y_k(\mathbf{x} + \varepsilon) - t_k]^2 p(t_k | \mathbf{x}) p(\mathbf{x}) \tilde{p}(\varepsilon) d\mathbf{x} dt_k d\varepsilon \quad (14.10)$$

A reasonable assumption is to consider the noise sufficiently small as to allow for the expansion of  $y_i(\mathbf{x} + \varepsilon)$  in a Taylor series, neglecting the third order and above terms:

$$y_k(\mathbf{x} + \varepsilon) = y_k(\mathbf{x}) + \sum_{i=1}^N \varepsilon_i \frac{\partial y_k}{\partial x_i} \Big|_{\varepsilon=0} + \frac{1}{2} \sum_{i=1}^N \sum_{j=1}^N \varepsilon_i \varepsilon_j \frac{\partial^2 y_k}{\partial x_i \partial x_j} \Big|_{\varepsilon=0} + \mathcal{O}(\varepsilon^3) \quad (14.11)$$

It is also reasonable to assume that the noise have zero mean and uncorrelated components:

$$\int_{\varepsilon} \varepsilon_i \tilde{p}(\varepsilon) d\varepsilon = 0 \quad \text{and} \quad \iint_{\varepsilon} \varepsilon_i \varepsilon_j \tilde{p}(\varepsilon) d\varepsilon = \nu \delta_{ij} \quad (14.12)$$

❖  $\nu$

where  $\nu$  is the variance of noise.

By using (14.11) in (14.10) and integrating over  $\varepsilon$  with the aid of (14.12), the error function may be written as:

$$\tilde{E} = E + \nu \Omega$$

---

<sup>8</sup>See also section 14.5.

<sup>14.3</sup>See [Bis95] pp. 346–349.

<sup>9</sup>See chapter “Error Functions”.

where  $\Omega$  becomes a regularization parameter of the form:

$$\Omega = \frac{1}{2} \sum_{k=1}^K \sum_{i=1}^N \int_Y \int_X \left\{ \left( \frac{\partial y_k}{\partial x_i} \right)^2 + \frac{1}{2} [y_k(\mathbf{x}) - t_k] \frac{\partial^2 y_k}{\partial x_i^2} \right\} p(t_k | \mathbf{x}) p(\mathbf{x}) d\mathbf{x} dt_k \quad (14.13)$$

where the noise do not appear anymore (normalization of  $p(\varepsilon)$  was also used).

The network mapping which minimize the sum-of-squares error function  $E$  is<sup>10</sup>  $y_k = \langle t_k | \mathbf{x} \rangle$ .

Thus the network mapping which minimize the regularized error function  $\tilde{E}$ , see (14.3), should be of the form  $y_k = \langle t_k | \mathbf{x} \rangle + \mathcal{O}(\nu)$ . Then the second term in (14.13):

$$\frac{1}{4} \sum_{k=1}^K \sum_{i=1}^N \int_X [y_k(\mathbf{x}) - \langle t_k | \mathbf{x} \rangle] \frac{\partial y_k}{\partial x_i} p(\mathbf{x}) d\mathbf{x}$$

(the integration over  $Y$  have been performed) cancels to the lowest order of  $\nu$ , at the minima of error  $\tilde{E}$ . This property makes the computation of second order derivatives of  $y_k$  (a task computationally intensive) avoidable, thus leaving the regularization function of the form:

$$\Omega = \frac{1}{2} \sum_{i=1}^N \sum_{k=1}^K \int_X \left( \frac{\partial y_k}{\partial x_i} \right)^2 p(\mathbf{x}) d\mathbf{x} \quad (14.14)$$

or, for a discrete series of training vectors:

$$\Omega = \frac{1}{2P} \sum_{p=1}^P \sum_{i=1}^N \sum_{k=1}^K \left( \frac{\partial y_k}{\partial x_i} \Big|_{\mathbf{x}_p} \right)^2$$

## 14.4 Soft Weight Sharing

This method encourages groups of weights to have similar values<sup>11</sup> by using some regularization links.

The soft weight sharing relates to weight decay method. Considering a Gaussian distribution for weights, of the form:

$$p(w) = \frac{1}{\sqrt{2\pi}} \exp \left( -\frac{w^2}{2} \right)$$

then the likelihood<sup>12</sup> of the whole set of weights is:

$$\mathcal{L} = \prod_{i=1}^{N_W} p(w_i) = \frac{1}{(2\pi)^{N_W/2}} \exp \left( -\frac{1}{2} \sum_{i=1}^{N_W} w_i^2 \right) \quad (14.15)$$

$N_W$  being the total number of weights. The same way, the negative logarithm of a like-  $\diamond N_W$

<sup>10</sup> See chapter “Error functions”.

<sup>14.4</sup> See [Bis95] pp. 349–353.

<sup>11</sup> A hard sharing method is discussed in the chapter “Feature Extraction”.

<sup>12</sup> See chapter “Pattern Recognition”.

lihood gives the error function<sup>13</sup>, the negative logarithm of likelihood (14.15) gives the regularization function (14.4) (up to an additive constant which plays no role in a learning process).

It is possible to encourage weights to group together by using a mixture<sup>14</sup> of Gaussian distributions:

$$p(w) = \sum_{m=1}^M \alpha_m p_m(w)$$

- ❖  $M, \alpha_m, p_m(w)$  where  $M$  is the number of mixture components,  $\alpha_m$  are the mixing coefficients and  $p_m(w)$  are the mixture components of the (assumed) Gaussian form:

$$p_m(w) = \frac{1}{\sqrt{2\pi\sigma_m^2}} \exp\left[-\frac{(w - \mu_m)^2}{2\sigma_m^2}\right] \quad (14.16)$$

- ❖  $\mu, \sigma$   $\mu$  being the mean and  $\sigma$  being the standard deviation.

### Remarks:

- ➔ In the mixture model  $\alpha_m = P(m)$  is the *prior* probability of the mixture component  $m$ .

Then the regularization function is:

$$\Omega = -\ln \mathcal{L} = -\ln \prod_{i=1}^{N_w} p(w_i) = -\sum_{i=1}^{N_w} \ln \sum_{m=1}^M \alpha_m p_m(w_i) \quad (14.17)$$

The regularized error function:

$$\tilde{E} = E + \nu \Omega \quad (14.18)$$

have to be optimized with respect to weights  $w_i$  and parameters  $\alpha_m, \mu_m$  and  $\sigma_m$ .

### Remarks:

- ➔ An optimization with respect to  $\nu$  is not possible as it will lead to  $\nu = 0$  and  $\tilde{E} \rightarrow E$ , i.e. the network will end up by being over-fitted. See section 14.2.3.

- ❖  $\pi_m$

The *posterior* probability of mixture component  $m$  is:

$$\pi_m(w) \equiv \frac{\alpha_m p_m(w)}{\sum_{n=1}^M \alpha_n p_n(w)} \quad (14.19)$$

From (14.18), (14.17), (14.19) and (14.16), the error derivatives with respect to  $w_i$  are:

$$\frac{\partial \tilde{E}}{\partial w_i} = \frac{\partial E}{\partial w_i} + \nu \sum_{m=1}^M \pi_m(w_i) \frac{w_i - \mu_m}{\sigma_m^2}$$

---

<sup>13</sup>See chapter "Error Functions".

<sup>14</sup>See also the chapter "Pattern recognition" regarding the mixture models.

which shows that the weights  $w_i$  are pulled towards the distribution centers  $\mu_m$  with "forces" proportional to the posterior probabilities  $\pi_m$ .

Similarly the error derivatives with respect to  $\mu_m$  are:

$$\frac{\partial \tilde{E}}{\partial \mu_m} = \nu \sum_{i=1}^{N_W} \pi_m(w_i) \frac{\mu_m - w_i}{\sigma_m^2}$$

which shows that  $\mu_m$  are pulled towards the sum of all weights, weighted by  $\pi_m$ .

Finally, the error derivatives with respect to  $\sigma_m$  are:

$$\frac{\partial \tilde{E}}{\partial \sigma_m} = \nu \sum_{i=1}^{N_W} \pi_m(w_i) \left[ \frac{1}{\sigma_m} - \frac{(w_i - \mu_m)^2}{\sigma_m^3} \right]$$



#### Remarks:

- In practice the  $\sigma_m$  parameters are taken of the form  $\sigma_m = \exp(\eta_m)$  and optimization is performed with respect to  $\eta_m$ . This ensures that  $\sigma_m$  remains *strictly* positive.

As  $\alpha_m$  plays the role of a prior probability it have to follow the probability constrains

$\alpha_m \in [0, 1]$  and  $\sum_{m=1}^M \alpha_m = 1$ . This is best done by using the softmax function method:

$$\alpha_m = \frac{\exp(\gamma_m)}{\sum_{n=1}^M \exp(\gamma_n)} \quad \Rightarrow \quad \frac{\partial \alpha_m}{\partial \gamma_n} = \delta_{mn} \alpha_n - \alpha_m \alpha_n$$

Then the derivative of  $\tilde{E}$  with respect to  $\gamma_m$  is (by similar ways as for previous derivatives

and using the normalization  $\sum_{n=1}^M \alpha_n = 1$ ):

$$\frac{\partial \tilde{E}}{\partial \gamma_m} = \sum_{n=1}^M \frac{\partial \tilde{E}}{\partial \alpha_n} \frac{\partial \alpha_n}{\partial \gamma_m} = \sum_{n=1}^M \left[ \left( -\sum_{i=1}^{N_W} \frac{\pi_n(w_i)}{\alpha_n} \right) (\delta_{nm} \alpha_m - \alpha_n \alpha_m) \right] = \sum_{i=1}^{N_W} [\alpha_m - \pi_m(w_i)]$$



#### Remarks:

- In practice care should be taken when initializing weights and related parameters. One way of doing it is to initialize weights with values over a finite interval, then divide the interval into  $M$  subintervals and assigning one to each  $p_m(w_i)$ , i.e. equal  $\alpha_m$ ,  $\mu_m$  centered into the respective subinterval and  $\sigma_m$  equal to the width of the subinterval. This method of initialization ensures a partial soft sharing and allows better learning from the training set.

## → 14.5 Growing And Pruning Methods

The network architecture may play a significant role in the final generalization performance. To allow for the finding of best architecture, e.g. the number of hidden layers and number of neurons per hidden layer, two general ways (which avoids an extensive search) may be used:

growing method

- The *growing method*: The starting network is small and then neurons are added one at a time till some criteria for optimization have been satisfied.

pruning method

- The *pruning method*: The starting network is big and then neurons are removed one at a time till some criteria for optimization have been satisfied.

### 14.5.1 Cascade Correlation

Cascade correlation is of growing method type.

The network starts with a single fully connected layer, where all inputs are linked to outputs.

At each stage — after the network have been trained for an empirical amount of time — a new hidden neuron is added, such that it receive input from all inputs *and all previously added (hidden) neurons* and send its output to all output neurons. See figure 14.4 on the next page.

The weights from inputs and all previously added neurons to the actually being added hidden neuron are calculated in one step and *after insertion they remain fixed*, e.g.— in figure 14.4 — the weights on link ① are calculated prior to  $z_1$  insertion, then the weights on links ④ and ③ are calculated prior to insertion of  $z_2$  and so on.

The weights from inputs to outputs and from all hidden neurons to outputs remain variable, to be further adapted, e.g.— in figure 14.4 — only weights on main (input → output) link and those on links ② and ⑤ will continue to adapt during further learning.



#### Remarks:

- By the above architecture the network is similar to one having just one “active” layer (the output) which receive input from input neurons and all hidden (added) neurons.

The “fixed” weights of the new (in the process of being inserted) neuron  $z$  is computed as follows:

❖  $\varepsilon_k, \langle \varepsilon_k \rangle$

- The error  $\varepsilon_k$  of output neuron  $y_k$  and the mean error  $\langle \varepsilon_k \rangle$  over all training set are calculated first:

$$\varepsilon_k = y_k - t_k \quad , \quad \langle \varepsilon_k \rangle = \frac{1}{P} \sum_{p=1}^P \varepsilon_{kp}$$

❖  $\varepsilon_{kp}$

( $\varepsilon_{kp}$  being the error  $\varepsilon_k$  for input vector  $x_p$  from the training set).

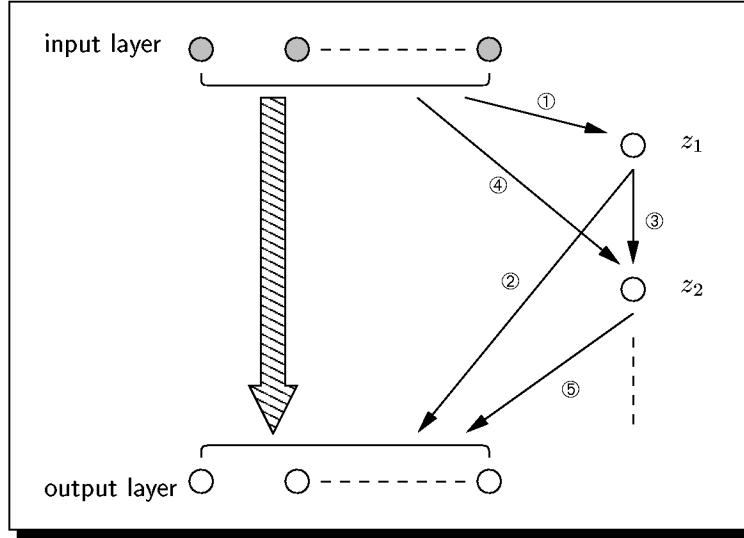
❖  $w_i$

- The weights  $w_i$  of all inputs to the new neuron  $z$  are initialized — weights for links from all inputs and all previous inserted hidden neurons.

❖  $z_p, \langle z \rangle$

The output of the new neuron  $z$  and the mean output over all training set  $\langle z \rangle$  will be:

<sup>14.5</sup>See [Bis95] pp. 353–364.



**Figure 14.4:** The cascade correlation method. The network architecture starts with input and output layers fully interconnected (the thick arrow shows that all output neurons receive all components of the input vector). The first hidden neuron  $z_1$  is added and the connections ① ( $z_1$  receives all inputs) and ② ( $z_1$  sends its output to all output neurons) are established. later, when  $z_2$  is added connections ③ ( $z_2$  receive input from  $z_1$ ), ④ and ⑤ are also added. And so on.

$$z_p = f \left( \sum_{\text{all inputs}} w_i x_{ip} \right) , \quad \langle z \rangle = \frac{1}{P} \sum_{p=1}^P z_p$$

where the sum in  $z_p$  is done also over the previous inserted hidden neurons ( $f$  being the activation function and  $z_p$  being the new neuron output for input vector  $\mathbf{x}_p$ ).

- The weights are optimized by maximizing the correlation, i.e. covariance,  $S$  between the output of the new neuron to be inserted and the residual actual (before the insertion takes place) error of the network output — similar to the Fisher discriminant<sup>15</sup>:

$$S = \sum_{k=1}^K \left| \sum_{p=1}^P (z_p - \langle z \rangle)(\varepsilon_k - \langle \varepsilon_k \rangle) \right|$$

- The maximisation of  $S$  is performed by using the derivative with respect to  $w_i$ :

$$\frac{\partial S}{\partial w_i} = \pm \sum_{k=1}^K \sum_{p=1}^P f' \cdot x_{ip} \cdot (\varepsilon_k - \langle \varepsilon_k \rangle)$$

where the sign is given by the expression inside the module from  $S$ .

<sup>15</sup>See chapter “Single Layer Neural Networks”.

❖  $f, z_p$

❖  $S$

The maximisation may be done using the above derivative in a way similar to the methods used in backpropagation.

### 14.5.2 Pruning Techniques

#### *Saliency of weights*

The pruning technique involves starting with a (relatively) large network, training it, then removing some neuronal connections, then training again and repeat the training/pruning process.

saliency

To decide about what network links are less important and thus susceptible to removal it is necessary to assess the relative importance of weights, this process being named *saliency*.

#### *Optimal Brain Damage*

❖  $\delta E, \delta w_i, N_W$

Let consider a small variation in error function  $\delta E$  due to a small variation of weights  $\delta w_i$ . By developing in series and taking only the first 2 terms ( $N_W$  is the total number of weights):

$$\delta E = \sum_{i=1}^{N_W} \frac{\partial E}{\partial w_i} \delta w_i + \frac{1}{2} \sum_{i=1}^{N_W} \sum_{j=1}^{N_W} H_{ij} \delta w_i \delta w_j + \mathcal{O}(\delta w^3)$$

❖  $H_{ij}$  where  $H$  is the Hessian whose elements are  $H_{ij} = \frac{\partial^2 E}{\partial w_i \partial w_j}$ .

At minimum of  $E$  its first derivatives become zero. Assuming that the Hessian may be approximated by making all non-diagonal elements equals to zero — *this representing the main assumption of this technique* — then:

$$\delta E = \frac{1}{2} \sum_{i=1}^W H_{ii} (\delta w_i)^2$$

To remove a neuronal connection is equivalent to make its corresponding weight equal zero:

$$\delta w_i = 0 - w_i = -w_i \quad (14.20)$$

and then a measure of saliency for weight  $w_i$  would be:

$$\text{saliency} = \frac{H_{ii} w_i^2}{2} \quad (14.21)$$

The *optimal brain damage* technique involves removal of connections defined by the weights of lowest saliency.

#### **Remarks:**

- ➔ In practice the number of weights being deleted, the amount of training between weight removal and the overall stop conditions are empirically established.

### **Optimal Brain Surgeon**

The optimal brain surgeon technique works in the same manner as the optimal brain damage but does not assume that the Hessian is diagonal as this may lead to poor results in some cases.

The variation around minima of  $E$  is then ( $\delta W$  is the vector of  $\delta w_i$ ):

❖  $\delta W$

$$\delta E = \frac{1}{2} \delta W^T H \delta W \quad (14.22)$$

Assuming that weight  $w_i$  is pruned then, equivalent to (14.20):

$$\delta w_i = 0 - w_i = -w_i \Leftrightarrow \mathbf{e}_i^T \delta W + w_i = 0 \quad (14.23)$$

where  $\mathbf{e}_i$  is a unit vector parallel to  $w_i$  axis, i.e.  $\mathbf{e}_i^T \delta W$  is the projection of  $\delta w$  on  $w_i$  axis;  $\mathbf{e}_i$  is of the form  $\mathbf{e}_i^T = (0 \ \cdots \ 0 \ 1 \ 0 \ \cdots \ 0)$ , with 1 in the  $i$ -th position and  $N_W$  dimensional.

❖  $\mathbf{e}_i$

To find the new (pruned) weights,  $\delta E$  have to be minimized subject to condition (14.23).

The Lagrange multipliers are used<sup>16</sup>; the Lagrange function is:

$$L = E + \lambda(\mathbf{e}_i^T \delta W + w_i) = \frac{1}{2} \delta W^T H \delta W + \lambda(\mathbf{e}_i^T \delta W + w_i)$$

and then by zeroing the derivative with respect to  $\delta W$ :

$$\nabla L = H \delta W + \lambda \mathbf{e}_i = \hat{\mathbf{0}} \Rightarrow \delta W = -\lambda H^{-1} \mathbf{e}_i$$

and, by replacing in (14.23) and considering the form of  $\mathbf{e}_i^T$  and thus  $\mathbf{e}_i^T H^{-1} \mathbf{e}_i = \{H^{-1}\}_{ii}$  then:

$$-\lambda \mathbf{e}_i^T H \mathbf{e}_i + w_i = 0 \Rightarrow \lambda = \frac{w_i}{\{H^{-1}\}_{ii}}$$

such that finally:

$$\delta W = -\frac{w_i}{\{H^{-1}\}_{ii}} H^{-1} \mathbf{e}_i \quad (14.24)$$

Replacing (14.24) into (14.22) the minimal error variation due to the removal of neural link corresponding to  $w_i$  is ( $H$  is symmetric and then  $H^{-1}$  is as well, also use matrix property  $(AB)^T = B^T A^T$ ):

$$\begin{aligned} \delta E &= \frac{1}{2} \frac{w_i^2}{\{H^{-1}\}_{ii}^2} \mathbf{e}_i^T (H^{-1})^T H H^{-1} \mathbf{e}_i = \frac{1}{2} \frac{w_i^2}{\{H^{-1}\}_{ii}^2} \mathbf{e}_i^T (H^{-1})^T \mathbf{e}_i \\ &= \frac{1}{2} \frac{w_i^2}{\{H^{-1}\}_{ii}} \end{aligned} \quad (14.25)$$

Optimal brain surgery works in similar way to optimal brain damage: after some training the inverse Hessian is calculated and some weights involving the smaller error variation, as given by (14.22), are zeroed, i.e. the corresponding inter-neuronal links removed. Then the procedure is repeated.

---

<sup>16</sup>See mathematical appendix.

### 14.5.3 Neuron Pruning

Instead of pruning inter-neuronal links, it is possible to prune whole neurons.

❖  $s_j$

To be able to choose which neurons may be removed it is necessary to have a measure of neuron saliency. Such a measure may be the difference in network output error made by neuron removal:

$$s_j = E_{\text{with neuron } j} - E_{\text{without neuron } j}$$

❖  $\alpha_j$

As the above measure is computationally intensive, an alternative approach is to modify the neuronal input by adding an overall multiplying factor  $\alpha_j$ . The neuronal output is then written as:

$$z_j = f \left( \alpha_j \sum_i w_{ji} z_i \right)$$

where the activation function  $f$  is defined such that  $f(0) = 0$ , e.g.  $f \equiv \tanh$ . Then for  $\alpha_j = 1$  the neuron  $j$  is present, for  $\alpha_j = 0$  the neuron is removed.

The saliency of neuron  $j$  becomes:

$$s_j = E|_{\alpha_j=1} - E|_{\alpha_j=0}$$

which shows that the derivative of  $E$  with respect to  $\alpha_j$  is also a good measure of neuronal saliency:

$$\tilde{s}_j = - \left. \frac{\partial E}{\partial \alpha_j} \right|_{\alpha_j=1}$$

which may be evaluated in a backpropagation way. Note the “-” sign; usually the error increases after the removal of a neuron, while  $\alpha_j$  decreases and  $s_j$  is taken as a positive quantity (and the bigger it is, the more important the corresponding neuron is).

## → 14.6 Committees of Networks

As, in practice it is common to train different networks (with different architectures) in order to choose the best it would be much better to combine several networks to form a committee (it's even not required to be a network, it may be any kind of model).

❖  $M$

Let assume that each network have only one output and there are  $M$  such networks. The mapping achieved by each network  $y_m(\mathbf{x})$  may be seen as being the desired mapping  $h(\mathbf{x})$  plus some added error  $\varepsilon_m(\mathbf{x})$ :

$$y_m(\mathbf{x}) = h(\mathbf{x}) + \varepsilon_m(\mathbf{x})$$

The averaged sum-of-squares error for network  $m$  is:

$$\langle E_m \rangle = \mathcal{E}\{[y_m(\mathbf{x}) - h(\mathbf{x})]^2\} = \mathcal{E}\{\varepsilon_m^2\} = \int_X \varepsilon_m^2(\mathbf{x}) p(\mathbf{x}) d\mathbf{x}$$

---

<sup>14.6</sup>See [Bis95] pp. 364–369.

The average error over the whole set of networks *when acting individually* (not as committee) is:

$$E_{\text{av.}} = \frac{1}{M} \sum_{m=1}^M \langle E_m \rangle = \frac{1}{M} \sum_{m=1}^M \mathcal{E}\{\varepsilon_m^2\}$$

❖  $E_{\text{av.}}$

### Simple committee

The simplest way of building a committee of networks is to consider the output of the whole system  $y_{\text{com.}}$  as being the average of individual network outputs:

❖  $y_{\text{com.}}$

$$y_{\text{com.}} = \frac{1}{M} \sum_{m=1}^M y_m(\mathbf{x}) \quad (14.26)$$

The averaged sum-of-squares error for the committee is:

❖  $\langle E_{\text{com.}} \rangle$

$$\langle E_{\text{com.}} \rangle = \mathcal{E} \left\{ \left[ \frac{1}{M} \sum_{m=1}^M y_m(\mathbf{x}) \right]^2 \right\} = \mathcal{E} \left\{ \left[ \frac{1}{M} \sum_{m=1}^M \varepsilon_m(\mathbf{x}) \right]^2 \right\}$$

By using the Cauchy inequality in the form:  $\left( \sum_{m=1}^M \varepsilon_m(\mathbf{x}) \right)^2 \leq L \sum_{m=1}^M \varepsilon_m^2(\mathbf{x})$  it follows that  $\langle E_{\text{com.}} \rangle \leq E_{\text{av.}}$ , i.e. the error of committee is less than the average error over independent networks.



#### Remarks:

► Considering that the error  $\varepsilon_m(\mathbf{x})$  have zero mean and are uncorrelated:

$$\mathcal{E}\{\varepsilon_m\} = 0 \quad \text{and} \quad \mathcal{E}\{\varepsilon_m \varepsilon_n\} = 0 \quad \text{for } m \neq n$$

then the error  $\langle E_{\text{com.}} \rangle$  becomes:

$$\langle E_{\text{com.}} \rangle = \frac{1}{M^2} \sum_{m=1}^M \mathcal{E}\{\varepsilon_m^2\} = \frac{1}{M} E_{\text{av.}}$$

which shows a big improvement. However in practice the error  $\varepsilon_m(\mathbf{x})$  is strongly correlated, the same input vector  $\mathbf{x}$  giving a similar error on different networks.

### Weighted committee

Another way of building a committee is to make an *weighted* mean over individual network outputs:

❖  $\alpha_m$

$$y_{\text{com.}} = \sum_{m=1}^M \alpha_m y_m(\mathbf{x}) \quad (14.27)$$

As  $y_{\text{com.}}$  have the same meaning as  $y_m(\mathbf{x})$  then clearly the  $\alpha_m$  coefficients should have a meaning of probability, i.e.  $\alpha_m$  is the probability of  $y_m$  being the desired output from all

$\{y_m\}$  set. The the following conditions should be imposed:

$$\alpha_m \in [0, 1], \forall m \text{ and } \sum_{m=1}^M \alpha_m = 1 \quad (14.28)$$

and they will be determined as to minimize the network committee error.

 **Remarks:**

- ➔ By comparing (14.27) with (14.26) it's clear that in the simple model all networks have equal weight  $1/M$ .

❖  $C$

The error correlation matrix  $C$  is defined as the square and symmetrical matrix whose elements are:  $C_{mn} = \mathcal{E}\{\varepsilon_m(\mathbf{x})\varepsilon_n(\mathbf{x})\}$  (as it's defined as expectation, it does not depend on  $\mathbf{x}$ ).

The averaged sum-of-squares error of the committee becomes:

$$\begin{aligned} \langle E_{\text{com.}} \rangle &= \mathcal{E}\{[y_{\text{com.}}(\mathbf{x}) - h(\mathbf{x})]^2\} = \mathcal{E}\left\{\left(\sum_{m=1}^M \alpha_m \varepsilon_m\right)^2\right\} \\ &= \mathcal{E}\left\{\left(\sum_{m=1}^M \alpha_m \varepsilon_m\right)\left(\sum_{n=1}^M \alpha_n \varepsilon_n\right)\right\} = \sum_{m=1}^M \sum_{n=1}^M \alpha_m \alpha_n C_{mn} \end{aligned} \quad (14.29)$$

To find the minima of  $\langle E_{\text{com.}} \rangle$  subject to constraint (14.28) the Lagrange multipliers method<sup>17</sup> is used. The Lagrange function is:

$$L = E + \lambda \left( \sum_{m=1}^M \alpha_m - 1 \right) = \sum_{m=1}^M \sum_{n=1}^M \alpha_m \alpha_n C_{mn} + \lambda \left( \sum_{m=1}^M \alpha_m - 1 \right)$$

and by zeroing its derivatives with respect to  $\alpha_m$ :

$$\frac{\partial L}{\partial \alpha_m} = 0 \Rightarrow 2 \sum_{n=1}^M \alpha_n C_{mn} + \lambda = 0, \text{ for } m = \overline{1, M}$$

❖  $\boldsymbol{\alpha}, \boldsymbol{\lambda}$

(as  $C$  is symmetrical then  $C_{ij} = C_{ji}$ ). Considering the vectors  $\boldsymbol{\alpha}^T = (\alpha_1 \dots \alpha_M)$  and  $\boldsymbol{\lambda} = \lambda \hat{\mathbf{1}}$  then the above set of equations may be written in matrix form as:

$$2C\boldsymbol{\alpha} + \boldsymbol{\lambda} = \hat{\mathbf{0}}$$

which may be solved easily as:

$$\boldsymbol{\alpha} = -\frac{1}{2} C^{-1} \boldsymbol{\lambda} \Leftrightarrow \alpha_m = -\frac{\lambda}{2} \sum_{n=1}^M \{C^{-1}\}_{mn} \quad (14.30)$$

By replacing the value of  $\alpha_m$  from (14.30) into (14.28),  $\lambda$  is found to be:

$$\lambda = -\frac{2}{\sum_{m=1}^M \sum_{n=1}^M \{C^{-1}\}_{mn}}$$

---

<sup>17</sup>See mathematical appendix.

and then the replacement of  $\lambda$  back into (14.30) gives the final values for  $\alpha_m$ :

$$\boldsymbol{\alpha} = \frac{C^{-1}\hat{\mathbf{1}}}{\sum_{m=1}^M \sum_{n=1}^M \{C^{(-1)}\}_{mn}} \Rightarrow \alpha_m = \frac{\sum_{n=1}^L \{C^{-1}\}_{mn}}{\sum_{n=1}^M \sum_{o=1}^M \{C^{-1}\}_{no}} \quad (14.31)$$

The error (14.29) may be written in matrix form as  $\langle E_{\text{com.}} \rangle = \boldsymbol{\alpha}^T C \boldsymbol{\alpha}$  and then, by replacing the value of  $\boldsymbol{\alpha}$ , and using the relations:

$$(C^{-1}\hat{\mathbf{1}})^T = \hat{\mathbf{1}}^T (C^{-1})^T = \hat{\mathbf{1}}^T C^{-1} \quad \text{and} \quad \hat{\mathbf{1}}^T C^{-1} \hat{\mathbf{1}} = \sum_{m=1}^M \sum_{n=1}^M \{C^{-1}\}_{mn}$$

( $C$  is symmetrical) then the minimum error is  $\langle E_{\text{com.}} \rangle = \frac{1}{\sum_{m=1}^M \sum_{n=1}^M \{C^{-1}\}_{mn}}$

As the weighted committee is similar to the simple committee, the same criteria apply to prove that  $\langle E_{\text{com.}} \rangle \leq E_{\text{av.}}$



#### Remarks:

- The coefficients found in (14.31) are not guaranteed to be positive so this have to be enforced by other means. However if all  $\alpha_m$  are positive then from  $\forall \alpha_m \geq 0$  and  $\sum_{m=1}^M \alpha_m = 1 \Rightarrow \forall \alpha_m \in [0, 1]$  (worst case when all coefficients are zero except one which is 1).

## 14.7 Mixture Of Experts

The mixture of experts model divides the input space in sub-areas, using a separate, specially trained, network for each sub-space — *the expert* — and a supplementary network to act as a gateway, deciding what expert will be allowed to generate the final output. See figure 14.5 on the following page.

The error function is build from the negative logarithm of the likelihood function, considering a mixture model<sup>18</sup> of  $M$  Gaussian distributions. The number of training vectors is  $P$ .

$$E = - \sum_{p=1}^P \ln \left[ \sum_{m=1}^M \alpha_m(\mathbf{x}_p) p_m(\mathbf{t}_p | \mathbf{x}_p) \right] \quad (14.32)$$

where the  $p_m(\mathbf{t}|\mathbf{x})$  components are Gaussians of the form:

❖  $p_m(\mathbf{t}|\mathbf{x})$

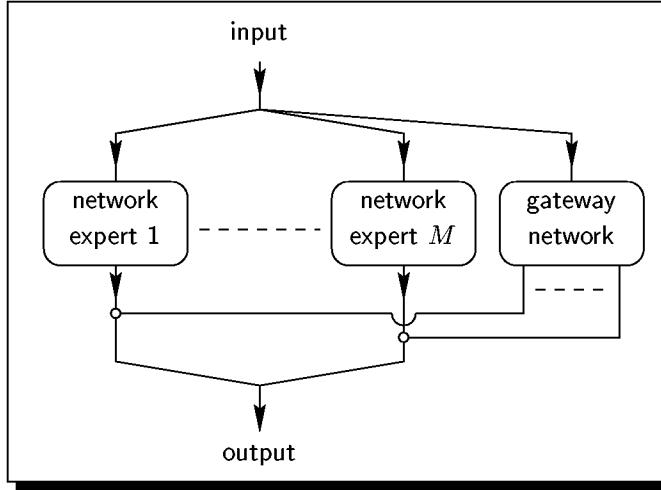
$$p_m(\mathbf{t}|\mathbf{x}) = \frac{1}{(2\pi)^{N/2}} \exp \left[ -\frac{\|\mathbf{t} - \mu_m(\mathbf{x})\|^2}{2} \right]$$

( $N$  being the dimensionality of  $\mathbf{x}$ ). The  $\mu_m(\mathbf{x})$  means are functions of input and the

❖  $\mu_m(\mathbf{x})$

<sup>14.7</sup> See [Bis95] pp. 369–371.

<sup>18</sup> See chapter “Pattern Recognition” and also chapter “Error Functions” regarding the modeling of conditional distributions.



**Figure 14.5:** The mixture of experts model. Each “expert” network is specialized in one input sub-space. The gateway network decide what expert will be allowed to give the final output (by blocking all others). There are  $M$  “experts” and the gateway have one output for each “expert”.

covariance is set to unity.

Each “expert” will represent an Gaussian and will output the  $\mu_m(\mathbf{x})$ . The  $\alpha_m$  coefficients are generated trough a softmax function from the outputs  $\gamma_m$  of the gateway:

$$\alpha_m = \frac{\exp(\gamma_m)}{\sum_{n=1}^M \exp(\gamma_n)}$$

The mixture of experts is trained simultaneously, including the gateway, by adjusting the weights such that the error (14.32) is minimized.

#### ☞ **Remarks:**

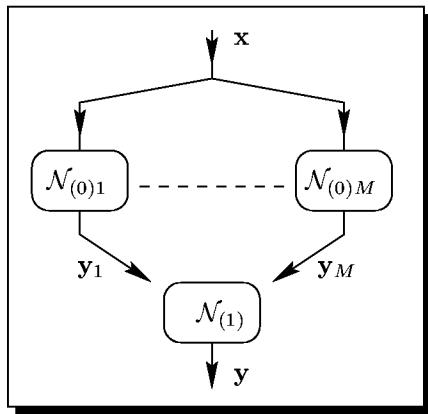
- ➡ The model presented here may be extended to a level where each expert becomes an embedded mixture of experts.

## ► 14.8 Other Training Techniques

### 14.8.1 Cross-validation

Often, in practice, several models are build and then the efficiency of each is estimated, e.g. generalization capability using a validation set, in order to select the best one.

<sup>14.8</sup>See [Bis95] pp. 371–377 and [Rip96] pg. 41.



**Figure 14.6:** The stacked generalization method. The set of networks  $\mathcal{N}_{(0)1}, \dots, \mathcal{N}_{(0)M}$  are trained using  $P - 1$  vectors from the training set then the network  $\mathcal{N}_{(1)}$  is used to assess the generalization capability of the level 0 networks.

However sometimes the available data is too scarce to afford to put aside a set for validation. In this case the cross-validation technique may be used.

The training set is divided into  $S$  subsets. The model to be considered is trained using  $S - 1$  subsets and the one left as a validation set. There are  $S$  such combinations. Then the efficiency of the model is calculated by making an average over all  $S$  training/validation results.

### 14.8.2 Stacked Generalization

This method is also applicable when the quantity of available data is small and it is desirable to keep all the “good parts” of various models.

Considering that the number of available training patterns is  $P$  then a set of  $M$  level 0 networks  $\mathcal{N}_{(0)1}, \dots, \mathcal{N}_{(0)M}$  are trained *using only*  $P - 1$  training patterns. See figure 14.6.

The left-out pattern vector is run through the set of networks  $\mathcal{N}_{(0)1}, \dots, \mathcal{N}_{(0)M}$  this will give rise to a new pattern (for the next network layer) of the form  $\{y_1, \dots, y_M\}$ .

The whole procedure is repeated in turn for each of the  $P$  patterns, this giving rise to a new set of  $P$  vectors. This new set is used to train a second level network  $\mathcal{N}_{(1)}$  using as target the desired output  $y$ . Obviously the  $\mathcal{N}_{(1)}$  assess the generalization capability of the  $\mathcal{N}_{(0)1}, \dots, \mathcal{N}_{(0)M}$  networks.

Finally the  $\mathcal{N}_{(0)1}, \dots, \mathcal{N}_{(0)M}$  are trained using all  $P$  training  $x$  patterns.

### 14.8.3 Complexity Criteria

Complexity criteria are measures of the generalization and complexity of models.

The prediction error is defined as being the sum between the training error and a measure

of the complexity of model:

$$\text{prediction error} = \text{training error} + \text{complexity measure}$$

where the complexity measure may be the number of weights.

For small networks the training error will be large and complexity measure low. For large networks the training error will be low and the complexity measure high. Thus by finding the minimum of prediction error helps finding the optimal tradeoff between model complexity and generalization capability.

The prediction error, for a sum-of-squares error function is defined as:

$$\text{prediction error} = \frac{2E}{P} + \frac{2N_W}{P} \sigma^2$$

where  $E$  is the error given by the training set, after learning was completed,  $P$  is the number of training patterns,  $N_W$  is the total number of weights and  $\sigma$  is the variance of noise in data — to be estimated.

Another way of defining prediction error — which is applicable for non-linear models and regularized error functions — is:

$$\text{prediction error} = \frac{2E}{P} + \frac{2\gamma}{P} \sigma^2$$

where  $\gamma$  is named the *effective number of parameters* and is defined as:

$$\gamma = \sum_{i=1}^{N_W} \frac{\lambda_i}{\lambda_i + \nu}$$

$\lambda_i$  being the eigenvalues of Hessian and  $\nu$  being the multiplication parameter of the regularization term.

#### 14.8.4 Model For Mixed Discrete And Continuous Data

It may happen that the input pattern vector  $\mathbf{x}$  have a discrete component along a continuous one, i.e. is of the form  $\mathbf{x}^T = (a \ x_1 \ \dots \ x_N)$  where  $a$  takes discrete values and  $\{x_i\}$  continuous ones. In this case one way of modeling the distribution  $p(\mathbf{x})$  is to try to find a conditional Gaussian distribution of the form:

$$p(\mathbf{x}) = \frac{p_\alpha(a)}{(2\pi)^{N/2} |\Sigma|^{1/2}} \exp \left[ -\frac{1}{2} (\hat{\mathbf{x}} - \boldsymbol{\mu}_{\alpha a})^T \Sigma^{-1} (\hat{\mathbf{x}} - \boldsymbol{\mu}_{\alpha a}) \right]$$

where  $p_\alpha(a)$  is the probability of  $a$  taking the value  $\alpha$ ,  $\hat{\mathbf{x}}^T = (x_1 \ \dots \ x_N)$  is the continuous part of the input vector and  $\boldsymbol{\mu}_{\alpha a}$  and  $\Sigma$  are the means and respectively the covariance matrix (which may be  $\alpha$  and  $a$  dependent).

## CHAPTER 15

# Bayesian Techniques

### ► 15.1 Bayesian Learning

#### 15.1.1 Weight Distribution

Let consider a given network, i.e. the number of layers, number of neurons, activation function of neurons are all known and fixed.

Let  $p(W)$  be the prior probability density of weights,  $N_W$  the total number of weights,  $\mathbf{W}^T = (w_1 \dots w_{N_W})$  the weight vector,  $P$  the number of training patterns and  $T = \{\mathbf{t}_1, \dots, \mathbf{t}_P\}$  the training set of targets. ❖  $p(W), N_W, T$

The posterior probability density of weights  $p(W|T)$  is (using Bayes theorem):

$$p(W|T) = \frac{p(T|W)p(W)}{p(T)} \quad (15.1)$$

where  $p(T)$  represents a normalization factor which ensures that  $p(W|T)$  integrated over all weight space equals unity, thus  $p(T) = \int_W p(T|W)p(W) dW$ .

#### Remarks:

- ➔ As the training set consists of inputs as well, the the probability densities in (15.1) should be conditioned also on input  $p(W|T, X) = \frac{p(T|W, X)p(W|X)}{p(T|X)}$ , however the networks do not model the probability density  $p(x)$  of inputs and then  $X$  appears always as a conditioning factor and it will be omitted for brevity.

---

<sup>15.1</sup>See [Bis95] pp. 385–398.

The Bayesian learning involves the following steps:

- Some prior probability density  $p(W)$  is established for weights. This will have a rather large breath as little is known at this stage.
- The posterior probability of the targets  $p(T|W)$  is established.
- Using the Bayes theorem (15.1) the posterior conditioned probability density  $p(W|T)$  is found.

### 15.1.2 Gaussian Prior Weight Distribution

As explained in the previous section the prior probability density  $p(W)$  should be defined in a form which will define some characteristics of the model but on the other side leave enough freedom for weights.

Let consider an exponential form:

$$p(W) = \frac{1}{Z_W(\alpha)} \exp(-\alpha E_W) \quad (15.2)$$

❖  $E_W$ ,  $Z_W$

where  $E_W$  is a function of weights and  $Z_W$  is the normalization factor:

$$Z_W(\alpha) = \int_W \exp(-\alpha E_W) d\mathbf{W} \quad (15.3)$$

such that  $\int_W p(W) d\mathbf{W} = 1$ .

One possible choice for  $E_W$  is:

$$E_W = \frac{1}{2} \|\mathbf{W}\|^2 = \frac{1}{2} \sum_{i=1}^{N_W} w_i^2 \quad (15.4)$$

which encourages small weights since for large  $\|\mathbf{W}\|$ ,  $E_W$  is large and consequently  $p(W)$  is small and thus  $W$  have unlikely value.

From (15.3) and (15.4) (see also the mathematical appendix, Gaussian integrals):

$$Z_W = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \exp \left( -\alpha \sum_{i=1}^{N_W} w_i^2 \right) dw_1 \dots dw_{N_W} = \left( \frac{2\pi}{\alpha} \right)^{\frac{N_W}{2}} \quad (15.5)$$

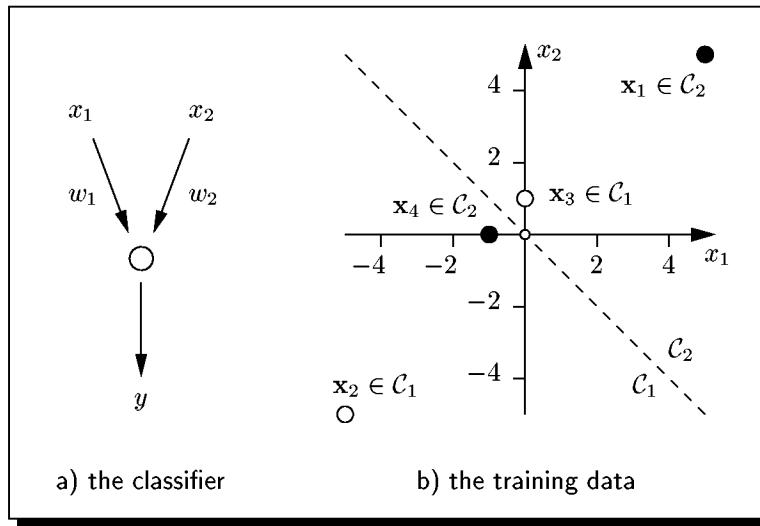
$$p(W) = \left( \frac{\alpha}{2\pi} \right)^{\frac{N_W}{2}} \exp \left( -\frac{\alpha}{2} \|\mathbf{W}\|^2 \right) \quad (15.6)$$

### 15.1.3 Application — Simple Classifier

Let consider a neuron with two inputs  $x_1$  and  $x_2$  and one output  $y$ . The neuron classifies the input vector  $\mathbf{x}^T = (x_1 \ x_2)$  in two classes  $\mathcal{C}_1$  and  $\mathcal{C}_2$ . The weights are  $w_1$  and  $w_2$ , i.e. the vector  $\mathbf{W}^T = (w_1 \ w_2)$ , for inputs  $x_1$  respectively  $x_2$ . See figure 15.1 on the facing page-a. The output  $y$  represents the probability<sup>1</sup> of  $\mathbf{x} \in \mathcal{C}_1$  and  $1 - y$  the probability of  $\mathbf{x} \in \mathcal{C}_2$ .

---

<sup>1</sup>See chapter "Single Layer Neural Networks".



**Figure 15.1:** a) The simple classifier: one neuron with two inputs and one output. b) The training data for the classifier, 4 pattern vectors. The dashed line represents the class decision boundary,  $\mathbf{x}_3$  and  $\mathbf{x}_4$  are exceptions.

Let consider that there are 4 training patterns:

$$\mathbf{x}_1 = \begin{pmatrix} 5 \\ 5 \end{pmatrix} \in \mathcal{C}_2, \quad \mathbf{x}_2 = \begin{pmatrix} -5 \\ -5 \end{pmatrix} \in \mathcal{C}_1, \quad \mathbf{x}_3 = \begin{pmatrix} 0 \\ 1 \end{pmatrix} \in \mathcal{C}_1, \quad \mathbf{x}_4 = \begin{pmatrix} -1 \\ 0 \end{pmatrix} \in \mathcal{C}_2$$

The reason for this choice is the following:  $\mathbf{x}_1$  and  $\mathbf{x}_2$  are good examples of their respective class while  $\mathbf{x}_3$  and  $\mathbf{x}_4$  are exceptions — it is not reasonable to expect correct classification from one single neuron for them as the decision boundary will not be convex<sup>2</sup> (the problem is not linearly separable). See figure 15.1-b. However  $\mathbf{x}_3$  and  $\mathbf{x}_4$  do carry some information: together with  $\mathbf{x}_1$  and  $\mathbf{x}_2$  it suggest the decision boundary is rather as depicted in figure 15.1; if they would have been absent it the decision of where to draw the “line” would have been more difficult (lower probability to be the one chosen).

The neuronal activation function is chosen as the sigmoidal function:

$$y = \frac{1}{1 + \exp(-\mathbf{W}^T \mathbf{x})} = \frac{1}{1 + \exp(-w_1 x_1 - w_2 x_2)}$$

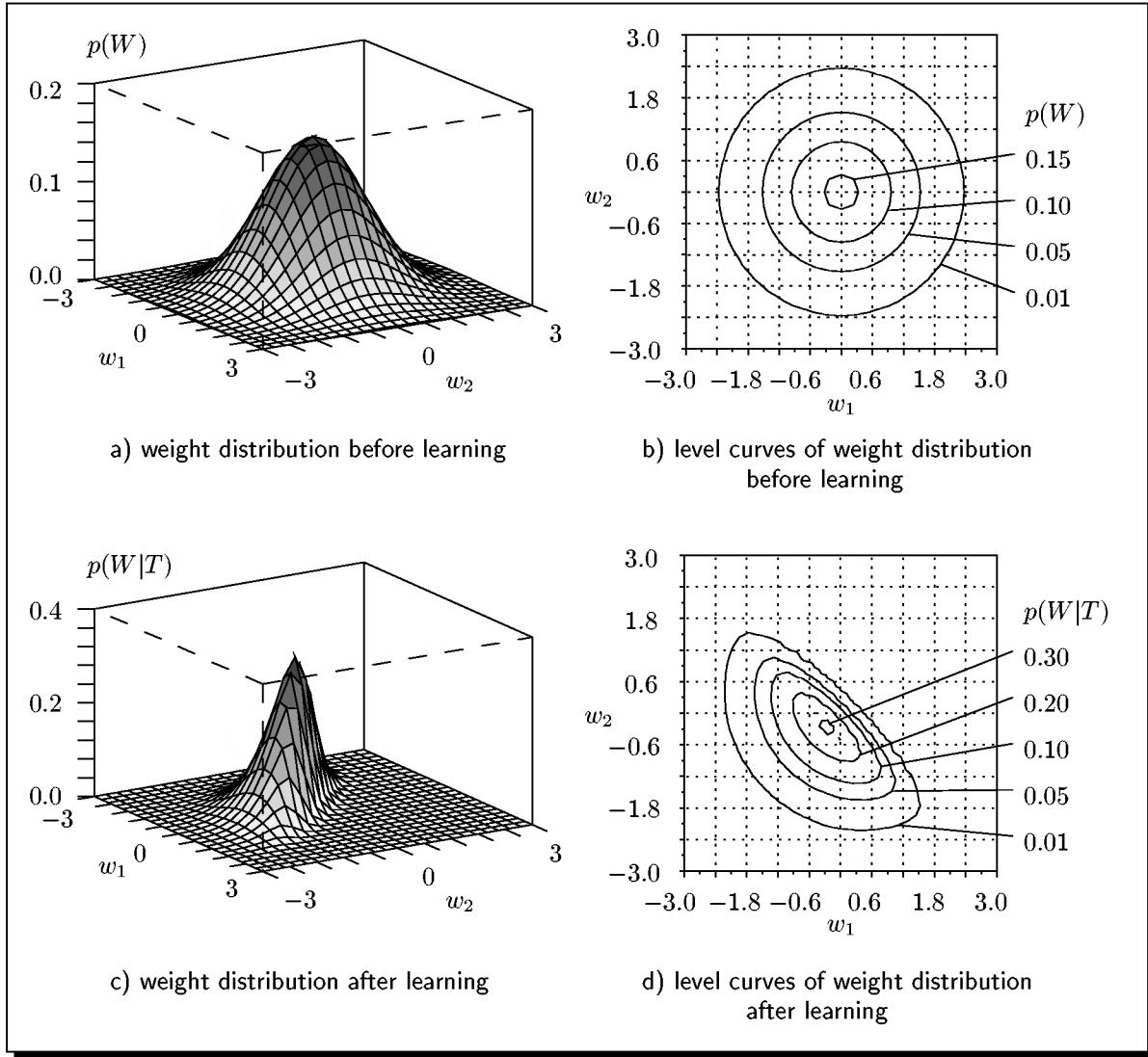
As probability of  $\mathbf{x} \in \mathcal{C}_1$  is  $y$  and probability of  $\mathbf{x} \in \mathcal{C}_2$  is  $1 - y$  then the probability of targets, conditioned on weights is:

$$p(T|W) = \prod_{\mathbf{x}_p \in \mathcal{C}_1} y(\mathbf{x}_p) \prod_{\mathbf{x}_p \in \mathcal{C}_2} [1 - y(\mathbf{x}_p)] = [1 - y(\mathbf{x}_1)] y(\mathbf{x}_2) y(\mathbf{x}_3) [1 - y(\mathbf{x}_4)]$$

The prior probability density for weights is chosen as a Gaussian of type (15.6) with  $\alpha = 1$ :

$$p(W) = p(w_1, w_2) = \frac{1}{2\pi} \exp\left(-\frac{w_1^2 + w_2^2}{2}\right)$$

<sup>2</sup>See chapter “Single Layer Neural Networks”



**Figure 15.2:** The probability density for weights: figures a) and b) show the prior probability  $p(W)$ ; figures c) and d) show the posterior probability  $p(W|T)$ .

and the graphic of this function is drawn in figure 15.2 on the facing page a) and b).

The normalization factor  $p(T) = \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} p(T|W) p(W) dw_1 dw_2$  may be calculated numerically.

Finally, the posterior probabilities of weights is  $p(W|T) = \frac{p(T|W) p(W)}{p(T)}$  and its graphic is depicted in figures 15.2–c and 15.2–d.

The best weights correspond to the maximum of  $p(W|T)$  which occurs at  $w_1 \approx -0.3$  and  $w_2 \approx -0.3$ .



### Remarks:

- ➔ Before any data is available the best weights correspond to the maximum of prior probability which occurs at  $w_1 = 0$  and  $w_2 = 0$ . This will give  $y(\mathbf{x}) = 0.5$ ,  $\forall \mathbf{x}$ , i.e. there is an equal probability of  $\mathbf{x}$  belonging to either class — the result reflecting the absence of all data on which the decision is made.
- ➔ After the data becomes available the weights are shifted to  $w_1 = -0.3$  and  $w_2 = -0.3$  which gives  $y(\mathbf{x}_1) \approx 0.0475$ ,  $y(\mathbf{x}_2) \approx 0.9525$ ,  $y(\mathbf{x}_3) \approx 0.4255$ ,  $y(\mathbf{x}_4) \approx 0.5744$ . The  $\mathbf{x}_3$  and  $\mathbf{x}_4$  are misclassified (as it should have been  $y(\mathbf{x}_3 \in \mathcal{C}_1) > 0.5$  and  $y(\mathbf{x}_4 \in \mathcal{C}_2) < 0.5$ ) but this was to be expected, and the patterns still carry some useful information (they are used to reinforce the established decision boundary).
- ➔ In general the prior probability  $p(W)$  is wide and have a low maximum and posterior probability is narrow and have a high peak(s) — this may be seen also in figure 15.2 on the preceding page.

#### 15.1.4 Gaussian Noise Model

In general, the likelihood function, i.e.  $p(T|W)$  may be written in exponential form as:

$$p(T|W) = \frac{1}{Z_T(\beta)} \exp(-\beta E_T) \quad (15.7)$$

where  $E_T$  is the error function,  $\beta$  is a parameter and  $Z_T(\beta)$  is a normalization factor which ensure that the  $p(T|W)$  is normalized:

$$Z_T(\beta) = \int_Y \exp(-\beta E_T) d\mathbf{t}_1 \dots d\mathbf{t}_P \quad (15.8)$$

❖  $E_T$ ,  $\beta$ ,  $Z_T$

Assuming a sum-of-squares error function<sup>3</sup>, and that the targets are generated from a smooth function to which a zero-mean noise have been added, then:

$$p(t|\mathbf{x}, W) \sim \exp\left(-\frac{\beta}{2} [y(\mathbf{x}, W) - t]^2\right) \quad \text{and thus} \quad (15.9)$$

$$p(T|W) = \prod_{p=1}^P p(t_p|\mathbf{x}_p, W) = \frac{1}{Z_T(\beta)} \exp\left(-\frac{\beta}{2} \sum_{p=1}^P [y(\mathbf{x}_p, W) - t_p]^2\right)$$

---

<sup>3</sup>See chapter “Error Functions”.

$$\text{i.e. } E_T = \frac{1}{2} \sum_{p=1}^P [y(\mathbf{x}_p, W) - t_p]^2 \quad (15.10)$$

and it becomes evident that  $\beta$  controls the variance:  $\sigma = 1/\sqrt{\beta}$ , and by replacing  $E_T$  into (15.8) and integrating<sup>4</sup>

$$Z_T(\beta) = \left( \frac{2\pi}{\beta} \right)^{\frac{P}{2}} \quad (15.11)$$

### 15.1.5 Gaussian Posterior Weight Distribution

❖  $Z_S$

From (15.1), (15.2) and (15.7), and defining  $Z_S \equiv p(T)$  as the normalization factor, then

$$p(W|T) = \frac{1}{Z_S} \exp(-\alpha E_W - \beta E_T) = \frac{1}{Z_S} \exp[-S(W)] \quad (15.12)$$

❖  $S$

where  $S(W) = \alpha E_W + \beta E_T$  and  $Z_S(\alpha, \beta) = \int \int \exp[-S(W)] d\mathbf{W} dt_1 \dots dt_P$ .

Considering the expression (15.10) of  $E_T$  and (15.4) of  $E_W$  then

$$S(W) = \frac{\beta}{2} \sum_{p=1}^P [y(\mathbf{x}_p, W) - t_p]^2 + \frac{\alpha}{2} \sum_{i=1}^{N_W} w_i^2$$

which represents a sum-of-squares error function with a weight decay regularization function<sup>5</sup>. Since an overall multiplicative term does not count, the multiplicative constant of the regularization term is  $\alpha/\beta$ . For the most probable weight vector  $W^*$  for which  $p(W|T)$  is maximum,  $S$  is minimum and thus the regularized error is minimum.

❖  $W^*$

### 15.1.6 Consistent Prior Weight Distribution

The plain weight decay, as the one from (15.4), is not consistent with linear transformation<sup>6</sup>. However for two layers the simple weight decay may be changed to the form

$$E_W = \frac{\alpha_1}{2} \sum_{\text{hidden layer}} w^2 + \frac{\alpha_2}{2} \sum_{\text{output layer}} w^2$$

### 15.1.7 Approximation Of Weight Distribution

In order to simplify the computational process of finding the (maximum of) posterior probability, the function  $S(W)$  may be developed in series and only the most significant terms retained:

$$S(W) = S(W^*) + \frac{1}{2} (\mathbf{W} - \mathbf{W}^*)^T H (\mathbf{W} - \mathbf{W}^*) + \mathcal{O}[(W - W^*)^3]$$

<sup>4</sup>See mathematical appendix, regarding the Gaussian integrals.

<sup>5</sup>See chapter "Learning Optimization".

<sup>6</sup>See chapter "Learning Optimization", also for the form of changed weight decay expression.

$$\simeq S(W^*) + \frac{1}{2}(\mathbf{W} - \mathbf{W}^*)^T H(\mathbf{W} - \mathbf{W}^*)$$

where  $H_S$  is the Hessian of the *regularized* error function (the term proportional in  $(\mathbf{W} - \mathbf{W}^*)$  is zero due to the fact that the series development is done around minimum).  $\diamond H_S$

Considering the gradient as a vectorial operator  $\nabla^T = \begin{pmatrix} \frac{\partial}{\partial w_1} & \cdots & \frac{\partial}{\partial w_{N_W}} \end{pmatrix}$  then, considering a weigh decay  $E_W$ , the Hessian is<sup>7</sup>:

$$H_S = (\nabla \nabla^T) S(W^*) = \beta (\nabla \nabla^T) E_T(W^*) + \alpha I = \beta H + \alpha I$$

The posterior distribution becomes:

$$p(W|T) = \frac{1}{Z_S^*} \exp \left[ -S(W^*) - \frac{1}{2} (\mathbf{W} - \mathbf{W}^*)^T H_S(\mathbf{W} - \mathbf{W}^*) \right] \quad (15.13)$$

where  $Z_S^*$  is the normalization coefficient for the distribution (15.13) and then<sup>8</sup>:  $\diamond Z_S^*$

$$\begin{aligned} Z_S^* &= \exp[-S(W^*)] \int_W \exp \left[ -\frac{1}{2} (\mathbf{W} - \mathbf{W}^*)^T H_S(\mathbf{W} - \mathbf{W}^*) \right] d\mathbf{W} \\ &= \exp[-S(W^*)] (2\pi)^{N_W/2} |H_S|^{-1/2} \end{aligned} \quad (15.14)$$

## 15.2 Network Outputs Distribution

The probability density of targets is dependent on input vector and on weights trough the training set:

$$p(t|\mathbf{x}, T) = \int_W p(t|\mathbf{x}, W) p(W|T) dW$$

Considering a sum-of-squares error function, i.e.  $p(t|\mathbf{x}, W)$  given by (15.9) and the quadratic approximation of  $p(W|T)$  given by (15.13), then

$$p(t|\mathbf{x}, T) \propto \int_W \exp \left( -\frac{\beta}{2} [t - y(\mathbf{x}, W)]^2 \right) \exp \left( \frac{1}{2} (\mathbf{W} - \mathbf{W}^*)^T H_S(\mathbf{W} - \mathbf{W}^*) \right) d\mathbf{W} \quad (15.15)$$

Let define  $\mathbf{g} \equiv \nabla y|_{W^*}$  and  $\Delta \mathbf{W} = \mathbf{W} - \mathbf{W}^*$ . As the error function was approximated after a series development around  $W^*$ , the same can be done with  $y(\mathbf{x}, W)$ :  $\diamond \mathbf{g}, \Delta W, y^*$

$$y(\mathbf{x}, W) \simeq y^* + \mathbf{g}^T \Delta \mathbf{W} \quad \text{where } y^* \equiv y(\mathbf{x}, W^*)$$

---

<sup>7</sup>The method of calculation of the Hessian  $H$  of the error function is discussed in chapter “Multi Layer Neural Networks”.

<sup>8</sup>See also mathematical appendix.

<sup>15.2</sup>See [Bis95] pp. 398–402.

The targets probability density becomes:

$$p(t|\mathbf{x}, T) = C \int_W \exp \left( -\frac{\beta}{2} [t - y^* - \mathbf{g}^T \Delta \mathbf{W}]^2 - \frac{1}{2} \Delta \mathbf{W}^T H_S \Delta \mathbf{W} \right) d\mathbf{W}$$

where  $C$  is a multiplicative constant such that  $\int_Y p(t|\mathbf{x}, T) dt = 1$ .

As  $(\mathbf{g}^T \Delta \mathbf{W})^2 = \Delta \mathbf{W}^T \mathbf{g} \mathbf{g}^T \Delta \mathbf{W}$  (by using the matrix property:  $(AB)^T = B^T A^T$ ) then:

$$\begin{aligned} p(t|\mathbf{x}, T) &= C \exp \left[ -\frac{\beta}{2} (t - y^*)^2 \right] \times \\ &\quad \times \int_W \exp \left[ -\frac{1}{2} \Delta \mathbf{W}^T (H_S + \beta \mathbf{g} \mathbf{g}^T) \Delta \mathbf{W} + \beta(t - y^*) \mathbf{g}^T \Delta \mathbf{W} \right] d\mathbf{W} \end{aligned}$$

which represent a Gaussian integral with a linear term<sup>9</sup>, and then:

$$\begin{aligned} p(t|\mathbf{x}, T) &= C \exp \left[ -\frac{\beta}{2} (t - y^*)^2 \right] (2\pi)^{N_W/2} |H_S + \beta \mathbf{g} \mathbf{g}^T|^{-1/2} \times \\ &\quad \times \exp \left[ \frac{\beta^2}{2} (t - y^*)^2 \mathbf{g}^T (H_S + \beta \mathbf{g} \mathbf{g}^T)^{-1} \mathbf{g} \right] \\ &= C' \exp \left\{ -\frac{(t - y^*)^2}{2} \left[ \beta - \beta^2 \mathbf{g}^T (H_S + \beta \mathbf{g} \mathbf{g}^T)^{-1} \mathbf{g} \right] \right\} \end{aligned}$$

❖  $C'$

where  $C' = \text{const.}$   
The normalization condition  $\int_Y p(t|\mathbf{x}, T) dt = 1$  (where  $Y \equiv (-\infty, \infty)$ ) gives:

$$C' \left[ \frac{2\pi}{\beta - \beta^2 \mathbf{g}^T (H_S + \beta \mathbf{g} \mathbf{g}^T)^{-1}} \right]^{1/2} = 1$$

❖  $\sigma_t$

leading to the distribution wanted in the form:

$$\begin{aligned} p(t|\mathbf{x}, T) &= \frac{1}{\sqrt{2\pi\sigma_t^2}} \exp \left[ -\frac{(t - y^*)^2}{2\sigma_t^2} \right] \quad \text{where} \\ \sigma_t^2 &= \frac{1}{\beta - \beta^2 \mathbf{g}^T (H_S + \beta \mathbf{g} \mathbf{g}^T)^{-1} \mathbf{g}} \end{aligned} \tag{15.16}$$

representing a Gaussian distribution<sup>10</sup> with  $\sigma_t$  the variance and  $\langle t \rangle = y^*$  the mean of  $t$ .

To simplify the expression of  $\sigma_t^2$  first the following transformation is done  $H_S \rightarrow H'_S = H_S/\beta \Rightarrow H_S^{-1} \rightarrow H'^{-1}_S = \beta H_S$ , and then the numerator and denominator are both multiplied by the number  $\mathbf{g}^T(I + H'^{-1}_S \mathbf{g} \mathbf{g}^T)\mathbf{g}$  which gives:

$$\beta\sigma_t^2 = \frac{\mathbf{g}^T(I + H'^{-1}_S \mathbf{g} \mathbf{g}^T)\mathbf{g}}{\mathbf{g}^T(I + H'^{-1}_S \mathbf{g} \mathbf{g}^T)\mathbf{g} - \mathbf{g}^T(H'_S + \mathbf{g} \mathbf{g}^T)^{-1} \mathbf{g} \mathbf{g}^T(I + H'^{-1}_S \mathbf{g} \mathbf{g}^T)\mathbf{g}}$$

<sup>9</sup>See the mathematical appendix.

<sup>10</sup>See also the chapter "Pattern Recognition".

The first term of the denominator reduces to  $\mathbf{g}^T \mathbf{g} + \mathbf{g}^T H_S'^{-1} \mathbf{g} \mathbf{g}^T \mathbf{g}$  while for the second, the matrix property  $(AB)^{-1} = B^{-1} A^{-1}$  is applied, in the form:

$$(H_S' + \mathbf{g} \mathbf{g}^T)^{-1} = [H_S'(I + H_S'^{-1} \mathbf{g} \mathbf{g}^T)]^{-1} = (I + H_S'^{-1} \mathbf{g} \mathbf{g}^T)^{-1} H_S'^{-1}$$

and then the second term of the denominator becomes:

$$\mathbf{g}^T (I + H_S'^{-1} \mathbf{g} \mathbf{g}^T)^{-1} H_S'^{-1} \mathbf{g} \mathbf{g}^T (I + H_S'^{-1} \mathbf{g} \mathbf{g}^T) \mathbf{g}$$

To simplify the expression a  $\mathbf{g}^T I \mathbf{g}$  is added and subtracted and, in the added term,  $I$  is changed to  $I = (I + H_S'^{-1} \mathbf{g} \mathbf{g}^T)^{-1} I (I + H_S'^{-1} \mathbf{g} \mathbf{g}^T)$ . By regrouping, the whole expression is reduced to  $\mathbf{g}^T H_S'^{-1} \mathbf{g} \mathbf{g}^T \mathbf{g}$ .

Finally the variance (15.16) becomes:

$$\sigma_t^2 = \frac{1}{\beta} + \mathbf{g}^T H_S \mathbf{g}$$

As it measures the “width” of distribution (see the remarks below) the variance  $\sigma_t$  may be considered as an error bar for  $y^*$ . The  $1/\beta$  part is due to noise in data while the  $\mathbf{g}^T H_S \mathbf{g}$  part is due to the “width” of the posterior probability of weights. The error bar is between  $y(\mathbf{x}) - C\sigma$  and  $y(\mathbf{x}) + C\sigma$ , the value of constant  $C$  to be established application dependent (see also the remarks below).



### Remarks:

- ➡ Considering a unidimensional distribution

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x - \langle x \rangle)^2}{2\sigma^2}\right]$$

the “width” of it — see figure 15.3 on the following page — is proportional with the variance  $\sigma$ . The width of distribution, at the level half of maximum:

$$p(x) = \frac{1}{2} p_{\max} = \frac{1}{2} \frac{1}{\sqrt{2\pi}\sigma} \Leftrightarrow x = \langle x \rangle \pm \sqrt{2 \ln 2} \cdot \sigma$$

is only  $\sigma$  dependent, being “width” =  $2\sqrt{2 \ln 2} \cdot \sigma \approx 2.35 \cdot \sigma$ .

- ➡ The width of probability distribution equals  $2\sigma$  for  $x = \langle x \rangle \pm \sigma$  at which point the probability drops at  $p(\langle x \rangle \pm \sigma) = p_{\max}/\sqrt{e} \approx 0.606 p_{\max}$ .

### 15.2.1 Generalized Linear Networks

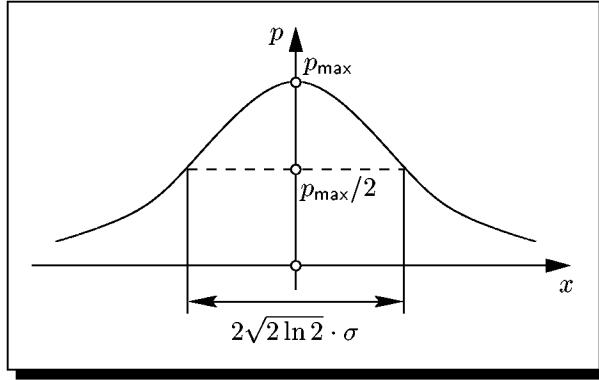
The generalized linear network have one layer and the output is of the form<sup>11</sup>

❖  $\varphi, \varphi_j$

$$y(\mathbf{x}, W) = \mathbf{W}^T \varphi(\mathbf{x}) \quad \text{where} \quad \varphi^T(\mathbf{x}) = (\varphi_1(\mathbf{x}) \quad \dots \quad \varphi_H(\mathbf{x}))$$

$\varphi_j(\mathbf{x})$  being the activation function of neuron  $j$ .

<sup>11</sup>See chapter “Radial Basis Function Networks”.



**Figure 15.3:** The Gaussian width. At  $p(x) = p_{\max}/2$  the width is  $2\sqrt{2 \ln 2} \cdot \sigma$ .

The error function of this network is:

$$S(W) = \frac{\beta}{2} \sum_{p=1}^P [t_p - \mathbf{W}^T \varphi(\mathbf{x}_p)]^2 + \frac{\alpha}{2} \|\mathbf{W}\|^2$$

and, as is a quadratic function in  $\mathbf{W}$ , then the weight probability density is strictly Gaussian with one single maximum.

❖  $\mathbf{W}^*, y^*$

Considering  $\mathbf{W}^*$  as the weight vector corresponding to the maxima of posterior weight probability and  $y^*(\mathbf{x}) \equiv y(\mathbf{x}, W^*)$  then:

$$y(\mathbf{x}, W) = y(\mathbf{x}) + \Delta \mathbf{W}^T \varphi(\mathbf{x})$$

where  $\Delta \mathbf{W} = \mathbf{W} - \mathbf{W}^*$ .

The Hessian is:  $H_S = (\nabla \cdot \nabla^T) S(W^*) = \beta \sum_{p=1}^P \varphi(\mathbf{x}_p) \varphi(\mathbf{x}_p)^T + \alpha I$ .

The posterior probability of targets is calculated in a similar way as for (15.15):

$$p(t|\mathbf{x}, T) \propto \int_W \exp \left\{ -\frac{\beta}{2} [t - \mathbf{W}^T \varphi(\mathbf{x})]^2 - \frac{1}{2} \Delta \mathbf{W}^T H_S \Delta \mathbf{W} \right\} d\mathbf{W}$$

with a variance  $\sigma_t^2 = \frac{1}{\beta} + \varphi^T H_S \varphi$ .

## 15.3 Classification

Let consider a classification problem with two classes  $\mathcal{C}_1$  and  $\mathcal{C}_2$  with a network involving only one output  $y$  representing the probability of the input vector  $\mathbf{x}$  being of class  $\mathcal{C}_1$ , then  $(1 - y)$  represents the probability of being of class  $\mathcal{C}_2$ . Obviously the targets  $t \in \{0, 1\}$  and the network output  $y \in [0, 1]$ .

<sup>15.3</sup>See [Bis95] pp. 403–406.

The likelihood function for observing the whole training set is<sup>12</sup>:

$$p(T|W) = \prod_{p=1}^P [y(\mathbf{x}_p)]^{t_p} [1 - y(\mathbf{x}_p)]^{1-t_p} = \mathcal{L} = \exp[-G(T|W)] \quad (15.17)$$

where  $G(T|W)$  is the cross-entropy:

$$G(T|W) = - \sum_{p=1}^P \{t_p \ln y(\mathbf{x}_p) + (1 - t_p) \ln[1 - y(\mathbf{x}_p)]\}$$

❖  $G$



### Remarks:

→ The normalization condition for the distribution (15.17) is  $\sum_{t_p \in \{0,1\}} p(T|W) = 1$ .

After the replacement of  $p(T|W)$  from (15.17), the result is a product of terms of the form  $y(\mathbf{x}_p) + [1 - y(\mathbf{x}_p)] = 1$ , i.e. the distribution  $p(T|W)$  is normalized.

The neuron activation function is taken as the sigmoidal  $y(\mathbf{x}) = f(a) = \frac{1}{1+\exp(-a)}$  where  $a = \sum_j w_j z_j$ ;  $w_j$  being the weights of connections from hidden neurons  $z_j$  coming into the output neuron  $y$  and  $a$  is the total input.

Considering a prior distribution (15.2) then the posterior distribution, from (15.1), similar to (15.12), will be:

$$p(W|T) = \frac{1}{Z_S} \exp(-G - \alpha E_W) = \frac{1}{Z_S} \exp[-S(\mathbf{w})]$$

where  $Z_S$  is the normalization coefficient.

Considering a quadratic approximation as in (15.13), the distribution may be approximated as:

$$p(W|T) = \frac{1}{Z_S^*} \exp \left( -S(W^*) - \frac{1}{2} \Delta \mathbf{W}^T H_S \Delta \mathbf{W} \right) \quad (15.18)$$

$Z_S^*$  being the normalization coefficient, and  $\int_W p(W|T) d\mathbf{W} = 1$  gives:

❖  $Z_S^*$

$$Z_S^* = \int_W \exp \left( -S(W^*) - \frac{1}{2} \Delta \mathbf{W}^T H_S \Delta \mathbf{W} \right) d\mathbf{W} = e^{-S(W^*)} \prod_{i=1}^{N_W} \sqrt{\frac{2\pi}{\lambda_i}} \quad (15.19)$$

(see also the mathematical appendix).

As the network output is interpreted as probability, i.e.  $y(\mathbf{x}, W) = P(C_1|\mathbf{x}, W)$  then, for a new vector  $\mathbf{x}$  the Bayesian learning involves an integration over all possible weights, in the form:

$$P(C_1|\mathbf{x}, T) = \int_W P(C_1|\mathbf{x}, W) p(W) d\mathbf{W} = \int_W y(\mathbf{x}, W) p(W) d\mathbf{W}$$

---

<sup>12</sup>See chapter "Error functions" — section "Cross entropy".

Assuming a linear approximation of weights for the total input  $a$  then

$$a(\mathbf{x}, W) = a^*(\mathbf{x}) + \mathbf{g}^T \Delta \mathbf{W} \quad (15.20)$$

❖  $a^*$ ,  $\mathbf{g}$

where  $a^*(\mathbf{x}) \equiv a(\mathbf{x}, W^*)$  and  $\mathbf{g}(\mathbf{x}) \equiv \nabla a(\mathbf{x}, W)|_{W^*}$ .

The posterior probability of  $a$  may be written as:

$$p(a|\mathbf{x}, T) = \int_W p(a|\mathbf{x}, W) p(W|T) d\mathbf{W} = \int_W \delta_D(a - a^* - \mathbf{g}^T \Delta \mathbf{W}) p(W|T) d\mathbf{W} \quad (15.21)$$

Since, from (15.20),  $a$  and  $\Delta \mathbf{W}$  are linearly related, and  $p(W|T)$  is a Gaussian, then the distribution of  $a$  is also a Gaussian of the form:

$$p(a|\mathbf{x}, T) = \frac{1}{\sqrt{2\pi s^2}} \exp\left(-\frac{(a - \langle a \rangle)^2}{2s^2}\right) \quad (15.22)$$

❖  $\langle a \rangle$ ,  $s$

with the mean  $\langle a \rangle$  and variance  $s$  being:

$$\langle a \rangle = a^* \quad \text{and} \quad s^2 = \mathbf{g}^T H_S \mathbf{g}$$

*Proof.* 1. Using (15.21) and (15.18):

$$\begin{aligned} \langle a \rangle &= \frac{e^{-S(W^*)}}{Z_S^*} \int_A \int_W a \delta_D(a - a^* - \mathbf{g}^T \Delta \mathbf{W}) \exp\left(-\frac{1}{2} \Delta \mathbf{W}^T H_S \Delta \mathbf{W}\right) d\mathbf{W} da \\ &= \frac{e^{-S(W^*)}}{Z_S^*} \int_W (a^* + \mathbf{g}^T \Delta \mathbf{W}) \exp\left(-\frac{1}{2} \Delta \mathbf{W}^T H_S \Delta \mathbf{W}\right) d\mathbf{W} \end{aligned}$$

Replacing (15.19), considering that  $a^* = \text{const.}$  and  $\int_W \mathbf{g}^T \Delta \mathbf{W} \exp\left(-\frac{1}{2} \Delta \mathbf{W}^T H_S \Delta \mathbf{W}\right) d\mathbf{W} = 0$  because the integrand is odd function<sup>13</sup> in  $\Delta \mathbf{W}$  (and  $d\mathbf{W} = d(\Delta \mathbf{W})$ ) and the integral is done over a origin centered interval, then  $\langle a \rangle = a^*$ .

2. The variance is:

$$\begin{aligned} s^2 &= \int_A \int_W (a - a^*)^2 \delta_D(a - a^* - \mathbf{g}^T \Delta \mathbf{W}) \frac{e^{-S(W^*)}}{Z_S^*} \exp\left(-\frac{1}{2} \Delta \mathbf{W}^T H_S \Delta \mathbf{W}\right) d\mathbf{W} \\ &= \frac{e^{-S(W^*)}}{Z_S^*} \int_W (\mathbf{g}^T \Delta \mathbf{W})^2 \exp\left(-\frac{1}{2} \Delta \mathbf{W}^T H_S \Delta \mathbf{W}\right) d\mathbf{W} \end{aligned}$$

Let  $\mathbf{u}_i$  be the eigenvectors and  $\lambda_i$  the eigenvalues of  $H_S$ , i.e.  $H_S \mathbf{u}_i = \lambda_i \mathbf{u}_i$ . As  $H_S$  is symmetrical, then it is possible<sup>14</sup> to build an orthogonal system of eigenvectors. Let consider:

$$\Delta \mathbf{W} = \sum_i \Delta w_i \mathbf{u}_i \quad \text{and} \quad \mathbf{g} = \sum_i g_i \mathbf{u}_i$$

and by replacing into the above expression of variance:

$$s^2 = \frac{e^{-S(W^*)}}{Z_S^*} \int_W \left( \sum_i g_i \Delta w_i \right)^2 \exp\left(-\frac{1}{2} \lambda_i \Delta w_i^2\right) d\mathbf{W}$$

---

<sup>13</sup>An function  $f$  is odd when  $f(-x) = -f(x)$ .

<sup>14</sup>See mathematical appendix.

Developing the square of the sum, the variance becomes a sum of two types of integrals. The first one,  $\forall i \neq j$ , is of the form

$$\begin{aligned} & \int_{-\infty}^{\infty} \int_{-\infty}^{\infty} g_i \Delta w_i g_j \Delta w_j \exp \left( -\frac{1}{2} (\lambda_i \Delta w_i^2 + \lambda_j \Delta w_j^2) \right) d(\Delta w_i) d(\Delta w_j) = \\ &= \left[ \int_{-\infty}^{\infty} g_i \Delta w_i \exp \left( -\frac{1}{2} \lambda_i \Delta w_i^2 \right) d(\Delta w_i) \right] \left[ \int_{-\infty}^{\infty} g_j \Delta w_j \exp \left( -\frac{1}{2} \lambda_j \Delta w_j^2 \right) d(\Delta w_j) \right] = 0 \end{aligned}$$

because the integrand are odd function and the integral is done over origin centered interval. The second one is for  $i = j$ :

$$\int_{-\infty}^{\infty} (g_i \Delta w_i)^2 \exp \left( -\frac{\lambda_i \Delta w_i^2}{2} \right) d(\Delta w_i) = \frac{g_i^2}{\lambda_i} \sqrt{\frac{2\pi}{\lambda_i}}$$

As the  $H_S$  matrix may be diagonalized using the eigenvalues then  $s^2 = \sum_{i=1}^{N_W} \frac{1}{\lambda_i} g_i^2 = \mathbf{g}^T H_S \mathbf{g}$ .  $\square$

The posterior probability  $P(\mathcal{C}_1|\mathbf{x}, T)$  may be written in terms of total neuronal input  $a$  as:

$$P(\mathcal{C}_1|\mathbf{x}, T) = \int_A P(\mathcal{C}_1|a) p(a|\mathbf{x}, T) da = \int_A f(a) p(a|\mathbf{x}, T) da$$

which does not have generally an analytic solution but may be approximated by:

$$P(\mathcal{C}_1|\mathbf{x}, T) \simeq f(a^* \kappa(s)) \quad \text{where} \quad \kappa(s) = \left( 1 + \frac{\pi s^2}{8} \right)^{-\frac{1}{2}}$$

### Remarks:

- For the simple, two class problem described above, the decision boundary is established for  $P(\mathcal{C}_1|\mathbf{x}, T) = 0.5$  which corresponds to  $a = 0$ . The same result is obtained using just the most probable weights  $W^*$  and just the network output:  $y(\mathbf{x}, W^*) = 0.5 \Rightarrow a = 0$ . Thus the two methods give the same results unless there are some more complex rules involved, e.g. a loss matrix.

## 15.4 The Evidence Approximation For $\alpha$ And $\beta$

As the parameters  $\alpha$  and  $\beta$  which do appear in the expression of posterior weight distribution — see equation (15.12) — are them-self not known, then the Bayesian framework require an integration over all possible values:

$$p(W|T) = \iint p(W|\alpha, \beta, T) p(\alpha, \beta|T) d\alpha d\beta \quad (15.23)$$

One possible way of dealing with  $\alpha$  and  $\beta$  parameters is known as the *evidence approximation* and is discussed below.

The posterior distribution of  $\alpha$  and  $\beta$ , i.e.  $p(\alpha, \beta|T)$ , it is assumed to have a “sharp peak”

<sup>15.4</sup> See [Bis95] pp. 406–415.

evidence approximation

❖  $p(\alpha, \beta|T)$ ,  $\alpha^*$ ,  $\beta^*$

around the most probable values  $\alpha^*$  and  $\beta^*$ . Then  $p(\alpha^*, \beta^*|T) \lesssim 1$  and, using also the normalization condition  $\iint p(\alpha^*, \beta^*|T) d\alpha d\beta = 1$ , the distribution (15.23) may be approximated as:

$$p(W|T) \simeq p(W|\alpha^*, \beta^*, T) \iint p(\alpha, \beta|T) d\alpha d\beta = p(W|\alpha^*, \beta^*, T)$$

i.e. the integral over all possible values of  $\alpha$  and  $\beta$  is replaced with the use of the most probable values:  $\alpha^*$  and  $\beta^*$ .

hyperprior

To find the most probable  $\alpha^*$  and  $\beta^*$  values, it is necessary to estimate the posterior probability for  $\alpha$  and  $\beta$ . This is achieved by using the Bayes theorem  $p(\alpha, \beta|T) = \frac{p(T|\alpha, \beta) p(\alpha, \beta)}{p(T)}$  where  $p(\alpha, \beta)$  is the prior probability density, also known as the *hyperprior*. If little is known about the model to be build then the hyperprior should give a relatively equal value for all possible  $\alpha$  and  $\beta$  parameters, the same way  $p(W)$  prior operates. The term  $p(T|\alpha, \beta)$  is named *evidence*. The term  $p(T)$  is the normalization factor.

evidence

The evidence may be written in the form of explicit dependencies over  $\alpha$  and  $\beta$  in the posterior distribution of targets and weights:

$$p(T|\alpha, \beta) = \int p(T|W, \alpha, \beta) p(W|\alpha, \beta) dW = \int p(T|W, \beta) p(W|\alpha) dW$$

as the prior weight distribution is independent of  $\beta$  (which is data related)  $p(W|\alpha, \beta) = p(W|\alpha)$  and the likelihood function is independent of  $\alpha$ :  $p(T|W, \alpha, \beta) = p(T|W, \beta)$  — see equations (15.2) and (15.7).

From the same set of equations — (15.2) and (15.7):

$$p(T|\alpha, \beta) = \frac{1}{Z_W(\alpha) Z_T(\beta)} \int \exp(-S(W)) dW = \frac{Z_S(\alpha, \beta)}{Z_W(\alpha) Z_T(\beta)}$$

and considering the values of  $Z_S$ ,  $Z_W$  and  $Z_T$  from (15.14), (15.5) and (15.11) respectively and as  $S(W^*) = \alpha E_W^* + \beta E_T^*$  then:

$$\ln p(T|\alpha, \beta) = -\alpha E_W^* - \beta E_T^* - \frac{1}{2} \ln |H_S| + \frac{N_W}{2} \ln \alpha + \frac{P}{2} \ln \beta - \frac{N_W + P}{2} \ln(2\pi) \quad (15.24)$$

Considering  $E_W$  an quadratic form on weights then:

$$H_S = (\nabla \cdot \nabla^T)(\alpha E_W + \beta E_T) = \alpha I + \beta (\nabla \cdot \nabla^T) E_T = \alpha I + H$$

$H$  being the Hessian of error function. As the Hessian is a symmetric matrix then it may be diagonalized using its eigenvalues<sup>15</sup>  $\lambda_i$  and, obviously,  $H_S$  have the eigenvalues  $\lambda_i + \alpha$  and  $|H_S| = \prod_i (\lambda_i + \alpha)$ . Finally:

$$\frac{d}{d\alpha} \ln |H_S| = \frac{d}{d\alpha} \ln \left( \prod_{i=1}^{N_W} (\lambda_i + \alpha) \right) = \sum_{i=1}^{N_W} \frac{1}{\lambda_i + \alpha} = \text{Tr } H_S^{-1} \quad (15.25)$$

where  $1/(\lambda_i + \alpha)$  are the eigenvalues of  $H_S^{-1}$  and the  $\lambda_i$  eigenvalues were supposed to be  $\alpha$ -independent, i.e.  $E_T$  is also a quadratic of weights.

---

<sup>15</sup>See mathematical appendix.

The condition of minimum of  $\ln p(T|\alpha, \beta)$  with respect to  $\alpha$  is  $\frac{\partial \ln p}{\partial \alpha} = 0$  which, from (15.24), gives:

◆  $\gamma, \gamma_i$

$$2\alpha E_W^* = N_W - \sum_{i=1}^{N_W} \frac{\alpha}{\lambda_i + \alpha} = \gamma \quad \text{where} \quad \gamma = \sum_{i=1}^{N_W} \frac{\lambda_i}{\lambda_i + \alpha} = \sum_{i=1}^{N_W} \gamma_i \quad (15.26)$$

and may be interpreted as follows:

- The prior distribution of weights is usually chosen to be centered in origin and with a spherical symmetry, so, in the absence of data, the most probable weight vector is  $\mathbf{W}^* = \hat{\mathbf{0}}$  and consequently  $E_W^* = 0$ .
- When there are data available, the  $E_W^*$  shifts to a position given by (15.26). Considering a system of coordinates rotated such that the axes are parallel to the eigenvectors of  $H$  then the quantities by which  $\mathbf{W}^*$  is shifted (from the origin) along each axis are given by  $\gamma_i = \frac{\lambda_i}{\lambda_i + \alpha}$ .

For  $\lambda_i \ll \alpha \Rightarrow \gamma_i \gtrsim 0$ , i.e.  $w_i \gtrsim 0$ ,  $w_i$  is not shifted much from the origin and the main contribution to this particular weight(s) is given by the prior weight distribution.

For  $\lambda_i \gg \alpha \Rightarrow \gamma_i \lesssim 1$ , i.e.  $w_i \lesssim 1$ , the main contribution to this weight is given by data (through  $\lambda_i$ ). These kind of  $w_i$  weights are named *well-determined parameters*.

*Thus  $\gamma$  measures the effective number of weights changed by the data present in the training set* (all others being given rather small values from the prior distribution).

well-determined  
parameters  
effective number

The Hessian is  $H = \beta(\nabla \cdot \nabla^T)E_T$  this means that  $\lambda_i \propto \beta$  (as the Hessian may be diagonalized using the eigenvalues)  $\Rightarrow d\lambda_i \propto d\beta$  and then:

$$\frac{d\lambda_i}{d\beta} = \frac{\lambda_i}{\beta} \quad (15.27)$$

Using this result, and similar to the previous derivative:

$$\frac{d}{d\beta} \ln |H_S| = \frac{d}{d\beta} \ln \left( \prod_{i=1}^{N_W} (\lambda_i + \alpha) \right) = \frac{1}{\beta} \sum_{i=1}^{N_W} \frac{\lambda_i}{\lambda_i + \alpha} \quad (15.28)$$

The condition of minimum of  $\ln p(T|\alpha, \beta)$  with respect to  $\beta$  is  $\frac{\partial \ln p}{\partial \beta} = 0$  which, from (15.24), gives:

$$2\beta E_T^* = P - \sum_{i=1}^{N_W} \frac{\lambda_i}{\lambda_i + \alpha} = P - \gamma \quad (15.29)$$

and the same comments as above apply.

As  $S = \alpha E_W + \beta E_T$  then, from (15.26) and (15.29),  $S(W^*) = P/2$ .

To find out the  $\alpha$  and  $\beta$  parameters an iterative algorithm may be used. Starting with some initial values  $\alpha_0$  and  $\beta_0$ , the values at step  $t+1$  may be computed from (15.26) and (15.29):

$$\begin{cases} \alpha_{(t+1)} = \frac{\gamma_{(t)}}{2E_{W(t)}} \\ \beta_{(t+1)} = \frac{P - \gamma_{(t)}}{2E_{T(t)}} \end{cases}$$

For large training sets, i.e.  $P \gg N_W$ , all weights may be approximated as well-determined parameters and then  $\gamma_i \approx 1$  and  $\gamma \approx N_W$  which leads to the much faster updating formulas:

$$\begin{cases} \alpha_{(t+1)} = \frac{N_W}{2E_{W(t)}} \\ \beta_{(t+1)} = \frac{P}{2E_{T(t)}} \end{cases}$$

Considering a Gaussian approximation for  $p(T|\alpha, \beta)$  then:

$$p(T|\ln \alpha, \ln \beta) = p(T|\ln \alpha^*, \ln \beta^*) \exp\left(-\frac{1}{2} \Delta_{\ln \alpha}^T \Sigma^{-1} \Delta_{\ln \beta}\right)$$

where  $\Delta_{\ln \alpha} = \begin{pmatrix} \ln \alpha - \ln \alpha^* \\ \ln \beta - \ln \beta^* \end{pmatrix}$ ,  $\Sigma = \begin{pmatrix} \sigma_{\ln \alpha}^2 & \sigma_{\ln \alpha}^2 \\ \sigma_{\ln \beta}^2 & \sigma_{\ln \beta}^2 \end{pmatrix}$  and:

$$\Sigma^{-1} = \begin{pmatrix} \sigma_{\ln \alpha}^{(-1)2} & \sigma_{\ln \beta}^{(-1)2} \\ \sigma_{\ln \beta}^{(-1)2} & \sigma_{\ln \beta}^{(-1)2} \end{pmatrix} = \frac{1}{\sigma_{\ln \alpha}^2 \sigma_{\ln \beta}^2 - \sigma_{\ln \alpha}^4} \begin{pmatrix} \sigma_{\ln \beta}^2 & -\sigma_{\ln \alpha}^2 \\ -\sigma_{\ln \alpha}^2 & \sigma_{\ln \alpha}^2 \end{pmatrix}$$

Note that the logarithm of  $\alpha$  and  $\beta$  have been considered (instead of  $\alpha, \beta$ ) as they are scaling hyper-parameters.

From the above equations and because  $\frac{\partial}{\partial \ln \alpha} = \alpha \frac{\partial}{\partial \alpha}$ , and same for  $\beta$ , then:

$$\sigma_{\ln \alpha}^{(-1)2} = \frac{\sigma_{\ln \beta}^2}{\sigma_{\ln \alpha}^2 \sigma_{\ln \beta}^2 - \sigma_{\ln \alpha}^4} = -\frac{\partial^2 \ln p}{\partial (\ln \alpha)^2} = -\alpha \frac{\partial}{\partial \alpha} \left( \alpha \frac{\partial \ln p}{\partial \alpha} \right)$$

$$\sigma_{\ln \beta}^{(-1)2} = \frac{\sigma_{\ln \alpha}^2}{\sigma_{\ln \alpha}^2 \sigma_{\ln \beta}^2 - \sigma_{\ln \beta}^4} = -\frac{\partial^2 \ln p}{\partial (\ln \beta)^2} = -\beta \frac{\partial}{\partial \beta} \left( \beta \frac{\partial \ln p}{\partial \beta} \right)$$

$$\sigma_{\ln \alpha}^{(-1)2} = \frac{-\sigma_{\ln \alpha}^2}{\sigma_{\ln \alpha}^2 \sigma_{\ln \beta}^2 - \sigma_{\ln \alpha}^4} = \frac{\partial^2 \ln p}{\partial \ln \alpha \partial \ln \beta} = \beta \frac{\partial}{\partial \beta} \left( \alpha \frac{\partial \ln p}{\partial \alpha} \right)$$

By using (15.24) when calculating the last set of derivatives from the above equations (and using (15.25), (15.28) and (15.27) as well):

$$\sigma_{\ln \alpha}^{(-1)2} = \alpha E_W^* + \frac{1}{2} \sum_{i=1}^{N_W} \frac{\alpha \lambda_i}{(\lambda_i + \alpha)^2}, \quad \sigma_{\ln \beta}^{(-1)2} = \beta E_D^* + \frac{1}{2} \sum_{i=1}^{N_W} \frac{\alpha \lambda_i}{(\lambda_i + \alpha)^2}$$

$$\sigma_{\ln \alpha}^{(-1)2} = \frac{1}{2} \sum_{i=1}^{N_W} \frac{\alpha \lambda_i}{(\lambda_i + \alpha)^2}$$

and then, using (15.26) and (15.29):

$$\sigma_{\ln \alpha}^{(-1)2} = \frac{\gamma}{2} + \sigma_{\ln \alpha}^{(-1)2} \quad \text{and} \quad \sigma_{\ln \beta}^{(-1)2} = \frac{N - \gamma}{2} + \sigma_{\ln \beta}^{(-1)2}$$

The  $\sigma_{\ln \frac{\alpha}{\beta}}^{(-1)2}$  is a sum of terms  $\frac{\alpha \lambda_i}{(\lambda_i + \alpha)^2}$ ; for  $\lambda_i \gg \alpha$  it reduces to  $\alpha/\lambda_i \ll 1$  while for  $\lambda_i \ll \alpha$

it reduces to  $\lambda_i/\alpha \ll 1$ ; the only significant terms come from  $\lambda \simeq \alpha$  which are usually in a small number. Then the the following approximations may be performed:

$$\begin{aligned} \sigma_{\ln \frac{\alpha}{\beta}}^{(-1)2} &\ll \sigma_{\ln \alpha}^{(-1)2} \quad , \quad \sigma_{\ln \frac{\alpha}{\beta}}^{(-1)2} \ll \sigma_{\ln \beta}^{(-1)2} \\ \sigma_{\ln \alpha}^{(-1)2} &\simeq \frac{1}{\sigma_{\ln \alpha}^2} \simeq \frac{\gamma}{2} \quad , \quad \sigma_{\ln \beta}^{(-1)2} \simeq \frac{1}{\sigma_{\ln \beta}^2} \simeq \frac{P - \gamma}{2} \end{aligned} \quad (15.30)$$

and the  $\alpha$  and  $\beta$  parameters may be considered statistically independent and their distribution is of the form:

$$\begin{aligned} p(T | \ln \alpha) &= p(T | \ln \alpha^*) \exp \left( -\frac{(\ln \alpha - \ln \alpha^*)^2}{2\sigma_{\ln \alpha}^2} \right) \\ p(T | \ln \beta) &= p(T | \ln \beta^*) \exp \left( -\frac{(\ln \beta - \ln \beta^*)^2}{2\sigma_{\ln \beta}^2} \right) \end{aligned} \quad (15.31)$$

## ► 15.5 Integration Over $\alpha$ And $\beta$

The Bayesian approach require an integration over all possible values for unknown parameters, i.e. the posterior probability density for weights is:

$$p(W|T) = \iint p(W, \alpha, \beta | T) d\alpha d\beta = \iint p(W|T, \alpha, \beta) p(\alpha, \beta) d\alpha d\beta$$

Using the Bayes theorem (15.1) and as  $p(T|W, \alpha, \beta) = p(T|W, \beta)$  (is independent of  $\alpha$ , see also section 15.1.4) and  $p(W|\alpha, \beta) = p(W|\alpha)$  (is independent of  $\beta$ , see also section 15.1.2) and considering also that  $\alpha$  and  $\beta$  are statistically independent, i.e.  $p(\alpha, \beta) = p(\alpha)p(\beta)$  (see also section 15.4) then:

$$p(W|T) = \frac{1}{p(T)} \iint p(T|W, \beta) p(W|\alpha) p(\alpha) p(\beta) d\alpha d\beta$$

Now, a form of  $p(\alpha)$  and  $p(\beta)$  have to be chosen. The best option would be to choose them such that  $p(\ln \alpha)$  and  $p(\ln \beta)$  have a relatively large breath. One possibility, leading to easy integration, would be:

$$p(\alpha) = \frac{1}{\alpha} \quad \text{and} \quad p(\beta) = \frac{1}{\beta}$$

and the integrals over  $\alpha$  and  $\beta$  may now be separated.

Using (15.2) and (15.5) the prior of weights becomes<sup>16</sup>:

$$p(W) = \int_0^\infty p(W|\alpha) p(\alpha) d\alpha = \int_0^\infty \frac{1}{Z_W(\alpha)} \exp(-\alpha E_W) \frac{1}{\alpha} d\alpha$$

<sup>15.5</sup> See [Bis95] pp. 415–417.

<sup>16</sup> See also mathematical appendix regarding Euler functions.

$$= \frac{1}{(2\pi)^{N_W/2}} \int_0^\infty \exp(-\alpha E_W) \alpha^{\frac{N_W}{2}-1} d\alpha = \frac{\Gamma_E(N_W/2)}{(2\pi E_W)^{N_W/2}}$$

where  $\Gamma_E$  is the Euler function.

The  $p(T|W)$  distribution is calculated in the same way, using (15.7) and (15.11):

$$p(T|W) = \frac{\Gamma_E(P/2)}{(2\pi E_T)^{P/2}}$$

From the above equations, the negative logarithm of the posterior distribution of weights is:

$$-\ln p(W|T) = \frac{P}{2} \ln E_T + \frac{N_W}{2} \ln E_W + \text{const.}$$

and then its gradient may be written as:

$$-\nabla \ln p(W|T) = \beta_c \nabla E_T + \alpha_c \nabla E_W \quad (15.32)$$

❖  $\alpha_c, \beta_c$

where

$$\alpha_c = \frac{N_W}{2E_W} \quad \text{and} \quad \beta_c = \frac{P}{2E_D} \quad (15.33)$$

are the *current* values of the parameters.

The minima of  $-\ln p(W|T)$  may be found by iteratively using (15.32) and (15.33)

### Remarks:

- While the direct integration over  $\alpha$  and  $\beta$  seems to be better than the evidence approximation (see section 15.4), in practice the approximations required after integration may give worst results.

## 15.6 Model Comparison

Let consider that there are several models  $\{\mathcal{M}_m\}$  for the same problem and set of data  $T$ .

❖  $P(\mathcal{M}_m)$ ,  
 $p(T|\mathcal{M}_m)$

The posterior probability of a particular model  $\mathcal{M}_m$  is given by the Bayes theorem

$$P(\mathcal{M}_m|T) = \frac{p(T|\mathcal{M}_m) P(\mathcal{M}_m)}{p(T)}$$

$\mathcal{M}_m$  evidence

where  $P(\mathcal{M}_m)$  is the prior probability for model  $\mathcal{M}_m$  and  $p(T|\mathcal{M}_m)$  is the evidence for  $\mathcal{M}_m$ .

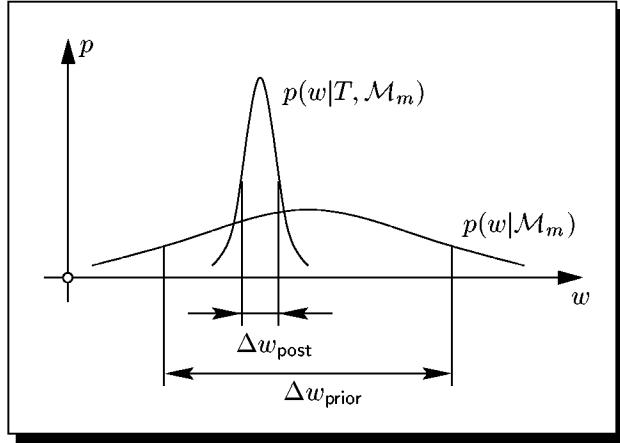
An interpretation of the model evidence may be given as follows below. Let consider first the weight dependency: in the Bayesian framework:

$$p(T|\mathcal{M}_m) = \int_W p(T|W, \mathcal{M}_m) p(W|\mathcal{M}_m) dW \quad (15.34)$$

❖  $\Delta w_{\text{prior}}, \Delta w_{\text{post}}$  and let consider one single weight: the prior distribution  $p(W|\mathcal{M}_m)$  have a low maximum

---

<sup>15.6</sup>See [Bis95] pp. 418–422.



**Figure 15.4:** The prior and posterior distribution of weights. The prior distribution  $p(W|M_m)$  have a (relatively) low maximum and a wide width  $\Delta w_{\text{prior}}$  — all weights have approximatively the same probability, denoting the absence of data on what to make a decision. The posterior probability density  $p(W|T, M_m)$  have a high maximum and a small width  $\Delta w_{\text{post}}$ .

and a large width  $\Delta w_{\text{prior}}$  (in fact it should be almost constant over a large range of weight values) while the posterior distribution  $p(W|T, M_m)$  have (usually) a high maximum and a small width  $\Delta w_{\text{post}}$ . See figure 15.4.

Considering a sharp peak of the posterior weight distribution around some maximum  $w^*$  the the integral (15.34) may be approximated as

$$p(T|M_m) \simeq p(T|w^*, M_m) p(w^*|M_m) \Delta w_{\text{post}}$$

and also the prior distribution (as is normated) should have a inverse dependency of its widening  $p(W|M_m) \propto 1/\Delta w_{\text{prior}}$  (the wider is the distribution, the smaller is the maximum and subsequently all values) and then:

$$p(T|M_m) \propto p(T|w^*, M_m) \frac{\Delta w_{\text{post}}}{\Delta w_{\text{prior}}}$$

which represents the product between the likelihood  $p(T|w^*, M_m)$ , estimated at the most probable weight value, and a term named the *Occam factor*. A model with a good fit will have a high likelihood, however these models are usually complex and consequently have a very high and narrow posterior distribution peak, i.e. a small Occam factor, and reciprocal. Also for different models which make the same predictions, i.e. have the same  $p(T|W)$  the Occam factor advantages the simpler model, i.e. the one with a larger factor.

Let consider the  $\alpha$  and  $\beta$  hyper-parameter dependency. The Bayesian framework require an integration over all possible values:

$$p(T|M_m) = \iint p(T|\alpha, \beta, M_m) p(\alpha, \beta|M_m) d\alpha d\beta \quad (15.35)$$

where  $p(T|\alpha, \beta, M_m)$  represents the evidence for  $\alpha$  and  $\beta$  — see section 15.4.

❖  $\Omega$

The  $\alpha$  and  $\beta$  parameters are considered statistically independent (see again section 15.4). By using the Gaussian approximation given by (15.31) and considering an uniform prior distribution  $p(\alpha) = p(\beta) = 1/\ln \Omega$ , where  $\Omega$  is a region containing  $\alpha^*$  respectively  $\beta^*$ , then the integral (15.35) may be split in the form:

$$\begin{aligned} p(T|\mathcal{M}_m) &= \frac{p(T|\alpha^*, \beta^*, \mathcal{M}_m)}{(\ln \Omega)^2} \int_{-\infty}^{\infty} \exp\left(-\frac{(\ln \alpha - \ln \alpha^*)^2}{2\sigma_{\ln \alpha}^2}\right) d(\ln \alpha) \times \\ &\quad \times \int_{-\infty}^{\infty} \exp\left(-\frac{(\ln \beta - \ln \beta^*)^2}{2\sigma_{\ln \beta}^2}\right) d(\ln \beta) = \\ &= p(T|\alpha^*, \beta^*, \mathcal{M}_m) \frac{2\pi\sigma_{\ln \alpha}\sigma_{\ln \beta}}{(\ln \Omega)^2} \end{aligned}$$

❖  $R$

The above result was obtained by integrating over a single Gaussian. However in networks with hidden neurons there is a symmetry and many equivalent maximums<sup>17</sup>, thus the model evidence  $p(T|\mathcal{M}_m)$  have to be multiplied by this redundancy factor  $R$ , e.g. for a 2-layer network with  $H$  hidden neurons there are  $R = 2^H H!$  equivalent maximums, and the model evidence have to be multiplied by this factor.

On similar grounds as for (15.24), and using (15.30), the logarithm of evidence becomes:

$$\begin{aligned} \ln p(T|\mathcal{M}_m) &= -\alpha^* E_W^* - \beta^* E_D^* - \frac{1}{2} \ln |H_S| + \frac{N_W}{2} \ln \alpha^* + \frac{P}{2} \ln \beta^* + \ln R \\ &\quad - \frac{1}{2} \ln \frac{\gamma}{2} - \frac{1}{2} \ln \frac{P - \gamma}{2} + \text{const.} \end{aligned}$$

where the additive constant is model independent.

By the above means it is possible to calculate the probabilities of various models. However there are several comments to be made:

- The model evidence is not particularly easy to calculate due to the Hessian  $|H_S|$ .
- Choosing the model with highest evidence is not necessary the best option as there may be several models with significant/comparable evidence.

## 15.7 Committee Of Networks

❖  $m_m$

Usually the error have several local, non-equivalent, minima, i.e. not due to the symmetry<sup>18</sup>.

The posterior probability density may be written as a sum of all posterior distributions corresponding to the local minima  $m_m$ :

$$p(W|T) = \sum_m p(W, m_m|T) = \sum_m p(W|m_m, T) P(m_m|T)$$

<sup>17</sup> See chapter “Multi Layer Neural Networks”, section “Weight-Space Symmetry”.

<sup>15.7</sup> See [Bis95] pp. 422–424.

<sup>18</sup> See footnote 17.

By using the above distribution decomposition, other parameters may be calculated by integration over the weight space  $W$ , e.g. the averaged output is:

$$\begin{aligned}\langle y \rangle &= \int_W y(\mathbf{x}, W) p(W|T) d\mathbf{W} = \sum_m P(m_m|T) \int_{W_m} y(\mathbf{x}, W) p(W|m_m, T) d\mathbf{W} \\ &= \sum_m P(m_m|T) \langle y_m \rangle\end{aligned}$$

where  $W_i$  is the portion of the weight space corresponding to the minima  $m_m$  and  $\langle y_m \rangle$  is the output average corresponding to  $m_m$ . The above formula shows an weighted average of the outputs corresponding to different local minima.

❖  $W_m, \langle y_m \rangle$

## 15.8 Monte Carlo Integration

The Bayesian techniques often require integration over a large number of weights, i.e. the computation of integrals of the form:

$$\mathcal{I} = \int_W F(W) p(W|T) d\mathbf{W} \quad (15.36)$$

where  $F(W)$  is some integrand. As the number of weights is usually very big, the classical numerical methods of integration leads to a large computational task.

One way to approximate the above type of integrals is to use *Monte Carlo method*, i.e. to select a sample set of weight vectors  $\{\mathbf{W}_i\}_{i=1,L}$  from the distribution  $p(W|T)$  (i.e. the weights are randomly chosen such that their distribution equals  $p(W|T)$ ) and then approximate the integral by a finite sum:

$$\mathcal{I} \simeq \frac{V_W}{L} \sum_{i=1}^L F(\mathbf{W}_i)$$

where  $V_W$  is the volume of weight space and  $V_W/L$  replaces (approximate)  $d\mathbf{W}$ .

While usually the posterior distribution  $p(W|T)$  may be calculated relatively with ease, the selection of sample set  $\{\mathbf{W}_i\}$  may be difficult. An alternative is to draw the sample weight set from another distribution  $q(W)$ , in which case the integral (15.36) becomes:

$$\mathcal{I} = \int_W F(W) \frac{p(W|T)}{q(W)} q(W) d\mathbf{W} \simeq \frac{V_W}{L} \sum_{i=1}^L F(\mathbf{W}_i) \frac{p(\mathbf{W}_i|T)}{q(\mathbf{W}_i)}$$

As the normalization of  $p(W|T)$  requires itself an integration of the type (15.36) with  $F(W) = 1$  (e.g. see the calculation of  $Z_W, Z_T$  in previous sections) then the integral may be approximated by using the non-normalized distribution  $\tilde{p}(W|T)$ :

importance sampling  
❖  $\tilde{p}(W|T)$

<sup>15.8</sup>See [Bis95] pp. 425–429.

$$\mathcal{I} \simeq \frac{\sum_{i=1}^L F(W_i) \frac{\tilde{p}(W_i|T)}{q(W_i)}}{\sum_{i=1}^L \frac{\tilde{p}(W_i|T)}{q(W_i)}} \quad (15.37)$$

this procedure being called *importance sampling*.

The importance sampling method still have one problem requiring attention. In practice the posterior distribution is usually almost zero in all weight space except some narrow areas (see section 15.1.3 and figure 15.2 on page 278). In order to ensure the computation of integral (15.37) with enough precision it is necessary to choose an  $L$  big enough such that the areas with significant posterior distribution  $p(W|T)$  will have adequate coverage.

Metropolis  
algorithm

To avoid the previous problem, the *Metropolis algorithm* was developed. The weight vectors in the sample set  $\{W_i\}$  form a discrete time series in the form:

$$\mathbf{W}_{(t+1)} = \mathbf{W}_{(t)} + \varepsilon$$

random walking  
where  $\varepsilon$  is a randomly chosen vector from a distribution (e.g. Gaussian) with spherical symmetry; this kind of series being named *random walking*. Then the new  $\mathbf{W}_{(t+1)}$  are accepted or rejected following the rules:

$$\begin{cases} \text{accept} & \text{if } p(W_{(t+1)}|T) > p(W_{(t)}|T) \\ \text{accept with probability } \frac{p(W_{(t+1)}|T)}{p(W_{(t)}|T)} & \text{if } p(W_{(t+1)}|T) > p(W_{(t)}|T) \end{cases}$$

and considering an error function of the form  $E = -\ln p(W|T)$ , then the above rules may be rewritten in the form:

$$\begin{cases} \text{accept} & \text{if } E_{(t+1)} < E_{(t)} \\ \text{accept with probability } \exp[-(E_{(t+1)} - E_{(t)})] & \text{if } E_{(t+1)} > E_{(t)} \end{cases} \quad (15.38)$$

$\diamond p(\varepsilon)$

The Metropolis algorithm still leaves a problem with respect to local minima. Assuming that the weights are strongly correlated this means that around (local) minima the hypersurfaces corresponding to constant distribution  $p(W|T) = \text{const.}$  are highly elongated hyperellipses<sup>19</sup>. As the distribution of  $\varepsilon$ , i.e.  $p(\varepsilon)$ , have spherical symmetry this will lead, according to the rules (15.38), to many rejected  $\mathbf{W}$  and the algorithm have a tendency to slow down around the local minima. See figure 15.5 on the next page.

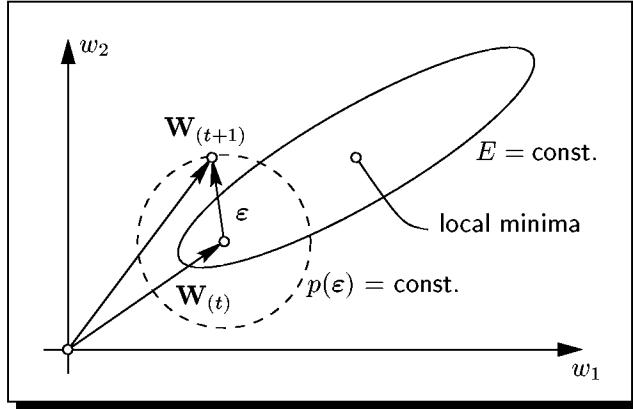
To correct the problem introduced by the Metropolis algorithm the rules (15.38) may be changed to:

$$\begin{cases} \text{accept} & \text{if } E_{(t+1)} < E_{(t)} \\ \text{accept with probability } \exp\left[-\frac{E_{(t+1)} - E_{(t)}}{T_{(t+1)}}\right] & \text{if } E_{(t+1)} > E_{(t)} \end{cases}$$

simulated  
annealing  
 $\diamond T$

leading to the algorithm named *simulated annealing*. The  $T_{(t+1)}$  — named “temperature” — is chosen to have large starting value  $T_{(0)} \gg 1$  and decreasing in time, this way the algorithm jumping fast over local minima found near the starting point. For  $T = 1$ , the simulated annealing is equivalent to the Metropolis algorithm.

<sup>19</sup>See chapter “Parameter Optimization”, section “Local quadratic approximation”



**Figure 15.5:** The Metropolis algorithm. Only the  $\mathbf{W}_{(t+1)}$  which falls in the “area” which may be represented (sort of) by the intersection of the dotted circle ( $p(\varepsilon) = \text{const.}$ ) and the ellipse  $E = \text{const.}$  are certainly accepted. Only some of the other  $\mathbf{W}_{(t+1)}$  — from the rest of the circled area — are accepted. As the (hyper) volumes of the two areas (of certainly acceptance, respectively partial acceptance) are proportional with the weight space dimensionality the algorithm slows more around local minima in the highly dimensional spaces, i.e. the problem worsens with the increase of weights number.

## 15.9 Minimum Description Length

Let consider that a “sender” wants to transmit some data  $D$  to a “receiver” such that the message have the shortest length possible. Beyond of the simple method of sending the data itself; if the quantity of data is sufficiently large then there is a possibility to shorten the message by sending a model  $\mathcal{M}$  of the data plus some information regarding the difference between the *actual* data set and the *generated* (by the model) data set. In this situation the message length will be the sum between the length of the model description  $L(\mathcal{M})$  and the length of the difference  $L(D|\mathcal{M})$ , which is model dependent:

$$\text{message length} = L(\mathcal{M}) + L(D|\mathcal{M})$$

❖  $D$

❖  $L(\mathcal{M})$ ,  
 $L(D|\mathcal{M})$

The  $L(\mathcal{M})$  quantity may also be seen as a measure of model complexity (as the more complex a model is the bigger its “description” is) and the  $L(D|\mathcal{M})$  may also be seen as the error of model (difference between model output and actual data target). The the message length may be written as:

$$\text{message length} = \text{model complexity} + \text{error} \quad (15.39)$$

The more complex the model is, i.e.  $L(\mathcal{M})$  is bigger, the more accurate its predictions are, and thus the error  $L(D|\mathcal{M})$  is small. Reciprocally a simple model, i.e.  $L(\mathcal{M})$  small, will

<sup>15.9</sup> See [Bis95] pp. 429–431.

generate many errors, leading to a large  $L(D|\mathcal{M})$ . This reasoning involves that there should be an optimum balance (tradeoff) between the two, resulting in a minimal message length.

For some variable  $x$  the information needed to be transmitted is  $-\ln p(x)$  where  $p(x)$  is the probability density<sup>20</sup>. Then:

$$\text{message length} = -\ln p(\mathcal{M}) - \ln p(D|\mathcal{M})$$

and by using the Bayes theorem:  $p(\mathcal{M}|D) = \frac{p(D|\mathcal{M})p(\mathcal{M})}{p(D)}$  (where  $p(\mathcal{M}|D)$  is the probability density for the model  $\mathcal{M}$ , given the data  $D$ ) it becomes:

$$\text{message length} = -\ln p(\mathcal{M}|D) - \ln p(D)$$

Let consider that the model  $\mathcal{M}$  represents a neural network. Then the message length becomes the length of the weight vector and data, given the specified model,  $L(W, D|\mathcal{M})$ . The model complexity is measured by the probability of the weight vector given the model  $-\ln p(W|\mathcal{M})$  and the error is calculated given the weight vector and the model  $-\ln p(D|W, \mathcal{M})$ . The equation (15.39) becomes:

$$L(W, D|\mathcal{M}) = -\ln p(W|\mathcal{M}) - \ln p(D|W, \mathcal{M}) \quad (15.40)$$

To transmit the distributions, both the sender and receiver must agree upon the general form of the distributions. Let consider the weight distribution as a Gaussian with zero mean and  $1/\alpha$  variance:

$$p(W|\mathcal{M}) = \left( \frac{\alpha}{2\pi} \right)^{\frac{N_W}{2}} \exp \left( -\frac{\alpha}{2} \|\mathbf{W}\|^2 \right)$$

and the error distribution as a Gaussian centered around data to be transmitted. Assuming one output for network  $y$  and  $P$  number of targets  $\{t_p\}$  to be transmitted then:

$$p(D|W, \mathcal{M}) = \left( \frac{\beta}{2\pi} \right)^{\frac{P}{2}} \exp \left[ -\frac{\beta}{2} \sum_{p=1}^P [y(\mathbf{x}_p) - t_p]^2 \right]$$

The message length becomes the sum-of-squares error function with the weight decay regularization factor:

$$L(D|\mathcal{M}) = \frac{\beta}{2} \sum_{p=1}^P [y(\mathbf{x}_p) - t_p]^2 + \frac{\alpha}{2} \|\mathbf{W}\|^2$$

## 15.10 Performance Of Models

### 15.10.1 Risk Averaging

Given a input vector  $\mathbf{x}$ , a classifier  $\mathcal{M}$  will categorize it of class  $\mathcal{C}_k$  for which the posterior probability is maximum (according to the Bayes rule<sup>21</sup>)  $P(\mathcal{C}_k|\mathbf{x}) = \max_{\ell} P(\mathcal{C}_{\ell}|\mathbf{x})$ . Then the

---

<sup>20</sup>See chapter “Error Function”, section “Entropy”.

15.10.1 See [Rip96] pg. 68.

<sup>21</sup>See chapter “Pattern Recognition”.

probability of the model to make a correct classification equals to the probability of the class chosen (according to the Bayes rule and for given  $\mathbf{x}$ ) to be the correct one, i.e. its posterior probability:

$$P_{\text{correct}}(\mathbf{x}) = P(\mathcal{C}_k|\mathbf{x}) = \mathcal{E}\{\max_{\ell} P(\mathcal{C}_{\ell}|\mathbf{x})\}$$

(as the posterior probability is also  $W$ -dependent, and  $W$  have a distribution associated, the expected value of maximum was used).

The probability of misclassification is the complement of probability of correct classification:

$$P_{\text{mc}}(\mathbf{x}) = 1 - P_{\text{correct}}(\mathbf{x}) = 1 - \mathcal{E}\{\max_{\ell} P(\mathcal{C}_{\ell}|\mathbf{x})\}$$

The above formula for misclassification does not depend on correct classification of  $\mathbf{x}$  and then it may be used successfully in the situations where gathering raw data is easy but the number of classified patterns is low. This procedure is known as *risk averaging*.

As the probabilities are normated, i.e.  $\sum_{k=1}^K P(\mathcal{C}_k|\mathbf{x}) = 1$  then the worst value for  $\max_{\ell} P(\mathcal{C}_{\ell}|\mathbf{x})$  is the one for which  $P(\mathcal{C}_k|\mathbf{x}) = 1/K$ ,  $\forall k = \overline{1, K}$ , and:

$$\left(\mathcal{E}\{1 - \max_{\ell} P(\mathcal{C}_{\ell}|\mathbf{x})\}\right)^2 \leq \left(1 - \frac{1}{K}\right) \mathcal{E}\{1 - \max_{\ell} P(\mathcal{C}_{\ell}|\mathbf{x})\}$$

The variance<sup>22</sup> of  $\max_{\ell} P(\mathcal{C}_{\ell}|\mathbf{x})$  is:

$$\begin{aligned} \mathcal{V}\{\max_i P(\mathcal{C}_i|\mathbf{x})\} &= \mathcal{E}\{[(1 - \max_{\ell} P(\mathcal{C}_{\ell}|\mathbf{x})) - \mathcal{E}\{1 - \max_{\ell} P(\mathcal{C}_{\ell}|\mathbf{x})\}]^2\} \\ &= \mathcal{E}\{(1 - \max_{\ell} P(\mathcal{C}_{\ell}|\mathbf{x}))^2\} - \left[\mathcal{E}\{1 - \max_{\ell} P(\mathcal{C}_{\ell}|\mathbf{x})\}\right]^2 \\ &\leq \left(1 - \frac{1}{K}\right) P_{\text{mc}}(\mathbf{x}) - P_{\text{mc}}^2(\mathbf{x}) \end{aligned}$$



### Remarks:

- ➔ In the process of estimating the probability of misclassification is better to use the posterior class probability  $P(\mathcal{C}_k|\mathbf{x})$  given by the Bayesian inference, rather than the one given the most probable  $W$  set of parameters, because it takes into account the variability of  $W$  and gives superior results especially for small probabilities.

---

<sup>22</sup>Variance of a random variable  $x$  being defined as  $\mathcal{V}\{(x - \mathcal{E}\{x\})^2\}$ .



## CHAPTER 16

# Tree Based Classifiers

### ► 16.1 Tree Classifiers

The decision trees are usually built from top to bottom. At each (nonterminal) node a decision is made till a terminal node, named also *leaf*, is reached. Each leaf should contain a class label, each nonterminal node should contain a decisional question. See figure 16.1 on the following page.

The main problem is to build the tree classifier using a training set (obviously having a limited size). This problem may be complicated by the overlapping between class areas, in which case there is noise present.

The tree is build by transforming a leaf into a decision making node and growing the tree further down — this process being named *splitting*. In the presence of noise the resulting tree may be overfitted (on the training set) so some pruning may be required.

splitting



#### Remarks:

- ➔ As the pattern space is separated into a decision areas (by the decision boundaries) the tree based classifiers may be seen as a hierarchical way of describing the partition of input space.
- ➔ Usually there are many possibilities to build a tree structured classifier for the same classification problem so an exhaustive search for the best one is not possible.

---

<sup>16.1</sup>See [Rip96] pp. 213–216.

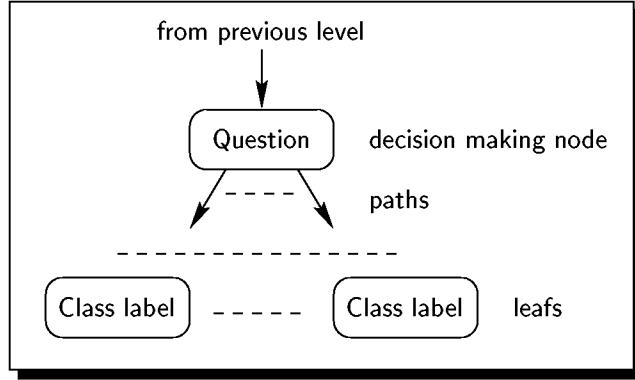


Figure 16.1: The tree based classifier.

## ► 16.2 Splitting

In general the tree is build considering one feature (i.e. component of the input vector) at a time. For binary features the choice is obvious but for continuous ones the problem is more difficult, especially if a small subset of features may greatly simplify the emerging tree.

### 16.2.1 Impurity based method

One method of deciding over the splitting method (feature selection and decision boundaries among features) is to increase “purity”, i.e. the pattern vectors who passes trough new build path should be, with greater probability, from some class(es) (rather than other). Alternatively the target is to decrease “impurity” which is easier to define in quantitative terms.

❖  $i(n)$

Impurity  $i(n)$  at the output of node  $n$  should be defined such that is zero if all  $P(\mathcal{C}_k|\mathbf{x})$  are zero except one (which will have the value 1, due to normalization) and be maximum if all  $P(\mathcal{C}_k|\mathbf{x})$  are equal. Two definitions of impurity are widely used — the probabilities refer to the current node  $n$ :

- Entropy:  $i(n) = - \sum_{k=1}^K P(\mathcal{C}_k|\mathbf{x}) \ln P(\mathcal{C}_k|\mathbf{x})$ .

Because  $\lim_{P \rightarrow 0} P \ln P = 0$  (by L'Hospital rule),  $\ln 1 = 0$  and  $P \leq 1 \Rightarrow \ln P \leq 0$ , then the defining conditions for impurity are met.

Gini index

- Gini index:  $i(n) = \sum_{\substack{k, \ell=1 \\ \ell \neq k}}^K P(\mathcal{C}_k|\mathbf{x}) P(\mathcal{C}_\ell|\mathbf{x}) = 1 - \sum_{k=1}^K P^2(\mathcal{C}_k|\mathbf{x})$

(last equality derived from normalization condition, squared, i.e.  $\left( \sum_{k=1}^K P(\mathcal{C}_k|\mathbf{x}) \right)^2 = 1$ ).

---

<sup>16.2</sup>See [Rip96] pp. 216–221.

The average decrease in impurity after splitting by feature  $x$  is:

$$-\Delta i(n) = \int_X p(x) i(n) dx$$

such that usually the algorithm building the tree classifier will try to choose that feature who maximize the above expression.

The average impurity of the whole tree may be defined as

$$i_{\text{tree}} = \sum_{k=1}^K q_k i(k) \quad (16.1)$$

where  $q_k$  is the probability of a pattern vector reaching leaf  $k$  (assuming that the final tree have a leaf for each class).  $\diamond q_k$

### 16.2.2 Deviance based method

Another approach to the building of the decisional tree is to consider it as a probabilistic model. Each leaf  $k$  may be associated with a distribution which show the probability that a pattern reaching the node is of some particular class, i.e.  $P(C_\ell, k)$ .

Considering  $P(n)$  the probability of a pattern to reach node  $n$  then the conditional probability density  $p(C_\ell|n)$  is:

$$P(C_\ell, n) = P(C_\ell|n) P(n) \Rightarrow P(C_\ell|n) = \frac{P(C_\ell, n)}{P(n)} \quad (16.2)$$

Also, taking the number of patterns (from the training set), arriving at leaf  $k$  and of class  $C_\ell$ , as being  $P_{k\ell}$ , then the likelihood of the training set is:  $\diamond P_{k\ell}$

$$\mathcal{L} = \prod_{k=1}^K \prod_{\ell=1}^K [P(C_\ell|k)]^{P_{k\ell}}$$

The deviance<sup>1</sup> is:

$$D_{\text{tree}} = 2(\ln \mathcal{L})_{\text{for perfect model}} - 2 \ln \mathcal{L} = \sum_{k=1}^K D_k \quad \text{where} \quad D_k = -2 \sum_{\ell=1}^K P_{k\ell} \ln P(C_\ell|k) \quad (16.3)$$

because for the perfect model  $p(C_\ell|k) = 1$  for  $P_{k\ell} > 0$  and equals zero otherwise (and  $\lim_{x \rightarrow 0} x \ln x = 0$ ) and thus the deviance term associated with the perfect model cancels.

If the total number of patterns arriving at leaf  $k$  is  $P_k$  then an estimate of  $p(C_\ell|k)$  would be  $\hat{p}(C_\ell|k) = P_{k\ell}/P_k$  (also from (16.2):  $p(C_\ell, k) \propto N_{k\ell}$ ,  $p(k) \propto N_k$ ), note also that the training set is assumed to be a unbiased sample from the true distribution). From (16.3),

---

<sup>1</sup>See chapter "Pattern Recognition" for the definition.

and using  $\sum_{k=1}^K P_{k\ell} = P_k$ :

$$D_{\text{tree}} = 2 \left( \sum_{k=1}^K P_k \ln P_k - \sum_{\ell,k=1}^K P_{k\ell} \ln P_{k\ell} \right)$$

Considering the tree impurity (16.1) then  $q_k$  would be  $q_k = P_k/P$  and for the entropy impurity:

$$i(k) = - \sum_{\ell=1}^K \frac{P_{k\ell}}{P_k} \ln \frac{P_{k\ell}}{P_k} \Rightarrow i_{\text{tree}} = - \sum_{k,\ell=1}^K \frac{P_k}{P} \frac{P_{k\ell}}{P_k} \ln \frac{P_{k\ell}}{P_k} = \frac{D_{\text{tree}}}{2P}$$

and so the entropy and deviance based splitting methods are equivalent.

### Remarks:

- ➔ In practice it may well happen that the training set is biased, e.g. it contains a larger number of examples from rare classes than it would have been occurred in a random sample. In this case the probabilities  $P(\mathcal{C}_\ell|k)$  should be estimated separately. One way to do this is to attach “weights” to the training patterns and consider  $P_k$ ,  $P_{k\ell}$  and  $P$  as the sum of “weights” rather than actual count of patterns.
- ➔ If there are “costs” associated with misclassification then these may be inserted directly into the Gini index in the form:

$$i(n) = \sum_{\substack{k,\ell=1 \\ k \neq \ell}}^K C_{k\ell} P(\mathcal{C}_k|\mathbf{x}) P(\mathcal{C}_\ell|\mathbf{x})$$

where  $C_{k\ell}$  is the cost for misclassification between classes  $\mathcal{C}_k$  and  $\mathcal{C}_\ell$ . Note that this leads to completely symmetrical “costs” as the total coefficient of  $P(\mathcal{C}_k|\mathbf{x}) P(\mathcal{C}_\ell|\mathbf{x})$  (in the above sum) is  $C_{k\ell} + C_{\ell k}$ . Thus this approach is ineffective for two class problems.

## ► 16.3 Pruning

❖  $T$ ,  $R(T)$ ,  $S(T)$

Let  $R(T)$  be a measure of the tree  $T$  such that the better  $T$  is the smaller  $R(T)$  becomes and such that it have contributions from (and only from) all (its) leaves. Possible choices are the total number of misclassification over a training/test set or the entropy or deviance. Let  $S(T)$  be the size of the tree  $T$  proportional to the number of leaves.

❖  $\alpha$ ,  $R_\alpha(T)$

A good criterion for characterizing the tree is:

$$R_\alpha(T) = R(T) + \alpha S(T) \tag{16.4}$$

which is minimal for a good one.  $\alpha$  is a positive parameter that penalizes the tree size; for  $\alpha = 0$  the tree is chosen based only on error rate.

❖  $T(\alpha)$

For a given  $R_\alpha(T)$  there are several possible trees, let  $T(\alpha)$  be the optimal one.

<sup>16.3</sup>See [Rip96] pp. 221–225.

Let consider a tree  $T$  and for a non-leaf node  $n$  the subtree  $T_n$  having as root the node  $n$ .  $\diamond T_n$

Let  $g(n, T)$  a measure of reduction in  $R$  by adding  $T_n$  to node  $n$ , relative to the increase in size:

$$g(n, T) = \frac{R(n) - R(T_n)}{S(T_n) - S(n)} \quad (16.5)$$

$S(n)$  is the size corresponding to one node  $n$ , it is assumed that  $S(T_n)$  represents a subtree with at least two nodes (it doesn't make sense to add just one leaf to another one) and thus  $S(T_n) \geq S(n)$ .  $R(n)$  is the measure  $R$  of node  $n$ , considering it a leaf.

From (16.4):  $R_\alpha(n) = R(n) + \alpha S(n)$  and  $R_\alpha(T_n) = R(T_n) + \alpha S(T_n)$  and using (16.5) then  $g(n, T)$  may be written as:

$$g(n, T) = \alpha + \frac{R_\alpha(n) - R_\alpha(T_n)}{S(T_n) - S(n)} \quad (16.6)$$

As the denominator is always positive then:

$$g(n, T) > \alpha \Leftrightarrow R_\alpha(n) > R_\alpha(T_n) \quad (16.7)$$

**Proposition 16.3.1.** *Let consider a tree  $T$  and number its nodes from bottom up such that each child node have a number label smaller than its parent node. Let visit each node in its number order (i.e. from bottom up) and prune at the current node  $n$  if  $R_\alpha(n) \leq R_\alpha(T'_n)$  where  $T'$  is the current tree. After visiting all nodes the result is  $T(\alpha)$ .*  $\diamond T'$

*Proof.* It is demonstrated by induction.

For the unpruned tree  $T$  all leaves are optimally pruned.

Let consider a current node  $n$ . This one is either pruned with the value  $R_\alpha(n)$  or is not, in which case

$$R_\alpha(T'_n) = \sum_{\text{branches } B} R_\alpha(T'_B) < R_\alpha(n)$$

the sum being done over all branches  $B$  of node  $n$ .

But if it is not pruned then it's not possible to have a tree  $T''_n$  with a smaller  $R_\alpha$  because in this case it will be (at least) one branch  $B$  such that  $R_\alpha(T''_B) < R_\alpha(T'_B)$  and thus  $T'_n$  wouldn't be optimally pruned; i.e. if the tree is not pruned at node  $n$  then the whole subtree (from node  $n$  downwards) is optimally pruned.

After analyzing the last node, which (according to the numbering scheme) is the root of whole tree, the tree is optimally pruned.  $\square$

**Proposition 16.3.2.** *Let  $\alpha_1$  be the smallest value of  $g(n, T)$  for any non-leaf node of  $T$ . The optimally pruned tree is either  $T$  for  $\alpha < \alpha_1$  or  $T(\alpha_1)$  obtained by pruning all nodes with  $g(n, T) = \alpha_1$ .*  $\diamond \alpha_1$

*Proof.*  $\alpha_1$  is chosen such that  $\alpha_1 = \min_n g(n, T) \leq g(n, T), \forall n$  non-leaf node.

Let consider the first case when  $\alpha < \alpha_1$ . But then  $\alpha < g(n, T)$  for all non-leaf  $n$  and thus from (16.7) it follows that  $R_\alpha(n) > R_\alpha(T_n)$  for all non-leaf nodes and according to the previous proposition no pruning is performed and the tree is  $T(\alpha)$ .

Considering the second case, after pruning all nodes with  $g(n, T) = \alpha_1 \equiv \min_n g(n, T)$ , for all non-terminal nodes left in the tree:  $g(n, T) > \alpha_1$ . This means that, according to (16.7),  $R_{\alpha_1}(n) > R_{\alpha_1}(T_n)$  and, by using previous proposition, no node pruning takes place and the tree is  $T(\alpha_1)$ .  $\square$

**Proposition 16.3.3.** *For  $\beta > \alpha$ ,  $T(\beta)$  is a subtree of  $T(\alpha)$*

*Proof.* It will be shown by induction that  $T_n(\beta)$  is a subtree of  $T_n(\alpha)$  for any node  $n$  and thus for the root node as well.

The proposition is true for all terminal nodes (leafs).

It have to be shown that if  $R_\alpha(n) \leq R_\alpha(T_n)$  is true then  $R_\beta(n) \leq R_\beta(T_n)$  is also true and thus when pruning by the method described in the first proposition (above) then  $T(\alpha)$  will contain  $T(\beta)$ .

From (16.4):

$$\begin{cases} R_\alpha(n) = R(n) + \alpha S(n) \\ R_\beta(n) = R(n) + \beta S(n) \end{cases} \Rightarrow R_\beta(n) = R_\alpha(n) + (\beta - \alpha) S(n)$$

and also (the same way)

$$\begin{cases} R_\alpha(T_n) = R(T_n) + \alpha S(T_n) \\ R_\beta(T_n) = R(T_n) + \beta S(T_n) \end{cases} \Rightarrow R_\beta(T_n) = R_\alpha(T_n) + (\beta - \alpha) S(T_n)$$

and by subtracting the above two equations:

$$R_\beta(n) - R_\beta(T_n) = [R_\alpha(n) - R_\alpha(T_n)] + (\beta - \alpha)[S(n) - S(T_n)] \quad (16.8)$$

Considering  $R_\alpha(n) \leq R_\alpha(T_n)$  and as  $\beta > \alpha$  and  $S(n) < S(T_n)$  then  $R_\beta(n) - R_\beta(T_n) \leq 0$ .  $\square$

The last two propositions show the following:

- There are a series of parameters  $\alpha_1 < \alpha_2 \dots$  found by ordering all  $g(n, T)$ . For each  $(\alpha_{i-1}, \alpha_i]$  there is only one optimal tree  $T(\alpha_i)$ .
- For  $j > i$  the  $T(\alpha_j)$  is embedded in  $T(\alpha_i)$  (as  $\alpha_j > \alpha_i$  and by applying the last proposition).

## 16.4 Missing Data

One of the advantages of the tree classifiers is the ease by which the missing data may be treated.

The alternatives when data is partially missing from a pattern vector are:

- *“Drop”*. Work with the available data “pushing” the pattern down the tree as far as possible and then use the distribution at the reached node to make a prediction (if not a leaf, of course).
- *Surrogate splits*. Create a set of surrogate split rules at non-terminal nodes, to be used if real data is missing.
- *Missing feature*. Consider the “missing” as a possible/supplemental value for the feature and create a separate branch/split/sub-tree for it.
- *Split examples*. When an pattern vector (from the training set) reaches a node where a split should occur over its missing features then it is split in fractions over the branches nodes. Theoretically this should be done using a posterior probability conditioned on available data but this is seldom available. However it is possible to use the probability of going along (the node’s) branches using the (previous) complete (without missing data) pattern vectors.

---

<sup>16.4</sup>See [Rip96] pp. 231–233.

## CHAPTER 17

# Belief Networks

### 17.1 Graphs

**Definition 17.1.1.** A **graph** is a collection of vertices and edges.

graph

The **vertices** represent (in the context of belief networks) a set of random variables (each drawn from some distribution).

vertices

A **edge** represent a pair of distinct vertices.

edge  
(un)directed  
graph

If the pair is ordered then the graph is **directed**, otherwise the graph is **undirected**. For the ordered edges, the first vertex is named *parent* and the second *child*.

The graphs are represented visually by using nodes for vertices and connecting lines for edges, ordered edges are shown using arrows. See figure 17.1 on the next page.

path  
cycle

**Definition 17.1.2.** A **path** is a list of vertices each of them being connected trough an edge. A **cycle** is a path which ends on the same vertex where it started and do not go trough a vertex more than once.

A **subgraph** is a subset of vertices (from the same graph) together with all edges connecting them.

subgraph

A (sub)graph is **connected** if there is a path for every possible pair of vertices.

connected graph

A (sub)graph is **complete** if every possible edge is present.

complete graph

A maximal complete subgraph (of a graph) is named **clique**.

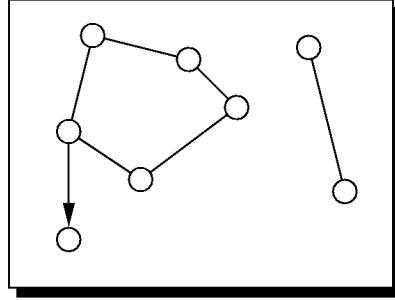
clique

**Definition 17.1.3.** A **tree** is a connected graph with no cycles. A directed tree is a connected directed acyced graph, abbreviated *DAG*.

tree  
DAG

A directed tree have the property that there is a vertex, named *root*, such that there is a

<sup>17.1</sup>See [Rip96] pp. 243–249.



**Figure 17.1:** A graph. The one represented here is unconnected, have a cycle and one ordered edge (shown by the arrow).

root *directed path from the root to any other vertex and any vertex except root have exactly one incoming edge (arrow), i.e. have just one parent.*

ancestral subgraph *An **ancestral subgraph** of a directed graph contains all the parents of its vertices, i.e. it contain also the root (which have no parent).*

polytree *A **polytree** is a singly connected graph, i.e. there is only one path between any two vertices.*

### 17.1.1 Markov Properties

❖  $\mathcal{G}, V$

**Definition 17.1.5.** Let consider 3 subsets of vertices:  $A, B, C \subset V$ , where  $V$  is the whole set of vertices of graph  $\mathcal{G}$ . Then  $C$  **separate**  $A$  and  $B$  in  $\mathcal{G}$  if any path from any vertex in  $A$  to any vertex in  $B$  have to pass trough a vertex from  $C$ .

❖  $\perp\!\!\!\perp$

Let  $\mathbf{x}_A$  be the set of (random) variables associated with  $A$  and similarly  $\mathbf{x}_B, \mathbf{x}_C$ . Then the conditional independence of  $\mathbf{x}_A$  and  $\mathbf{x}_B$ , given  $\mathbf{x}_C$ , is written as:

$$\mathbf{x}_A \perp\!\!\!\perp \mathbf{x}_B | \mathbf{x}_C \quad \text{or (in short)} \quad A \perp\!\!\!\perp B | C$$

❖  $A^C, \partial A$   
boundary

**Definition 17.1.6.** Let  $A^C$  be the complementary set of  $A$  vertices, i.e.  $A^C = V \setminus A$ . The **boundary** of  $A$ , notated  $\partial A$  is the set of all vertices from  $A^C$  who are connected to a vertex in  $A$  trough an edge.

Markov properties

**Definition 17.1.7.** The following **Markov properties** are defined:

1. **Global:** if, for any subsets  $A, B$  and  $C$  of vertices, it is true that  $\mathbf{x}_A \perp\!\!\!\perp \mathbf{x}_B | \mathbf{x}_C$ .
2. **Local:** if, for some subset  $A$  the conditional distribution of  $\mathbf{x}_A$ , given  $\mathbf{x}_{V \setminus \{A \cup \partial A\}}$  depends only on  $\mathbf{x}_{\partial A}$ , i.e.  $\mathbf{x}_A \perp\!\!\!\perp \mathbf{x}_{V \setminus \{A \cup \partial A\}} | \mathbf{x}_{\partial A}$ . Otherwise said, the  $\mathbf{x}_A$  variables and those not directly connected to them, are conditionally independent.
3. **Pairwise:** if, for some subsets  $A$  and  $B$ ,  $\mathbf{x}_A$  and  $\mathbf{x}_B$  are conditionally independent given all other (stochastic) variables and there is no edge from  $A$  to  $B$ .

❖  $\varphi_C(\mathbf{x}_C)$

**Proposition 17.1.1.** 1. Let consider a set of (random) variables defined on the vertices of a graph  $\mathcal{G}$ . If the graph have the pairwise Markov property then there exists a set of positive functions  $\varphi_C(\mathbf{x}_C)$ , defined over the cliques of  $\mathcal{G}$ , symmetric in their arguments, such that

$$P(\mathbf{x}_V) \propto \prod_{\text{cliques } C} \varphi_C(\mathbf{x}_C) \tag{17.1}$$

<sup>17.1.1</sup>See [Rip96] pp. 249–252.

i.e. the probability of the graph random variables to have the values  $\mathbf{x}_V$  (all values are collected together into a vector) is proportional to the product (over its cliques) of functions  $\varphi_C$  (the values of components of  $\mathbf{x}_C$  are the same with the corresponding components of  $\mathbf{x}_V$ , vertex wise).

**2.** If  $P(\mathbf{x}_V)$  may be represented in the form (17.1) then the graph have the global Markov property.

*Proof.* 1. It is assumed that for any vertex  $s$  of  $\mathcal{G}$  the associated variable may take the value 0 (this can be done by some "re-labeling" procedure if it's not true initially).

It will be shown by induction over the size of an subgraph  $A = \{\text{vertex } s | x_s \neq 0\} \subset V_{\mathcal{G}}$ .

The desired probability is built in the form:

$$P(\mathbf{x}_V) = P(\mathbf{x}_V = \hat{0}) \prod_{C \subset A} \varphi_C(\mathbf{x}_C) \quad (17.2)$$

where  $\varphi_C$  is defined recursively as:

$$\varphi_C(\mathbf{x}_C) = \frac{P(\mathbf{x}_C, \mathbf{x}_{C^c} = \hat{0})}{P(\mathbf{x}_V = \hat{0}) \prod_{D \subsetneq C} \varphi_D(\mathbf{x}_D)} \quad (17.3)$$

and for the empty set  $D = \emptyset \Rightarrow \varphi_D(\hat{0}) \equiv 1$  (the sum over  $D$  being done for all strict subsets of  $C$ ). Also  $\varphi_C \equiv 1$  for  $C$  non-complete.

Now, for the cases when  $A = \emptyset$  and  $A$  contains just one vertex, equations (17.2) and (17.3) gives the identities  $P(\hat{0}) = P(\hat{0})$ , respectively  $P(\mathbf{x}_A) = P(\mathbf{x}_A, \mathbf{x}_{A^c} = \hat{0})$ . so (17.1) holds.

The (17.2) and (17.3) equations are condensed to:

$$P(\mathbf{x}_A) = P(\hat{0}) \prod_{C \subset A} \frac{P(\mathbf{x}_C, \mathbf{x}_{C^c} = \hat{0})}{P(\mathbf{x}_A = \hat{0}) \prod_{D \subsetneq C} \varphi_D(\mathbf{x}_D)}$$

If  $A$  is complete then any of its subgraphs is also part of one of its cliques, so the above equation may be written in the form (17.1). So the proposition also holds for  $A$  complete.

Let assume that (17.1) holds for non-complete  $A$  having  $i$  vertices and let study a new  $A$  with  $i+1$  vertices. As  $A$  is not complete then it is possible to write  $A = B \cup s \cup t$  where  $B$  is a subgraph of  $A$  having  $i-1$  vertices and  $s$  and  $t$  are not neighbors (there is no edge  $(s, t)$ ), i.e.  $x_s \perp\!\!\!\perp x_t | \mathbf{x}_B, \mathbf{x}_{A^c}$ . Also:

$$\begin{aligned} P(\mathbf{x}_V) &= P(\mathbf{x}_B, x_s, x_t, \mathbf{x}_{A^c} = \hat{0}) \\ &= P(\mathbf{x}_B, x_s, x_t = 0, \mathbf{x}_{A^c} = \hat{0}) \frac{P(x_t | \mathbf{x}_B, x_s, \mathbf{x}_{A^c} = \hat{0})}{P(x_t = 0 | \mathbf{x}_B, x_s, \mathbf{x}_{A^c} = \hat{0})} \end{aligned}$$

and considering the conditional independence of  $s$  and  $t$  then:

$$\begin{aligned} P(\mathbf{x}_V) &= P(\mathbf{x}_B, x_s, x_t = 0, \mathbf{x}_{A^c} = \hat{0}) \frac{P(x_t | \mathbf{x}_B, x_s = 0, \mathbf{x}_{A^c} = \hat{0})}{P(x_t = 0 | \mathbf{x}_B, x_s = 0, \mathbf{x}_{A^c} = \hat{0})} \\ &= P(\mathbf{x}_B, x_s, x_t = 0, \mathbf{x}_{A^c} = \hat{0}) \frac{P(x_t, \mathbf{x}_B, x_s = 0, \mathbf{x}_{A^c} = \hat{0})}{P(x_t = 0, \mathbf{x}_B, x_s = 0, \mathbf{x}_{A^c} = \hat{0})} \end{aligned}$$

By using (17.2) (supposed true, induction):

$$\begin{aligned} P(\mathbf{x}_B, x_s = 0, x_t = 0, \mathbf{x}_{A^c} = \hat{0}) &= P(\mathbf{x}_V = \hat{0}) \prod_{C \subset B} \varphi_C(\mathbf{x}_C) \quad \text{for } \mathbf{x}_{B^c} = \hat{0} \\ P(\mathbf{x}_B, x_s, x_t = 0, \mathbf{x}_{A^c} = \hat{0}) &= P(\mathbf{x}_V = \hat{0}) \prod_{C \subset \{B \cup s\}} \varphi_C(\mathbf{x}_C) \quad \text{for } \mathbf{x}_{\{B \cup s\}^c} = \hat{0} \\ P(\mathbf{x}_B, x_s = 0, x_t, \mathbf{x}_{A^c} = \hat{0}) &= P(\mathbf{x}_V = \hat{0}) \prod_{C \subset \{B \cup t\}} \varphi_C(\mathbf{x}_C) \quad \text{for } \mathbf{x}_{\{B \cup t\}^c} = \hat{0} \end{aligned}$$

and then:

$$P(\mathbf{x}_V) = P(\mathbf{x}_V = \hat{0}) \prod_{C \subset \{B \cup s\}} \varphi_C(\mathbf{x}_C) \frac{\prod_{C' \subset \{B \cup t\}} \varphi_{C'}(\mathbf{x}_{C'})}{\prod_{C'' \subset B} \varphi_{C''}(\mathbf{x}_{C''})}$$

As a complete subgraph  $C$  of  $A$  cannot contain both  $s$  and  $t$  vertices (because there is no path between them) then the  $P(\mathbf{x}_V)$  can be finally written as:

$$\Pr(\mathbf{x}_V) = \Pr(\mathbf{x}_V = \hat{0}) \prod_{C \subset \{B \cup s \cup t\}} \varphi_C(\mathbf{x}_C)$$

which shows that the  $A$  subgraph of size  $i+1$  may be written in the same form as the  $A$  subgraph of size  $i$ .

2. It is assumed that (17.1) is true. For an subgraph  $A \subset \mathcal{G}$ :

$$P(\mathbf{x}_V) = P(\mathbf{x}_A | \mathbf{x}_{A^c}) P(\mathbf{x}_{A^c}) \Rightarrow P(\mathbf{x}_A | \mathbf{x}_{A^c}) = \frac{P(\mathbf{x}_V)}{P(\mathbf{x}_{A^c})}$$

For  $P(\mathbf{x}_V)$  the formula (17.1) may be used directly, for  $P(\mathbf{x}_{A^c})$ , as only  $\mathbf{x}_{A^c}$  is fixed and  $\mathbf{x}_A$  may take any value, a sum over all  $\mathbf{x}_A$  values is to be performed:

$$P(\mathbf{x}_A | \mathbf{x}_{A^c}) = \frac{\prod_{\substack{\text{cliques } C \subset \mathcal{G} \\ C \cap A \neq \emptyset}} \varphi_C(\mathbf{x}_C)}{\sum_{\mathbf{x}_A} \prod_{\substack{\text{cliques } C \subset \mathcal{G} \\ C \cap A = \emptyset}} \varphi_C(\mathbf{x}_C)} = \frac{\prod_{\substack{\text{cliques } C \subset \mathcal{G} \\ C \cap A \neq \emptyset}} \varphi_C(\mathbf{x}_C)}{\sum_{\mathbf{x}_A} \prod_{\substack{\text{cliques } C \subset \mathcal{G} \\ C \cap A = \emptyset}} \varphi_C(\mathbf{x}_C)}$$

because, at the denominator, the terms corresponding to cliques  $C$  disjoint from  $A$  (i.e.  $C \cap A = \emptyset$ ) may be extracted as a common multiplicative factor (of the sum) and then canceled with the corresponding terms of the numerator.

$C$  is a clique and some of its vertices are in  $A$ . Then  $C \subset \{A \cup \partial A\}$  (assume by absurd that there is a vertex  $s \in A$  and  $s \in C$  and another one  $t \notin \{A \cup \partial A\}$  and  $t \in C$ ; as  $s, t \in C$  and  $C$  is complete then the edge  $(s, t)$  does exists, then  $t \in \{A \cup \partial A\}$ , contradiction). This means that the right-hand part of the above equation is in fact just a function of  $\mathbf{x}_{A \cup \partial A}$ , i.e.  $P(\mathbf{x}_A | \mathbf{x}_{A^c}) = P(\mathbf{x}_A | \mathbf{x}_{\partial A})$ .

Let be  $A, B, C$  such that  $C$  separate  $A$  and  $B$ . Let  $B'$  be the set of vertices which may be reached from  $B$  using a path not going through  $C$ , thus  $B \subset B'$ , and let  $D = \{B' \cup C\}^C$ . Then  $B'$ ,  $C$  and  $D$  are disjoint by construction (i.e.  $B' \cap C = \emptyset$ ,  $B' \cap D = \emptyset$  and  $C \cap D = \emptyset$ ).

As  $A$  is separated from  $B$  by  $C$ , while  $B'$  is not, then no vertex from  $A$  may be in either  $C$  or  $B'$  thus  $A \subset D$ .

By construction  $B' \cup C \cup D = V_G$  (and they are disjoint). Also as  $B'$  is formed by all vertices which may be connected through a path not passing through  $C$  then  $B'$  and  $D$  are separated by  $C$ , i.e.  $B' \perp\!\!\!\perp D | C$  and, as  $A \subset D$  and  $B \subset B'$  then  $A \perp\!\!\!\perp B | C$  and thus the global Markov property.  $\square$

### 17.1.2 Markov Trees

Considering a tree, there is a unique path between each node; an undirected tree may be transformed into a directed one by choosing any vertex as root and then ordering the edges as to have the same direction as the paths leading from root to other vertices.

chain

The simplest tree is the *chain*, where each vertex have just one parent and one child, and each vertex splits the graph in two conditionally independent subgraphs.

#### Remarks:

- ➔ Markov chains may be used in time series and conditional independence in this case may be expressed as “past and future are independent, given the present”.

<sup>17.1.2</sup>See [Rip96] pp. 252–254.

Let consider that, for directed chains, each vertex is labeled with a number such that for each  $i$  its parent is  $i - 1$ , the root having label 1. Then the probability of the whole tree is:

$$P(\mathbf{x}_V) = P(x_1) \prod_i P(x_i | x_{i-1}, i \neq 1)$$

(the root doesn't have a parent and thus its probability is unconditioned).

For directed trees the above formula is to be slightly modified to:

$$P(\mathbf{x}_V) = P(x_1) \prod_i P(x_i | x_j, j \text{ parent of } i)$$

Let consider a vertex  $t$  and its associated random variable  $x_t$ . Let consider that the parent (ancestor) of  $t$  is  $s$  and the associated  $x_s$  may take the values  $\mathbf{x}_s \in \{x_{s1}, \dots\}$ . Then the distribution of  $x_t$  is  $p_t$  expressed by the means of distribution  $p_s$  at vertex  $s$ :

$$p_t(x_t) = \sum_i p_s(x_{si}) P(x_t | x_{si})$$

(the sum being done over the possible values for  $x_s$ ). Thus, given the distribution of root, the other distributions may be calculated recursively, from root to leafs.

Let  $E_s^-$  be the events on the descendants of  $s$ ,  $E_s^+$  all other events and  $E_s = E_s^- \cup E_s^+$ . The distribution  $p_s(x_s) = P(x_s | E_s)$  is to be found (i.e. given the values for some random variables at the vertices of the graph the problem is to find the probability of a random variable taking a particular value for a given vertex). Then (using Bayes theorem):

$$P(x_s | E_s) = P(x_s | E_s^-, E_s^+) \propto P(E_s^- | x_s, E_s^+) P(x_s | E_s^+)$$

and, as  $E_s^- \perp\!\!\!\perp E_s^+ | x_s$  then  $P(E_s^- | x_s, E_s^+) = P(E_s^- | x_s)$  and:

$$P(x_s | E_s) \propto P(E_s^- | x_s) P(x_s | E_s^+)$$

The first term is:

$$P(E_s^- | x_s) = \begin{cases} 1 & \text{if } s \text{ have no children} \\ \prod_{\substack{\text{children } t \\ \text{of } s}} P(E_t^- | x_s) & \text{otherwise} \end{cases}$$

where  $P(E_t^- | x_s) = \sum_{x_t} P(E_t^- | x_t) P(x_t | x_s)$ .

For the other term,  $x_s$  is conditionally separated from  $E_s^+$  by its parent  $r$ :

$$P(x_s | E_s^+) = \sum_{x_r} P(x_s | x_r) P(x_r | E_s^+)$$

(the sum being done over all possible values for  $x_r$ ). The restriction over the possible values of  $x_r$  are given by  $E_s^+$  through  $E_r^+$  and  $E_q^-$  where  $q$  is any child of  $r$ , except  $s$ .

$$P(x_r | E_s^+) = P(x_r | E_r^+) \prod_q P(E_q^- | x_r)$$

and finally the  $p_s(x_s)$  may be calculated by using the above formulas recursively.

### 17.1.3 Decomposable Trees

triangulated graph **Definition 17.1.8.** A graph is named **triangulated** (or **chordal**) if every cycle of four vertices/edges or more have a chord, i.e. an edge connecting two non-consecutive vertices.

join tree **Definition 17.1.9.** A **join tree** is the tree associated with a graph, having its cliques as vertices. The vertices of the tree are connected in such a way that if all cliques containing one vertex of the graph are removed, the join tree remains connected.

Considering a join tree and two cliques containing the same vertex (of the graph) then all cliques sitting on the path, in the join tree, between the two contains the vertex.

For a triangulated graph, the join tree may be built according to the following procedure:

1. *Numbering the graph vertices:* Starting at any point, number the vertices of the graph by maximum cardinality search, i.e. number next the vertex with the maximum number of already numbered neighbors.
2. Starting with the highest numbered vertex, check if all its lower numbered neighbors are also neighbors between them. If they are then the original graph is triangulated; if they aren't then by adding missing edges the graph may be triangulated.
3. Identify all cliques and order them by the highest numbered vertex in the clique.
4. Build the join tree by connecting each clique with a predecessor which have the most common vertices.

The above building procedure ensures also that the (unique) path (in the join tree) from clique  $C_1$  to some  $C_i$  passes through cliques having increasing order numbers.

For  $i \geq 2$  let  $C_{j(i)}$  be the parent of  $C_i$  in the join tree. Let  $S_i = C_i \cap (C_1 \cup \dots \cup C_{i-1})$ . Then  $S_i \subset C_{j(i)}$ .

*Proof.* As  $C_{j(i)}$  is one of  $C_1, \dots, C_{i-1}$  then  $S_i$  contains all vertices of  $C_i \cap C_{j(i)}$ . There is not possible to have a vertex  $s$  such that  $s \in C_i \cap C_k$  and  $k \neq j(i)$  and  $s \notin C_{j(i)}$  because of the way the join tree is built (steps 3 and 4). Alternatively: as  $s \in C_i \cap C_k$  then  $s$  is contained by each clique in the path between  $C_i$  and  $C_k$  (by direct consequence of the definition, see above) and the path must go through  $C_{j(i)}$ , the parent of  $C_i$ , as is a path in a tree and by the way the clique have been numbered,  $C_k$  lies somewhere above  $C_i$  (on the way to root) or on another tree.  $\square$

❖  $H_i, R_i$  Let  $H_i = C_1 \cup \dots \cup C_{i-1}$  and  $R_i = C_i \setminus S_i$ . Then  $\partial R_i \cap H_i$  is a complete subgraph.

*Proof.* Let consider a vertex  $s$  from  $\partial R_i$ , then either  $s \in C_i$  or  $s \in S_i$ .

Now, let consider an  $s \in \partial R_i \cap H_i$ . From the previous reasoning, either  $s \in C_i \cap H_i = S_i$ , or  $s \in S_i \cap H_i \subset S_i$ . In either case  $s \in S_i \subset C_{j(i)}$  which is complete.  $\square$

**Proposition 17.1.2.**  $S_i$  separate  $R_i$  from  $H_i \setminus S_i$ , i.e.  $R_i \perp\!\!\!\perp (H_i \setminus S_i) | S_i$ .

*Proof.* Let consider a path  $\mathcal{P}$  from  $R_i$  to  $H_i$  which contains a vertex  $s \in R_j$  for a  $j > i$  and  $s \notin \bigcup_{k>j} R_k$  (it may happen that there is no  $k > j$ ).

Let  $r$  and  $t$  be two vertices before and after  $s$ , in the order of numbering given by the procedure of join tree building, such that  $r, t \notin R_j$  but they are on  $\mathcal{P}$  and  $r \in R_i$  and  $t \in H_i$  ( $\mathcal{P}$  starts somewhere in  $R_i$  and ends on  $H_i$ ).

By the way of selecting  $r$  and  $t$  and also from the numbering procedure,  $r, t \in \partial R_j$ . Being in the vicinity of  $R_j$  then  $r, t$  are either in  $S_j \subset H_j$  or in  $\partial C_j \subset H_j$ . Thus  $r, t \in \partial R_j \cap H_j$ . As  $\partial R_j \cap H_j$  is complete then the edge  $(r, t)$  does exists.

<sup>17.1.3</sup>See [Rip96] pp. 258–261.

If the  $(r, t)$  edge exists then  $r$  and  $t$  should be members of the same clique  $C_k$ . As  $r \in R_i$  then  $k \geq i$  and as  $r, t \in H_j$  then  $k < j$ .

If  $k > i$ , as  $H_k \subset H_j$ , then  $r, t \in C_k \cap H_k \subset C_\ell$ , where  $\ell > k$ . Repeat this procedure as necessary, considering  $C_\ell$  instead of  $C_k$ , till  $\ell = i$  and thus  $t \in C_i \cap H_i = S_i$ , i.e.  $S_i$  separates  $R_i$  and  $H_i \setminus S_i$ .  $\square$

**Proposition 17.1.3.** *A distribution which is decomposable with respect to a graph  $\mathcal{G}$  can be written as a product of the cliques  $C_i$  of  $\mathcal{G}$  divided by the product of the distributions of their intersections  $S_i$ :*

$$P(\mathbf{x}_V) = \prod_i \frac{P(\mathbf{x}_{C_i})}{P(\mathbf{x}_{S_i})}$$

known as the set-chain/marginal representation. If any denominator term is zero then the whole expression is considered zero.

*Proof.* For a given  $j$ :  $\bigcup_{i < j} R_i = \bigcup_{i < j} C_i = H_j$

and then  $P(\mathbf{x}_V) = \prod_i P(\mathbf{x}_{R_i} | \mathbf{x}_{R_1}, \dots, \mathbf{x}_{R_{i-1}}) = \prod_i P(\mathbf{x}_{R_i} | \mathbf{x}_{H_i})$  and, as  $\mathbf{x}_{R_i} \perp\!\!\!\perp \mathbf{x}_{H_i} | \mathbf{x}_{S_i}$  (see proposition 17.1.2), then  $P(\mathbf{x}_V) = \prod_i P(\mathbf{x}_{R_i} | \mathbf{x}_{S_i})$ .

$C_i = R_i \cup S_i$  and then  $P(\mathbf{x}_{C_i}) = P(\mathbf{x}_{R_i}, \mathbf{x}_{S_i}) = P(\mathbf{x}_{R_i} | \mathbf{x}_{S_i}) P(\mathbf{x}_{S_i})$  and the final result is obtained by replacing back into  $P(\mathbf{x}_V)$ .  $\square$

**Proposition 17.1.4.** *Let consider the sets of cliques  $\tilde{C}_A$  and  $\tilde{C}_B$ , in the join tree, separated by  $\tilde{C}_C$ . Considering that  $\tilde{C}_A$  contains the set of vertices  $A$ ,  $\tilde{C}_B$  contains  $B$  and  $\tilde{C}_C$  contains the set of vertices  $C$ , then  $\mathbf{x}_A \perp\!\!\!\perp \mathbf{x}_B | \mathbf{x}_C$ , i.e.  $A$  and  $B$  are separated by  $C$ .*

*Proof.* It is first proven, by contradiction, that  $C$  separates  $A \setminus C$  and  $B \setminus C$  on  $\mathcal{G}$ .

Let consider that there is a path from  $v_0 \in A$  to  $v_n \in B$ , passing through vertices  $v_1, \dots, v_{n-1} \notin C$ .

Let consider that  $v_0 \in C_0 \in \tilde{C}_A$  (where  $C_0$  is some clique containing  $v_0$ ). Let consider some vertex  $v_j$  such that  $v_{j-1}, v_j \in C_j$  (some  $C_j$ , note that  $(v_{j-1}, v_j)$  is an edge so there some clique containing it). As  $v_{j-1}, v_j \notin C$  then  $C_{j-1}, C_j \notin C_C$ . Then, in the join tree, the path  $C_{j-1}$  to  $C_j$  do not pass through  $\tilde{C}_C$ , as all contain the  $v_j$  vertex (by the way the join tree was built).

In this way, by repeating the procedure (with  $v_{j-2}, v_{j-1}$ , e.t.c), it is possible to build a path from  $\tilde{C}_A$  to  $\tilde{C}_B$  not passing through  $\tilde{C}_C$ . Contradiction.

Finally, by using the global Markov property, the proposition is demonstrated.  $\square$

### Remarks:

➔ Instead of working of the original graph  $\mathcal{G}$ , it is possible to work on triangulated  $\mathcal{G}^\Delta$  (obtained from  $\mathcal{G}$  by the procedure previously explained).

❖  $\mathcal{G}^\Delta$

As  $\mathcal{G}^\Delta$  have all edges of  $\mathcal{G}$  plus (maybe) some more then the separation properties on  $\mathcal{G}^\Delta$  hold on the original  $\mathcal{G}$ .

## 17.2 Casual Networks

Basically the casual networks are represented by DAG's. The vertices are numbered according to the *topological sort* order, i.e. each vertex have associated a number  $i$  grater than

<sup>17.2</sup>See [Rip96] pp. 261–265.

the order number of its parent  $j(i)$ . Then, considering the graph, the probability of  $\mathbf{x}_V$  is:

$$P(\mathbf{x}_V) = P(x_1) \prod_{i>1} P(x_i|x_{j(i)}) \quad (17.4)$$

(of course the root having no parent its distribution is unconditioned). The directions being from root to the leafs this also means that:

$$x_k \perp\!\!\!\perp x_i | x_{j(i)} \quad \text{for } k < i$$

recursive model the DAG having the above property being also named a *recursive model*.

moral graph **Definition 17.2.1.** The **moral graph** of a DAG is build following the procedure:

1. All directed edges are replaced by undirected ones.
2. All the (common) parents of a vertex are joined by edges (by adding them if necessary).

**Proposition 17.2.1.** A recursive model on a DAG have the global Markov property and a potential representation on its associated moral graph of the form (17.1).

*Proof.* The potential representation is built as follows:

1. The potential of each clique is set to 1.
2. For each vertex  $i$ , select a clique (any) which contain both  $i$  and its parent  $j(i)$  and multiply its potential by  $P(x_i|x_{j(i)})$ .

In this way the expression (17.4) is transformed into a potential representation of the form (17.1). By using proposition 17.1.1 it follows that the graph have the global Markov property.  $\square$

## 17.3 The Boltzmann Machine

The Boltzmann machine have binary random variable associated with the vertices of a graph completely connected. The probability of a variable  $x_i$  associated to vertex  $i$  is obtained considering a regression over all other vertices. The join distribution is defined as:

$$P(\mathbf{x}_V) = \frac{1}{Z} \exp \left( \sum_{i,j, i < j} w_{ij} x_i x_j \right) \quad \text{where} \quad Z = \sum_{\mathbf{x}_V} \exp \left( \sum_{i,j, i < j} w_{ij} x_i x_j \right)$$

the parameters  $w_{ij}$  are the “connection weights”, symmetric ( $w_{ij} = w_{ji}$ );  $Z$  is the normalization constant (obtained from  $\sum_{\mathbf{x}_V} P(\mathbf{x}_V) = 1$ ).

The Boltzmann machine learns the join distribution of some inputs  $\mathbf{x}_I$  and outputs  $\mathbf{x}_Y$ ; some vertices  $\mathbf{x}_H$  are “hidden”. The join probability over (given) input and output  $\mathbf{x}_S = \mathbf{x}_I \cup \mathbf{x}_Y$  is obtained by summation over all possibilities for the hidden vertices:

$$P(\mathbf{x}_S) = \sum_{\mathbf{x}_H} P(\mathbf{x}_V)$$

---

<sup>17.3</sup>See [Rip96] pp. 279–281.

The problem is to find the weights  $w_{ij}$  given the training set. This is achieved by a gradient ascent method applied on the log-likelihood function:

$$\mathcal{L} = \sum_{\text{training set}} \ln P(\mathbf{x}_V) = \sum_{\text{training set}} \ln \left[ \sum_{\mathbf{x}_H} \exp \left( \sum_{i,j, i < j} w_{ij} x_i x_j \right) \right] - \ln Z$$

The derivative of  $\ln Z$  is:

$$\frac{\partial \ln Z}{\partial w_{ij}} = \frac{1}{Z} \sum_{\mathbf{x}_V} x_i x_j \exp \left( \sum_{i,j, i < j} w_{ij} x_i x_j \right) = P(x_i = 1, x_j = 1)$$

as all terms for which  $x_i = 0$  or  $x_j = 0$  cancels.

Considering  $\mathcal{L}_1$ , the contribution of just one training pattern to the log-likelihood function then:

$$\begin{aligned} \frac{\partial \mathcal{L}_1}{\partial w_{ij}} &= \frac{\sum_{\mathbf{x}_H} x_i x_j \exp \left( \sum_{i,j, i < j} w_{ij} x_i x_j \right)}{\sum_{\mathbf{x}_H} \exp \left( \sum_{i,j, i < j} w_{ij} x_i x_j \right)} - \frac{\partial \ln Z}{\partial w_{ij}} \\ &= P(x_i = 1, x_j = 1 | \mathbf{x}_S) - P(x_i = 1, x_j = 1) \end{aligned}$$

and for the whole log-likelihood:

$$\frac{\partial \mathcal{L}}{\partial w_{ij}} = \sum_{\text{training set}} [\Pr(x_i = 1, x_j = 1 | \mathbf{x}_S) - \Pr(x_i = 1, x_j = 1)]$$



### Remarks:

- ➔ To evaluate the above expression there are necessary two simulations: one for  $P(x_i = 1, x_j = 1)$  and one for  $P(x_i = 1, x_j = 1 | \mathbf{x}_S)$  (with “clamped” inputs and outputs). The resulting algorithm is very slow.



## **Advanced Topics**

---



## CHAPTER 18

# Matrix Operations on ANN

### ► 18.1 New Matrix Operations

As already seen, 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 write fully matrix formulas for many ANN algorithms. However some operations are the same across several algorithms and it would make sense to introduce new matrix operations for them, in order to avoid unnecessary operations and waste of storage space, on digital simulations.

**Definition 18.1.1.** *The addition/subtraction on rows/columns between a constant and a vector or a vector and a matrix is defined as follows:*

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

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

$$a \overset{R}{\oplus} \mathbf{x}^T \equiv a\hat{\mathbf{1}}^T + \mathbf{x}^T \quad a \overset{C}{\oplus} \mathbf{x} \equiv a\hat{\mathbf{1}} + \mathbf{x}$$

$$a \overset{R}{\ominus} \mathbf{x}^T \equiv a\hat{\mathbf{1}}^T - \mathbf{x}^T \quad a \overset{C}{\ominus} \mathbf{x} \equiv a\hat{\mathbf{1}} - \mathbf{x}$$

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

$$\mathbf{x}^T \overset{R}{\oplus} A \equiv \hat{\mathbf{1}}\mathbf{x}^T + A \quad \mathbf{x} \overset{C}{\oplus} A \equiv \mathbf{x}\hat{\mathbf{1}}^T + A$$

$$\mathbf{x}^T \overset{R}{\ominus} A \equiv \hat{\mathbf{1}}\mathbf{x}^T - A \quad \mathbf{x} \overset{C}{\ominus} A \equiv \mathbf{x}\hat{\mathbf{1}}^T - A$$



#### Remarks:

- ➡ These operations avoid an unnecessary expansion of a constant/vector to a vector/matrix, before doing an addition/subtraction.

- ➔ The operations defined above are commutative.
- ➔ When the operation involves a constant then it represents in fact an addition/subtraction from all elements of the vector/matrix. In this situation  $\overset{R}{\oplus}$  is practically equivalent with  $\overset{C}{\oplus}$  and  $\overset{R}{\ominus}$  is equivalent with  $\overset{C}{\ominus}$  (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.

❖  $\overset{R}{\odot}, \overset{C}{\odot}$

**Definition 18.1.2.** *The Hadamard product between a matrix and vector (row/column matrix) is defined as follows:*

$$\mathbf{x}^T \overset{R}{\odot} A \equiv \hat{\mathbf{1}} \mathbf{x}^T \odot A \quad \text{and} \quad \mathbf{x} \overset{C}{\odot} A \equiv \mathbf{x} \hat{\mathbf{1}}^T \odot A$$



**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 18.1.3.** *The (meta)operator  $\mathcal{H}$  takes as arguments two matrices of the same size and three operators. Depending over 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  $\alpha$ ,  $\beta$  and  $\gamma$  then  $\mathcal{H}\{A, B, \alpha, \beta, \gamma\} = B'$ , the elements of  $B'$  being:*

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

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



**Remarks:**

- ➔ While  $\mathcal{H}$  could be replaced by some operations with the sign function, it wouldn't be as efficient when used in simulations, and it may be used in several ANN algorithms.

## 18.2 Algorithms

### 18.2.1 Backpropagation

#### Plain backpropagation

Using definition 18.1.1, formulas (2.7b) and (2.7c) are written as:

$$\nabla_{\mathbf{z}_\ell} E = c W_{\ell+1}^T \cdot \left[ \nabla_{\mathbf{z}_{\ell+1}} E \odot \mathbf{z}_{\ell+1} \odot (1 \overset{C}{\ominus} \mathbf{z}_{\ell+1}) \right] \quad \text{for } \ell = \overline{1, L-1}$$

$$(\nabla E)_\ell = c \left[ \nabla_{\mathbf{z}_\ell} E \odot \mathbf{z}_\ell \odot (1 \stackrel{\text{C}}{\ominus} \mathbf{z}_\ell) \right] \cdot \mathbf{z}_{\ell-1}^T \quad \text{for } \ell = \overline{1, L}$$

and equations (2.10b) and (2.10c) becomes:

$$\nabla_{\mathbf{z}_\ell} E = c W_{\ell+1}^T \cdot \left[ \nabla_{\mathbf{z}_{\ell+1}} E \odot \mathbf{z}_{\ell+1} \odot (1 \stackrel{\text{C}}{\ominus} \mathbf{z}_{\ell+1}) \right] \quad \text{for } \ell = \overline{1, L-1}$$

$$\widetilde{(\nabla E)}_\ell = c \left[ \nabla_{\mathbf{z}_\ell} E \odot \mathbf{z}_\ell \odot (1 \stackrel{\text{C}}{\ominus} \mathbf{z}_\ell) \right] \cdot \tilde{\mathbf{z}}_{\ell-1}^T \quad \text{for } \ell = \overline{1, L}$$

### **Backpropagation with momentum**

Formulas (2.12a), (2.12b) and (2.12c) change to:

$$(\nabla E)_{\ell, \text{pseudo}} = c \left\{ \nabla_{\mathbf{z}_\ell} E \odot \left[ \mathbf{z}_\ell \odot (1 \stackrel{\text{C}}{\ominus} \mathbf{z}_\ell) \stackrel{\text{C}}{\oplus} c_f \right] \right\} \cdot \mathbf{z}_{\ell-1}^T$$

$$\widetilde{(\nabla E)}_{\ell, \text{pseudo}} = \left\{ \nabla_{\mathbf{z}_\ell} E \odot \left[ f'(\mathbf{a}_\ell) \stackrel{\text{C}}{\oplus} c_f \right] \right\} \cdot \tilde{\mathbf{z}}_{\ell-1}^T$$

$$\widetilde{(\nabla E)}_{\ell, \text{pseudo}} = c \left\{ \nabla_{\mathbf{z}_\ell} E \odot \left[ \mathbf{z}_\ell \odot (1 \stackrel{\text{C}}{\ominus} \mathbf{z}_\ell) \stackrel{\text{C}}{\oplus} c_f \right] \right\} \cdot \tilde{\mathbf{z}}_{\ell-1}^T$$

### **Adaptive Backpropagation**

From (2.13) and using definition 18.1.3, (2.14) is replaced by:

$$\mu(t) = \mathcal{H}\{\Delta W(t) \odot \Delta W(t-1), \mu(t-1), \mathcal{I}\cdot, \mathcal{I}\cdot, \mathcal{D}\cdot\}$$

### **SuperSAB**

Using the  $\mathcal{H}$  operator, the SuperSAB rules may be written as:

$$\mu(t) = \mathcal{H}\{\Delta W(t) \odot \Delta W(t-1), \mu(t-1), \mathcal{I}\cdot, \mathcal{I}\cdot, \mathcal{D}\cdot\}$$

$$\Delta W(t+1) = -\mu(t) \odot \nabla E - \mathcal{H}\{\Delta W(t) \odot \Delta W(t-1), \Delta W(t), = 0, = 0, -\alpha \cdot\}$$

(note that the product  $\Delta W(t) \odot \Delta W(t-1)$  is used twice and it is probably better to calculate it just once, before applying these formulas).

## 18.2.2 SOM/Kohonen Networks

The algorithms heavily depend over the  $dW/dt$  equation chosen to model the learning process.

The trivial equation (3.1) is changed to:  $\frac{dW}{dt} = \alpha \mathbf{x}^T \stackrel{\text{R}}{\ominus} \beta W$ .

The Riccati equation (3.5) becomes:  $\frac{dW}{dt} = \alpha \mathbf{x}^T \stackrel{\text{R}}{\ominus} (W \mathbf{x}) \stackrel{\text{C}}{\odot} W$  (see proof of (3.5),  $\frac{dW}{dt} = \alpha \hat{\mathbf{1}} \mathbf{x}^T - \beta \mathbf{a} \stackrel{\text{C}}{\odot} W$ ).

The more general equations (3.2.1) and (3.2.2):

$$\frac{dW}{dt} = \alpha \mathbf{x}^T \stackrel{\text{R}}{\ominus} \gamma(W\mathbf{x}) \stackrel{\text{C}}{\odot} W \quad \text{and} \quad \frac{dW}{dt} = \alpha W \mathbf{x} \mathbf{x}^T - \gamma(W\mathbf{x}) \stackrel{\text{C}}{\odot} W$$

The trivial model with a neuronal neighborhood and a stop condition (3.22) will be written as:

$$W(t+1) = W(t) + \tau(t) h(|\mathbf{x}_{(K)} \stackrel{\text{C}}{\odot} x_{(K)k}|) \stackrel{\text{C}}{\odot} \left( \alpha \mathbf{x}^T \stackrel{\text{R}}{\ominus} \beta W \right)$$



#### Remarks:

- The above equations are just examples. There are many possible variations in SOM/Kohonen networks and it's very easy to build own equations, according to the model chosen.

### 18.2.3 BAM/Hopfield Networks

#### BAM networks

The (4.3) formulas change to:

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

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

and for working in reverse, (4.4) become:

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

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

The final algorithm change accordingly.

#### Discrete Hopfield memory

Formula (4.6) transforms to:

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

#### Continuous Hopfield memory

(4.10) may be written as:

$$\mathbf{y}(t+1) = \mathbf{y}(t) + \left[ W \mathbf{y} + \mathbf{x} - \frac{1}{\lambda} \mathbf{t} \odot \ln \left( \mathbf{y}(t) \oslash (1 \stackrel{\text{C}}{\ominus} \mathbf{y}(t)) \right) \right] \odot \mathbf{y}(t) (1 \stackrel{\text{C}}{\ominus} \mathbf{y}(t))$$



Here  $\oslash$  signify the element-wise division between  $\mathbf{y}(t)$  and  $1 \stackrel{\text{C}}{\ominus} \mathbf{y}(t)$ . The  $\ln$  function follows the convention used in this book for scalar functions applied to vectors: it applies to each vector element in turn.

## 18.3 Conclusions

The new matrix operations seems to be justified by their usage across several very different network architectures. The table 18.1 shows their usage.

Operation	ANN architectures
$\oplus^R$	—
$\odot^R$	SOM
$\oplus^C$	momentum
$\odot^C$	backpropagation, momentum, SOM, continuous Hopfield
$\odot^R$	—
$\odot^C$	SOM
$\mathcal{H}$	adaptive backpropagation, SuperSAB, BAM, discrete Hopfield

**Table 18.1:** *The usage of new matrix operations across ANN architectures.*

It may be seen that two operations, i.e.  $\oplus^R$  and  $\odot^R$  were not used in the ANN algorithms studied here. However they were defined because:

- there are many other yet unchecked algorithms and they may find an usage;
- together with the rest of operations, they form a complete (symmetrical) system allowing for a large variety of matrix/vector manipulations.

The fact that so different ANN architectures could be expressed in terms of fully matrix equations *strongly suggests* that many other algorithms (if not all) may be converted to full matrix formulas.

One other operator, the element-wise “Hadamard division”  $\odot$ , also seems to be useful; it represents the “opposite” of Hadamard product, possibly filling a gap in matrix operations.

The usage of matrix formulas on numerical simulations have 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 already implemented;
- *ANN algorithms expressed trough the new matrix operations, do not lead to 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;
- makes debugging of ANN implementations easier.

In order to fully take advantage of the matrix formulas, it may be necessary to have some supplemental support:

- scalar functions when applied to vectors, do in fact apply to each element in turn.
- some algorithms use the summation over all elements of a vector, i.e. a operation of type  $\mathbf{x}^T \hat{\mathbf{1}}$ .

## APPENDIX A

# Mathematical Sidelines

## ► A.1 Distances

### A.1.1 Euclidean Distance

Let be two real vectors  $\mathbf{x}, \mathbf{y} \in \mathbb{R}^n$  of dimension  $n \in \mathbb{N}$ .

❖  $\mathbf{x}, \mathbf{y}, n$

The Euclidean distance  $d$  between the vectors  $\mathbf{x}$  and  $\mathbf{y}$  is defined as:

❖  $d$

$$d = \|\mathbf{x} - \mathbf{y}\| = \sqrt{\sum_{i=1}^n (x_i - y_i)^2} \quad (\text{A.1})$$

Also, considering the vectors as column matrices then  $d = \sqrt{\mathbf{x}^T \mathbf{y}}$ .

### A.1.2 Hamming Distance

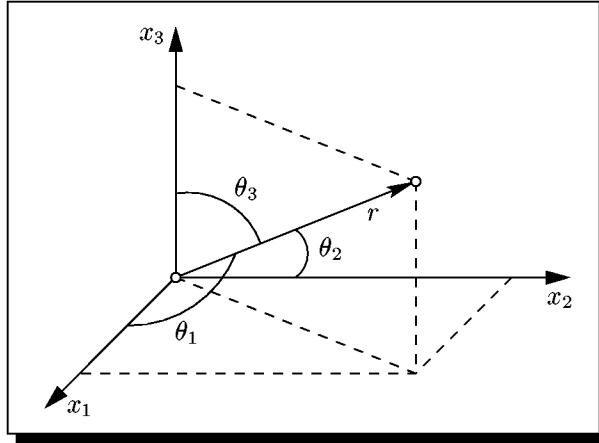
The Hamming space of dimension  $n$  is defined as:

$$\mathbf{H}^n = \{\mathbf{x}^T = (x_1 \dots x_n) \in \mathbb{R}^n | x_i \in \{-1, 1\}\}$$

so in an Euclidean space the Hamming space can be represented as a set of  $2^n$  points at equal distance from origin (corners of a hyper-cube).

The Hamming distance between 2 (Hamming) points  $\mathbf{x}$  and  $\mathbf{y}$  is defined as:

$$h = \|\mathbf{x} - \mathbf{y}\|_H = \sum_{i=1}^n \delta(x_i, y_i) \quad \text{where} \quad \delta(x_i, y_i) = \begin{cases} 0 & \text{if } x_i = y_i \\ 1 & \text{if } x_i \neq y_i \end{cases}$$



**Figure A.1:** The generalized spherical coordinates. The angles  $\theta_i$  are measured from the position vector of the point to the axes of a Cartesian system of coordinates.

i.e.  $h$  represents the number of mismatched components of  $\mathbf{x}$  and  $\mathbf{y}$ .

**Remarks:**

- For 2 Euclidean vectors  $\mathbf{x}$  and  $\mathbf{y}$  subject to the restriction that  $x_i, y_i \in \{-1, 1\}$

$$\text{then (see (A.1))}: (x_i - y_i)^2 = \begin{cases} 0 & \text{if } x_i = y_i \\ 4 & \text{if } x_i \neq y_i \end{cases} \Rightarrow h = \frac{d^2}{4}.$$

## ► A.2 Generalized Spherical Coordinates

Considering an  $n$ -dimensional space it is possible to define the position of an arbitrary point by the means of  $n$  angles and a distance:  $\{\theta_1, \dots, \theta_n, r\}$ .  $r$  represents the distance from the (current) point to the origin of the coordinates system while the angles  $\theta_i$  are measured between the position vector and the axes of a Cartesian orthogonal system. See figure A.1.

**Remarks:**

- Note that the system  $\{r, \theta_1, \theta_n\}$  have  $n + 1$  elements and thus the coordinates are not independent.

By using repetitively the Pitagora theorem:

$$|r|^2 = (r \cos \theta_1)^2 + \dots + (r \cos \theta_n)^2 \Rightarrow \sum_{i=1}^n \cos^2 \theta_i = 1$$

## ► A.3 Properties of Symmetric Matrices

A matrix  $A$  is called symmetric if it's square ( $A_{ij} = A_{ji}$ ) and symmetric, i.e. the matrix is equal with its transposed  $A = A^T$ .

**Proposition A.3.1.** *The inverse of a symmetric matrix, if exists, is also symmetric.*

*Proof.* It is assumed that the inverse  $A^{-1}$  does exist. Then, by definition,  $A^{-1}A = I$ , where  $I$  is the unit matrix.

For any two matrices  $A$  and  $B$  it is true that  $(AB)^T = B^TA^T$ . Applying this result, it gives that  $A^TA^{-1T} = I$ .

Finally, multiplying with  $A^{-1}$ , to the left, and knowing that  $A^T = A$ , it follows that  $A^{-1T} = A^{-1}$ .  $\square$

### A.3.1 Eigenvectors and Eigenvalues

Let assume that there are a set of eigenvectors  $\{\mathbf{u}_i\}_{i=1,n}$  and a corresponding set of eigenvalues  $\{\lambda_i\}_{i=1,n}$ , such that  $\diamond \mathbf{u}_i, \lambda_i$

$$A\mathbf{u}_i = \lambda_i \mathbf{u}_i \quad , \quad i = \overline{1, n} \quad (\text{A.2})$$

The eigenvalues are found from the general equation:

$$Ax = \lambda x \Leftrightarrow (A - \lambda I)x = \hat{0}$$

If the matrix  $A - \lambda I$  would have an inverse, then by multiplying the above equation by its inverse it would give that  $x = \hat{0}$ , i.e. all eigenvectors are zero. To avoid this situation is necessary to impose the condition that  $A - \lambda I$  matrix is not inversable, i.e. its determinant is null:

$$|A - \lambda I| = 0$$

and this leads to the characteristic polynom of  $A$ , of the  $n$ -th degree in  $\lambda$ , whose roots are the eigenvalues. The set of  $\{\lambda_i\}_{i=1,n}$  is also named the *spectrum of  $A$* .

**Proposition A.3.2.** *If two eigen vectors are parallel then they represent the same eigen value — assuming that they are non-zero.*

*Proof.* Let assume, by absurd, that the above proposition is not true, i.e. there are two eigenvectors  $\mathbf{u}_1 \parallel \mathbf{u}_2 \Leftrightarrow \mathbf{u}_1 = \alpha \mathbf{u}_2$ , where  $\alpha$  is a any non-zero constant, such that  $A\mathbf{u}_1 = \lambda_1 \mathbf{u}_1$  and  $A\mathbf{u}_2 = \lambda_2 \mathbf{u}_2$  and  $\lambda_1 \neq \lambda_2$ .

But then the following is also true:  $A\mathbf{u}_1 = \lambda_1 \mathbf{u}_1$  and  $A\alpha \mathbf{u}_1 = \lambda_2 \alpha \mathbf{u}_1$  and, by subtracting the equations it follows that  $1 - \alpha = \lambda_1 - \alpha \lambda_2$ . Finally, for  $\alpha = 1$  it gets that the two eigenvalues are equal.  $\square$

The eigenvectors are defined up to a multiplicative constant; if a vector  $\mathbf{u}_i$  is an eigenvector then the  $\alpha_i \mathbf{u}_i$  is also an eigenvector for the same eigenvalue (where  $\alpha_i$  is some constant).

Let consider two arbitrary chosen eigenvectors  $\mathbf{u}_i$  and  $\mathbf{u}_j$ . By multiplying (A.2) with  $\mathbf{u}_j^T$  and the equation  $A\mathbf{u}_j = \lambda_j \mathbf{u}_j$  with  $\mathbf{u}_i^T$ :

$$\mathbf{u}_j^T A \mathbf{u}_i = \lambda_i \mathbf{u}_j^T \mathbf{u}_i \quad \text{and} \quad \mathbf{u}_i^T A \mathbf{u}_j = \lambda_j \mathbf{u}_i^T \mathbf{u}_j$$

<sup>A.3</sup>See [Bis95] pp. 440–443.

Considering  $A$  is symmetric then  $\mathbf{u}_j^T A \mathbf{u}_i = \mathbf{u}_i^T A \mathbf{u}_j$  (using  $(AB)^T = B^T A^T$ ) and  $\mathbf{u}_j^T \mathbf{u}_i = \mathbf{u}_i^T \mathbf{u}_j$  (whatever  $\mathbf{u}_i$  and  $\mathbf{u}_j$ ). By subtracting the two above equations:

$$(\lambda_i - \lambda_j) \mathbf{u}_i^T \mathbf{u}_j = 0$$

Two situations arises:

- $\lambda_i \neq \lambda_j$ : Then  $\mathbf{u}_i^T \mathbf{u}_j = 0$ , i.e.  $\mathbf{u}_i \perp \mathbf{u}_j$  — the vectors are orthogonal.
- $\lambda_i = \lambda_j$ : by substituting  $\lambda_i$  and respectively  $\lambda_j$  in (A.2) and adding the two equation obtained, it gets that a linear combination of the two eigenvectors  $\alpha \mathbf{u}_i + \beta \mathbf{u}_j$  is also an eigenvector. Because the two vectors are not parallel  $\mathbf{u}_i \nparallel \mathbf{u}_j$  then they define a plane and, a pair of orthogonal vectors as a linear combination, of the two may be chosen.

The same rationing may be done for more than 2 equal eigen values.

❖  $U$

Considering the above discussion, then the eigenvector set  $\{\mathbf{u}_i\}$  may be easily normalized, such that  $\mathbf{u}_i^T \mathbf{u}_j = \delta_{ij}$ ,  $\forall i, j$ . Also the associated matrix  $U$ , built using  $\{\mathbf{u}_i\}$  as columns, is orthogonal  $U^T U = U U^T = I$ , i.e.  $U^T = U^{-1}$  (by multiplying the true relation  $U^T U = I$ , to the left, by  $U^{-1}$ ).

**Proposition A.3.3.** *The inverse of the matrix  $A$  have the same eigenvectors and the  $1/\lambda_i$  eigenvalues, i.e.  $A^{-1} \mathbf{u}_i = \frac{1}{\lambda_i} \mathbf{u}_i$ .*

*Proof.* By multiplying (A.2) with  $A^{-1}$  to the left and  $A^{-1} A = I$  it gives  $\mathbf{u}_i = \lambda_i A^{-1} \mathbf{u}_i$ . □

❖  $\Lambda$

### Remarks:

► The  $A$  matrix may be diagonalized. From  $A \mathbf{u}_i = \lambda_i \mathbf{u}_i$  and, by multiplying by  $\mathbf{u}_j^T$  to the left:  $\mathbf{u}_j^T A \mathbf{u}_i = \lambda_i \delta_{ij}$ , so, in matrix notation, it may be written as

$$U^T A U = \Lambda, \text{ where } \Lambda = \begin{pmatrix} \lambda_1 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & \lambda_n \end{pmatrix}. \text{ Then:}$$

$$|U^T A U| = |U^T| |A| |U| = |U^T U| |A| = |I| |A| = |A| = |\Lambda| = \prod_{i=1}^n \lambda_i$$

**Proposition A.3.4. Rayleigh Quotient.** *For  $A$  a square matrix and any  $\mathbf{a}$  vector, it is true that:*

$$\frac{\mathbf{a}^T A \mathbf{a}}{\|\mathbf{a}\|^2} \leq \lambda_{\max}$$

where  $\lambda_{\max} = \max_i \lambda_i$  is the maximum eigenvalue; Euclidean metric being used here.

*Proof.* The above relation should be unaffected by a coordinate transformation. Then let use the coordinate transformation defined by matrix  $U$ . The new representation of vector  $\mathbf{a}$  is then  $\mathbf{a}' = U \mathbf{a}$  and respectively  $\mathbf{a}'^T = \mathbf{a}^T U^T$  (use  $(AB)^T = B^T A^T$ ). The (Euclidean) norm remains unchanged  $\|\mathbf{a}'\|^2 = \mathbf{a}'^T \mathbf{a}' = \mathbf{a}^T U^T U \mathbf{a} = \|\mathbf{a}\|^2$  ( $U^T U = I$ ). Then:

$$\frac{\mathbf{a}^T A \mathbf{a}}{\|\mathbf{a}\|^2} \leq \lambda_{\max} \Leftrightarrow \frac{\mathbf{a}'^T A \mathbf{a}'}{\|\mathbf{a}'\|^2} \leq \lambda_{\max} \Leftrightarrow \frac{\mathbf{a}^T U^T A U \mathbf{a}}{\|\mathbf{a}\|^2} \leq \lambda_{\max}$$

and as  $U^T A U = \Lambda$  then the above inequation is equivalent with:

$$\mathbf{a}^T \Lambda \mathbf{a} = \sum_i \lambda_i a_i^2 \leq \lambda_{\max} \|\mathbf{a}\|^2 = \lambda_{\max} \sum_i a_i^2$$

which, obviously, is true.  $\square$

**Definition A.3.1.** A matrix  $A$  is positive definite if  $\mathbf{x}^T A \mathbf{x} > 0, \forall \mathbf{x} \neq 0$ .

From (A.2), considering a orthonormal set of eigen vectors and by multiplying to the left by  $\mathbf{u}_i^T$  then  $\lambda_i = \mathbf{u}_i^T A \mathbf{u}_i$  and the eigen values are positive for positive definite matrix.

### A.3.2 Rotation

**Proposition A.3.5.** If the matrix  $U$  is used for a coordinate transformation, the result is a rotation.  $U$  is supposed normated.

*Proof.* Let  $\tilde{\mathbf{x}} = U^T \mathbf{x}$ . Then  $\|\tilde{\mathbf{x}}\|^2 = \tilde{\mathbf{x}}^T \tilde{\mathbf{x}} = \mathbf{x}^T U U^T \mathbf{x} = \|\mathbf{x}\|^2$  (use  $(AB)^T = B^T A^T$ ), i.e. the length of the vector is not changed.

Let be two vectors  $\tilde{\mathbf{x}}_1 = U^T \mathbf{x}_1$  and  $\tilde{\mathbf{x}}_2 = U^T \mathbf{x}_2$ . Then  $\tilde{\mathbf{x}}_1^T \tilde{\mathbf{x}}_2 = \mathbf{x}_1^T U U^T \mathbf{x}_2 = \mathbf{x}_1^T \mathbf{x}_2$ , i.e. the angle between two vectors is preserved.

The only transformation preserving lengths and angles is the rotation.  $\square$

### A.3.3 Quadratic Forms

A quadratic form of type:  $F(\mathbf{x}) = \mathbf{x}^T A \mathbf{x}$  where  $A$  is an arbitrary square matrix.

By using the eigenvectors of  $A$  the function  $F(\mathbf{x})$  becomes:

$$F(\mathbf{x}) = \mathbf{x}^T A \mathbf{x} = \mathbf{x}^T U U^T A U U^T \mathbf{x} = \tilde{\mathbf{x}}^T U^T A U \tilde{\mathbf{x}} = \tilde{\mathbf{x}}^T \Lambda \tilde{\mathbf{x}} = \sum_{i=1}^n \lambda_i \tilde{x}_i^2$$

(because  $U U^T = I$ ,  $\tilde{\mathbf{x}} = U^T \mathbf{x}$  and  $U^T A U = \Lambda$ ).

## ► A.4 The Gaussian Integrals

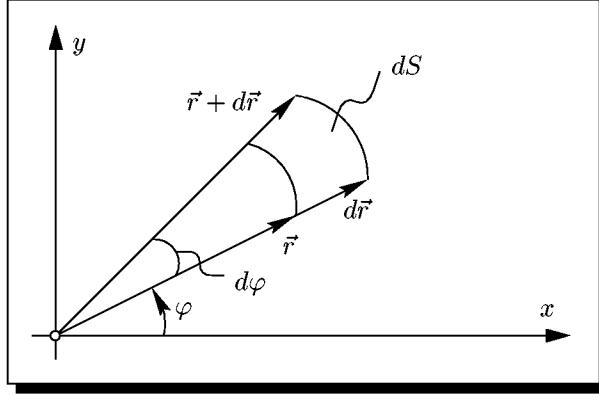
### A.4.1 The Unidimensional Case

Let  $I = \int_{-\infty}^{\infty} e^{-x^2} dx$  and then  $I^2 = \int_{-\infty}^{\infty} e^{-x^2} dx \int_{-\infty}^{\infty} e^{-y^2} dy = \iint_{\mathbb{R}^2} e^{-(x^2+y^2)} dS$ , where  $dS = dx dy$  is the elementary surface.

By switching to polar coordinates, see figure A.2 on the following page,  $x^2 + y^2 = r^2$  and  $dS = dr \cdot r d\varphi$ ; then the integral becomes:

$$I^2 = \iint_{\mathbb{R}^2} e^{-r^2} dS = \int_0^\infty \int_0^{2\pi} e^{-r^2} r dr d\varphi = 2\pi \int_0^\infty r e^{-r^2} dr = 2\pi \left( -\frac{1}{2} e^{-r^2} \right) \Big|_0^\infty = \pi$$

<sup>A.4</sup>See [Bis95] pp. 444-447.



**Figure A.2:** Polar coordinates: The surface element  $dS$ .

Finally  $\int_{-\infty}^{\infty} e^{-x^2} dx = \sqrt{\pi}$  and  $\int_0^{\infty} e^{-x^2} dx = \frac{\sqrt{\pi}}{2}$  because  $e^{-x^2}$  is an even function (same value for  $x$  and  $-x$ ).

#### A.4.2 The Multidimensional Case

Let  $I = \int_{\mathbb{R}^n} \exp\left(-\frac{\mathbf{x}^T A \mathbf{x}}{2}\right) d\mathbf{x}$  where  $A$  is a  $n \times n$  square and symmetric matrix and  $\mathbf{x} \in \mathbb{R}^n$  ( $d\mathbf{x} = dx_1 \cdot dx_2 \dots dx_n$ ).

Since  $A$  is symmetrical, it is possible to build a orthonormal set of eigenvectors  $\{\mathbf{u}_i\}_{i=1,n}$  such that  $\mathbf{u}_i^T \mathbf{u}_j = \delta_{ij}$  (see section A.3), and then the  $\mathbf{x}$  vector may be written as  $\mathbf{x} = \sum_{i=1}^n \alpha_i \mathbf{u}_i$ .

The change of variable from  $\{x_i\}_{i=1,n}$  to  $\{\alpha_i\}_{i=1,n}$  is done. Then  $\mathbf{x}^T A \mathbf{x} = \sum_{i=1}^n \lambda_i \alpha_i^2$ , where  $\{\lambda_i\}_{i=1,n}$  are the eigenvalues of  $A$ , and  $d\mathbf{x} = \left| \left\{ \frac{\partial x_i}{\partial \alpha_j} \right\}_{ij} \right| d\alpha_1 \dots d\alpha_n$ .

❖  $u_{ij} = \frac{\partial x_i}{\partial \alpha_j}$  — where  $u_{ij}$  is the  $i$ -th element of the vector  $\mathbf{u}_j$  — and, because the  $\{\mathbf{u}_j\}_{j=1,n}$  is orthonormal, then for the associated matrix  $U$  is true that  $U^T U = I$  (the matrix is orthogonal, see section A.3) and the Jacobian determinant  $|J| = \left| \left\{ \frac{\partial x_i}{\partial \alpha_j} \right\}_{ij} \right|$  becomes:

$$|J|^2 = |U|^2 = |U^T| |U| = |U^T U| = |I| = 1 \Rightarrow |J| = 1$$

(the integrand  $\exp\left(-\frac{\mathbf{x}^T A \mathbf{x}}{2}\right)$  is positive over all space  $\mathbb{R}^n$ , then the integral is positive and then the solution  $|J| = -1$  is not acceptable). Finally the integral becomes:

$$I = \prod_{i=1}^n \int_{-\infty}^{\infty} \exp\left(-\frac{\lambda_i \alpha_i^2}{2}\right) d\alpha_i = \prod_{i=1}^n \sqrt{\frac{2\pi}{\lambda_i}}$$

Because  $|A| = \prod_{i=1}^n \lambda_i$  then  $I = \frac{(2\pi)^{n/2}}{\sqrt{|A|}}$

### A.4.3 The multidimensional Gaussian integral with a linear term

Let  $I = \int_{\mathbb{R}^n} \exp\left(-\frac{x^T A x}{2} + c^T x\right) dx$  where  $A$  is a  $n \times n$  square and symmetric matrix,  $x \in \mathbb{R}^n$  and  $c \in \mathbb{R}^n$  is a constant vector ( $dx = dx_1 \cdot dx_2 \dots dx_n$ ).

Let  $\{\mathbf{u}_i\}_{i=1,n}$  be the set of orthonormalized eigenvectors of  $A$ .

The  $c$  vector may be written by means of the eigenvectors set as  $c = \sum_{i=1}^n c_i \mathbf{u}_i$  where  $c_i = c^T \mathbf{u}_i$ , (as  $\mathbf{u}_i^T \mathbf{u}_j = \delta_{ij}$ ), are called the projections of  $c$  on  $\mathbf{u}_i$ .

Finally, similar to the multidimensional Gaussian integral (above) the integral may be transformed into a product of independent integrals

$$I = \prod_{i=1}^n \int_{-\infty}^{\infty} \exp\left(-\frac{\lambda_i \alpha_i^2}{2} + c_i \alpha_i\right) d\alpha_1 \dots d\alpha_n$$

A square is forced in the exponent:  $-\frac{\lambda_i \alpha_i^2}{2} + c_i \alpha_i = -\frac{\lambda_i}{2} \left(\alpha_i - \frac{c_i}{\lambda_i}\right)^2 + \frac{c_i^2}{2\lambda_i}$ , such that the integral becomes:

$$I = \prod_{i=1}^n \exp\left(\frac{c_i^2}{2\lambda_i}\right) \int_{-\infty}^{\infty} \exp\left[-\frac{\lambda_i}{2} \left(\alpha_i - \frac{c_i}{\lambda_i}\right)^2\right] d\alpha_i$$

A new change of variable is done:  $\tilde{\alpha}_i = \alpha_i - \frac{c_i}{\lambda_i}$ , then  $d\tilde{\alpha}_i = d\alpha_i$ , the integral limits remain the same, and:

$$I = \exp\left(\sum_{i=1}^n \frac{c_i^2}{2\lambda_i}\right) \prod_{i=1}^n \int_{-\infty}^{\infty} \exp\left(-\frac{\lambda_i \tilde{\alpha}_i^2}{2}\right) d\tilde{\alpha}_i$$

Similar as for multidimensional Gaussian integrals:  $I = \frac{(2\pi)^{n/2}}{\sqrt{|A|}} \exp\left(\sum_{i=1}^n \frac{c_i^2}{2\lambda_i}\right)$ . Because  $A^{-1} \mathbf{u}_i = \frac{1}{\lambda_i} \mathbf{u}_i$  (see section A.3) then:  $c^T A^{-1} c = \sum_{i=1}^n \frac{c_i^2}{\lambda_i}$  and, finally:

$$I = \frac{(2\pi)^{n/2}}{\sqrt{|A|}} \exp\left(\frac{c^T A^{-1} c}{2}\right)$$

## ► A.5 The Euler Functions

### A.5.1 The Euler function

The Euler function  $\Gamma_E(x)$  is defined as being:

❖  $\Gamma_E$

$$\Gamma_E(x) = \int_0^\infty e^{-t} t^{x-1} dt \quad (\text{A.3})$$

and is convergent for  $x > 0$ .

**Proposition A.5.1.** *For the Euler function it is true that  $\Gamma_E(x+1) = x\Gamma_E(x)$*

*Proof.* Integrating by parts:

$$\Gamma_E(x) = \int_0^\infty e^{-t} t^{x-1} dt = e^{-t} \frac{t^x}{x} \Big|_0^\infty + \frac{1}{x} \int_0^\infty e^{-t} t^{-x} dt = \frac{1}{x} \Gamma_E(x+1) \quad \square$$

**Proposition A.5.2.** *If  $n \in \mathbb{N}$  then  $n! = \Gamma_E(n+1)$  where  $0! = 1$  by definition.*

*Proof.* It is demonstrated by mathematical induction.

For  $n = 0$ :  $\Gamma_E(1) = \int_0^\infty e^{-t} dt = -e^{-t} \Big|_0^\infty = 1 = 0!$  and for  $n = 1$ , by using proposition A.5.1:  $\Gamma_E(2) = 1 \cdot \Gamma_E(1) = 1 = 1!$ .

It is assumed that  $n! = \Gamma_E(n+1)$  is true and then:

$$(n+1)! = (n+1) \cdot n! = (n+1)\Gamma_E(n+1) = \Gamma_E(n+2)$$

the equation  $(n+1)! = \Gamma_E(n+2)$  is also true.  $\square$

## A.5.2 The sphere volume in the n-dimensional space

It is assumed that the volume of a sphere of radius  $r$  into a  $n$ -dimensional space  $V_n$  is proportional with the power  $n$  of the radius:

$$V_n = C_n r^n , \quad C_n = \text{const.}$$

where  $C_n$  is to be found.

The integral:

$$I_n = \int_{-\infty}^{\infty} \cdots \int_{-\infty}^{\infty} \exp[a(x_1^2 + \cdots + x_n^2)] dx_1 \dots dx_n , \quad a = \text{const.}$$

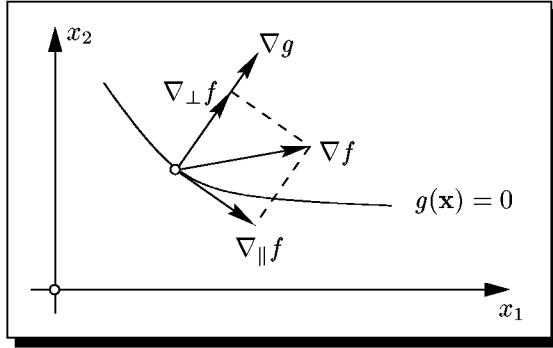
is calculated in two ways:

1. The integrals from  $I_n$  are decoupled such that  $I_n = \left( \int_{-\infty}^{\infty} e^{-ax^2} dx \right)^n = \left( \frac{\pi}{a} \right)^{n/2}$ .
2. The change of variable from Cartesian coordinates to generalized spherical coordinates is performed:

$$x_1^2 + \cdots + x_n^2 = r^2 , \quad dx_1 \dots dx_n = dV_n = nC_n r^{n-1} dr$$

where the elementary volume in spherical coordinates may be assumed as an infinitesimal spherical layer, due to the symmetry of the integrand relatively to origin. Then

$$I_n \text{ becomes: } I_n = nC_n \int_0^\infty r^{n-1} e^{-ar^2} dr.$$



**Figure A.3:** The gradient vectors  $\nabla f$  and  $\nabla g$  into a bidimensional space.

A new change of variable is performed:  $ar^2 = x \Rightarrow \begin{cases} dr = \frac{1}{2\sqrt{a}} x^{-1/2} dx \\ r^{n-1} = \frac{1}{a^{\frac{n-1}{2}}} x^{\frac{n-1}{2}} \end{cases}$  and the integral becomes:

$$I_n = \frac{nC_n}{2a^{n/2}} \int_0^\infty x^{n/2-1} e^{-x} dx = \frac{nC_n}{2a^{n/2}} \Gamma_E\left(\frac{n}{2}\right) = \frac{C_n}{a^{n/2}} \Gamma_E\left(\frac{n}{2} + 1\right)$$

By comparing the two results  $C_n = \frac{\pi^{n/2}}{\Gamma_E\left(\frac{n}{2} + 1\right)}$  and finally:

$$V_n = \frac{\pi^{n/2}}{\Gamma_E\left(\frac{n}{2} + 1\right)} r^n$$

## ► A.6 The Lagrange Multipliers

The problem is to find the stationary points of a function  $f(\mathbf{x})$  subject to a relation between the components of vector  $\mathbf{x}$ , given as  $g(\mathbf{x}) = 0$ .

Geometrically,  $g(\mathbf{x}) = 0$  represents a surface in the  $X^n$  space, where  $n$  is the dimension of that space. At each point  $\nabla g$  represents a vector perpendicular on that surface and  $\nabla f$  may be expressed as  $\nabla f = \nabla_{\parallel} f + \nabla_{\perp} f$ , where  $\nabla_{\parallel} f$  is the component parallel with the surface  $g(\mathbf{x}) = 0$  and  $\nabla_{\perp} f$  is the component perpendicular on it. See figure A.3.

❖  $\nabla_{\parallel}$ ,  $\nabla_{\perp}$

### Remarks:

- ➔ Considering a point in the vicinity of  $\mathbf{x}$ , on the  $g(\mathbf{x})$  surface, such that it is defined by the vector  $\mathbf{x} + \boldsymbol{\varepsilon}$ , where  $\boldsymbol{\varepsilon}$  lies within the surface defined by  $g(\mathbf{x}) = 0$ , then the Taylor development around  $\mathbf{x}$  is:

$$g(\mathbf{x} + \boldsymbol{\varepsilon}) = g(\mathbf{x}) + \boldsymbol{\varepsilon}^T \nabla g(\mathbf{x})$$

<sup>A.6</sup>See [Bis95] pp. 448–450.

Lagrange  
multiplier

and, on the other hand,  $g(\mathbf{x} + \varepsilon) = g(\mathbf{x}) = 0$  because of the choice of  $\varepsilon$ .

Then  $\varepsilon^T \nabla g(\mathbf{x}) = 0$ , i.e.  $\nabla g(\mathbf{x})$  is perpendicular on the surface  $g(\mathbf{x}) = 0$ .

As  $\nabla_{\perp} f \parallel \nabla g$  (see above), then it is possible to write  $\nabla_{\perp} f = -\lambda \nabla g$  where  $\lambda$  is called the *Lagrange multiplier* or *undetermined multiplier*, and  $\nabla_{\parallel} f = \nabla f + \lambda \nabla g$ .

The following *Lagrange function* is defined:

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \lambda g(\mathbf{x})$$

such that  $\nabla L = \nabla_{\parallel} f$  and the condition for the stationary points of  $f$  is  $\nabla L = \hat{\mathbf{0}}$ .

For the  $n$ -dimensional space, the  $\nabla L = \hat{\mathbf{0}}$  condition gives  $n + 1$  equations:

$$\frac{\partial L}{\partial x_i} = 0, \quad i = \overline{1, n} \quad \text{and} \quad \frac{\partial L}{\partial \lambda} = g(\mathbf{x}) = 0$$

and the constraint  $g(\mathbf{x}) = 0$  is also met.

More general, for a set of constraints  $g_i(\mathbf{x}) = 0, i = \overline{1, m}$ , the Lagrange function have the form:

$$L(\mathbf{x}, \lambda_1, \dots, \lambda_m) = f(\mathbf{x}) + \sum_{i=1}^m \lambda_i g_i(\mathbf{x})$$

## → A.7 Useful Mathematical equations

### A.7.1 Combinatorics

Let consider  $N$  different objects. The number of ways it is possible to choose  $n$  objects out of the  $N$  set is:

$$\binom{N}{n} \equiv \frac{N!}{(N-n)!n!}$$

Considering the above expression then:

$$\binom{N-1}{n} = \frac{(N-n)!}{(N-n-1)!n!} = \frac{N-n}{N} \binom{N}{n} = \binom{N}{n} - \binom{N-1}{n-1}$$

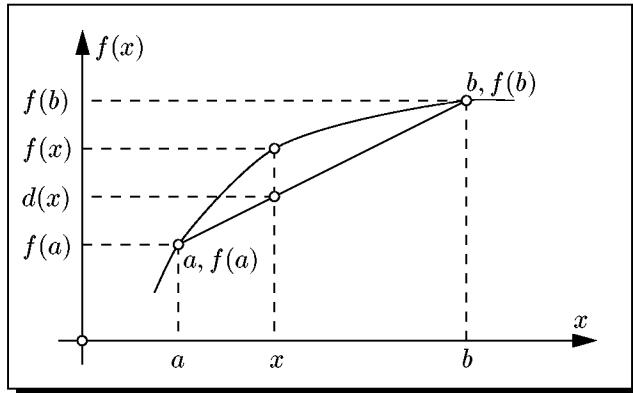
representing the recurrent formula:  $\binom{N+1}{n} = \binom{N}{n} + \binom{N}{n-1}$ .

### A.7.2 The Jensen's inequality

Let consider a convex function, i.e. a function for which all points from a chord, between any two points of the graph, are “below” the graph of the function. See figure A.4 on the facing page.

---

<sup>A.7</sup>See [Str81] pp. 200–201.



**Figure A.4:** A convex function  $f$ . A chord between arbitrary points  $a$  and  $b$  is under the graph of the function.

**Proposition A.7.1.** Considering a convex function  $f$ . a set of  $N \geq 2$  points  $\{x_i\}_{i=1,N}$

and a set of  $N$  numbers  $\{\alpha_i\}_{i=1,N}$  such that  $\sum_{i=1}^N \alpha_i = 1$  and  $\alpha_i \geq 0, \forall i$  then it is true that:

$$f\left(\sum_{i=1}^N \alpha_i x_i\right) \geq \sum_{i=1}^N \alpha_i f(x_i)$$

which is called Jensen's inequality.

Jensen's inequality

*Proof.* Let first consider two points  $a$  and  $b$  and two numbers  $0 \leq t \leq 1$  and  $1 - t$ ; such that they respect the condition of the theorem.

The points  $(a, f(a))$  and  $(b, f(b))$  defines a chord whose equation is (equation of a straight line passing through 2 points):

$$d(x) = \frac{bf(a) - af(b)}{b - a} + \frac{f(b) - f(a)}{b - a} x$$

then, for any  $x \in [a, b]$  it will be true that  $d(x) \leq f(x)$ . See also figure A.4.

By expressing  $x$  in the form of  $x = a + t(b - a)$ ,  $t \in [0, 1]$ , and replacing in the expression of  $d(x)$ , it gives:

$$f(a) + t[f(b) - f(a)] \leq f[a + t(b - a)] \Leftrightarrow f[(1 - t)a + tb] \geq (1 - t)f(a) + tf(b)$$

i.e. the Jensen's inequality holds for two numbers ( $t$  and  $1 - t$ ).

Let  $c$  be a point inside the  $[a, b]$  interval,  $f'_-(c)$  the derivative to the left of  $f(x)$  at  $c$  and  $f'_+(c)$  the derivative to the right of  $f(x)$  at the same point  $c$ . For a continuous derivative in  $c$  they are equal:  $f'_-(c) = f'_+(c) = f'(c)$ .

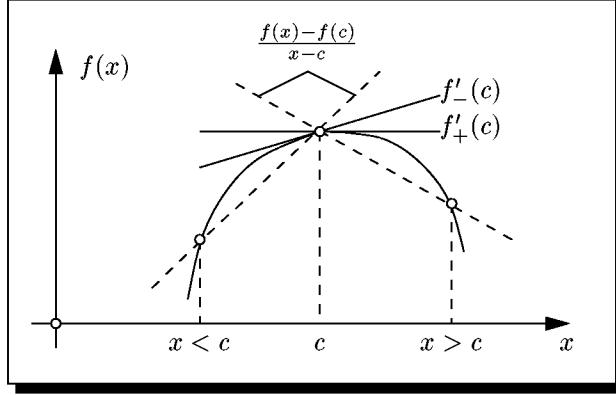
The expression  $\frac{f(x) - f(c)}{x - c}$  represents the tangent of the angle between the chord — passing through the points  $(x, f(x))$  and  $(c, f(c))$  — and the  $Ox$  axis. Similarly  $f'(c)$  represents the tangent of the angle made by the tangent.

Let  $m$  be a number  $f'_-(c) \leq m \leq f'_+(c)$ . Because  $f$  is convex then it is true that:

$$\frac{f(x) - f(c)}{x - c} \geq m \quad \text{for } x < c \quad \text{and} \quad \frac{f(x) - f(c)}{x - c} \leq m \quad \text{for } x > c$$

see also figure A.5 on the following page.

Finally, from the above equations, it is true that  $f(x) \leq m(x - c) + f(c), \forall x \in [a, b]$ .



**Figure A.5:** A convex function  $f$ , its derivatives in point  $c$  — to the left:  $f'_-(c)$ ; and to the right  $f'_+(c)$ . The chords for  $x < c$  and respectively for  $x > c$  are drawn with dashed lines. Parameters  $f'_-(c)$ ,  $f'_+(c)$  and  $\frac{f(x)-f(c)}{x-c}$  are the tangents of the angles between the tangents in  $c$ , respectively the chords, and the  $Ox$  axis.

Considering now a set of numbers  $\{x_i\}_{i=1,N} \in [a, b]$  and a set of parameters  $\{\alpha_i\}_{i=1,N} \in [0, 1]$  such that  $\sum_{i=1}^N \alpha_i = 1$  then:  $a \leq x_i \leq b \Rightarrow \alpha_i a \leq \alpha_i x_i \leq \alpha_i b$  and after a summation over  $i$ :  $a \leq \sum_{i=1}^N \alpha_i x_i \leq b$ .

Let  $c = \sum_{i=1}^N \alpha_i x_i \in [a, b]$ , then:

$$f(x_i) \leq m(x_i - c) + f(c) \Rightarrow \alpha_i f(x_i) \leq m(\alpha_i x_i - \alpha_i c) + \alpha_i f(c)$$

and the Jensen's inequality is obtained by summation over  $i = \overline{1, N}$ .  $\square$

### A.7.3 The Stirling Formula

**Proposition A.7.2.** For  $n \in \mathbb{N}^*$ ,  $n \gg 1$  it is true that:

$$\ln n! \simeq n \ln n - n = n \ln \frac{n}{e}$$

*Proof.* The Euler function  $\Gamma_E(x+1) = \int_0^\infty e^{-t} t^x dt$  (see (A.3)) is estimated for  $x \rightarrow \infty$  by the method of saddle point — the integrand is developed in series around maximum and then the superior order terms are neglected.

The derivative of integrand  $e^{-t} t^x = \exp(-t + x \ln t)$  is zero at maximum:

$$\frac{d}{dt} [\exp(-t + x \ln t)] = 0 \Leftrightarrow \left( -1 + \frac{x}{t} \right) \exp(-t + x \ln t) = 0$$

i.e. maximum is at point  $t = x$  (because the exp is never 0).

The exponent is developed in series around  $t = x$ :

$$-t + x \ln t = -x + x \ln x + \frac{(t-x)^2}{2!} \frac{d^2}{dt^2} (-t + x \ln t) \Big|_{t=x} + \dots \simeq -x + x \ln x - \frac{(t-x)^2}{2x}$$

because  $\frac{d}{dt}(-t + x \ln t) \Big|_{t=x} = 0$ , and just the first term from the series development is kept. Then:

$$\Gamma_E(x+1) \simeq \exp(x \ln x - x) \int_0^\infty \exp\left[-\frac{(t-x)^2}{2x}\right] dt$$

and the  $t \rightarrow t - x$  change of variable is performed and then:

$$\Gamma_E(x+1) \simeq \exp(x \ln x - x) \int_{-x}^\infty \exp\left(-\frac{t^2}{2x}\right) dt$$

and by using the limit  $x \rightarrow \infty$ :

$$\Gamma_E(x+1) \simeq \exp(x \ln x - x) \int_{-\infty}^\infty \exp\left(-\frac{t^2}{2x}\right) dt$$

and finally another change of variable  $s = \frac{t}{\sqrt{2x}}$  (see also section A.4):

$$\Gamma_E(x+1) \simeq \sqrt{2x} \exp(x \ln x - x) \int_{-\infty}^\infty e^{-s^2} ds = \sqrt{2x} \exp(x \ln x - x) \sqrt{\pi}$$

For large  $x = n \in \mathbb{N}^*$ :  $\ln \Gamma_E(n+1) = \ln n! \simeq n \ln n - n + \ln \sqrt{2\pi n} \simeq n \ln n - n$ .  $\square$

## ► A.8 Calculus of Variations

The change in a function  $f(x)$  when the variable  $x$  changes by a small amount  $\delta x$  is:

$$\delta f = \frac{df}{dx} \delta x + \mathcal{O}(\delta x^2)$$

where  $\mathcal{O}(\delta x^2)$  represents the superior terms (depending at most as  $\delta x^2$ ).

For a function of several variables  $f(x_1, \dots, x_n)$  the above expression becomes:

$$\delta f = \sum_{i=1}^n \frac{\partial f}{\partial x_i} \delta x_i + \mathcal{O}(\delta x^2)$$

A functional  $E[f]$  is a form which takes a *function* as variable and returns a value.

Considering an arbitrary *function*  $\delta f(x)$ , which have small values everywhere, then the variation of  $E$  is (by similarity with  $\delta f$ ,  $\sum \rightarrow \int$ ):

$$\delta E = E[f + \delta f] - E[f] = \int_X \frac{\delta E}{\delta f(x)} \delta f(x) dx + \mathcal{O}(\delta f^2) \quad (\text{A.4})$$

$X$  being the space of  $x$ .

**Proposition A.8.1. The fundamental lemma of the calculus of variations.** The condition of stationarity for  $E$ , to the lowest order in  $\delta f$ , involves the requirement  $\frac{\delta E}{\delta f} = 0$ , assuming the continuity of  $E$ .

*Proof.* Stationarity, to the lowest order, involves  $\delta E = 0$  and  $\mathcal{O}(\delta f^2) \simeq 0$ . (A.4) gives  $\int_X \frac{\delta E}{\delta f} \delta f dx = 0$ .

Let assume that there is an  $\tilde{x}$  for which  $\left. \frac{\delta E}{\delta f} \right|_{\tilde{x}} \neq 0$ . Then the continuity condition implies that there is a whole vicinity  $[\tilde{x}_1, \tilde{x}_2]$  of  $\tilde{x}$ , such that  $\left. \frac{\delta E}{\delta f} \right|_{x \in [\tilde{x}_1, \tilde{x}_2]} \neq 0$  and *keeps its sign*.

As  $\delta f$  is arbitrary then it is chosen as  $\delta f = \begin{cases} \neq 0 \text{ and keeps its sign} & x \in [\tilde{x}_1, \tilde{x}_2] \\ = 0 & \text{in rest} \end{cases}$ . Then, it follows that  $\int_X \frac{\delta E}{\delta f} \delta f dx \neq 0$  and the lemma assumptions are contradicted.  $\square$



### Remarks:

► A functional form, used in regularization theory, is:

$$E[f] = \int_X \left[ f^2 + \left( \frac{df}{dx} \right)^2 \right] dx \quad (\text{A.5})$$

Then, by replacing  $f$  with  $f + \delta f$  in the above equation:

$$E[f + \delta f] = E[f] + 2 \int_X \left[ f \delta f + \frac{df}{dx} \frac{d(\delta f)}{dx} \right] dx + \mathcal{O}(\delta f^2)$$

The term  $\int_X \frac{df}{dx} \frac{d(\delta f)}{dx} dx$ , integrated by parts, gives  $\left. \frac{df}{dx} \delta f \right|_{X \text{ boundaries}} - \frac{d^2 f}{dx^2} \delta f$ .

Considering the boundary term equal to 0 ( $\left. \frac{df}{dx} \right|_{X \text{ boundaries}} = 0$ ) then (A.5) becomes:

$$\delta E = \int_X \left[ 2f - 2 \frac{d^2 f}{dx^2} \right] \delta f dx + \mathcal{O}(\delta f^2)$$

By comparing the above equation with (A.4) it follows that  $\frac{\delta E}{\delta f} = 2f - 2 \frac{d^2 f}{dx^2}$ .

Defining the operator  $D \equiv \frac{d}{dx}$  then the functional and its derivative may be written as:

$$E = \int_X [f^2 + (Df)^2] dx \quad \text{and} \quad \frac{\delta E}{\delta f} = 2f + 2\hat{D}Df$$

where  $\hat{D} = -\frac{d}{df}$  is the adjoint operator of  $D$ .

## ► A.9 Principal Components

Let consider, into the space  $X$  of dimensionality  $n$ , a set of orthonormated vectors  $\{\mathbf{u}_i\}_{i=1,n}$ :

$$\mathbf{u}_i^T \mathbf{u}_j = \delta_{ij} \quad (\text{A.6})$$

and a set of vectors  $\{\mathbf{x}_i\}_{i=1,N}$ , having the mean  $\langle \mathbf{x} \rangle = \frac{1}{N} \sum_{i=1}^N \mathbf{x}_i$ .

❖  $X, n, N, \mathbf{u}_i, \mathbf{x}_i, \langle \mathbf{x} \rangle$   
❖  $E$

The residual error  $E$  is defined as:

$$E = \frac{1}{2} \sum_{i=K+1}^n \sum_{j=1}^N [\mathbf{u}_i^T (\mathbf{x}_j - \langle \mathbf{x} \rangle)]^2 , \quad K = \text{const. } \in [0, 1, \dots, n-1]$$

and may be written as (use  $(AB)^T = B^T A^T$  matrix property):

$$E = \frac{1}{2} \sum_{i=K+1}^n \mathbf{u}_i^T \Sigma \mathbf{u}_i \quad \text{where} \quad \Sigma = \sum_{i=1}^N (\mathbf{x}_i - \langle \mathbf{x} \rangle) (\mathbf{x}_i - \langle \mathbf{x} \rangle)^T$$

$\Sigma$  being the covariance matrix of  $\{\mathbf{x}_i\}$  set.

❖  $\Sigma$

The problem is to find the minima of  $E$ , with respect to  $\{\mathbf{u}_i\}$ , subject to constraints (A.6). This is done by using a set of Lagrange multipliers  $\{\mu_{ij}\}$ . Then the Lagrange function to minimize is:

$$L = E - \frac{1}{2} \sum_{i=K+1}^n \sum_{j=K+1}^n \mu_{ij} (\mathbf{u}_i^T \mathbf{u}_j - \delta_{ij})$$

(because of symmetry  $\mathbf{u}_i^T \mathbf{u}_j = \mathbf{u}_j^T \mathbf{u}_i$  then  $\mu_{ij} = \mu_{ji}$  and each term under the sums appears twice and thus the factor  $1/2$  is inserted to count each different term once).

Let consider the matrices:

$$U = (\mathbf{u}_{K+1} \quad \dots \quad \mathbf{u}_n) \quad \text{and} \quad M = \begin{pmatrix} \mu_{K+1,K+1} & \cdots & \mu_{K+1,n} \\ \vdots & \ddots & \vdots \\ \mu_{n,K+1} & \cdots & \mu_{n,n} \end{pmatrix}$$

$U$  being formed using  $\mathbf{u}_i$  as *columns* and  $M$  being a symmetrical matrix. Then the Lagrange function becomes:

$$L = \frac{1}{2} \text{Tr}(U^T \Sigma U) - \frac{1}{2} \text{Tr}[M(U^T U - I)]$$

Minimizing  $L$  with respect to  $\mathbf{u}_i$  means the set of conditions  $\frac{\partial L}{\partial u_{ij}} = 0$ , i.e. in matrix format (use  $(AB)^T = B^T A^T$  matrix property):

$$(\Sigma + \Sigma^T)U - U(M + M^T) = \tilde{0} \Rightarrow \Sigma U = UM$$

and, by using the property of orthogonality of  $\{\mathbf{u}_i\}$ , i.e.  $U^T U = I$ , it becomes:

$$U^T \Sigma U = M \tag{A.7}$$

One *particular* solution of the above equation is to choose  $\{\mathbf{u}_i\}$  to be the eigenvectors of  $\Sigma$  (as  $\Sigma$  is symmetrical it is possible to build an orthogonal system of eigenvectors) and to choose  $M$  as the diagonal matrix of eigenvalues (i.e.  $\mu_{ij} = \delta_{ij} \lambda_i$  where  $\lambda_i$  are the eigenvalues of  $\Sigma$ ).

An alternative is to consider the eigenvectors of  $M$ :  $\{\psi_i\}$  and the matrix  $\Psi$  built by using them as columns. Let  $\Lambda$  be the diagonal matrix of eigenvalues of  $M$ , i.e.  $\{\Lambda\}_{ij} = \delta_{ij} \lambda_i$ . As  $M$  is symmetric, it is possible to choose an orthogonal set  $\{\psi_i\}$ , i.e.  $\Psi^T \Psi = I$ .

❖  $\psi_i, \Psi, \Lambda, \lambda_i$

From eigenvector equation  $M\Psi = \Psi\Lambda$ , and by multiplying to the right by  $\Psi^T$ :  
 $\Lambda = \Psi^T M \Psi$ .

By replacing  $M$  from (A.7)

$$\Lambda = \Psi^T U^T \Sigma U \Psi = (U\Psi)^T \Sigma (U\Psi) = \tilde{U}^T \Sigma \tilde{U}$$

❖  $\tilde{U}$

where  $\tilde{U} = U\Psi$ .

This means that if there are a particular solution  $U$  to (A.7) then  $\tilde{U} = U\Psi$  is also solution:

$$\tilde{U}^T \Sigma \tilde{U} = M$$

and the residual error may be written as:

$$E = \frac{1}{2} \text{Tr}(U^T \Sigma U) = \frac{1}{2} \text{Tr}(M) = \frac{1}{2} \text{Tr}(\tilde{U}^T \Sigma \tilde{U})$$



#### Remarks:

- There is an invariance to the orthogonal transformation defined by  $\Psi$ .

## APPENDIX B

# Statistical Sidelines

## ► B.1 Probabilities

### B.1.1 Probabilities and Bayes Theorem

Let consider some pattern vectors  $\{\mathbf{x}_p\}$  and some classes  $\{\mathcal{C}_k\}$  these patterns have to be classified into.

**Definition B.1.1.** The **prior probability**  $P(\mathcal{C}_k)$  represents the probability of a pattern as being of class  $k$  while belonging to a very large set of samples:

$$P(\mathcal{C}_k) = \frac{\text{number of patterns of class } \mathcal{C}_k}{\text{total number of patterns}} \in [0, 1] \quad (\text{B.1})$$

when “total number of patterns”  $\rightarrow \infty$ .

**Definition B.1.2.** The **join probability**  $P(\mathcal{C}_k, X_\ell)$  represents the probability of a pattern as being of class  $k$  and — at the same time — the pattern vector being in the pattern subspace  $X_\ell \subset X$ ; the pattern belonging to a very large set of samples.

$$P(\mathcal{C}_k, X_\ell) = \frac{\text{number of patterns of class } \mathcal{C}_k \text{ with } \mathbf{x} \in X_\ell}{\text{total number of patterns}} \in [0, 1] \quad (\text{B.2})$$

when “total number of patterns”  $\rightarrow \infty$ .

#### Remarks:

- ➔ For discrete pattern spaces  $\mathbf{x} \in X_\ell$  may be replaced with  $\mathbf{x} \in \{X_{\ell 1}, \dots\}$ , where  $X_{\ell 1}, \dots$  are also pattern vectors.

B.1.1 See [Bis95] pp. 17–28 and [Rip96] pp. 19–20, 75.

- ➔ For *continuous* pattern spaces either  $X_\ell$  defines a volume in pattern space, in which the *point*  $\mathbf{x}$  should be, or a point in which case  $\mathbf{x} \in X_\ell$  is replaced with  $\mathbf{x} = X_\ell$  but, in this case,  $P(\mathcal{C}_k, X_\ell)$  represents an infinitesimal quantity.

❖  $P(X_\ell|\mathcal{C}_k)$

**Definition B.1.3.** The **class-conditional probability**  $P(X_\ell|\mathcal{C}_k)$  represents the probability for a pattern of class  $\mathcal{C}_k$  to have its pattern vector in the pattern subspace area defined by  $X_\ell$ .

$$P(X_\ell|\mathcal{C}_k) = \frac{\text{number of patterns of class } \mathcal{C}_k \text{ with } \mathbf{x} \in X_\ell}{\text{total number of patterns of class } \mathcal{C}_k} \in [0, 1] \quad (\text{B.3})$$

when “total number of patterns of class  $\mathcal{C}_k$ ”  $\rightarrow \infty$ .

❖  $P(X_\ell)$

**Definition B.1.4.** The **distribution probability**  $P(X_\ell)$  represents the probability of a pattern to have its associated vector  $\mathbf{x}$  in the subspace  $X_\ell$ .

$$P(X_\ell) = \frac{\text{number of patterns with } \mathbf{x} \in X_\ell}{\text{total number of patterns}} \in [0, 1] \quad (\text{B.4})$$

when “total number of patterns”  $\rightarrow \infty$ .

❖  $P(\mathcal{C}_k|X_\ell)$

**Definition B.1.5.** The **posterior probability**  $P(\mathcal{C}_k|X_\ell)$  represents the probability for a pattern which have its associated vector in subspace  $X_\ell$  to be of class  $\mathcal{C}_k$ :

$$P(\mathcal{C}_k|X_\ell) = \frac{\text{number of patterns with } \mathbf{x} \in X_\ell \text{ and of class } \mathcal{C}_k}{\text{total number of patterns with } \mathbf{x} \in X_\ell} \in [0, 1] \quad (\text{B.5})$$

when “total number of patterns with  $\mathbf{x} \in X_\ell$ ”  $\rightarrow \infty$ .



#### Remarks:

- ➔ Regarding  $X_\ell$  and probabilities same previous remarks apply.
- ➔ The prior probability refers to knowledge available *before* the pattern vector is known while the posterior probability refers to knowledge available *after* the pattern vector is known.
- ➔ By assigning a pattern to a class for which the posterior probability is maximum, the errors of misclassification are minimized.

**Theorem B.1.1. Bayes.** The posterior probability is the normalized product between prior and class-conditional probabilities:

$$P(\mathcal{C}_k|X_\ell) = \frac{P(X_\ell|\mathcal{C}_k) P(\mathcal{C}_k)}{P(X_\ell)} \quad (\text{B.6})$$

$P(X_\ell)$  being the normalization factor.

*Proof.* By multiplying (B.3) and (B.1) and comparing the result with (B.2) it follows that

$$P(\mathcal{C}_k, X_\ell) = P(X_\ell|\mathcal{C}_k) P(\mathcal{C}_k) \quad (\text{B.7})$$

similarly, from (B.5) and (B.4)

$$P(\mathcal{C}_k, X_\ell) = P(\mathcal{C}_k|X_\ell) P(X_\ell) \quad (\text{B.8})$$

The final result is obtained by comparing (B.7) and (B.8).  $\square$

**Remarks:**

- ➔ When working with classification, all patterns belong to a class: if a pattern can not be classified into a “normal” class there may be an outliers class containing all patterns not classifiable in any other class.
- ➔  $P(X_\ell)$  represents the normalization factor of  $P(X_\ell|\mathcal{C}_k) P(\mathcal{C}_k)$ .

*Proof.* Because each pattern should be classified into a class then

$$\sum_{k=1}^K P(\mathcal{C}_k|X_\ell) = 1 \quad (\text{B.9})$$

By using the Bayes theorem (B.6) in (B.9) the distribution probability may be expressed as:

$$P(X_\ell) = \sum_{k=1}^K P(X_\ell|\mathcal{C}_k) P(\mathcal{C}_k)$$

and then  $P(\mathcal{C}_k|X_\ell)$  is normalized, i.e.  $\sum_{k=1}^K P(\mathcal{C}_k|X_\ell) = 1$ .  $\square$

## B.1.2 Probability Density, Expectation and Variance

**Definition B.1.6.** The **probability density function**  $p(\mathbf{x})$  is the function for which

$$P(X_\ell) = \int_{X_\ell} p(\mathbf{x}) d\mathbf{x} \quad (\text{B.10})$$

probability  
density

where  $X_\ell$  is a pattern subspace.

Similarly, the following probability densities may be defined:

- **Joint probability density**  $p(\mathcal{C}_k, \mathbf{x})$ :

$$P(\mathcal{C}_k, X_\ell) = \int_{X_\ell} p(\mathcal{C}_k, \mathbf{x}) d\mathbf{x}$$

- **Class-conditional probability density**  $p(\mathbf{x}|\mathcal{C}_k)$ :

$$P(X_\ell|\mathcal{C}_k) = \int_{X_\ell} p(\mathbf{x}|\mathcal{C}_k) d\mathbf{x}$$

- **Posterior probability density**  $p(\mathcal{C}_k|\mathbf{x})$ :

$$P(\mathcal{C}_k|X_\ell) = \int_{X_\ell} p(\mathcal{C}_k|\mathbf{x}) d\mathbf{x}$$

**Definition B.1.7.** The **expectation** (expected value)  $\mathcal{E}\{Q\}$  of a function  $Q(\mathbf{x})$  is:

expectation

$$\mathcal{E}\{Q\} = \int_X Q(\mathbf{x}) p(\mathbf{x}) d\mathbf{x}$$

The **variance**  $\mathcal{V}\{Q\}$  of a function  $Q(\mathbf{x})$  is:

$$\mathcal{V}\{Q\} = \sqrt{\int_X [Q(\mathbf{x}) - \mathcal{E}\{Q\}]^2 p(\mathbf{x}) d\mathbf{x}}$$

**Proposition B.1.1.** *Using probability densities, the Bayes theorem B.1.1 may be written as:*

$$P(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k) P(\mathcal{C}_k)}{p(\mathbf{x})} \quad (\text{B.11})$$

*Proof.* For  $X_\ell$  being a point in the pattern space, (B.10) may be rewritten as:

$$dP(\mathbf{x}) = p(\mathbf{x}) dx$$

and similarly with the other types of probabilities; the final formula is obtained doing the replacement into (B.6).  $\square$

As in Bayes theorem,  $p(\mathbf{x})$  is a normalization factor for  $P(\mathcal{C}_k|\mathbf{x})$ .

*Proof.* The  $p(\mathbf{x})$  represents the probability density for the pattern vector of being  $\mathbf{x}$  no matter what class, it represents the sum of joint probability densities for the pattern vector of being  $\mathbf{x}$  and the pattern being of class  $\mathcal{C}_k$ , over all classes:

$$p(\mathbf{x}) = \sum_{k=1}^K p(\mathcal{C}_k, \mathbf{x}) = \sum_{k=1}^K p(\mathbf{x}|\mathcal{C}_k) P(\mathcal{C}_k)$$

and comparing to (B.11) it shows that  $P(\mathcal{C}_k|\mathbf{x})$  is normalized.  $\square$

### Remarks:

- The  $p(\mathbf{x}|\mathcal{C}_k)$  probability density may be seen as the *likelihood* probability that a pattern of class  $\mathcal{C}_k$  will have its pattern vector  $\mathbf{x}$ . The  $p(\mathbf{x})$  represents a normalization factor such that the sum of all posterior probabilities sum to one. Then the Bayes theorem may be expressed as:

$$\text{posterior probability} = \frac{\text{likelihood} \times \text{prior probability}}{\text{normalization factor}}$$

## → B.2 Modeling the Density of Probability

Let be a training set of classified patterns  $\{\mathbf{x}_p\}_{1,P}$ . The problem is to find a good approximation for probability density starting from the training set. Knowing it, from the Bayes theorem and Bayes rule<sup>1</sup> it is possible to built the device able to classify new input patterns.

There are several approaches to this problem:

- *The parametric method:* A specific functional form is assumed. There are a small number of tunable parameters which are optimized such that the model fits the training set. Disadvantages: there are limits to the capability of generalization: the functional forms chosen may not generalize well.
- *The non-parametric method:* The form of the probability density is determined from the data (no functional form is assumed). Disadvantages: The number of parameters grow with the size of the training set.
- *The semi-parametric method:* Tries to combine the above 2 methods by using a very general class of functional forms and by allowing the number of parameters to vary independently from the size of training set. *Feed-forward ANN* are of this type.

<sup>1</sup>See "Pattern Recognition" chapter.

### B.2.1 The Parametric Method

The parametric method uses functions with few tunable parameters to model the probability density. These functions are named distributions. The most widely used is the Gaussian due to its properties and good approximation of many real world processes.

#### **Gaussian Unidimensional**

For a unidimensional space the Gaussian distribution is defined as:

$$p(x) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x-\mu)^2}{2\sigma^2}\right] , \quad \sigma, \mu = \text{const.} \quad (\text{B.12})$$

This function have the following properties:

1.  $p(x)$  is normalized, i.e.  $\int_{-\infty}^{\infty} p(x) dx = 1$ .
2. Expected value of  $x$  is  $\mu$ , i.e.  $\mathcal{E}\{x\} = \int_{-\infty}^{\infty} xp(x) dx = \mu$ .
3. The variance (standard deviation) of  $x$  is  $\sigma$ , i.e.

$$\mathcal{V}\{x\} = \sqrt{\int_{-\infty}^{\infty} [x - \mathcal{E}\{x\}]^2 p(x) dx} = \sigma$$

*Proof.* 1. By making the change of variable:

$$y = \frac{x - \mu}{\sqrt{2}\sigma} \Leftrightarrow dy = \frac{dx}{\sqrt{2}\sigma}$$

and because  $\int_{-\infty}^{\infty} e^{-y^2} dy = \sqrt{\pi}$  (see the mathematical appendix) then  $\int_{-\infty}^{\infty} p(x) dx = 1$  i.e. the probability density is normalized (this is the role of the  $\frac{1}{\sqrt{2\pi}\sigma}$  factor) as it should be, because the probability of finding  $x$  in the whole space is 1 (certainty).

2. The *mean* value of  $x$  is the expectation (see definition B.1.7)

$$\mathcal{E}\{x\} = \int_{-\infty}^{\infty} xp(x) dx$$

and by making the same change of variable as above ( $x = \sqrt{2}\sigma y + \mu$ )

$$\begin{aligned} \mathcal{E}\{x\} &= \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} (\sqrt{2}\sigma y + \mu) e^{-y^2} \sqrt{2}\sigma dy \\ &= \sqrt{\frac{2}{\pi}} \sigma \int_{-\infty}^{\infty} ye^{-y^2} dy + \frac{\mu}{\sqrt{\pi}} \int_{-\infty}^{\infty} e^{-y^2} dy = \mu \end{aligned}$$

because the first integral  $\int_{-\infty}^{\infty} ye^{-y^2} dy$  is 0 — the integrand is an odd function (the value for  $-x$  is minus the value for  $x$ ) and the integration interval is symmetric relatively to origin; and the second integral is  $\sqrt{\pi}$  (see the mathematical appendixes).

B.2.1 See [Bis95] pp. 34–49 and [Rip96] pp. 21, 30–31.

3. The variance is (see definition B.1.7):

$$\mathcal{V}\{x\} = \int_{-\infty}^{\infty} (x - \mu)^2 p(x) dx$$

(as  $\mathcal{E}\{x\} = \mu$ ) and same change of variable leads to an integral solvable by parts:

$$\begin{aligned} \mathcal{V}\{x\} &= \frac{1}{\sqrt{2\pi}\sigma} \int_{-\infty}^{\infty} (x - \mu)^2 \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right) dx = \frac{2\sigma^2}{\sqrt{\pi}} \int_{-\infty}^{\infty} y^2 e^{-y^2} dy \\ &= -\frac{\sigma^2}{\sqrt{\pi}} \int_{-\infty}^{\infty} y d(e^{-y^2}) = -\frac{\sigma^2}{\sqrt{\pi}} y e^{-y^2} \Big|_{-\infty}^{\infty} + \frac{\sigma^2}{\sqrt{\pi}} \int_{-\infty}^{\infty} e^{-y^2} dy = \sigma^2 \quad \square \end{aligned}$$



### Remarks:

- To apply to a stochastic process calculate the mean and variance of training set, then replace in (B.12) to find an *approximation* of real probability density by an Gaussian.

### Gaussian Multidimensional

In general, into a  $N$ -dimensional space, the Gaussian distribution is defined as:

$$p(\mathbf{x}) = \frac{1}{(2\pi)^{N/2} \sqrt{|\Sigma|}} \exp\left(-\frac{(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})}{2}\right) \quad (\text{B.13})$$

where  $\boldsymbol{\mu}$  is a  $N$ -dimensional vector and  $\Sigma$  is a  $N \times N$  matrix, symmetric and inversable.

This function have the following properties:

❖  $d\mathbf{x}$

1.  $p(\mathbf{x})$  is normalized, i.e.  $\int_{\mathbb{R}^N} p(\mathbf{x}) d\mathbf{x} = 1$ , where  $d\mathbf{x} = dx_1 \dots dx_N$ .
2. Expected value of  $\mathbf{x}$  is  $\boldsymbol{\mu}$ , i.e.  $\mathcal{E}\{\mathbf{x}\} = \int_{\mathbb{R}^N} \mathbf{x} p(\mathbf{x}) d\mathbf{x} = \boldsymbol{\mu}$ .
3. Expected value of  $(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T$  is  $\Sigma$ , i.e.

$$\mathcal{E}\{(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T\} = \Sigma$$

*Proof.* 1. Because  $\det(\Sigma^{-1}) \det(\Sigma) = \det(\Sigma^{-1}\Sigma) = \det(I) = 1$  then  $\det(\Sigma^{-1}) = \frac{1}{\det(\Sigma)}$ , and, by making the change of variable  $\tilde{\mathbf{x}} = \mathbf{x} - \boldsymbol{\mu}$ , the integral become:

$$\int_{\mathbb{R}^N} p(\mathbf{x}) d\mathbf{x} = \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \int_{\mathbb{R}^N} \exp\left(-\frac{\tilde{\mathbf{x}}^T \Sigma^{-1} \tilde{\mathbf{x}}}{2}\right) d\tilde{\mathbf{x}} = 1$$

(see also the mathematical appendix regarding Gaussian integrals).

2. The mean value for  $\mathbf{x}$  is the *expectation*

$$\mathcal{E}\{\mathbf{x}\} = \int_{\mathbb{R}^N} \mathbf{x} p(\mathbf{x}) d\mathbf{x} = \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \int_{\mathbb{R}^N} \mathbf{x} \exp\left[-\frac{(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})}{2}\right] d\mathbf{x}$$

and, by making the same change of variable as above  $\tilde{\mathbf{x}} = \mathbf{x} - \boldsymbol{\mu}$ , it becomes

$$\mathcal{E}\{\mathbf{x}\} = \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \left[ \int_{\mathbb{R}^N} \tilde{\mathbf{x}} \exp\left(-\frac{\tilde{\mathbf{x}}^T \Sigma^{-1} \tilde{\mathbf{x}}}{2}\right) d\tilde{\mathbf{x}} + \boldsymbol{\mu} \int_{\mathbb{R}^N} \exp\left(-\frac{\tilde{\mathbf{x}}^T \Sigma^{-1} \tilde{\mathbf{x}}}{2}\right) d\tilde{\mathbf{x}} \right]$$

The function  $\exp\left(-\frac{\tilde{x}^T \Sigma^{-1} \tilde{x}}{2}\right)$  is even (same value for  $\tilde{x}$  and  $-\tilde{x}$ ), such that the integrand of the first integral is an odd function and, the interval of integration being symmetric to the origin, the integral is zero. The second integral have the value  $\frac{1}{(2\pi)^n/2 \sqrt{|\Sigma|}}$  and, as  $|\Sigma^{-1}| = \frac{1}{|\Sigma|}$ , then finally  $\mathcal{E}\{\mathbf{x}\} = \boldsymbol{\mu}$ .

3. The expectation value for that matrix is:

$$\begin{aligned}\mathcal{E}\{(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T\} &= \int_{\mathbb{R}^N} (\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T p(\mathbf{x}) d\mathbf{x} \\ &= \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \int_{\mathbb{R}^N} (\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T \exp\left(-\frac{(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})}{2}\right) d\mathbf{x}\end{aligned}$$

Same change of variable as above  $\tilde{\mathbf{x}} = \mathbf{x} - \boldsymbol{\mu}$  and:

$$\mathcal{E}\{(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T\} = \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \int_{\mathbb{R}^N} \tilde{\mathbf{x}} \tilde{\mathbf{x}}^T \exp\left(-\frac{\tilde{\mathbf{x}}^T \Sigma^{-1} \tilde{\mathbf{x}}}{2}\right) d\tilde{\mathbf{x}}$$

Let  $\{\mathbf{u}_i\}_{i=1,N}$  be the eigenvectors and  $\{\lambda_i\}_{i=1,N}$  the eigenvalues of  $\Sigma$  such that  $\Sigma \mathbf{u}_i = \lambda_i \mathbf{u}_i$ . The consider the set of eigenvectors chosen such that is orthonormalized (see the mathematical appendix regarding the properties of symmetrical matrices).  $\diamond \mathbf{u}_i, \lambda_i$

Let be  $U$  the matrix build using the eigenvectors of  $\Sigma$  as columns and  $\Lambda$  the matrix of eigenvalues  $\Lambda_{ij} = \delta_{ij} \lambda_i$  ( $\delta_{ij}$  being the Kronecker symbol), i.e.  $\Lambda$  is a diagonal matrix with eigenvalues on main diagonal and 0 in rest.  $\diamond U, \Lambda$

By multiplying  $\mathcal{E}\{(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T\}$  with  $U^T$  to the left and with  $U$  to the right and because the set of eigenvectors is orthonormalized then  $U^T U = I \Rightarrow U^{-1} = U^T \Rightarrow U U^T = I$  and it gets that:

$$U^T \mathcal{E}\{(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T\} U = \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \int_{\mathbb{R}^N} U^T \tilde{\mathbf{x}} \tilde{\mathbf{x}}^T U \exp\left(-\frac{\tilde{\mathbf{x}}^T U U^T \Sigma^{-1} U U^T \tilde{\mathbf{x}}}{2}\right) d\tilde{\mathbf{x}}$$

A new change of variable is performed:  $\mathbf{y} = U^T \tilde{\mathbf{x}}$  and then  $\mathbf{y}^T = \tilde{\mathbf{x}}^T U$  and  $d\mathbf{y} = d\tilde{\mathbf{x}}$  — because this transformation conserve the distances and angles. Also  $\Sigma^{-1}$  have the same eigenvectors as  $\Sigma$ , the eigenvalues being  $\left\{\frac{1}{\lambda_i}\right\}_{i=1,N}$  and respectively the eigenmatrix  $\Lambda^{-1}$  is defined by  $\Lambda_{ij}^{-1} = \delta_{ij} \frac{1}{\lambda_i}$ . Then:

$$\begin{aligned}U^T \mathcal{E}\{(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T\} U &= \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \int_{\mathbb{R}^N} \mathbf{y} \mathbf{y}^T \exp\left(-\frac{\mathbf{y}^T \Lambda^{-1} \mathbf{y}}{2}\right) d\mathbf{y} \\ &= \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \int_{\mathbb{R}^N} \mathbf{y} \mathbf{y}^T \exp\left(-\sum_{i=1}^N \frac{y_i^2}{2\lambda_i}\right) d\mathbf{y} = \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \int_{\mathbb{R}^N} \mathbf{y} \mathbf{y}^T \prod_{i=1}^N \exp\left(-\frac{y_i^2}{2\lambda_i}\right) d\mathbf{y}\end{aligned}$$

i.e. the integrals now may be decoupled. First the  $\mathbf{y} \mathbf{y}^T$  is of the form

$$\mathbf{y} \mathbf{y}^T = \begin{pmatrix} y_1 \\ \vdots \\ y_N \end{pmatrix} (y_1 \quad \cdots \quad y_N) = \begin{pmatrix} y_1^2 & \cdots & y_1 y_N \\ \vdots & \ddots & \\ y_N y_1 & \cdots & y_N^2 \end{pmatrix}$$

Each element of the matrix  $\{U^T \mathcal{E}\{(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^{(t)}\} U\}_{i,j}$  is computed separately. There are two cases: non-diagonal and diagonal elements. Also  $\int_{\mathbb{R}} \Leftrightarrow \int_{-\infty}^{\infty}$ .

The non-diagonal elements are

$$\begin{aligned}&\{U^T \mathcal{E}\{(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T\} U\}_{i,j} \\ &= \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \left[ \prod_{\substack{k=1 \\ k \neq i,j}}^N \int_{-\infty}^{\infty} \exp\left(-\frac{y_k^2}{2\lambda_k}\right) dy_k \right] \left[ \int_{-\infty}^{\infty} y_i \exp\left(-\frac{y_i^2}{2\lambda_i}\right) dy_i \right] \left[ \int_{-\infty}^{\infty} y_j \exp\left(-\frac{y_j^2}{2\lambda_j}\right) dy_j \right]\end{aligned}$$

and because the function  $y_i \exp\left(-\frac{y_i^2}{2\lambda_i}\right)$  is odd, the corresponding integrals are 0 such that

$$\{U^T \mathcal{E}\{(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T\}U\}_{i,j} = 0$$

The diagonal elements are

$$\{U^T \mathcal{E}\{(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T\}U\}_{i,i} = \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \left[ \prod_{\substack{k=1 \\ k \neq i}}^N \int_{-\infty}^{\infty} \exp\left(-\frac{y_k^2}{2\lambda_k}\right) dy_k \right] \left[ \int_{-\infty}^{\infty} y_i^2 \exp\left(-\frac{y_i^2}{2\lambda_i}\right) dy_i \right]$$

and the individual integrals appearing above are:

$$\int_{-\infty}^{\infty} \exp\left(-\frac{y_k^2}{2\lambda_k}\right) dy_k = \sqrt{2\pi\lambda_k} \quad \text{and} \quad \int_{-\infty}^{\infty} y_i^2 \exp\left(-\frac{y_i^2}{2\lambda_i}\right) dy_i = \lambda_i \sqrt{2\pi\lambda_i}$$

(calculated same way as for the unidimensional case) and  $|\Sigma| = \prod_{i=1}^N \lambda_i$ ; so finally

$$U^T \mathcal{E}\{(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^{(t)}\}U = \Lambda \quad \Rightarrow \quad \mathcal{E}[(\mathbf{x} - \boldsymbol{\mu})(\mathbf{x} - \boldsymbol{\mu})^T] = U\Lambda U^T = \Sigma \quad \square$$



### Remarks:

- By applying the transformation (equivalent to a rotation)

$$\tilde{\mathbf{x}} = U^T(\mathbf{x} - \boldsymbol{\mu})$$

the probability distribution  $p(\mathbf{x})$  becomes (similar to the above calculations)

$$\begin{aligned} p(\mathbf{x}) &= \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \exp\left(-\frac{\tilde{\mathbf{x}}^T U^T \Sigma^{-1} U \tilde{\mathbf{x}}}{2}\right) = \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \exp\left(-\frac{\tilde{\mathbf{x}}^T \Lambda^{-1} \tilde{\mathbf{x}}}{2}\right) = \\ &= \frac{\sqrt{|\Sigma^{-1}|}}{(2\pi)^{N/2}} \exp\left(-\sum_{i=1}^N \frac{\tilde{x}_i^2}{2\lambda_i}\right) = \prod_{i=1}^N p_i \end{aligned}$$

where  $p_i$  is

$$p_i = \frac{1}{(2\pi)^{N/2} \sqrt{\lambda_i}} \exp\left(-\sum_{i=1}^N \frac{\tilde{x}_i^2}{2\lambda_i}\right)$$

and then the probabilities are decoupled, i.e. the components of  $\mathbf{x}$  are *statistically independent*.

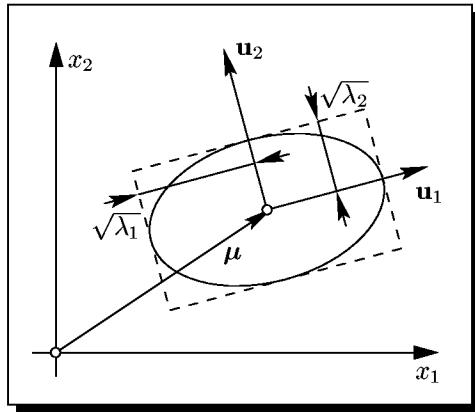
- The expression

$$\Delta = \sqrt{(\mathbf{x} - \boldsymbol{\mu})^T \Sigma^{-1} (\mathbf{x} - \boldsymbol{\mu})}$$

Mahalanobis distance

is called *Mahalanobis distance* between vectors  $\mathbf{x}$  and  $\boldsymbol{\mu}$ .

- For  $\Delta = \text{const.}$  the probability density is constant so  $\Delta$  represents surfaces of equal probability for  $\mathbf{x}$ .



**Figure B.1:** Equal probability density for Gauss probability density in two dimensions. The ellipse represents all points where  $p(x_1, x_2) = e^{-1/2}$ . The  $\mu$  vector points to the center of the ellipse.

- By applying the transformation  $\tilde{\mathbf{x}} = U^T(\mathbf{x} - \mu)$  the Mahalanobis distance becomes:

$$\Delta^2 = \sum_{i=1}^N \lambda_i \tilde{\mathbf{x}}_i^2$$

i.e. the surfaces of equal probability are hyper-ellipsoids. The main axes of the ellipsoid are proportional with  $\sqrt{\lambda_i}$ . The  $\mu$  vector points to the location of highest probability density. See figure B.1

The transformation  $\tilde{\mathbf{x}} = U^T(\mathbf{x} - \mu)$  is from  $(x_1, x_2)$  to  $\{\mathbf{u}_1, \mathbf{u}_2\}$ , i.e. a translation by  $\mu$  then a rotation such that  $\{\mathbf{u}_1, \mathbf{u}_2\}$  becomes the new set of versors.

The probability density for a two dimensional pattern space is shown in figure B.2 on the following page.

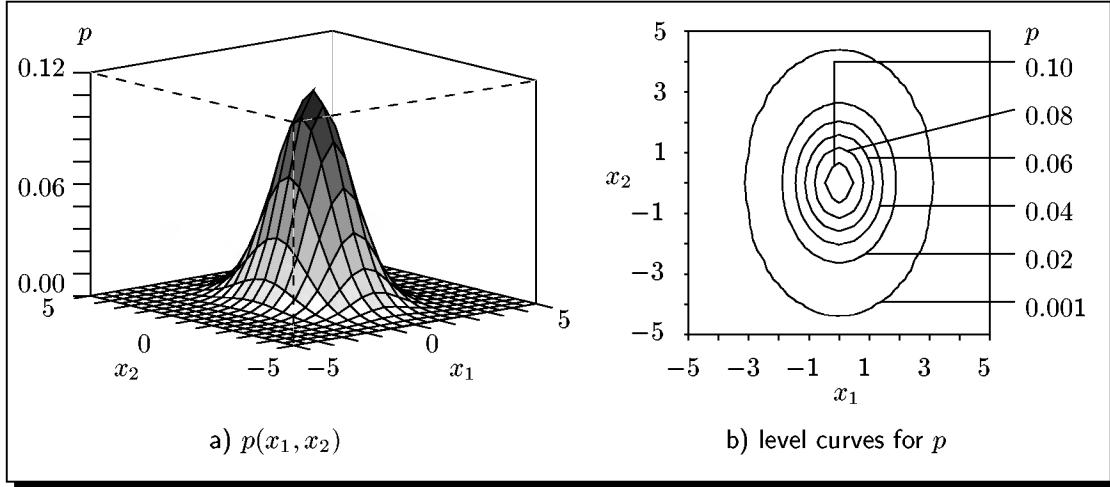
- The number of parameters defining the Gaussian distribution is  $1 + \dots + N = \frac{N(N+1)}{2}$  for  $\Sigma$  (symmetrical matrix) plus  $N$  parameters for  $\mu$  so the Gaussian distribution is completely defined by  $\frac{N(N+3)}{2}$  number of parameters.

**Definition B.2.1.** A  $N$ -dimensional vector  $\mathbf{x}$  it is said to be normal, i.e.  $\mathbf{x} \sim N_N\{\mu, \Sigma\}$  if it have a Gaussian distribution of the form (B.13) with a mean  $\mu$  and covariance matrix  $\Sigma$ .  $\diamond N_N\{\mu, \Sigma\}$

### Remarks:

- Let consider a set of several classes  $1, \dots, K$  such that each may be modelled using a multidimensional Gaussian, each with its own  $\mu_k$  and  $\Sigma_k$ . Considering that the prior probabilities  $P(\mathcal{C}_k)$  are equal then the biggest posterior probability for a given vector  $\mathbf{x}$  is the one corresponding to the minimum of the Mahalanobis distance  $\Delta_k \mathbf{x} = \min_{\ell} \Delta_{\ell} \mathbf{x}$ . This type of classification is named *linear discriminant analysis*.

linear discriminant analysis



**Figure B.2:** The Gaussian probability density function in two dimensional pattern space for  $\mu = \hat{0}$  and  $\Sigma = \begin{pmatrix} 1 & 0 \\ 0 & 2 \end{pmatrix}$ . The function was sampled in  $\Delta x_{1,2} = 0.5$  steps.

- Considering the covariance matrix equal to the identity matrix, i.e.  $\Sigma = I$ , then the Mahalanobis distance reduces to the simple Euclidean distance (see mathematical appendix) and then the pattern vectors  $\mathbf{x}$  are simply classified to the class  $\mathcal{C}_k$  with the closest mean  $\mu_k$ .

### B.2.2 The non-parametric method

Non-parametric method try to solve the problem of finding a probability distribution of data using a training set and without making any assumption about the form of the distribution function.

#### Histograms

❖  $X_k$

In the histogram method the pattern space is divided in subspaces (areas)  $X_k$  and the probability density is estimated from the number of patterns in each area. See figure B.3 on the next page.

The size of  $X_k$  determine the model complexity: if it is too large then it fits poorly the data, if it is too small then it overfits the exceptions/noise which may be present in the training set, i.e. the size of  $X_k$  controls the *model complexity*. See figure B.3 on the facing page.

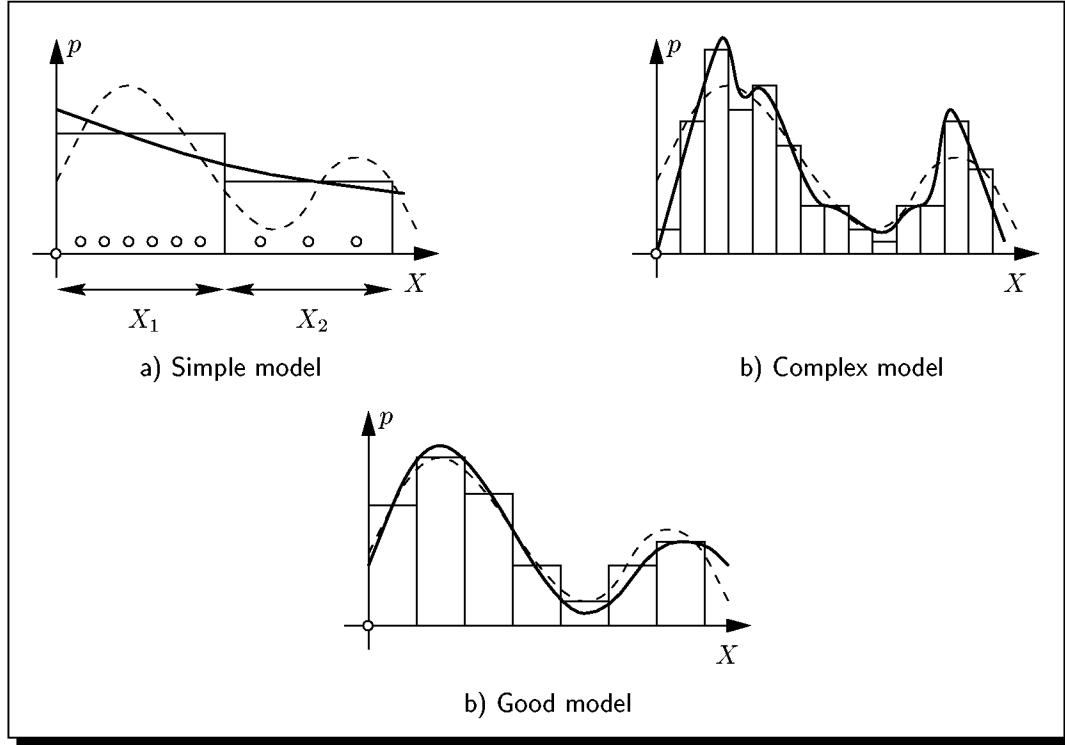
❖  $K$

Let take one area  $X_k$  and let  $K$  be the number of patterns from the training set which are in the area  $X_k$ .

Assuming a sufficiently large number of patterns in the training set (such that the training set is statistically significant) then the probability that a pattern will fall in the area  $X_k$  is approximatively:

$$P(X_k) \simeq \frac{K}{P}$$

B.2.2 See [Bis95] pp. 49–59 and [Rip96] pp. 190, 201–206.



**Figure B.3:** Histograms: probability density  $p$  versus pattern space  $X$ . The true probability density is shown with a dotted line. The estimated probability density is shown with a thick line. The rectangles represent the number of patterns from the training set which falls in the designated regions  $X_k$ . The small circles represent the patterns.

On the other hand, assuming that the probability density is approximatively constant to  $\tilde{p}(\mathbf{x})$  in all points from  $X_k$  then:

$$P(X_k) = \int_{X_k} p(\mathbf{x}) d\mathbf{x} \simeq \tilde{p}(\mathbf{x}) \int_{X_k} d\mathbf{x} = \tilde{p}(\mathbf{x}) V_{X_k}$$

where  $V_{X_k}$  is the volume of the pattern area  $X_k$ . Finally:

$$p(\mathbf{x}) \simeq \tilde{p}(\mathbf{x}) = \frac{K_P}{PV_{X_k}} \quad (\text{B.14})$$

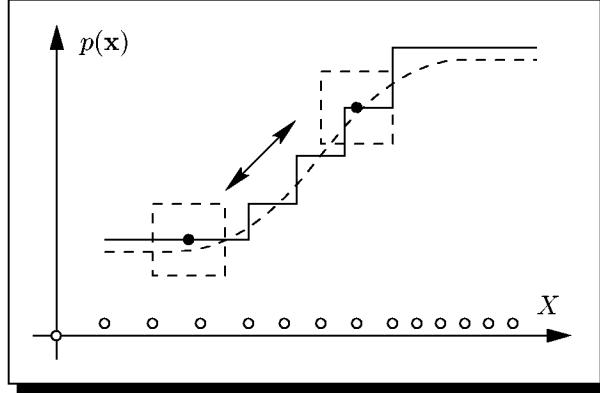
where  $\tilde{p}(\mathbf{x})$  is the estimated probability density.

❖  $V_{X_k}$

❖  $\tilde{p}(\mathbf{x})$

To solve (B.14) further, two approaches may be taken.

- $X_k$ , respectively  $V_{X_k}$  is fixed and  $K_P$  is counted — this represents the *kernel based method* kernel method
- $K$  is fixed and the  $V_{X_k}$  is calculated — this represents the *K-nearest-neighbors method* K nearest neighbors



**Figure B.4:** The kernel-based method. The dashed line is the real probability density  $p(\mathbf{x})$ ; the continuous line is the estimated probability density  $\tilde{p}(\mathbf{x})$  based on counting the patterns in the hypercube surrounding the current point and represented by a dashed rectangle.

### The kernel-based method

Let  $X_k$  be a hypercube having the side of length  $\ell$  and being centered in  $\mathbf{x}$ . Then its volume is  $V_{X_k} = \ell^N$  ( $N$  being the dimension of the pattern space).

❖  $H(\mathbf{x})$

The following *kernel function*<sup>2</sup>  $H(\mathbf{x})$  is defined

$$H(\mathbf{x}) = \begin{cases} 1 & \text{if } x_i < \frac{1}{2} \text{ for } i = 1, N \\ 0 & \text{otherwise} \end{cases}$$

such that  $H(\mathbf{x})$  is 1 if the point  $\mathbf{x}$  is inside the unit hypercube centered in the origin and 0 otherwise. Then  $H\left(\frac{\mathbf{x}-\mathbf{x}_p}{\ell}\right)$  will indicate if the  $\mathbf{x}_p$  point from the training set is in the hypercube  $X_k$  or not. The total number of patterns falling in  $X_k$  is:

$$K = \sum_{p=1}^P H\left(\frac{\mathbf{x}-\mathbf{x}_p}{\ell}\right)$$

and then, the estimate for the probability density is:

$$\tilde{p}(\mathbf{x}) = \frac{1}{P} \sum_{p=1}^P \frac{1}{\ell^N} H\left(\frac{\mathbf{x}-\mathbf{x}_p}{\ell}\right)$$

this may be visualized as a sliding hypercube in the pattern space, centered in the current point  $\mathbf{x}$ . While moving it, some of the  $\mathbf{x}_p$  points will enter it while others will leave it such that — unless the total number remains constant —  $\tilde{p}(\mathbf{x})$  will have a step jumps. See figure B.4.

The function  $\tilde{p}(\mathbf{x})$  may be “smoothened” by replacing the kernel function with a continuous

<sup>2</sup>Known also as the *Parzen window*.

function:

$$H(\mathbf{x}) = \frac{1}{(2\pi)^{N/2}} \exp\left(-\frac{\|\mathbf{x}\|^2}{2}\right)$$

and then the estimation of the probability density becomes:

$$\tilde{p}(\mathbf{x}) = \frac{1}{P} \sum_{p=1}^P \frac{1}{(2\pi\ell^2)^{N/2}} \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}_p\|^2}{2\ell^2}\right)$$

In general any function bounded by the conditions:

$$H(\mathbf{x}) \geq 0 \quad \text{and} \quad \int_X H(\mathbf{x}) d\mathbf{x} = 1$$

is suitable to be used as a kernel function.

Let examine the expectation of the estimated probability density, considering that  $P \rightarrow \infty$ :

$$\mathcal{E}\{\tilde{p}(\mathbf{x})\} = \frac{1}{P} \sum_{p=1}^P \mathcal{E}\left\{\frac{1}{\ell^N} H\left(\frac{\mathbf{x} - \mathbf{x}_p}{\ell}\right)\right\} \rightarrow \int_X \frac{1}{\ell^N} H\left(\frac{\mathbf{x} - \mathbf{x}'}{\ell}\right) p(\mathbf{x}') d\mathbf{x}'$$

(where  $\mathbf{x}'$  represents the integral variable).

This formula shows that the expectation of the estimated probability density is a convolution of the (true) probability density with the kernel function.

For  $\ell \rightarrow 0$  and  $P \rightarrow \infty$  the estimated probability density approaches the true one while the kernel function approaches the  $\delta$ -Dirac function.

### The K-nearest-neighbors method

Let  $K$  be fixed and  $X_k$  a hyper-sphere centered in  $\mathbf{x}$  and with variable radius, such that it will contain *always* the same number  $K$  of vectors from the training set. See figure B.5 on the following page.

The estimation of probability density is found from:

$$\tilde{p}(\mathbf{x}) = \frac{K}{PV_{X_k}}$$

The volume  $V_{(N)}$  of a sphere, of radius  $r$  in the  $N$ -dimensional space is:

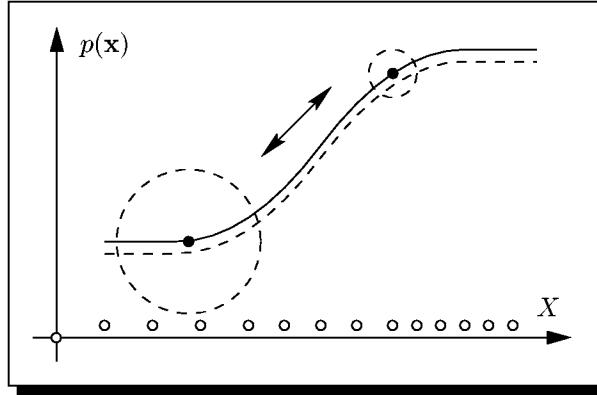
❖  $V_{(N)}$

$$V_{(N)} = \frac{\pi^{N/2}}{\Gamma_E\left(\frac{N}{2} + 1\right)} r^N \quad \text{where} \quad \Gamma_E(x) = \int_0^\infty e^{-t} t^{x-1} dt$$

$\Gamma_E$  being the Euler function, see the mathematical appendix.

Let consider a set of classes and a training set. Let  $P_k$  be the number of patterns of class  $C_k$  in the training set such that  $\sum_k P_k = P$ . Let  $K_k$  be the number of patterns of class  $C_k$  in the hyper-sphere of volume  $V$ . Then

$$p(\mathbf{x}|C_k) = \frac{K_k}{P_k V} \quad , \quad p(\mathbf{x}) = \frac{K}{PV} \quad \text{and} \quad p(C_k) = \frac{P_k}{P}$$



**Figure B.5:** The  $K$ -nearest-neighbors based method. The dashed line is the real probability density  $p(\mathbf{x})$ ; the continuous line is the estimated probability density  $\tilde{p}(\mathbf{x})$  based on estimating the volume of the hyper-sphere with variable radius and represented by a dashed circle (the hyper-sphere is defined by fixing the  $K$  number).

From the Bayes theorem

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k) p(\mathcal{C}_k)}{p(\mathbf{x})} = \frac{K_k}{K}$$

which is known as the  *$K$ -nearest-neighbors classification rule*. This means that once the volume of the hyper-sphere was established (by fixing  $K$ ) a new pattern  $\mathbf{x}$  is classified as being of that class which have most representatives ( $K_k$ ) included into the hyper-sphere, i.e. to that class  $\mathcal{C}_k$  for which:

$$p(\mathcal{C}_k|\mathbf{x}) = \max_{\ell} p(\mathcal{C}_{\ell}|\mathbf{x})$$

(according to the Bayes rule, see “Pattern Recognition” chapter).



#### Remarks:

- The parameters governing the smoothness of the histogram method are  $V$  for kernel based procedure and  $K$  for  $K$  nearest neighbors procedure. If this tunable parameter is too large then an excessive smoothness occurs and the resulting model is too *simple*. If the parameter is chosen too small then the *variance* of the model is too large, the model will approximate well the probability for the training set but will have poor generalization, the model will be too *complex*.

### B.2.3 The Semi–Parametric Method

#### The mixture model

The mixture model consider that the probability density is a superposition of probability densities, each of them having a different weight by which contributes to the total.

B.2.3 See [Bis95] pp. 59–73.

The procedure below is repeated for each class  $\mathcal{C}_k$  in turn.

Considering a superposition of  $M$  probability densities then:

❖  $M$

$$p(\mathbf{x}) = \sum_{m=1}^M p(\mathbf{x}|m)P(m) \quad (\text{B.15})$$

where  $p(\mathbf{x}|m)$  represents the probability density of pattern as being  $\mathbf{x}$ , from all patterns generated by component  $m$  of the superposition, and the weight is  $P(m)$ , the prior probability of the pattern  $\mathbf{x}$  having been created by the component  $m$  of the superposition.  $M$  becomes also a parameter of the model.

❖  $p(\mathbf{x}|m)$ ,  $P(m)$

All these probabilities have to be normalized:

$$\sum_{m=1}^M P(m) = 1 \quad , \quad P(m) \in [0, 1] \quad \text{and} \quad \int_X p(\mathbf{x}|m) d\mathbf{x} = 1$$



#### Remarks:

- ➔ The training set have the patterns classified in classes but does *not* have the patterns classified by the superposition components  $m$ , i.e. *the training set is incomplete* and the complete model have to provide a mean to determine this.

incomplete training set

The posterior probability is given by the Bayesian theorem and is normalized:

$$P(m|\mathbf{x}) = \frac{p(\mathbf{x}|m)P(m)}{p(\mathbf{x})} \quad \text{and} \quad \sum_{m=1}^M P(m|\mathbf{x}) = 1 \quad (\text{B.16})$$

The problem is to determinate the components of the superposition of probability densities.



#### Remarks:

- ➔ One possibility is to model the conditional probability densities  $p(\mathbf{x}|m)$  as Gaussian, defined by the parameters  $\mu_m$  and  $\Sigma_m = \sigma_m I$ :

$$p(\mathbf{x}|m) = \frac{1}{(2\pi\sigma_m^2)^{N/2}} \exp\left(-\frac{\|\mathbf{x} - \mu_m\|^2}{2\sigma_m^2}\right) \quad (\text{B.17})$$

and then a search for optimal parameters  $\mu$  and  $\sigma$  may be done.

- ➔ To avoid singularities the conditions:

$$\mu_m \neq \mathbf{x}_p \quad m = \overline{1, M}, p = \overline{1, P} \quad \text{and} \quad \sigma_m \neq 0 \quad m = \overline{1, M}$$

have to be imposed ( $\mathbf{x}_p$  are the training vectors).

#### **The maximum likelihood method**

The parameters are searched by maximizing the likelihood function, defined as<sup>3</sup>:

$$\mathcal{L} = \prod_{p=1}^P p(\mathbf{x}_p|W) \quad (\text{B.18})$$

<sup>3</sup>See "Pattern Recognition" chapter.

equivalent to minimize the negative log-likelihood function  $E = -\ln \mathcal{L}$  which may act as an error function.

$$E = -\ln \mathcal{L} = -\sum_{p=1}^P \ln p(\mathbf{x}_p) = -\sum_{p=1}^P \ln \left[ \sum_{m=1}^M p(\mathbf{x}_p|m)P(m) \right] \quad (\text{B.19})$$

 **Remarks:**

- Considering the Gaussian model then from (B.16) and (B.17):

$$E = -\sum_{p=1}^P \ln \left[ \sum_{m=1}^M \frac{p(\mathbf{x}_p)P(m|\mathbf{x}_p)}{(2\pi\sigma_m^2)^{N/2}} \exp \left( -\frac{\|\mathbf{x}_p - \boldsymbol{\mu}_m\|^2}{2\sigma_m^2} \right) \right]$$

The minimum of  $E$  is found by searching for the roots of its derivative, i.e.  $E$  is minimum for those values of  $\boldsymbol{\mu}_m$  and  $\sigma_m$  for which:

$$\nabla_{\boldsymbol{\mu}_m} E = 0 \Leftrightarrow \sum_{p=1}^P P(m|\mathbf{x}_p) \frac{\mathbf{x}_p - \boldsymbol{\mu}_m}{\sigma_m^2} = 0 \quad \text{and}$$

$$\frac{\partial E}{\partial \sigma_m} = 0 \Leftrightarrow \sum_{p=1}^P P(m|\mathbf{x}_p) \left( N - \frac{\|\mathbf{x}_p - \boldsymbol{\mu}_m\|^2}{\sigma_m^2} \right) = 0$$

(the denominator of derivative shouldn't be 0 anyway, also  $\sigma_m \neq 0$ , see previous remarks).

The above equations gives the following *estimates* for  $\boldsymbol{\mu}_m$  and  $\sigma_m$  parameters:

$$\tilde{\boldsymbol{\mu}}_m = \frac{\sum_{p=1}^P P(m|\mathbf{x}_p) \mathbf{x}_p}{\sum_{p=1}^P P(m|\mathbf{x}_p)} \quad \text{and} \quad \tilde{\sigma}_m = \frac{\sum_{p=1}^P \|\mathbf{x}_p - \boldsymbol{\mu}_m\|^2 P(m|\mathbf{x}_p)}{N \sum_{p=1}^P P(m|\mathbf{x}_p)} \quad (\text{B.20})$$

In order to automatically ensure normalization, the  $P(m)$  parameters may be expressed by the means of  $M$  parameters  $\alpha_m$  as follows:

$$P(m) = \frac{e^{\alpha_m}}{\sum_{q=1}^M e^{\alpha_q}} \quad \alpha_m \in \mathbb{R}, m = \overline{1, M}$$

softmax  
function

These expressions are called *softmax functions*. Then the  $\alpha_m$  are also parameters of the model and  $E$  which depends upon them have to be minimized with respect to them, i.e. its derivative with respect to  $\alpha_m$  should be 0 at minimum.

From the softmax expression:  $\frac{\partial P(q)}{\partial \alpha_m} = \delta_{mq} P(m) - P(m)P(q)$ , also  $\frac{\partial E}{\partial \alpha_m} = \sum_{q=1}^M \frac{\partial E}{\partial P(q)} \frac{\partial P(q)}{\partial \alpha_m}$ ,

( $\delta_{mq}$  being the Kronecker symbol) and then from (B.19):

$$\frac{\partial E}{\partial \alpha_m} = \sum_{q=1}^M \left\{ \sum_{p=1}^P \frac{p(\mathbf{x}_p|q)}{\sum_{\ell=1}^M p(\mathbf{x}_p|\ell)P(\ell)} \right\} [\delta_{mq} P(m) - P(m)P(q)]$$

$$= \sum_{p=1}^P \frac{p(\mathbf{x}_p|m)P(m)}{\sum_{\ell=1}^M p(\mathbf{x}_p|\ell)P(\ell)} - P(m) \sum_{q=1}^M \sum_{p=1}^P \frac{p(\mathbf{x}_p|q)P(q)}{\sum_{\ell=1}^M p(\mathbf{x}_p|\ell)P(\ell)}$$

By applying the Bayes theorem and considering the normalization of  $P(m|\mathbf{x})$ , see (B.16), the first term becomes:

$$\sum_{p=1}^P \frac{p(\mathbf{x}_p|m)P(m)}{\sum_{\ell=1}^M p(\mathbf{x}_p|\ell)P(\ell)} = \sum_{p=1}^P \frac{P(m|\mathbf{x}_p) p(\mathbf{x}_p)}{p(\mathbf{x}_p) \sum_{\ell=1}^M P(\ell|\mathbf{x}_p)} = \sum_{p=1}^P P(m|\mathbf{x}_p)$$

while the second term is:

$$P(m) \sum_{q=1}^M \sum_{p=1}^P \frac{p(\mathbf{x}_p|q) P(q)}{\sum_{\ell=1}^M p(\mathbf{x}_p|\ell)P(\ell)} = P(m) \sum_{p=1}^P \frac{p(\mathbf{x}_p) \sum_{q=1}^M P(q|\mathbf{x}_p)}{p(\mathbf{x}_p) \sum_{\ell=1}^M P(\ell|\mathbf{x}_p)} = PP(m)$$

so finally:

$$\frac{\partial E}{\partial \alpha_m} = 0 \Leftrightarrow P(m) = \frac{1}{P} \sum_{p=1}^P P(m|\mathbf{x}_p) \quad (\text{B.21})$$

### The EM (expectation–maximisation) algorithm

The algorithm works with formulas (B.20) and (B.21) iteratively, in steps, starting with some initial values for the parameters at the first step  $t = 1$  and then recalculating  $\tilde{\mu}_{(t+1)m}$ ,  $\tilde{\sigma}_{(t+1)m}$  and the estimated  $\tilde{P}_{(t+1)}(m)$  by using the old values at the previous step  $\tilde{\mu}_{(t)m}$ ,  $\tilde{\sigma}_{(t)m}$  and  $\tilde{P}_{(t)}(m)$ . It is supposed that  $E$  function gets smaller at each step till it reaches the minimum.

The variation in the error function  $E$  (given by (B.19)), from one step to the next, is:

$$\Delta E = E_{(t+1)} - E_{(t)} = - \sum_{p=1}^P \ln \frac{p_{(t+1)}(\mathbf{x}_p)}{p_{(t)}(\mathbf{x}_p)} \quad (\text{B.22})$$

and, using (B.15) for  $p_{(t+1)}(\mathbf{x}_p)$ :

$$\Delta E = - \sum_{p=1}^P \ln \left[ \frac{\sum_{m=1}^M p_{(t+1)}(\mathbf{x}_p|m) P_{(t+1)}(m)}{p_{(t)}(\mathbf{x}_p)} \frac{P_{(t)}(m|\mathbf{x}_p)}{P_{(t)}(m|\mathbf{x}_p)} \right] \quad (\text{B.23})$$

### Remarks:

- ➔ The *Jensen's inequality* states that given a function  $f$ , convex down on an interval  $[a, b]$ , a set of  $P$  points in that interval  $\{\mathbf{x}_p\}_{p=1}^P \in [a, b]$  and a set of numbers

Jensen's inequality

$\{\alpha_p\}_{p=1}^P \in [0, 1]$  such that  $\sum_{p=1}^P \alpha_p = 1$ , then:

$$f \left( \sum_{p=1}^P \alpha_p x_p \right) \geq \sum_{p=1}^P \alpha_p f(x_p)$$

see the mathematical appendix.

By applying the Jensen's inequality to (B.23):  $f \Leftrightarrow \ln$  and  $\alpha_p \Leftrightarrow P_{(t)}(m|\mathbf{x}_p)$  ( $P_{(t)}(m|\mathbf{x}_p)$  are normated) then:

$$\Delta E \leq \sum_{p=1}^P \sum_{m=1}^M P_{(t)}(m|\mathbf{x}_p) \ln \left[ \frac{P_{(t+1)}(m) p_{(t+1)}(\mathbf{x}_p|m)}{p_{(t)}(\mathbf{x}_p) P_{(t)}(m|\mathbf{x}_p)} \right] \equiv Q \quad (\text{B.24})$$

❖  $Q$  and  $E_{(t+1)} \leq E_{(t)} + Q$ , i.e.  $E_{(t+1)}$  is bounded above and it may be minimized by minimizing  $Q$ . Generally  $Q = Q(W_{(t+1)})$ , the old parameters  $W_{(t)}$  being already established at the previous  $t$  step. Eventually, minimizing  $Q$  is equivalent to minimizing:

$$\tilde{Q} = - \sum_{p=1}^P \sum_{m=1}^M P_{(t)}(m|\mathbf{x}_p) \ln [P_{(t+1)}(m) p_{(t+1)}(\mathbf{x}_p|m)] \quad (\text{B.25})$$

For the Gaussian distribution, see (B.17),  $\tilde{Q}$  becomes

$$\tilde{Q} = - \sum_{p=1}^P \sum_{m=1}^M P_{(t)}(m|\mathbf{x}_p) \left[ \ln P_{(t+1)}(m) - N \ln \sigma_{(t+1)m} - \frac{\|\mathbf{x}_p - \boldsymbol{\mu}_{(t+1)m}\|^2}{2\sigma_{(t+1)m}^2} \right] + \text{const.}$$

The problem is to minimize  $\tilde{Q}$  with respect to  $(t+1)$  parameters, i.e. to find the parameters at step  $t+1$ , such that the condition of normalization for  $P_{(t+1)(m)}$  ( $\sum_{m=1}^M P_{(t+1)}(m) = 1$ ) is met. The *Lagrange multiplier* method<sup>4</sup> is used here. The Lagrange function is:

$$L = \tilde{Q} + \lambda \left[ \sum_{m=1}^M P_{(t+1)}(m) - 1 \right]$$

The value of the parameter  $\lambda$  is found by putting the set of conditions:  $\frac{\partial L}{\partial P_{(t+1)}(m)} = 0$  which gives:

$$-\sum_{p=1}^P \frac{P_{(t)}(m|\mathbf{x}_p)}{P_{(t+1)}(m)} + \lambda = 0 \quad , \quad m = \overline{1, M}$$

and by summation over  $m$ , and because both  $P_{(t)}(m|\mathbf{x}_p)$  and  $P_{(t+1)}(m)$  are normated:

$$\sum_{m=1}^M P_{(t)}(m|\mathbf{x}_p) = 1 \quad \text{and} \quad \sum_{m=1}^M P_{(t+1)}(m) = 1$$

---

<sup>4</sup>See mathematical appendix.

then  $\lambda = M$ .

The required parameters are found by setting the conditions:

$$\nabla_{\mu_{(t+1)m}} L = 0 \quad , \quad \frac{\partial L}{\partial \sigma_m} = 0 \quad \text{and} \quad \frac{\partial L}{\partial P_{(t+1)}(m)} = 0$$

which gives the solutions:

$$\boldsymbol{\mu}_{(t+1)m} = \frac{\sum_{p=1}^P P_{(t)}(m|\mathbf{x}_p) \mathbf{x}_p}{\sum_{p=1}^P P_{(t)}(m|\mathbf{x}_p)} \quad (\text{B.26a})$$

$$\sigma_{(t+1)m} = \sqrt{\frac{\sum_{p=1}^P P_{(t)}(m|\mathbf{x}_p) \|\mathbf{x}_p - \boldsymbol{\mu}_{(t+1)m}\|^2}{N \sum_{p=1}^P P_{(t)}(m|\mathbf{x}_p)}} \quad (\text{B.26b})$$

$$P_{(t+1)}(m) = \frac{1}{N} \sum_{p=1}^P P_{(t)}(m|\mathbf{x}_p) \quad (\text{B.26c})$$

As discussed earlier (see the remarks regarding the incomplete training set) usually the data available is not classified in terms of probability components  $m$ ,  $m = \overline{1, M}$ . A variable  $z_p$ ,  $z_p \in [1, \dots, M]$  may be associated with each training vector  $\mathbf{x}_p$ , to hold the probability component. The error function then becomes:

$$\begin{aligned} E &= -\ln \mathcal{L} = -\sum_{p=1}^P \ln p_{(t+1)}(\mathbf{x}_p, z_p) = -\sum_{p=1}^P \ln[P_{(t+1)}(z_p) p_{(t+1)}(\mathbf{x}_p|z_p)] \\ &= -\sum_{p=1}^P \sum_{m=1}^M \delta_{mz_p} \ln[P_{(t+1)}(z_p) p_{(t+1)}(\mathbf{x}_p|z_p)] \end{aligned}$$

$P_{(t)}(z_p|\mathbf{x}_p)$  represents the probability of  $z_p$  for a given  $\mathbf{x}_p$ , at step  $t$ . The probability of  $E$  for a given set of  $\{z_p\}_{p=\overline{1,P}}$  is the product of all  $P_{(t)}(z_p|\mathbf{x}_p)$ , i.e.  $\prod_{p=1}^P P_{(t)}(z_p|\mathbf{x}_p)$ . The expectation  $\mathcal{E}\{E\}$  is the sum of  $E$  over all values of  $\{z_p\}_{p=\overline{1,P}}$  weighted by the probability of the  $\{z_p\}_{p=\overline{1,P}}$  set:

$$\begin{aligned} \mathcal{E}\{E\} &= -\sum_{z_1=1}^M \cdots \sum_{z_P=1}^M E \prod_{p=1}^P P_{(t)}(z_p|\mathbf{x}_p) \\ &= -\sum_{p=1}^P \sum_{m=1}^M \left[ \sum_{z_1=1}^M \cdots \sum_{z_P=1}^M \delta_{mz_p} \prod_{q=1}^P P_{(t)}(z_q|\mathbf{x}_q) \right] \ln[P_{(t+1)}(z_p) p_{(t+1)}(\mathbf{x}_p|z_p)] \end{aligned}$$

On similar grounds as for  $\mathcal{E}\{E\}$ , the expression in square parenthesis from the above equation represents the expectation  $\mathcal{E}\{\delta_{mz_p}\}$ :

$$\mathcal{E}\{\delta_{mz_p}\} = \sum_{z_1=1}^M \cdots \sum_{z_P=1}^M \delta_{mz_p} \prod_{p=1}^P P_{(t)}(z_p|\mathbf{x}_p) = P_{(t)}(m|\mathbf{x}_p)$$

which represents exactly the probability  $P_{(t)}(m|\mathbf{x}_p)$ . Finally:

$$\mathcal{E}\{E\} = - \sum_{p=1}^P \sum_{m=1}^M P_{(t)}(m|\mathbf{x}_p) \ln[P_{(t+1)}(m) p_{(t+1)}(\mathbf{x}_p|m)]$$

which is identical with the expression of  $\tilde{Q}$ , see (B.25), and thus minimizing  $\tilde{Q}$  is equivalent to minimizing  $\mathcal{E}\{E\}$  at the same time.

### **Stochastic estimation**

Let consider that the training vectors came one at a time. For a set of  $P$  training vectors the  $\mu_{(P)m}$  parameter from the Gaussian distribution is (see (B.26a)):

$$\mu_{(P)m} = \frac{\sum_{p=1}^P P(m|\mathbf{x}_p) \mathbf{x}_p}{\sum_{p=1}^P P(m|\mathbf{x}_p)}$$

and, after the  $P + 1$  training vector have arrived:

$$\mu_{(P+1)m} = \frac{\sum_{p=1}^{P+1} P(m|\mathbf{x}_p) \mathbf{x}_p}{\sum_{p=1}^{P+1} P(m|\mathbf{x}_p)} = \mu_{(P)m} + \alpha_{(P+1)m} (\mathbf{x}_{P+1} - \mu_{(P)m})$$

where

$$\alpha_{(P+1)m} = \frac{P(m|\mathbf{x}_{P+1})}{\sum_{p=1}^{P+1} P(m|\mathbf{x}_p)}$$

To avoid keeping all old  $\{\mathbf{x}_p\}_{p=1}^{P+1}$  (to calculate  $\sum_{p=1}^{P+1} P(m|\mathbf{x}_p)$ ) use either (B.21) such that:

$$\alpha_{(P+1)m} = \frac{P(m|\mathbf{x}_{P+1})}{(P+1)P(m)}$$

or, directly from the expression of  $\alpha_{(P+1)m}$ :

$$\alpha_{(P+1)m} = \frac{1}{1 + \frac{P(m|\mathbf{x}_P)}{\alpha_{(P)m} P(m|\mathbf{x}_{P+1})}}$$

## ► B.3 The Bayesian Inference

Unlike other techniques where the probabilities are build by finding the best set of  $W$  parameters, the Bayesian inference assumes a probability density for  $W$  itself.

The following procedure is repeated for each class  $C_k$  in turn. First a *prior* probability is chosen  $p(W)$ , with a large coverage for the *unknown* parameters, then using the training set  $\{\mathbf{x}_p\}_P$  the posterior probability density  $p(W|\{\mathbf{x}_p\}_P)$  is found trough the Bayes theorem.

The process of finding  $p(W|\{\mathbf{x}_p\}_P)$ , from  $p(W)$  and  $\{\mathbf{x}_p\}_P$ , is named *Bayesian learning*.

Let  $p(\mathbf{x}|\{\mathbf{x}_p\}_P)$  be the probability density for a pattern from  $\{\mathbf{x}_p\}_P$  to have its pattern vector  $\mathbf{x}$  and let  $p(\mathbf{x}, W|\{\mathbf{x}_p\}_P)$  be the join probability density that a pattern from  $\{\mathbf{x}_p\}_P$  have its pattern vector  $\mathbf{x}$  and the parameters of the probability density are defined trough  $W$ . Then:

$$p(\mathbf{x}|\{\mathbf{x}_p\}_P) = \int_W p(\mathbf{x}, W|\{\mathbf{x}_p\}_P) dW \quad (\text{B.27})$$

the integral being done over all possible values of  $W$ , i.e. in the  $W$  space.

$p(\mathbf{x}, W|\{\mathbf{x}_p\}_P)$  represents the ratio of pattern vectors being  $\mathbf{x}$  with their probability density characterized by  $W$  and being into the training set  $\{\mathbf{x}_p\}_P$  relative to the total number of training vectors:

$$p(\mathbf{x}, W|\{\mathbf{x}_p\}_P) = \frac{\text{no. patterns being } \mathbf{x}, \text{ with } W, \text{ in } \{\mathbf{x}_p\}_P}{\text{no. patterns in } \{\mathbf{x}_p\}_P}$$

$p(\mathbf{x}|W, \{\mathbf{x}_p\}_P)$  represents the ratio of pattern vectors being  $\mathbf{x}$  with their probability density characterized by  $W$  and being into the training set  $\{\mathbf{x}_p\}_P$  relative to the number of training vectors with their probability density characterized by  $W$  from the training set  $\{\mathbf{x}_p\}_P$ :

$$p(\mathbf{x}|W, \{\mathbf{x}_p\}_P) = \frac{\text{no. patterns being } \mathbf{x}, \text{ with } W, \text{ in } \{\mathbf{x}_p\}_P}{\text{no. patterns with } W, \text{ in } \{\mathbf{x}_p\}_P}$$

$p(W|\{\mathbf{x}_p\}_P)$  represents the ratio of pattern vectors with their probability density characterized by  $W$  and being into the training set  $\{\mathbf{x}_p\}_P$  relative to the number of training vectors:

$$p(W|\{\mathbf{x}_p\}_P) = \frac{\text{no. patterns with } W, \text{ in } \{\mathbf{x}_p\}_P}{\text{no. patterns in } \{\mathbf{x}_p\}_P}$$

Then, from the above equations, it gets that:

$$p(\mathbf{x}, W|\{\mathbf{x}_p\}_P) = p(\mathbf{x}|W, \{\mathbf{x}_p\}_P) p(W|\{\mathbf{x}_p\}_P)$$

The probability density  $p(\mathbf{x}|W, \{\mathbf{x}_p\}_P)$  have to be independent of the choice of the statistically valid training set (same ratio should be in any training set) and consequently it reduces to  $p(\mathbf{x}|W)$ . Finally:

$$p(\mathbf{x}|\{\mathbf{x}_p\}_P) = \int_W p(\mathbf{x}|W) p(W|\{\mathbf{x}_p\}_P) dW \quad (\text{B.28})$$

❖  $\{\mathbf{x}_p\}_P$

Bayesian learning

❖  $p(\mathbf{x}|\{\mathbf{x}_p\}_P)$ ,  
 $p(\mathbf{x}, W|\{\mathbf{x}_p\}_P)$

❖  $p(\mathbf{x}|W, \{\mathbf{x}_p\}_P)$

❖  $p(W|\{\mathbf{x}_p\}_P)$

**Remarks:**

- By contrast with other statistical methods who tries to find the best set of parameters  $W$ , the Bayesian inference method performs an weighted average over the  $W$  space, using all possible sets of  $W$  parameters according to their probability to be the right choice.

The probability density of the whole set  $\{\mathbf{x}_p\}_P$  is the product of densities for each  $\mathbf{x}_p$  (assuming that the set is statistically significant):

$$p(\{\mathbf{x}_p\}_P|W) = \prod_{p=1}^P p(\mathbf{x}_p|W)$$

From the Bayes theorem and using the above equation:

$$p(W|\{\mathbf{x}_p\}_P) = \frac{p(\{\mathbf{x}_p\}_P|W)p(W)}{p(\{\mathbf{x}_p\}_P)} = \frac{p(W)}{p(\{\mathbf{x}_p\}_P)} \prod_{p=1}^P p(\mathbf{x}_p|W) \quad (\text{B.29})$$

$p(\{\mathbf{x}_p\}_P)$  plays the role of a normalization factor, and from the condition of normalization for  $p(W|\{\mathbf{x}_p\}_P)$ :  $\int_W p(W|\{\mathbf{x}_p\}_P) dW = 1$ , it follows that:

$$p(\{\mathbf{x}_p\}_P) = \int_W p(W) \prod_{p=1}^P p(\mathbf{x}_p|W) dW \quad (\text{B.30})$$

**Remarks:**

- Let consider a unidimensional Gaussian distribution with the standard deviation  $\sigma$  known and try to find the parameter  $\mu$  from a training set  $\{\mathbf{x}_p\}_P$ .

The probability density of  $\mu$  will be modeled also as a Gaussian characterized by parameters  $\mu_\mu$  and  $\sigma_\mu$ .

❖  $\mu_\mu, \sigma_\mu$

$$p(\mu) = \frac{1}{\sqrt{2\pi}\sigma_\mu} \exp\left[-\frac{(\mu - \mu_\mu)^2}{2\sigma_\mu^2}\right] \quad (\text{B.31})$$

where this form of  $p(\mu)$  expresses the prior knowledge of the probability density for  $\mu$  and then a large value for  $\sigma_\mu$  should be chosen (large variance).

Using the training set, the posterior probability  $p(\mu|\{\mathbf{x}_p\}_P)$  is calculated for  $\mu$ :

$$p(\mu|\{\mathbf{x}_p\}_P) = \frac{p(\mu)}{p(\{\mathbf{x}_p\}_P)} \prod_{p=1}^P p(x_p|\mu) \quad (\text{B.32})$$

Assuming a Gaussian distribution for  $p(x_p|\mu)$  then:

$$p(x_p|\mu) = \frac{1}{\sqrt{2\pi}\sigma} \exp\left[-\frac{(x_p - \mu)^2}{2\sigma^2}\right] \quad (\text{B.33})$$

From (B.30) and (B.33)

$$p(\{\mathbf{x}_p\}_P) = \int_{-\infty}^{\infty} p(\mu) \prod_{p=1}^P p(x_p|\mu) d\mu \quad (\text{B.34})$$

$$= \frac{1}{(2\pi)^{\frac{P+1}{2}} \sigma_\mu \sigma^P} \int_{-\infty}^{\infty} \exp \left[ -\frac{(\mu - \mu_\mu)^2}{2\sigma_\mu^2} - \frac{1}{2\sigma^2} \sum_{p=1}^P (\mu - x_p)^2 \right] d\mu$$

Let  $\langle x \rangle = \frac{1}{P} \sum_{p=1}^P x_p$  be the mean of the training set. ❖  $\langle x \rangle$

Replacing (B.31), (B.33) and (B.34) back into (B.32) gives:

$$p(\mu | \{\mathbf{x}_p\}_P) \propto \exp \left( -\frac{\mu^2}{2\sigma_\mu^2} - \frac{\mu\mu_\mu}{\sigma_\mu^2} - \frac{P\mu^2}{2\sigma^2} - \frac{\mu P\langle x \rangle}{\sigma^2} \right) \Rightarrow$$

$$p(\mu | \{\mathbf{x}_p\}_P) = \text{const.} \exp \left[ -\frac{\frac{1}{\sigma_\mu^2} + \frac{P}{\sigma^2}}{2} \left( \mu - \frac{\frac{\mu_\mu}{\sigma_\mu^2} + \frac{P\langle x \rangle}{\sigma^2}}{\frac{1}{\sigma_\mu^2} + \frac{P}{\sigma^2}} \right)^2 \right]$$

(*const.* being the normalization factor). This expression shows that  $p(\mu | \{\mathbf{x}_p\}_P)$  is also a Gaussian distribution characterized by the parameters:

$$\tilde{\mu} = \frac{P\sigma_\mu^2 \langle x \rangle + \sigma^2 \mu_\mu^2}{P\sigma_\mu^2 + \sigma^2} \quad \text{and} \quad \tilde{\sigma} = \sqrt{\frac{\sigma^2 \sigma_\mu^2}{P\sigma_\mu^2 + \sigma^2}}$$

For  $P \rightarrow \infty$ :  $\tilde{\mu} \rightarrow \langle x \rangle$  and  $\tilde{\sigma} \rightarrow 0$ .  $\lim_{P \rightarrow \infty} \tilde{\sigma} = 0$  shows that, for  $P \rightarrow \infty$ ,  $\mu$  itself will assume the limit value  $\langle x \rangle$ .

- There is a relationship between the Bayesian inference and maximum likelihood methods. The likelihood function is defined as:

$$p(\{\mathbf{x}_p\}_P | W) = \prod_{p=1}^P p(\mathbf{x}_p | W) \equiv \mathcal{L}(W)$$

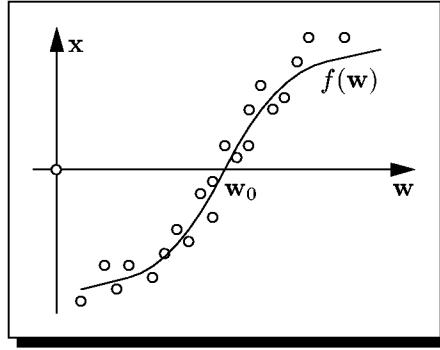
then from (B.29):

$$p(W | \{\mathbf{x}_p\}_P) = \frac{\mathcal{L}(W)p(W)}{p(\{\mathbf{x}_p\}_P)}$$

$p(W)$  represents the prior knowledge about  $W$  which is low and so  $p(W)$  should be relatively flat, i.e. all  $W$  have (relatively) the same probability (chance). Also, by construction,  $\mathcal{L}(W)$  is maximum for the most probable value for  $W$ , let  $\tilde{W}$  be that one. Then  $p(W | \{\mathbf{x}_p\}_P)$  have a maximum around  $\tilde{W}$ .

If the  $p(W | \{\mathbf{x}_p\}_P)$ , maximum in  $\tilde{W}$ , is relatively sharp then for  $P \rightarrow \infty$  the integral (B.28) is dominated by the area around  $\tilde{W}$  and:

$$\begin{aligned} p(\mathbf{x} | \{\mathbf{x}_p\}_P) &= \int_W p(\mathbf{x} | W) p(W | \{\mathbf{x}_p\}_P) dW \\ &\simeq p(\mathbf{x} | \tilde{W}) \int_W p(W | \{\mathbf{x}_p\}_P) dW = p(\mathbf{x} | \tilde{W}) \end{aligned}$$



**Figure B.6:** The regression function  $f(\mathbf{w})$  approximate the  $\mathbf{x}(\mathbf{w})$  dependency. The roots  $\mathbf{w}_0$  of  $f$  are found by the Robbins-Monro algorithm.

(because  $\int_W p(W|\{\mathbf{x}_p\}_P) dW = 1$ , i.e.  $p(W|\{\mathbf{x}_p\}_P)$  is normalized).

The conclusion is that for a large number of training patterns  $P \rightarrow \infty$  the Bayesian inference solution (B.27) approaches the maximum likelihood solution  $p(\mathbf{x}|\widetilde{W})$ .

## ► B.4 The Robbins–Monro algorithm

This algorithm shows a way of finding the roots of a function stochastically defined.

Let consider 2 variables  $\mathbf{x}$  and  $\mathbf{w}$  which are correlated  $\mathbf{x} = \mathbf{x}(\mathbf{w})$ . Let  $\mathcal{E}\{\mathbf{x}|\mathbf{w}\}$  be the expectation of  $\mathbf{x}$  for a given  $\mathbf{w}$  — this expression defines a function of  $\mathbf{w}$

$$f(\mathbf{w}) = \mathcal{E}\{\mathbf{x}|\mathbf{w}\}$$

regression  
function

this types of functions being named *regression functions*.

The regression function  $f$  expresses the dependency between the *mean* of  $\mathbf{x}$  and  $\mathbf{w}$ . See figure B.6.

Let  $\mathbf{w}_0$  be the wanted root. It is assumed that  $\mathbf{x}$  have a finite variance:

$$\mathcal{E}\{(\mathbf{x} - f)^2 | \mathbf{w}\} = \text{finite} \quad (\text{B.35})$$

and, without any loss of generality, that  $f(\mathbf{w}) < 0$  for  $\mathbf{w} < \mathbf{w}_0$  and  $f(\mathbf{w}) > 0$  for  $\mathbf{w} > \mathbf{w}_0$ , as in figure B.6.

**Theorem B.4.1.** *The root  $\mathbf{w}_0$ , of the regression function  $f$  is found by successive iteration, starting from a value  $\mathbf{w}_1$  in the vicinity of  $\mathbf{w}_0$ , as follows:*

$$\mathbf{w}_{i+1} = \mathbf{w}_i - \alpha_i \mathbf{x}(\mathbf{w}_i)$$

---

<sup>B.4</sup>See [Fuk90] pp. 378–380.

where  $\alpha_i$  have to satisfy the following conditions:

$$\lim_{i \rightarrow \infty} \alpha_i = 0 \quad (\text{B.36a})$$

$$\sum_{i=1}^{\infty} \alpha_i = \infty \quad (\text{B.36b})$$

$$\sum_{i=1}^{\infty} \alpha_i^2 = \text{finite} \quad (\text{B.36c})$$

*Proof.* The  $x(w)$  may be expressed as a sum between the regression function  $f(w)$  and some noise  $\xi$ :  $\diamond \xi$

$$x(w) = f(w) + \xi \quad (\text{B.37})$$

then, from the definition of the regression function  $f$

$$\mathcal{E}\{\xi|w\} = \mathcal{E}\{x|w\} - f(w) = 0$$

( $f$  is well defined, so its expectation is  $f$  itself).

The difference between  $w_{i+1}$  and  $w_0$  is

$$w_{i+1} - w_0 = w_i - w_0 - \alpha_i f(w_i) - \alpha_i \xi_i$$

where  $\xi_i$  is the noise from  $x(w_i)$ . Taking the expectation of the square of the above expression:  $\diamond \xi_i$

$$\mathcal{E}\{(w_{i+1} - w_0)^2\} - \mathcal{E}\{(w_i - w_0)^2\} = \alpha_i^2 f^2(w_i) + \alpha_i^2 \mathcal{E}\{\xi_i^2\} - 2\alpha_i \mathcal{E}\{(w_i - w_0)f(w_i)\}$$

$(\mathcal{E}\{f(w_i)\}\xi_i) = \mathcal{E}\{f(w_i)\}\mathcal{E}\{\xi_i\}$  because  $f$  and  $\xi$  are statistically independent, so  $2\alpha_i \mathcal{E}\{f(w_i)\}\xi_i = 2\alpha_i f(w_i)\mathcal{E}\{\xi_i\} = 0$  because of the expectation of  $\xi$ .

By repeating the above procedure over  $N$  steps and doing the sum gives:

$$\mathcal{E}\{(w_{N+1} - w_0)^2\} - \mathcal{E}\{(w_1 - w_0)^2\} = \sum_{i=1}^N \alpha_i^2 [f^2(w_i) + \mathcal{E}\{\xi_i^2\}] - 2 \sum_{i=1}^N \alpha_i \mathcal{E}\{(w_i - w_0)f(w_i)\}$$

It is reasonable to assume that  $w_1$  is chosen such that  $f^2$  is bounded in the chosen vicinity of the searched root, then let  $f^2(w_i) \leq F$  for all  $i \in \{1, \dots, N+1\}$  and let  $\mathcal{E}\{\xi_i^2\} \leq \Xi^2$ . (it is assumed that  $\xi^2$  is bounded, see (B.35) and (B.37)). Then:

$$\mathcal{E}\{(w_{N+1} - w_0)^2\} - \mathcal{E}\{(w_1 - w_0)^2\} \leq (F - \Xi^2) \sum_{i=1}^N \alpha_i^2 - 2 \sum_{i=1}^N \alpha_i \mathcal{E}\{(w_i - w_0)f(w_i)\} \quad (\text{B.38})$$

$\mathcal{E}\{(w_{N+1} - w_0)^2\} > 0$  as being the expectation of a positive quantity. As it was already discussed,  $w_1$  is chosen such that  $\mathcal{E}\{(w_1 - w_0)^2\}$  is limited. Then the left part of the equation (B.38) is bounded below by 0.

The  $(F - \Xi^2) \sum_{i=1}^{N-1} \alpha_i^2$  term is also finite because of the condition (B.36c) then, from (B.38):

$$0 \leq 2 \sum_{i=1}^N \alpha_i \mathcal{E}\{(w_i - w_0)f(w_i)\} \leq (F - \Xi^2) \sum_{i=1}^N \alpha_i^2 = \text{finite} \Rightarrow$$

$$\sum_{i=1}^N \alpha_i \mathcal{E}\{(w_i - w_0)f(w_i)\} = \text{finite}$$

Because of the conditions put on the signs of  $f$  and  $w_i$  then  $(w_i - w_0)f(w_i) > 0, \forall w_i$ , and its expectation is also positive. Eventually, because of the condition (B.36b) and the above equation then:

$$\lim_{i \rightarrow \infty} \mathcal{E}\{(w_i - w_0)f(w_i)\} = 0$$

i.e.  $\lim_{i \rightarrow \infty} w_i = w_0$ . and because  $f$  changes sign around  $w_0$  then it's a root for the regression function.  $\square$

**Remarks:**

- ➔ Condition (B.36a) ensure that the process of finding the root is convergent. Condition (B.36b) ensure that each iterative correction to the solution  $\mathbf{w}_i$  is large enough. Condition (B.36c) ensure that the accumulated noise (the difference between  $\mathbf{x}(\mathbf{w})$  and  $f(\mathbf{w})$ ) does not break the convergence process.
- ➔ The Robbins–Monro algorithm allows for finding the roots of  $f(\mathbf{w})$  without knowing the exact form of the regression function.
- ➔ A possible candidate for  $\alpha_i$  coefficients is

$$\alpha_i = \frac{1}{i^n} \quad , \quad n \in (1/2, 1]$$

## ► B.5 Learning vector quantization

**Remarks:**

❖  $\mathbf{m}_k$   
codebook

- ➔ Vector quantization is used in signal processing to replace a vector  $\mathbf{x} \in \mathcal{C}_k$  by a representative  $\mathbf{m}_k$  for its class. The collection of all representatives  $\{\mathbf{m}_k\}$  is named *codebook*. Then, instead of sending each  $\mathbf{x}$ , the codebook is send first and then just the index of  $\mathbf{m}_k$  (closest to  $\mathbf{x}$ ) instead of  $\mathbf{x}$ .

The representatives are chosen by starting with some set and then updating it using a rule similar to the following: at step  $t$  a vector  $\mathbf{x}$  from the “training set” is taken in turn and the representatives are updated/changed in discrete steps:

$$\mathbf{m}_{k(t+1)} = \mathbf{m}_{k(t)} + \alpha(t) (\mathbf{x} - \mathbf{m}_{k(t)}) \quad \text{for } \mathbf{m}_k \text{ closest to } \mathbf{x}$$

$$\mathbf{m}_{\ell(t+1)} = \mathbf{m}_{j(s)} \quad \text{no change, for all other } \mathbf{m}_\ell$$

❖  $\alpha(t)$

where  $\alpha(t)$  is a function of the “discrete time”  $t$ , usually decreasing in time.

The  $\mathbf{m}_k$  vectors should be chosen as representative as possible. It is also possible to choose several  $\mathbf{m}_k$  vectors per each class (this is particularly necessary if a class is represented by some disconnected subspaces  $X_k \in X$  in the pattern space).

LVQ1

There are several methods of updating the code vectors. The simplest method is named LVQ1 and the updating rule is:

$$\mathbf{m}_{k(t+1)} = \mathbf{m}_{k(t)} + \alpha(t) (\mathbf{x} - \mathbf{m}_{k(t)}) \quad \text{for } \mathbf{x} \in \mathcal{C}_k \tag{B.39}$$

$$\mathbf{m}_{\ell(t+1)} = \mathbf{m}_{k(t)} - \alpha(t) (\mathbf{x} - \mathbf{m}_{k(t)}) \quad \text{for all other } \mathbf{m}_\ell$$

i.e. not only the  $\mathbf{m}_k$  is pushed *towards* an “average” of class  $\mathcal{C}_k$  but all others are spread in the *opposite* direction. The  $\alpha(t)$  is chosen to start with some small value,  $\alpha(0) < 0.1$  and decreasing linearly towards zero.

OLVQ1  
 $\alpha_k$

Another variant, named OLVQ1, uses the same rule (B.39) but different  $\alpha_k(t)$  function for

each  $\mathbf{m}_k$

$$\alpha_k(t+1) = \begin{cases} \frac{\alpha_k(t)}{1 + \alpha_k(t)} & \text{for } \mathbf{x} \in \mathcal{C}_k \\ \frac{\alpha_k(t)}{1 - \alpha_k(t)} & \text{otherwise} \end{cases} \quad (\text{B.40})$$

i.e.  $\alpha_k$  is decreased for the correct class and increased otherwise. The above formula may be justified as follows. Let  $\delta(t)$  be defined as:

$$\delta(t) = \begin{cases} +1 & \text{for correct classification} \\ -1 & \text{otherwise} \end{cases}$$

then, from (B.39), it follows that:

$$\begin{aligned} \mathbf{m}_{k(t+1)} &= \mathbf{m}_{k(t)} + \delta(t) \alpha(t) [\mathbf{x}_{(t)} - \mathbf{m}_{k(t)}] = [1 - \delta(t) \alpha(t)] \mathbf{m}_{k(t)} + \delta(t) \alpha(t) \mathbf{x}_{(t)} \\ \mathbf{m}_{k(t)} &= [1 - \delta(t-1) \alpha(t-1)] \mathbf{m}_{k(t-1)} + \delta(t-1) \alpha(t-1) \mathbf{x}_{(t-1)} \\ \mathbf{m}_{k(t+1)} &= [1 - \delta(t) \alpha(t)] [1 - \delta(t-1) \alpha(t-1)] \mathbf{m}_{k(t-1)} \\ &\quad + [1 - \delta(t) \alpha(t)] \delta(t-1) \alpha(t-1) \mathbf{x}_{(t-1)} + \delta(t) \alpha(t) \mathbf{x}_{(t)} \end{aligned} \quad (\text{B.41})$$

i.e. the code vectors  $\mathbf{m}_{k(t+1)}$  are a linear combination of the training set and the initial code vectors  $\mathbf{m}_{k(0)}$ .

Now let consider that  $\mathbf{x}_{(t-1)} = \mathbf{x}_{(t)}$ , then consistency requires that  $\mathbf{m}_{k(t)} = \mathbf{m}_{k(t+1)}$  and thus  $\delta(t-1) = \delta(t)$ .  $\mathbf{x}_{(t-1)}$  and  $\mathbf{x}_{(t)}$  being identical should have the same contribution to  $\mathbf{m}_{k(t+1)}$ , i.e. their coefficients in (B.41) should be equal:

$$[1 - \delta(t) \alpha(t)] \alpha(t-1) = \alpha(t)$$

which leads directly to (B.40).

Another procedure to build the code vectors is LVQ2.1. For each  $\mathbf{x}$  the two closest code vectors are found. Then these two code vectors are updated if:

- one is of the same class as  $\mathbf{x}$ , let  $\mathbf{m}_\equiv$  be this one,
- the other is of different class, let  $\mathbf{m}_\neq$  be that one and
- $\mathbf{x}$  is near the middle between the two code vectors:

$$\min \left( \frac{\|\mathbf{x} - \mathbf{m}_\equiv\|}{\|\mathbf{x} - \mathbf{m}_\neq\|}, \frac{\|\mathbf{x} - \mathbf{m}_\neq\|}{\|\mathbf{x} - \mathbf{m}_\equiv\|} \right) > \frac{1-f}{1+f}$$

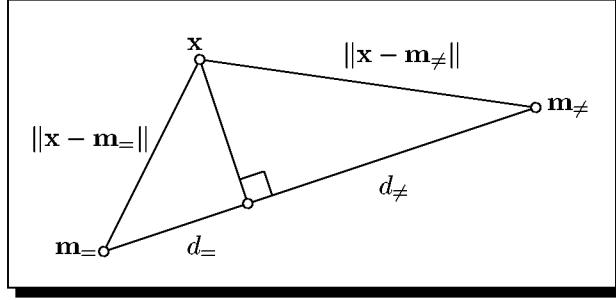
where  $f \simeq 0.25$ .

### Remarks:

- ➔ The last rule from above may be geometrically interpreted as follows: let consider that the  $\mathbf{x}$  is sufficiently close to the line connecting  $\mathbf{m}_\equiv$  and  $\mathbf{m}_\neq$ . See figure B.7 on the next page.

In this case it is possible to make the approximations:

$$\|\mathbf{x} - \mathbf{m}_\equiv\| \simeq d_\equiv \quad \text{and} \quad \|\mathbf{x} - \mathbf{m}_\neq\| \simeq d_\neq$$



**Figure B.7:** The geometrical interpretation of the LVQ2.1 rule. The  $x$  point is projected onto the line connecting  $\mathbf{m}_\neq$  and  $\mathbf{m}_\neq$ .

and there are 2 cases, one of them being

$$\begin{aligned} \min \left( \frac{d_\neq}{d_\neq}, \frac{d_\neq}{d_\neq} \right) &= \frac{d_\neq}{d_\neq} > \frac{1-f}{1+f} \quad \Rightarrow \quad d_\neq < d_\neq \frac{1+f}{1-f} \\ \Rightarrow \quad d &= d_\neq + d_\neq < d_\neq \frac{2}{1-f} \quad \Rightarrow \quad \frac{d_\neq}{d} > \frac{1-f}{2} \end{aligned}$$

the other case giving a similar result:  $\frac{d_\neq}{d} > \frac{1-f}{2}$ , i.e. in either case the projection of  $x$  is at least at a fraction  $(1-f)/2$  of  $d$  from  $\mathbf{m}_\neq$  and  $\mathbf{m}_\neq$ .

The updating formula for LVQ2.1 is:

$$\begin{aligned} \mathbf{m}_{=(t+1)} &= \mathbf{m}_{=(t)} + \alpha(t)[\mathbf{x} - \mathbf{m}_{=(t)}] \\ \mathbf{m}_{\neq(t+1)} &= \mathbf{m}_{\neq(t)} - \alpha(t)[\mathbf{x} - \mathbf{m}_{\neq(t)}] \end{aligned}$$

While the LVQ2.1 updates the codebook less frequently than the previous procedures it tends to over adapt the code vectors.

The LVQ3 was developed to prevent the over adaptation of LVQ2.1 method. This algorithm is similar to LVQ2.1 but if the two closest code vectors, let  $\mathbf{m}'_\neq$  and  $\mathbf{m}''_\neq$  be the ones, are of the same class then they are also updated according to the formula:

$$\begin{aligned} \mathbf{m}'_{=(t+1)} &= \mathbf{m}'_{=(t)} + \varepsilon \alpha(t)[\mathbf{x} - \mathbf{m}'_{=(t)}] \quad \text{and} \\ \mathbf{m}''_{=(t+1)} &= \mathbf{m}''_{=(t)} + \varepsilon \alpha(t)[\mathbf{x} - \mathbf{m}''_{=(t)}] \end{aligned}$$

where  $\varepsilon$  is a tunable parameter, usually chosen in the interval  $[0.1, 0.5]$ .

### Remarks:

- In practice usually OLVQ1 is run first (on the rapid changing code vectors part) and then LVQ1 and/or LVQ3 is used to make the fine adjustments.

# 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.
- [CG87] *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.
- [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.
- [Fuk90] *Keinosuke Fukunaga.* **Introduction to Statistical Pattern Recognition.** Academic Press, San Diego, 2nd edition, 1990. ISBN 0-12-269851-7.
- [Koh88] *Tuovo Kohonen.* **Self-Organization and Associative Memory.** Springer-Verlag, New York, 2nd edition, 1988. ISBN 0-201-51376-5.
- [McC97] *Martin McCarthy.* **What is multi-threading?** *Linux Journal*, -(34):31–40, February 1997.
- [Mos97] *David Mosberger.* **Linux and the alpha: How to make your application fly, part 2.** *Linux Journal*, -(43):68–75, November 1997.
- [Rip96] *Brian D. Ripley.* **Pattern Recognition and Neural Networks.** Cambridge University Press, New York, 1996. ISBN 0-521-46086-7.
- [Str81] *Karl R. Stromberg.* **An Introduction to Classical Real Analysis.** Wadsworth International Group, Belmont, California, 1981. ISBN 0-534-98012-0.



# Index

- 
- 2/3 rule, 85, 86, 94, 98, 101
- activation function, *see* function, activation  
adeline, *see* perceptron  
adaptive backpropagation, *see* algorithm, backpropagation, adaptive  
Adaptive Resonance Theory, *see* ART  
algorithm  
    ART1, 99  
    ART2, 107  
    backpropagation, 16, 162, 320  
        adaptive, 21, 23, 321  
        batch, 163  
        momentum, 20, 23, 24, 321  
        quick, 224  
        standard, 13, 19  
    SuperSAB, 23, 321  
    vanilla, *see* standard  
BAM, 49, 53, 322  
branch and bound, 242  
CPN, 81  
EM, 181, 240, 357  
expectation–maximization, *see* EM  
gradient descent, 218  
Levenberg–Marquardt, 234  
line search, 225  
Metropolis, 296  
model trust region, 235  
optimal brain damage, 266  
optimal brain surgeon, 267  
Robbins–Monro, 124, 181, 219  
sequential search, 243  
simulated annealing, 296  
SOM, 40, 321
- ANN, iii, 3  
    output, 112, 190  
asymmetric divergence, 125, 211
- BAM, 48
- Bayes  
    rule, 116, 119, 354  
    theorem, 288, 342
- Bayesian learning, 361
- bias, 18, 123, 129, 160, 188, 254  
    average, 254
- bias-variance tradeoff, 255
- Bidirectional Associative Memory, *see* BAM  
bit, 211
- Boltzmann constant, 208
- certain event, 210
- city-block metric, *see* error, city-block metric
- class, 111
- classification, 116, 197
- clique, 307
- codebook, 366
- complexity criteria, 273
- conditional independence, 308
- confusion matrix, 120
- conjugate direction, 226, 228
- conjugate directions, 226
- contrast enhancement, 86, 108  
    function, *see* function, contrast enhancement
- counterpropagation network, *see* network, CPN
- course of dimensionality, 113, 237, 255
- cross-validation, 273
- curse of dimensionality, 241
- cycle, 307
- DAG, 307, 313
- decision  
    boundary, 116, 117, 119  
    region, 116  
    tree, 301  
        splitting, 301
- delta rule, 12, 142, 143, 145, 146, 162, 163, 219
- deviance, 127
- dimensionality reduction, 237, 245
- discrete time approximation, 219
- distance  
    Euclidean, 325, 350  
    Hamming, 325  
    Kullback–Leiber, *see* asymmetric divergence  
    Mahalanobis, 241, 348
- distribution, 191  
    Bernoulli, 136  
    conditional, 194  
    Gaussian, 191, 209, 261, 274, 345  
        multidimensional, 346  
        unidimensional, 345  
    Laplacian, 192
- edge, 307
- eigenvalue, 36, 327, 328  
    spectrum, 327
- eigenvector, 36, 327, 347
- encoding  
    one-of- $k$ , 68, 73, 81, 108, 197, 203, 205, 212, 239

- entropy, 208, 210, 302  
 cross-entropy, 202, 203  
 differential, 209
- equation  
 Riccati, 33  
 simple, 32  
 trivial, 32
- error, 163, 176, 186, 212, 215, 218  
 absolute, 203  
 bar, 191  
 city-block metric, 192  
 cross-entropy, 213  
 function, 114, 185  
 gradient, 12–15, 17  
 Minkowski, 192, 213  
 quadratic, 232  
 relative, 203  
 RMS, 115, 187  
 root-mean-square, *see* error, RMS  
 sum-of-squares, 11, 12, 15, 17, 19, 114,  
 137, 143, 149, 163, 167, 171, 176,  
 177, 182, 190, 203, 213, 253, 268,  
 274, 279–281  
 surface, 12, 20, 24, 26, 215, 217, 256
- Euler-Lagrange equation, 176
- evidence, 288  
 approximation, 287
- exception, 115
- exemplars, 47
- expectation, 87, 90, 343
- feature extraction, 114, 237
- feedback, 31  
 auto, 54  
 indirect, 30, 91  
 lateral, 41, 91
- feedforward network, *see* network, feedforward
- Fisher criterion, 148, 152, 199
- Fisher discriminant, 149, 265
- flat spot elimination, 20
- Fletcher-Reeves formula, 230
- function  
 $\delta$ -Dirac, 125, 176, 197, 204, 353  
 activation, 3–6, 40, 57, 90, 92, 134, 198,  
 217  
 exponential-distribution, 7  
 hyperbolic-tangent, 6  
 identity, 48, 175, 199  
 logistic, 5, 6, 10, 15, 19, 135, 163, 202,  
 206, 259  
 pulse-coded, 7  
 ratio-polynomial, 7  
 sigmoidal, 157, 285  
 threshold, 5, 6, 7, 135, 154
- BAM  
 energy, 51
- contrast enhancement, 102
- discriminant, 117
- energy  
 BAM, 52, 56
- error, *see* error
- Euler, 192, 292, 331, 353
- even, 330
- feedback, 30, 74, 75, 91
- Gaussian, 174, 175
- Green, 176
- Heaviside, 160
- Hopfield  
 energy, 55, 56, 57, 59
- kernel, 178, 195, 248, 352
- Lagrange, 334
- Liapunov, 51
- likelihood, 121, 123, 185, 192, 202, 205,  
 261, 315
- multi-quadratic, 174
- radial basis, 173, 174
- regression, 364
- softmax, 195, 206, 263, 272, 356
- stop, 43
- generalization, 15, 112, 115
- Gini index, 302
- graph, 307  
 ancestral, 308  
 boundary, 308  
 chordal, *see* graph, triangulated  
 complete, 307  
 connected, 307  
 directed, 307  
 directed acyclic, *see* DAG  
 moral, 314  
 polytree, 308  
 tree, 307  
 chain, 310  
 join, 312  
 triangulated, 312  
 undirected, 307
- Hadamard product, iii
- Hamming  
 distance, *see* distance, Hamming  
 space, 50, 325  
 vector, *see* vector, bipolar
- Hessian, 127, 166, 172, 256, 266, 281, 288  
 inverse, 166, 168
- Hestenes-Stiefel formula, 230
- histogram, 350  
 K-nearest-neighbors, 351, 353  
 kernel method, 351
- hyperprior, 288
- importance sampling, 296
- information, 210  
 criterion, 128
- input normalization, 238
- invariance, 247  
 translation, 250
- Jacobian, 164, 247
- Jensen's inequality, 132, 335, 357
- Karhunen-Loéve transformation, 245
- kernel function, *see* function, kernel
- Kohonen, *see* SOM
- Kronecker symbol, 118, 165, 196, 197, 347, 356
- Kronecker symbol, 38, 48, 61

- L'Hospital rule, 302
- Lagrange multiplier, 334
- layer
  - competitive, 74, 75, 88, 91
  - hidden, 73
  - input, 68
  - output, 76
- learning, 5, 11, 112, 217
  - ART1
    - fast, 94, 95
  - ART2
    - fast, 104
  - batch, 219
  - Bayesian, 275, 276
  - constant, 12, 17, 22, 27, 81, 107, 142, 163, 181, 219
    - adaptive decreasing factor, 22
    - adaptive increasing factor, 22
    - error backpropagation threshold, 17
    - flat spot elimination, 20, 27
    - momentum, 20, 27, 221
  - convergence, 146
  - incomplete, 38, 40, 45
  - reinforced, 112
  - sequential, 219
  - set, 5, 112, *see set, learning*
  - speed, 221
  - stop, 218
  - supervised, 5, 11, 112, 182
  - unsupervised, 5, 29, 31, 39, 112, 243
- Learning Vector Quantization, *see LVQ*
- least squares technique, 182
- lexicographic convention, 65
- linear discriminant analysis, 349
- linear separability, 129, 132, 146
- loss matrix, *see matrix, loss*
- LVQ
  - LVQ1, 366
  - LVQ2.1, 367
  - LVQ3, 368
  - OLVQ1, 366
- macrostate, 207
- Markov
  - chain, 310
  - properties, 308
- matrix
  - covariance, 198
    - between-class, 148, 151
    - within-class, 148, 151
  - loss, 117, 201
  - positive definite, 329
  - pseudo-inverse, 140
- memory, 20, 22, 24
  - associative, 47
    - interpolative, 47
  - autoassociative, 48, 54
  - crosstalk, 52
  - heteroassociative, 47, 67
- Hopfield
  - continuous, 57, 60, 322
  - discrete, 54, 60, 322
  - gain parameter, 57
- saturation, 52
- microstate, 207
- misclassification, 304
  - penalty, 117, 118
- missing data, 97, 239, 306
- mixture model, 179, 194
- mixture of experts, 271
- moment, 248
  - central, 248
  - regular, 248
- momentum, *see algorithm, backpropagation, momentum*
- Monte Carlo method, 295
- multiplicity, 207
- Nadaraya-Watson estimator, 178
- nat, 211
- network
  - ART, 85, 155
  - ART1, 85
  - ART2, 100
  - autoassociative, 245
  - backpropagation, 9
  - BAM, 48
  - cascade correlation, 264
  - committee, 218, 268
  - CPN, 67, 155
    - architecture, 67
  - feedforward, 3, 4, 9, 153, 154, 170
  - growing method, 264
  - higher order, 5, 250
  - Hopfield
    - continuous, 57
    - discrete, 54
  - Kohonen, *see network, SOM*
  - layer
    - hidden, 26
    - output, 197
    - performance, 113
    - pruning method, 264
    - recurrent, 3
    - SOM, 4, 29
  - neuron, 3
    - excitation, 68
    - gain control, 85, 88
    - hidden, 72
    - instar, 71, 73
    - neighborhood, 31, 39, 41, 42
      - function, 41
    - output, 70
    - outstar, 76
    - pruning, 268
    - reset, 85
    - saliency, 268
    - winner, 41
  - neuronal neighborhood, *see neuron, neighborhood*
  - Newton direction, 233
  - noise, 52, 97, 98, 186, 253
  - norm
    - $L_R$ , 192
    - Euclidean, 102, 107, 189
  - NP-problem, 60

Occam factor, 293  
 operator  
     Nabla, 126  
 outlier, 113, 188, 193, 343  
 overadaptation, 142  
 overtraining, 16  
 Parzen window, *see* function, kernel path, 307  
 pattern, 111  
     reflectance, 71  
     space, 112  
 perceptron, 136, 144, 179  
 Polak-Ribiere formula, 230  
 postprocessing, 237  
 prediction, 191  
 preprocessing, 237, 248  
 principal component, 243, 245  
 probability  
     class-conditional, 342  
     density, 343  
     distribution, 342, 343  
     doubt, 117  
     join, 341  
     misclassification, 116  
     posterior, 116, 197, 342  
     prior, 341  
 pruning, 266  
 random walking, 296  
 Rayleigh quotient, 38, 328  
 recursive model, 314  
 reflectance pattern, *see* pattern, reflectance  
 regression, 112  
 regularization, 115, 247, 255  
     parameter, 176, 248, 255  
     weight decay, 280  
 reject area, 119  
 rejects, 113  
 representation  
     marginal, 313  
     potential, 314  
     set-chain, *see* representation, marginal  
 risk, 117  
     averaging, 299  
 Self Organizing Maps, *see* SOM  
 sequential parameter estimation, 123  
 set  
     learning, 10, 114  
     test, 15, 199  
     training, 253  
     validation, 218, 258  
 signal function, *see* function, activation  
 SOM, 40  
 SuperSAB, *see* algorithm, backpropagation, SuperSAB  
 theorem  
     noiseless coding, 211  
 training, *see* learning  
 variance, 191, 218, 238, 254, 343