Handout for the use of Matlab & Netlab to handle neural networks

1) Getting started with Matlab
2) Getting started with Netlab
3) The basic structure of the neural network code

1. MATLAB

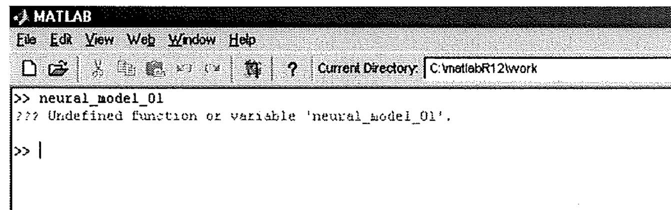
Starting Matlab

Matlab is a computer language that has several advantages like handling matrix operations very easily. Matlab is available on every computer in the JMU libraries. You just have to look under....

Start > JMU Applications > Subject Software > Natural Sciences and Maths → Matlab (Version 5 or 6)

Starting a program

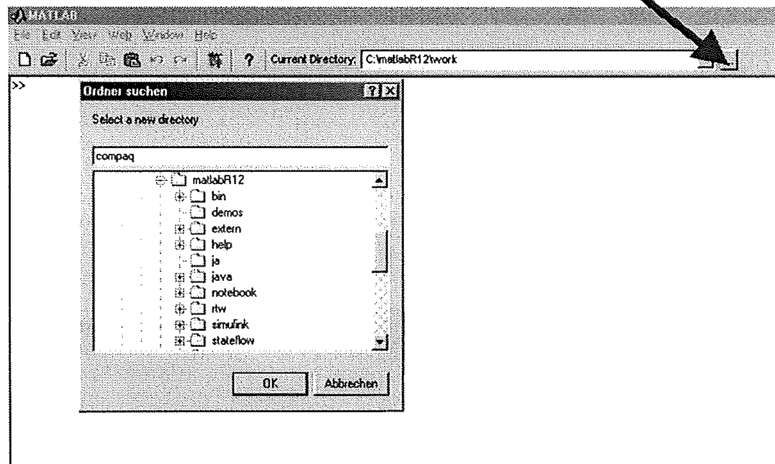
To start your own program, you have to type in its name (e.g. 'neural_model_01')!



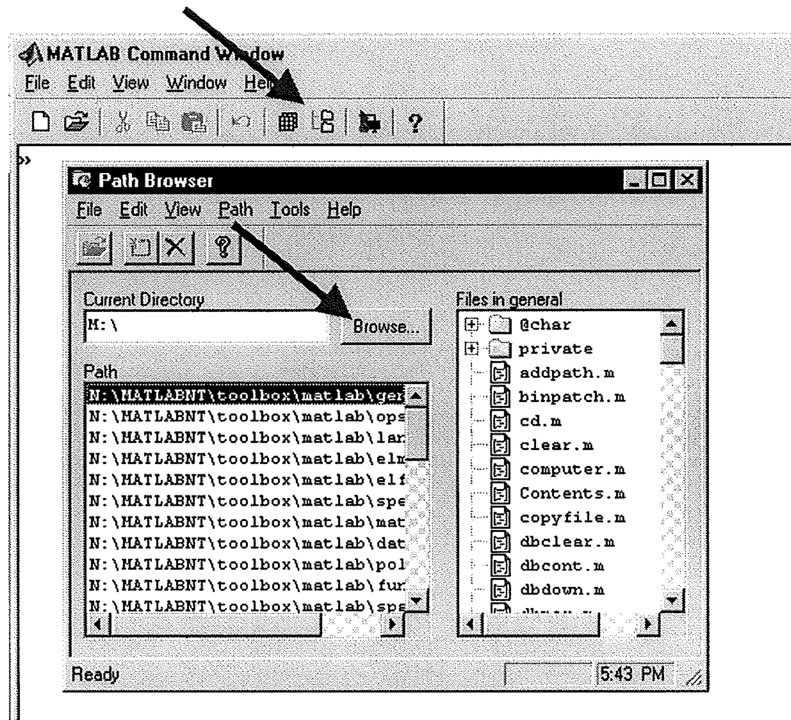
Undefined function?? Before everything else, you have to tell Matlab in which folder it should search for your program!

There are many ways to do so:
a) You could either select the folder in which your program is stored by clicking the button below...

(Matlab 6)



(Matlab 5)



b) ... or you are using the following command code:

path(path,'M:\your_folder\your_program')

[I found out that method (a) is somehow not working on every computer. In case you are in front of one of them, use method (b). It should work all the time!].

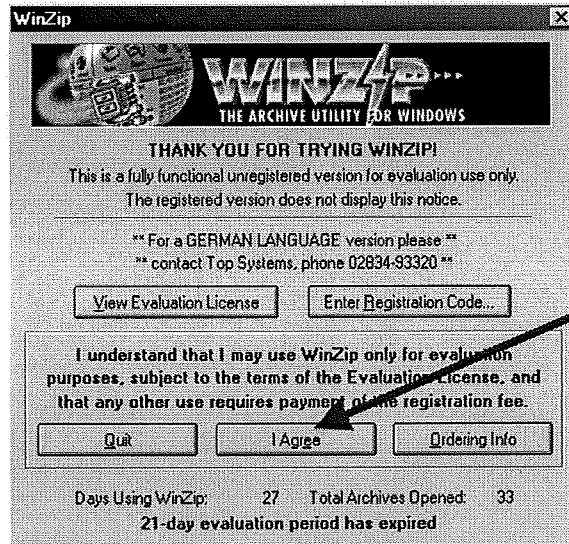
NETLAB

Netlab is a free available software from the Aston University/Birmingham. It contains a lot of useful programs, which we could use to build our own neural network. All those functions are written in Matlab code which is the reason we first have to start Matlab until we can use Netlab.

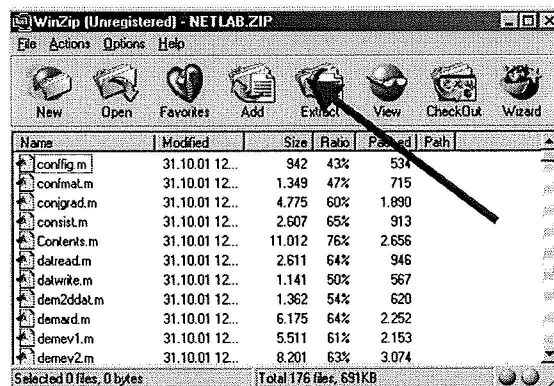
You could download Netlab from the Internet from the following address:

http://www.ncrg.aston.ac.uk/netlab/

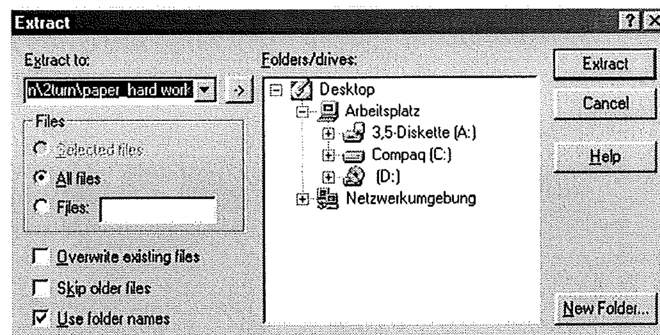
Please create a new folder in your M-drive and save the Netlab file there.
Next, double-click it. Since it is in WinZip-format, you should see the following window next. Please click 'I aggree'.



On the next window, you should click 'extract'.



Doing so, you get the chance to specify the folder. Please select the one you have created in your M-drive.



Now you are able to create your own neural network. You have all the tools!

The basic structure of the neural network code

Basics

A Matlab-file that should contain the program code could be created by clicking in the Matlab window: FILE>NEW>M-FILE. Some older versions don't have this editor. In this case, you could use the Microsoft program 'Wordpad' (or any other word processing program like 'Word'). When you save a file which is intended to be a Matlab file, you have to save it with the extension '.m' (e.g. file_name.m'). Otherwise Matlab won't be able to identify your program as a file that contains Matlab code.

Get data

First, we will start with the problem how to get the time series / the data into Matlab. If you have the input time series (1.1, 2.2, 3.3) you have to write:

```
Time_series_name = [1.1; 2.2; 3.3]
```

or

```
Time_series_name = [
1.1
2.2
3.3]
```

That creates a column vector with the name 'Time_series_name' that includes your values.

After saving your program and starting it by typing in its name in the Matlab command window and pressing the enter button, Matlab shows the vector.

Tip: If you would like to define a vector but you don't want Matlab to show it all the time, you could type a ';' after each command line.

A way to handle large time series much easier is to create a separate file (with the name 'Time_series_name.m') which includes the vector.

```
function [vector] = Time_series_name;

vector= [
1.1
2.2
3.3
];
```

Tip: You have a neural network with 2 input nodes (that is you have 2 time series as input for your neural network). Unfortunately, your data are all in Excel. If your time series are in columns (if not, transpose them in Excel by copying them and using EDIT>PASTE SPECIAL >TRANSPOSE) you could simply copy [mark the data and press 'Ctrl+c' in Excel]/paste [press 'Ctrl+v' in Matlab] them into your Matlab file. Make sure, you paste them at the right place which is...

Bernau

```
function [vector] = Time_series_name;

vector= [
...here!!!
e.g.:
1       2
11      22
111     222
];
```

For your neural network, you have to create two files. One with the name 'input_TRAINING' which includes the training data and another 'output_TRAINING' which includes the corresponding target values.

Starting our neural network program, we define a variable x, which includes the input training data and a variable t, which represents the target values. The Matlab code looks like this:

```
x           = [input_TRAINING];
t           = [output_TRAINING];
```

Set up network parameters.

Now we can define the parameter of the network:
(Please note, that you could add comments in your Matlab code by marking their beginning with a '%'. Everything that follows a '%' will be ignored by Matlab)

```
nin         = 10;           % Number of inputs.
nhidden     = 5;            % Number of hidden units.
nout        = 1;            % Number of outputs.
```

Create a 2-layer feed-forward network.

```
net         = mlp (nin, nhidden, nout, 'linear');
```

This command takes the number of inputs, hidden units and output units for a 2-layer feed-forward network (the definition varies sometimes; here a 2-layer feed forward network consists of one hidden layer), together with string 'linear' which specifies the output unit activation function (beside 'linear', its values could also be 'logistic' or 'softmax'), and returns a data structure net.

The weights are drawn from a zero mean, unit variance Gaussian distribution (=normal distribution). The hidden units use the tanh activation function (hyperbolic tangent).

If you save your program now and execute it in Matlab, you get the following reply by typing in 'net' (which is the result of the above used function 'mlp').

```
type            = 'mlp'
nin             = number of inputs
nhidden         = number of hidden units
```

nout	= number of outputs
nwts	= total number of weights and biases
actfn	= string describing the output unit activation function
w1	= first-layer weight matrix
b1	= first-layer bias vector
w2	= second-layer weight matrix
b2	= second-layer bias vector

Since the weights are stored as vectors, you have to type 'net.w1' to see the values of the weights 'w1'.

Here:

w1	has dimensions nin times nhidden,
b1	has dimensions 1 times nhidden,
w2	has dimensions nhidden times nout, and
b2	has dimensions 1 times nout.

Explanation: a bias weight (here: b1 and b2) has the same function as the intercept *b* in the equation $y = m*x + b$.

% Set up vector of options for the optimiser.

```
options       = zeros(1,18);   % sets each of the 18 options to zero
options(1)    = 1;             % This provides display of error values.
options(14)   = 10;            % Number of training cycles.
options(17)   = 0.001;         % momentum
options(18)   = 0.0005;        % lrate
```

By defining values for each of the 18 options, you get the chance to influence the number of training iterations, the 'speed' of the learning process, ... and many more.

Train

Having now created a neural network structure, having initialised our weights with random numbers and having specified the training parameters, we now simply have to train our network.

This is done via the command 'netopt':

```
net    = netopt(net, options, x, t, 'graddesc');
```

where the function 'netopt' needs as input the network structure 'net', the values of the parameters stored in 'options', the training input data set 'x', the corresponding training data targets 't' and a specification of the training algorithm (here: gradient descend).

As a result of the function 'netopt' you get the updated network structure (in fact, only the weights should have changed!).

Application to out-of-sample data

Since we now have the trained neural network, we want to apply it to unseen data. This could be done with the function 'mlpfwd'. This function takes the input data and calculates the network output based on the actual (now trained) network weights.

y = mlpfwd(net, input_1TEST);

The function needs as an input the network ('net') and the out of sample data (which here have the name 'input_1test').

The result of your calculation is the vector y. This vector contains the output values corresponding to each line of your input vector.

If you want to use a neural network to forecast stock prices (e.g. percentage changes), the vector y will contain them. To calculate the profit you would have made by applying the network to the out of sample time period, you have to compare the forecast values with the actual values.

If the actual values are represented by the vector 'output_1Test', you just have to look at the sign of your forecast. Say if the actual value would have been an upmove of +3.4% and your forecast would have been +2.5%, you would have gained the actual value of +3.4% (not the forecast +2.5%). To calculate your actual profit, you have to take the actual values and compare their sign (+ or -) with your forecast. The function **SIGN(...)** returns a vector where the input values are transformed in a way that a positive value is represented by a +1, a negative values by a –1 and a 0 by a 0. If you transform your forecast in that way and multiply it with the actual value, you get the actual gains.

y = sign(y) .* (output_1TEST);

Note: If you want to multiply two vectors element by element (and not in the way a vector multiplication would take place) you have to write '.*' instead of '*'!!!

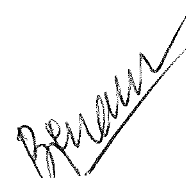
The vector y contains now the achieved gain corresponding to each input data line. To sum the elements up, use the command **SUM(...)**.

total_gain = sum(y)

Summary

To create a neural network, you have to follow the following steps:

1. Create data vectors (Training data [input and target] and out of sample data [input]).
2. Create network structure (using the command **MLP()**).
3. Train network with the training data (using the command **NETOPT()**).
4. Calculate the forecast corresponding to the out of sample input data (using **MPLFWD()**).

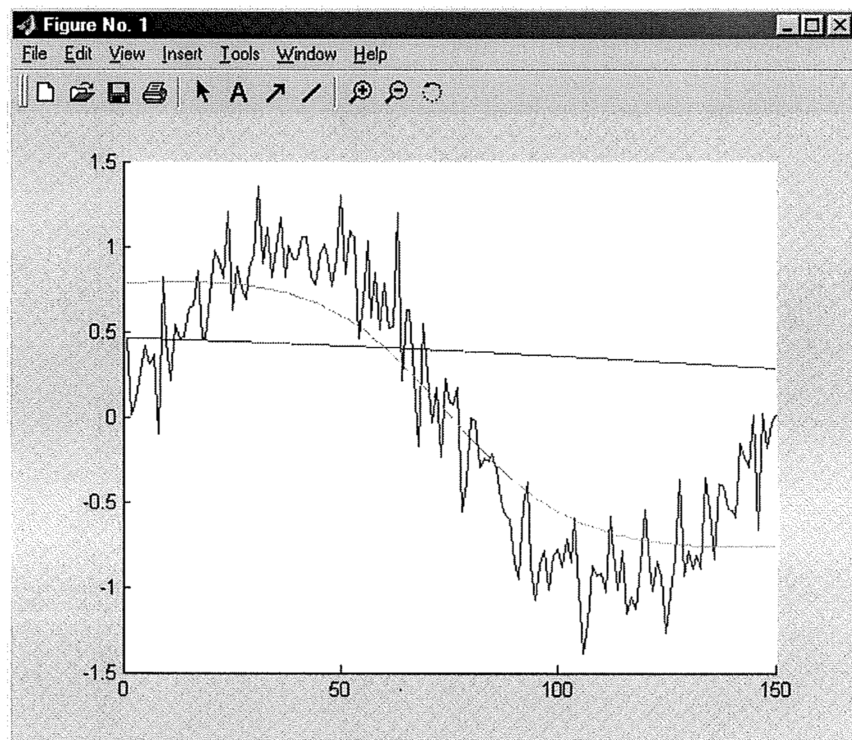


A test program

```
% Generate artificial data
%______________________

ndata = 150;                          % Number of data points.
noise = 0.2;                          % Standard deviation of noise distribution.

Input_Test = [0:1/(ndata - 1):1]';    % create vector with increasing equally spaced 'ndata'
                                      numbers between 0 and 1  [0;.....;1]
Output_Test = sin(2*pi*x) + noise*randn(ndata, 1);     % create 'ndata' numbers forming a sinus
                                                       curve with noise

% Set up parameters
%__________________

nin          = 1;          % Number of inputs.
nhidden      = 5;          % Number of hidden units.
nout         = 1;          % Number of outputs.

% Set up vector of options for the optimiser.
%_____________________________________

options      = zeros(1,18);
options(1)   = 1;          % This provides display of error values.
options(14)  = 500;        % Number of training cycles.
options(17)  = 0.001;      % momentum
options(18)  = 0.005;      % lrate

% Create and initialise network
%_________________________

net = mlp(nin, nhidden, nout, 'linear');

% Get results before training
%________________________

results_before = mlpfwd(net, input_Test);  % network output before training
                                           (based on the random weights)

%Train the net
%________________

[net, options] = netopt(net, options, input_Test, output_Test, 'graddesc');

%Get results after training
%______________________

results_after = mlpfwd(net, input_Test);  % network output after the training through
                                          forward calculation
```

```
% Plot results
%____________

clf;                                % clear current figure
hold on;                            % necessary to allow to plot many lines in the same figure

 plot(output_Test);                 % plot output_Test in blue (standard)
 plot(results_before,'r-');         % plot results before training in red
 plot(results_after,'g-');          % plot results after training in green

hold off;

% END
```

As a result, you get the following graph with 'blue' being the target values, 'red' being the best guess of the untrained network and 'green' being the results of the trained network.



A real data application

The program below has in principle the main structure as the example described above. However it has two important differences.

First, you are able to save or load network parameters. This is done by setting the variable: '**saving_flag**'. If you choose 1, the network data are going to be saved while you could load an already saved network by setting the variable to 2. Leaving it set to 0 will create a new network but will not save it. The file name consists of two parts: a number and a name (e.g. 3data_stock). The number has to be defined in the variable '**net_number**' while the name is defined through the variable '**name_files**'.

The other difference compared to the example previously described is the fact, that this model requires 6 time series to be defined by you. Two series (an input and an output series) each for the training data set, the test data set and the validation data set. The program will take the training data set to train the network weights in a way that you get the best results based on the test data set. Those weights are going to be fixed and will be applied in the next step to the out of sample data (validation data set). As you may have noticed there is the variable '**training_blocks**'. In the previous example you have defined the training iterations once. Here, the number training iterations are repeated as often as defined by the variable training_block. The network is therefore trained as often as (training_blocks) x (training_iterations). The reason for doing this is to define the point in time, when the network (being trained on the training data set) has reached its optimal result on the test data set. Therefore after each training_block on the training data set, the corresponding result (with the same actual weights) is calculated on the test data set. In the end, the program chooses automatically those weights that leads to the best results measured on the test data set. This procedure should help to avoid overfitting, that is the network should learn only the relevant structure and ignore the noise in the signal.

For your own application, you have therefore to define:

- the data (the training, test and validation data sets)
- the network structure
- the training procedure (number of training_iterations and training_blocks) and/or
- the option to save the new network or to load a previous one.

```
% DEMO PRG_01
%
%
%________________
clc;                             % clear screen
disp('Press any key to start.'); % display text
pause;                           % wait until a key is pressed
%________________
% Generate artificial data
```

```
input_Training_data     = input_4Test;
output_Training_data    = output_4Test;

input_Test_data         = input_TEST;
output_Test_data        = output_TEST;

input_Validation_data   = input_Validation;
output_Validation_data  = output_Validation;
```

```
%________________
% Set up network parameters
```

```
nin      = 10;     % Number of inputs
nhidden  = 5;      % Number of hidden units
nout     = 1;      % Number of outputs
```

% Set up vector of options for the optimiser

```
options         = zeros(1,18);
options(1)      = 1;              % This provides display of error values
options(14)     = 5;              % Number of training cycles
options(17)     = 0.001;          % momentum
options(18)     = 0.003;          % lrate
training_blocks = 20;
```

% Set up parameter for data saving

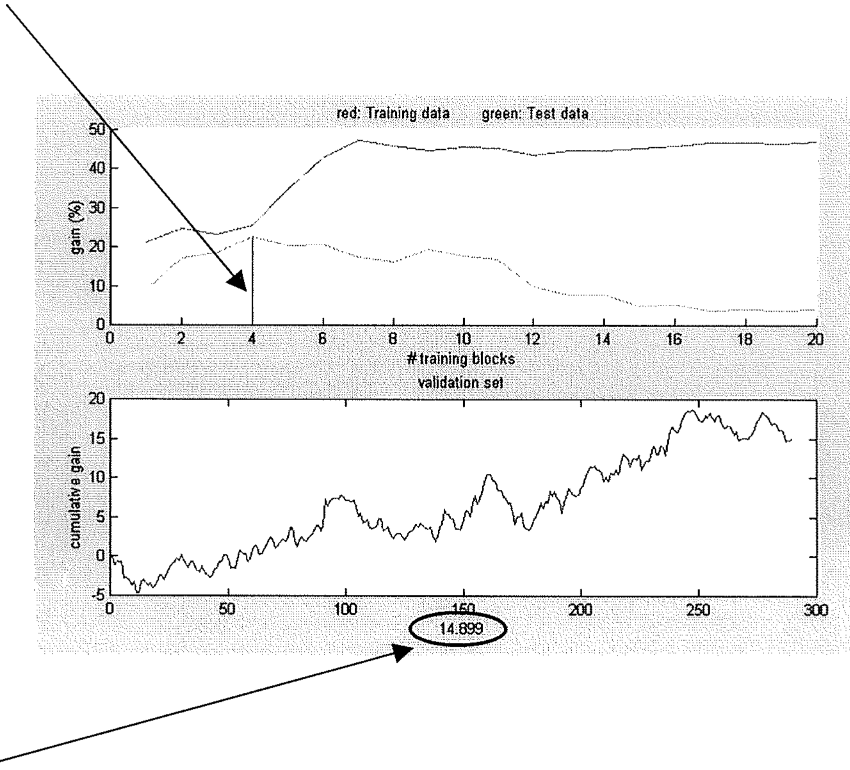
```
saving_flag     = 2;              % 1= save net data; 2= load net data; 0= none of it
name_files      = 'file_name_here';
net_number      = 1;
```

```
%================
% Create and initialize network or load saved one

outputfile = [int2str(net_number), name_files ];
if saving_flag==2; eval(['load ',outputfile]); end;    % load net_data in case options are set to do so!
if saving_flag~=2; net = mlp(nin, nhidden, nout, 'linear');end;      % create new net_data in case
                                                                      options are set to do so!
%__________________
% Train network

sum_Training        = [];
sum_Test            = [];
net_max             = [];
gain_max            = -exp(10);
loop_max            = 0;
clf;

for loop_01 = 1: training_blocks;

%Train network
if saving_flag ~= 2;
[net, options] = netopt(net, options, input_Training_data, output_Training_data, 'graddesc');
end;

results_training    = mlpfwd(net, input_Training_data);
results_test        = mlpfwd(net, input_Test_data);

gain_training       = sum(sign(results_training).*(output_Training_data));
gain_test           = sum(sign(results_test).*(output_Test_data));

sum_Training        = [sum_Training; gain_training];
sum_Test            = [sum_Test; gain_test];

 if gain_test     > gain_max;
   gain_max      = gain_test;
   loop_max      = loop_01;
```

```
   net_max        = net;
 end;

end;

%__________________
% save net_data if options are set to do so!

if saving_flag == 1; eval(['save ',outputfile,' net_max']);end;

%__________________

subplot(2,1,1);
hold on;
plot(sum_Training,'r-');
plot(sum_Test,'g-');
plot([loop_max loop_max], [0 gain_max]);
hold off;

title('red: Training data      green: Test data');
xlabel('# training blocks');
ylabel('gain');

%__________________

subplot(2,1,2);
net                         = net_max;
results_validation          = mlpfwd(net, input_Validation_data);
gain_validation             = sum(sign(results_validation).*(output_Validation_data));

plot(cumsum ((sign(results_validation).*(output_Validation_data))));

title('validation set');
xlabel(gain_validation);
ylabel('cumulative gain');

% END_____________
```

The red line shows that the potential gain on the training data set increases as the training process goes on. The green line shows the potential gain on the test data set when the same weights are applied to the test data rather than to the training data.

The blue line indicates that the potential gain on the test data peaks after the 4. training block. The corresponding weights are used to be applied to the out-of-sample data set (=validation data set).



The realised gain with the above mentioned weight set is in this example about 15% on the validation set.

End