A neural network toolbox for Octave
Developer's Guide
Version: 0.1.9

M. Schmid

October 9, 2008

Contents

Chapter 1

Introduction

This documentation isn't a well defined and structured docu for the neural network toolbox. It's more a *collection of my ideas and minds.*

1.1 Installed system

I'm developing and testing the actual version of the neural network toolbox on following program versions:

- Octave 2.9.5
- octave-forge-2006.01.28
- OctPlot svn version

.. and on another system

- Octave 2.9.12
- octave-forge packages ...
- OctPlot svn version

1.2 Version numbers of the neural network toolbox

The first number describes the major release. Version number V1.0 will be the first toolbox release which should have the same functions like the Matlab R14 SP3 neural network Toolbox.

The second number defines the finished functions. So to start, only the MLPs will realised and so this will be the number V0.1.0.

The third number defines the status of the actual development and function. V0.0.1 means no release with MLP, actually, everything under construction... ;-D.

Right now it's version V0.1.3 which means MLP works and currently the transfer function logsig is added.

1.3 Code convention

The main function of this toolbox will be programed with help of the book in [4]. So the variables will have the same names like in the book with one exception: Variables, only with one letter, will have two letters, e.g.

- $X \rightarrow Xx$
- $Q \rightarrow Qq$
- $a \rightarrow aa$

and so on ...

This is only to make it possible to search for variable names.

Chapter 2

Octave's available functions

2.1 Available functions

2.1.1 min_max

Checks for minimal and maximal values of an input matrix for **newff**.

Syntax:

$pr = min\_max(mInputs)$

Description:

mInputs must be a matrix with input training data sets. This means in the case, for a 9-2-1 MLP (this means 9 input-, 2 hidden- and 1 output-neuron) with 100 input training data sets, the matrix must be an 9x100 matrix. *pr* will then be a 9x2 matrix with minimal values in the first column and maximal values in the second column. If a row holds 2 zeros, a warning will appear (no information in this row!).

Important:

The equival function in MATLAB(TM) is called *minmax*. This is not possible because the functions *min* and *max* in Octave are programed in minmax.cc!

Chapter 3

Coding Guideline

Some genereal descriptions why a variable has a chosen name. This is valid for the complete toolbox... or so :-)
Here is only the description of variable names, which aren't visible to the user. Visible names are described in the User's Guide!
The variable identifiers are taken from [4]. One difference is purposeful added. If a variable has only one letter, a second small letter will be added to make it searchable. Who has ever tried to search a variable called "t"?

3.1 Variable identifier

Identifier	Description:
Aa	hold the network values after transfer function.
blf	**b**atch **l**earning **f**unction
btf	**b**atch **t**rainings **f**unction
Jj	Jacobi matrix
Nn	hold the network values before transfer function.
net	structure which holds the neural network informations
pf	**p**erformance **f**unction
Pp	input matrix; nInputs x nDataSets
Pr	input range, this is a Nx2 matrix, that's why the capitalized P
trf	**tr**ansfer **f**unction
Tt	target matrix, nTargets x nDataSets
ss	row vector with numbers of neurons in the layers, for each layer, one entry
vE	row vector with errors... size depends on network structure.
VV	Validation structure
xx	Weight vector in column format

3.1.1 Nn

Nn is a cell array and has one entry for each layer. In reality, this will have 2 or 3 layers.
In **Nn{1,1}** are the values for the first hidden layer. The size of this matrix depends on the number of neurons used for this layer.
In **Nn{2,1}** are the values for the second hidden layer or the output layer. The size of this matrix depends on the number of neurons used for this layer and so on ...
Nn{x,1} where **x** can be ∞.

3.1.2 Aa

Aa is a cell array and has one entry for each layer.
In **Aa{1,1}** are the values for the first hidden layer. The size of this matrix depends on the number of neurons used for this layer.
In **Aa{2,1}** are the values for the second hidden layer or the output layer. The size of this matrix depends on the number of neurons used for this layer.
See **Nn** for a more detailed description.

3.1.3 vE

vE is also a cell array which holds in the last (second) element the error vector. It's not completly clear, why in the last (second) element.
The number of rows depends on the number of output neurons. For one output neuron, **vE** holds only one row, for 2 output neurons, this holds of course 2 rows, and so on.

3.1.4 Jj

This is the short term for the Jacobi matrix.

Chapter 4

Algorithm

Here are some general thoughts about calculating parts are used in algorithm.

4.1 Levenberg Marquardt

This algorithm will be programed with [4].

4.1.1 Sensitivity Matrix

How does this looks like?

1. for a 1-1-1 MLP
2. for a 2-1-1 MLP
3. for a 1-2-1 MLP

1-1-1 MLP

In this case, the MLP holds one input neuron, one hidden neuron and one output neuron. The number of weights needed for this MLP is 4 (2 weights, 2 biases).

It needs two sensitivity matrices because the two layers. Each sensitivity matrix will hold 1 element. This is taken from [4], example P12.5 page 12-44. Attention, in this example are two data sets used, this is the reason for the 4 elements...!

2-1-1 MLP

In this case, the MLP holds two input neurons, one hidden neuron and one output neuron. The number of weights needed for this MLP is 5 (3 weights, 2 biases).

It needs also two sensitivity matrices because the two layers. Actually, the input is not only a scalar, it is a vector with 2 elements. Even though, again after [4]. I think the sensitivity matrices will hold only 1 element. So the number of elements will bi proportional to the number of hidden neurons and the number of output neurons.

1-2-1 MLP

In this case, the MLP holds one input neuron, two hidden neurons and one output neuron. The number of weights needed for this MLP is 7 (4 weights, 3 biases).

It needs also two sensitivity matrices because the two layers. Actually, the input is again only a scalar. Now calculating n_1^1 will result in a row vector with 2 elements. n_1^2 will hold only one element and so we have 3 elements in the sensitivity matrix.

We can say, the number of hidden neurons is responsible for the dimension of the sensitivity matrices. For example, a 4-3-1 MLP with 100 data sets will generate following sensitivity matrix for the first layer: $\tilde{\mathbf{S}}^1 = [\tilde{\mathbf{S}}_1^1|\tilde{\mathbf{S}}_2^1|...|\tilde{\mathbf{S}}_{100}^1]$
$\tilde{\mathbf{S}}_1^1$ will hold 3 elements $\tilde{\mathbf{S}}_1^1 = [\tilde{\mathbf{S}}_{1,1}^1\ \tilde{\mathbf{S}}_{2,1}^1\ \tilde{\mathbf{S}}_{3,1}^1]^T$; $\tilde{\mathbf{S}}_2^1 = [\tilde{\mathbf{S}}_{1,2}^1\ \tilde{\mathbf{S}}_{2,2}^1\ \tilde{\mathbf{S}}_{3,2}^1]^T$ and so on. So matrix will have a size of 3x100 for $\tilde{\mathbf{S}}_1^1$ and a size of 1x100 for $\tilde{\mathbf{S}}_2^1$.

By the way, the jacobian matrix will be a 100x20 matrix ..

Chapter 5

Test

5.1 isposint

```
%!shared
%! disp("testing isposint")
%!assert(isposint(1)) # this should pass
%!assert(isposint(0.5),0) # should return zero
%!assert(isposint(-1),0) # should return zero
%!assert(isposint(-1.5),0) # should return zero
%!assert(isposint(0),0) # should return zero
%!fail("isposint([0 0])","Input argument must not be a vector, only scalars are allowed!")
%!fail("isposint('testString')","Input argument must not be a vector, only scalars are allowed!")
```

5.2 min_max

```
%!shared
%! disp("testing min_max")
%!test fail("min_max(1)","Argument must be a matrix.")
%!test fail("min_max('testString')","Argument must be a matrix.")
%!test fail("min_max(cellA{1}=1)","Argument must be a matrix.")
%!test fail("min_max([1+1i, 2+2i])","Argument must be a matrix.")
%!test fail("min_max([1+1i, 2+2i; 3+1i, 4+2i])","Argument has illegal type.")
```

5.3 newff

```
%!shared
%! disp("testing newff")
%!test
%! Pr = [1;2];
%! fail("newff(Pr,[1 1],{'tansig','purelin'},'trainlm','unused','mse')","Input ranges must be a t
%!test
%! Pr = [1 2 ; 4  6];
%! assert(__checknetstruct(newff(Pr,[1 1],{'tansig','purelin'},'trainlm','unused','mse')))
%!test
%! Pr = [1 2 3; 4 5 6];
%! fail("newff(Pr,[1 1],{'tansig','purelin'},'trainlm','unused','mse')","Input ranges must be a t
%!test
%! Pr = [5 3; 4 5];
```

```
%! fail("newff(Pr,[1 1],{'tansig','purelin'},'trainlm','unused','mse')",\
%!  "Input ranges has values in the second column larger as in the same row of the first column."
%!test
%! Pr = [1 2 ; 4 6];
%! fail("newff(Pr,[1 1; 2 3],{'tansig','purelin'},'trainlm','unused','mse')",\
%!  "Layer sizes is not a row vector.")
%!test
%! Pr = [1 2 ; 4 6];
%! assert(__checknetstruct(newff(Pr,[ 2 3],{'tansig','purelin'},'trainlm','unused','mse')))
%!test
%! Pr = [1 2 ; 4 6];
%! fail("newff(Pr,[1],{'tansig','purelin'},'trainlm','unused','mse')",\
%!  "There must be at least one hidden layer and one output layer!")
%!test
%! Pr = [1 2 ; 4 6];
%! fail("newff(Pr,[-1 1],{'tansig','purelin'},'trainlm','unused','mse')",\
%!  "Layer sizes is not a row vector of positive integers.")
```

5.4 prestd

```
%!shared Pp, Tt, pn
%!  Pp = [1 2 3 4; -1 3 2 -1];
%!  Tt = [3 4 5 6];
%!  [pn,meanp,stdp] = prestd(Pp);
%!assert(pn,[-1.16190 -0.38730 0.38730 1.16190; -0.84887 1.09141 0.60634 -0.84887],0.00001);
```

5.5 purelin

```
%!assert(purelin(2),2);
%!assert(purelin(-2),-2);
%!assert(purelin(0),0);
%!error  # this test must throw an error!
%! assert(purelin(2),1);
```

5.6 subset

```
%!shared matrix, nTargets, mTrain, mTest, mVali
%! disp("testing subset")
%! matrix = [1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 18 20; \
%!  0 2 4 1 3 5 3 4 1 -1 -2 -9 -1 10 12 20 11 11 11 11; \
%! -2 2 2 2 2 0 0 0 0  0 10 12 13 12 13 44 33 32 98 11; \
%!  0 0 0 0 1 1 1 1 0  0  1  1  1  0  0  1  1  1  0  0; \
%!           4 4 4 4 4 4 4 4 4  4  4  4  4  4  4  4  4  4; \
%!           1 2 3 4 5 6 7 8 9 10 11 12 13 33 44 55 66 77 88 99];
%! nTargets = 1; # the last row is equivalent to the target values.
%! [mTrain, mTest, mVali] = subset(matrix,nTargets);  ############################
%!assert(size(mTrain,2)==10);# 50% of 20
%!assert(size(mTest,2)==6);# 1/3 of 20 = 6 (floor)
%!assert(size(mVali,2)==4);# 1/6 of 20 = 4 (floor)
%! # It's not possible to test the column order with this call!
%! # randomizing is used! But all max and min values should be
%! # in the training set
```

```
%!assert(max(mTrain(1,:))==max(matrix(1,:)));
%!assert(min(mTrain(1,:))==min(matrix(1,:)));
%!assert(max(mTrain(2,:))==max(matrix(2,:)));
%!assert(min(mTrain(2,:))==min(matrix(2,:)));
%!assert(max(mTrain(3,:))==max(matrix(3,:)));
%!assert(min(mTrain(3,:))==min(matrix(3,:)));
%!assert(max(mTrain(4,:))==max(matrix(4,:)));
%!assert(min(mTrain(4,:))==min(matrix(4,:)));
%!
%!
%! [mTrain, mTest, mVali] = subset(matrix,nTargets,0);  ############################
%!assert(size(mTrain,2)==10);# 50% of 20
%!assert(size(mTest,2)==6);# 1/3 of 20 = 6 (floor)
%!assert(size(mVali,2)==4);# 1/6 of 20 = 4 (floor)
%!assert(mTrain==matrix(:,1:10));
%!assert(mTest==matrix(:,11:16));
%!assert(mVali==matrix(:,17:20));
%!
%!
%! [mTrain, mTest, mVali] = subset(matrix,nTargets,2);  ############################
%!assert(size(mTrain,2)==10);# 50% of 20
%!assert(size(mTest,2)==6);# 1/3 of 20 = 6 (floor)
%!assert(size(mVali,2)==4);# 1/6 of 20 = 4 (floor)
%!assert(max(mTrain(1,:))==max(matrix(1,:)));
%!assert(min(mTrain(1,:))==min(matrix(1,:)));
%!assert(max(mTrain(2,:))==max(matrix(2,:)));
%!assert(min(mTrain(2,:))==min(matrix(2,:)));
%!assert(max(mTrain(3,:))==max(matrix(3,:)));
%!assert(min(mTrain(3,:))==min(matrix(3,:)));
%!assert(max(mTrain(4,:))==max(matrix(4,:)));
%!assert(min(mTrain(4,:))==min(matrix(4,:)));
%!
%!
%! ## next test ... optimize twice
%! matrix = [1 2 3 4 5 6 7 20 8 10 11 12 13 14 15 16 17 18 18 9; \
%!  0 2 4 1 3 5 3 4 1 -1 -2 -9 -1 10 12 20 11 11 11 11; \
%! -2 2 2 2 2 0 0 0 0  0 10 12 13 12 13 44 33 32 98 11; \
%!  0 0 0 0 1 1 1 1 0  0  1  1  1  0  0  1  1  1  0  0; \
%!           4 4 4 4 4 4 4 4 4  4  4  4  4  4  4  4  4  4  4  4; \
%!           1 2 3 4 5 6 7 8 9 10 11 12 13 33 44 55 66 77 88 99];
%! [mTrain, mTest, mVali] = subset(matrix,nTargets,2);  ############################
%!assert(max(mTrain(1,:))==max(matrix(1,:)));
%!assert(min(mTrain(1,:))==min(matrix(1,:)));
%!assert(max(mTrain(2,:))==max(matrix(2,:)));
%!assert(min(mTrain(2,:))==min(matrix(2,:)));
%!assert(max(mTrain(3,:))==max(matrix(3,:)));
%!assert(min(mTrain(3,:))==min(matrix(3,:)));
%!assert(max(mTrain(4,:))==max(matrix(4,:)));
%!assert(min(mTrain(4,:))==min(matrix(4,:)));
```

5.7 __analyzerows

```
%!shared b, retmat
```

```
%! disp("testing __analyzerows")
%! b = [1 0 0 1; 1 0 0 0; 1 2 0 1];
%! retmat = __analyzerows(b);
%!assert(retmat(1,1)==1);#%!assert(retmat(1,1)==1);
%!assert(retmat(2,1)==1);
%!assert(retmat(3,1)==0);
%! b = [1 0 0 2; 1 0 0 0; 1 1 1 1];
%! retmat = __analyzerows(b);
%!assert(retmat(1,2)==0);
%!assert(retmat(2,2)==0);
%!assert(retmat(3,2)==1);
%! b = [1 0 0 2; 1 0 0 0; 1 1 1 1];
%! retmat = __analyzerows(b);
%!assert(retmat(1,3)==2);
%!assert(retmat(2,3)==0);
%!assert(retmat(3,3)==0);
%! retmat = __analyzerows(b);
%!assert(retmat(1,4)==1);
%!assert(retmat(2,4)==0);
%!assert(retmat(3,4)==0);
```

5.8 __copycoltopos1

```
%!shared a, retmat
%! disp("testing __copycoltopos1")
%! a = [0 1 2 3 4; 5 6 7 8 9];
%! retmat = __copycoltopos1(a,3);
%!assert(retmat(1,1)==2);
%!assert(retmat(2,1)==7);
%! retmat = __copycoltopos1(a,5);
%!assert(retmat(1,1)==4);
%!assert(retmat(2,1)==9);
```

5.9 __optimizedatasets

```
%!shared retmatrix, matrix
%! disp("testing __optimizedatasets")
%! matrix = [1 2 3 2 1 2 3 0 5 4 3 2 2 2 2 2 2; \
%!  0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0; \
%! -1 3 2 4 9 1 1 1 1 1 9 1 1 1 9 9 0; \
%!  2 3 2 3 2 2 2 2 3 3 3 3 1 1 1 1 1];
%! ## The last row is equal to the neural network targets
%! retmatrix = __optimizedatasets(matrix,9,1);
%! ## the above statement can't be tested with assert!
%! ## it contains random values! So pass a "success" message
%!assert(1==1);
%! matrix = [1 2 3 2 1 2 3 0 5 4 3 2 2 2 2 2 2; \
%!  0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0; \
%! -1 3 2 4 9 1 1 1 1 1 9 1 1 1 9 9 0; \
%!  2 3 2 3 2 2 2 2 3 3 3 3 1 1 1 1 1];
%! ## The last row is equal to the neural network targets
%! retmatrix = __optimizedatasets(matrix,9,1,0);
```

```
%!assert(retmatrix(1,1)==5);
%!assert(retmatrix(2,1)==0);
%!assert(retmatrix(3,1)==1);
%!assert(retmatrix(4,1)==3);
```

5.10 __randomisecols

```
%!# no test possible, contains randperm which is using
%!# some randome functions
```

5.11 __rerangecolumns

```
%!shared matrix,analyzeMatrix,nTrainSets, returnmatrix
%! disp("testing __rerangecolumns")
%! matrix = [0 1 0 0 0 0 1 0 1 1;  \
%!  4 4 4 4 4 4 4 4 4 4;  \
%!        -1.1 -1.1 2 3 4 3.2 1 8 9 10; \
%!           0 1.1 3 4 5 2 10 10 2 3; \
%!          -1 1 1 1 1 2 3 4 1 5];
%! analyzeMatrix = [1 0 0 0; 0 1 0 0; 0 0 2 1; 0 0 1 2; 0 0 1 1];
%! nTrainSets = 8;
%! returnmatrix = __rerangecolumns(matrix,analyzeMatrix,nTrainSets);
%!assert(returnmatrix(1,1)==1);
%!assert(returnmatrix(2,1)==4);
%!assert(returnmatrix(3,1)==1);
%!assert(returnmatrix(4,1)==10);
%!assert(returnmatrix(5,1)==3);
%! matrix = [0 1 0 0 0 0 1 0 1 1;  \
%!  4 4 4 4 4 4 4 4 4 4;  \
%!          -1.1 -1.1 2 3 4 3.2 1 8 9 10;  \
%!           0 1.1 3 4 5 2 10 10 2 3;  \
%!          -1 1 1 1 1 2 3 4 1 5;      \
%!  0 1 2 1 2 1 2 3 4 5;];  # the last row is euqal to the nnet targets
%! analyzeMatrix = [1 0 0 0; 0 1 0 0; 0 0 2 1; 0 0 1 2; 0 0 1 1];
%! nTrainSets = 8;
%! returnmatrix = __rerangecolumns(matrix,analyzeMatrix,nTrainSets);
%!assert(returnmatrix(1,1)==1);
%!assert(returnmatrix(2,1)==4);
%!assert(returnmatrix(3,1)==1);
%!assert(returnmatrix(4,1)==10);
%!assert(returnmatrix(5,1)==3);
%!assert(returnmatrix(6,1)==2);
```

Chapter 6

analyzing matlab functions

6.1 analyzing newff

First, *newff* will be analyzed for a X-X-X mlp. This means, maximum 3 layers, including the input layer. Or in words, one input- one hidden- and one output-layer. The number of neurons are choosable.

Following command will be used, to create a new feed-forward neural network:
MLPnet = newff(mMinMaxElements,[nHiddenNeurons nOutputNeurons],...
{'tansig','purelin'},'trainlm','learngdm','mse');

newff is the matlab command, mMinMaxElements is a $Rx2$-Matrix with minimum and maximum values of the inputs. R is equal to the number of input neurons. [nHiddenNeurons nOutputNeurons] are the scalar values, to define the number of neurons in the hidden and output layer. One value, for each layer. {'tansig','purelin'} are the transfer functions, for each layer. This means, 'tansig' for the hidden layer and 'purelin' for the output layer. 'trainlm' is the training algorithm, in this case, Levenberg-Marquardt. 'learngdm' is the learn algorithm and 'mse' is the performance function, **m**ean-**s**quare-**e**rror.
MLPnet will be a structure with following content:

```
Neural Network object:

    architecture:

         numInputs: 1
         numLayers: 2
       biasConnect: [1; 1]
      inputConnect: [1; 0]
      layerConnect: [0 0; 1 0]
     outputConnect: [0 1]
     targetConnect: [0 1]

        numOutputs: 1  (read-only)
        numTargets: 1  (read-only)
    numInputDelays: 0  (read-only)
    numLayerDelays: 0  (read-only)

    subobject structures:

            inputs: {1x1 cell} of inputs
            layers: {2x1 cell} of layers
```

```
          outputs: {1x2 cell} containing 1 output
          targets: {1x2 cell} containing 1 target
           biases: {2x1 cell} containing 2 biases
    inputWeights: {2x1 cell} containing 1 input weight
    layerWeights: {2x2 cell} containing 1 layer weight

functions:

        adaptFcn: 'trains'
         initFcn: 'initlay'
      performFcn: 'mse'
        trainFcn: 'trainlm'

parameters:

      adaptParam: .passes
       initParam: (none)
    performParam: (none)
      trainParam: .epochs, .goal, .max_fail, .mem_reduc,
                  .min_grad, .mu, .mu_dec, .mu_inc,
                  .mu_max, .show, .time

weight and bias values:

              IW: {2x1 cell} containing 1 input weight matrix
              LW: {2x2 cell} containing 1 layer weight matrix
               b: {2x1 cell} containing 2 bias vectors

other:

        userdata: (user stuff)
```

numInputs: 1: one input layer
numLayers: 2: one hidden and one output layer
biasConnect: [1; 1]: unknown till now!!
inputConnect: [1; 0]: unknown till now!!
layerConnect: [0 0; 1 0]: unknown till now!!
outputConnect: [0 1]: unknown till now!!
targetConnect: [0 1]: unknown till now!!
numOutputs: 1 (read-only): unknown till now!!
numTargets: 1 (read-only): unknown till now!!
numInputDelays: 0 (read-only): unknown till now!!
numLayerDelays: 0 (read-only): unknown till now!!
inputs: 1x1 cell of inputs: input layer definition
Because we have defined only one input layer, you can see the detailed definition with following command in the matlab prompt:

```
MLPnet.inputs{1}

ans =

       range: [26x2 double]
        size: 26
    userdata: [1x1 struct]
```

range are the min. and max. values of the inputs. size is the number of input neurons and userdata are user specified inputs...!
layers: 2x1 cell of layers: hidden and output layer definition
The dimension of $2x1cell$ is because we have one hidden and one output layer. So too see the details of the hidden layer definitions, we have to enter:

```
K>> MLPnet.layers{1}

ans =

     dimensions: 2
    distanceFcn: ''
      distances: []
        initFcn: 'initnw'
    netInputFcn: 'netsum'
      positions: [0 1]
           size: 2
    topologyFcn: 'hextop'
    transferFcn: 'tansig'
       userdata: [1x1 struct]
```

and for the output layer:

```
K>> MLPnet.layers{2}

ans =

     dimensions: 1
    distanceFcn: ''
      distances: []
        initFcn: 'initnw'
    netInputFcn: 'netsum'
      positions: 0
           size: 1
    topologyFcn: 'hextop'
    transferFcn: 'purelin'
       userdata: [1x1 struct]
```

outputs: 1x2 cell containing 1 output: output layer definitions

```
K>> MLPnet.outputs

ans =

     []    [1x1 struct]
```

How knows, why this is a $1x2cell$? The next command will also show the detailed definition! Of course, realy simple.

```
K>> MLPnet.outputs{2}

ans =

        size: 1
    userdata: [1x1 struct]
```

targets: 1x2 cell containing 1 target: unknow till now

biases: 2x1 cell containing 2 biases: detailed definitions, for the biases

```
K>> MLPnet.biases

ans =

    [1x1 struct]
    [1x1 struct]

K>> MLPnet.biases{1}

ans =

       initFcn: ''
         learn: 1
      learnFcn: 'learngdm'
    learnParam: [1x1 struct]
          size: 2
      userdata: [1x1 struct]

K>> MLPnet.biases{2}

ans =

       initFcn: ''
         learn: 1
      learnFcn: 'learngdm'
    learnParam: [1x1 struct]
          size: 1
      userdata: [1x1 struct]
```

inputWeights: 2x1 cell containing 1 input weight layerWeights: 2x2 cell containing 1 layer weight

weight and bias values:

IW:

```
K>> MLPnet.IW

ans =

    [2x26 double]
                []
```

LW:

```
K>> MLPnet.LW

ans =

                []      []
    [1x2 double]      []
```

b:

```
K>> MLPnet.b

ans =

    [2x1 double]
    [   -0.3908]
```

net.trainParam: Output for the Levenberg-Marquardt train algorithm.

```
K>> MLPnet.trainParam

ans =

       epochs: 100
         goal: 0
     max_fail: 5
    mem_reduc: 1
     min_grad: 1.0000e-010
           mu: 0.0010
       mu_dec: 0.1000
       mu_inc: 10
       mu_max: 1.0000e+010
         show: 25
         time: Inf
```

6.2 analyzing newp

Following command will be used, to create a new neural perceptron:
net = newfp(mMinMaxElements,nNeurons);

newp is the matlab command, mMinMaxElements is a $Rx2$-Matrix with minimum and maximum values of the inputs. R is equal to the number of input neurons.

net = newp([0 1; -2 2],1)

```
net =

    Neural Network object:

    architecture:

         numInputs: 1
         numLayers: 1
       biasConnect: [1]
      inputConnect: [1]
      layerConnect: [0]
     outputConnect: [1]
     targetConnect: [1]

        numOutputs: 1  (read-only)
        numTargets: 1  (read-only)
    numInputDelays: 0  (read-only)
```

```
    numLayerDelays: 0  (read-only)

    subobject structures:

            inputs: {1x1 cell} of inputs
            layers: {1x1 cell} of layers
           outputs: {1x1 cell} containing 1 output
           targets: {1x1 cell} containing 1 target
            biases: {1x1 cell} containing 1 bias
      inputWeights: {1x1 cell} containing 1 input weight
      layerWeights: {1x1 cell} containing no layer weights

    functions:

          adaptFcn: 'trains'
           initFcn: 'initlay'
        performFcn: 'mae'
          trainFcn: 'trainc'

    parameters:

        adaptParam: .passes
         initParam: (none)
      performParam: (none)
        trainParam: .epochs, .goal, .show, .time

    weight and bias values:

                IW: {1x1 cell} containing 1 input weight matrix
                LW: {1x1 cell} containing no layer weight matrices
                 b: {1x1 cell} containing 1 bias vector

    other:

          userdata: (user stuff)
```

numInputs: 1: one input layer
numLayers: 1: one output layer
biasConnect: [1]: unknown till now!!
inputConnect: [1]: unknown till now!!
layerConnect: [0]: unknown till now!!
outputConnect: [1]: unknown till now!!
targetConnect: [1]: unknown till now!!
numOutputs: 1 (read-only): unknown till now!!
numTargets: 1 (read-only): unknown till now!!
numInputDelays: 0 (read-only): unknown till now!!
numLayerDelays: 0 (read-only): unknown till now!!
inputs: 1x1 cell of inputs: input layer definition
Because we have defined only one input layer, you can see the detailed definition with following command in the matlab prompt:

```
>> net.inputs{1}

ans =
```

```
       range: [2x2 double]
        size: 2
    userdata: [1x1 struct]

net.inputs{1}.range

ans =

     0     1
    -2     2

net.inputs{1}.size

ans =

     2

net.inputs{1}.userdata

ans =

    note: 'Put your custom input information here.'
```

range are the min. and max. values of the inputs. size is the number of input neurons and userdata are user specified inputs...!
layers: 1x1 cell of layers: actually no idea what's inside this cell!!!
To see the details of this layer definition, we have to enter:

```
net.layers{1}

ans =

     dimensions: 1
    distanceFcn: ''
      distances: []
        initFcn: 'initwb'
    netInputFcn: 'netsum'
      positions: 0
           size: 1
    topologyFcn: 'hextop'
    transferFcn: 'hardlim'
       userdata: [1x1 struct]
```

and for the output layer:

```
net.outputs{1}

ans =

        size: 1
    userdata: [1x1 struct]

 net.outputs{1}.userdata

ans =
```

```
    note: 'Put your custom output information here.'
net.targets

ans =

    [1x1 struct]
net.targets{1}

ans =

        size: 1
    userdata: [1x1 struct]
net.targets{1}.userdata

ans =

    note: 'Put your custom output information here.'
net.biases{1}

ans =

       initFcn: 'initzero'
         learn: 1
      learnFcn: 'learnp'
    learnParam: []
          size: 1
      userdata: [1x1 struct]
net.biases{1}.userdata

ans =

    note: 'Put your custom bias information here.'
net.inputWeights

ans =

    [1x1 struct]
net.inputWeights{1}

ans =

        delays: 0
       initFcn: 'initzero'
         learn: 1
      learnFcn: 'learnp'
    learnParam: []
          size: [1 2]
      userdata: [1x1 struct]
     weightFcn: 'dotprod'
```

```
net.layerWeights

ans =

    {[]}

net.LW

ans =

    {[]}

net.IW

ans =

    [1x2 double]

net.IW{1}

ans =

     0     0

net.b

ans =

    [0]
```

Appendix A

Examples

A.1 Example 1

This MLP is designed with 2-2-1. This is not a complete example but it will help to understand the dimensions of all matrices and vectores are used inside the Levenberg-Marquardt algorithm.

A.1.1 Data matrices

The input matrix will be defined like in equation (A.1) and the output matrix like in equation (A.2).

$$mInput = \begin{bmatrix} 1 & 2 & 3 & 1 \\ 1 & 1 & 1 & 2 \\ 1 & 2 & 1 & 2 \end{bmatrix} \tag{A.1}$$

$$mOutput = \begin{bmatrix} 1 & 1.5 & 2 & 3 \end{bmatrix} \tag{A.2}$$

A.1.2 Weight matrices

The first layer matrix will hold 2x3 weights. The second layer matrix will hold 1x2 weights. The first bias holds 3x1 weights and the second holds only a scalar element.

A.1.3 Sensitivity Matrices

This part is right now not so clear in my mind. What is the dimension of these two matrices? The first layer sensitivity matrix should be about 2x71. Number of hidden neurons in the rows and number of train data sets in the columns.

In the actual version, the dimension is about 71x71 .. so it seems to have a mistake inside the algorithm :-(

Bibliography

[1] John W. Eaton
GNU Octave Manual, Edition 3, PDF-Version, February 1997

[2] The MathWorks, Inc.
MATLAB Online-Help

[3] Steven W. Smith
The Scientist and Engineer's Guide to Digital Signal Processing ISBN 0-9660176-3-3, California Technical Publishing, 1997

[4] Martin T. Hagan, Howard B. Demuth, Mark Beale
Neural Network Design, ISBN 0971732108, PWS Publishing Company, USA, Boston, 1996