Modified from OPAL Technical Note TN501
May 21th, 2002
Used With the permission of R. Hawkings.

Tools for training and using neural networks

Original author: Richard Hawkings (CERN)
Adapted to Linux and SNO: Alain Bellerive (Carleton University)

Abstract

Neural networks are a powerful technique for event classification in particle physics. Tools are described which ease the training and implementation of neural networks based on the JETNET package using HBOOK ntuples. Some examples relating to OPAL b-physics are described. The tools and examples are available from the SNO-Carleton neural network web page.

1 Introduction

Neural networks are a powerful technique for event classification in particle physics and are becoming ever more widely used for pattern recognition. Many of these networks are based on the JETNET3 package [1], which provides a convenient FORTRAN library for the training of feed-forward networks. For typical applications, some effort is still required to interface this program to the user's data, and to implement the resulting network in a standalone program, for example for eventual inclusion in a standard code package or event selection. This note describes two tools, NETTRA and NETGEN, which have been written to help in this process. They have been used to develop the b-tagging networks described in the OPAL R_b publication [2], but are sufficiently general to be useful in other contexts.

The application of these programs will generally be as follows:

- A Monte Carlo dataset is created, containing several classes of events which must be separated as cleanly as possible, and several candidate separation variables which could be used as input to the neural net. This is typically, though not necessarily, in the form of an HBOOK ntuple file.

- The NETTRA program is used to train neural networks based on the various available inputs. The performance of various network configurations and sets of input variables are compared, and an optimum neural network is trained. This is output from NETTRA as a JETNET format file, which can be subsequently re-used using the JETNET program elements.
- From the JETNET file, the NETGEN program is used to create a FORTRAN subroutine to calculate the neural network output given the appropriate inputs. This can be used in a standalone program, as part of further selections, or become part of a standard library, without the need to load the JETNET software.

The remainder of this note describes the procedure in more detail. Section 2 gives an overview of the NETTRA program, describing the subroutines the user must provide, the input datacards controlling the program operation, and the output histograms and data file. Section 3 gives some concrete examples of the use of NETTRA. Section 4 describes the NETGEN program, and section 5 gives some general hints and tips on the use of neural networks.

The code for the programs, and the examples, can be obtained from a sub-page of the SNO-Carleton neural nets web page, at:

`www.physics.carleton.ca/research/sno/anal/tools/nnopal/index.html`

The general SNO neural networks page is also accessible from the Carleton physics tools page, and gives a lot more information and advice about neural networks from several people who have used them in the past. It is accessible at:

`www.physics.carleton.ca/research/sno/anal/snocarleton.html`

The programs have been tested with JETNET3.4 (which is also available from the web page), but may also work with more older versions of JETNET, depending on the common block definitions.

2 The NETTRA program—description

The NETTRA program is written as a main FORTRAN program with several subroutines. It must be linked with the JETNET code and several subroutines supplied by the user. These routines are responsible for any required initialisation (opening and booking ntuples, defining FFREAD cards), termination, and the supplying of input patterns to the neural network training routines.

NETTRA initialises JETNET itself, and sets sensible defaults for the JETNET parameters, which have been found to work well for typical applications. The user can override any of these parameters in the routines `UNINIT` and `UNSETR`. The program loops over all the input events several times ('training epochs'). In each epoch, part of the sample is used to train the network, and part to test the performance of the training so far. There is no algorithm to tell NETTRA when to stop training—it always performs a set number of epochs. The training and testing information calculated shows when the network has stopped learning and can help the user to decide on a sensible number of training epochs.

2.1 User code elements

The following routines must be supplied by the user:

```
SUBROUTINE UNINIT
```

This routine is called before the standard JETNET initialisation routine `JNINIT`. The user can set JETNET options, define user FFREAD cards and open and book the ntuple containing the network training data.

```
SUBROUTINE UNSETR
```

This routine is called after all standard initialisation is complete, and `JNINIT` has been called. It can be used for printing options set by the user FFREAD cards, or further modification of the JETNET parameters.

```
SUBROUTINE UNPATN(IEVT,NSUB,LSEL,LFIN)
```

Called by NETTRA to request an input pattern. The requested pattern is identified by integer `IEVT`, which represents *e.g.* the ntuple event number. The user must set the variables `NSUB`, `LSEL` and `LFIN` to indicate what is being returned.

In the simplest case, there is one input pattern corresponding to each value of `IEVT`. In this case, the variable `NSUB` should be set to -1 on exit, and the logical variable `LSEL` should be set to `.TRUE.` if the event is to be used for network training or testing, or `.FALSE.` otherwise, allowing cuts to be applied in `UNPATN` on which events are actually to be used. The logical variable `LFIN` should be set to `.TRUE.` if all events have already been processed—*i.e.* if `IEVT` exceeds the number of events available in the ntuple. NETTRA will then finish the current 'epoch' and start by requesting event `IEVT=1` again on the next call, if appropriate.

The routine `UNPATN` should place the neural network input values corresponding to the current event in the array `OIN()`, *i.e.* network input 1 in `OIN(1)`, input 2 in `OIN(2)` and so on. The required output pattern should be placed in the array `OUT()`. In most cases this will only require setting `OIN(1)` to 0.0 or 1.0, depending on whether the event should be classified as background or signal respectively. In applications with multiple outputs, the second and subsequent output values should be placed in `OUT(2)`, `OUT(3)` and so on. The arrays `OIN()` and `OUT()` are defined in the common block `JNCOMM` (jncomm.inc) which is defined in the `nettra.f` file.

More complicated use is possible via the variable `NSUB` (set to -1 above). If the event (identified by `IEVT`) has several sub-events (*e.g.* several array entries in a column-wise ntuple), NETTRA can loop over them separately. This is achieved by setting `NSUB` equal to the number of sub-events for the current event `IEVT`. NETTRA will then call `UNPATS` for each of the `NSUB` sub events, and `UNPATS` is responsible for setting the appropriate input and output patterns in `OIN()` and `OUT()`. The variable `LSEL` is used to accept or skip the entire set of sub-events associated to `IEVT`, and `LFIN` is used to signal the end of the data set as before. `NSUB` may be zero for a particular `IEVT`, in which case `UNPATS` will not be called.

```
      SUBROUTINE UNPATS(ISUB,LSEL)
```

This routine is called to fill the `OIN()` and `OUT()` arrays with the input and output patterns for a particular sub-event `ISUB`. It is only used when 'multiple sub-events' are triggered by setting `NSUB` positive in `UNPATN`. The variable `LSEL` must be set to `.TRUE.` if this particular sub-event is to be used for training or testing the net, as with `UNPATN`.

```
      SUBROUTINE UNFIN
```

This routine is called at the end of network training and testing. It may be used to close the input ntuple, print summary statistics etc.

It should be noted that NETTRA does not require the use of HBOOK ntuples for the storing of input data. Any format can be used, as long as the user routines are written appropriately.

2.2 Input datacards

The following FFREAD datacards are defined by the NETTRA package, and may be used in the input file to control the program execution. All have sensible default values:

CONFIG <nlay> <nnode1> <nnode2> . . . Define the network configuration. The total number of layers is given, followed by the number of nodes in each layer (input layer, each hidden layer and output layer). For example, a 4:6:1 network is specified as `CONFIG 3 4 6 1`. A maximum number of two hidden layers is allowed. The default network configuration is 3:4:1.

CACHE OFF|ON Switches on the memory caching of events (default OFF). By default, on each epoch, NETTRA calls `UNPATN` to read each event. If the memory cache is enabled, events will be saved on epoch 1, and will be read from memory on subsequent epochs, rather than calling `UNPATN` again. This can save time and disk I/O for large ntuples, and when a large fraction of the events are actually rejected by cuts in `UNPATN`—only those accepted will be saved. To enable this feature, a memory buffer is required, which dimensioned to a very small value by default. It can be resized by defining the parameters in the include file, for example in `examp2.f (netch.inc)`:

```
*  parameters for cache size, must correspond to the maximum number
*  of events to be used, and the number of input and output nodes
      PARAMETER (netcp_mxevt=12000,netcp_ninp=6,netcp_nout=1)
```

If too many events are read for the dimensioned buffer size, a warning message is issued and the memory cache feature is automatically disabled. The number of events saved in the cache is printed at the end of epoch 1.

EVENTS <n> The number of valid events that will be used by the program; if zero, all the events will be used. This can be used to run on a small sample for development work (default 0).

FTEST <f> The fraction of events that will be used for testing rather than training the network (default 0.25).

NBINS <n> Define the number of bins used for evaluating the figure of merit for the network performance (default 50, maximum 50).

NIBIN <n> Define the number of bins used for the input histograms, default 60.

LIBIN <low> <up> Define the (REAL) lower and upper limits of the input histograms, default -0.1 to 1.1.

NCLASS <n> Define the number of event classes. Normally this is two for a single output network, and equal to the number of outputs n_{out} for a multiple decision network splitting the events into n_{out} classes (default 2).

NEPOCH <n> Define the number of training epochs (default 10).

NORM OFF|ON Switches on the automatic normalisation of output variables in a multiple output network (default OFF).

MASK OFF|ON OFF|ON ... Allows network inputs to be masked (i.e. set to zero). The mask status (off or on) is given for each input in order. By default, all masks are off. This provides a convenient way of removing the information from one of the inputs to the network without changing the user code, allowing the influence of particular input variables on the network performance to be studied.

In addition to the above, the user may define her own datacards in the `UNINIT` subroutine, and the standard FFREAD cards (e.g. `LIST`) may be used.

2.3 Output and histograms

The NETTRA program produces some output (on standard output), and some HBOOK histograms on the file `nettra.hb`. The output file reports the configuration of the network, and the number of events found in each class (class 1 is background, corresponding to `OUT(1)=0.0`, and class 2 is signal, for which `OUT(1)=1.0`). It then lists the 'figure of merit' (FoM) derived for the training and the test data for each epoch. The figure of merit F measures the separation achieved in the neural network output distribution between the signal and background distributions—values close to zero indicate no separation, whilst one indicates complete separation. The figure of merit is defined by the 'inverse correlation integral':

$$F = \alpha_1 \alpha_2 \int \frac{(f_1(x) - f_2(x))^2}{\alpha_1 f_1(x) + \alpha_2 f_2(x)} \mathrm{d}x \qquad (1)$$

where class 1 contributes a fraction α_1 of the sample with normalised distribution $f_1(x)$, and class 2 contributes $\alpha_2 = 1 - \alpha_1$ with normalised distribution $f_2(x)$ [3].

During training, the figures of merit for both training and test samples should increase, quickly at first, then more slowly as the optimum network is reached. When the required number of epochs (set by the `NEPOCH` datacard) is reached, training stops, and the final figures of merit are printed, together with the individual figures of merit for each input variable. This can be used to give an indication of which input variables are contributing most to the separation.

Finally, several histograms are produced to aid in debugging and evaluating the performance of the net. They are detailed below together with their HBOOK ID numbers.

Figure of merit vs. epoch: for the training (10) and test samples (20).

Signal purity vs. network output: for the training (30) and test (40) samples.

Neural network output for each class: for the training (from 100) and test (from 200) samples.

Neural network inputs: from 300 for class 1 (background), from 400 for class 2 (signal). These histograms include data from both the training and test samples.

At the end of the network training, the network configuration is saved in the output file `nettra.jn`. This file is produced by the JETNET routine `JNDUMP`, and can subsequently be read back into JETNET or used by NETGEN to generate a standalone FORTRAN subroutine.

3 The NETTRA program—examples of use

Two examples are provided on the web page, demonstrating simple applications of the program. Each example consists of a fortran `.f` file containing the user routines, an HBOOK `.hb` ntuple for training, a datacard `.dat` file, and the output and histogram files from running the example. These should be studied in conjunction with the following explanations.

Look in the SNOBOY directory for a the source of the codes for the examples described below (/data/data001/snoboy/tools/nnet).

3.1 Example 1: simple b-tagging

The first example `examp1` shows the training of a network for tagging jet which contains a b quarks, a simplified version of those described in [2]. The input ntuple `examp1.hb` contains the flavour of each event, the number of tracks in the secondary vertex reconstructed using the BTGVFT algorithm, the decay length and error of this vertex, and the decay length and error of the 'reduced' secondary vertex with the most significant track removed. The routine `UNPATN` in example file `examp1.f` uses these quantities to form four input variables: the decay length significance, the decay length itself, the reduced decay length significance and the number of tracks in the secondary vertex. The inputs are transformed to keep them within reasonable ranges *e.g.* between zero and one.

The four input variables are shown for background and signal events in figure 1—these histograms are those of the input distributions produced by NETTRA. The target output is set to zero for light flavour and charm events, and one for b-events. A preselection is applied in `UNPATN` so that only vertices with a multiplicity of at least 3 and a decay length significance of at least 3.0 are used, giving a sample that is already 75 % pure in b-events.

The datacard file `examp1.dat` instructs NETTRA to train a 3 layer net with 4 input nodes, 8 hidden nodes and 1 output node, over 10 epochs with the memory cache enabled. The resulting output histograms are shown in figure 2.

The neural network output plots (figure 2(a) and (b)) show large spikes in the signal at values close to one, whilst the background is more evenly distributed. This shows that the net has managed to isolate a sub-sample of b-events with very high purity, but not a sample of highly pure non-b events (which would give a spike at zero). This corresponds to expectation—some b-hadrons decay very close to the primary vertex making them indistinguishable from light quark events. The output distributions for training and testing events are similar, indicating the network has not 'over learned' the training sample.

Figure 2(c) shows the network figure of merit after each epoch. It can be seen that the network learns quickly and reaches optimum performance after only a few epochs, the performance on training and test samples being similar. Figure 2(d) shows the signal purity at each value of network output—for a properly trained net this should be linear, with the network output representing the signal purity, allowing it to be used for weighting purposes.

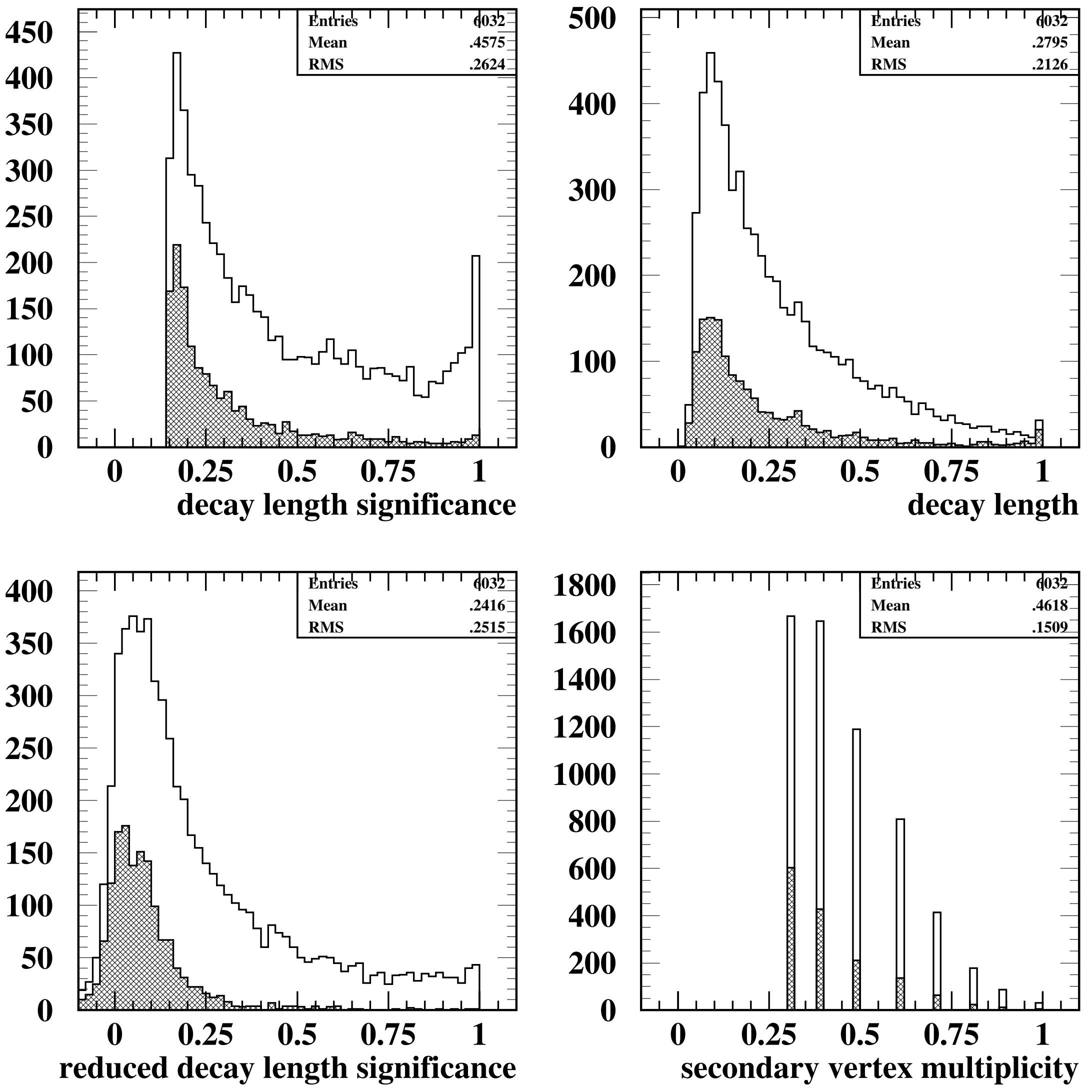

Figure 1: Distributions of the transformed neural network input variables for example 1. Signal events are shown as the open histograms, while background events are shown as hatched histograms.

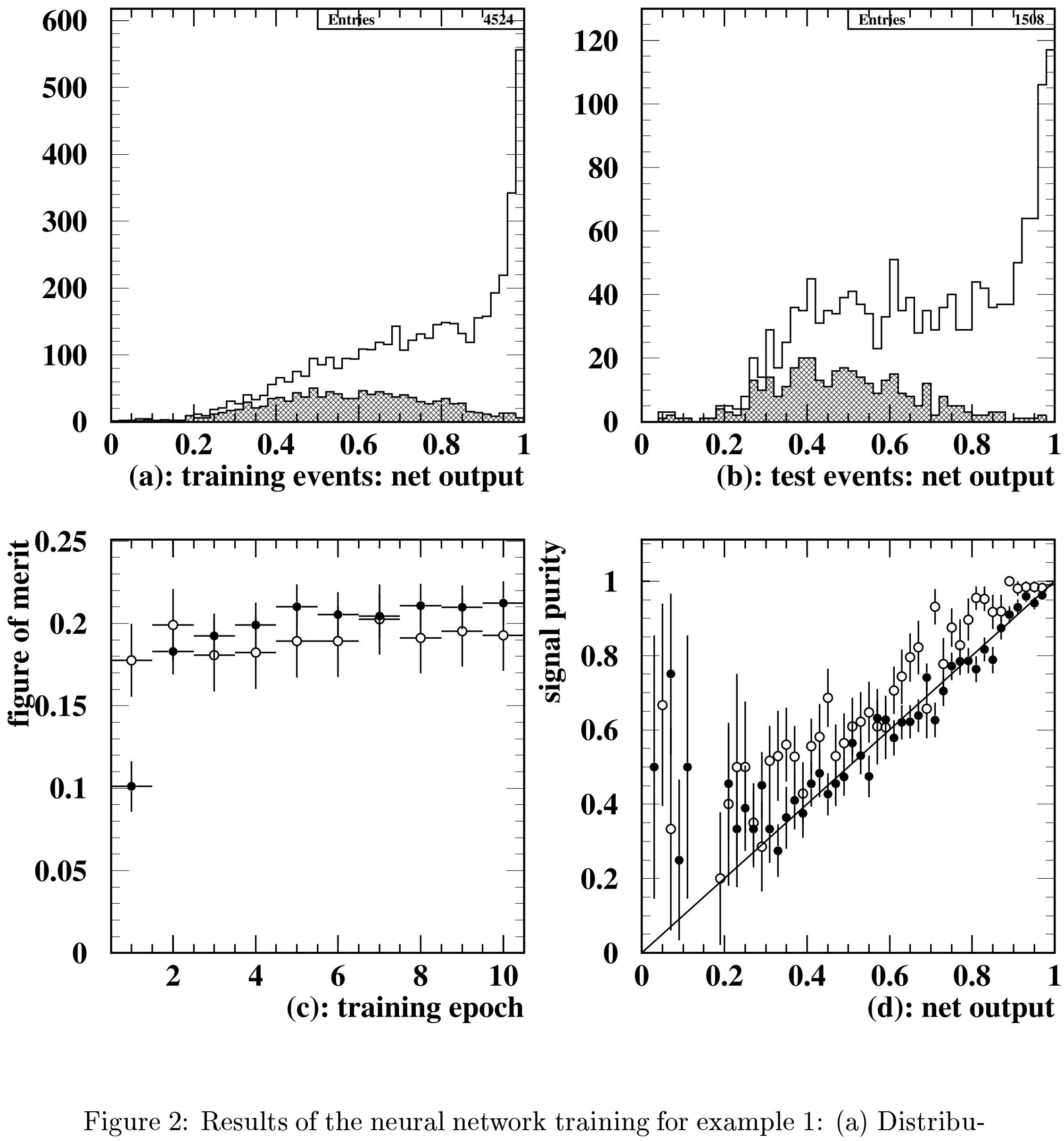

Figure 2: Results of the neural network training for example 1: (a) Distributions of neural network output for signal (open histogram) and background (hatched histogram) events in the training sample; (b) distributions of neural network output for the test sample; (c) neural network figures of merit after each epoch for the training (filled circles) and test (open circles) samples; (d) signal purity *vs.* neural network output for the training (filled circles) and test (open circles) samples.

3.2 Example 2: b-decay track separation

The second example shows the training of a neural network to discriminate between fragmentation and b-decay tracks in jets tagged by significantly separated vertices. This is a simplified version of the network used to derive the mass information for the 'Net5' b-tag described in [2].

The ntuple is similar to that for example 1, but contains an additional block with details of the track momentum p, transverse momentum with respect to the jet p_t, and d_0 and z_0 impact parameter significances with respect to the reconstructed primary and secondary vertices. These are contained in arrays indexed by track number. The routine `UNPATN` selects b-events with a significantly separated vertex, and returns in `NSUB` the number of tracks to be looped over. The input data is then filled by `UNPATS` which is called once for each track in selected events.

The datacard file `examp2.dat` instructs NETTRA to train a 3 layer network with 6 input nodes, 10 hidden nodes and 1 output node, with 30 training epochs. The results are shown in figure 3. It can be seen that in this case more training epochs are required, with significant improvements still being seen after 10 epochs. The linearity of the purity plot is good, indicating that this network output can be used directly as a track weight, for example in vertex charge calculations.

4 Implementing the networks - NETGEN

After training, a `nettra.jn` file is produced containing all the details of the network configuration, thresholds and weights. With this file, it is possible to produce a standalone subroutine to calculate the network output, given the set of inputs. This is done by the NETGEN utility, also available from the web page.

NETGEN is implemented in the form of a shell script, taking three arguments:

```
netgen your_jetnet_file output_file subroutine_name
```

For example, to apply it to the results of example 1:

```
netgen nettra.jn myfile.f MYNET
```

The file `myfile.f` is created with the contents shown in figure 4. The array elements `VNN(1,1)`, `VNN(2,1)` *etc* must be filled with the neural network input variables, perhaps by passing in an additional array. After execution, the variable `OUT` contains the calculated value of the neural network output.

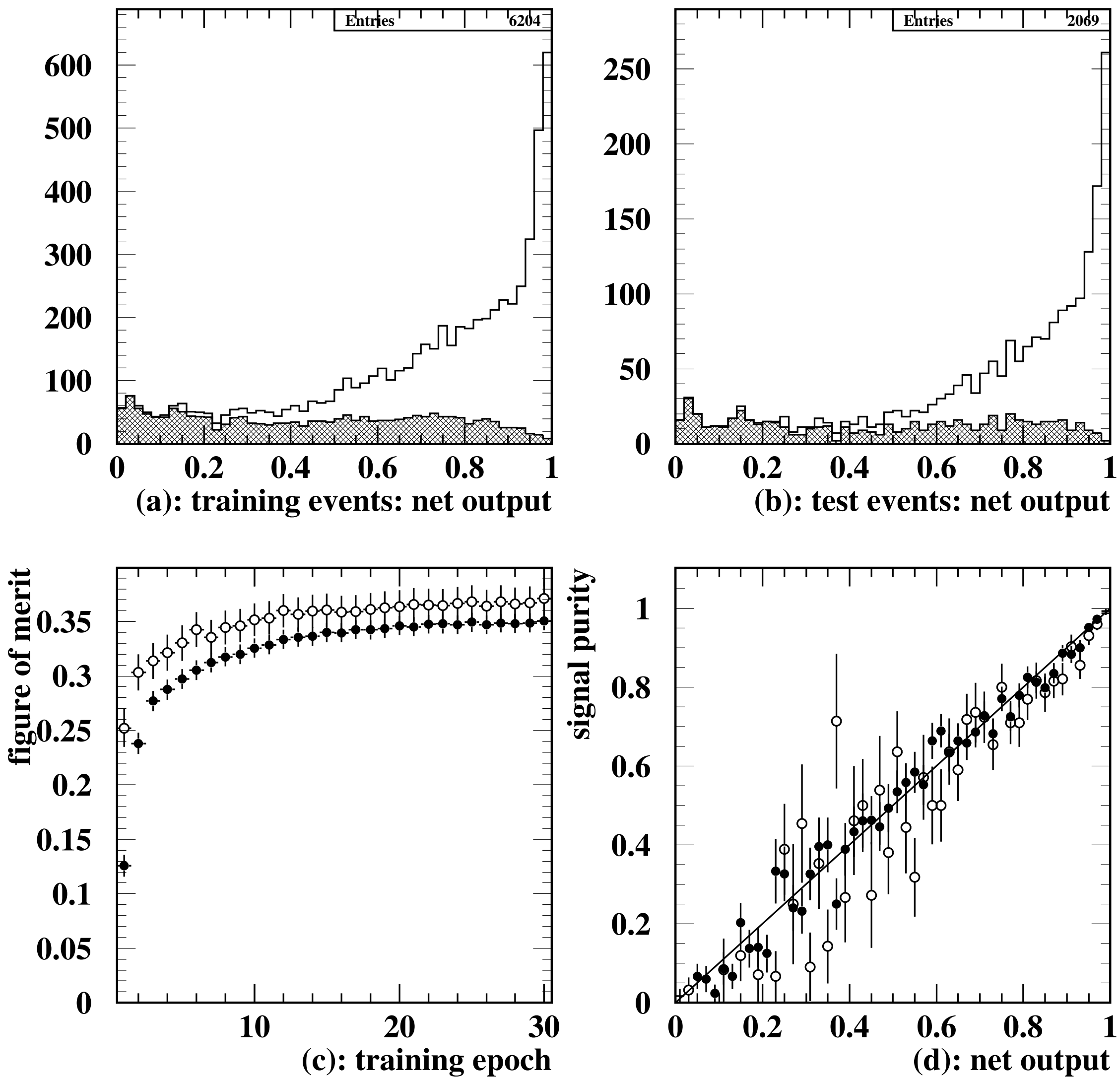

Figure 3: Results of the neural network training for example 2: (a) Distributions of neural network output for signal (open histogram) and background (hatched histogram) events in the training sample; (b) distributions of neural network output for the test sample; (c) neural network figures of merit after each epoch for the training (filled circles) and test (open circles) samples; (d) signal purity *vs.* neural network output for the training (filled circles) and test (open circles) samples.

Since the code to execute the neural network is always the same, and only the `DATA` statements contain the details of the particular network configuration, it is possible to combine several networks in a single subroutine. This is done by the NETGEM (multiple network) program which produces a single subroutine encapsulating several neural networks. Again, the program is presented as a shell script, having the syntax:

```
netgem input_file
```

The input file is an FFREAD datacard file containing a list of the `.jn` files to be converted, *e.g.*

```
NAME 'mymnet'
*CONV 'examp1.jn'
*CONV 'examp2.jn'
STOP
```

where the `NAME` directive sets the name of the subroutine to be generated and the file on which it is written (here `nmymnet.f`), and each `*CONV` card gives the name of a JETNET file containing a network to be implemented. The generated subroutine has an extra argument specifying which neural network is to be used. This NETGEM program was used to construct the routine in which all the OPAL 3D b-tagging networks are implemented.

5 Hints and Tips

The following is a collection of hints and 'rules of thumb' that try to address some of the common questions concerning neural network training. They are based more on practical experience with these tools than theoretical understanding:

What inputs to use: In general, try to choose input variables that are known to have some separation power and that are not too correlated. A small set of well chosen inputs is likely to be better and easier to handle than putting in every possible variable and hoping for the best. It is worth experimenting by removing one variable at a time, retraining the network and comparing the performance, to see if any variables are not contributing anything useful to the separation.

Normalisation of inputs: The network will train more quickly if all the input vary in the approximately the same range, e.g. $[0 \ldots 1]$ or $[-1 \ldots 1]$. Try to avoid a few events with 'pathological values' far from the typical range, such as -9999. Such inputs may cause network nodes to 'saturate' and stop learning. This can sometimes be spotted as spikes in the neural network output distribution.

How many hidden nodes: In theory, twice the number of inputs should be enough for any problem, but often fewer hidden nodes can be used. For simple problems, they can all be put in one hidden layer, or in two for more complex applications. Networks with more than two hidden layers are not supported by NETTRA, though this can be changed easily.

How many events are needed: In general, the more the better, though 1000 per hidden node should be ample. In many cases however, this may not be possible due to finite Monte Carlo statistics. Beware of training a net with very many hidden nodes on few events—this can lead to difficulties with the network training on the properties of individual events rather than general properties of the training sample.

Systematic errors: There is no 'magic' way to evaluate systematic errors for neural network selections (the same is true of likelihoods or other complex selections). Various approaches have been used in the past *e.g.* reweighting or correcting the input distributions and looking at the changes in the output, putting a 'bias' on the the input to the last network node, or training the network with an alternative Monte Carlo dataset. It is obviously easier if the inputs are well described by the Monte Carlo, and it is also important that the correlations are well modelled. But in principle it is no more difficult than with other selection methods—the neural network is just a cut on a complicated derived quantity, rather than cuts on individual inputs in sequence.

Of course, neural networks are not applicable to all problems, some of which may be better suited to likelihood or other selection techniques. But it is hoped that the programs described here will make it easy to evaluate the performance of a neural network approach quickly, and facilitate comparison with other methods.

References

[1] JETNET 3.0—A Versatile Artificial Neural Network Package, Carsten Peterson, Thorsteinn Rögnvaldsson, Leif Lönnblad, LU TP 93-29, CERN-TH.7135/94, submitted to Computer Physics Communications.

[2] A Measurement of R_b using a Double Tagging Method, The OPAL Collaboration G. Abbiendi et al. 31st August 1998, CERN-EP/98-137, Eur. Phys. J. C8 (1999) 217-239.

[3] G. Bahan and R. Barlow, Identification of b jets using neural networks, MAN/HEP/92/1.

```
      SUBROUTINE MYNET(OUT)
*  simulation of JETNET 3.4 neural net with hardwired parameters
*  routine skeleton generated by netgen - please fill in your own inputs
      IMPLICIT NONE
      REAL OUT
*
      INTEGER ILAY,IIN,IIM,IJC,IJT
*
      INTEGER NLAY,NCO,NTH,NINP,MXNODE
      REAL TEMPN
*  parameters of the network size and temperature
      PARAMETER (NLAY=2,NCO=  40,NTH=   9,NINP= 4,MXNODE=  8,
     +  TEMPN= 1.000)
      INTEGER NNODE(NLAY+1)
      REAL CO(NCO),TH(NTH),VN(MXNODE,NLAY+1),VNT
      DATA NNODE/  4,  8,
     +    1/
      DATA CO/
     +  -.8202000E-02, .1280000E+01,-.1436000E+01,-.9774000E+00,
     +  -.8429000E+00, .8610000E+00,-.1051000E+01,-.3021000E+00,
     +  -.2110000E+00, .5764000E+00,-.8135000E+00,-.2461000E+00,
     +  -.2256000E+01, .8981000E-01,-.1612000E+01,-.6649000E-01,
     +   .5220000E+00,-.6550000E+00, .1356000E+01, .6438000E+00,
     +  -.2145000E+00, .3304000E+00, .3209000E+00,-.2701000E+00,
     +   .1282000E+01,-.8455000E+00, .1380000E+01, .7319000E+00,
     +   .8299000E-01,-.1129000E+00, .5326000E+00,-.3305000E+00,
     +  -.1267000E+01,-.9480000E+00,-.5881000E+00,-.1592000E+01,
     +   .9309000E+00,-.5832000E-01, .1349000E+01,
     +   .2706000E+00/
      DATA TH/
     +  -.5968000E+00,-.5151000E+00,-.7587000E+00, .1887000E+00,
     +  -.9674000E+00,-.7764000E+00,-.1282000E+01,-.5788000E+00,
     +   .2865000E+00/
*
*   fill in your own input values into the first layer of the net
      VN(1,1)=  ??
      VN(2,1)=  ??
      VN(3,1)=  ??
*    etc...
*
*   loop over all the layers, calculating the neuron values for that layer
      IJC=1
      IJT=1
      DO ILAY=1,NLAY
*   loop over all the neurons in the layer being evaluated
        DO IIN=1,NNODE(ILAY+1)
*   starting value for neuron is the threshold value
          VNT=TH(IJT)
          IJT=IJT+1
*   now add in the values from the previous layer of neurons
          DO IIM=1,NNODE(ILAY)
            VNT=VNT+CO(IJC)*VN(IIM,ILAY)
*   and increment the coefficient pointer
            IJC=IJC+1
          ENDDO
*   apply the net transfer function
          VN(IIN,ILAY+1)=0.5*(1.0+TANH(VNT/TEMPN))
        ENDDO
      ENDDO
*   now we have the answer, copy to output node
      OUT=VN(1,NLAY+1)
      END
```

Figure 4: FORTRAN subroutine produced by NETGEN to implement the network of example 1.