DD

DFUB 95/16

NEURAL 2.00

A Program for Neural Net and Statistical Pattern Recognition*

R. Odorico⁺

University of Bologna, Department of Physics
and
Istituto Nazionale di Fisica Nucleare, Sezione di Bologna
Via Irnerio 46, 40126 Bologna, Italy

SW 95 47

Abstract

A neural net program for pattern classification is presented, which includes: i) an improved version of Kohonen's Learning Vector Quantization (LVQ with Training Count); ii) Feed-Forward Neural Networks with Back-Propagation training; iii) Gaussian (or Mahalanobis distance) classification; iv) Fisher linear discrimination. Back-Prop trainings with emulations of Intel's ETANN and Siemens' MA16 neural chips are available as options. The program has been developed for High Energy Physics applications.

* Program file available from WWW URL: <http://www.bo.infn.it/preprint/odorico.html>

+ E-mail: odorico@bo.infn.it

PROGRAM SUMMARY

Title of program:	NEURAL
Catalogue number:	
Program obtainable from:	CPC Program Library, Queen's University of Belfast, N. Ireland.
Computer for which the program is designed and others on which it is operable:	Any computer running a FORTRAN 77 compiler. The program has been tested on a Alpha VAX workstation running under VMS.
Programming language used:	FORTRAN ANSI/77
High speed storage required:	It depends on the complexity of the problem. Can be modified by changing array dimension parameters.
No. of bits in a word:	32 or more.
No. of magnetic tapes required:	none.
Other peripherals used:	none.
No. of cards in combined program and test decks:	11814 (11734 for NEURAL file alone).
CPC Library subprogram used:	none.
Keywords:	neural nets, learning vector quantization, back propagation, multilayered perceptron, feed forward networks, statistical discrimination, pattern classification.
Nature of physical problem:	Pattern classification.
Method of solution:	Four different methods for pattern classification are employed: i) Learning Vector Quantization with Training Count, ii) Feed-Forward neural networks with Back-Propagation training, iii) Gaussian classification, iv) Fisher's linear discriminant.
Restrictions on the complexity of the problem:	none.
Typical running time:	It depends on the complexity of the problem. The test run was completed in 140 sec of cpu time on a Alpha VAX 3000-400.
Reference:	A. Cherubini and R. Odorico, Comp. Phys. Commun. 72 (1992) 249.

1. Introduction

The identification of specific physical features in the events produced at high energy colliders poses special challenges, which can be hardly met with the time-honored but simplistic discrimination techniques used so far. That is especially true when one is forced to exploit so called "jet signatures", yielding higher statistical rates at the price of easier confusion with the abundant multijet event background due to more conventional hadronic dynamics.

The utilization of statistical discrimination to single out proton + antiproton $\rightarrow$ top-quark + antitop-quark events from background was already discussed several years ago in [1] and more recently in [2]. Statistical discrimination reaches the Bayesian limit when the event classes one wants to discriminate have distributions which are normal (i.e. Gaussian) in the pattern space, i.e. the hyperspace spanned by the vectors of event feature variables used to code events. In other words, for such distributions there is no way to improve the classification performance over what can be achieved by statistical discrimination, the absolute upper bound on performance being represented by the amount of overlap among class distributions [3]. That is no longer true if the distributions appreciably deviate from normal behavior.

Discrimination can also be achieved by means of Neural Networks (NN). The most successful NN architectures exploited to date are represented by Kohonen's Learning Vector Quantization (LVQ) [4] and Feed Forward Neural Networks (FFNNs) with Back Propagation (BP) training [5].

The present FORTRAN program represents an extension of program LVQNET [6] including also FFNNs trained with BP. Specifically it codes:

i) an improved version of LVQ, Learning Vector Quantization with Training Count (LVQTC) [7], in which the detailed counting of the degree of training received by each neuron is exploited (the original LVQ can be recovered by specific settings of the parameters). The training algorithm makes use of information from the pooled-over-classes covariance matrix for neuron initialization and for setting the distance metric.

ii) The BP algorithm for training FFNNs [5]. Optional emulation during training of the neural chips ETANN, an analog device produced by Intel [8], and MA16, a 16-bit I/O digital microprocessor produced by Siemens [9], are included.

iii) Statistical discrimination based on the Gaussian classifier and on the Fisher's linear discriminant variable [10,3].

The relative performance of the various classifiers can largely depend on the specifics of the problem and on the choice of the feature variables. Hence, it is a wise practice to try several of them on the problem under study. A comparative assessment of the classifiers in a High Energy Physics (HEP) problem is given in [11]. One can make the following points: i) No classifier can outperform the Gaussian if class

distributions are normal, and the same is true for the Fisher discriminant if class distributions also have equal shapes; a strong deviation from normal shapes may be due e.g. to the presence of concavities in the distributions; ii) BP (where neurons are associated with hyperplanes in pattern space) typically requires fewer parameters than LVQTC (where neurons are associated with points), and then tends to have better generalization properties; iii) LVQTC is better suited to handle problems with complex class interpenetrations; iv) BP training is much longer (by typically one or two order of magnitudes) than LVQTC, and may get stuck into non-optimal configurations.

Note that the static (external) connection of files to I/O units (applying to the IBM/VM-CMS system), which was optionally available in LVQNET, is no longer supported.

2. Statistical Discrimination

Let us suppose that class distributions are normal in the pattern space of s vectors, i.e. of the form:

$$p(s) = G \exp\{ - 0.5 (s - \langle s \rangle)^T C (s - \langle s \rangle) \}$$

where c is the inverse of the class covariance matrix C :

$$C(i,j) = \langle (s(i) - \langle s(i) \rangle) ((s(j) - \langle s(j) \rangle)) \rangle$$

$s(i)$ being the i -th variable of s , and $\langle x \rangle$ denoting the mean value of x over the class distribution. G is proportional to the global probability of the class.

Once c , $\langle s \rangle$ and G are known for all classes, one can calculate the $p(s)$ of each class for an unknown class pattern s and assign s to the class with the highest $p(s)$. Such a classifier reaches the Bayesian limit, i.e. no other classifier can outperform it, given the overlaps among the various class distributions. $\sqrt{(s - \langle s \rangle)^T C (s - \langle s \rangle)}$ is called the Mahalanobis distance of s from the centroid $\langle s \rangle$ of the class. The above classification amounts to assign s to the class whose centroid has minimum Mahalanobis distance from s .

If there are just two classes C_1 and C_2 and they have equal covariance matrices, a Bayesian classifier is provided by

$$\ln\{p_1(s)/p_2(s)\} = F(s) + \ln(G_1/G_2)$$

It is easy to find that

$$F(s) = f_0 + f^T s$$

where f is the vector

$$f = c (<s_1> - <s_2>)$$

with $<s_1>$ and $<s_2>$ being the centroids of C_1 and C_2 , and

$$f_0 = - 0.5 f (<s_1> + <s_2>)$$

The Fisher variable $F(s)$ [10,3] is linear in s and much faster to compute than the Mahalanobis distances from the two class centroids.

To the extent to which class distributions can be well approximated by normal distributions, no classifier (including neural ones) can outperform the Mahalanobis distance. Similarly, when having two class distributions well approximated by normal distributions with the same covariance matrices, no classifier can outperform the linear Fisher discriminant. Fisher's linear discrimination can also be extended to handle more than two classes, but we are not going to discuss that (e.g., for three classes one considers the subspace orthogonal to the straight line connecting $<s_1>$ and $<s_2>$, and then introduces another linear variable relating to the common $<s_1>$, $<s_2>$ point and to $<s_3>$ in such a subspace in the same way as the first Fisher's variable relates to $<s_1>$ and to $<s_2>$).

Class centroids and covariance matrices can be determined by calculating the corresponding averages with a single sweep over the set of class certified training patterns. With Fisher's linear discrimination the full training statistics is employed for the determination of just the Fisher's vector f , which has the same number of parameters as a neuron in a standard neural net: a point to consider when dealing with limited training statistics. The statistical error of the Fisher's vector f components is given by

$$\text{err}\{f(i)\} = \sqrt{c(i,i) (1/N_1 + 1/N_2) + [f(i)^2 + (f \cdot (<s_1> - <s_2>)) c(i,i)] / (N_1 + N_2)}$$

where N_1 and N_2 are the number of training patterns for classes C_1 and C_2 , respectively [3]. That can also be used for a test of redundancy: if $\text{err}\{f(i)\} > |f(i)|$, it is sensible to discard the i -th variable in the pattern vectors, thus simplifying the pattern representation. (There are also other tests of variable redundancy, which however require combined consideration of all variables: their practical implementations are controversial.)

Statistical discrimination can be used in much the same way as single-variable cuts. For instance, in a typical problem of discriminating a signal of interest from the associated backgrounds one can consider the variable given by the estimated contamination from other classes of patterns classified as signal:

$$\text{COPGAU} = 1 - p_{\text{signal}}(s) / p_{\text{tot}}$$

where p_{tot} is the sum over the $p(s)$ of all classes, $p(s)$ being the normal density function of a class. One can look at the distributions of certified signal and background events in this variable, and then decide about a suitable cut making a trade-off between purity of the signal and acceptance efficiency for it.

3. Learning Vector Quantization with Training Count (LVQTC)

Vector quantization amounts to a subdivision of the pattern space in populated regions (so called Voronoi tessellation) described by some suitably defined centroid vectors. Learning vector quantization (LVQ) is an algorithm, devised by Kohonen [4], which adaptively builds up vector quantization from a training set of patterns. It can be used for the sake of data compression or, if more pattern classes are present, for pattern classification.

Learning vector quantization with training count (LVQTC) represents a modification of the original LVQ scheme, where additional neuron attributes are introduced keeping track of the training undergone by neurons, with the possibility of replacing neurons with poor training performance and of creating new neurons when they are needed [7]. Aim of LVQTC is to classify pattern vectors s according to pattern classes $C_1, C_2, C_3 \dots$.

Neurons are defined by the following set of attributes:

- i) a reference vector m in the pattern space;
- ii) a class label;
- iii) a counter for each class $P_1, P_2, P_3 \dots$, storing the number of times vectors of that class have trained the neuron (training counters).
- iv) a vector w in the pattern space, representing the centroid of wrong class (i.e. different from the neuron class) patterns which have trained the neuron.

Initialization of neurons makes use of the information on class distributions available from the class certified input patterns used to train the net. The initial total number of neurons is taken as a small fraction (about 5% or less) of the total number of input patterns. Neurons are allocated to classes proportionally to the volumetric and linear sizes of their distributions in the pattern space, in units of the square root of the pooled-over-classes variance for each pattern variable. Class volume is estimated as the square root of the determinant of the corresponding covariance matrix (times a factor of 2 for each pattern variable), and the class linear size as twice the square root of its trace. Neuron reference vectors m are initialized by training patterns of the same class, taken at random.

Training of neurons is arranged in a succession of epoches. For each epoch the following steps are taken:

- i) Neuron resetting: set all neuron training counters and w 's to zero.
- ii) Neuron training: present the whole training set of class-labeled vectors $s(t)$, $t = 1, 2, 3 \dots$, choosing the pattern class at random each time. For each $s(t)$ find the neuron with the closest reference vector, m_c . Let the training vector $s(t)$ belong to class C_s , and suppose that m_c is labeled according to class C_c . Increment the training counter P_s of the neuron for class C_s by 1. Update m_c , leaving the other neuron reference vectors unchanged, according to:

$$\begin{aligned} m_c(t+1) &= m_c(t) + (\alpha_r/P_{tot})[s(t)-m_c(t)] & \text{if } C_s = C_c \\ m_c(t+1) &= m_c(t) - (\alpha_w/P_{tot})[s(t)-m_c(t)] & \text{if } C_s \neq C_c \end{aligned}$$

α_r and α_w are two distinct learning parameters. One should take: $\alpha_w < \alpha_r < 1$. $P_{tot} = P_1 + P_2 + P_3 \dots$ is the current sum over all training counters of the m_c neuron. α_r and α_w are monotonically decreasing with the number of epoches: for each successive epoch they are reduced by a factor $FREDAL < 1$. If $C_s \neq C_c$, update the w of the m_c neuron.

- iii) Neuron pruning, merging and creation: after the presentation of the training set and before a new epoch is started, prune undertrained neurons, merge close neurons of the same class and create new neurons to reduce neuron contamination from wrong class training patterns, according to the following rules applied in order.

- a) Neuron pruning: eliminate neurons with

$$P_{tot} = P_1 + P_2 + \dots < CPRUNE$$

where $CPRUNE$ is a cutoff parameter. Replace the vectors m and w of the same-class neuron closest to the pruned one by a sum of the m 's and w 's of the two weighted by the corresponding P_{tot} 's, and add to its training counters those of the pruned neuron.

- b) Neuron merging: merge pairs of neurons of the same class for which

$$D_w/D_s > RMERGE$$

where D_s is the distance between the neurons, D_w is the minimum distance of the two neurons from neurons of other classes, and $RMERGE$ is a cutoff parameter. The resulting neuron has m , w and training counters determined as in the case of neuron pruning.

- c) Neuron creation: if a neuron has the maximum P_w of wrong class training counters larger than $CPRUNE$, create a neuron of the same class as that of P_w with a vector m equal to the w of the original neuron.

Training stops when for three successive epoches the number of right classifications on the training set is not larger than $(1+FCONVG)$ times the number of right classifications of the previous epoch, where $0 < FCONVG < 1$.

At the end of the last epoch, no neuron pruning, merging and creation is made. Training counters are recalculated by reading the whole set of training patterns and keeping neurons frozen. At the same time, one calculates the neuron contamination by wrong class trainings, COPNEU, and a neuron radius, RADNEU, as follows.

COPNEU is defined by:

$$\text{COPNEU} = A_x / A_{\text{tot}}$$

where

$$A_{\text{tot}} = A_1 + A_2 + A_3 + \dots$$

with

$$A_i = G_i P_i / N_i$$

G_i and N_i being the global probability and the total number of training patterns for class C_i , respectively, P_i being the training counter for class C_i of the neuron, and

$$A_x = \text{sum over all } A_i \text{ except } A_k$$

k indicating the class C_k which labels the neuron. A similar quantity, COXNEU, is calculated: it is defined in the same way as COPNEU, but removing the class probability factors G_i .

RADNEU is calculated as the square root of the mean square (r.m.s.) of the distance from the neuron of training patterns of its same class which have it as the closest neuron.

The neuron quantities COPNEU, COXNEU and RADNEU are used, together with P_{tot} , in the classification process.

Classification of a pattern vector s of unknown class is carried through by assigning the vector to the class of its closest neuron. An estimate of the classification error is provided by the number of wrong class trainings received by the neuron. Also, if the neuron has a too low P_{tot} or is too distant from s the classification may be considered uncertain. Then, the classification is considered unreliable if for the closest neuron one has:

$$P_{\text{tot}} < \text{CCLSTA}$$

$$f_{\text{cont}} > \text{CCLCON}$$

$$D > \text{DISTMX}$$

where CCLSTA, CCLCON and DISTMX are cutoff parameters; f_{cont} is taken equal to the COPNEU (default) or to the COXNEU of the neuron; D is the distance of s from the neuron measured in units of its RADNEU.

Distance metric can be chosen from a variety of options. Straightforward Euclidean metric is fast to calculate, but it may be inadequate if pattern variables range over widely different scales. A wiser choice is provided by a Euclidean metric weighted by the inverses of the pooled-over-classes variances of the single variables, which does not cost a great loss in computing speed. In principle, an optimal distance metric is given by the Mahalanobis distance associated with the pooled-over-classes covariance matrix, but that takes a much longer time to compute.

Two important consequences of keeping track of the training count for each neuron are:

- i) The number of neurons assigned to each class must no longer be proportional to the global probability for the class, as in standard LVQ. In fact, while in the latter the point density of neurons for each class represents the only ingredient to (statistically) control classification in overlap regions, in LVQTC this role is largely taken over by the training count. That can be exploited by assigning relatively few neurons to classes which are concentrated in small regions of the pattern space, and more neurons to classes which are spread out over large regions. In this way, with a given total number of neurons, one can better represent the shapes of the class distributions.
- ii) Similarly, the number of training patterns for each class must no longer be proportional to the global probability for the class, as in standard LVQ. As for neurons, one can more usefully allocate the total number of training patterns to classes according to the effective sizes of their distributions in the pattern space, so as to have a more uniform neuron training. That may also be expedient when dealing with limited samples of training patterns for some, or all, classes: all training vectors can be exploited without having to leave part of them unused so as to satisfy the constraint of proportionality to class probabilities.

Aim of LVQTC is to improve the purity of classification by increasing the degree of adaptation of the neuron net structure to the problem at hand. That is achieved by making a fuller use of the information available on the distribution of training patterns, which is stored into the neuron training counters. Neurons are left more sparse in regions of small class overlap and concentrated in regions of substantial class overlap. Undertrained neurons which would cause uncertain classifications and slow down training convergence are removed. Training counters, together with neuron "radius" information, are also of help in estimating the reliability of classifications.

On a general ground, LVQ can be viewed as a 3-layered neural network: i) the first, input, layer has as many neurons as the pattern variables; ii) the second, intermediate layer, contains the neurons proper; iii) the third, output, layer has as many neurons as the number of classes. The intermediate neurons are governed by a winner-takes-all dynamics and compete for being corrected by the excitations of input

neurons. Each one of them is connected to just one output neuron (according to its class label), whose excitation can be either 0 or 1. With LVQTC, the excitation of an output neuron can be identified with the complement to 1 of the contamination of the intermediate neuron firing into it, i.e. the neuron purity, and thus varies continuously between 0 and 1. Also, the firing can be inhibited if the intermediate neuron has not received enough training or if the input pattern is too distant from it. An advantage with respect to other NN architectures is that at the end of training the reference vectors of intermediate neurons can be directly interpreted as "typical" class patterns.

LVQTC classification is expected to outperform statistical discrimination when class distributions in the pattern space are appreciably non-normal. In particular when, differently from the convex shape of a normal distribution, class distributions develop concavities.

Overlaps between class distributions can be estimated from LVQTC. Let us define the overlap between the normalized distributions $p_1(s)$ and $p_2(s)$ of classes C_1 and C_2 , respectively, as

$$O(1,2) = 1 - \int | p_1(s) - p_2(s) | ds / \int | p_1(s) + p_2(s) | ds$$

(so that the overlap of a class distribution with itself is one, and the overlap of two non-overlapping class distributions is zero). Then $O(1,2)$ can be estimated from the LVQTC neuron training counters as

$$O(1,2) = 1 - \sum_n | P_1(n) G_1/N_1 - P_2(n) G_2/N_2 | / [G_1 + G_2]$$

where the sum is extended over all neurons, $P_i(n)$ is the training counter of neuron n for class C_i , G_i is the global probability and N_i is the number of training patterns for class C_i .

The various LVQTC training and classification parameters should be varied to achieve optimal classification of an independent test set of class certified vectors. α_r and α_w can be increased to improve speed or decreased to refine convergence (initial values of about 0.1 are typically used). Decreasing CPRUNE will increase the number of neurons and thus resolution in the feature space, while decreasing it will make neuron positions more reliable statistically and thus classification more stable. A high value of RMERGE will inhibit neuron merging even when neurons are very close to each other and therefore redundant, whereas a low value for it will bring up the level of neuron contamination. The total number of neurons can be increased to better represent details of the shapes of the class distributions. However, the number of neurons should be kept much smaller than the number of training patterns, to avoid that neurons simply act as a look-up table for the training patterns, which would hamper generalization. Also, a small number of neurons speeds up classification. The classification cutoff parameters CCLSTA, CCLCON and DISTMX should be chosen according

to how much purity one wishes to achieve in the classification process at the price of depriving classification efficiency. Parameter tuning can be realized by embedding LVQTC in a minimization program and choosing a suitable cost function (e.g., the number of misclassification on the test set of patterns). Each call from the minimization program is to be answered by a full training and classification LVQTC run. A robust minimization algorithm not using gradient information, like e.g. Simplex, would be adequate. LVQTC training is typically fast enough to allow for that.

4. Feed Forward Neural Network with Back-Propagation Training (BP)

In a Feed Forward Neural Network (FFNN) the relevant neurons are not associated with points as in LVQTC but rather with hyperplanes in pattern space. In such a net, neurons (or nodes) are arranged in successive layers, the excitations of neurons in a layer being determined by the excitations of neurons in the previous layer (hence the name of Multi Layered Perceptrons sometimes used for nets of this type).

FFNN Architecture. The excitations of nodes in the input layer ($L=0$), one for each pattern variable, are directly given by the values of the corresponding pattern variables $s(k)$:

$$x(k, L=0) = s(k)$$

The excitation $x(i, L=1)$ of the neuron i in the next layer is determined by feeding the value of the linear expression

$$a(i, L=1) = \sum_k w(i, k, L=1) x(k, L=0) + h(i, L=1)$$

into a saturating transfer function

$$x(i, L=1) = g\{ a(i, L=1) \}$$

$a(i, L=1)$ can be visualized as the distance of the pattern from a hyperplane, whose orientation and position are determined by the weight parameters $w(i, k, L=1)$ and the threshold (or bias) term $h(i, L=1)$ associated with the neuron.

Several layers can be included, the excitations of their neurons are determined by iterating the above procedure over the intermediate, or hidden, layers. The last, output, layer may consist of several nodes, but one is sufficient if the classification classes are just 2. If the classes are more than 2, it is convenient to have an output node for each class, responding to patterns of that class.

A FFNN with no hidden layers and one output node amounts to a Fisher discriminant. In general, a FFNN can be seen as a combination of Fisher discriminants, represented by the individual neurons. The non-linearity of the transfer function is critical: without it the role of the hidden layers would be ineffectual, each output node excitation simply becoming a linear function of the input patterns.

FFNN Training. The parameters of the net are determined by introducing a cost function E , gauging the discrepancy between current outputs $o(i)$ ($= x(i, L_{out})$) and the desired (target) ones $t(i)$ over a training set of class-certified input patterns, and by correcting the parameters so as to reduce E to some minimum value. Back Propagation (BP) [5] amounts to the conventional gradient descent method applied for this sake. The involved derivatives are fast to compute because of: i) the structure of E , which simply adds up the individual contributions from the training patterns; ii) the iterative form over layers of the chain derivation rules, allowed by the FFNN structure; iii) the simple form taken by the derivatives of the most used transfer functions.

Defining

$$\text{delta}(i, L) = \frac{\partial E}{\partial a(i, L)}$$

one can write

$$\frac{\partial E}{\partial w(i, k, L)} = \text{delta}(i, L) \cdot x(k, L-1)$$

$$\frac{\partial E}{\partial h(i, L)} = \text{delta}(i, L)$$

Using the chain rule for derivation, it is easy to verify the following back-propagating recurrence formula

$$\text{delta}(k, L) = \sum_i \{ \text{delta}(i, L+1) \cdot w(i, k, L+1) \cdot g' \{ a(i, L) \} \}$$

where $g'\{x\}$ is the derivative of $g\{x\}$.

The cost function E is the sum of the individual input-pattern contributions E_p :

$$E = \sum_p \{ E_p(p) \}$$

The following types of pattern cost function E_p are considered in the program (to be selected by the KECOST parameter):

i) The Euclidean cost function (KECOST=0):

$$E_p = 0.5 \sum_i \{ [o(i) - t(i)]^2 \}$$

which gives

$$\text{delta}(i, L_{\text{out}}) = [o(i) - t(i)] g' \{ a(i, L_{\text{out}}) \}$$

ii) The cost function of [12] (KECOST=1):

$$E_p = - \sum_i \{ [t(i) - g_{\min}] \ln[o(i) - g_{\min}] + [g_{\max} - t(i)] \ln[g_{\max} - o(i)] \}$$

where $g_{\min}$ and $g_{\max}$ are the minimum and maximum values of g .

It gives

$$\text{delta}(i, L_{\text{out}}) = g' \{ a(i, L_{\text{out}}) \} \{ [g_{\max} - t(i)] / [g_{\max} - o(i)] - [t(i) - g_{\min}] / [o(i) - g_{\min}] \}$$

The following types of transfer function $g(x)$ are considered in the program (to be selected by the KTSFE parameter):

i) Sigmoid (KTSFE=1):

$$g(x) = 1/(1+\exp(-x)) = 0.5 (1+\tanh(x/2)) ;$$

$$g'(x) = g(x) (1 - g(x)) ;$$

$$g_{\min} = 0 ; g_{\max} = 1 .$$

ii) Hyperbolic tangent (KTSFE=2):

$$g(x) = \tanh(x) ;$$

$$g'(x) = (1 - g(x)) (1 + g(x)) ;$$

$$g_{\min} = -1 ; g_{\max} = 1 .$$

iii) Breit-Wigner (or Laurentian [13]) (KTSFE=3):

$$g(x) = \frac{1}{(1+x^2)} ;$$

$$g'(x) = -2x g(x)^2 ;$$

$$g_{\min} = 0 ; g_{\max} = 0 .$$

When adopting a sigmoid or hyperbolic tangent $g(x)$, the pattern cost function of [12] gives:

$$\text{delta}(i, L_{\text{out}}) = [o(i) - t(i)] \quad \text{for the sigmoid;}$$

$$\text{delta}(i, L_{\text{out}}) = 2 [o(i) - t(i)] \quad \text{for the hyperbolic tangent;}$$

i.e. it gives essentially the same delta's as the Euclidean cost function, but without the $g'\{a(i, L_{out})\}$ factor. That has the advantage of removing a cause for the slowing down of the convergence process in the neighborhood of the E minimum.

Again in order to reduce the chances of convergence stalling, a small term (GPADD parameter) is added to $g'(x)$.

After averaging weight and threshold derivatives over a number of training patterns (= NUPDAT), these parameters are updated according to the correction formula:

$$w(i,k,L) \rightarrow w(i,k,L) + d(w(i,k,L))$$

$$h(i,L) \rightarrow h(i,L) + d(h(i,L))$$

with

$$d(w(i,k,L)) = -\eta d(E)/d(w(i,k,L)) + \alpha_{bp} d'(w(i,k,L))$$

$$d(h(i,L)) = -\eta d(E)/d(h(i,L)) + \alpha_{bp} d'(h(i,L))$$

where $d'(w(i,k,L))$ and $d'(h(i,L))$ refer to the previous update, and η and α_{bp} are user-changeable parameters. The momentum terms, controlled by α_{bp} , are meant to reduce fluctuations in the updating process. In principle, in the spirit of the gradient descent method NUPDAT should cover the full training set. However, heuristically convergence turns out to be faster by making more frequent updates.

Initially weights and thresholds are generated uniformly at random in the interval $-WIDTHW$, $+WIDTHW$. They can also be read from a file (see command WEIGHTINI, file fwein). By the command WEIGHTFILE a file with the same format can be generated at the end of training, holding the best values found for weights and thresholds.

Training is terminated after NEPOBP epoches if $NEPOBP > 0$, one epoch corresponding to a full pass over the training set of patterns. If $NEPOBP < 0$, training proceeds for up to a maximum of $|NEPOBP|$ epoches and is stopped if the convergence criterion, controlled by the FCVGBP parameter, is met.

The training set is used as follows. In order to reduce the chances of overtraining, it is divided in two equal parts: one is used for determining the parameter updates, the other for verification of the convergence criterion. During training, one pattern for each class is read in succession, following the user provided order. If class files have different pattern lengths, exhausted classes are read over again: an epoch is considered completed when the longest class file has been fully read. Pattern variables may be rescaled, so has to stay within some interval, see the REASCALEBPINP command. Rescaling conditions are

kept in the training file, generated at the end of training, and are automatically applied in CLASSIFY and TEST tasks, as well as in slave mode classification, without any required intervention by the user. The same is true for MASK commands, screening out unwanted pattern variables.

All training parameters can be set by the user, overriding defaults, by the commands TRAINB1 for the net architecture, TRAINBP2 for the floating-point parameters, and TRAINBP3 for the integer parameters. The output targets can be set by the TARGETBP command. The default architecture has one hidden layer, with one output node if there are two classes (target = gmax for the 1st class and = gmin for the 2nd class) and one output node for each class otherwise (target = gmax for patterns belonging to the class and = gmin otherwise). As a rule of thumb, the default number of hidden neurons is set equal to $\{\text{\#input_nodes} \cdot \text{\#output_nodes}\}^{1/2} + 2$.

FFNN Classification of a pattern is done by assigning it to the class with the closest target to the corresponding FFNN output. For this sake Euclidean distance is adopted in the program, independently of the cost function used during training. Other classification criteria can be applied to the FFNN output by the user.

Chip Emulations during training and other tasks are optionally available for two neural chips: i) the analog ETANN device by Intel; ii) the digital 16-bit I/O systolic microprocessor MA16 by Siemens. They are activated by the commands BPETANN and BPMA16, respectively. As additional output, files are generated for input to the development systems of the two chips. See the Reference section on Usage for details about the commands. For specifics about the emulations see [14-18]. The emulations refer to specific net architectures, which cannot be changed by the user.

5. Generalities on Usage

NEURAL can be run as a stand-alone job or in slave mode, called by another program. In the stand-alone mode it can do training and/or classification of patterns arranged in files. In the slave mode it can do classification only, using a training file generated by a previous stand-alone job, and classifying one pattern at the time by feeding it as an argument into an appropriate subroutine call.

NEURAL is maintained with the PATCHY code management system. The appropriate FORTRAN codes are obtained by means of suitable pilot patches.

NEURAL is entirely written in standard FORTRAN77. It should run, therefore, on any computer with a compiler implementing the FORTRAN77 standard. No "static memory" option is needed for the compiler.

NEURAL does not require the use of external routines. It has its own portable random number generator, based on the add-and-carry algorithm of Marsaglia and Zaman as coded by James [19].

All subroutines of NEURAL have names starting with NE... . All functions of NEURAL have names terminating with ...NE . There are no block data modules. Subroutine and functions are placed in separate PATCHY decks carrying the same names and alphabetically ordered. All common blocks of NEURAL have names starting with NE... and are alphabetically ordered.

A detailed Reference Manual, together with the contents of this article, is contained in the NEURAL file as a patch, meant to be updated for program modifications. Section 6 provides directions for getting it. The following sections contain specifications of the PATCHY structure of the NEURAL file, a primer and sample jobs for the VAX/VMS.

6. PATCHY

PATCHY is a code management system developed at CERN. The NEURAL file is supplied in the form of a PATCHY CARDS file. Instructions for using PATCHY and a PATCHY installation tape are available on request from the CERN DD Division. If PATCHY is not installed, a simplified mock-up of it is available as program MPATCH, described at the end of this section.

The NEURAL PAM file contains the following PATCHes:

MAINPR	minimal main program for NEURAL stand-alone mode.
NECDE	contains all common blocks, etc. with KEEP cards in alphabetical order.
NEURAL	contains NEURAL routines.
TESTS	contains test jobs for NEURAL.
MPATCH	contains simplified mock-up of PATCHY.
NEUTEXT	contains the manual for using NEURAL.

One must run YPATCHY to get from the NEURAL file the desired ASM file containing the selected FORTRAN code ready for compilation. In order to facilitate input to YPATCHY, the NEURAL file contains the following pilot patches:

*NEURAL	+USE selects NECDE, NEURAL. The ASM file in output contains the NEURAL FORTRAN code ready for compilation.
*NEURALM	+USE selects MAINPR, NECDE, NEURAL. The ASM file in output contains the minimal FORTRAN code for a NEURAL stand-alone

job.
 *NEUTEXT +USE selects NECDE, NEUTEXT. The ASM file in output
 contains the manual.

Other pilot patches are present for test runs only:

*TFILES ASM file: program to generate test data files.
 *TSLAV1 ASM file: a minimal test program for slave mode.
 *TSLAV2 ASM file: a more complete test program for slave mode.

In order to get the desired ASM file, it is sufficient to run
 YPATCHY on the NEURAL (cards) file with the following cradle, i.e. input
 control cards read by YPATCHY:

```
+EXE
+USE, (*NEURAL, *NEURALM, *NEUTEXT, *TFILES, *TSLAV1, *TSLAV2).        choose one
+PAM, T=C
+QUIT
```

If PATCHY is not available, one can use the FORTRAN program MPATCH
 as its mock-up, for the limited usage referred to in the manual. The
 program is contained in patch MPATCH (look for line starting with
 "+PATCH,MPATCH", include lines from the next one until the next line
 starting with "+PATCH," excluded). It can be used interactively,
 replying to prompts: give the NEURAL file name as name of the input
 PATCHY file, then the name of the output file on which the selected code
 is to be written, and when prompted for USE cards insert the full PATCHY
 cradles quoted in the manual (cards which are not USE cards or the
 terminal PAM card are ignored). MPATCH does not handle deck
 modifications (card replacements, deletions etc.).

7. Usage: Primer

Let us suppose we wish to classify events into 2 classes, named A
 and B, having relative global probabilities 1. and 3., respectively.
 First of all we must choose a set of event feature variables whose
 values, arranged in a pattern vector, represent each single event. Let
 us suppose we use 4 feature variables. We must then get class certified
 samples of events of classes A and B, reflecting their natural or
 expected distributions (but NOT bound to have their numbers of entries
 proportional to the global probabilities of classes). We must calculate
 their corresponding pattern vectors and arrange them sequentially in 2
 separate files, one for each class. Let us call these files a.dat and
 b.dat respectively. They may be formatted or unformatted. Each file will
 be read by NEURAL, one pattern vector at the time. If the file is
 formatted, each pattern vector is read by:

```
READ(#unit,*) (PATTRN(I),I=1,LRPAT)
```

If the file is unformatted, each pattern vector is read by:

```
READ(#unit) (PATTRN(I),I=1,LRPAT)
```

PATTRN contains the pattern vector and LRPAT is its length, equal to 4 in our example. Let us suppose we use formatted a.dat and b.dat files.

We can now train the net to recognize events. For this we must use NEURAL as a stand-alone job. To get the FORTRAN code, one must run YPATCHY on the NEURAL (cards) file with the following cradle:

```
+EXE
+USE, *NEURALM
+PAM, T=C
+QUIT
```

The FORTRAN code one gets with the *NEURALM pilot patch is a complete FORTRAN program (including a main program) without additions required, with unit 5 connected to the input stream and unit 6 connected to the output stream (these preconnections can be overridden). To run the training job, one must place the following command lines in the input stream:

```
TRAIN
0 'training.dat' 4 0 0 0 0
CLASS
0 'a.dat' 0 0 1. 'A'
CLASS
0 'b.dat' 0 0 3. 'B'
ENDTASK
STOP
```

A formatted file training.dat will be generated, which will be read in a subsequent test or classification run. Printed output will be issued, specifying the complete set-up conditions for the training job, including default parameter values, and giving details about the training and the expected classification performance. In the command lines, the 0's before file names indicate that they are to be taken as formatted (they should be changed into a 1 for unformatted files).

One can make a classification test on independent (test) sets of patterns belonging to classes A and B, thus checking the ability of the net to generalize. The corresponding files atest.dat and btest.dat, respectively, must have the same record arrangement as a.dat and b.dat. The YPATCHY cradle to get the FORTRAN program is the same as that for the training job. The command lines to put in the input stream are now:

```
TEST
0 'training.dat'
TESTCLASS
0 'atest.dat' 0
TESTCLASS
```

```
0 'btest.dat' 0
ENDTASK
STOP
```

Printed output will be issued, specifying the complete set-up conditions for the test job, including default parameter values, with details about the training file, and giving the classification performance on the test set.

Let us now classify a set of unknown events, represented by the same set of feature variables. We have two options: i) arrange their pattern vectors in a file and classify them by a NEURAL stand-alone job; ii) classify them within the same program which generates their pattern vectors (slave mode).

If we choose the stand-alone mode, the file containing the pattern vectors to classify must be arranged like the above files a.dat and b.dat. Let us call this file x.dat, and let us suppose it is formatted. The YPATCHY cradle to get the FORTRAN program is the same as that for the training job. The command lines to put in the input stream are now:

```
CLASSIFY
0 'training.dat' 0 'x.dat' 0 'classf.dat' 0
ENDTASK
STOP
```

Pattern vectors are classified one by one in the output file classf.dat. If the file is formatted, as in our case, its first record can be read by:

```
READ(#unit,*) LRPAT,NCLASS,NODOUT,KNOLVQ,KNOBP,KNOGAU,KNOFIS
```

The subsequent data for each input pattern can be read by:

```
READ(#unit,*) (PATTRN(I),I=1,LRPAT)
READ(#unit,*) LCLLVQ,LCLBP,LCLGAU,LCLFIS
IF(KNOLVQ.EQ.0) READ(#unit,*)
&DNRST, TTCNRS, COPNRS, COXNRS, NNRST
IF(KNOBP.EQ.0) READ(#unit,*) (OUT(LN),LN=1,NODOUT)
IF(KNOGAU.EQ.0) READ(#unit,*)
&DGAUM, COPGAU, COXGAU, (DGAU(L),L=1,NCLASS)
IF(KNOFIS.EQ.0) READ(#unit,*) FISHER
```

For an unformatted file, just drop ",*" in READ statements. PATTRN contains the pattern vector and LRPAT is its length. ABS(LCLLVQ), ABS(LCLBP), ABS(LCLGAU), and LCLFIS are the numbers of the classes into which the pattern is classified according to LVQTC, BP, Gauss and Fisher, respectively. Classes are numbered as 1, 2, 3, ... according to the order of appearance of the corresponding CLASS commands in the original training job. The classification is performed: i) for LVQTC, by the class of the closest neuron according to the adopted metric (KMETRC); ii) for BP, by the class of the closest target according to Euclidean metric; iii) for Gauss, by the class of the closest class-

centroid according to Mahalanobis distance; iv) for Fisher, by the sign of (FISHER-SCLFIS) (class 1 if positive, class 2 if negative). If LCLLVQ is < 0 , the nearest neuron is considered unreliable for classification according to the CCLSTA, CCLCON and DISTMX cuts, for which default values are used in our case. If LCLBP < 0 , the ratio $d/(d+d2)$ is larger than $|CCLBP|$, where d and $d2$ are the euclidean distances from the closest and second closest target, respectively. If LCLGAU < 0 , the estimated contamination (under the assumption of normal distributions) is larger than that given by the CCLGAU cut. The remaining lines contain pattern parameters relevant for classification according to LVQTC, BP, Gauss and Fisher, in order. They are specified in the part of the "Usage: Reference" of the manual containing instructions for the Slave Mode of operation (calls to routines NECLVQ etc.). In the output stream, one finds printed: i) the training set-up and results, as read from training.dat; ii) the complete set-up for the classification job, including default parameter values; iii) a summary of the classification results.

If we choose the slave mode for classification, in order to get the necessary FORTRAN code one must run YPATCHY on the NEURAL (cards) file with the following cradle:

```
+EXE
+USE,*NEURAL
+PAM,T=C
+QUIT
```

The code so obtained must be linked to the user's main program. In the latter, calls to the NEURAL classification routines must be preceded by a single call to the initializing routine NECINI:

```
CALL NECINI(0,'training.dat',0,2,0,MPATTR,MCLASS)
```

The routine prints the training set-up and results, as read from training.dat, and returns in MPATTR and MCLASS the pattern vector length and the number of classes, respectively, for check or possible use (in our case 4 and 2, respectively, should be returned). To classify a pattern vector we must call:

```
CALL NECLVQ(PATTRN,LCLLVQ)
CALL NECBP(PATTRN,LCLBP)
CALL NECGAU(PATTRN,LCLGAU)
CALL NECFIS(PATTRN,LCLFIS)
```

to get classifications of the pattern vector according to LVQTC, BP, Gauss and Fisher, respectively. PATTRN is a 1-dimensional array of single precision type containing the pattern vector, and must be dimensioned to a length equal to or greater than MPATTR (i.e. 4 or larger in our case). LCLLVQ, LCLBP, LCLGAU, and LCLFIS have the same meaning as in the stand-alone mode.

One can combine training, testing and classification into a single stand-alone job. One can use the same set-up of the training job presented above, with the following command cards:

```

TRAIN
0 'training.dat' 4 0 0 0 0
CLASS
0 'a.dat' 0 0 1. 'A'
CLASS
0 'b.dat' 0 0 3. 'B'
ENDTASK

TEST
0 '#'
TESTCLASS
0 'atest.dat' 0
TESTCLASS
0 'btest.dat' 0
ENDTASK

CLASSIFY
0 '#' 0 'atest.dat' 0 'clatest.dat' 0
ENDTASK

CLASSIFY
0 '#' 0 'btest.dat' 0 'clbtest.dat' 0
ENDTASK

STOP

```

(blank lines are ignored.) The option # for specification of the training file to be used in the test and classification tasks means that the training set-up already stored in memory should be employed.

In the stand-alone mode, if one wishes to override the defaults for training, testing and classification parameters in the program, one can use the appropriate command cards. In the slave mode, the classification cutoff parameters can be changed by placing a call

```
CALL NECUTS(CCLSTA,CCLCON,DISTMX,CCLBP,CCLGAU,SCLFIS)
```

immediately after the call to NECINI. Also, in the slave mode one can do the classification according to one's own criteria by using the information available via the /NECLAL/, /NECLAB/, /NECLAF/ and /NECLAF/ common blocks.

In practice, user's requirements for purity and efficiency of classification may vary widely and the classification criteria adopted in the stand-alone mode and in the classification routines of the slave mode may be felt too rigid. Then, the user is advised to use stand-alone mode classification only to have a quick check of the generalization capabilities of the classifiers, and to perform the actual classification by means of the more versatile slave mode, tailoring classification according to the purity/efficiency needs by employing the detailed information available from the common blocks.

8. Usage: Sample Jobs

The specifics of the following job examples, with control cards for the VAX/VMS, concern a simple classification problem. Classes A and B are made of points in the plane uniformly distributed within two disconnected regions and have equal global probabilities. Class A points are contained within a circle of radius $r = 1$ centered at the origin, and class B points are contained within a ring $2 < r < 3$. As pattern features, the cartesian x and y variables are used. The problem specifics can be easily changed to accommodate other classification problems.

First of all, one must get files containing class A and B patterns for training and testing, i.e. the files a.dat, b.dat, atest.dat and btest.dat referred to in the Usage Primer section. After having made available the NEURAL (cards) file as neural.car, these files, with 1000 patterns each, can be generated by submitting the following VAX/VMS command file:

```
-----
$ ! TFILES.COM
$ ! Generates formatted files a.dat, b.dat, atest.dat and btest.dat
$ set def [directory of choice]
$ !
$ ypatchy
neural.car tfiles.for tty tty - - - - :go
+SELF. Cradle for generating test data files
+EXE
+USE,*TFILES
+PAM,T=C
+QUIT
$ for/nolist tfiles.for
$ lin/nomap tfiles
$ run tfiles
$ exit
-----
```

(file tfiles.for in output contains the FORTRAN code of the job.) The pattern files so generated are formatted and actually contain patterns with 4 variables corresponding to the features: x , y , x^2 , y^2 . We have included the two extra features to make a point later about the dependence of the results on the features which are used. In order to select features x and y only, one can include a MASK command in the input to NEURAL.

Now, let us make a training on pattern files a.dat and b.dat, with a test on the certified patterns contained in files atest.dat and btest.dat and two separate classifications of the same. That can be done in a single stand-alone job run by submitting the following VAX/VMS command file:

```

-----
$ ! NEURAL.COM
$ ! Compiles and runs NEURAL stand-alone job
$ set def [directory of choice]
$ !
$ ypatchy
neural.car neural tty tty - - - - :go
+SELF. Cradle for NEURAL stand-alone mode
+EXE
+USE,*NEURALM.          N.B.: not *NEURAL !
+PAM,T=C
+QUIT
$ for/nolist neural
$ lin/nomap neural
$ run neural
TRAIN
0 'training2.dat' 4 0 0 0 0
CLASS
0 'a.dat' 0 0 1. 'A'
CLASS
0 'b.dat' 0 0 1. 'B'
TRFILETITLE
'Two rings. A:  $r < 1$ ; B:  $2 < r < 3$ . Features: x,y'
VARNAMES
'x' 'y' 'x**2' 'y**2'
MASK
1 1 0 0
ENDTASK

TEST
0 '#'
CLASSIFYCUTS
0. 1. 1.E6 1. 0.
TESTCLASS
0 'atest.dat' 0
TESTCLASS
0 'btest.dat' 0
ENDTASK

CLASSIFY
0 '#' 0 'atest.dat' 0 'clatest.dat' 0
CLASSIFYCUTS
0. 1. 1.E6 1. 0.
ENDTASK

CLASSIFY
0 '#' 0 'btest.dat' 0 'clbtest.dat' 0
CLASSIFYCUTS
0. 1. 1.E6 1. 0.
ENDTASK

STOP
$ exit
-----

```

In the train task, there has been included a TRFILETITLE command giving a title to the training file training2.dat, which will appear in the first line of the file, and a VARNAMES command giving names to the pattern variables, for reference in the printout. In the test and classify tasks there has been included a CLASSIFYCUTS command, specifying the classification cutoffs CCLSTA, CCLCON, DISTMX, CCLBP,

CCLGAU and SCLFIS to be employed (the numeric values given correspond to void cuts). From the printout (in the .log file) it will appear that all patterns are exactly classified by LVQTC and BP, while Gaussian classification and Fisher discrimination are doing a bad job. That is to be expected, since the distribution of class B points presents a concavity within which class A points are contained, and the centroids of the two distributions coincide. It is instructive to look at what happens when one also includes the extra feature variables x^2 and y^2 in the patterns. For that, let us eliminate the two cards of the MASK command, so that all pattern variables remain active, and change the TRAIN and TRFILETITLE command cards with the following ones:

```
TRAIN
0 'training4.dat' 4 0 0 0 0
TRFILETITLE
'Two rings. A:  $r < 1$ ; B:  $2 < r < 3$ . Features:  $x, y, x^2, y^2$ '
```

From the printout it will appear that now also Gaussian and Fisher discrimination are doing a perfect job. The x and y components of the Fisher vector have values smaller than errors (a cue to neglect them for Fisher discrimination) and the x^2 and y^2 components have equal values within errors: Fisher learns that $x^2 + y^2$ is the good discrimination variable.

The FORTRAN code for an example of main program using NEURAL in slave mode, to classify patterns read from file atest.dat, can be obtained in file tslavl.for generated by submitting the following VAX/VMS command file:

```
-----
$ ! TSLAV1CODE.COM
$ ! Extracts FORTRAN code for main program using NEURAL in slave mode
$ ! (--> tslavl.for)
$ set def [directory of choice]
$ !
$ ypatchy
neural.car tslavl.for tty tty - - - - :go
+SELF. Cradle for extracting tslavl.for (main program only)
+EXE
+USE, *TSLAV1
+PAM, T=C
+QUIT
$ exit
-----
```

The main program one obtains in this way is a minimal one: it does not use NEURAL classification output available from common blocks. In order to have a more complete sample program in this vein, one can simply replace the pilot patch *TSLAV1 by *TSLAV2 in the above command file. If one likes to run the actual job, one can submit the following VAX/VMS command file:

```
-----
$ ! TSLAV1.COM
```



```

$ ! Compiles and runs program using NEURAL in slave mode
$ set def [directory of choice]
$ !
$ ypatchy
neural.car tslavljob.for tty tty - - - - :go
+SELF. Cradle for full FORTRAN code to run slave mode job
+EXE
+USE,*TSLAV1
+USE,*NEURAL
+PAM,T=C
+QUIT
$ for/nolist tslavljob
$ lin/nomap tslavljob
$ run tslavljob
$ exit
-----

```

(*TSLAV1 can be replaced by *TSLAV2.)

If the CERN Program Library is not available, one can extract the FORTRAN program listed in patch MPATCH, place it in file MPATCH.FOR and substitute in the above command files the lines

```

$ ypatchy
neural.car fileout.for ...

```

with

```

$ for/nolist mpatch
$ lin/nomap mpatch
$ run mpatch
neural.car
fileout.for

```

9. Test Run

The test run job corresponds to the conditions of the NEURAL.COM VAX/VMS job listed in section 8. It also generates the test data files. It does not require the use of the CERN Program Library. The user should provide the NEURAL file as file NEURAL.CAR, and should extract from its patch MPATCH (look for "+PATCH,MPATCH...", until the next "+PATCH,..." line) the FORTRAN program to be placed in file MPATCH.FOR.

References[#]

- [1] R. Odorico, Phys. Lett. 120B (1983) 219; G. Ballocci and R. Odorico, Nucl. Phys. B229 (1983) 1.
- [2] A. Cherubini and R. Odorico, Z. Physik C 47 (1990) 547.
- [3] M. Kendall, A. Stuart and J.K. Ord, "The Advanced Theory of Statistics", Vol. 3, 4th ed., C. Griffin & Co. Ltd., London (1983).
- [4] T. Kohonen, Self-Organization and Associative Memory, 2nd ed., Springer, Berlin (1988).
- [5] D.E. Rumelhart, G.E. Hinton and R.J. Williams, "Learning Internal Representations by Error Propagation", in D.E. Rumelhart and J.L. McClelland (Eds.), Parallel Distributed Processing: Explorations in the Microstructure of Cognition (Vol. 1), MIT Press (1986).
- [6] A. Cherubini and R. Odorico, Comp. Phys. Commun. 72 (1992) 249.
- [7] A. Cherubini and R. Odorico, Z. Physik C 53 (1992) 139; Proc. of the Workshop on "Neural Networks: from Biology to High Energy Physics, Marciana Marina, Isola d'Elba, Italy (1991), p. 515.
- [8] 80170 Electrically Trainable Analog Neural Network Data Booklet, Intel Corp., 2250 Mission College Boulevard, Santa Clara, CA 95052-8125, USA.
- [9] U. Ramacher, J. Beichter and N. Bruels, MA16 -- Programmable VLSI Array Processor for Neural Networks and Matrix-Based Signal Processing, User Description, Version 1.4, Siemens AG, ZFE T ME2, 81730 Munich, Germany (nico@dali9.zfe.siemens.de).
- [10] R.A. Fisher, Ann. Eugen. 7 (1936) 179.
- [11] P. Mazzanti and R. Odorico, Zeit. Phys. C59 (1993) 273.
- [12] A. Van Ooyen and B. Nienhuis, Neural Networks 5 (1992) 465.
- [13] B.G. Giraud et al., Saclay report T94/047 (1994), Neural Networks in press.
- [14] C. Baldanza et al., U. Bologna report DFUB 95/15 (1995).
- [15] C. Baldanza et al., Nucl. Instr. Meth. A361 (1995) 506.
- [16] C. Baldanza et al., U. Bologna report DFUB 95/14 (1995).
- [17] C. Baldanza et al., Proc. AIHENP'95, Pisa, Italy (1995), U. Bologna report DFUB 95/2 (1995).
- [18] C. Baldanza et al., Proc. AIHEP'93, Oberammergau, Germany (1995), p. 391.
- [19] F. James, Comp. Phys. Comm., 60 (1990) 329, and Erratum 69 (1992) 486.

[#] Unpublished papers by the author (including those by Baldanza et al.) can be downloaded as postscript files from URL:
<http://www.bo.infn.it/preprint/odorico.html>

Test Run Input

```

-----
$ ! NEURALTRUN.COM
$ ! Generates test data files, compiles and runs NEURAL stand-alone job
$ ! WITHOUT use of the CERN Program Library
$ ! Files used: NEURAL.CAR
$ !           MPATCH.FOR (extracted from patch MPATCH of NEURAL.CAR)
$ set def [directory of choice]
$ !
$ for/nolist mpatch
$ lin/nomap mpatch
$ !
$ ! Generate test files a.dat, b.dat, atest.dat, btest.dat
$ ! -----
$ run mpatch
neural.car
tfiles.for
+SELF. Cradle for generating test data files
+EXE
+USE,*TFILES
+PAM,T=C
+QUIT
$ for/nolist tfiles.for
$ lin/nomap tfiles
$ run tfiles
$ !
$ ! Generate file neural.for with NEURAL FORTRAN code
$ ! -----
$ run mpatch
neural.car
neural.for
+SELF. Cradle for NEURAL stand-alone mode
+EXE
+USE,*NEURALM.          N.B.: not *NEURAL !
+PAM,T=C
+QUIT
$ !
$ ! NEURAL stand-alone job
$ ! -----
$ for/nolist neural
$ lin/nomap neural
$ run neural
TRAIN
0 'training2.dat' 4 0 0 0 0
CLASS
0 'a.dat' 0 0 1. 'A'
CLASS
0 'b.dat' 0 0 1. 'B'
TRFILETITLE
'Two rings. A:  $r < 1$ ; B:  $2 < r < 3$ . Features: x,y'
VARNAMES
'x' 'y' 'x**2' 'y**2'
MASK
1 1 0 0
ENDTASK

TEST
0 '#'
CLASSIFYCUTS
0. 1. 1.E6 1. 1. 0.
TESTCLASS

```

```
0 'atest.dat' 0
TESTCLASS
0 'btest.dat' 0
ENDTASK
```

```
CLASSIFY
0 '#' 0 'atest.dat' 0 'clatest.dat' 0
CLASSIFYCUTS
0. 1. 1.E6 1. 1. 0.
ENDTASK
```

```
CLASSIFY
0 '#' 0 'btest.dat' 0 'clbtest.dat' 0
CLASSIFYCUTS
0. 1. 1.E6 1. 1. 0.
ENDTASK
```

```
STOP
$ exit
```

Excerpts from Test Run Output

NEURAL Version 2.00 date: 951010

=====

Current values of dimension control parameters:

```

MXCLAS =          2
MXNLAY =          3
MXLPAT =          70
MXLRPA =          70
MXMPAT =       1500000
MXNEUR =          100
MXNODS =          100
MXNPET =          301
MXNWE  =       10000

```

...

Training results

LVQTC:

```

# training epochs =          4
Convergence reached after          1 epochs

```

Overlaps between classes (including input class probs):

class\class	1	2
1	0.100E+01	
2	-0.119E-06	0.100E+01

```

# neurons =          44
# neurons per class:
1:          9;      2:      35;

```

BP:

```

# training epochs =          803
Convergence reached after          793 epochs

```

From the correlation matrix:

#class	class	#patterns	volume	diagonal
1	A	1000	0.587E+00	0.108E+01
2	B	1000	0.740E+01	0.385E+01

Mahalanobis distance between class centroids (c.m. of row class):

class\class	1	2
1	0.000E+00	0.269E+00
2	0.749E-01	0.000E+00

Fisher vector:

name	I	int	FISHV(I)	ERFISV(I)	ERFISV(I) > FISHV(I)
	0		0.478E-02		
x	1	1	-0.399E-01	0.344E-01	
y	2	2	0.657E-01	0.334E-01	
x**2	3		not active		
y**2	4		not active		

Fisher-variable values of class centroids:

```

1:  0.520E-02      2: -0.520E-02

```

Fisher-variable r.m.s. for classes:

```

1:  0.503E+00      2:  0.180E+01

```

Performance test on training set

LVQTC:

Total fraction of right classifications = 1.000000

Pattern class Classification fractions

		1	2
1	A	0.100E+01	0.000E+00
2	B	0.000E+00	0.100E+01

BP:

Total fraction of right classifications = 1.000000

Pattern class Classification fractions

		1	2
1	A	0.100E+01	0.000E+00
2	B	0.000E+00	0.100E+01

Gaussian:

Total fraction of right classifications = 0.5000000

Pattern class Classification fractions

		1	2
1	A	0.000E+00	0.100E+01
2	B	0.000E+00	0.100E+01

Fisher:

Total fraction of right classifications = 0.5285000

Pattern class Classification fractions

		1	2
1	A	0.534E+00	0.466E+00
2	B	0.477E+00	0.523E+00

...