Computer Physics Communications 70 (1992) 167–182
North-Holland

Computer Physics Communications

Pattern recognition in high energy physics with artificial neural networks – JETNET 2.0

Leif Lönnblad, Carsten Peterson and Thorsteinn Rögnvaldsson

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

Received 27 August 1991

A F77 package of adaptive artificial neural network algorithms, JETNET 2.0, is presented. Its primary target is the high energy physics community, but it is general enough to be used in any pattern-recognition application area. The basic ingredients are the multilayer perceptron back-propagation algorithm and the topological self-organizing map. The package consists of a set of subroutines, which can either be used with standard options or be easily modified to host alternative architectures and procedures.

PROGRAM SUMMARY

Title of program: JETNET version 2.0

Catalogue number: ACGV

Program obtainable from: CPC Program Library, Queen's University of Belfast, N. Ireland (see application form in this issue)

Licensing provisions: none

Computer for which the program is designed: DECstation, SUN, Apollo, VAX, IBM and others with a F77 compiler
Computer: DECstation 3100; *Installation*: Department of Theoretical Physics, University of Lund, Sweden

Operating system under which the program has been tested: ULTRIX RISC 4.2

Programming language used: FORTRAN 77

Memory required to execute with typical data: ~ 90 kwords

No. of bits in a word: 32

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

No. of lines in distributed program, including test deck data, etc.: 3345

Keywords: pattern recognition, jet identification, artificial neural network

Nature of physical problem
High energy physics offers many challenging pattern-recognition problems. It could be separating photons from leptons based on calorimeter information or the identification of a quark based on the kinematics of the hadronic fragmentation products. Standard procedures for such recognition problems is the introduction of relevant cuts in the multi-dimensional data.

Method of solution
Artificial neural networks (ANN) have turned out to be a very powerful paradigm for automated feature recognition in a wide range of problem areas. In particular feed-forward multilayer networks are widely used due to their simplicity and good performance. JETNET 2.0 implements such a network with the back-propagation updating algorithm in

F77. Also a self-organizing map algorithm is included. JETNET 2.0 consists of a number of subroutines, most of which handle training and test data, which should be loaded with a main application specific program supplied by the user. The package was originally mainly intended for jet-triggering applications, where it has been used with success for heavy-quark tagging and quark–gluon separation, but it is of general nature and can be used for any pattern-recognition problem area.

Restriction on the complexity of the problem
The only restriction on the complexity of an application is set by available memory and CPU time (see below). For a problem that is encoded with n_i input nodes, n_o output (feature) nodes, H layers of hidden nodes with $n_{h(j)}(j=1,\dots,H)$ nodes in each layer the program requires (with full connectivity) the storage of $2M_c$ real numbers given by

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

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

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

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

Typical running time
The CPU time consumption for the learning process is proportional to M_c, the number of training patterns N_p and the number of learning passes (or epochs) N_{epoch} needed:

$\tau \propto N_{epoch} N_p M_c.$

As an example we take a b-quark identification as in ref. [1] where 100 epochs are required to train a network with 16 input nodes, one output node and one hidden layer with 10 nodes. With 6000 training patterns, $\tau \approx 360$ s (on the DEC 3100).

Reference
[1] L. Lönnblad, C. Peterson and T. Rögnvaldsson, Using neural networks to identify jets, Nucl. Phys. B 349 (1991) 675.

LONG WRITE-UP

1. Introduction

When analyzing experimental data, it is a standard procedure to make various cuts in observed kinematical variables x_k in order to single out desired features. A specific choice of cuts corresponds to a particular set of feature functions $o_i = F_i(x_1, x_2 \dots) = F_i(\boldsymbol{x})$ 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 funcitons F_i, which is exactly what feature-recognition artificial neural networks (ANN) aim at. For a feed-forward ANN the following form of F_i is often chosen:

$$F_i(\boldsymbol{x}) = g\left(\frac{1}{T}\sum_j \omega_{ij} g\left(\frac{1}{T}\sum_k \omega_{jk} x_k + \theta_j\right) + \theta_i\right), \qquad (1)$$

where ω_{ij} and ω_{jk} are the parameters to be fitted to the data distributions and $g(x)$ is the non-linear neuron *transfer function*

$$g(x) = 0.5[1 + \tanh(x)]. \qquad (2)$$

Equation (1) corresponds to the feed-forward architecture of fig. 1. The bias, or threshold, terms θ_j and θ_i are easily generalized into the weights ω_{ij} and ω_{jk} by adding an extra "dummy" unit to each layer. The bottom layer (input) corresponds to the measured kinematical variables x_k and the top layer to the (output) features o_i. The task of the so-called hidden layer is to build up an internal representation of the observed data. Equation (1) and fig. 1 are easily generalized to more than one hidden layer. Fitting to a given data set (or learning) takes place with e.g. gradient descent

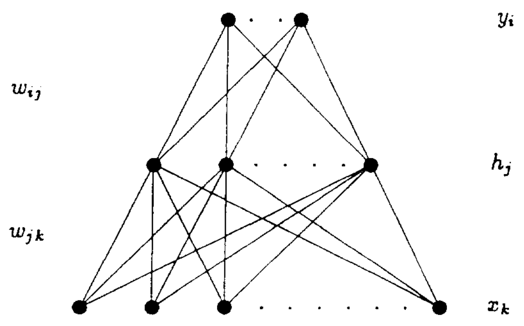

Fig. 1. A one-hidden-layer feed-forward neural-network architecture.

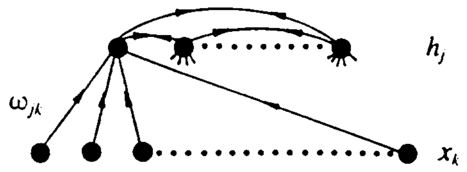

Fig. 2. A one-layer self-organizing network. Lateral interactions between the feature nodes correspond to the "mexican-hat"-like potential.

on a summed square error between o_i and the desired feature values t_i (targets) with respect to the weights ω_{ij} and ω_{jk}.

This *back-propagation* [2] procedure assumes knowledge about what features (o_i) are relevant from the outset and separates the data accordingly. There are also alternative approaches, like *self-organization* where the network organizes the data into features without any external teacher (no target values) [3]. The underlying architecture consists of an input layer (x_k) and a layer of feature nodes denoted h_j (see fig. 2). These feature nodes are updated according to

$$h_j = G\left(\sum_k (\omega_{jk} - x_k)^2 \right) = G\left(\| \boldsymbol{\omega}_j - \boldsymbol{x} \|^2 \right), \tag{3}$$

where $G(x)$, the neuron *response function*, is a monotonically decreasing function. For all patterns presented the weights are updated such that the distances between $\boldsymbol{\omega}_j$ and $\boldsymbol{x}$ are minimized. Also, topological ordering between the feature nodes h_j is introduced with a "mexican-hat"-like potential, such that neighboring nodes in a plane react to similar features.

The JETNET 2.0 package implements both the back-propagation and self-organizing algorithms in a stream-lined way. In sections 2, 3 and 4 we briefly describe these algorithms together with modifications, extensions and practical hints. For a more extensive list of references we refer the reader to refs. [1–5].

JETNET 2.0, which is backwards compatible with the earlier 1.0 version [1] has been extended with the following capabilities:

- includes self-organization (JETMAP 1.0);
- includes learning vector quantization (JETMAP 1.0);
- allows for selective receptive fields;
- allows for different learning rates between different layers;
- allows for different temperatures (gains) at different layers;
- allows for different initial widths of weights in different layers;
- allows for decay of unused weights;
- allows for limited precision (number of bits) simulations;
- includes (Langevin) noise in learning;
- includes a pruning procedure for determining minimum architecture;
- includes the possibility of monitoring network saturation.

2. The back-propagation learning algorithm

The key ingredient is the fitting procedure, i.e. minimizing a summed square error for all patterns p,

$$E = \tfrac{1}{2} \sum_p \sum_i (o_i - t_i)^2, \tag{4}$$

with respect to the parameters ω. Gradient descent with respect to ω_{ij} and ω_{jk} for each pattern corresponds to

$$\Delta \omega_{ij} = -\eta \frac{\partial E}{\partial \omega_{ij}} \tag{5}$$

and

$$\Delta \omega_{jk} = -\eta \frac{\partial E}{\partial \omega_{jk}}. \tag{6}$$

The chain rule for the partial derivatives with respect to E gives

$$\frac{\partial E}{\partial \omega_{ij}} = \delta_i h_j \tag{7}$$

and

$$\frac{\partial E}{\partial \omega_{jk}} = \sum_i \delta_i \omega_{ij} g'(a_j) x_k / T, \tag{8}$$

where δ_i is given by

$$\delta_i = (o_i + t_i) g'(a_i)/T. \tag{9}$$

The "temperature" T sets the gain of $g(x)$ and a_i is the summed input signal to node i. Equations (7) and (8) trivially generalize to architectures with more than one hidden layer; the δ's are just back-propagated further down according to equations identical to eq. (8).

A few useful variations and extensions of this algorithm exist:

Alternative error measures. Other error measures than the one in eq. (4) are sometimes used. One is the so called *cross-entropy error*:

$$E = -\sum_p \sum_i \left[t_i^{(p)} \log o_i + \left(1 - t_i^{(p)}\right) \log(1 - o_i) \right]. \tag{10}$$

In this case the $g'(\ )$ factor in the updating of ω_{ij} disappears, but the updating of the hidden–hidden and input–hidden connections are the same (eq. (8)).

Another error measure variant is the so-called *Kullback measure*,

$$E = \sum_p \sum_i t_i^{(p)} \log \frac{t_i^{(p)}}{o_i}, \tag{11}$$

which is used when replacing the binary neurons in the output layer with K-valued ones (see e.g. ref. [1]). The sigmoid updating rule of eq. (2) is then replaced by

$$o_i = o(a_1, a_2, \ldots, a_n, T) = \frac{\exp(a_i/T)}{\sum_l \exp(a_l/T)}, \tag{12}$$

satisfying

$$\sum_i^n o_i = 1. \tag{13}$$

This encoding is often efficient in "winner-takes-all" situations. The back-propagation algorithm turns out to be the same as in the case of the entropy error above, again without a $g'(\ )$ between top hidden and output layers.

Linear output nodes. So far we have only been concerned about using the ANN for classification, i.e. the output nodes represent binary decisions. In case one wants to use the network as a plain "fitting engine", the output nodes should encode any real number; i.e. be linear ($g(x) = x$). In this case $g'(\ )$ also disappears from eqs. (7) but the lower layers are the same.

Updating parameters. Back-propagation in its basic form has two parameters, the temperature (inverse gain) T and the learning rate η. Sometimes it is profitable to have different values for these parameters for different layers. Also, learning can be more efficient (less oscillatory) by including a so-called momentum term in eqs. (5) and (6),

$$\Delta\omega(t+1) = -\eta \frac{\partial E}{\partial \omega} + \alpha\, \Delta\omega(t), \tag{14}$$

where $\Delta\omega(t)$ refers to the previous updating and α is a parameter.

Pruning. An important question is what ANN architecture (number of layers, hidden nodes and degree of connectivity) to use for a particular problem. Clearly one should use as few parameters (ω) as possible, in order to have powerful generalization properties. A straight-forward, but costly, way is of course trial-and-error. However, it would be advantageous to use a more algorithmic method. One such *pruning* procedure goes as follows [6]: Add to the error function (eq. (4)) a complexity term [6]

$$E \to E + \lambda \sum_{ij} \frac{\omega_{ij}^2/\omega_0^2}{1 + \omega_{ij}^2/\omega_0^2}, \tag{15}$$

where the sum extends over all weights. For large weights $|\omega_{ij}|$, the cost is λ, whereas for small weights it is zero. The scale of the weights is ω_0. Hence the network gets pruned to only contain weights that are really needed to represent the problem.

Weight decay. An alternative way of shrinking the number of weights is to allow unused weights to decay. The updating equations then become

$$\Delta\omega(t+1) = -\eta\frac{\partial E}{\partial\omega} - \epsilon\omega, \tag{16}$$

where ϵ is the decay parameter, typically a very small number, $\mathcal{O}(10^{-4})$. Weights that are not updated frequently are thus allowed to decay away.

3. Self-organization

The self-organizing algorithm is very similar to the *k-means* clustering algorithm [7]. To every feature node h_j belongs a weight vector $\boldsymbol{\omega}_j$ of the same dimension as the input vectors $\boldsymbol{x}$. For every input pattern p the node h_m whose weight vector ω_m is closest (by some measure, e.g. Euclidean distance) to the pattern $\boldsymbol{x}^{(p)}$ is termed the "winner". The winning vector ω_m is then moved closer to the input pattern p:

$$\Delta\boldsymbol{\omega}_m = \eta\left(\boldsymbol{x}^{(p)} - \boldsymbol{\omega}_m\right), \tag{17}$$

where again η is the learning-rate parameter. This amounts to gradient descent on the error function

$$E_m = \tfrac{1}{2}\sum_{p\in\mathcal{M}}\left(\boldsymbol{x}^{(p)} - \boldsymbol{\omega}_m\right)^2, \tag{18}$$

where $\mathcal{M}$ is the set of input patterns that has node h_m as winner *. This results in the weight vector ω_m approaching $\langle \boldsymbol{x}\rangle_{\mathcal{M}}$, the center-of-mass of the "cluster" $\mathcal{M}$. The feature nodes will thus divide the input-space into a number of polyhedral regions.

The topological ordering is achieved by also updating the neighbors of the winner node, the updating equation becomes (cf. eq. (17))

$$\Delta\boldsymbol{\omega}_j = \eta\Lambda(j, m)\left(\boldsymbol{x}^{(p)} - \boldsymbol{\omega}_j\right), \tag{19}$$

where $\Lambda(j, m)$ is the *neighborhood function*, emulating lateral connections between the feature nodes. $\Lambda(j, m)$ can be "mexican-hat"-like function, or any function that is 1 for $j = m$ and falls off with distance $\|\boldsymbol{r}_j - \boldsymbol{r}_m\|$ from the winner unit. JETNET 2.0 implements the so-called *short-cut* version where the entire neighborhood is updated equally, i.e.

$$\Lambda(j, m) = \begin{cases} 1, & \text{if } \|\boldsymbol{r}_j - \boldsymbol{r}_m\| \le \lambda, \\ 0, & \text{otherwise}. \end{cases} \tag{20}$$

One can supplement the self-organizing algorithm with supervised learning, *learning vector quantization* (*LVQ*), which amounts to changing the sign of η in eq. (17) whenever input data activates the wrong units – moving the weight vectors away from undesired input patterns.

* Note that $\mathcal{M}$ is not static – it changes during training.

4. Practical implementation issues

4.1. Back-propagation

Number of layers: Feed-forward networks are commonly used for prediction and classification tasks. In principle any such task can be performed with two hidden layers [8], or even with only one hidden layer [9] (this could be at the expense of an enormous total number of hidden units).

Number of nodes: In prediction and classification one aims at good performance in *generalization*, i.e. network performance on a set of test data it has never seen before. Using too many parameters (weights) will cause the net to overfit the data. Too few parameters, on the other hand, increase the probability of the network getting stuck in a local minimum during training. The generalization error for one hidden layer is $\mathcal{O}(N_\omega/N_p)$, where N_ω is the number of weights and N_p is the number of patterns (see ref. [11]). As a rule of thumb, one should have $N_\omega \le 0.1\ N_p$.

Initial weight values: The weights are initiated at random in the range $[-\omega_0, +\omega_0]$. In the beginning of learning, one should be in the central (linear) part of the sigmoid with large gradient. The width should thus be $\omega \approx \omega_0/(\text{fan-in})$, where fan-in is the number of units feeding to the actual unit and ω_0 is sufficiently small to be in the central part of the sigmoid. JETNET 2.0 allows for different initial widths for different layers with different fan-in.

Learning rate: Ideally, the learning rate η should be allowed to change during training, depending on how well the network is learning. Too high η-values cause oscillations in the training. η should approach zero in the end of training to allow the net to settle into a stable state.

The optimum learning rate scales with fan-in, i.e. the learning rate η_{ij} affecting the weights ω_{ij} (see fig. 1) scales like $\eta_{ij} \propto 1/(\text{fan-in to node } j)$. JETNET 2.0 allows for different learning rates in different layers.

Updating frequency: Gradient descent on eq. (4) means averaging over the complete training set, so-called *batch-mode* or *off-line* back-propagation. An alternative (*on-line*) way is to update the weights after each pattern, or a small subset of patterns. The latter method avoids local minima, provided that patterns are chosen at random, and is often faster.

Momentum term: The momentum term (eq. (4)) averages over past updatings and prevents oscillation. The parameter α must be between 0 and 1; it is often set very close to 1. The optimum α-value varies from problem to problem. Sometimes a momentum term deteriorates the performance.

Receptive fields: In many pattern recognition problems there are invariances inherent in the data, like translational invariance. With large enough training sets the network should capture these. However, for finite training sets it could pay off to impose these explicitly. One way is to preprocess the data performing Fourier transforms. For translational invariance it has turned out to be efficient to introduce receptive fields of weights that cover different areas of the input space and are linked together in the updating (see e.g. ref. [2]).

Manhattan updating: In cases where there are many patterns/updates, it might be beneficial to replace the standard gradient descent with "Manhattan" updating [10]

$$\Delta\omega_{ij} = -\eta \operatorname{sign}(\partial E/\partial\omega_{ij}) \tag{21}$$

In this case the learning is bounded and it is easier to find an appropriate value for η, which should decrease with increasing learning. This updating procedure is convenient for networks with more than one hidden layer.

Langevin updating: In order to avoid getting stuck in local minima one might want to introduce noise into the system. One option is to add noise to the input data, which is easily done (not an option in JETNET 2.0). Another alternative is to add a noise term to the updating equations. This is accomplished in JETNET 2.0 with so-called Langevin updating equations, which generate weight configurations within the Boltzmann ensemble.

Pruning: The pruning algorithm adopted in JETNET 2.0 is described in ref. [6], and we refer to this reference for a more extensive description. It automatically adjusts λ in eq. (15) by comparing three measures:

- E_{n-1}, previous error;
- A_n, exponentially weighted average error. $A_n = \gamma A_{n-1} + (1-\gamma)E_n$ with γ relatively close to 1.
- D, desired error, which is externally provided. For hard (e.g. "real") problems, D is set just below the chance performance error.

The crucial thing is to set D right. A value just below chance performance works fine on mirror symmetry problems of different dimensionality. The parameter ω_0 in eq. (15) should be of order unity for transfer functions of order unity. After each epoch, E_n is evaluated and λ is changed accordingly:

- $\lambda = \lambda + \Delta\lambda$ if the error is below the desired error or still falling ($E_n < D$ or $E_n < E_{n-1}$).
- $\lambda = \lambda - \Delta\lambda$ if the error is above the desired error and increasing, but still below the long

term average ($E_n > D$, $E_n > E_{n-1}$ and $E_n < A_n$). If λ becomes negative it is set to zero.

- $\lambda = 0.9\lambda$ if the error is above the desired error, increasing and larger than the long time average ($E_n > D$, $E_n > E_{n-1}$ and $E_n > A_n$). In this last case the pruning has grown too strong and λ has to be "cut".

4.2. Self-organization

Number of nodes: The goal of unsupervised self-organization is usualy to extract some features of the data or relations between data-types. There is consequently no problem of generalization, unless one is doing LVQ. For LVQ the comments in the previous section also apply.

The relative generalization performance of LVQ and back-propagation (BP) depends on the problem. For high-dimensional problems BP gives best performance, since LVQ fills the input-space with polyhedrals, whereas BP (see e.g (1)) divides the space by using hyperplanes. The latter is more economical for high-dimensinal problems

Topology of the network: The optimum topology of the net (dimensionality, periodic boundaries, etc.) is difficult to predict without explicit knowledge of the topology of the input data. If the dimensionality of the input space is higher than the dimensionality of the feature map, the map will try to fill out the space as well as possible (like a snake rolling up to fill the bottom of a basket), provided that the plasticity of the net is high. JETNET 2.0 only supports one- and two-dimensional feature maps.

Initial weight values: The input data usually occupies only a small subspace of the input-space. To shorten learning time, initial weight values should be chosen inside this subspace, by assigning values from some input data to the weights, or at least in the same "quadrant" as the input data.

Learning rate: The initialization of weights values may produce twists and kinks in the net. The initial adjustments of the net should therefore be large in order to produce the correct topological order. Towards the end of training the adjustments should approach zero in order for the net to settle into a stable state.

Size of neighborhood: Initially, all weight vectors need adjusting. The initial size of the neighborhood, the parameter λ, should thus be large. The size is then allowed to shrink, in order for the weight vectors to converge towards their cluster centers. Decreasing λ means decreasing the coupling between the weight vectors, increasing the plasticity of the net.

Temperature: The default response function for the self-organizing map is

$$h_j = \exp\left(-\| \boldsymbol{\omega}_j - \boldsymbol{x} \|^2 / T\right) \qquad (22)$$

where the "temperature" T sets the width of the Gaussian. If T is too small, the exponent will overflow. This will manifest itself by a "no response in net" warning issued by JETNET. In case this happens, the temperature T, should thus be increased (which means decreasing the inverse temperature β).

5. Limited precision – hardware implementations

A real asset in the ANN approach is the possibility of custom-made hardware for real-time applications. In case such an implementation is planned for a particular application, it is important to do the training with the number of bits precision that the hardware will provide. We have included such an option in the back-propagation case.

6. Program components

JETNET 2.0 is a F77 subroutine package, meaning that it contains a number of subroutines that are called from a main program written by the user. The only non-standard function needed to run the program is a random number generator, with a flat distribution in the interval]0, 1[. The random number generator is called RLU

(ISEED) and if the user has access to a running version of JETSET 7.3, this function is already implemented. JETNET 2.0 is divided in two parts, one for the feed-forward back-propagation network and another for the self-organizing map. 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. All passing of training data and parameters/switches is done via the common blocks /JNDAT1/ and /JNDAT2/.

6.1. Feed-forward network

6.1.1. Main subroutines

The main routines are JNINIT, JNTRAL, JNTEST, JNDUMP, JNREAD (JNROLD), JNSEFI and JNSTAT. 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 and 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 value.

SUBROUTINE JNINIT
Initializes the network according to the switches in /JNDAT1/ and /JNDAT2/. Weights are given initial random values according to PARJN(4) or WIDL(I). Cannot be called after a call to JMINIT (see below).

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.

The actual updating of the weights is performed every MSTJN(2) call to JNTRAL and the procedure is specified by the switch MSTJN(5).

SUBROUTINE JNTEST
Feeds the input signal 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 previous versions JETNET 1.0 and JETNET 1.1.

SUBROUTINE JNSEFI(ILA, I1, I2, J1, J2, NO)
Switches off ($NO \leq 0$) 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. 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 blocks /JNDAT1/ and /JNDAT2/.

IS = 3 writes out an approximate time factor and the effective number of weights in the net.

6.1.2. Internal subroutines

The 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 inconsistencies.

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. The error is back-propagated to calculate the δ's (see eq. (9)); the updatings for each weight layer.

SUBROUTINE JNSATM
Calculates the saturation measure s of each layer if the transfer function $g(x)$ for that layer is a sigmoid. The saturation measure is given by

$$s = \begin{cases} \sum_j \left(1 - 2 \cdot h_j^2\right) & \text{if } g(x) \in]0, 1[, \\ \sum_j h_j^2 & \text{if } g(x) \in]\text{-}1, 1[\end{cases} \tag{23}$$

and is a measure of the average "information" in each layer (i.e. if the nodes have wandered out into their flat regions). 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 JNTDEC
A "test-deck", which automatically tests JETNET.

6.1.3. 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)
Gives transfer function $g(x)$ of type N with argument X. The transfer function type is determined by switches MSTJN(3) or IGFN(I). (See common block /JNDAT1/ for more details.)

REAL FUNCTION GPJN(X, N)
Gives the derivative $g'(x)$ of $g(x)$.

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

6.2. Self-organizing map

6.2.1. Main subroutines

The main subroutines are JMINIT, JMINWE, JMTRAL, JMTEST, JMDUMP, JMREAD, JMSTAT and JMINDX. The network is defined by setting the switches in /JNDAT1/ and invoking JMINIT. Training is done by putting a pattern into the vector OIN and calling JMTRAL. If the learning vector quantization option is used, a target pattern has to be placed in the vector OUT before JMTRAL is called. The network is tested by reading a pattern into OIN and calling JMTEST, that puts the network response into the vector OUT.

SUBROUTINE JMINIT
Initializes the map according to the switches set in /JNDAT1/ and gives start values to the weights determined by MSTJM(2) and PARJM (4). JMINIT cannot be called after a call to JNINIT (see above).

SUBROUTINE JMINWE(INOD)
Sets the weight vector belonging to network node INOD equal to the current values in vector

OIN. Used to make the initial weight vectors lie inside the "problem area" (see section 4.2).

SUBROUTINE JMTRAL
Takes the pattern stored in vector OIN and trains the network. If learning vector quantization is used, a corresponding output must be stored in the vector OUT before JMTRAL is called. The updating option used is determined by the switch MTSJM(5).

SUBROUTINE JMTEST
Takes the pattern stored in the vector OIN and uses the current values of the weights in the network to produce an output pattern which is stored in the vector OUT.

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

SUBROUTINE JMSTAT(IS)
Writes out information about the network on file number MSTJM(6).
IS = 1 gives a header and number of nodes in each dimension (done automatically when the network is initialized).
IS = 2 gives the switches and parameters used in common block /JNDAT1/.
IS = 3 writes out an approximate time factor and the effective number of weights in the net.

SUBROUTINE JMINDX(INOD, I, J)
If the map topology is two-dimensional, JMINDX is used to transform map coordinates (I,J) to and from the index INOD used in the node vectors OUT and O (see common block /JNINT1/ for details). If INOD = 0, network coordinates (I, J) are returned. Otherwise, if INOD = 0, network coordinates (I, J) are used to calculate INOD, which is returned.

6.2.2. Internal subroutines
The subroutines are presented in the order they are used when training the map.

SUBROUTINE JMSEPA
Sets parameters and switches in common block /JMINT1/. Checks for inconcistencies in switches set in /JNDAT1/.

SUBROUTINE JMNBHD
Specifies the neighbourhood of every node in the map. The result is stored in the vector NBHD. JMNBHD is called every time the neighborhood size MSTJM(9) is changed.

SUBROUTINE JMNORM
Normalizes the weight vectors so that $\|\boldsymbol{\omega}_j\| = 1$ for all units j. Used when the response function $g(x)$ is a sigmoid function and the magnitude of the input vectors is unimportant.

SUBROUTINE JMFEED
Feeds the input in OIN to the network and calculates network response and winner node. The index for the winner node is placed in MXNDJM. If no winner unit is found, JMFEED issues a warning message. JMFEED also calculates the weight updates (see eq. (17)).

SUBROUTINE JMWARN(IWARN)
If a minor error occurs during execution of the program, JMWARN is called and issues a warning message on file number MSTJM(6).

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

6.2.3. Internal functions
REAL FUNCTION GJM(X,N)
Gives response function $g(x)$ of type N with argument X. The type is determined by switch MSTJM(3).

6.3. Common blocks

6.3.1. 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 transfer function used in the net:
$1 \rightarrow g(x) = 1/[1 + \exp(-2x)]$,
$2 \rightarrow g(x) = \tanh(x)$,
$3 \rightarrow g(x) = \exp(x)$ (only used internally for Potts-nodes),
$4 \rightarrow g(x) = x$,
$5 \rightarrow g(x) = 1/[1 + \exp(-2x)]$ (only used internally for entropy error).

MSTJN(4) (D = 0) error measure:
$0 \rightarrow$ summed square error,
$1 \rightarrow$ entropy error,
$\geq 2 \rightarrow$ Kullback error with Potts nodes of dimension MSTJN(4).

MSTJN(5) (D = 0) updating procedure:
$0 \rightarrow$ normal updating,
$1 \rightarrow$ "Manhattan" updating,
$2 \rightarrow$ "Langevin" 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 $\rightarrow$ 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 ($0 \rightarrow$ machine precision) for sigmoid functions (28), thresholds (29) and weights (30).

MSTJN(31) (D = 1) procedure for handeling warnings:
$0 \rightarrow$ no action is taken after a warning,
$1 \rightarrow$ the execution is stopped after the program has experienced MSTJN(32) warnings. In any case only MSTJN(32) warning messages are printed out.

MSTJN(32) (D = 10) maximum number of warning messages to be printed. As described above.

MSTJN(33) (R) code for latest warning issued by the program.

MSTJN(34) (I) number of warnings issued by the program so far.

MSTJN(35–40) no used.

Switches 2, 5, 6, 21, 22, 28, 29 and 30 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 = 0.01) 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.

PARJN(5) (D = 0.0) weight decay parameter ϵ.

PARJN(6) (D = 0.0) width 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).
PARJN(12)	not used.
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 "cut-off".
PARJN(18)	(D = 1.0) scale parameter ω_0 used in pruning.
PARJN(19)	(D = 0.0) target error D used in pruning.
PARJN(20–40)	not used.

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

MSTJM is a vector of switches used to define the self-organizing network used:

MSTJM(1)	(D = 1) number of dimensions in the net (maximum = 2).
MSTJM(2)	(D = 0) symmetry of initial weights: $0 \rightarrow [0, +\omega]$, $1 \rightarrow [-\omega, +\omega]$.
MSTJM(3)	(D = 2) overall response function used in the net: $1 \rightarrow g(x) = 1/[1 + \exp(-2x)]$ (if data is normalized), $2 \rightarrow g(x) = \exp(-x)$.
MSTJM(4)	(D = 1) error measure: $1 \rightarrow$ summed square error.
MSTJM(5)	(D = 0) updating procedure: $0 \rightarrow$ unsupervised clustering and topological ordering, $1 \rightarrow$ learning vector quantization, $2 \rightarrow$ LVQ with neighborhood function.
MSTJM(6)	(D = 6) file number for output statistics.
MSTJM(7)	(D = 0) normalize weights ($1 \rightarrow$ on).
MSTJM(8)	(I) initialization done.
MSTJM(9)	(D = 0) neighborhood size λ: $> 0 \rightarrow$ square neighborhood, $< 0 \rightarrow$ circular neighborhood.
MSTJM(10 + I)	number of nodes in dimension I (I = $0 \rightarrow$ input).
MSTJM(10)	(D = 8)
MSTJM(11)	(D = 10)
MSTJM(12)	(D = 1)
MSTJM(13–20)	not used.

Switches 5, 6, 7 and 8 can be changed at any time by the user.

PARJM is a vector of parameters determining the performance of the self-organizing net:

PARJM(1)	(D = 0.001) learning rate η.
PARJM(2)	(D = 0.0) not used.
PARJM(3)	(D = 0.01) overall inverse network temperature $\beta = 1/T$.
PARJM(4)	(D = 0.5) width ω of initial weights.
PARJM(5–20)	not used.

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

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.

MXNDJM is the index of the winner node in the vector OUT (when using the self-organizing map).

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.

6.3.2. Internal common blocks

COMMON /JNINT1/ O(2000), A(2000), D (2000), T(2000), DT(2000), W(150000), DW (150000), NSELF(150000), NTSELF(2000)

/JNINT1/ contains weight and node vectors for the net. Node vectors are O, A, D, T, DT and NTSELF. Weight vectors are W, DW and NSELF.

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 (see eq. (9)) to node I.
D(I)	current value of the δ_i at node I (see eq. (9)).
T(I)	current value of the threshold θ_i (see eq. (1)) 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. For self-organizing map, NTSELF is equivalent to the vector INDW.
NBHD(I,NB)	contains the indexes to the neighborhood nodes of node I, the first element (NB = 0) tells how many units there are in the neighborhood.
INDW(I)	the index to the weight ω_{i1} belonging to unit I.

COMMON /JNINT2/ M(0 : 10), MVO(10), MMO(10), 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),
MVO(I)	offset index for node vectors – tells the index in the node vectors for the last node in layer (I − 1).
MMO(I)	offset index for weight vectors – tells the index in the weight vectors for the last weight going from layer (I − 1).
NG(I)	transfer function $g(x)$ for layer I.
NL	number of layers except input layer.
IPOTT	switch if Potts units are used in output layer.
ER1, ER2	internal variables used for calculating PARJN(7–9).
SM(I)	internal variable used for calculating saturation measures.

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 negative, weights from equivalent receptive field nodes in the first hidden layer to each node in the second hidden layer are clamped.

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) correspond 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 /JMINT1/ NDIM, ISW(10), *NODES*(0 : *MAXD* + 1), *NBO*

/JMINT1/ contains pointers and switches for the self-organizing map.

NDIM	number of dimensions in the map.
ISW(I)	internal switches.
NODES(I)	number of nodes: I = 0 → input units, I = 1 → first dimension, I = 2 → second dimension, I = 3 → total number of nodes in map.
NBO	last used neighborhood size λ.

7. Restrictions and technical information

JETNET 2.0 includes a "test deck" subroutine called JNTDEC. If you call this subroutine, it will test JETNET automatically.

The maximum number of layers in JETNET is 11, although it is very difficult to find a situation where mode 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 100. 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 150 000. This can be changed by changing the parameter MAXM in the PARAMETER statement in each routine.

The maximum number of dimensions for the self-organizing map is 2. This number cannot be changed.

The program must be loaded together with a function called RLU returning random numbers callable by eg. X = RLU(ISEED), where ISEED is an integer. RLU should return random numbers with a flat distribution within]0, 1[.

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 and GPJN.

References

[1] L. Lönnblad, C. Peterson and T. Rögnvaldsson, Using neural networks to identify jets, Nucl. Phys. B 349 (1991) 675.

[2] D.E. Rumelhart, G.E. Hinton and R.J. Williams, Learning internal representations by error propagation, in: Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Vol. 1 D.E. Rumelhart and J.L. McClelland, eds. (MIT Press, Cambridge, MA 1986).

[3] T. Kohonen, Self organized formation of topologically correct feature maps, Biol. Cybernet. 43 (1982) 59. T. Kohonen, Self-organization and Associative Memory, 3rd ed. (Springer, Berlin, 1990).

[4] L. Lönnblad, C. Peterson and T. Rögnvaldsson, Finding gluon jets with a neural trigger, Phys. Rev. Lett. 65 (1990) 1321.

[5] L. Lönnblad, C. Peterson, H. Pi and T. Rögnvaldsson, Self-organizing networks for extracting jet features, Comput. Phys. Commun. 67 (1991) 193.

[6] A. Weigend, B. Huberman and D. Rumelhart, Predicting the future: a connectionist approach, Int. J. Neural Syst. 3 (1990) 193.
[7] J. MacQueen, Some methods for classification and analysis of multivariate observations, in: Proc. 5th Berkeley Symp. Math. Stat. and Prob., L.M. LeCam and J. Neyman, eds. (Univ. California Press, Berkeley, 1967).
[8] R. Lippman, An introduction to computing with neural nets, IEEE ASSP Mag. 4 (1987).
[9] G. Cybenko, Approxiamtion by Superpositions of a Sigmoidal Function, Tech. Report No. 856, Univ. of Illinois (1989).
[10] C. Peterson and E. Hartman, Explorations of the mean field theory learning algorithm, Neural Networks 2 (1989) 475.
[11] E. Baum and D. Haussler, What size net gives valid generalization?, Neural Comput. 1 (1989) 151.

SAMPLE PROGRAM

```
*** A mirror symmetry problem; training and test Patterns of 6*6 pixels
*** symmetrical around either a horisontal or a vertical lines are assumed
*** generated and stored in 'mirror_train.data' and 'mirror_test.data'
*** respectively.  The output node [DESO(IPAT)] is either 1 or 0 depending
*** upon symmetry.  NRIGHT collects scores in the test phase.


      PARAMETER(MAXPAT=200)
      DIMENSION PAT(MAXPAT,36),DESO(MAXPAT)

      PARAMETER(MAXI=1000,MAXO=1000)

      COMMON /JNDAT1/ MSTJN(40),PARJN(40),MSTJM(20),PARJM(20),
     &                OIN(MAXI),OUT(MAXO),MXNDJM
      COMMON /JNDAT2/ TINV(10),IGFN(10),ETAL(10),WIDL(10),SATM(10)

C...set number of layers:
      MSTJN(1)=3
C...set number of nodes in input layer:
      MSTJN(10)=36
C...set number of nodes in hidden layer:
      MSTJN(11)=10
C...set number of nodes in output layer:
      MSTJN(12)=1
C...set learning rate
      PARJN(1)=0.2
C...all other parameters and switches keeps their default value


C...initialize the net:
      CALL JNINIT

C...loop over training cycles:
      DO 1000 ICYCLE=1,80

C...get training patterns
        DO 10 IPAT=1,MAXPAT
          OPEN(8,FILE='mirror_train.data',STATUS='OLD')
          READ(8,*) DESO(IPAT)
          READ(8,*) (PAT(IPAT,i),i=1,36)
10      CONTINUE
        CLOSE(8)

        NRIGHT=0

C...loop over all patterns:
        DO 200 IPAT=1,MAXPAT
C...put pattern IPAT in OIN
          DO 100 I=1,36
             OIN(I)=PAT(IPAT,I)
100       CONTINUE
C...put desired output pattern in OUT
          OUT(1)=DESO(IPAT)
C...train the net
          CALL JNTRAL
200     CONTINUE

        OPEN(8,FILE='WEIGHTS',STATUS='UNKNOWN')
C...dump all information of the net
        CALL JNDUMP(8)
        CLOSE(8)

C...get test patterns
        DO 110 IPAT=1,MAXPAT
          OPEN(8,FILE='mirror_test.data',STATUS='OLD')
          READ(8,*) DESO(IPAT)
          READ(8,*) (PAT(IPAT,i),i=1,36)
110     CONTINUE
        CLOSE(8)

C...loop over all patterns:
        DO 400 IPAT=1,MAXPAT
C...put pattern IPAT in OIN
          DO 300 I=1,36
             OIN(I)=PAT(IPAT,I)
300       CONTINUE
C...test the net
          CALL JNTEST
          IF(DESO(IPAT).EQ.1.0.AND.OUT(1).GT.0.5) NRIGHT=NRIGHT+1
          IF(DESO(IPAT).EQ.0.0.AND.OUT(1).LT.0.5) NRIGHT=NRIGHT+1
400     CONTINUE

        PTEST=FLOAT(NRIGHT)/FLOAT(MAXPAT)
        WRITE(*,*) ICYCLE,PTEST

1000  CONTINUE

      END
```