



Using Multi-Layer Perceptrons in PAW



J.Schwindling & B.Mansoulié

DAPNIA/SPP

CEA Saclay

91191 Gif sur Yvette CEDEX

FRANCE

O.Couet

CERN - IT/ASD

CH - 1211 Geneva 23

SWITZERLAND

Contents

- [Contents](#)
 - [Introduction](#)
 - [Neural Networks](#)
 - [Multi-Layer Perceptrons](#)
 - [Multi-Layer Perceptrons in PAW](#)
 - [Limitations, results on different platforms](#)
 - [Organisation of this manual](#)
 - [Acknowledgments](#)
 - [Tutorial](#)
 - [Classification from Ntuples](#)
 - [A note on defining examples from column-wise Ntuples](#)
 - [Fitting a function on a 1d or 2d histogram](#)
 - [Reading and writing ASCII files](#)
 - [Using the modified vec/fit command](#)
 - [Learning methods](#)
 - [Further examples](#)
 - [Fitting 1d functions](#)
 - [A few advice for using Multi-Layer Perceptrons](#)
 - [Normalize your inputs](#)
 - [Use one \(or more\) test samples](#)
 - [Choose the network size](#)
 - [Choose the number of learning examples](#)
 - [Bibliography](#)
-

Introduction

Subsections

- [Neural Networks](#)
 - [Multi-Layer Perceptrons](#)
 - [Multi-Layer Perceptrons in PAW](#)
 - [Limitations, results on different platforms](#)
 - [Organisation of this manual](#)
 - [Acknowledgments](#)
-

Neural Networks

Neural Networks in general, and in particular the [Multi-Layer Perceptron](#) (MLP), are now very widely used in several fields, for example:

- in industry for automatic process control, quality control, optimization of resources allocation.
- in medicine for image analysis and help to diagnosis.
- in meteorology for weather forecast.
- ...

In Particle Physics, they are commonly used, mainly for offline classification tasks (particle identification, event classification, search for new physics). They are also used for track reconstruction or for online triggering.

Multi-Layer Perceptrons

The Multi-layer perceptron is the most widely used type of neural network. It is both simple and based on solid mathematical grounds. Input quantities are processed through successive layers of "neurons". There is always an input layer, with a number of neurons equal to the number of variables of the problem, and an output layer, where the perceptron response is made available, with a number of neurons equal to the desired number of quantities computed from the inputs (very often only one). The layers in between are called "hidden" layers. With no hidden layer, the perceptron can only perform linear tasks (for example a linear discriminant analysis, which is already useful). All problems which can be solved by a perceptron can be solved with only one hidden layer, but it is sometimes more efficient to use 2 hidden layers. Each neuron of a layer other than the input layer computes first a linear combination of the outputs of the neurons of the previous layer, plus a bias. The coefficients of the linear combinations plus the biases are called the weights. They are usually determined from examples to minimize, on the set of examples, the (Euclidian) norm of the desired output - net output vector. Neurons in the hidden layer then compute a non-linear function of their input. In MLPfit, the non-linear function is the sigmoid function $y(x) = 1/(1+\exp(-x))$. The output neuron(s) has its output equal to the linear combination. Thus, a Multi-Layer Perceptron with 1 hidden layer basically performs a linear combination of sigmoid function of the inputs. A linear combination of sigmoids is useful because of 2 theorems:

- a linear function of sigmoids can approximate any continuous function of 1 or more variable(s). This is useful to obtain a continuous function fitting a finite set of points when no underlying model is available.
- trained with a desired answer = 1 for signal and 0 for background, the approximated function is the probability of signal knowing the input values. This second theorem is the basic ground for all classification applications.

Multi-Layer Perceptrons in PAW

The Multi-Layer perceptron interface in PAW:

- can be used for both approximation and classification tasks.
- provides performant minimisation methods to determine the weights.
- allows to interactively define, train and use the neural network.

More precisely, by using a few commands with a syntax close to the usual PAW commands, it is possible to:

- define the network structure
- modify the default learning parameters
- read/write weight files
- define the examples from ASCII files, histograms or Ntuples. When examples are defined from Ntuples, selection criteria may be added
- train the network and follow the learning curve while training
- write out the function for use in any other code or for direct use in PAW

Multi-Layer Perceptrons are implemented in PAW through an interface to the MLPfit package (version 1.33). In addition to its interface with PAW, the MLPfit package can be used in a standalone mode, from calls in a user fortran or C code or through an interface to LabVIEW. Details on this package can be found in

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

Limitations, results on different platforms

There are currently the following limitations to the multi-layer perceptrons in PAW (in brackets is indicated whether the limitation comes from the `MLPfit` package or from PAW):

- Maximum number of hidden layers = 2 (`MLPfit`). This should not be a limitation for most applications.
- Maximum number of neurons per layer = 100 (`MLPfit`). This is unlikely to be a limitation for most of the usual applications in High Energy Physics.
- Maximum number of inputs + outputs when set from an N-tuple = 29 (`PAW`).

The initial weights are taken randomly. There is no guarantee that they are the same on different platforms. This may lead to different results on different machines. The results may even be very different if the network is trapped in a local minimum.

Organisation of this manual

This manual is organized as follows: chapter [2](#) is a simple tutorial for using MLPfit in PAW. Chapter [3](#) gives a brief description of the various learning methods. Further examples are given in chapter [4](#) and, finally, a few general advice on how to use correctly a Multi-Layer Perceptron are given in chapter [5](#).

Warnings about possible mistakes are written in bold, for example:

Warning: this manual is not supposed to replace a full description of Multi-Layer Perceptrons and of their usage. However, please read carefully chapter [5](#) before starting.

Acknowledgments

We wish to thank Boris Tuchming for trying a preliminary version of the interface and for suggesting useful improvements.


Tutorial

This sections shows, by simple examples, how to use the multi-layer perceptron package in PAW.

Subsections

- [Classification from Ntuples](#)
 - [A note on defining examples from column-wise Ntuples](#)
 - [Fitting a function on a 1d or 2d histogram](#)
 - [Reading and writing ASCII files](#)
 - [Using the modified `vec/fit` command](#)
-

Classification from Ntuples

This section describes how to train a network for a classification task using examples contained in 2 ntuples (*ww.ntup* and *qq.ntup*) 

After entering PAW, type

```
h/fil 1 ww.ntup
n/pri 2000
```

Ntuple *ww.ntup* contains 11297 events, with 14 discriminating variables for each event (*v1* to *v14*), plus a variable (*typ*) which is 0 for $WW \rightarrow qq\bar{q}\bar{q}$ events and 1 for $WW \rightarrow l\nu qq$ decays.

Assume that you want to try a 4-8-1 network to select the $WW \rightarrow qq\bar{q}\bar{q}$ events. You must first define the network structure by typing:

```
mlp/create 4 8
```

The `mlp/create` command is followed by the number of neurons in each layer (up to 4 layers). The default number of neurons in the output layer is 1. To create a network with 2 output neurons, the command `mlp/create 4 8 ! 2` must be issued, since the third argument is the number of neurons in the second hidden layer (by default 0).

The examples must then be set by typing:

```
mlp/lpat/set 2000 sqrt(v1)%v2%v4%v5 1. 1. 1000 1 typ=0
mlp/lpat/set 2000 sqrt(v1)%v2%v4%v5 0. 1. 1000 1 typ=1 +
```

The first `mlp/lpat/set` command defines the signal events. The inputs of the neural net are $\sqrt{v1}$, $v2$, $v4$, $v5$.

Combinations such as $v1 + v2$, etc, would also have been possible. The desired answer is set to 1. The next ``1" is a weight for the signal events. Then follows the number of events (1000) and the first (1) to consider. Finally, a selection criteria (`typ=0`) is applied.

the second `mlp/lpat/set` command defines the background examples. The desired answer is set to 0. These examples are added to the previously defined ones by the ``+" statement.

Warning: please note that `mlp/create` should be issued before defining the examples. Note also that `mlp/create` initializes the weights randomly.

One can then train the network by the command:

```
mlp/learn 100
```

where 100 is the number of epochs. The BFGS minimization algorithm is used by default. The learning method can be changed by:

```
mlp/lmet 7
```

The learning methods are described in chapter 3. One can then continue to train the network, from the previous state, by the command:

```
mlp/learn 100 +
```

While training, the error is displayed on the screen. The curve is also put in histogram 2000000. To avoid over-fitting, the network performance should be monitored on independent examples, called *test patterns*. One can define test patterns by:

```
mlp/tpat/set 2000 sqrt(v1)%v2%v4%v5 1. 1. 1000 1001 typ=0
mlp/tpat/set 2000 sqrt(v1)%v2%v4%v5 0. 1. 1000 1001 typ=1 +
```

If one continues to train the network by

```
mlp/learn 100 +
```

one gets this time two curves, corresponding to the errors on the learning and test samples. The error curve for the test sample is stored in histogram 2000001.

After training, the Multi-Layer selection function is written in file *pawmlp.f*. In case the examples were taken from N-tuples, one can use this function to test the network by typing:

```
n/plot 2000.pawmlp.f(0.) typ=0 1000 2001
n/plot 2000.pawmlp.f(0.) typ=1 1000 2001 ! s
```

Finally, one may want to save the network weights for use in another PAW session. This can be done by:

```
mlp/weights/save w.out
```

Let's see now how to define signal and background examples from 2 different N-tuples:

```
mlp/reset | to reset everything to 0
```

```
mlp/create 8 5 | create the network first
```

```
h/fil 1 ww.ntup
mlp/lpat/set //lun1/2000 sqrt(v1)%v2%v3%v4%v5%v6%v7%v8 1. 1. 1000 1 typ=0
mlp/tpat/set //lun1/2000 sqrt(v1)%v2%v3%v4%v5%v6%v7%v8 1. 1. 4000 5001 typ=0
h/fil 2 qq.ntup
n/pri 2000
mlp/lpat/set //lun2/2000 sqrt(v1)%v2%v3%v4%v5%v6%v7%v8 0. 1. 1000 1 ! +
mlp/tpat/set //lun2/2000 sqrt(v1)%v2%v3%v4%v5%v6%v7%v8 0. 1. 4000 5001 ! +
mlp/learn 100
read suite
ld 1000 ' ' 100 -0.5 1.5
ld 1100 ' ' 100 -0.5 1.5
nt/proj 1000 //lun1/2000.pawmlp.f(0.) typ=0
nt/proj 1100 //lun2/2000.pawmlp.f(0.)
set xmg1 2.5
```

```
set xlab 1.7
his/pl 1100
set htyp -3
his/pl 1000 s
set htyp
atit 'NN output' 'Number of events'
```

should produce a plot looking like figure [2.1](#).

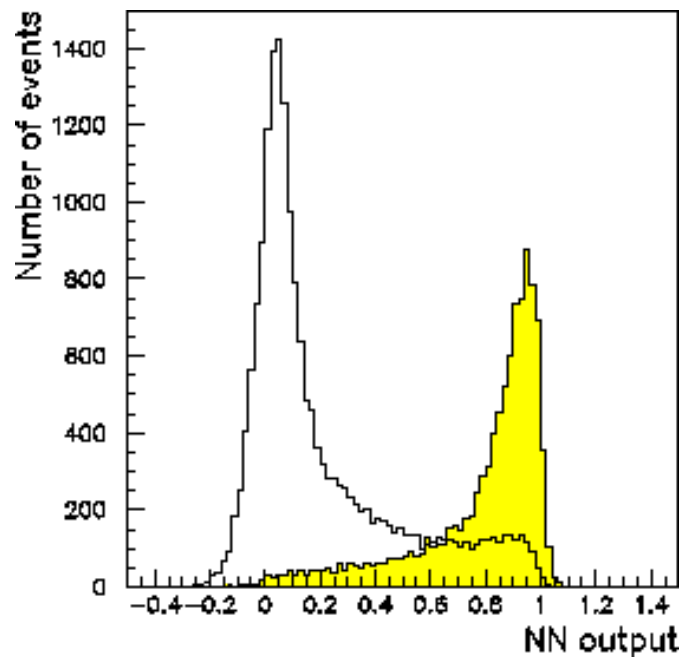


Figure 2.1: Test of the neural network performance on signal (grey histogram) and background events.

A note on defining examples from column-wise Ntuples

The previous section described how to define examples from row-wise Ntuples. The syntax for defining examples from column-wise Ntuples is quite similar:

```
mlp/lpat/set //lun1/2000 sqrt(v1(1))%v2(1)%v3(1) 1. 1. 1000 1 typ(1)=0
```

is allowed, as well as (if it makes sense):

```
mlp/lpat/set //lun1/2000 sqrt(v1(1))%v2(1)%v3(1)%v3(2) 1. 1. 1000 1 typ(1)=0
```

The following line is **not** allowed if v2 has more than 1 column:

```
mlp/lpat/set //lun1/2000 sqrt(v1(1))%v2%v3(1)%v3(2) 1. 1. 1000 1 typ(1)=0
```

If one wishes to set examples from 2 columns, the following commands should be used:

```
mlp/lpat/set //lun1/2000 sqrt(v1(1))%v2(1)%v3(1) 1. 1. 1000 1 typ(1)=0  
mlp/lpat/set //lun1/2000 sqrt(v1(2))%v2(2)%v3(2) 1. 1. 1000 1 typ(2)=0 +
```


Fitting a function on a 1d or 2d histogram

A multi-layer perceptron can be used to fit a continuous function of one or more variables. This can be done with the variables and the desired values in an ntuple, by setting the examples by the command:

```
mlp/lpat/set 2000 x1%x2%x3 y
```

This is useful to fit a function of 3 variables or more.

Warning: even in the case of 1 or 2 variables, `pawmlp.f` is intended to be used in `nt/plot` and cannot be used directly by `fun1` (or `fun2`). 

In the case of a function of 1 or 2 variables, the examples can be set directly from 1d or 2d histograms. For example, if you want to fit a function on the histogram 100 shown in figure 2.2  you can type:

```
mlp/create 1 4
mlp/lpat/set 100
mlp/learn 100
```

and plot the result:

```
his/pl 100
func/pl pawmlp.f(x) -5. 5. s
```

Figure 2.2 compares the fit results of the 1-4-1 mlp and of a polynomial with the same number of parameters (13).

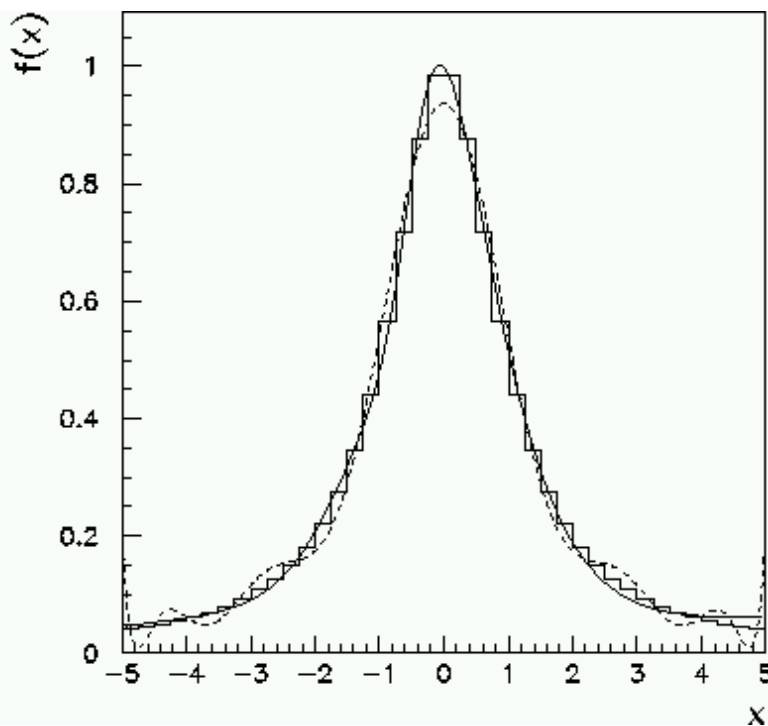
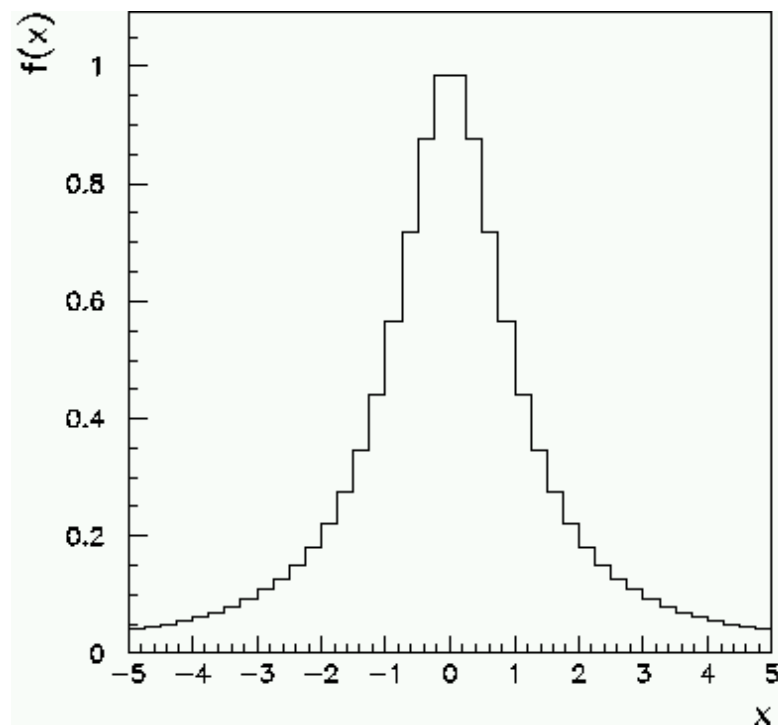


Figure 2.2: 1d histogram to fit. **Figure 2.3:** Result of the fit by a 1-4-1 neural network (full curve) and by a 12th order polynomial (dashed curve).

An example of the fit of a 2d histogram is the following:

```
fun2 200 x**2*sin(x*y) 40 -2. 2. 40 -2. 2.
for/fil 67 fun2.eps
meta 67 -113
surf 200
```

Fitting a function on a 1d or 2d histogram

```
atit 'x?1!' 'x?2!' 'y'  
close 67
```

```
mlp/create 2 15  
mlp/lpat/set 200  
mlp/lmet 7  
mlp/learn 200
```

```
fun2 300 pawmlp.f 40 -2. 2. 40 -2. 2.  
his/ope/add 200 300 1000 1. -1.
```

```
for/fil 67 diff.eps  
meta 67 -113  
surf 1000  
atit 'x?1!' 'x?2!' 'y'  
close 67
```

with fun2.eps and diff.eps shown on figure [2.4](#) and [2.5](#).

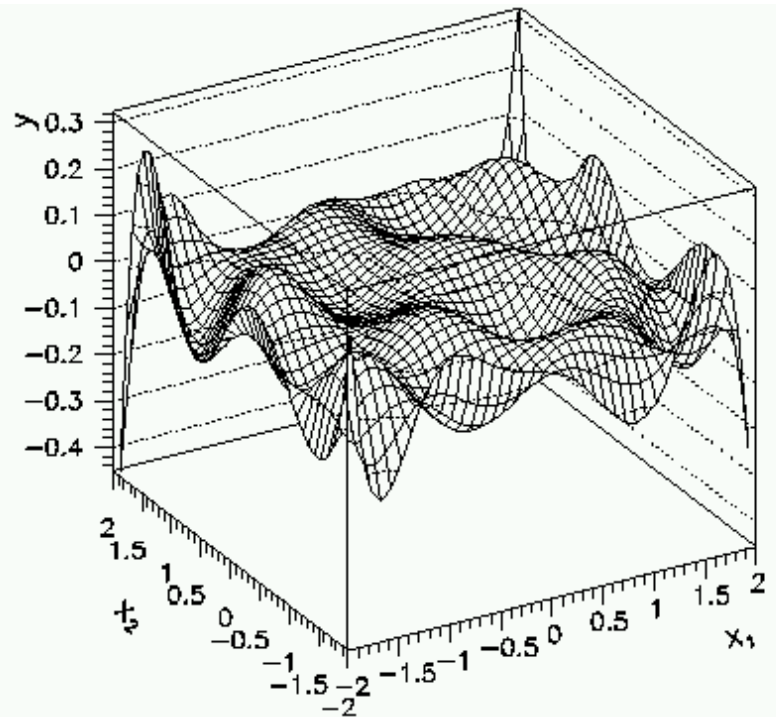
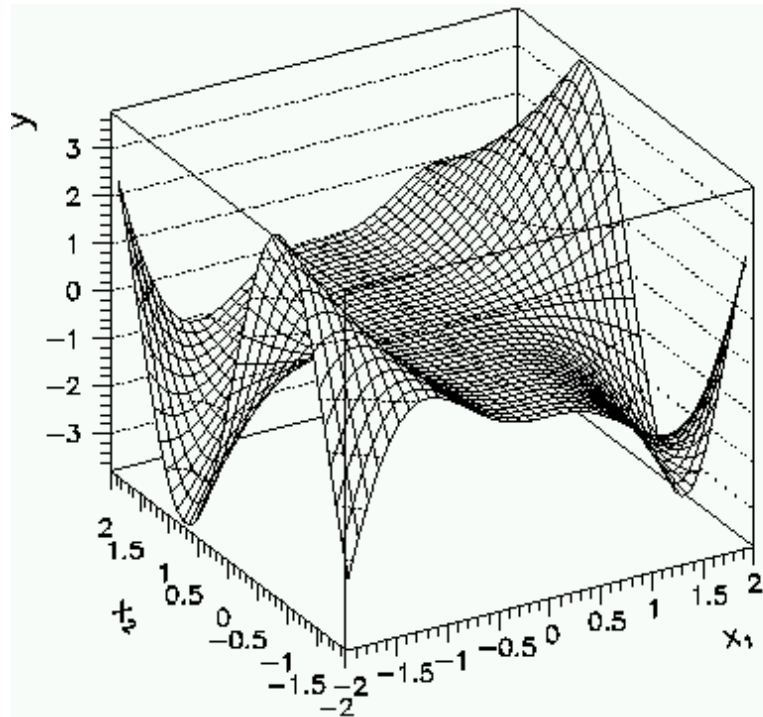


Figure 2.4: fun2.eps Figure 2.5: diff.eps

[→](#) [↑](#) [←](#) [🏠](#) Next: [Reading and writing ASCII](#) Up: [Tutorial](#) Previous: [A note on defining](#)

Reading and writing ASCII files

Let's start again from the histogram defined by

```
fun2 200 x**2*sin(x*y) 40 -2. 2. 40 -2. 2.
```

define the network, the examples and train for 20 epochs with the hybrid method by:

```
mlp/create 2 15
mlp/lpat/set 200
mlp/lmet 7
mlp/learn 20
```

At this stage, one can save the weights by

```
mlp/weights/write w.out
```

One can then continue to train with the hybrid method, save the learning curve, reload the weights and try the BFGS method instead:

```
mlp/learn 20 +
h/copy 2000000 100
mlp/weights/read w.out
mlp/lmet 6
mlp/learn 20 +
his/pl 100
his/pl 2000000 s
```

The learning examples used here can also be saved to an ASCII file by typing:

```
mlp/lpat/write h2d.pat
```

The format of the output file is compatible with the format used by the MLPfit package, allowing, for example, the use of the same data by the LabVIEW interface. The example file can be loaded (in another PAW session) by:

```
mlp/lpat/read h2d.pat
```

Similar commands can be used to read/write the test examples.

The MLPfit example files may also contain other command lines, for example:

```
# number of layers
NLAY 3
# number of neurons in each layer
NNEU 2,10,1
# learning method
LMET 7
# regularisation
LAMB 1.
# n reset
NRES 2000
# tau
LTAU 3.0
# learning parameter
LPAR 0.1
# decay speed
DCAY 1.0
# momentum factor
MOME 0.0
# number of epochs
NEPO 1000
```

These lines, if present, are also interpreted when reading examples, and can be used to define the network and the learning parameters.

    **Next:** [Using the modified vec/fit](#) **Up:** [Tutorial](#) **Previous:** [Fitting a function on](#)

Using the modified `vec/fit` command

The PAW `vec/fit` command has been modified to allow fitting a vector of points (with error bars) by a multi-layer perceptron function. This can be done by the command:

```
vec/fit x y ey NNi [chopt nepoch]
```

with

```
i[,j]      ``Number of neurons in the hidden layer(s)''
```

```
chopt C ``options" D= ''
```

```
nepoch I ``number of epochs" D=100
```

Warning: the learning method is the BFGS method and cannot be changed.

For example:

```
vec/create x(14) r 0. 0.5 1. 1.5 2. 2.5 3. 3.5 4. 4.5 5.0 5.5 6.0 6.5
vec/create y(14) r 0.05 0.1 .3 .4 .3 .2 .2 .3 .4 .4 .42 .44 .46 .48
vec/create ey(14) r .02 .02 .02 .02 .05 .02 .02 0.02 0.05 .06 .02 .02 .02 .02

ld 100 ' ' 14 -0.25 6.75
his/put_vect/content 100 y
his/put_vect/errors 100 ey

set mtyp 20
his/pl 100 pe

vec/fit x y ey NN4 s 400
```

If the option '+' is given, the network continues to minimize starting from the previous weights. If the option 'I' is given, different random weights are used. Thus, repeated `vec/fit x y ey NN4 ! 400` commands all give the same result, whereas repeated `vec/fit x y ey NN4 I 400` commands all lead to a different result because of different initial weights.

Learning methods

This chapter is only intended to give a very crude idea of what the minimization methods available through the PAW - MLPfit interface are. More information is available on the [MLPfit web page](#).

In MLPfit, the multi-layer perceptrons are standard feed-forward networks with 1 or 2 hidden layers.

The activation function of the hidden neurons is the usual sigmoid function $y = 1/(1 + e^{-x})$,

whereas the output neuron is linear. This choice is natural for function approximation but can also be used for classification.

Several learning methods are available. They all try to minimize $E = \sum_p e_p$, where p runs on all examples (patterns) of the learning sample, and

$$e_p = \frac{1}{2} \omega_p (o_p - t_p)^2$$

where o_p is the output value of the neural network for pattern p , t_p the desired value and ω_p a

weight per example, usually set to 1 but which is set to $1/\sigma_p^2$ for fitting with error bars.

In all the methods, one needs to compute the first order derivatives $\partial e_p / \partial w_{ij}$ (or

$\partial E / \partial w_{ij} = \sum_p \partial e_p / \partial w_{ij}$), with respect to the weights w_{ij} . This computation is done by

back-propagation of the errors.

The minimization are then:

- *Stochastic minimization*: this is the traditional learning method, still wrongly called "standard backpropagation". It is the Robbins-Monro stochastic approximation method [1] applied to multi-layer perceptron. This method is known to be very slow.
- *Minimization methods with line search*: these methods are taken from the theory of unconstrained minimization, with the parameters being the weights w_{ij} , which can be seen as a vector \vec{w}_t .

The minimization is an iterative process, with each iteration t (an *epoch*) of the form:

- 1) from the gradient $\partial E / \partial w_{ij}$ find a direction \vec{s}_t
- 2) find α_m which minimizes $E(\vec{w}_t + \alpha \vec{s}_t)$
- 3) set $\vec{w}_{t+1} = \vec{w}_t + \alpha_m \vec{s}_t$


Step 2 is called the "*Line Search*". Various minimization algorithms differ by the way step 1 is done. The simplest one is the *steepest descent algorithm*, where $\vec{s} = -\partial E / \partial w_{ij}$. The two

other methods which have been tried are the *Conjugate Gradients* (CG) (with the Fletcher-Reeves updating formula) and the *Broyden, Fletcher, Goldfarb, Shanno* (BFGS) methods. These methods are described in detail in [2].

- *The hybrid linear-BFGS method*: for a given set w_{NL} of input \rightarrow hidden weights, the set w_L^* of hidden \rightarrow output weights which minimizes E can be determined by solving a linear system of equation [3]. This is possible because the output neuron is linear, and because of the quadratic form of the error E . 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 becomes:

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

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

The minimization methods with line search, especially the BFGS  or the hybrid method, are, at least on relatively small problems (a few inputs, a few tens of weights, a few thousand examples), about 10 times faster than the standard stochastic minimization method. On larger problems (15 inputs, hundreds of weights, more than 50 000 examples), the stochastic minimization performs well.

    **Next:** [Further examples](#) **Up:** [pawmlp](#) **Previous:** [Using the modified vec/fit](#)



Further examples

Subsections

- [Fitting 1d functions](#)
-

Fitting 1d functions

This example shows how to fit 1d functions and display the result while learning.

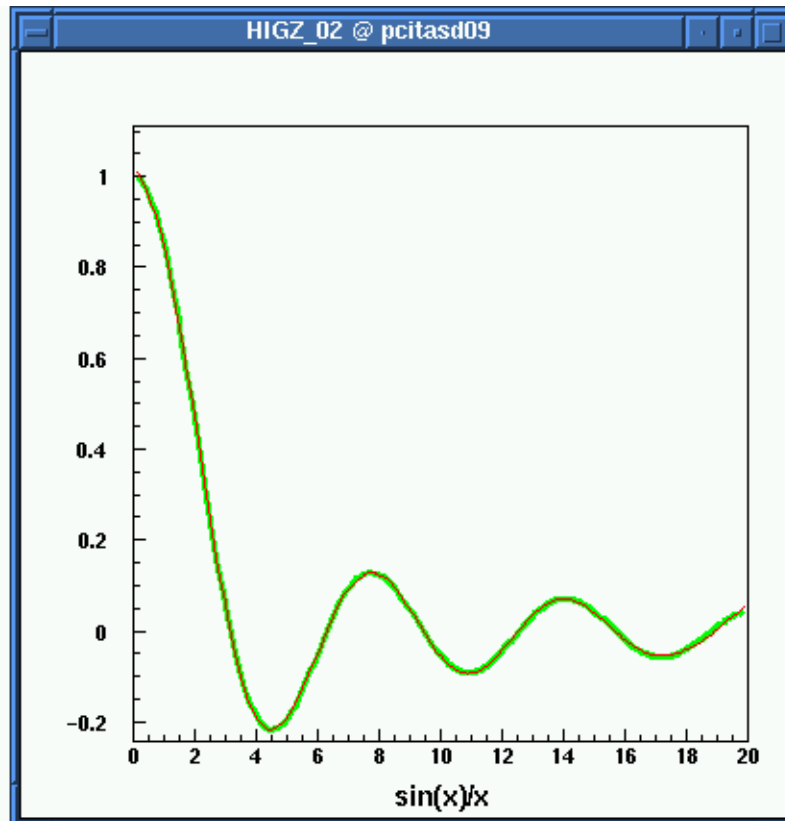
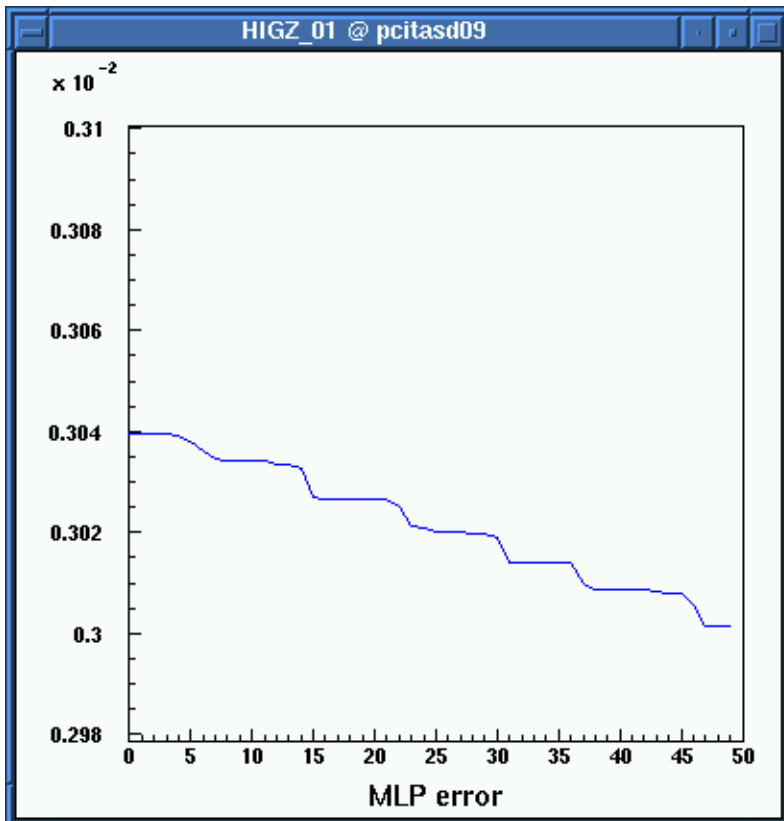
First, create in your working directory the file `higz_windows.dat` with the following lines:

```
0000 0000 0400 0400
0420 0000 0400 0400
```

This file will be used to define where the positions and sizes of the two windows.

Then create and execute the following kumac:

```
fun1 10 sin(x)/x 100 0. 20.
mlp/create 1 5
mlp/lpat/set 10
mlp/learn 1
work 2 0 2
do i=1,80
mlp/learn 50 +q
work 1 D
work 2 A
h/pl 10
fun/plot pawmlp.f 0 20 s
work 2 D
work 1 A
enddo
```



A few advice for using Multi-Layer Perceptrons

Subsections

- [Normalize your inputs](#)
 - [Use one \(or more\) test samples](#)
 - [Choose the network size](#)
 - [Choose the number of learning examples](#)
-

Normalize your inputs

In order for the exponentials in the sigmoid transfer functions to make sense, their input should not be too large. The initial weights being chosen randomly between -1 and 1, the input variables should also be of the order of 1. For example, if a variable is between 0 and 100, try to divide it by 100. If a variable is between 80 and 81, subtract 80, etc.

In the same way, as the output layer computes a linear combination of sigmoids (which are between 0 and 1), it is difficult for the network to reach large values. So, if answers are, for example, between 0 and 20, divide by 20, etc.



Use one (or more) test samples

As any fit, MLPs are subject to over-fitting if the number of parameters (weights) is too large. When over-fitting occurs, the error on the learning sample keeps decreasing while the error on an independent test sample starts to increase. It is possible to define a test sample by using the `mlp/tpat/set` command. The error on the test sample is displayed together with the learning curve.

In any case, it is recommended to perform as many independent checks of the network performance as possible.

Choose the network size

In principle, one hidden layer is sufficient to solve any problem, but the number of neurons needed is not specified. In practice, it may happen that two hidden layers with a small number of neurons may work better (and/or learn faster) than a network with a single layer. It is not possible to use more than 2 hidden layers in MLPfit.

There are no rules to choose the number of hidden neurons. One should thus fix it in an empirical way, by following the evolution of the error with the number of epochs, on the test examples, for different layer sizes. This is illustrated on figure 5.1, where it can be seen that:

- With too few neurons, the performance is poor.
- With too many neurons, the network first converges first to a good performance but then, on the test file, degrades because of over-fitting.

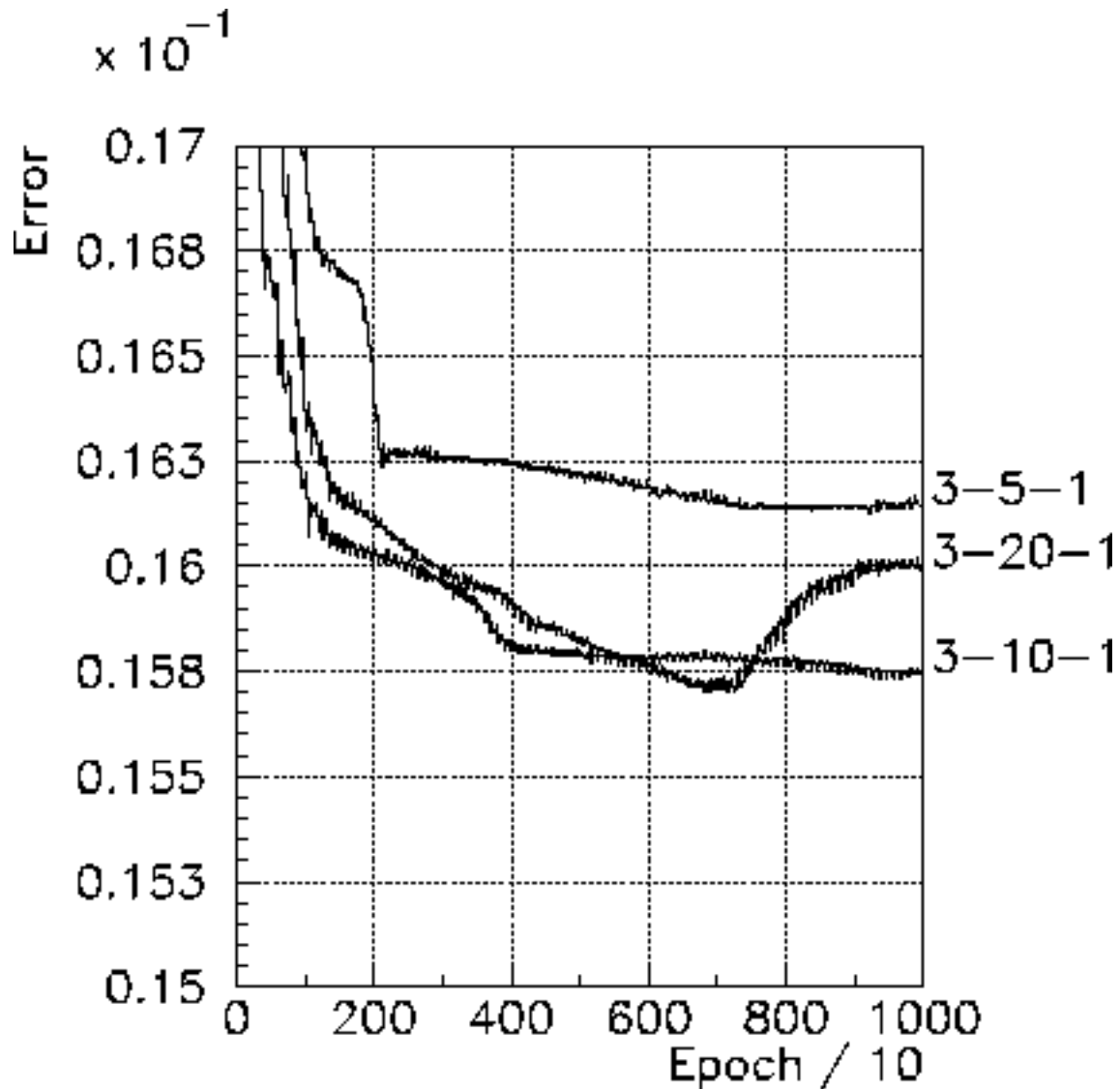






Figure 5.1: Evolution of the error on the test file as a function of the

number of epochs for various network sizes.

It seems better to start with a small number of neurons, because:

- learning is faster
- it is often enough
- it avoids over-fitting problems

    **Next:** [Choose the number of](#) **Up:** [A few advice for](#) **Previous:** [Use one \(or more\)](#)

Choose the number of learning examples

While the performance can be improved by increasing the number of hidden neurons, the learning sample must be increased accordingly to avoid over-fitting. There seem to be no strict rule on the ratio Number of examples / Number of weights, which should be between 10 and 100.



Bibliography

1

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

2

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

3

A Hybrid Linear / Nonlinear Training Algorithm for Feedforward Neural Networks, S. McLoone et al., *IEEE Transactions on Neural Networks*, vol. 9, **11** 4 (1998), p. 669
