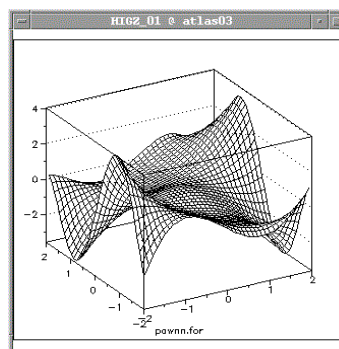
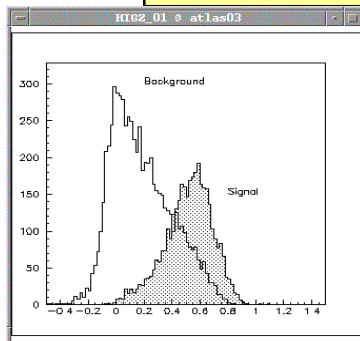
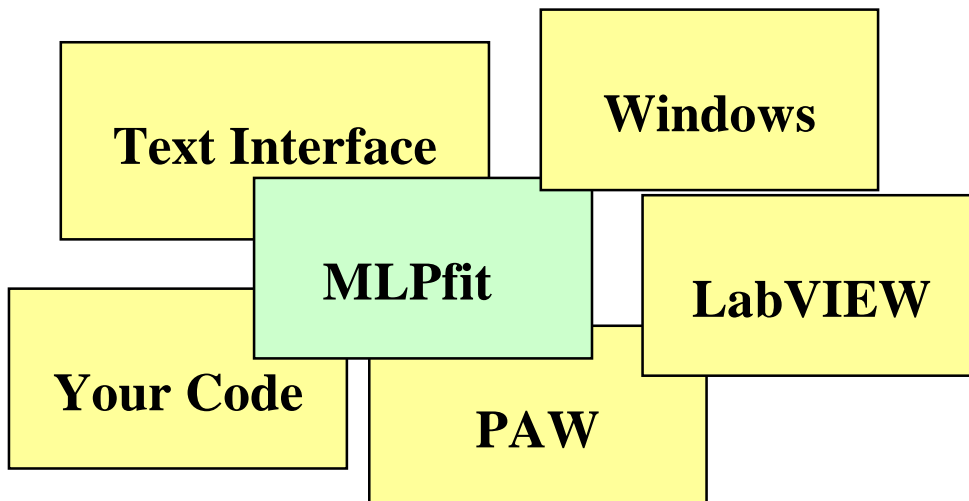


MLPfit : a tool for designing and using Multi-Layer Perceptrons

Version 1.40, January 2000



Jérôme Schwindling & Bruno Mansoulié

DAPNIA/SPP
CE Saclay
91191 Gif sur Yvette CEDEX
FRANCE

e-mails :

jerome@hep.saclay.cea.fr

mansoulie@hep.saclay.cea.fr

COPYRIGHTS	5
USING MLPFIT	5
NON WARRANTY	5
SUPPORT	5
INTRODUCTION	6
NEURAL NETWORKS.....	6
THE PHILOSOPHY OF MLPFIT	6
WHERE TO FIND THE CODE AND DOCUMENTATION	7
PLATFORMS.....	7
CHANGES WITH RESPECT TO VERSION 1.33.....	7
THE MULTI-LAYER PERCEPTRON	9
A DESCRIPTION OF THE MULTI-LAYER PERCEPTRON.....	9
LEARNING METHODS.....	10
A COMPARISON OF THE VARIOUS LEARNING METHODS	12
USING MLPFIT AS A STANDALONE PACKAGE	14
DESCRIPTION OF THE INPUT/OUTPUT FILES.....	14
<i>learn.pat</i>	14
<i>test.pat</i>	17
<i>weights.in, weights.out</i>	17
<i>mlpfun.f</i>	17
<i>mlpfit.his</i>	17
ADDITIONAL POSSIBILITIES	17
<i>FILE filename</i>	17
<i>POND mode [weight1, ..., weight n]</i>	18
<i>NORM n</i>	18
<i>OUTF C</i>	19
HOW TO RUN THE CODE.....	19
HOW TO RECOMPILE THE CODE.....	19
CALLING MLPFIT FUNCTIONS FROM YOUR OWN CODE	20
INTRODUCTION.....	20
THE ROUTINES AVAILABLE.....	20
<i>ierr = mlpsetnet(nlayer,nneurons)</i>	20
<i>ierr = mlpsetlearn(lmethod, eta, decay, epsilon, delta, nreset, tau, rlambda)</i>	20
<i>mlpinitw(mode)</i>	21
<i>mlpprw</i>	21
<i>mlpsavew(filename)</i>	21
<i>mlploadw(filename)</i>	21
<i>mlpprfun(lang)</i>	21
<i>mlpsetnpat(ifile, npat, ipond, ninputs, noutputs)</i>	21
<i>mlpsetpat(ifile, ipat, rin, rans, pond)</i>	22
<i>err = mlpepoch(iepoch)</i>	22
<i>err = mlptest(ifile)</i>	22
<i>mlpcompute(rin, rout)</i>	22
<i>mlpfree</i>	22
<i>mlpnorm</i>	23
USING MLPFIT IN PAW	24
INTRODUCTION.....	24
USING MLPFIT IN LABVIEW	25
THE BASIC INTERFACE VI'S.....	25
<i>SetNet.vi</i>	25
<i>InitWeights.vi</i>	25

<i>SetLearn.vi</i>	25
<i>SetNPat.vi</i>	26
<i>SetPat.vi</i>	26
<i>Epoch.vi</i>	26
<i>NNout.vi</i>	26
<i>SaveWeights.vi</i>	26
<i>LoadWeights.vi</i>	26
<i>Reset.vi</i>	26
THE HIGHER LEVEL VI'S	26
<i>Create.vi</i>	26
<i>Train.vi</i>	26
EXAMPLE	26
TROUBLE SHOOTING	28
USING THE WINDOWS INTERFACE.....	29
INTRODUCTION.....	29

Copyrights

Using MLPfit

MLPfit is free for any academic (research, education...) usage or for your private applications. It is not allowed to sell the results obtained with MLPfit, or to include MLPfit in any free or commercial software without authorization. Legal copyright rules are being worked on.

Non warranty

There is no guarantee of correctness of the results provided by MLPfit. The authors of MLPfit are not responsible for the direct or indirect consequences of using the package.

Support

There is no guarantee for any support of MLPfit in the future. However, the authors will do their best to provide help, correct bugs and add new features.

Introduction

Neural Networks

Neural Networks appeared in the 1940's, but their usage as a powerful processing tool has increased a lot in the last 10 years, with the development of personal computers. They are now very widely used, in several research or commercial fields, for example :

- in medicine, for image analysis or help to diagnosis
- in meteorology, for predictions
- in industry, for automatic process control, for quality checks by image processing or for optimization of resources allocation
- in particle physics, mainly for classification tasks (particle identification, rare event selection, search for new physics)

Although several types of Neural Networks exist, more than 50% of the applications are using the so-called **Multi-Layer Perceptron (MLP)** network, which is both simple and very well known. It will be briefly described in the next chapter.

The philosophy of MLPfit

MLPfit is a modular tool for designing and using Multi-Layer perceptron. It implements powerful learning methods. The code is simple to use and has been designed for an easy implementation of additional features and for an easy interfacing to additional packages. For example, there are currently five ways to use MLPfit :

- as a **standalone package**. The user defines the network, the learning parameters and the examples in a file (`learn.pat`) and MLPfit provides as an output the Neural Network function.
- by calling the MLPfit routines in **your own code** (C or fortran)
- by using MLPfit in a **modified version of PAW**
- through an **interface with LabVIEW**
- through an **interface to Windows**

These five methods are described in the following chapters, after a description of the learning methods. The various interfaces are mostly independent so, if you are interested only in one interface, you can read only the corresponding chapter.

Where to find the code and documentation

The code can be found at CERN by afs at the following location :

`/afs/cern.ch/user/s/schwind/public/MLPfit_1.40`

The following subdirectories contain :

- `doc` : documentation and examples of applications
- `code` : executables and libraries
- `yourcode` : examples of how to use the MLPfit routines from your own code
- `examples` : examples of how to use MLPfit as a standalone package
- `labview` : the code for the interface to LabVIEW

All this structure is also available on www at the url :

<http://home.cern.ch/~schwind/MLPfit.html>

Platforms

Executables for the standalone running mode and libraries for using MLPfit in your own code are currently available on the following platforms :

- Unix workstations : HP-UX 10-20, Dec OSF 4.1, Sun OS 5.6, IBM AIX 4.1, Silicon Graphics Irix 6.4
- PC : DOS, Windows, Linux 5.1

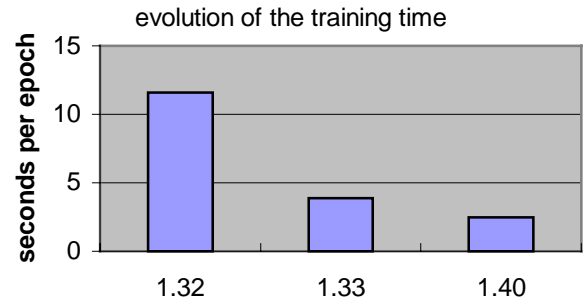
Changes with respect to version 1.33

The following changes have been introduced in version 1.40 :

- There is now no limitation on the number of layers, the number of neurons per layer or the number of examples, in the limit of the available memory
- It is possible to print statistics about the input variables and to normalize them to mean = 0, RMS = 1
- The transfer functions are now always sigmoids for the hidden neurons and linear for the output neuron. It is no longer possible to change them (with the TFUN card). This simplification allows to make the training

much faster and is probably not a limitation for 95% of the applications.

- Training is 40% faster than with MLPfit 1.33, for all learning methods. The figure in the margin shows the evolution of the time per learning epoch on a problem with 68900 learning examples, 15000 test examples, using a 14-40-1 network and the stochastic learning method. MLPfit is now a factor 3 to 5 faster than other Neural Network packages.



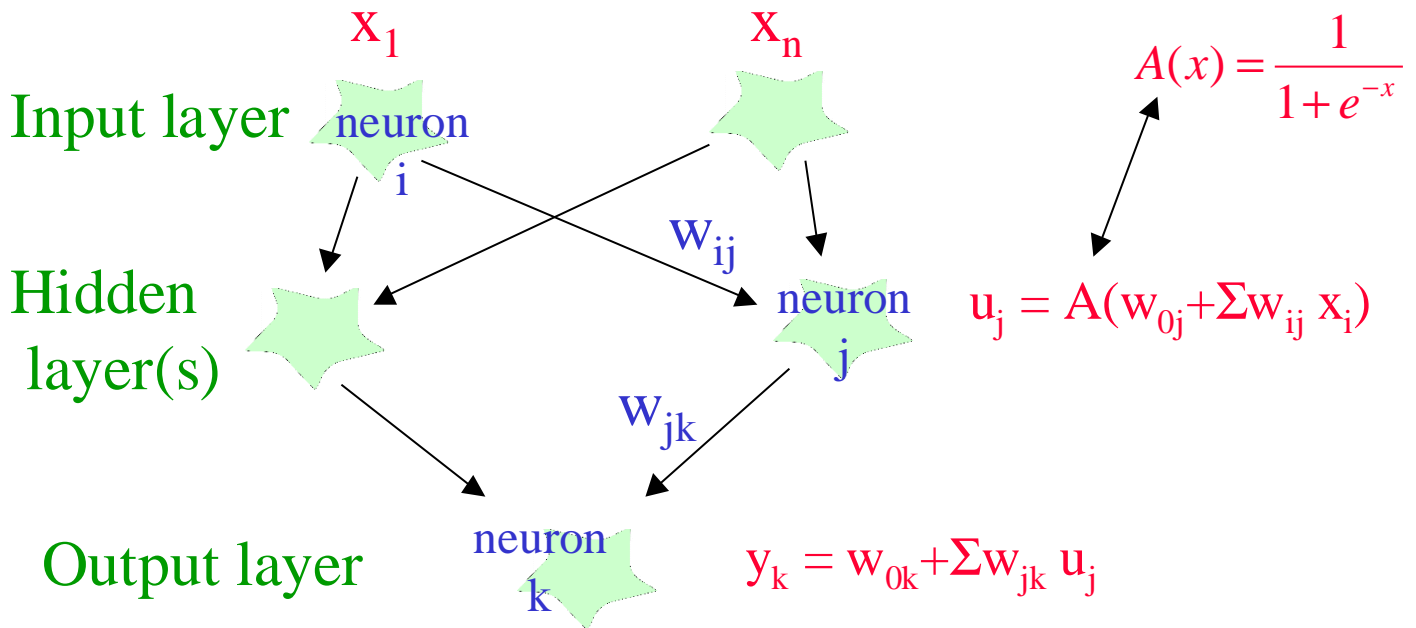
- Memory allocation is now better protected, and the program should exit with a meaningful error message if the available memory is too small.
- In the standalone interface, the NLAY card is no longer needed.
- The following interface functions have changed :
 - `mlpsavew` and `mlploadw` not take as arguments a file name, and return an error code if the file could not be opened
 - `mlpsetnpat` now takes as additional arguments the number of inputs and the number of outputs
- The interface to LabVIEW has updated and improved graphically.

The Multi-Layer Perceptron

This chapter contains a brief description of the Multi-Layer Perceptron and of the learning methods available in MLPfit.

A description of the Multi-Layer Perceptron

The Multi-Layer Perceptron is a simple feed-forward network with the following structure :



The input neurons receive the inputs and forward them to the hidden layer. The neuron j of the hidden or output layer first computes a linear combination of the outputs y_i of the neurons of the previous layer, with a bias, according to the formula :

$$x_j = w_{0j} + \sum_i w_{ij} y_i$$

The output y_j of neuron j is then a function F of its input x_j . The function F , called the transfer function, has a code in MLPfit, which can be :

0 : $y_j = 0$. The neuron is not activated.

1 : $y_j = x_j$. The neuron is linear.

2 : $y_j = \frac{1}{1 + e^{-x_j}}$. This function is called the sigmoid function.

By default, in MLPfit, the transfer function of the output neuron(s) is linear, whereas the transfer function of the hidden neurons is the sigmoid function. Thus, a multi-layer perceptron with one hidden layer computes a linear combination of sigmoids. This is useful because of 2 theorems :

1. A linear combination of sigmoids can approximate any continuous function of one variable or more ¹
2. Trained with a desired answer of 1 for the signal and 0 for the background, the approximated function of the input vector \vec{x} is the probability of signal knowing \vec{x} ².

These two theorems are the basic mathematical foundations of the multilayer perceptron capabilities for function approximation and classification tasks.

Learning methods

The weights w_{ij} are set initially to random numbers between -0.5 and 0.5 . By comparing the MLP output o_p to the desired answer t_p on a set of examples p , they can be updated to minimize $E = \sum_p e_p$ where $e_p = \frac{1}{2} \omega_p (o_p - t_p)^2$. ω_p is an event by event weight which is usually set to 1.

In all the learning methods available in MLPfit, one needs to compute the first order derivatives $\partial e_p / \partial w_{ij}$ (or $\partial E_p / \partial w_{ij} = \sum_p \partial e_p / \partial w_{ij}$), with respect to the weights w_{ij} . This computation is called « back-propagation of the errors ». The following methods can then be used to update the weights :

1. *Stochastic minimization* : this is the traditional learning method, still often wrongly called « (online) standard backpropagation ». It is the Robbins-Monro stochastic approximation method³ applied to Multi-Layer Perceptrons. The weights are updated after each example according to the formula :

$$w_{ij}(t+1) = w_{ij}(t) + \Delta w_{ij}(t)$$

with

¹ For example : K.Hornik et al., *Multilayer Feedforward Networks are Universal Approximators*, Neural Networks, Vol. 2, pp 359-366 (1989)

² For example : D.W.Ruck et al., *The Multilayer Perceptron as an Approximation to a Bayes Optimal Discriminant Function*, IEEE Transactions on Neural Networks, Vol. 1, n° 4, pp296-298 (1990)

³ H.Robbins and S.Monro, *A Stochastic Approximation Method*, Annals of Math. Stat. 22 (1951), p. 400

$$\Delta w_{ij}(t) = -\eta \left(\frac{\partial e_p}{\partial w_{ij}} + \delta \right) + \varepsilon \Delta w_{ij}(t-1)$$

η is called the *learning parameter*. It should decrease while learning to ensure convergence. The term after ε is the momentum term. Fahlman⁴ proposes to add a small value δ to the derivatives to avoid problems when they are close to 0.

2. *Steepest descent with fixed step size*. This method is often called the « offline » or « batch » backpropagation algorithm. It is the same as the stochastic minimization method, but the weights are updated after considering all examples, with the total derivatives $\partial E / \partial w_{ij}$. A loop over all examples is called an *epoch*.
3. *Methods with line search*. These methods are taken from the (mathematical) theory of unconstrained minimization⁵, with the function to minimize being $E(w_{ij}) = E(\vec{w})$. They all work in the following way :

- a) compute a direction \vec{s}_t from the gradient ∇E
- b) find α_m which minimizes $E(\vec{w}_t + \alpha \vec{s}_t)$. This part of the algorithm is called the *Line Search*.
- c) set $\vec{w}_{t+1} = \vec{w}_t + \alpha_m \vec{s}_t$
- d) goto a)

The various algorithms differ by the way step a) is done. The simplest one is the *steepest descent algorithm*, where $\vec{s} = -\nabla E$. A more powerful method is the conjugate gradients algorithm. Two versions of the conjugate gradient algorithm are available (Fletcher-Reeves or Polak-Ribiere updating formulas). The Broyden, Fletcher, Goldfarb, Shanno (BFGS) method is yet another method, which implies $N_{weights} \times N_{weights}$ matrix computations, but seems more powerful than the conjugate gradient methods at least for $N_{weights} < 300$. In all these algorithms, the search direction \vec{s} is reset to the steepest descent direction from time to time.

4. *Hybrid method*. For a given set w_{NL} of input to hidden (or hidden to hidden) weights, the set w_L^* of hidden to output weights which minimizes E can be determined by solving a linear system of equations⁶ when the output neuron is linear (and because the function E is quadratic). At some learning steps, the weights may become too large when solving the linear system. To overcome this problem, a regularisation term is added to the error E which

⁴ S.E.Fahlman, *An Empirical Study of Learning Speed in Back-Propagation Networks*, CMU-CS-88-162 (1988)

⁵ R.Fletcher, *Practical Methods of Optimization*, second edition, Wiley (1987)

⁶ The Linear Least Square problem is solved using the `dgels` routine from the LAPACK library. The LAPACK library is available at <http://www.netlib.org/lapack/>. See E.Anderson et al., *LAPACK Users' Guide*, 2nd edition, Society for Industrial and Applied Mathematics (1995).

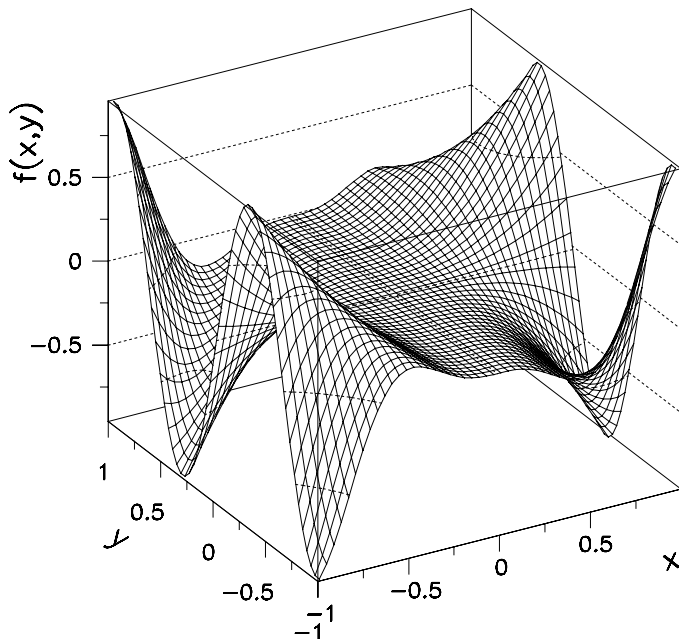
becomes

$$E' = E \left(1 + \lambda \frac{\|\vec{w}\|^2}{\|\vec{w}_{\max}\|^2} \right).$$

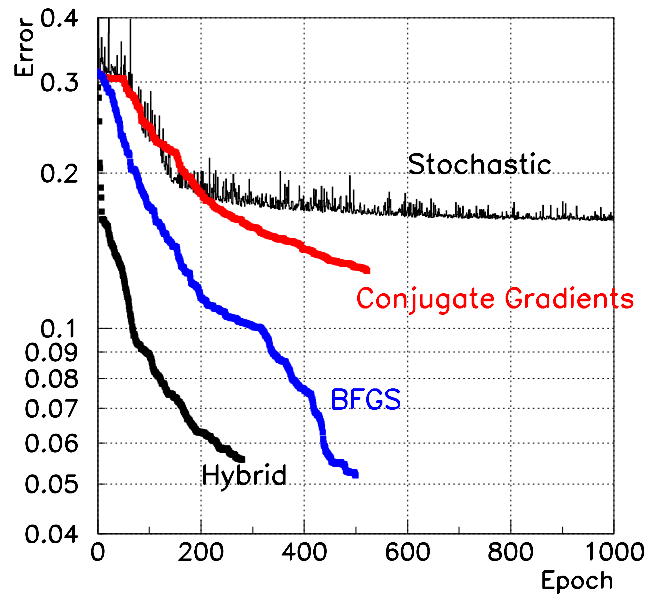
The BFGS method is then used to minimize $E'(w_{NL}, w_L^*(w_{NL}))$.

A comparison of the various learning methods

The various learning methods are compared on the problem of fitting the function $x^2 \sin(5xy)$ from 1681 examples taken on a regular grid in $[-1,1]^2$ by a 2-10-1 network. This function is shown in the following figure :

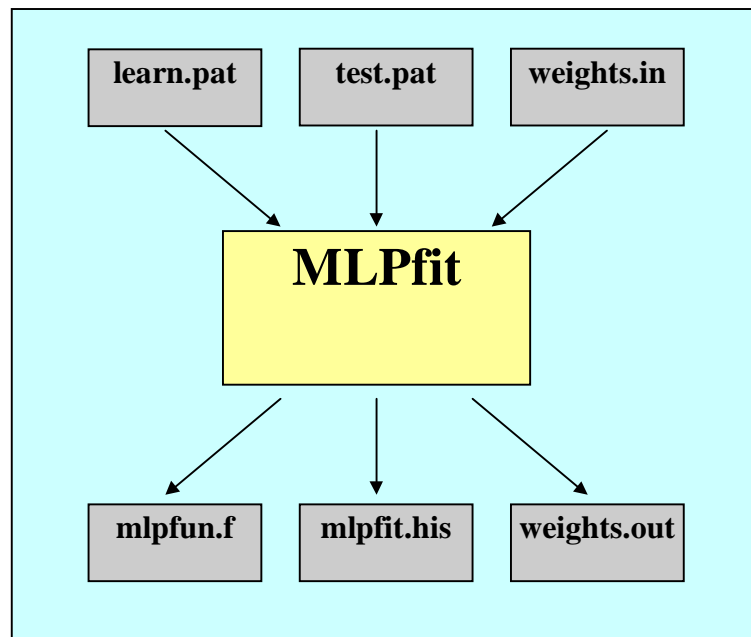


The learning curves for the various minimization algorithms are shown on the next figure. All curves correspond to the same total CPU. The conjugate gradient method uses the Fletcher-Reeves updating formula. At least for problems of this size (~2000 examples and ~30 weights), the BFGS and hybrid methods lead to the best results.



Using MLPfit as a standalone package

MLPfit can be used as a standalone package which reads ASCII input files and produces ASCII output files and a PAW histogram file, as described in the following diagram :



Description of the input/output files

learn.pat

The file `learn.pat` is an ASCII file describing the neural network structure, the learning method, the learning parameters, the running options and contains the examples for training the network. An example of `learn.pat` file is given below :

```
# debug
DEBU 1
# binning of histograms
DBIN 1
# frequency to write the weights in weights.out
OUTW 50
# print statistics
STAT 1
# normalize the inputs
NORM 1

# number of neurons in each layer
```

```

NNEU 2,10,1
# random weights -> 0, read weights in weights.in -> 1
RDWT 0
# learning method
LMET 7
# regularisation
LAMB 1.
# n reset
NRES 1000
# tau
LTAU 3.0
# learning parameter
LPAR 0.1
DCAY 1.0
# momentum factor
MOME 0.0
# flat spot elimination factor
FSPO 0.0
# number of epochs
NEPO 1000
# examples
NPAT 1681
NINP 2
NOUT 1
-1.0 -1.0
-.958924
-1.0 -.95
-.999293
-1.0 -.9
-.97753
-1.0 -.85
-.894989
-1.0 -.8
-.756802
.
.

```

Lines starting with # are comments. The following cards can be used :

- DEBU gives the debug level : 0 prints errors every DBIN epochs (default)
1 prints the errors after each epoch
2prints also the weights after each epoch
- DBIN. Histograms are filled every DBIN epochs. Also defines the frequency to print the errors when DEBU is 0
- OUTW defines the frequency to (over)write the weights in weights.out. By default, it is set to 100 epochs. The weights are always written out at the end of the job.
- STAT defines if statistics about the input variables must be printed (STAT 1) or not (STAT 0).
- NORM 1 allows to normalize the input variables to mean = 0, RMS = 1, as it is sometimes recommended. The normalization factors are

included in the output functions.

- NNEU defines the number of neurons in each layer
- RDWT : choice between random weights (RDWT 0) or read from file weights.in (RDWT 1)
- LMET : **learning method**
 - 1 : *stochastic minimization*. In this case, the following parameters are used : LPAR = learning parameter η , DCAY decay ($\eta = \eta * DCAY^{\text{iepoche}}$), MOME = weight ε of momentum term, FSPO = flat spot elimination factor δ .
 - 2 : *steepest descent with fixed step length* (« batch »learning). The same parameters are used as by method 1.
 - 3 : *steepest descent with Line Search*. The precision of the Line Search is dictated by the parameter LTAU. Lower LTAU = higher precision = slower search, higher value = lower precision = faster search. A value LTAU = 3 seems reasonable.
 - 4 : *conjugate gradients with the Polak-Ribiere updating formula*. The search direction is reset to steepest descent every NRES epochs. NRES > number of weights should be used. This method also uses the LTAU parameter.
 - 5 : *conjugate gradients with the Fletcher-Reeves updating formula*. Same parameters as method 4.
 - 6 : *BFGS method*. Same parameters as for the conjugate gradients methods.
 - 7 : *hybrid method*. Same parameters as the conjugate gradients or BFGS methods, with in addition LAMB = importance of the regularisation term.
- NEPO : number of learning epochs.
- NPAT : number of examples. Should be smaller or equal to the number of examples in the file.
- NINP : number of inputs. Should be smaller or equal to the number of columns in the input lines of the examples. If NINP is lower than the number of columns, only the NINP first are used. NINP should be equal to the number of neurons of the input layer.
- NOUT : number of outputs. Same remarks as for NINP.

The examples then follow, with the inputs on one line and the output(s) on the next line.

test.pat

The structure of test.pat is the same as for learn.pat, but the only mandatory cards are NPAT, NINP, NOUT and the examples. If other cards are present, they overwrite the ones in learn.pat.

weights.in, weights.out

These ASCII files contain the weights to start with or the weights written out. The first line is the number of epochs after which the weights have been obtained (only useful in weights.out, this information is not used when reading weights back). The weights are written one per line, starting with the bias of the first neuron of the first hidden layer and ending with the last weight to the last neuron of the output layer.

mlpfun.f

This file contains the final MLP function in fortran. It is also possible to write out a C function.

mlpfit.his

This file contains PAW histograms showing the learning curves on the learning file (histogram 100) and on the test file (histogram 200).

Additional possibilities

The following commands can be added in the learn.pat file.

FILE filename

MLPfit reads the given file. This option can be used to separate the datacards from the examples themselves. One can, for example, end the learn.pat file with the following lines :

```
# number of epochs  
NEPO 1000  
FILE examples.pat
```

And put the following in examples.pat :

```
# examples  
NPAT 1681  
NINP 2  
NOUT 1  
-1.0 -1.0  
-.958924  
-1.0 -.95  
-.999293  
-1.0 -.9  
-.97753  
-1.0 -.85  
-.894989  
-1.0 -.8  
-.756802
```

POND mode [weight1, ..., weight n]

It is possible to weight the events in the error function by a factor ω_p . By default, no weight is applied (POND 0). One can weight each event by using POND 1. The weights are then set to the values written after the answers, on the same line, as for example :

```
# examples
POND 1
NPAT 1681
NINP 2
NOUT 1
-1.0 -1.0
-.958924      0.1
-1.0 -.95
-.999293      0.3
-1.0 -.9
-.97753        0.2
-1.0 -.85
-.894989       0.5
-1.0 -.8
-.756802       1.0
.
```

One can also weight group of events by using POND 2 weight1 weightn, as for example :

```
# examples
POND 2 0.1 0.2 0.3
NPAT 1681
NINP 2
NOUT 1
-1.0 -1.0
-.958924      1.
-1.0 -.95
-.999293      2.
-1.0 -.9
-.97753        2.
-1.0 -.85
-.894989       3.
-1.0 -.8
-.756802       3.
.
```

The number which follows the answer(s) is then the group number, and the event is assigned the corresponding weight.

NORM n

If, obviously, the range of the input quantities should be compatible with the range of the initial weights (in the sense that their product should not exceed values which can be safely computed in the exponential), some people claim that having inputs all normalized to the same average and same RMS lead to a smoother learning. It is possible to automatically normalize the inputs to an average value of sigma and a width or 1 by using the datacard NORM 1. The normalization is written out in the output function mlpfun.f. The inputs nof the examples are

physically modified in memory before learning starts.

OUTF C

To write out the MLP function in the C language. By default, a fortran function is written (equivalent to OUTF F).

How to run the code

To run the code, simply define `learn.pat`, `test.pat` and run the executable.

How to recompile the code

Please let us know if you need to recompile the code.

Calling MLPfit functions from your own code

Introduction

For some applications, it might be simpler to call the MLPfit routines directly from a user's code. For example, this allows to directly define the examples, to use the MLP function in the same code or, for example, to loop automatically on network sizes.

A set of interface routines are available in MLP_inter.c. They can be called from fortran or C programs. They observe the following rules :

1. Their name start with `mlp`
2. Their name uses only lowercase characters
3. Their name end with `_` for being compatible with usual fortran compilation conventions. To call them from fortran, use the name without `_`. To call them from C, use the name with `_`
4. All arguments are passed by addresses

The routines are described in the next section. An exemple of how to use them is available in [MLPfit_1.40/yourcode](#).

The routines available

The definition of the routines follow the fortran conventions. The explanations for the parameters can be better understood by reading the previous chapter.

`ierr = mlpsetnet(nlayer,nneurons)`

To set the Neural Net structure.

Inputs : integer `nlayer` = number of layers
 integer `nneur(nlayer)` = number of neurons in each layer

Return value (integer): error code = 0 : no error
 2 : `nlayer < 2`
 -111 : not enough memory

`ierr = mlpsetlearn(lmethod, eta, decay, epsilon, delta, nreset, tau, rlambda)`

To set the [learning method and the learning parameters](#). It also initializes memory

for learning, and thus needs to be called after `mlpsetnet`.

Inputs : integer lmethod = learning method (see previous chapter for the possible codes)
 real eta = learning parameter
 real decay = eta decay rate
 real epsilon = momentum term
 real delta = flat spot elimination factor
 int reset = frequency to reset the search direction to steepest descent
 real tau = tau for line search
 real rlambda = regularisation term

Return value (integer) : error code = 0 : no error
 1 : method > 7
 -111 : not enough memory

mlpinitw(mode)

To initialize the weights to random numbers. As it needs to know the number of weights, this routine should be called after `mlpsetnet`.

Input : integer mode = 0 : weights are fully random, several successive calls to this routine with mode = 0 lead to different weights
 = 1 : several successive calls to this routine with mode = 1 lead to the same weights.

mlpprw

To print the weights on the screen.

mlpsavew(filename)

To write the weights in file `filename`.

Input : char *filename : name of the output file

Return value (integer) : error code = -1 if file could not be opened

mlploadw(filename)

To read the weights from file `filename`.

Input : char *filename : name of the input file

Return value (integer) : error code = -1 if file could not be opened

mlpprfun(lang)

To write the MLP function in a file.

Inputs : integer lang = 1 : fortran function written in file `mlpfun.f`
 2 : C function written in file `mlpfun.c`

mlpsetnpat(ifile, npat, ipond, ninputs, noutputs)

To set the number of examples.

memory must first be released by a call to `mlpfree`.

`mlpnorm`

Can be called to normalize the inputs.

Using MLPfit in PAW

Introduction

The MLPfit routines are interfaced to the **PAW** analysis package. With this interface, one can :

- Follow the learning curves in the **HIGZ** window while training the network.
- Use inline commands via the **KUIP** interpreter to define the network structure and learning parameters, define the examples, train and use the network, save and recall weights.
- Either choose to define the examples by reading the **learn.pat** and **test.pat** files, as with the standalone package, or define the examples directly from an ntuple.
- Plot the MLP function and study its performances right after training the network, in the same **PAW** session.
- Use a modified version of the **vec/fit** command to fit a linear combination of sigmoid on a vector of points.

The interface to MLPfit 1.33 is available in the CERNLIB 2000 release of end 1999. The interface to MLPfit 1.40 will be available in the next CERNLIB release.

We wish to thank Olivier Couet for the interest he gave to this interface, for the work he had to do to implement it in the standard PAW structure, for the useful advice he gave us about the code and for his help in documenting it.

Please check the PAW home page (<http://wwwinfo.cern.ch/asd/paw/>) for the code, for examples and for the documentation.

Using MLPfit in LabVIEW

LabVIEW is a commercial software for data acquisition by National Instruments. An interface to Multi-Layer perceptrons may be useful for applications in the domains of automatic process control or image analysis.

The MLPfit routines are interfaced to LabVIEW through a small set of vi's . The user can then create its own network application by using these vi's.

The basic interface vi's

The interface vi's are described in this section. For most vi's, the meaning of the inputs and of the outputs is obvious. All vi's have an optional *err in* argument and produce an *err out* value, for execution ordering purposes⁷.

SetNet.vi

To define the Neural Net structure (number of layers and number of neurons per layer). *Err out* has the same meaning as the return value of `mlpsetnet`⁸.

InitWeights.vi

To initialize the weights to random numbers between 0 and 1. *err out* = *err in*.

SetLearn.vi

To define the **learning method and the learning parameters**. The default learning method is the BFGS method. The name of the methods and the corresponding default parameters are listed in the following table :

method	name	parameters
1	Stochastic minimization	eta = 0.1, decay = 1, epsilon = 0, delta = 0
2	Steepest descent with fixed steps	Same as above
3	Steepest descent	Tau = 1.5
4	Ribiere-Polak conjugate gradients	Tau = 1.5, Nreset = 50
5	Fletcher-Reeves conjugate gradients	T au = 1.5, Nreset = 50

⁷ connecting *err out* of a first vi to *err in* of a second one ensures that the first one is executed before the second

⁸ see page 20

6	BFGS	Tau = 1.5, Nreset = 50
---	------	------------------------

The hybrid learning method is not yet available in the LabVIEW interface.

Err out is equal to the return value of `mlpsetlearn_9`.

SetNPat.vi

To define the number of examples (for file 0 = learning examples or file 1 = test examples). *Err out* is equal to the return value of `mlpsetnpat_10`.

SetPat.vi

To define an example. *Err out* is equal to the return value of `mlpsetpat_11`.

Epoch.vi

To run one learning epoch. *Err out* is equal to *err in*.

NNout.vi

Computes the Neural Net output. *Err out* is equal to *err in*.

SaveWeights.vi

To store the weights in a file. *Err out* = *Err in*.

LoadWeights.vi

To read the weights from a file. *Err out* = *Err in*.

Reset.vi

To reset everything concerning MLPfit. Should be called at the end of the job, otherwise memory problems may appear.

The higher level vi's

Create.vi

Creates a neural network, initializes the weights, set learning method and parameters to defaults.

Train.vi

Runs a given number of training epochs.

Example

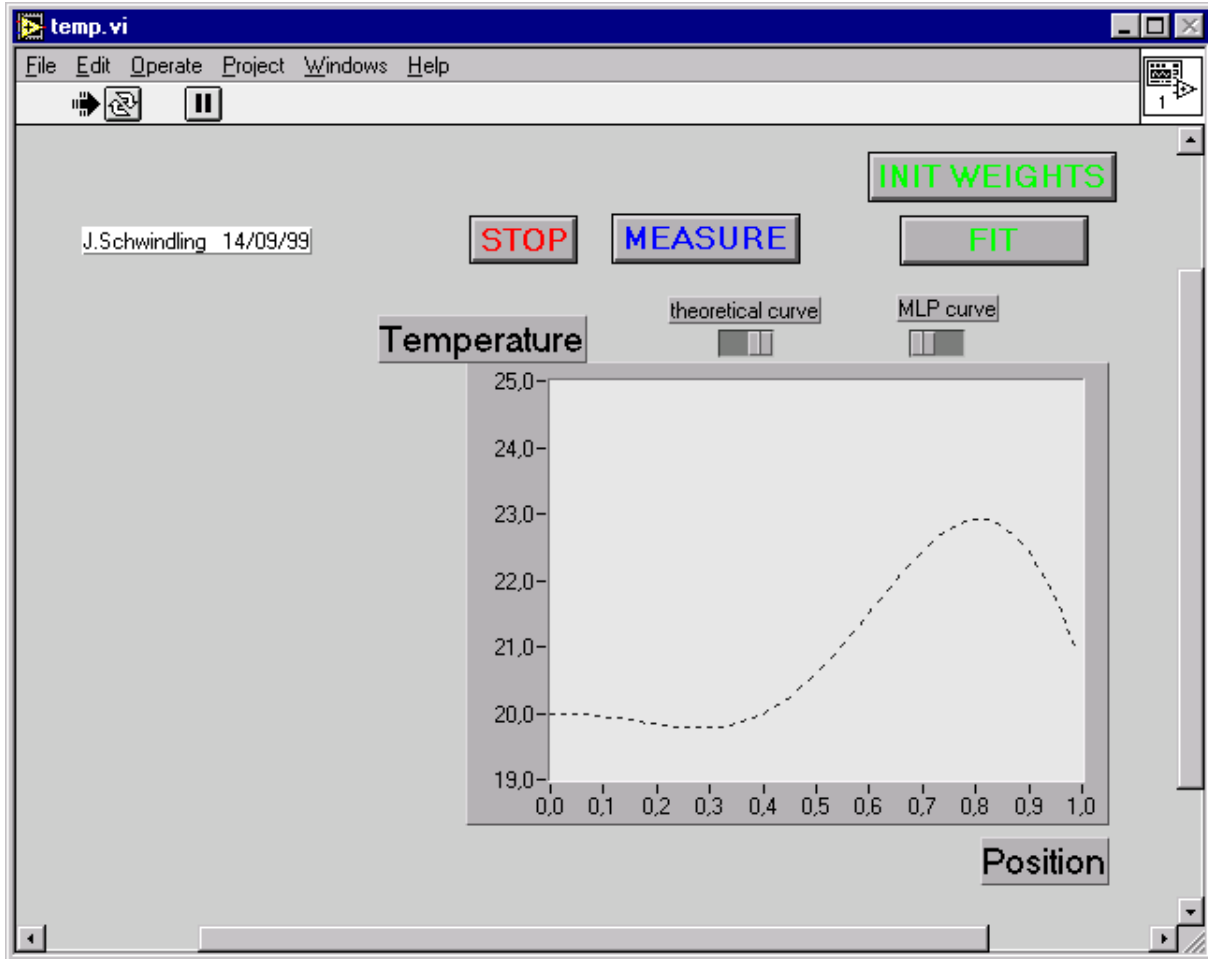
An example of an imaginary Neural Network application with LabVIEW can be found in the directory LabVIEW/Exemples/Temperature.

⁹ see page 20

¹⁰ see page 21

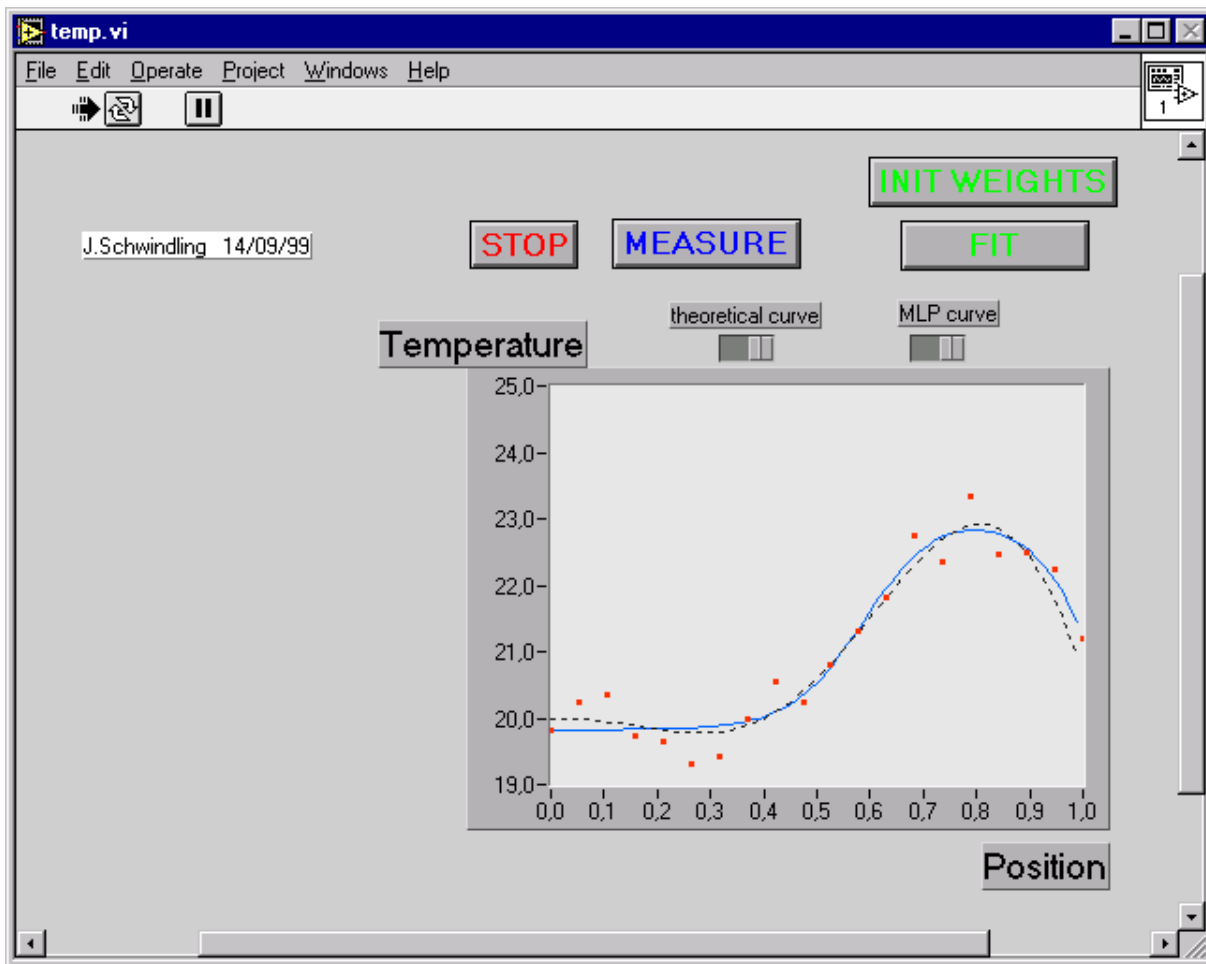
¹¹ see page 22

- Opening **Temp.vi** will start running the vi. The screen should look as follows :



- One sees an imaginary theoretical curve of the variation of a temperature along a direction.
- By clicking on *MEASURE*, 20 measurements are performed along the direction. The measurements have some errors, and are thus not exactly on the theoretical curve.
- Let's assume that the theoretical curve is not known. Remove its display by moving the small switch labeled *theoretical curve*. A MLP can be used to fit the experimental measurements with a continuous function.
- To display the MLP function, move the switch labelled *MLP curve*. A roughly flat curve, obtained from the random initial weights, should be drawn.
- To start fitting the points, simply click on *FIT*. The MLP is trained until the same button is hit again.
- While fitting, the weights can be re-initialized to random values by clicking on *INIT WEIGHTS*.
- New measurements can be performed by clicking on *MEASURE*. If one clicks

on *MEASURE* while training, the network adjusts itself to the new points.



- To stop the vi, click on the red *STOP* button.

Trouble shooting

Using MLPfit from LabVIEW (or from your own code) is very flexible, but also may lead to program crashes if the various MLPfit functions are not performed in the correct sequence :

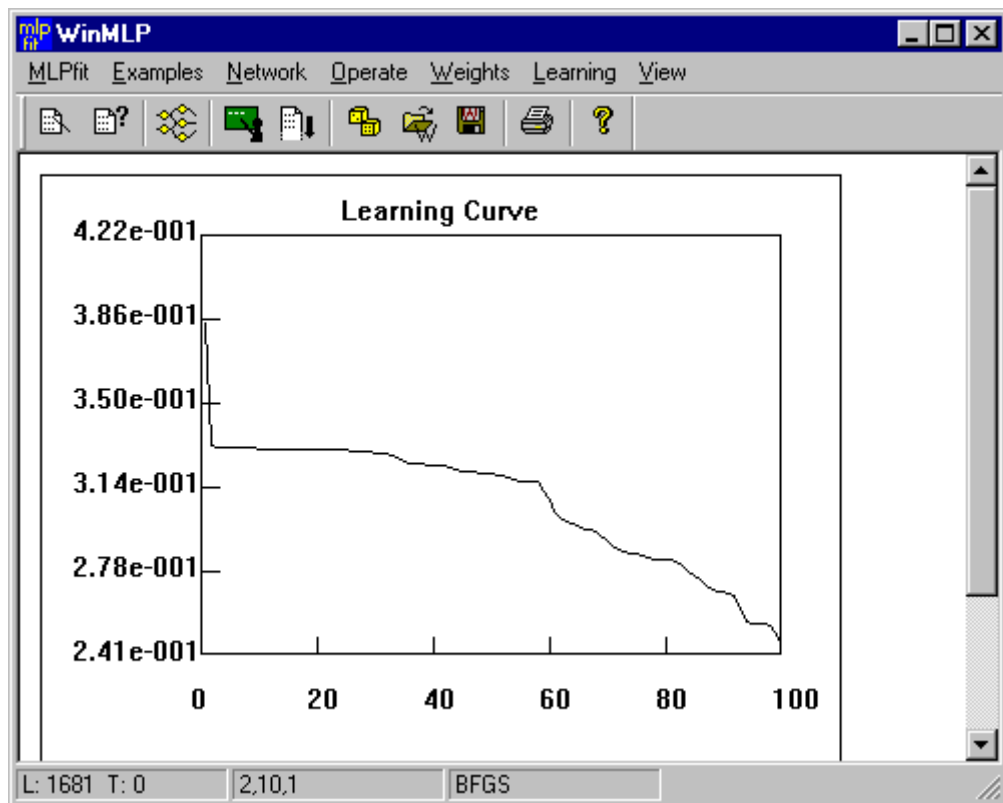
- Always define a network first
- Always define the number of examples before setting the examples
- Always reset MLPfit before going to another network configuration.
- Always test the *Err out* value of the vi's. It should always be 0.
- ...

Work is going on to make MLPfit more robust in case of unexpected situations.

Using the Windows interface

Introduction

MLPfit can be used interactively through a standard « Windows like » interface (called WinMLP) shown on the figure below :



The code for this interface together with examples of data files can be downloaded from the MLPfit web page.

The logic in the WinMLP interface is the following :

1. define the examples
2. then define a network structure
3. (optionally) define a learning method and set the learning

parameters. By default the BFGS method is used.

4. train the network
5. go back to point 1 or point 2

WinMLP tries to ensure a correct sequence by disabling the menu or toolbar items when not available. For example, it is not possible to train a network before having loaded examples and defined a network structure.

Run WinMLP by double-clicking on the WinMLP.exe icon. Help can be obtained by clicking on **Help Topics** in the **MLPfit** menu.