LU TP 93-29
CERN-TH.7135/94
December 1993

JETNET 3.0 – A Versatile Artificial Neural Network Package

Carsten Peterson and Thorsteinn Rögnvaldsson

Department of Theoretical Physics, University of Lund,
Sölvegatan 14 A, S-223 62 Lund, Sweden

Leif Lönnblad

Theory Division, CERN, CH 1211 Geneva 23, Switzerland

Submitted to **Computer Physics Communications**

PROGRAM SUMMARY

Title of Program: JETNET version 3.0

Catalogue number:

Program obtainable from: denni@thep.lu.se or via anonymous ftp from `thep.lu.se` in directory `pub/Jetnet/` or from `freehep.scri.fsu.edu` in directory `freehep/analysis/jetnet`.

Computer for which the programme is designed: DEC Alpha, DECstation, SUN, Apollo, VAX, IBM, Hewlett-Packard, and others with a F77 compiler

Computer: DEC Alpha 3000; installation: Department of Theoretical Physics, University of Lund, Lund, Sweden

Operating system: DEC OSF 1.3

Program language used: FORTRAN 77

High speed storage required: $\approx$ 90k words

No. of bits in a word: 32

Peripherals used: terminal for input, terminal or printer for output

No. of lines in combined program and test deck: 5753

CPC Library subroutines used: none

Keywords: pattern recognition, jet identification, data analysis, artificial neural network

Nature of physical problem
Challenging pattern recognition and non-linear modeling problems within high energy physics, ranging from off-line and on-line parton (or other constituent) identification tasks to accelerator beam control. Standard methods for such problems are typically confined to linear dependencies like Fischer discriminants, principal components analysis and ARMA models.

Method of solution
Artificial Neural Networks (**ANN**) constitute powerful nonlinear extensions of the conventional methods. In particular feed-forward multilayer perceptron (**MLP**) networks are widely used due to their simplicity and excellent performance. The F77 package `JETNET 2.0` [1] implemented "vanilla" versions of such networks using the back-propagation updating rule, and included a self-organizing map algorithm as well. The present version, `JETNET 3.0`, is backwards compatible with older versions and contains a number of powerful elaborate options for updating and analyzing MLP networks. A set of rules-of-thumb on when, why and how to use the various options is presented in this manual and the relation between the underlying algorithms and standard statistical methods is pointed out. The self-organizing part is unchanged and is hence not described here. The `JETNET 3.0` package consists of a number of subroutines, most of which handle training and test data, that must be loaded with a main application specific program supplied by the user. Even though the package was originally mainly intended for jet triggering applications [2, 3, 4], where it has been used with success for heavy quark tagging and quark-gluon separation, it is of general nature and can be used for any pattern recognition problem area.

Restriction of complexity of the problem
The only restriction of the complexity for an application is set by available memory and CPU time. For a problem that is encoded with n_i input nodes, n_o output (feature) nodes, H layers of hidden

nodes with $n_{h(j)}(j = 1, ..., H)$ nodes in each layer, the program requires the storage of $2M_c$ real numbers given by

$$M_c = n_i n_{h(1)} + \sum_{j=1}^{H-1} n_{h(j)} n_{h(j+1)} + n_{h(H)} n_o \qquad (1)$$

Also, the neurons requires the storage of $4M_n$ real numbers according to

$$M_n = n_i + \sum_{j=1}^{H} n_{h(j)} + n_o \qquad (2)$$

In addition one of course needs to at least temporarily store the patterns; $M_p = n_i + n_o$ real numbers.

If second order methods are employed, which keep track of past gradients, the storage requirement increases with $2(M_c + M_n)$. If individual learning rates are used, it increases with an additional $M_c + M_n$.

Typical running time
Running the test-deck problem, which has $M_c = 60$ and $M_n = 16$, for 100 epochs with 5000 training pattern presentations per epoch takes between 30 and 60 CPU-seconds on a DEC Alpha workstation 3000/400, depending on which method that is used. A real-world problem with $M_c = 240$ and $M_n = 34$, using 3770 patterns and training for 1000 epochs, takes 565 CPU-seconds on the same machine.

LONG WRITE-UP

1 Introduction

Feed-forward ANN have become increasingly popular over the last couple of years in feature recognition and function mapping problems in a wide area of applications. High energy physics (**HEP**) is no exception with its demanding on-line and off-line analysis tasks. To date, the most commonly used architectures and procedures are the Multilayer Perceptron (**MLP**) with backpropagation updating and self-organizing networks. Both these approaches were implemented in `JETNET 2.0`. For the self-organizing networks nothing is changed in `JETNET 3.0` and we refer the reader to refs. [1, 4] for information on this part. For the MLP the most important additions and changes concern additional **learning algorithm** variants, **learning parameters** and various tools for gauging performance and estimating error surfaces.

The following learning algorithms are included in `JETNET 3.0`:

- Standard Gradient Descent (back-propagation) [5]
- Langevin Updating [6]

- Conjugate Gradient [7]
- Scaled Conjugate Gradient [8]
- "Quickprop" [9]
- "Rprop" [10]

Also, among other things, the following options are included.

- Dynamic Learning Rates
- Saturation Measurement
- Computation and Monitoring of Hessian Eigenvalues
- Limited Precision

Besides a full description of the functionality and the use of the various `JETNET 3.0` subroutines this writeup also contains a set of "rules-of-thumb" and guidelines on how to use the package in different situations.

However, we emphasize that in addition to feature recognition and function mapping there are ANN applications in HEP that require feed-back networks, which are not included in this package. In particular, we think of optimization networks used for track finding [11, 12, 13, 14, 15, 16].

This write-up is organized as follows. In Sect. 2 we very briefly discuss the basic steps and variants when using feed-forward networks for learning. Discussions and prescriptions on what methods to use in various situations are found in Sect. 3. Some implementation issues with respect to `JETNET 3.0` are contained in Sect. 4. The program components together with switch and parameter descriptions are listed in Sect. 5. Finally Sect. 6 contains a list of technical restricions and Sect. 7 a sample program.

2 Learning in Feed-Forward Artificial Neural Networks

When analyzing experimental data the standard procedure is to make various cuts in observed kinematical variables x_k in order to single out desired features. A specific selection of cuts corresponds to a particular set of feature functions $o_i = F_i(x_1, x_2, ...) = F_i(\vec{x})$ in terms of the kinematical variables x_k. This procedure is often not very systematic and quite tedious. Ideally one would like to have an automated optimal choice of the functions F_i, which is exactly what feature recognition ANN aim at. For a feed-forward ANN the following form of F_i is often chosen

$$F_i(\vec{x}) = g[\frac{1}{T}\sum_j \omega_{ij} g(\frac{1}{T}\sum_k \omega_{jk} x_k + \theta_j) + \theta_i] \tag{3}$$

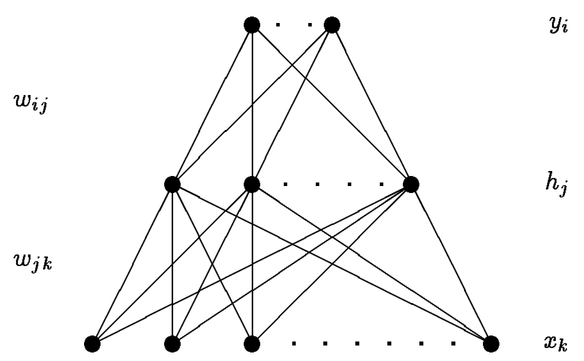

Figure 1: A one hidden layer feed-forward neural network architecture.

which corresponds to the architecture of fig. 1. Here the "weights" ω_{ij} and ω_{jk} are the parameters to be fitted to the data distributions and $g(x)$ is the non-linear neuron *activation function*, typically of the form

$$g(x) = \frac{1}{2}[1 + \tanh(x)] = (1 + e^{-2x})^{-1} \tag{4}$$

The bottom layer (input) in fig. 1 corresponds to sensor variables x_k and the top layer to the (output) features o_i (the feature functions F_i). The hidden layer enables non-linear modeling of the sensor data. Eq. (3) and fig. 1 are easily generalized to more than one hidden layer.

Using eq. (4) for the output assumes that the output variables represent classes and are of binary nature. The same architecture can be used for real function mapping if o_i are chosen linear, in which case the outermost g is removed from the left hand side of (3).

The weights ω_{ij} and ω_{jk} are determined by minimizing an error measure of the fit, e.g. a mean square error

$$E = \frac{1}{2N_p} \sum_{p=1}^{N_p} \sum_i (o_i^{(p)} - t_i^{(p)})^2 \tag{5}$$

between o_i and the desired feature values t_i (targets) with respect to the weights. In eq. (5) (p) denotes patterns. For architectures with non-linear hidden nodes no exact procedure exists for minimizing the error and one has to rely on iterative methods, some of which are described below.

Once the weights have been fitted to the data in this way, using labeled data, the network should be able to model data it has never seen before. The ability of the network to correctly model such unlabeled data is called **generalization** performance.

When modeling data it is always crucial for the generalization performance that the number of data points well exceeds the number of parameters (in our case the number of weights N_ω). For a given set of sensor variables this can be accomplished by

- Preprocessing using e.g. Principal Component Analysis.
- Building in *a priori* known symmetries into the problem – "weight sharing".
- Adding complexity terms to the error (eq. (5)) to **regularize** the network.
- Inspection of the final network to remove redundant parameters.

all of which we will return to later.

2.1 The Back-Propagation Family

Minimizing eq. (5) with gradient descent is the least sophisticated but nevertheless in many cases a sufficient method. It amounts to updating the weights according to the back-propagation (**BP**) learning rule [5]

$$\boldsymbol{\omega}_{t+1} = \boldsymbol{\omega}_t + \Delta\boldsymbol{\omega}_t \tag{6}$$

where

$$\Delta\boldsymbol{\omega}_t = -\eta\frac{\partial E_t}{\partial\boldsymbol{\omega}} = -\eta\nabla E_t \tag{7}$$

Here $\boldsymbol{\omega}$ refers to the whole vector of weights and thresholds used in the network[1].

A momentum term is often also added to stabilize the learning

$$\Delta\boldsymbol{\omega}_{t+1} = -\eta\frac{\partial E}{\partial\boldsymbol{\omega}} + \alpha\Delta\boldsymbol{\omega}_t \tag{8}$$

where $\alpha < 1$.

Initial "flat-spot" problems and local minima can to a large extent be avoided by introducing noise to the gradient descent updating rule of eq. (7). This is conveniently done by adding a properly normalized Gaussian noise term [6]

$$\Delta\boldsymbol{\omega} = -\eta\nabla E + \boldsymbol{\sigma} \tag{9}$$

which we refer to as Langevin updating, or by using the more crude non-strict gradient descent procedure provided by the **Manhattan** [17] updating rule[2]

$$\Delta\omega = -\eta\cdot\mathrm{sgn}[\frac{\partial E}{\partial\omega}] \tag{10}$$

2.2 Second-Order Algorithms

Gradient descent assumes a flat metric where the learning rate η in eq. (7) is identical in all directions in ω-space. This is usually not the optimal learning rate and it is wise to modify it according to the

[1] throughout this paper quantities written in *sans-serif* denote matrices and quantities written in boldface denote vectors

[2] Note that this last equation refers to individual weights and not to the whole weight vector.

appropriate metric. Ideally one would like to use a second order method like the **Newton rule**, that optimizes the updating step along each direction according to

$$\Delta\boldsymbol{\omega} = -\mathsf{H}^{-1}\nabla E \tag{11}$$

where H is the Hessian matrix

$$\mathsf{H} = \frac{\partial^2 E}{\partial\omega_{ij}\omega_{i'j'}} = \nabla^2 E \tag{12}$$

Unfortunately, computing the full Hessian for a network is too CPU and memory consuming to be of practical use. Also, H is often singular or ill-conditioned [18], in which case the Newton method breaks down. One therefore has to resort to approximate methods.

Below, we discuss those approximate methods that are implemented in `JETNET 3.0` – an extensive review of second order methods for ANN is found in [19].

2.2.1 Heuristic Methods

One well-known method to approximate the curvature information is the Quickprop (**QP**) algorithm [9], where the basic idea is to estimate the weight changes by assuming a parabolic shape for the error surface. The weight changes are then modified by the use of heuristic rules to ensure downhill motion at all times. Furthermore, a small constant ε is added to the derivative $g'(x)$ of the activation function to escape flat spots on the error surface. In short, the updating for each weight reads

$$\Delta\omega_{t+1} = -\eta\Theta(\partial_\omega E_{t+1}\cdot\partial_\omega E_t)\partial_\omega E_{t+1} + \frac{\partial_\omega E_{t+1}}{\partial_\omega E_t - \partial_\omega E_{t+1}}\Delta\omega_t \tag{13}$$

where Θ is the Heaviside step function and $\partial_\omega E$ is the derivative of E with respect to the actual weight. This updating corresponds to a "switched" gradient descent with a parabolic estimate for the momentum term. To prevent the weights from growing too large, which indicates that QP is going wrong, a maximum scale is set on the weight update and it is recommended to use a weight decay term (see below). The algorithm is also restarted if the weights grow too large [20].

Another heuristic method, suggested by several authors [10, 21, 22], is the use of individual learning rates for each weight that are adjusted according to how "well" the actual weight is doing. Ideally, these individual learning rates adjust to the curvature of the error surface and reflect the inverse of the Hessian. In our view, the most promising of these schemes is Rprop [10]. Rprop combines the use of individual learning rates with the Manhattan updating rule, eq. (10), adjusting the learning step for each weight according to

$$\eta_{\omega,t+1} = \begin{cases} \gamma_+\eta_{\omega,t} & \text{if } \partial_\omega E_{t+1}\cdot\partial_\omega E_t > 0 \\ \gamma_-\eta_{\omega,t} & \text{if } \partial_\omega E_{t+1}\cdot\partial_\omega E_t < 0 \end{cases} \tag{14}$$

where $0 < \gamma_- < 1 < \gamma_+$.

2.2.2 Conjugate Gradients

A somewhat different technique to use the (approximately) correct metric, without direct computation of the Hessian, is the method of Conjugate Gradients (**CG**), where E is iteratively minimized within separate one-dimensional subspaces of ω-space (see e.g. ref. [23]). The updating hence reads

$$\Delta\boldsymbol{\omega}_t = \eta_t \mathbf{d}_t \tag{15}$$

where the step length η is chosen, by employing a line search, such that E is minimized along the direction $\mathbf{d}$. The Hessian metric is taken into account by making the minimization directions $\mathbf{d}$ conjugate to each other such that

$$\mathbf{d}_t^T \mathsf{H} \mathbf{d}_{t'} \propto \delta_{tt'} \tag{16}$$

By using the negative gradient of E for the initial direction $\mathbf{d}_1$ it is possible to get all the subsequent conjugate directions, without ever actually computing the Hessian, through

$$\mathbf{d}_{t+1} = -\nabla E_{t+1} + \beta_t \mathbf{d}_t \tag{17}$$

where β is chosen such that eq. (16) is fulfilled. This technique is exact if E is a quadratic form and if all the minimizations within the subspaces are exact. However, since this is never the case, several methods have been suggested for how to compute the subsequent search directions. In `JETNET 3.0` we have implemented

$$\beta_t = \begin{cases} \nabla E_{t+1} \cdot (\nabla E_{t+1} - \nabla E_t)/\mathbf{d}_t \cdot (\nabla E_t - \nabla E_{t+1}) & \textit{Hestenes-Stiefel} \\ \nabla E_{t+1} \cdot (\nabla E_{t+1} - \nabla E_t)/\nabla E_t \cdot \nabla E_t & \textit{Polak-Ribière} \\ \nabla E_{t+1} \cdot \nabla E_{t+1}/\nabla E_t \cdot \nabla E_t & \textit{Fletcher-Reeves} \end{cases} \tag{18}$$

plus a fourth one, *Shanno*, which is too complicated to include here. We refer the reader to [7] and [23] for a thorough discussion on these matters.

The line search part of CG minimization can be tricky and there exists a variant, Scaled Conjugate Gradient (**SCG**) [8], that avoids the line search by estimating the minimization step η_t through

$$\eta_t = \frac{-\mathbf{d}_t \cdot \nabla E_t}{\mathbf{d}_t \cdot (\mathbf{s}_t + \lambda_t \mathbf{d}_t)} \tag{19}$$

where λ is a fudge factor to make the denominator positive and $\mathbf{s}$ is a difference approximation of $\mathsf{H}\mathbf{d}$. This SCG method is usually faster than normal CG.

2.3 Interpretation of the Results

Even though the target values t in a classifying problem are binary, the output units in an MLP will not take values that are exactly 1 or 0. However, one can interpret these outputs in a very useful way; they correspond to Bayesian *a posteriori* probabilities [24] provided that:

1. The training is accurate.
2. The outputs are of 1-of-M-type (the task is coded such that only one output unit is "on" at a time).
3. A mean square, cross entropy or Kullback error function is used.
4. Training data points are selected with the correct *a priori* probabilities.

This very important result enables the network outputs to be further processed in a controlled way. In the case of function mapping the output error can be estimated with standard methods based on distances to cluster centers in the training data.

3 Guidelines and Rules of Thumb

In this section we deal with the issue of when to use ANN as compared to more conventional approaches, together with guidelines on how to configure an MLP to obtain optimal results. The recommendations are based on experience and references in the literature. A few of the techniques and ideas discussed here are not implemented in `JETNET 3.0` but we have nevertheless chosen to include them in our discussion due to their general interest.

3.1 Choosing the Model and its Parameters

3.1.1 ANN versus other methods

There are many different methods around for doing multivariate statistical analysis, function fitting or prediction tasks and ANN represents only a small subset of these. From a statistical modeling point of view, ANN models belong to the general class of *non-parametric* methods that do not make any assumption about the parametric form of the function they model. In this sense they are more powerful than *parametric* methods that try to fit reality into a specific parametric form. However, non-parametric methods like ANN contain more free parameters and hence require more training data than parametric ones in order to achieve good generalization performance [25].

Fortunately, for most HEP problems one has access to big data samples, making it possible to exploit the capabilities of non-parametric models like ANN. Tests of ANN versus standard methods on pattern recognition HEP problems are therefore in favour of ANN models [26, 27, 28, 29, 30, 31]. Also, unbiased comparisons of ANN and non-ANN methods on prediction tasks are in favour of ANN [32].

Inevitably, the choice of method depends on many problem dependent factors. Is the problem complex enough to call for a non-parametric method like ANN? Is data easily available? Does the application require real-time execution? Hence, it is impossible to give a general rule on what strategy to follow (see e.g. ref. [33] for a discussion of the subject). However, ANN methods have a number of features that make them particularly attractive:

- The output nodes (o_i) are analytic functions of the arguments x_i, if the activation function g is analytic. Derivatives with respect to the inputs can therefore be computed, which simplifies error estimation.
- As discussed above the output nodes approximate the Bayes *a posteriori* probabilities [24], which are useful to make final decisions that minimize the overall risk [34].
- Sigmoid units are not "orthogonal" and two hidden units may well perform identical tasks, which to some extent avoids overfitting. Also, this property can be very practical and even desirable if the goal is to produce a distributed system that is robust to weight losses. By a "smart" addition of noise in the training process, the network can be forced to choose a solution where the information is maximally distributed among the weights [35].
- An ANN network is not a linear function of all its weights. This implies a very beneficial

scaling property – for some functions and networks the learning curves are independent of the number of inputs [36].

Due to their generality, ANN methods also have some drawbacks, the most prominent one being long training times. Other statistical methods learn in general much quicker. For instance, models with "orthogonal" units (e.g. polynomial ones) may just need one inversion of a matrix in order to be trained.

It is sometimes argued that statistical non-parametric methods, like decision trees etc., are preferable to ANN models since the former are easier to interpret. We disagree with this view. With the aid of a self-organizing network it is quite easy to interpret an ANN model [4].

3.1.2 Choice of ANN Model

Classification
In classification problems, the task is to model the decision boundary between a set of distributions in the feature space [34]. This decision boundary is a surface of dimension $N-1$, where N is the number of relevant features/inputs.

The conventional ANN algorithms for classification problems are the MLP and Learning Vector Quantization (**LVQ**) [37]. The MLP needs $N_h \sim a^{N-1}$ hidden units to create the decision surface, whereas a nearest neighbour approach, like LVQ, needs $N_h \sim a^N$ units [38]. Hence, the MLP is in general more parsimonious in parameters than nearest neighbour approaches for pattern classification. In special cases, when the decision surface is highly disconnected, the LVQ approach may work better. We have found the MLP to work better than LVQ for all HEP problems encountered so far.

Approaches that combine the advantages of MLP and LVQ [39] seem to work better than just using an MLP (see below on modular architectures).

Some MLP-like approaches with skip-layer connections and iterative construction algorithms, like the Cascade Correlation algorithm [40], can construct very complex decision boundaries with a small number of hidden units. It is, however, uncertain how sensitive they are to overtraining.

Function fitting and prediction
In a function fitting problem, the task is to model a real-valued target function f from a number of (noisy) examples.

The straightforward ANN approach is to use the MLP with appropriate number of layers and units [41, 43]. Another is the "local map" where a partitioning algorithm, like k-means clustering [44], is used to divide the feature space into subregions. Each subregion is then associated with a function – a local map [42, 45, 46]. This method is similar in spirit to statistical methods like regression trees and splines [47, 48]. Both the MLP and the local map approaches work well and which method to choose depends on how local the problem is.

A third approach, which is often suggested for time-series prediction, is to use recurrent networks with feed-back connections. However, in our experience with time series the simple MLP produces as good solutions as recurrent networks, within much shorter training times, given that one is using the appropriate time lagged inputs [49].

3.1.3 Number of hidden units

There is a trade-off between *bias*, which is the networks ability to solve the problem, and *variance*, which is the risk of overfitting the data. The ultimate goal is to select the model that minimizes the generalization error, which is the sum of the bias and the variance. Hence, it is necessary to estimate the generalization error to select the appropriate number of hidden units. Experimentally, this can be done with Cross Validation (**CV**), Jack-knife, or Bootstrap methods [50, 51]. For instance, in v-fold Cross Validation the data set is divided into v disjoint subsets, of which $v - 1$ are used for training and one for testing. The training procedure is repeated, identically, until all subsets have been used for testing and the CV estimate of the generalization error is the average error over these v experiments

$$E_{gen} \approx \langle E_{test} \rangle_v \tag{20}$$

To save time one can instead of experimental methods use analytical estimates for the generalization error [52, 53, 54]. One approximate form for the (summed square) generalization error is [52],

$$E_{gen} \approx E_{train} \left(1 + 2 \frac{N_\omega}{N_p} \right) \tag{21}$$

where N_ω is the number of weights in the network and N_p is the number of patterns in the training set. This measure agrees well with the experimental CV measure above [52].

However, the above methods are all *a posteriori* and work only in "trial and error" experiments where the generalization performance of different architectures are compared after training. Needless to say it is desirable to have an *a priori* method that selects the optimal number of hidden units before training. For classification problems, the dimension of the feature space is a rough indicator. If the network is expected to separate a closed volume in N dimensions from its exterior, the minimum number of hidden units needed is $N + 1$. For an open volume the minimum number of hidden units is much smaller.

In function fitting problems, estimates similar to eq. (21) can be made for certain classes of functions and networks. In ref. [36] the following scaling relationship is given for the number of hidden nodes that minimize the generalization error

$$N_h \sim C_f \sqrt{N_p / (N \log N_p)} \tag{22}$$

provided that a one hidden layer MLP with linear output is used. However, C_f is the first absolute moment of the Fourier magnitude distribution of the function f, which is unknown! This uncertainty limits the use of eq. (22) to being only a rough estimate on the number of units.

Fortunately, it is not necessary to know the exact number of hidden units beforehand. It is possible to start out with more units than needed and remove superfluous units during or after training. We discuss below how this pruning can be done.

3.1.4 Number of hidden layers

In theory, an MLP with one hidden layer is sufficient to model any continuous function [55]. In practice, two hidden layers can be more efficient [41, 43, 49] but more difficult to train. In our experience, MLP networks with one hidden layer are sufficient for most classification tasks, whereas two hidden layers are preferable for function fitting problems. We emphasize though that many HEP classification problems seem to have simple discrimination surfaces, which would explain why one hidden layer often is enough. Networks with many hidden layers are not justified unless the decision surface is complicated. In fact, it is completely unneccesary to use an ANN at all if the decision surface is very simple, like a hyperplane or a hypersphere.

3.1.5 The activation function

The choice of activation function can change the behaviour of the ANN network considerably.

Hidden units
The standard choice is the *sigmoid* function, eq. (4), either in symmetric $[-1,1]$ or asymmetric $[0,1]$ form. The sigmoid function is global in the sense that it divides the feature space into two halves, one where the response is approaching 1 and another where it is approaching 0 (-1). Hence it is very efficient for making sweeping cuts in the feature space.

Other choices are the *Gaussian bar* [41], which replaces the sigmoid function with a Gaussian, and the *radial basis function* [42]. These are examples of local activation functions that can be useful if the effective dimension of the problem is lower than the actual number of variables, or if the problem is local.

Output units
For classification tasks, the standard choice is the sigmoid. The outputs can also be normalized, such that they sum to one, by using so-called Potts or softmax output

$$o_i = o(a_1, a_2, ..., a_n, T) = \frac{e^{a_i/T}}{\sum_l e^{a_l/T}} \tag{23}$$

where a_i is the summed signal arriving at output i. For function fitting problems the output should be chosen linear.

Of these, `JETNET 3.0` implements all possibilities except for the Gaussian bar and radial basis function.

It is sometimes suggested to use piecewise linear functions instead of the more complicated hyperbolic tangent for the sigmoid, in order to speed up the training procedure. We have not found any speedup whatsoever when the simulations are run on RISC workstations. It might however be relevant if the simulations are run on small personal computers.

3.1.6 Exploiting symmetries

Symmetries in the problem can and should be exploited to reduce the connectivity and complexity of the network. For translational symmetries one can use so-called "receptive fields", in which the input field is divided into subfields with shared weights [1]. Also, if it is known that the important feature only occupies a small part of the input field, then one can use "selective fields", which is essentially the same as "receptive fields" without the shared weights property. If possible, the most robust and time saving technique is to preprocess the data such that it is presented to the network in an invariant form [56].

If it is suspected, but unknown, that the problem has a symmetry, then it is possible to use "soft weight sharing" [57], which clusters the weight values by adding a complexity term to the usual error measure (see the section on pruning below).

3.1.7 Modular methods

The optimal model is not necessarily one single model. Instead, it may be profitable to divide the problem into smaller subtasks, like separating "location" from "form", and use different models for the subtasks. Such modular systems are often more efficient and easier to train than systems based upon a single architecture only. They are also easier to train. One example is presented in [39] where an MLP with a superficial LVQ network is shown to be more efficient than just the single MLP for classifying hadronic events. The superficial LVQ layer is able to resolve non-linearities that remain even after the final hidden layer in the MLP. Another example is the n-class classification tasks, where it may be wise to train n networks to recognize one class and then combine them into a larger network. This avoids the problem of interference, which occurs when the recognition of one class interferes with the recognition of another class due to the non-locality of the MLP division process.

3.2 Choosing the Learning Algorithm

One major difference between `JETNET 3.0` and older versions is the existence of several alternative learning algorithms. Though, after extensive explorations of these new learning algorithms, we find that BP learning, sometimes with noise added, is not only sufficient but often superior for most tasks. It is a very stable learning algorithm that reaches as low or lower errors than any alternative algorithm. In what follows we summarize our experiences with the different learning algorithms implemented in `JETNET 3.0`. Ref. [58] contains a review of other ANN packages that implement other learning algorithms.

3.2.1 Back-propagation

Back-propagation is the most widely used learning algorithm since it is very simple to implement and, most importantly, it often outperforms other methods in spite of its simplicity. We emphasize, though, that this statement refers to the "on-line" variant of BP, where the weights are updated after presentation of only a small subset of training data. On-line BP is much faster than batch mode BP.

For networks with more than one hidden layer it is beneficial to use the Langevin updating variant (eq. (9)), where noise is added to the BP equations [6]. This is because the Hessian matrix easily becomes ill-conditioned with a flat subspace where the random search in Langevin updating is very efficient as compared to other alternatives.

3.2.2 Quickprop

In using "Quickprop" [9] we frequently encounter problems with getting a stable performance. It works well on parity and decoder problems but has difficulties with HEP problems – it often reaches very large weights and gets stuck[3].

3.2.3 Rprop

In a recent benchmark test on a medical data set [59] "Rprop" was reported to outperform all other learning algorithms, both in speed and quality. However, its superb performance in this test is related to its use of individual learning rates. Normalizing the data in the way described below makes BP perform as well, if not better [6].

3.2.4 Conjugate Gradients

In ref. [7] the CG method outperformed BP on the parity problem. However, our experience with CG on HEP problems is the opposite; it is often unable to find the true global minimum. The same conclusion was reached in an extensive benchmark test of different ANN learning algorithms [59]. Consequently, we see no reason to recommend using CG, although it learns toy problems very fast.

The strength of CG is in the rare cases when the path to the minimum follows a few long narrow valleys. However, it breaks down whenever the error surface is more or less flat, since the CG line search will attempt to find a minimum along a flat direction. As previously stated, flat surfaces often occur for networks with many hidden layers.

If one insists on using CG in such cases it is profitable to initialize the CG learning by a couple of BP sweeps in order to get out of the flat region. The use of a coarse line search is recommended. It is a waste of resources to search for a very exact minimum position along each conjugate direction. Also, the SCG algorithm is usually faster than standard CG since it avoids the line search.

[3] We have not performed extensive benchmarks of QP and its failure could therefore be related to our insufficient experience.

3.3 Preprocessing the data

Preprocessing the data is important for many reasons.

- To prevent overfitting by reducing the number of inputs and hence the number of weights.
- To avoid "stiffness" in the learning process by rescaling the data.
- To simplify the problem by precomputing useful signatures from the data [60].

The input space dimension can be reduced by performing a Principal Component Analysis (**PCA**) and select the n first principal axes as the basis in feature space. The PCA does not however guarantee that the chosen inputs are relevant for the output, it only selects the inputs with the largest variance. Also, one should keep in mind that PCA assumes linear dependencies and one might hence loose nonlinear information by employing it.

For function mapping problems the most powerful method, to our knowledge, for extracting functional dependencies between input and output is the so-called δ-test [61]. This test only assumes that the function is continuous and uses conditional probabilities to select the significant input variables.

Normalization of the input is done to prevent "stiffness", i.e. when weights need to be updated with very different learning rates. Two simple normalization options are; either scale the inputs to the range $[0, 1]$, or translate them to their mean values and rescale to unit variance. The former method is useful if the data is more or less evenly distributed over a limited range, whereas the latter is useful when the data contains outliers. In some cases, such normalizations reduce the learning time for the network by an order of magnitude.

A method suggested in [26] is to let the network handle the normalization by adding an extra layer of units. This is useful if the data is not available beforehand to compute the relevant scales.

3.4 When to Stop Training

Before attempting an ANN model on a problem, one should if possible estimate the outcome. For a classification problem, the optimal classification performance is the Bayes limit or Bayes error [34], which equals the Bayes risk with zero-one loss function. This upper classification limit can be estimated by the use of simpler classifiers, like the k-nearest-neighbours or Parzen windows [34, 50, 62][4]. With such an estimate at hand, it is much easier to evaluate the quality of the ANN model.

To determine the termination point for the training it is customary to use a *validation* data set. This validation set is not used directly in the training, i.e. not presented to the network, but used indirectly to monitor the performance on unknown data. A deteriorating performance on the validation set signals that the ANN is overlearning the training data and that training should

[4] Ref. [62] provides F77 code for doing this estimation

be stopped. When the training is stopped, a test set can be used to estimate the generalization performance. It is however imperative that the validation data are not used in the test set, since it is indirectly used in the training to choose a stopping point.

In cases where data is scarce and the use of a validation set is too costly, one can instead use a threshold value on the training error. For instance, when computing CV estimates one can train each network until it reaches a prespecified training error, which has been determined by a couple of trial runs.

3.5 Regularization and Pruning

As mentioned in Sect. 2 it is important to keep the number of weights minimal in order to avoid overfitting. With respect to weights connecting to sensor nodes this can be done by preprocessing. A more general approach valid for all weights is to add a complexity term to the fitness error (eq. (5)). The simplest such **pruning** procedure is weight decay, which reads

$$E \to E + \frac{\lambda}{2} \sum_{i,j} \omega_{ij}^2 \tag{24}$$

The sum extends over all weights in the network and λ is a Lagrange multiplier controlling the relative cost for large weights. Eq. (24) constrains the weights to a prior Gaussian probability distribution $P(\omega) \propto \exp[-\lambda\omega^2/2]$ with $-\ln P$ as the complexity cost. A slightly more sophisticated pruning option is [63]

$$E \to E + \lambda \sum_{ij} \frac{\omega_{ij}^2/\omega_0^2}{1 + \omega_{ij}^2/\omega_0^2} \tag{25}$$

which has zero cost for small weights and λ cost for large weights. Similar to weight decay, this corresponds to a prior weight distribution $P(\omega) \propto \exp[-(1 + \omega_0^2/\omega^2)^{-1})]$. Both the weight decay and the pruning method above are options in `JETNET 3.0`.

Of course, it is by no means necessary that an optimal network solution contains a set of small weights and only a few large weights. It may well be that the optimal weight distribution is multimodal, as is the case for problems with symmetries and shared weights. In ref. [57] a procedure is suggested, where the weight distribution is assumed to be a multimodal mixture of Gaussians whose means and widths are adjusted during learning. This method, which is valuable if the problem has unknown symmetries, is not implemented in `JETNET 3.0`.

There also exist a posteriori methods for pruning trained networks by measuring the relevance of the units [64] or by computing the Hessian matrix to remove superfluous weights [65, 66]. One extremely simple method that works surprisingly well is a posteriori pruning by visual inspection: Remove all weights with a magnitude less than some threshold, provided that the inputs have been normalized.

The network must be retrained after a posteriori pruning, in order to find the global solution given the new constraints.

4 Practical Implementation Issues

This section is intended as a guide to the program components section and addresses the practical aspects of using JETNET 3.0. We include information on all new features of JETNET 3.0 and on those in the earlier version that generated questions from the users. All subroutines, parameters and switches mentioned here are described in the program components section.

In some rare cases we mention techniques that are not part of the JETNET 3.0 package. This is because we consider them important, although we have not had the opportunity to implement them, and because the JETNET 3.0 user should be aware of their existence.

4.1 Initialization

JETNET 3.0 is initialized by calling the subroutine JNINIT. It allows for switching between different learning rates at convenience during execution, but each learning algorithm uses specific parameters that need to be initialized. The default values of these parameters give good results in most cases.

The ANN architecture (number of hidden layers, nodes, etc.) is designed through the switches MSTJN(1) and MSTJN(10-19). The distribution of the initial weights is set by the parameter PARJN(4). Naturally, these switches and parameters must be set prior to calling JNINIT.

4.1.1 Initial weight values

It is of utmost importance to ensure that the units are "active learners" and not saturated to their extreme values. The derivative of the activation function (eq. 4)) is zero for saturated units and thus inhibits learning. This can be avoided by proper weight initialization. If the input is normalized to unit size, one simply scales the weights in proportion to the number of units feeding to a unit (the "fan-in"). A suitable normalization for this is

$$\texttt{PARJN(4)} = \frac{0.1}{\max[\text{“fan-in”}]} \tag{26}$$

Another method, suggested in [59], is to set the width PARJN(4) to any value and then process the training data through the network once and adjust the thresholds to make the average argument of each unit equal to zero. Other suggestions are found in refs. [67, 68]. None of these methods are automated in JETNET 3.0.

4.1.2 Backpropagation

The BP algorithm, eq. (7), is selected by setting MSTJN(5) = 0 (default). Its main parameters are the learning rate PARJN(1) (η in eq. (7)), the momentum PARJN(2) (α in eq. (8)), and the number of patterns per update MSTJN(2). We strongly advocate the use of an on-line updating procedure where MSTJN(2) is small. Routinely we use ten patterns per update for most applications

– occasionally an order of magnitude more. The learning rate is the parameter that requires most attention. Typical initial values are in the range $[0.1, 1]$ and it is usually profitable to scale the learning rate in inverse proportion to the fan-in of the units so that different learning rates are used for different weight layers. The momentum should be in the range [0,1]. For HEP problems momentum values above 0.5 are seldom required. For parity problems and such, a momentum value close to unity is needed.

In contrast to earlier versions, `JETNET 3.0` uses a normalized error to make the gradient, and hence the learning parameters are independent of the number of patterns used per update.

4.1.3 Manhattan

Manhattan learning (eq. (10)) is selected by setting `MSTJN(5)` $= 1$. It uses basically the same learning parameters as BP. However, since the weight update is not reduced by the derivative of the sigmoid, the learning rate must be a few orders of magnitude smaller.

4.1.4 Langevin

Langevin learning (`MSTJN(5)` $= 2$) is identical to BP except for an additional Gaussian noise term (eq. (9)). In our view, LV is the most powerful of all the algorithms for networks with many hidden layers, even though it requires somewhat more CPU time [6, 49]. Except for the noise level (`PARJN(6)`), to which it is not very sensitive provided it is less or equal to 0.1, it uses the same parameters as BP.

4.1.5 Quickprop

Quickprop (`MSTJN(5)`$=3$), which estimates the updating step through eq. (13), has two learning parameters and two control parameters. These are the "learning rate" `PARJN(1)` (η), the sigmoid-prime addition `PARJN(23)` (ε), the maximum growth factor `PARJN(21)` and the maximum weight magnitude `PARJN(22)`. The latter is quite unimportant. Default values recommended in [9] for the other three parameters are; `PARJN(1)` of order unity, `PARJN(23)`$= 0.1$ and `PARJN(21)` $= 1.75$. We have not done extensive tests to to determine the problem dependence of these parameters.

It is recommended to run QP using a small weight decay (`PARJN(5)`).

4.1.6 Conjugate Gradient

There are eight different variants of CG in `JETNET 3.0`. Both standard CG (`MSTJN(5)` $= \{4, 5, 6, 7\}$) (eq. (15)) and Scaled CG (`MSTJN(5)` $= \{10, 11, 12, 13\}$) (eq. (19)) come with the four different options in eq. (18) for computing the next search direction. The CG parameters, which control the line search etc., are described in a separate section below. The SCG parameters are `PARJN(28)` and `PARJN(29)`. They correspond to the step used in computing the difference approximation s in eq. (19) and to the initial value used for λ, respectively. The default values for these parameters usually work fine and the user should not need to set any parameters when using SCG, although the algorithm is sometimes speeded up by increasing `PARJN(28)`.

The SCG runs best in combination with the Polak-Ribière formula for computing the next search direction (`MSTJN(5)`=10). The CG algorithm, with line search, runs best with the Polak-Ribière (`MSTJN(5)`=4) or Hestenes-Stiefel (`MSTJN(5)`=5) formulas. However, the Shanno formula is designed to always guarantee a descent direction and is hence more robust (but slower).

4.1.7 Rprop

Rprop (`MSTJN(5)` = 15) uses individual learning rates that are dynamically tuned during training according to eq. (14). It has two learning parameters and two control parameters besides the learning rates (in vector `ETAV`); the scale factors γ_+ and γ_- (`PARJN(28-29)`) and the maximum allowed scale-up and scale-down factors (`PARJN(30-31)`). According to [59] the final result is not very sensitive to the choice of the scale factors. Hence the only concern are the initial learning rates, which are set as in the BP case. `JETNET` uses the value stored in `PARJN(1)` or `ETAL` to initialize the learning rates.

4.1.8 Batch training

Care must be taken when using batch type algorithms, like QP, CG, SCG and Rprop. These algorithms depend heavily on changes in the error value between consecutive positions. Consequently, it is important that the same patterns are used for consecutive updates unless very large data samples are used so that fluctuations are negligible. This is done in `JETNET 3.0` by setting `MSTJN(2)` equal to the total number of training exemplars and `MSTJN(9)` equal to one. This ensures that `JETNET 3.0` will be evaluating the correct error function all the time.

4.2 Training the Network

After initialization, the network is trained by presenting training patterns and invoking the subroutine `JNTRAL`. The sample program illustrates how this is done.

4.3 Dynamic Learning Parameters

The optimal learning rate varies during learning. For BP and LV one should start out with a large learning rate and decrease it as the network converges towards the solution. Initial weight adjustments in general need to be large, since the probability of being close to the minimum is small, whereas final adjustments should be small in order for the network to settle properly. For BP and LV we use a so-called "bold driver" method [69] where the learning rate is increased if the error is decreasing, and decreased if the error increases:

$$\eta_{t+1} = \begin{cases} \eta_t \cdot \gamma & \text{if } E_{t+1} \geq E_t \\ \eta_t \cdot (1 + \frac{1-\gamma}{10}) & \text{otherwise} \end{cases} \tag{27}$$

The scale factor γ, which is set by the parameter `PARJN(11)`, is close to but less than one. For MH learning we recommend an exponential decrease of the learning rate, realized by choosing a negative value for `PARJN(11)`. Examples of other more advanced methods for regulating the learning rate are found in refs. [70, 71, 72].

The noise level used in LV updating should also decrease with time, preferably faster than the learning rate. We use an exponential decay governed by the scale parameter `PARJN(20)`. This procedure is sufficient to significantly improve the learning for networks with many hidden layers [6]. From the perspective of simulated annealing and global optimization, an exponentially decreasing noise level can also be justified given that the simulation time is finite [74].

Also implemented in `JETNET 3.0` are options of having the momentum and the temperature change each epoch. However, no improvements have been observed using these options.

4.4 Conjugate Gradient Learning

Several things must be kept in mind when using the CG option in `JETNET 3.0` since it is a batch-type learning algorithm that depends strongly on the uniqueness of the error function. Most importantly, an identical set of training patterns must be used for each evaluation of the error, otherwise the line search gets confused.

4.4.1 The line search

Although inspired by algorithms in [23, 73], the line search algorithm implemented in `JETNET 3.0` is somewhat unorthodox. In contrast to traditional line searches that start out in one point, evaluate the function at a different point, and then move back to the original point, the line search in `JETNET 3.0` does not move back to its original position after evaluating the error function. This results in somewhat confusing behaviour since `JETNET 3.0` outputs the error value at its current position in weight space. Hence there is no cause for alarm if the error value fluctuates during learning. It is the minimum error achieved during the learning that is important. However, this means that `JETNET 3.0` must be told when the user stops training so that it can move to the position with the minimum error so far. This is done by setting `MSTJN(5)` to 8 or 14, depending on whether CG or SCG is used, and continue training until the value of `MSTJN(5)` changes to 9, which signals that `JETNET 3.0` has moved to the best configuration so far and stopped.

In coarse outline, the line search works as follows: First it computes the error at the initial position and the gradient along the search direction. It then computes the error at two subsequent positions along the search direction, with the first step equal to the learning rate `PARJN(1)` and the second step computed from a parabolic fit using the gradient information. From these three error values it makes a new parabolic fit and moves to the predicted minimum position. Such parabolic steps are then repeated until the line search finds a satisfactory minimum or until it has used up the prespecified number of trial steps. In the latter case the line search is restarted with a new value for PARJN(1). If the line search does not find a satisfactory minimum even within the prespecified number of restarts, the whole CG process is restarted from the current minimum position, using a rescaled PARJN(1).

The most important control parameters for the line search are MSTJN(35), MSTJN(36), PARJN(24), and PARJN(25). The first two control the number of iterations and number of restarts that are allowed in searching for the minimum, and the latter two set the convergence criteria. The default value for both MSTJN(35) and MSTJN(36) is 10, which works fine for toy problems but need to be increased for real problems. The convergence criteria are

$$\text{if } E \leq E_0 + \varepsilon r \partial_d E_0 \tag{28}$$

or

$$\text{if } |r - r_{pred}| \leq \delta \tag{29}$$

where ε is the first control parameter PARJN(24), E_0 is the error at the initial position (where the line search was started), $\partial_d E_0$ is the initial gradient along the line search direction $\mathbf{d}$, r is the current distance from the initial position, r_{pred} is the predicted position of the minimum, and δ is the second control parameter (PARJN(25)). These convergence criteria are only checked for if the minimum has been bracketed.

4.5 Output

The main output from JETNET 3.0 is the training error. In contrast to previous versions, the error used in JETNET 3.0 is scaled with the size of the training data set according to eq. (5). Furthermore, while older versions always produced a summed square error, JETNET 3.0 gives the appropriate error (according to the value of MSTJN(4)), which is useful if one wants to use the pruning option.

4.6 Pruning

JETNET 3.0 implements the weight decay and the pruning method of eqs. (24) and (25). The Lagrange multipliers correspond to parameters PARJN(5) and PARJN(14) respectively. Hessian-based pruning can be done by computing the Hessian matrix, its eigenvalues and eigenvectors: Small weights that lie inside a flat subspace can be omitted.

Following [63], we tune λ in eq. (25) according to

- If ($E_t < D$ or $E_t < E_{t-1}$) $\Rightarrow$ increment $\lambda = \lambda + \Delta\lambda$.
- If ($E_t \geq E_{t-1}$ and $E_t < A_t$ and $E_t \geq D$) $\Rightarrow$ decrease $\lambda = \lambda - \Delta\lambda$.
- If ($E_t \geq E_{t-1}$ and $E_t \geq A_t$ and $E_t \geq D$) $\Rightarrow$ rescale $\lambda = c\lambda$, where $c < 1$.

where E_t is the current training error and the other quantities are defined as

A : The weighted average error: $A_t = \gamma A_{t-1} + (1-\gamma)E_t$.

D : The desired error, which acts as a threshold for the procedure. Solutions with error above D are not pruned unless the training error is decreasing. In ref. [63] it is advised for hard problems that D is set to random performance, which in practice means that pruning is always on.

Altough it is quite tricky to get it to work properly, we have used this procedure successfully on both toy and real-world problems. The recommended value for ω_0 in [63] is of order unity if the activation functions for the units are of order unity. This agrees with our experience, with the modification that ω_0 should follow $\omega \sim 1/$"fan-in". However, our tests were performed on problems where the number of inputs ranged between two and ten, whereas the largest networks in [63] have up to forty inputs. Also, on toy problems we find that the parameter $\Delta\lambda$ can be increased considerable (orders of magnitude) above the suggested default value of $1 \cdot 10^{-6}$.

In `JETNET 3.0` these pruning parameters correspond to; `PARJN(14)` for λ, `PARJN(15)` for $\Delta\lambda$, `PARJN(16)` for γ, `PARJN(17)` for c, `PARJN(18)` for ω_0, and `PARJN(19)` for the desired error D. Of these, `PARJN(15)`, `PARJN(18)` and `PARJN(19)` are crucial.

4.7 Computing the Hessian

A novelty in `JETNET 3.0` is the possibility to compute the Hessian matrix for an MLP network by invoking the subroutine `JNHESS`. The computation is done much in the same way as the training; training patterns are iteratively placed in the vectors `OIN` and `OUT` before `JNHESS` is called. When one full epoch, the size of which is controlled by `MSTJN(2)` and `MSTJN(9)`, has been presented, it normalizes and symmetrizes the Hessian and places it in the internal common block `/JNINT5/`.

If the Hessian has been symmetrized, the eigenvectors and eigenvalues of the Hessian can be computed by invoking `JNHEIG` (single precision). Eigenvalues are then placed in the vector `OUT` and the eigenvectors replace the columns of the Hessian matrix. The Hessian can be printed out by invoking the subroutine `JNSTAT`. However, anticipating possible questions, the terms of the Hessian matrix are ordered in `JETNET 3.0` according to (cf. fig. 1 and eq. (3))

$$\begin{pmatrix} \theta_i\theta_{i'} & \theta_i\omega_{i'j'} & \theta_i\theta_{j'} & \theta_i\omega_{j'k'} \\ \cdots & \omega_{ij}\omega_{i'j'} & \omega_{ij}\theta_{j'} & \omega_{ij}\omega_{j'k'} \\ \cdots & \cdots & \theta_j\theta_{j'} & \theta_j\omega_{j'k'} \\ \cdots & \cdots & \cdots & \omega_{jk}\omega_{j'k'} \end{pmatrix} \tag{30}$$

with obvious interpretation and extension to more layers.

`JNHESS` assumes the summed square error in eq. (5).

4.8 Receptive Fields and Shared Weights

`JETNET 3.0` offers the possibility to use shared weights for exploiting translational symmetries. An example is when the input units consist of a matrix of cells, e.g. $E_\perp$ cells in a calorimeter, where it is known *a priori* that identical features can occur anywhere in the matrix with translational symmetry. It is then profitable to configure the network so that the hidden (feature) nodes cover several overlapping smaller portions (large enough to cover the size of a subfeature) of the input matrix. Weights connecting to corresponding parts of the different receptive fields are then shared, i.e. assumed to be identical.

Such configurations can be achieved in `JETNET 3.0` using the switches `MSTJN(23-27)`. The geometry of the input matrix ($N_x \times N_y$) is specified with `MSTJN(23)` and `MSTJN(24)`. Periodic boundary conditions are assumed if these values are negative. The geometry of the receptive fields ($n_x \times n_y$) is specified with `MSTJN(25)` and `MSTJN(26)`, where $n_i \leq N_i$. The number of hidden units used for each receptive field is specified by `MSTJN(27)`. At initialization, an array of receptive fields is generated with maximum overlap, i.e. the fields are only shifted one (x or y) unit at a time, and new hidden units are generated if the specified number of hidden units is less than necessary. Any remaining hidden units are assumed to have full connectivity from the inputs.

However, it is faster to train small network modules and later combining them into a larger network. The above solution is inefficient for large input matrices, since all weights are nevertheless updated.

4.9 The Saturation Measure

The saturation s is defined as

$$s = \begin{cases} \sum_j (1 - 2 \cdot h_j^2) & \text{if } g(x) \in [0,1] \\ \sum_j h_j^2 & \text{if } g(x) \in [-1,1] \end{cases} \tag{31}$$

and measures the resolution of the units. An s-value close to unity signals that the unit has "made up its mind" whereas an s value close to zero means that it is still learning. The saturation, which is monitored by a nonzero value for `MSTJN(22)`, is consequently a measure of to what extent the network is learning or not. This is practical when proper learning parameter values are being tried out.

4.10 Limited Precision - Hardware Implementations

Since the final goal of many ANN applications in HEP is to design hardware to use for triggering etc. `JETNET 3.0` has an option of training a network with limited precision. The switches `MSTJN(28-30)` set the bit precision of the different components of the network.

5 Program Components

`JETNET 3.0` is a F77 subroutine package and contains a number of subroutines that are called from a main program written by the user. `JETNET 3.0` is divided in two parts, one for feed-forward back-propagation networks and another for self-organizing maps. The subroutines associated with the feed-forward net all begin with the letters `JN`, as in JetNet, whereas the self-organizing map subroutines all start with JM, as in JetMap. We will not discuss any of the JetMap subroutines and components here since they are basically unchanged from JETNET 2.0 [1]. JETNET 3.0 is backwards compatible with earlier versions.

5.1 Feed-forward Network (JN)

5.1.1 Main subroutines

The main routines are JNINIT, JNTRAL, JNTEST, JNDUMP and JNREAD (JNROLD). These are usually the only routines the user has to invoke. The network is initialized by setting switches in /JNDAT1/ and calling JNINIT. The net is trained by putting an input pattern in the vector OIN, the desired output values in the vector OUT and calling JNTRAL. Subsequent testing of the net is done by putting a test pattern in OIN, calling JNTEST and comparing the produced output values, stored in OUT, with the target values.

- SUBROUTINE JNINIT
  Initializes the network according to the setting of the switches in /JNDAT1/ and /JNDAT2/. Weights are given initial random values according to PARJN(4) or WIDL(I).

- SUBROUTINE JNTRAL
  Takes the pattern stored in the vector OIN and calls JNFEED to produce an output pattern, which is compared to the target pattern in the vector OUT. The resulting error is used to determine the change of the weights. The produced output pattern is stored in the vector OUT. Updating of the weights is performed every MSTJN(2) call to JNTRAL. The training algorithm is specified by the switch MSTJN(5).

- SUBROUTINE JNTEST
  Feeds the input signal in the vector OIN through the net and places the produced output in the vector OUT.

- SUBROUTINE JNDUMP(NF)
  Writes all relevant information of a network to the file ABS(NF). If NF is negative the output is unformatted, otherwise it is formatted. The user must make sure that the corresponding file is opened with write access accordingly.

- SUBROUTINE JNREAD(NF)
  Reads the information from the file ABS(NF) produced by JNDUMP and initializes the network described there. All switches and parameters in common blocks /JNDAT1/ and /JNDAT2/ set prior to a call to JNREAD are lost. The comments about the file number for JNDUMP apply also here.

- SUBROUTINE JNROLD(NF)
  Same as JNREAD(NF) except that it reads files produced by the older versions JETNET 1.0 and JETNET 1.1.

5.1.2 Other user invoked subroutines

Besides the main subroutines there are other subroutines that the user can use to define input windows, compute statistics and compute the Hessian matrix. These are JNSEFI, JNSTAT, JNHESS and JNHEIG.

Furthermore, the subroutine JNTDEC is a "test-deck" used to test the program on different platforms.

- SUBROUTINE JNSEFI(ILA,I1,I2,J1,J2,NO)
  Switches off (NO ≤ 0) or on (NO > 0) the updating of weights between nodes I1 to I2 in layer ILA and nodes J1 to J2 in layer ILA-1. The input layer has number 0. If NO=0 the weights are set to zero and if NO=1 the weights are reinitialized. This choice of enabling/disabling can be used for selective input fields or for training only portions of the network. Selective input fields means that a hidden node only sees a portion of the input pattern, which is not identical to using receptive fields.

- SUBROUTINE JNSTAT(IS)
  Writes out information about the network on file number MSTJN(6).
  IS=1 gives a header and number of nodes in each layer (done automatically when the network is initialized).
  IS=2 gives the switches and parameters used in common block /JNDAT1/ and /JNDAT2/.
  IS=3 writes out an approximate time factor and the effective number of weights in the net.
  IS=4 writes out the Hessian matrix.
  IS=5 writes out the diagonal elements of the Hessian matrix.

- SUBROUTINE JNHESS
  Computes the upper diagonal of the Hessian, assuming a summed square error, for the training pattern currently stored in OIN and OUT. After MSTJN(2)*MSTJN(9) calls, the Hessian is normalized and the upper diagonal copied onto the lower diagonal.

- SUBROUTINE JNHEIG(IGRAD)
  Diagonalizes the Hessian matrix and computes its eigenvalues. The eigenvalues of the Hessian are placed in the vector OUT. If IGRAD $\neq 0$ then the eigenvectors are computed and placed in the columns of the Hessian matrix (stored in the internal common block /JNINT5/). JNHEIG stops and writes out an error message if the Hessian is asymmetric, i.e. if JNHESS has not been presented with the full training set (no more and no less) before JNHEIG is called.

- SUBROUTINE JNTDEC(METHOD)
  A "test deck" that automatically tests JETNET 3.0. To check whether JETNET 3.0 is properly installed on your computer, just invoke this subroutine. It trains a feed-forward network to separate two overlapping Gaussian distributions. Which method to use is set by the switch METHOD.

5.1.3 Internal subroutines

Subroutines are presented in the order they are used when training the network.

- SUBROUTINE JNSEPA
  Sets the internal parameters and switches in common blocks /JNINT2/ and /JNINT3/. If receptive fields are used, the defined geometries are checked for inconcistencies.

- SUBROUTINE JNHEAD
  Writes a header on file number MSTJN(6).

- SUBROUTINE JNFEED
  Feeds the values in OIN through the network and calculates the values of the output nodes, without writing to OUT.

- SUBROUTINE JNDELT
  Calculates the error from the output nodes and current values in OUT.

- SUBROUTINE JNSATM
  Calculates the saturation measure s in eq. (31) for each layer. JNSATM is only called if MSTJN(22) $\neq$ 0.

- SUBROUTINE JNCHOP(ICHP)
  Switches on (ICHP>0) or off (ICHP<0) fixed precision weights, thresholds and sigmoid functions. If ICHP $\geq$ 0 the current values of the weights and thresholds are chopped to the fixed precision. The bit precision is set by switches MSTJN(28-30).

- SUBROUTINE JNERR(IERR)
  If something goes wrong or if any inconcistensies are encountered during execution, JNERR is called and writes out an error message and stops the execution.

- SUBROUTINE JNCOGR
  Controls the Conjugate Gradient training. It is called from JNTRAL if CG learning has been selected. JNCOGR calls the subroutines JNCGBE and JNLINS.

- SUBROUTINE JNCGBE(BETAK,IOP)
  If IOP $=1$, JNCGBE computes the CG momentum term in eq. (17) and returns it in BETAK. If IOP $=0$, JNCGBE sets BETAK $=0$ and computes the scalar product of the current and previous gradients.

- SUBROUTINE JNLINS
  Performs the line search with an algorithm that is a mixture of golden section search and quadratic interpolation. All parameters used in the line search are stored in the common block /JNINT4/.

- SUBROUTINE JNSCGR
  Controls the Scaled Conjugate Gradient training. It calls the subroutine JNCGBE.

- SUBROUTINE JNTRED and SUBROUTINE JNTQLI
  These routines are taken directly from [23] and are used to diagonalize the Hessian matrix and compute its eigenvectors and eigenvalues.

5.1.4 Internal functions

- INTEGER FUNCTION JNINDX(IL,I,J)
  Gives the node vector index of node I in layer IL if J=0, otherwise it gives the weight vector

index of the weight between node I in layer IL and node J in layer (IL-1). (See common block /JNINT1/ for details on the node and weight vectors.)

- REAL FUNCTION GJN(X,N)
  Returns the activation function $g(x)$ of type N with argument X. It also computes the derivatives $g'(x)$ and $g''(x)$ and stores them in the common block /JNSIGM/. The activation function type is determined by switches MSTJN(3) or IGFN(I).

- REAL FUNCTION GAUSJN(IDUM)
  Generates a Gaussian distributed random number with standard deviation 1.0 and mean 0.0 for use in Langevin updating.

- REAL FUNCTION ERRJN(IDUM)
  Returns the error of the network, determined by the switch MSTJN(4).

- REAL FUNCTION RJN(IDUM)
  A random number generator that returns random numbers in the open interval $]0,1[$. The routine is taken, basically unchanged, from the Lund program JETSET but the code owes very much to [75].

5.2 Common blocks

5.2.1 Interface common blocks

The user interface common blocks are /JNDAT1/ and /JNDAT2/. /JNDAT1/ is the main common block while /JNDAT2/ is intended for the "advanced" user.

- COMMON /JNDAT1/ MSTJN(40),PARJN(40),MSTJM(20),PARJM(20),OIN(1000),OUT(1000), MXNDJM

MSTJN is a vector of switches used to define the feed-forward network used:

MSTJN(1)	(D=3)	number of layers in the net
MSTJN(2)	(D=10)	number of patterns per update in JNTRAL
MSTJN(3)	(D=1)	overall activation function used in the net $1 \rightarrow g(x) = \frac{1}{1+\exp(-2x)}$ $2 \rightarrow g(x) = \tanh(x)$ $3 \rightarrow g(x) = \exp(x)$ (only used internally for Potts-nodes) $4 \rightarrow g(x) = x$ $5 \rightarrow g(x) = \frac{1}{1+\exp(-2x)}$ (only used internally for entropy error)
MSTJN(4)	(D=0)	error measure -1 $\rightarrow$ log-squared error: $E = -\sum \ln(1-(o-t)^2)$ 0 $\rightarrow$ summed square error: $E = \frac{1}{2}\sum(o-t)^2$ 1 $\rightarrow$ entropy error: $E = \sum\{(1-t)\ln(1-o) - t\ln(o)\}$ $\geq$2 $\rightarrow$ Kullback error with Potts nodes of dimension MSTJN(4) $E = \sum t \ln(t/o)$
MSTJN(5)	(D=0)	updating procedure 0 $\rightarrow$ normal updating

		1 → Manhattan updating
		2 → Langevin updating
		3 → Quickprop updating
		4 → Conj. Grad. updating – Polak-Ribiere
		5 → Conj. Grad. updating – Hestenes-Stiefel
		6 → Conj. Grad. updating – Fletcher-Reeves
		7 → Conj. Grad. updating – Shanno
		8 → terminate Conj. Grad. updating
		9 → no updating
		10 → Scaled Conj. Grad. updating – Polak-Ribiere
		11 → Scaled Conj. Grad. updating – Hestenes-Stiefel
		12 → Scaled Conj. Grad. updating – Fletcher-Reeves
		13 → Scaled Conj. Grad. updating – Shanno
		14 → terminate Scaled Conj. Grad. updating
		15 → Rprop updating
`MSTJN(6)`	(D=6)	file number for output statistics
`MSTJN(7)`	(R)	number of calls to `JNTRAL`
`MSTJN(8)`	(I)	initialization done
`MSTJN(9)`	(D=100)	number of updates per epoch
`MSTJN(10+I)`		number of nodes in layer I (I=0 → input layer)
`MSTJN(10)`	(D=16)	
`MSTJN(11)`	(D=8)	
`MSTJN(12)`	(D=1)	
`MSTJN(13-20)`	(D=0)	
`MSTJN(21)`	(D=0)	pruning ($> 0 \rightarrow$ on)
`MSTJN(22)`	(D=0)	saturation measure s ($\neq 0 \rightarrow$ on)
`MSTJN(23,24)`	(D=0)	(x,y)-geometry of input field when using receptive fields
		$< 0 \rightarrow$ periodic boundary conditions
		see `COMMON /JNINT3/` for further explanations
`MSTJN(25,26)`	(D=0)	(x,y)-geometry of receptive fields
`MSTJN(27)`	(D=1)	number of hidden nodes per receptive field
`MSTJN(28-30)`	(D=0)	bit-precision (0 → machine precision) for sigmoid
		functions (28), thresholds (29) and weights (30)
`MSTJN(31)`	(D=1)	procedure for handling warnings
		0 → warnings are ignored
		1 → execution is terminated after `MSTJN(32)` warnings
		in any case, only `MSTJN(32)` warning messages are issued
`MSTJN(32)`	(D=10)	maximum number of warning messages to be issued
		(see description above)
`MSTJN(33)`	(I)	code for latest warning issued by the program
`MSTJN(34)`	(I)	number of warnings issued by the program so far
`MSTJN(35)`	(D=10)	maximum number of iterations allowed in line search
`MSTJN(36)`	(D=10)	maximum number of allowed restarts for the line search
`MSTJN(37)`	(I)	current status of the line search
		0 → minimum found
		1 → searching for minimum
`MSTJN(38)`	(I)	number of restarts in QP/CG/SCG
`MSTJN(39)`	(I)	number of calls to `JNHESS`
`MSTJN(40)`		not used.

Switches 2, 5, 6, 21, 22, 28, 29, 30, 31, 32, 35 and 36 can be changed at any time by the user.

PARJN is a vector of parameters determining the performance of the feed-forward net:

PARJN(1)	(D=0.001)	learning rate η
PARJN(2)	(D=0.5)	momentum parameter α
PARJN(3)	(D=1.0)	overall inverse network temperature $\beta = 1/T$
PARJN(4)	(D=0.1)	width ω of initial weights $> 0 \rightarrow$ [-PARJN(4),PARJN(4)] $< 0 \rightarrow$ [0,-PARJN(4)]
PARJN(5)	(D=0.0)	weight decay parameter λ
PARJN(6)	(D=0.0)	width σ of Gaussian noise in Langevin updating
PARJN(7)	(R)	last error per node
PARJN(8)	(R)	mean error in last update
PARJN(9)	(R)	mean error last epoch (equal to MSTJN(9) updates)
PARJN(10)	(R)	weighted average error A_n used when pruning
PARJN(11)	(D=1.0)	decrease in η (scale factor per epoch) $> 0 \rightarrow$ "bold driver" dynamics $< 0 \rightarrow$ geometric decrease
PARJN(12)		decrease in momentum alpha (scale factor per epoch).
PARJN(13)	(D=1.0)	decrease in temperature T (scale factor per epoch)
PARJN(14)	(D=0.0)	pruning parameter λ
PARJN(15)	(D=10^{-6})	change $\Delta\lambda$ of λ
PARJN(16)	(D=0.9)	parameter γ used for calculation of PARJN(10)
PARJN(17)	(D=0.9)	pruning rescaling factor c
PARJN(18)	(D=1.0)	scale parameter ω_0 used in pruning
PARJN(19)	(D=0.0)	target error D used in pruning
PARJN(20)	(D=1.0)	decrease in Langevin noise (scale factor per epoch)
PARJN(21)	(D=1.75)	maximum scale in QP updating
PARJN(22)	(D=1000.0)	maximum allowed size of weights in QP
PARJN(23)	(D=0.0)	constant added to $g'(x)$ to avoid "flat spot" in QP
PARJN(24)	(D=0.1)	line search convergence parameter ε
PARJN(25)	(D=0.05)	tolerance δ of minimum in line search
PARJN(26)	(D=0.001)	minimum allowed change in error in line search
PARJN(27)	(D=2.0)	maximum allowed step size in line search
PARJN(28)	(D=10^{-4})	constant σ_0 used in SCG for computing s
PARJN(29)	(D=10^{-6})	initial value for λ in SCG
PARJN(30)	(D=1.2)	scale-up factor γ_+ used in Rprop
PARJN(31)	(D=0.5)	scale-down factor γ_- used in Rprop
PARJN(32)	(D=50.0)	maximum scale-up factor in Rprop
PARJN(33)	(D=10^{-6})	minimum scale-down factor in Rprop
PARJN(34-40)		not used

All parameters can be changed at any time by the user.

(See [1] for descriptions on switches MSTJM and parameters PARJM.)

OIN is the vector used to pass the values of the input nodes to the program.

OUT is a vector used both to pass the desired value of the output nodes to the program during supervised training and to pass the output produced by the network given an input pattern in OIN.

- COMMON /JNDAT2/ TINV(10),IGFN(10),ETAL(10),WIDL(10),SATM(10)

TINV(I)	(D=0.0)	if greater than 0.0, this value is used as inverse temperature β in the sigmoid function for layer I, otherwise the overall inverse temperature PARJN(3) is used. Can be changed at any time by the user.
IGFN(I)	(D=0)	if greater than 0, these switches determine the sigmoid function to be used in layer I, otherwise the overall function determined by MSTJN(3) is used. These switches are only active before the network is initialized with subroutine JNINIT.
ETAL(I)	(D=0.0)	if greater than 0.0, this value is used for the learning rate η for weights in weight layer I. The weights between input and first hidden layer is considered to be weight layer number one. Can be changed at any time by the user.
WIDL(I)	(D=0.0)	if greater than 0.0, this value is used for the width ω for initial weight values in weight layer I.
SATM(I)	(R)	if MSTJN(22) $\neq$ 0 this vector contains the average saturation of nodes in layer I.

5.2.2 Internal common blocks

- COMMON /JNINT1/ O(MAXV),A(MAXV),D(MAXV),T(MAXV),DT(MAXV),W(MAXM), DW(MAXM),NSELF(MAXM),NTSELF(MAXV),G(MAXM+MAXV), ODW(MAXM),ODT(MAXV),ETAV(MAXM+MAXV)
  /JNINT1/ contains weight and node vectors for the net. Node vectors are O, A, D, T, DT, NTSELF and ODT. Weight vectors are W, DW, NSELF, G, ODW and ETAV. The default dimensions for these vectors are MAXV = 2000 and MAXM = 150000.

O(I)	current value of node I in the network, does not include the input units
A(I)	current value of the summed input a_i to node I
D(I)	current value of the δ_i at node I
T(I)	current value of the threshold θ_i (see eq. (3)) at node I
DT(I)	current value of the update $\Delta\theta_i$ for the threshold at node I
W(IW)	current value for weight with index IW
DW(IW)	current value of the update for weight with index IW
NSELF(IW)	switches for updating weight with index IW 0 $\rightarrow$ do not update 1 $\rightarrow$ update for self-organizing map, NSELF is equivalent to the vector NBHD
NTSELF(I)	switches for updating threshold for node I. 0 $\rightarrow$ do not update 1 $\rightarrow$ update
G(IW)	temporary weight and threshold vector used in CG, SCG and QP

ODW(IW)	stores old weight gradient in CG, SCG, QP and Rprop
ODT(I)	stores old threshold gradient in CG, SCG, QP and Rprop
ETAV(IW)	individual learning rates used in Rprop

- COMMON /JNINT2/ M(0:10),MV0(11),MM0(11),NG(10),NL,IPOTT,ER1,ER2,SM(10),ICPON
  /JNINT2/ contains pointers and internal switches for the feed-forward network.

M(I)	number of nodes in layer I (I=0 $\rightarrow$ input layer)
MV0(I)	offset index for node vectors - tells the index in the node vectors for the last node in layer (I-1)
MM0(I)	offset index for weight vectors - tells the index in the weight vectors for the last weight going from layer (I-1)
NG(I)	activation function $g(x)$ for layer I
NL	number of layers except input layer
IPOTT	switch for use of Potts units IPOTT = dimension of Potts units
ER1, ER2	internal variables used for calculating PARJN(7-9)
SM(I)	internal variable used for calculating saturation measures
ICPON	switch for precision chopping 1 $\rightarrow$ on

- COMMON /JNINT3/ NXIN,NYIN,NXRF,NYRF,NXHRF,NYHRF NHRF,NRFW,NHPRF
  /JNINT3/ common block for receptive fields indices.

NXIN	x-width of input field (if negative $\rightarrow$ periodic boundary) set from MSTJN(23)
NYIN	y-height of input-field (if negative $\rightarrow$ periodic boundary) set from MSTJN(24)
NXRF	x-width of the receptive field – set from MSTJN(25)
NYRF	y-height of receptive field – set from MSTJN(26)
NXHRF	number of overlapping receptive field in x-direction
NYHRF	number of overlapping receptive fields in y-direction
NHRF	total number of receptive fields
NRFW	number of weights per receptive field
NHPRF	number of nodes per receptive field – set from MSTJN(27) If NHPRF is negative, weights from equivalent receptive field nodes in the first hidden layer to each node in the second hidden layer are shared.

The geometry of the receptive fields is defined as follows: The input nodes are assumed to be organized in a plane of NXIN*NYIN nodes (with periodic boundaries if NXIN or NYIN are negative). Note that the coordinates (IX,IY) corresponds to input node number IN=(IX-1)*NYIN+IY. Each receptive field node in the first hidden layer scan an area of NXRF*NYRF input nodes.

- COMMON /JNINT4/ ILINON,NC,G2,NIT,ERRLN(0:3),DERRLN,STEPLN(0:3), STEPMN,ERRMN,IEVAL,ISUCC,ICURVE,NSC,GVEC2
  /JNINT4/ common block for CG line search parameters and SCG parameters.

ILINON	switch that tells the current status of the line search. 0 → off 1 → on and trying to bracket the minimum -1 → on and looking for the minimum (knowing that its been bracketed)
NC	number of steps taken in CG minimization
G2	squared magnitude of the vectors DW(IW) and DT(I)
NIT	number of calls to the line search (along current direction)
ERRLN(I)	stored errors used in line search I=0 → current error I>0 → previous errors
DERRLN	gradient along search direction at initial point
STEPLN(I)	coordinates, relative to current position, for previous points corresponding to ERRLN(I)
STEPMN	position, relative to current point, of best minimum so far
ERRMN	minimum error so far
IEVAL	switch for the comparison parameter in SCG training 0 → do not compute the comparison parameter 1 → compute the comparison parameter
ISUCC	switch that tells if a successful step was made in the SCG algorithm > 0 → successful step
ICURVE	switch to tell whether curvature information exists or not – used in the SCG algorithm. 0 → curvature information does not exist 1 → curvature information exists
NSC	number of attempted steps in the SCG algorithm
GVEC2	squared magnitude of the vector G(IW)

- COMMON /JNINT5/ D2E(MAXD2E,MAXD2E)
  /JNINT5/ common block used to store the Hessian matrix.

D2E(IW,IW)	the Hessian matrix

- COMMON /JNSIGM/ GPJN(MAXV),GPPJN(MAXV)
  /JNSIGM/ common block containing first and second derivatives of the activation function $g(x)$. They are computed when the function GJN is called.

GPJN(I)	first derivative $g'(x)$
GPPJN(I)	second derivative $g''(x)$

6 Restrictions and Technical Information

JETNET 3.0 includes a "test deck" subroutine called JNTDEC. If you call this subroutine, it will test JETNET 3.0 automatically.

The maximum number of layers in JETNET 3.0 is 11, including the input layer, although it is very difficult to find a situation where more than 4 is needed.

The maximum number of input nodes is 1000. This can be changed by changing the parameter MAXI in the PARAMETER statement in each routine.

The maximum number of output nodes is 1000. This can be changed by changing the parameter MAXO in the PARAMETER statement in each routine.

The maximum total number of nodes is 2000 (not including the input nodes). This can be changed by changing the parameter MAXV in the PARAMETER statement in each routine.

The maximum total number of weights is 150000 . This can be changed by changing the parameter MAXM in the PARAMETER statement in each routine.

The maximum size of the Hessian matrix is 300×300. This can be changed by changing the parameter MAXD2E in all relevant PARAMETER statements.

The code is written entirely in FORTRAN 77. All subroutine and common block names start with the characters JN or JM, except for the functions GAUSJN, GJM, GJN, ERRJN and RJN.

7 Sample Program

This sample program is the subroutine JNTDEC. It demonstrates how all training algorithms are used.

```
      SUBROUTINE JNTDEC(METHOD)
C...JetNet subroutine Test-DECk

C...Runs a test-program using data from two overlapping Gaussian
C...distributions in the input space.  The test-program uses the
C...method specified by METHOD, with values corresponding to the
C...switch MSTJN(5).

      PARAMETER(MAXI=1000,MAXO=1000)

      COMMON /JNDAT1/ MSTJN(40),PARJN(40),MSTJM(20),PARJM(20),
     & OIN(MAXI),OUT(MAXO),MXNDJM
      SAVE /JNDAT1/

      PARAMETER(INDIM=5,HIDDEN=10,NTRAIN=5000,NTEST=10000,NEPOCH=100)
      PARAMETER(WID1=1.,WID2=2.,XI=0.00,BAYES=85.2)
      DIMENSION TIN(NTRAIN+NTEST,INDIM),TOUT(NTRAIN+NTEST)

      WRITE(MSTJN(6),600)

      WRITE(MSTJN(6),610)INDIM
      WRITE(MSTJN(6),620)WID1,WID2
      WRITE(MSTJN(6),621)XI
```

```
      WRITE(MSTJN(6),*)

C...Generate data:
      WRITE(MSTJN(6),625)
      DO 100 IPAT=1,NTRAIN+NTEST
        IDUM=IPAT
        IF (RJN(IDUM).GT.0.5) THEN
          DO 110 I=1,INDIM
             TIN(IPAT,I)=WID1*GAUSJN(IDUM)
110       CONTINUE
          TOUT(IPAT)=1.0
        ELSE
          TIN(IPAT,1)=WID2*GAUSJN(IDUM)+XI
          DO 120 I=2,INDIM
             TIN(IPAT,I)=WID2*GAUSJN(IDUM)
120       CONTINUE
          TOUT(IPAT)=0.0
        ENDIF
100   CONTINUE
      WRITE(MSTJN(6),626)

C...Set network architecture:  MSTJN(1)-layered network with
C...MSTJN(11) hidden nodes, MSTJN(12) output nodes and
C...MSTJN(10) inputs.
      MSTJN(1)=3
      MSTJN(10)=INDIM
      MSTJN(11)=HIDDEN
      MSTJN(12)=1

C...Set sigmoid function:
      MSTJN(3)=1

C...Initial width of weights:
      PARJN(4)=0.5

C...Choose updating method
      MSTJN(5)=METHOD
      IF ((MSTJN(5).EQ.8).OR.(MSTJN(5).EQ.9).OR.(MSTJN(5).EQ.14).OR.
     &(MSTJN(5).LT.0).OR.(MSTJN(5).GT.15)) THEN
        WRITE(MSTJN(6),660)
        STOP 0
      ENDIF

C...Initialize network:
      CALL JNINIT

C...Set parameters suitable for the given method of updating
      IF (MSTJN(5).EQ.0) THEN
C...Normal Backprop
```

```
        PARJN(1)=2.0
        PARJN(2)=0.5
        PARJN(11)=0.999
      ELSEIF (MSTJN(5).EQ.1) THEN
C...Manhattan
        PARJN(1)=0.05
        PARJN(2)=0.5
        PARJN(11)=-0.99
      ELSEIF (MSTJN(5).EQ.2) THEN
C...Langevin
        PARJN(1)=1.0
        PARJN(2)=0.5
        PARJN(6)=0.01
        PARJN(11)=0.999
        PARJN(20)=0.99
      ELSEIF (MSTJN(5).EQ.3) THEN
C...Quickprop
        PARJN(1)=2.0
        PARJN(2)=0.0
        PARJN(6)=0.0
        PARJN(11)=1.0
        PARJN(20)=1.0
        MSTJN(2)=NTRAIN
      ELSEIF ((MSTJN(5).GE.4).AND.(MSTJN(5).LE.7)) THEN
C...Conjugate Gradient
        PARJN(1)=1.0
        MSTJN(2)=NTRAIN
      ELSEIF ((MSTJN(5).GE.10).AND.(MSTJN(5).LE.13)) THEN
C...Scaled Conjugate Gradient
        MSTJN(2)=NTRAIN
      ELSEIF (MSTJN(5).EQ.15) THEN
C...Rprop
        PARJN(1)=1.0
        MSTJN(2)=NTRAIN
      ENDIF

C...Define the size of one epoch.  Note that for batch training, the
C...number of patterns per update, MSTJN(2), must be set to the
C...total number of training patterns, and hence MSTJN(9), the
C...number of updates per epoch must be set to one.
      MSTJN(9)=MAX(1,NTRAIN/MSTJN(2))

C...Other parameters keep their default values.

      WRITE(MSTJN(6),*)
      WRITE(MSTJN(6),630)

      TESTMX=0.0
      TRNMX=0.0
```

```
C...Main loop over epochs:
      DO 300 IEPOCH=1,NEPOCH

C...Training loop:
        NRIGHT=0
        DO 310 IP=1,NTRAIN
          IF (MSTJN(5).LE.2) THEN
C...Note that for non-batch training it is often a good idea to pick
C...training patterns at random
             IPAT=INT(RJN(IP)*FLOAT(NTRAIN))+1
          ELSE
             IPAT=IP
          ENDIF

C...Put pattern into OIN:
          DO 320 I=1,MSTJN(10)
             OIN(I)=TIN(IPAT,I)
320       CONTINUE
C...Put target output value into OUT:
          OUT(1)=TOUT(IPAT)

C...Invoke training algorithm:
          CALL JNTRAL

C...Calculate performance on training set:
          IF (ABS(OUT(1)-TOUT(IPAT)).LT.0.5) NRIGHT=NRIGHT+1

310     CONTINUE
        TRAIN=FLOAT(NRIGHT)/FLOAT(NTRAIN)

        IF (MOD(IEPOCH,10).EQ.0) THEN
C...Testing loop:
          NRIGHT=0
          DO 330 IPAT=NTRAIN+1,NTRAIN+NTEST

C...Put pattern into OIN:
             DO 340 I=1,MSTJN(10)
                OIN(I)=TIN(IPAT,I)
340          CONTINUE

C...Get network output:
             CALL JNTEST

C...Calculate performance on test set (=generalization):
             IF (ABS(OUT(1)-TOUT(IPAT)).LT.0.5) NRIGHT=NRIGHT+1
330       CONTINUE
          TEST=FLOAT(NRIGHT)/FLOAT(NTEST)
          IF ((MSTJN(5).GT.3).AND.(MSTJN(5).LT.15)) THEN
             IF (TEST.GT.TESTMX) TESTMX=TEST
```

```
            IF (TRAIN.GT.TRNMX) TRNMX=TRAIN
            TEST=TESTMX
            TRAIN=TRNMX
         ENDIF

C...Display performance:
         WRITE(MSTJN(6),640)IEPOCH,TRAIN,TEST
       ENDIF

C...Terminate CG and SCG training:
       IF (IEPOCH.EQ.NEPOCH-1) THEN
         IF ((MSTJN(5).GT.3).AND.(MSTJN(5).LT.15)) THEN
            IF (MSTJN(5).LT.9) THEN
               MSTJN(5)=8
            ELSE
               MSTJN(5)=14
            ENDIF
            TRNMX=0.0
            TESTMX=0.0
         ENDIF
       ENDIF

300   CONTINUE

      WRITE(MSTJN(6),*)
      WRITE(MSTJN(6),650)BAYES
      IF (METHOD.EQ.0) THEN
        WRITE(MSTJN(6),670)
      ELSEIF (METHOD.EQ.1) THEN
        WRITE(MSTJN(6),680)
      ELSEIF (METHOD.EQ.2) THEN
        WRITE(MSTJN(6),690)
      ELSEIF (METHOD.EQ.3) THEN
        WRITE(MSTJN(6),700)
      ELSEIF (METHOD.EQ.4) THEN
        WRITE(MSTJN(6),710)
      ELSEIF (METHOD.EQ.5) THEN
        WRITE(MSTJN(6),720)
      ELSEIF (METHOD.EQ.6) THEN
        WRITE(MSTJN(6),730)
      ELSEIF (METHOD.EQ.7) THEN
        WRITE(MSTJN(6),740)
      ELSEIF (METHOD.EQ.10) THEN
        WRITE(MSTJN(6),750)
      ELSEIF (METHOD.EQ.11) THEN
        WRITE(MSTJN(6),760)
      ELSEIF (METHOD.EQ.12) THEN
        WRITE(MSTJN(6),770)
      ELSEIF (METHOD.EQ.13) THEN
```

```
        WRITE(MSTJN(6),780)
      ELSEIF (METHOD.EQ.15) THEN
        WRITE(MSTJN(6),790)
      ENDIF

600   FORMAT(31X,'JETNET Test-Deck')
610   FORMAT(15X,'Two overlapping Gaussian distributions in ',
     &I2,' dimensions.')
620   FORMAT(15X,'Their standard deviations are ',F3.1,' and ',F3.1)
621   FORMAT(15X,'Their mean values are separated by ',F4.2)
625   FORMAT(15X,'Generating training and test patterns...')
626   FORMAT(15X,'...done generating data.')
630   FORMAT(' Epoch / Training / General.  ')
640   FORMAT(I8,2X,2(' /',F9.3,2X))
650   FORMAT(' The optimal generalization performance is ',F4.1,'%')
660   FORMAT(' Undefined training algorithm in call to JNTDEC')
670   FORMAT(' Backprop should reach (81.0 +- 2.2)% in 100 epochs')
680   FORMAT(' Manhattan should reach (84.3 +- 0.6)% in 100 epochs')
690   FORMAT(' Langevin should reach (82.9 +- 1.8)% in 100 epochs')
700   FORMAT(' Quickprop should reach (82.8 +- 8.8)% in 100 epochs')
710   FORMAT(' Polak-Ribiere CG should reach (79.0 +- 7.0)% in 100',
     &' epochs')
720   FORMAT(' Hestenes-Stiefel CG should reach (79.8 +- 5.6)% in 100',
     &' epochs')
730   FORMAT(' Fletcher-Reeves CG should reach (79.6 +- 5.6)% in 100',
     &' epochs')
740   FORMAT(' Shanno CG should reach (71.7 +- 11.6)% in 100 epochs')
750   FORMAT(' Polak-Ribiere SCG should reach (84.0 +- 1.6)% in 100',
     &' epochs')
760   FORMAT(' Hestenes-Stiefel SCG should reach (84.1 +- 2.6)% in 100',
     &' epochs')
770   FORMAT(' Fletcher-Reeves SCG should reach (81.4 +- 5.2)% in 100',
     &' epochs')
780   FORMAT(' Shanno SCG should reach (70.7 +- 8.1)% in 100 epochs')
790   FORMAT(' Rprop should reach (83.5 +- 2.2)% in 100 epochs')

      RETURN

C**** END OF JNTDEC ****************************************************
      END
```


Acknowledgements

One of the authors (T.R.) thanks Dr. M. Berggren for useful suggestions and for sharing his experience with Conjugate Gradients. The Göran Gustafsson Foundation for Research in Natural Science and Medicine is acknowledged for financial support.

References

[1] L. Lönnblad, C. Peterson and T. Rögnvaldsson, "Pattern Recognition in High Energy Physics with Artificial Neural Networks", *Comput. Phys. Commun.* **70**, 167 (1992).

[2] L. Lönnblad, C. Peterson and T. Rögnvaldsson, "Using Neural Networks to Identify Jets", *Nucl. Phys.* **B 349**, 675 (1991).

[3] L. Lönnblad, C. Peterson and T. Rögnvaldsson, "Finding Gluon Jets with a Neural trigger", *Phys. Rev. Lett.* **65**, 1321 (1990).

[4] L. Lönnblad, C. Peterson, H. Pi and T. Rögnvaldsson, "Self-organizing Networks for Extracting Jet Features", *Comput. Phys. Commun.* **67**, 193 (1991).

[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] T. Rögnvaldsson, "On Langevin Updating in Multilayer Perceptrons", Lund Preprint *LU TP 93-13* (to appear in *Neural Comput.*) (1994).

[7] E. M. Johansson, F. U. Dowla and D. M. Goodman, "Backpropagation Learning for Multilayer Feed-forward Neural Networks using the Conjugate Gradient Method", *Int. J. Neur. Syst.* **2**, 291 (1992).

[8] M. F. Møller, "A Scaled Conjugate Gradient Algorithm for Fast Supervised Learning", *Neural Networks* **6**, 525 (1993).

[9] S. E. Fahlman, "An Empirical Study of Learning Speed in Back-propagation Networks", Carnegie-Mellon Computer Science Rpt. *CMU-CS-88-162* (1988).

[10] M. Riedmiller and H. Braun, "A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP Algorithm", *Proc. ICNN*, San Fransisco (1993).

[11] B. Denby, "Neural Networks and Cellular Automata in Experimental High Energy Physics", *Comput. Phys. Commun.* **49**, 429 (1988).

[12] C. Peterson, "Track Finding with Neural Networks", *Nucl. Instrum. Methods* **A279**, 537 (1989).

[13] M. Gyulassy and H. Harlander, "Elastic Tracking and Neural Network Algorithms for Complex Pattern Recognition", *Comput. Phys. Commun.* **66**, 31 (1991).

[14] A. Yuille, K. Honda and C. Peterson, "Particle Tracking by Deformable Templates", *Proceedings of 1991 IEEE INNS International Joint Conference on Neural Networks*, Vol. 1, pp 7-12, Seattle, WA (July 1991)

[15] M. Ohlsson, C. Peterson and A. Yuille, "Track Finding with Deformable Templates – The Elastic Arms Approach", *Comput. Phys. Commun.* **71**, 77 (1992).

[16] M. Ohlsson, "Extensions and Explorations of the Elastic Arms Algorithm", *Comput. Phys. Commun.* **77**, 19 (1992).

[17] C. Peterson and E. Hartman, "Explorations of the Mean Field Theory Learning Algorithm", *Neural Networks* **2**, 475 (1989).

[18] S. Saarinen, R. Bramley and G. Cybenko, "Ill-conditioning in Neural Network Training Problem", *SIAM J. Sci. Comp.* **14**, 693 (1993).

[19] R. Battiti, "First- and Second-Order Methods for Learning: Between Steepest Descent and Newton's Method", *Neural Comput.* **4**, 141 (1992).

[20] A. C. Veitch and G. Holmes, "A Modified Quickprop Algorithm", *Neural Comput.* **3**, 310 (1991).

[21] R. Jacobs, "Increased Rates of Convergence Through Learning Rate Adaption", *Neural Networks* **1**, 295 (1988).

[22] T. Tollenaere, "SuperSAB: Fast Adaptive Backpropagation with Good Scaling Properties", *Neural Networks* **3**, 561 (1990).

[23] W. H. Press, B. P. Flannery, S. A. Teukolsky and W. T. Vetterling, *Numerical Recipes*, Cambridge Univ. Press, Cambridge UK (1986).

[24] M. D. Richard and R. P. Lippmann, "Neural Network Classifiers Estimate Bayesian *a posteriori* Probabilities", *Neural Comput.* **3**, 461 (1991).

[25] S. Geman, E. Bienenstock and R. Doursat, "Neural Networks and the Bias/Variance Dilemma", *Neural Comput.* **4**, 1 (1992).

[26] K. H. Becks, F. Block, J. Drees, P. Langefeld and F. Seidel, "B-quark Tagging using Neural Networks and Multivariate Statistical Methods – A Comparison of Both Techniques", *Nucl. Instrum. Methods* **A329**, 501 (1993).

[27] J. Proriol et. al., "Tagging B Quark Events in Aleph with Neural Networks", *Proceedings of Workshop in Neural Networks: From Biology to High Energy Physics, June 1991, Elba, Italy*, eds. O. Benhar, C. Bosio, P. Del Giudice and E. Tabet, ETS EDITRICE (Pisa 1991).

[28] R. Belloti *et al.*, "A Comparison Between a Neural Network and the Likelihood Method to Evaluate the Performance of a Transition Radiation Detector", *Comput. Phys. Commun.* **78**, 17 (1993).

[29] G. Stimpfl-Abele, "Recognition of Charged Tracks with Neural Network Techniques", *Comput. Phys. Commun.* **67**, 183 (1991).

[30] G. Stimpfl-Abele and P. Yepes, "Higgs Search and Neural Net Analysis", *Comput. Phys. Commun.* **78**, 1 (1993).

[31] W. S. Babbage and L. F. Thompson, "The Use of Neural Networks in $\gamma - \pi^0$ Discrimination", *Nucl. Instrum. Methods* **A330**, 482 (1993).

[32] J. F. Kreider and J. S. Haberl, "Predicting Hourly Building Energy Usage: The Great Energy Predictor Shootout – Overview and Discussion of Results", to appear in *1994 ASHRAE Trans.* **100**, part 2 (1994).

[33] B. D. Ripley, "Flexible Non-linear Approaches to Classification", in V. Cherkassky, J. H. Friedman and H. Wechsler (eds.) *From Statistics to Neural Networks* NATO ASI Proceedings, subseries F, Springer-Verlag (1993).

[34] R. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, Wiley: New York (1973).

[35] A. Murray, "Multilayer Perceptron Learning Optimized for On-chip Implementation: A Noise Robust System", *Neural Comput.* **4**, 366 (1992).

[36] A. R. Barron, "Approximation and Estimation Bounds for Artificial Neural Networks", *Machine Learning* **14**, 115 (1994).

[37] T. Kohonen, *Self-organization and Associative Memory* 3rd ed., Springer-Verlag, Heidelberg (1990).

[38] T. Rögnvaldsson, "Pattern Discrimination Using Feedforward Networks: A Benchmark Study of Scaling Behavior", *Neural Comput.* **5**, 483 (1993).

[39] J. Proriol, "Multi-modular Networks for the Classification of e^+e^- Hadronic Events", to appear in *Nucl. Instrum. Methods* **A**, (1994).

[40] S. E. Fahlman and C. Lebiere, "The Cascade Correlation Learning Architecture", Carnegie Mellon Computer Science Rpt. *CMU-CS-90-100* (1990).

[41] E. Hartman and J. D. Keeler, "Predicting the Future: Advantages of Semilocal Units", *Neural Comput.* **3**, 566 (1991).

[42] J. Moody and C. J. Darken, "Fast Learning in Networks of Locally-tuned Processing Units", *Neural Comput.* **1**, 281 (1989).

[43] L. Lönnblad, C. Peterson and T. Rögnvaldsson, "Mass Reconstruction with a Neural Network", *Phys. Lett.* **B278**, 181 (1992).

[44] J. MacQueen, "Some Methods for Classification and Analysis of Multivariate Observations", *Proc. 5th Berkeley Symposium Math. Stat. and Prob.*, J. M. LeCam and J. Neyman (eds.), Univ. of California Press, Berkeley (1967).

[45] T. M. Martinetz, H. Ritter and K. J. Schulten, "Three-dimensional Neural Net for Learning Visuomotor-Coordination of a Robot Arm", *IEEE Trans. Neur. Netw.* **1**, 131 (1989).

[46] T. M. Martinetz, S. G. Berkovich and K. J.Schulten, "Neural-Gas Network for Vector Quantization and its Application to Time-Series Prediction", *IEEE Trans. Neur. Netw.* **4**, 558 (1993).

[47] L. Breiman, J. H. Friedman, R. A. Olsen and C. J. Stone, *Classification and Regression Trees*, Wadsworth, Monterey CA (1984).

[48] J. H. Friedman, "Multivariate Adaptive Regression Splines", *Ann. of Stat.* **19**, 1 (1991).

[49] M. Ohlsson, C. Peterson, H. Pi, T.Rögnvaldsson and B. Söderberg, "Predicting Utility Loads with Artificial Neural Networks – Methods and Results from the Great Energy Predictor Shootout", Lund Preprint *LU TP 93-24* (to appear in *1994 ASHRAE Trans.* **100**, part2), (1994).

[50] K. Fukunaga, *Introduction to Statistical Pattern Recognition*, 2:nd ed., Academic Press Inc., San Diego CA (1990).

[51] A. A. Chilingarian and G. Z. Zazian, "A Bootstrap Method of Distribution Mixture Proportion Determination", *Pat. Rec. Lett.* **11**, 781 (1990).

[52] J. Utans and J. Moody, "Selecting Neural Network Architecture via the Prediction Risk: Application to Corporate Bond Rating Prediction", *Proc. First Intl. Conf. on AI Appl. on Wall Street*, IEEE Press, Los Alamitos CA (1991).

[53] J. Moody, "The Effective Number of Parameters: An Analysis of Generalization and Regularization in Nonlinear Learning Systems", *Adv. in Neur. Inf. Proc. Syst.* **4**, Morgan Kaufmann, San Mateo CA (1992).

[54] N. Murata, S. Yoshizawa and S. Amari, "Network Information Criterion – Determining the Number of Hidden Units for an Artificial Neural Network Model", to appear in *IEEE Trans. Neur. Netw.*, (1993).

[55] G. Cybenko, "Approximation by Superposition of a Sigmoidal Function", *Math. Control Signals Systems* **2**, 303 (1989).

[56] H. H. Szu, X. Yang, B. A. Telfer, Y. Sheng, "Neural Network and Wavelet Transform for Scale-invariant Data Classification", *Phys. Rev.* **E 48**, 48 (1993).

[57] S. J. Nowlan and G. E. Hinton, "Simplifying Neural Networks by Soft Weight-Sharing", *Neural Comput.* **4**, 473 (1992).

[58] J. M. J. Murre, "Neurosimulators", review to appear in M. A. Arbib (ed.) *Handbook of Brain Research and Neural Networks*, MIT Press (1995).

[59] W. Schiffmann, M. Joost and R. Werner, "Comparison of Optimized Backpropagation Algorithms", *Proc. ESANN 93* Brussels (1993).

[60] D. Buskulic, et. al., "Measurement of the Ratio $\Gamma_{b\bar{b}}/\Gamma_{had}$ using Event Shape Variables", *Phys. Lett.* **B313**, 549 (1993).

[61] H. Pi and C. Peterson, "Finding the Embedding Dimension and Variable Dependencies in Time Series", Lund Preprint *LU TP 93-4* (to appear in *Neural Comput.*) (1994).

[62] A. A. Chilingarian, "Statistical Decisions under Nonparametric a Priori Information", *Comput. Phys. Commun.* **54**, 381 (1989).

[63] A. Weigend, B. Huberman and D. Rumelhart, "Predicting Sunspots and Exchange Rates with Connectionist Networks", *Nonlin. Modeling and Forecasting*, Addison-Wesley (1991).

[64] M. C. Mozer and P. Smolensky, "Using Relevance to Reduce Network Size Automatically", *Connection Science* **1**, 3 (1989).

[65] Y. Le Cun, J. S. Denker and S. A. Solla, "Optimal Brain Damage", *Neur. Inform. Proc. Systems* **2**, 598 (1990).

[66] B. Hassibi and D. G. Stork, "Second Order Derivatives for Network Pruning: Optimal Brain Surgeon", *Neur. Inform. Proc. Systems* **5**, 164 (1993).

[67] L. F. A. Wessels and E. Barnard, "Avoiding False Local Minima by Proper Initialization of Connections", *IEEE Trans. Neur. Netw.* **3**, 899 (1992).

[68] T. Denoeux and R. Lengellé, "Initializing Back Propagation Networks with Prototypes", *Neural Networks* **6**, 351 (1993).

[69] T. P. Vogl *et al.*, "Accelerating the Convergence of the Back-Propagation Method", *Biol. Cybern.* **59**, 257 (1988).

[70] C. Darken, J. Chang and J. Moody, "Learning Rate Schedules for Faster Stochastic Gradient Descent", *Proc. 1992 IEEE Worksh. Neur. Netw. Signal Processing*, 3 (1992).

[71] Y. LeCun, P. Y. Simard and B. Pearlmutter, "Automatic Learning Rate Maximization by On-Line Estimation of the Hessian's Eigenvectors", *Proc. Neur. Inf. Proc. Syst.* **5**, 156 (1993).

[72] T. M. Heskes and B. Kappen, "Learning-parameter Adjustment in Neural Networks", *Phys. Rev.* **A 45**, 8885 (1992).

[73] F. James, "MINUIT", *Comput. Phys. Commun.* **10**, 343 (1975)

[74] O. Catoni, "Rough Large Deviation Estimates for Simulated Annealing Applications to Exponential Schedules", *Ann. of Probability* **20**, 1109 (1992).

[75] F. James, "A Review of Pseudorandom Number Generators", *Comput. Phys. Commun.* **60**, 329 (1990).